

高等计算机体系结构

第十四讲: 互连和片上多核

栾钟治
北京航空航天大学 计算机学院 中德联合软件研究所

1

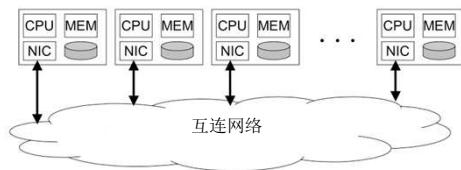
互连网络基础

2

2

哪里需要互连网络?

- 有组件需要互连
- 很多例子
 - 处理器和处理器
 - 处理器和内存(bank)
 - 处理器和cache (bank)
 - Cache和cache
 - I/O 设备



3

3

为什么这个很重要?

- 影响系统的可扩展性
 - 可以构建一个多大的系统?
 - 增加更多的处理器有多容易?
- 影响性能和能效
 - 处理器、cache和内存之间通信有多快?
 - 访存延迟有多大?
 - 通信消耗多少能量?

4

4

互连网络基本概念

- 拓扑
 - 指明组件连接的方式
 - 影响路由、可靠性、吞吐量、延迟
- 路由 (算法)
 - 消息如何从源到目的
 - 静态还是自适应
- 缓冲和流量控制
 - 在互连网络中存储些什么?
 - 完整的包, 部分包, 其它?
 - 如何在过载时进行调节?
 - 与路由策略紧耦合

5

5

拓扑

- 总线 (最简单)
- 点到点互连 (理想方式、成本最高)
- 交叉开关 (Crossbar)
- 环
- 树
- Omega
- 超立方
- 网状网 (Mesh)
- Torus
- 蝶形
- ...

6

6

评价互连网络的指标

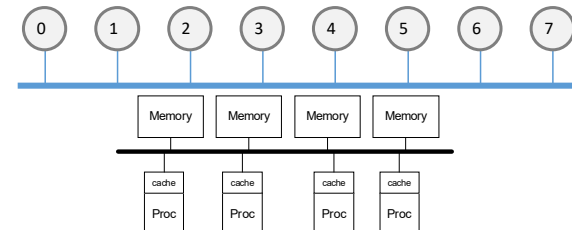
- 成本
- 延迟 (按跳, 单位纳秒)
- 竞争度
- 其它需要考虑的指标
 - 能耗
 - 带宽
 - 系统整体性能

7

7

总线

- + 简单
- + 节点数量少时成本效率高
- + 很容易实现一致性 (监听和顺序)
- 节点数量大时没有可扩展性 (受限的带宽, 电负载 → 频率降低)
- 高竞争度 → 快饱和



8

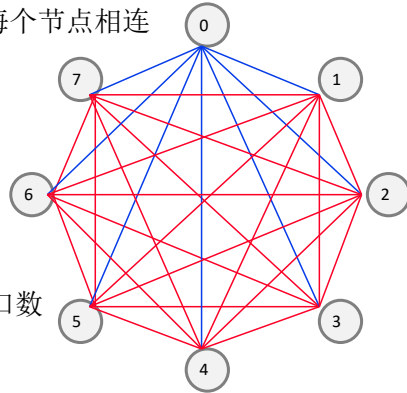
8

点到点

每个节点都与其它的所有节点相连

- + 竞争度最低
- + 潜在的最低延迟
- + 理想(如果不差钱)

- 成本最高
- $O(N)$ 连接/每节点端口数
- $O(N^2)$ 链路
- 没有扩展性
- 在芯片上如何布局?



9

9

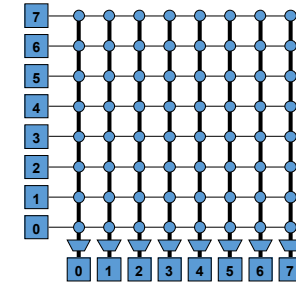
交叉开关(Crossbar)

- 每个节点可连接到任何其它节点(无阻塞), 不同的是任一节点可随时使用连接
- 允许同时向无冲突的目的节点发送
- 适用于节点数目较小的情况

- + 低延迟、高吞吐
- 昂贵
- 无可扩展性 $\rightarrow O(N^2)$ 成本
- 随着N的增加仲裁越来越困难

在核到cache到bank中使用

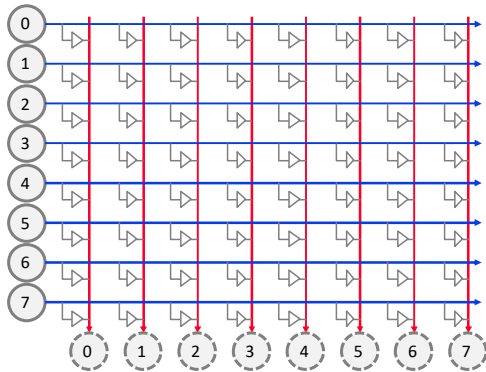
- IBM POWER5
- Sun Niagara I/II



10

10

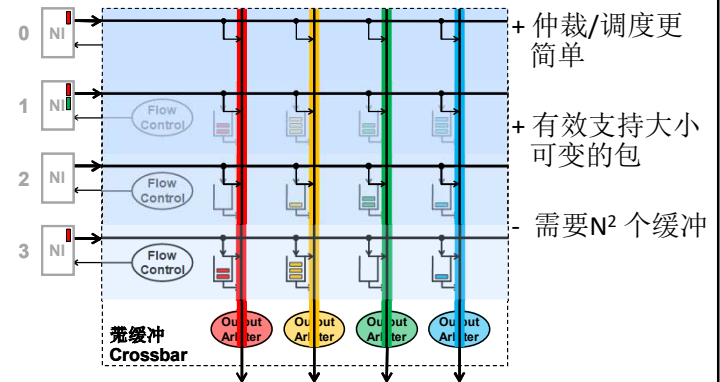
另一种 Crossbar 设计



11

11

带缓冲的 Crossbar



12

12

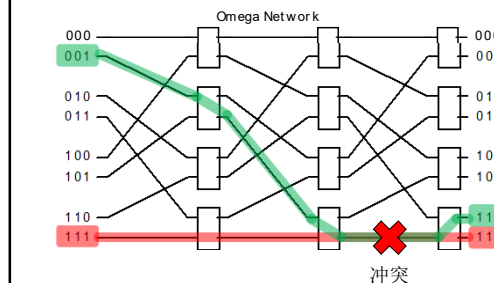
能比Crossbar成本更低吗?

- 仍能拥有低竞争度?
- 思路: 多阶段网络

13

多级对数网络

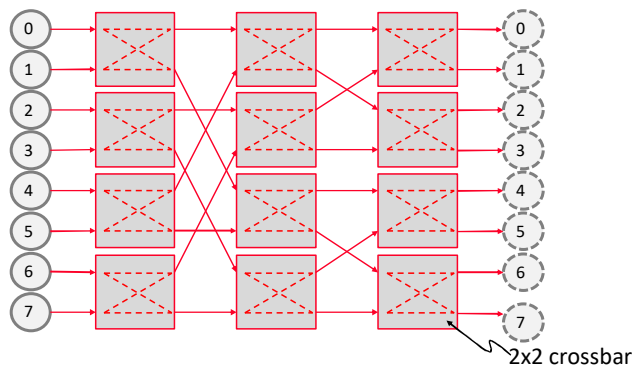
- 思路: 在终端/节点之间通过多层交换实现间接组网
- 成本: $O(N \log N)$, 延迟: $O(\log N)$
- 很多变种(Omega, 蝴蝶, Benes, Banyan, ...)
- Omega 网络:



Gottlieb et al. "The NYU Ultracomputer-designing a MIMD, shared-memory parallel machine," ISCA 1982.

14

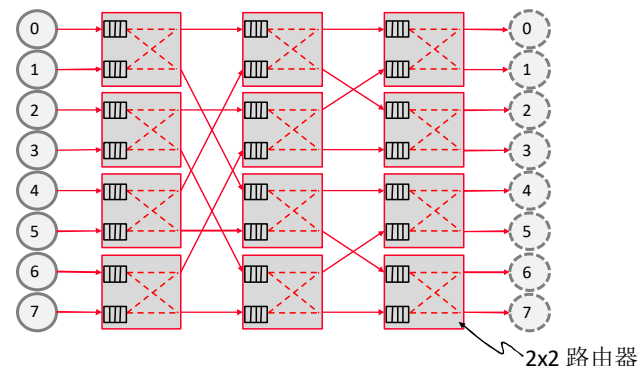
多级电路交换



- 对并发更严格
- 但是相对于crossbar的成本来说具有更好的可扩展性

15

多级包交换



- 包在路由器之间逐“跳”传递, 等待下一跳交换机和缓冲的可用性

16

交换 vs. 拓扑

- 电路/包交换的选择不依赖于拓扑
- 消息如何传送至目的地依靠高层协议
- 当然, 某些拓扑确实可能更适用于电路或者包交换

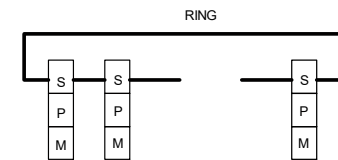
17

17

环

- + 便宜: $O(N)$ 成本
- 高延迟: $O(N)$
- 不易扩展
- 对分带宽是常数

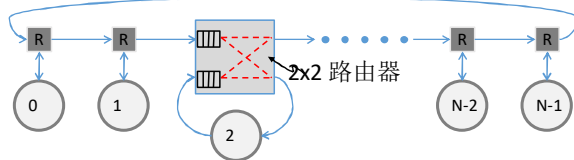
在Intel Haswell, Intel Larrabee, IBM Cell等很多现代商业化系统中使用



18

18

单向环



- 拓扑及实现简单
 - 如果 N 和带宽及延迟要求都相对较低, 性能比较合理
 - $O(N)$ 成本
 - $N/2$ 平均跳数; 延迟依赖于利用率

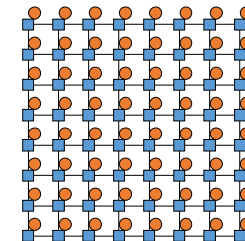
19

19

网状网(Mesh)

- $O(N)$ 成本
- 平均延迟: $O(\sqrt{N})$
- 容易在芯片上布局: 规则并且等长的连接
- 路径多样性: 从一点到另一点有很多条路

- Tiler 100核芯片中使用
- 是许多片上网络的原型

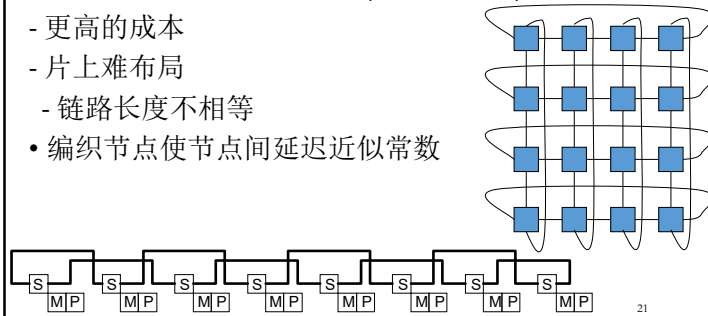


20

20

Torus

- Mesh 在边缘部分不对称: 在边缘放置任务时性能会非常敏感
- Torus 避免了这个问题
- + 比mesh更高的路径多样性(和对分带宽)
- 更高的成本
- 片上难布局
- 链路长度不相等
- 编织节点使节点间延迟近似常数



21

树

平面、分层拓扑结构

延迟: $O(\log N)$

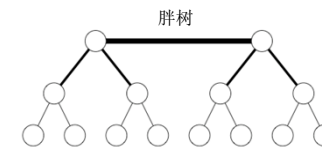
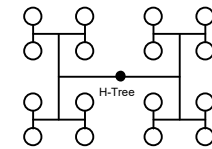
对局部流量很有效

+ 便宜: $O(N)$ 成本

+ 容易布局

- 根会成为瓶颈

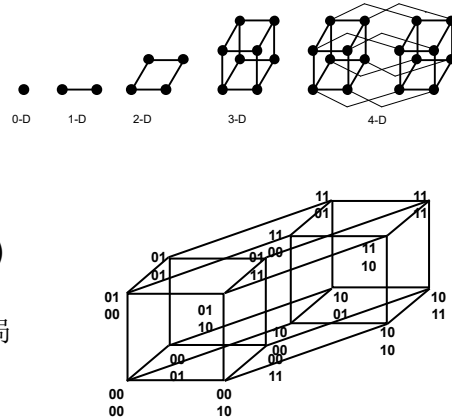
胖树可以避免这一问题(CM-5)



22

超立方

- 延迟: $O(\log N)$
- 基数: $O(\log N)$
- 连接数: $O(N \log N)$
- + 低延迟
- 在2D/3D中难布局

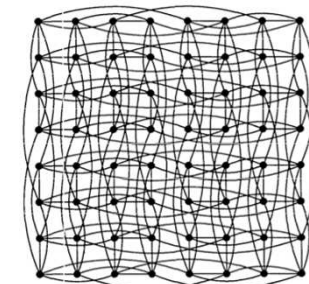
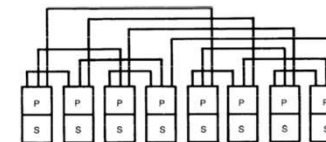


23

加州理工的宇宙立方

- 64节点的消息传递机器

- Seitz, "The Cosmic Cube," CACM 1985.

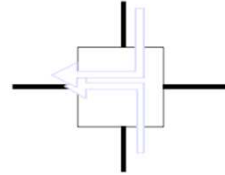


A hypercube connects $N = 2^n$ small computers, called nodes, through point-to-point communication channels in the Cosmic Cube. Shown here is a two-dimensional projection of a six-dimensional hypercube, or binary 6-cube, which corresponds to a 64-node machine.

FIGURE 1. A Hypercube (also known as a binary cube or a Boolean n -cube)

24

处理竞争



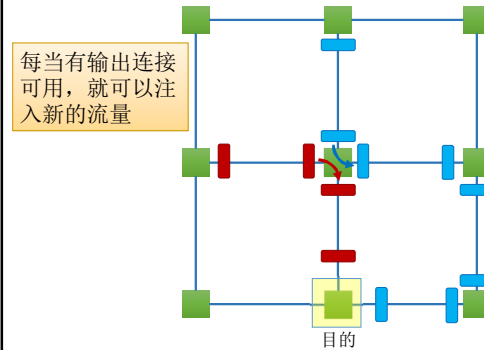
- 两个包试图同时使用同一个连接
- 如何处理?
 - 缓冲一个
 - 丢弃一个
 - 偏转 (错误路由) 一个
- Tradeoff?

25

25

无缓冲偏转路由

- 核心思路: 包在网络中绝不缓冲, 当两个包争用同一连接时, **偏转** 其中一个



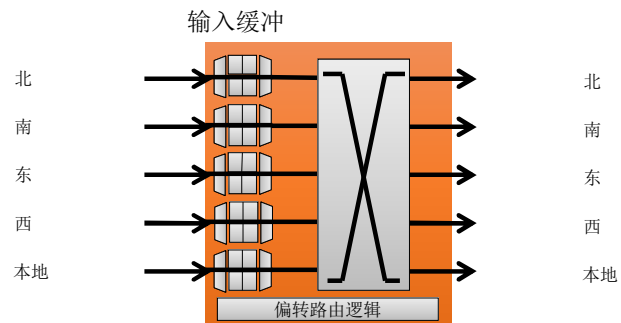
Baran, "On Distributed Communication Networks." RAND Tech. Report., 1962 / IEEE Trans.Comm., 1964.

26

26

无缓冲偏转路由

- 可以免除输入缓冲: 传输的包会通过**流水线锁存器**和**网络连接**“缓冲”



27

27

路由算法

- 类型
 - **确定的**: 总是为一个源-目的对之间的通信选择同样的路径
 - **健忘的**: 选择不同的路径, 不考虑网络状态
 - **适应的**: 可以选择不同的路径, 适应网络的状态
- 如何适应
 - 局部/全局反馈
 - 最小或非最小路径

28

28

确定性路由

- 相同(源, 目的)对的包走同样的路径
- 维度序路由
 - 比如, XY 路由(Cray T3D, 其它很多片上网络)
 - 先遍历维度 X, 再遍历维度 Y

+ 简单

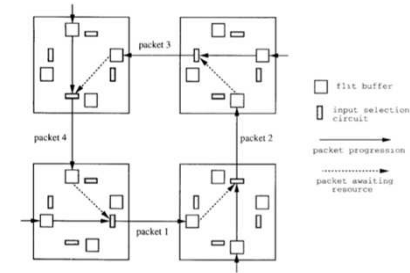
+ 无死锁(资源分配不需要时钟周期)

- 可能导致高度竞争
- 没有利用路径多样性

29

死锁

- 没有进程可以向前推进
- 由对资源的循环依赖引发
- 每个包等待被下游包占据的缓冲区



30

处理死锁

- 避免路由中的环
 - 维度序路由
 - 不会产生循环依赖
 - 限制每个包的“轮次”
- 通过增加缓冲避免死锁(逃生路径)
- 检测并突破死锁
 - 抢占缓冲区

31

避免死锁的转向模型

- 思路
 - 分析一个网络中包可能的转移方向
 - 确定这些转向可以构成的环
 - 禁止某些转向以打破可能的环
- Glass and Ni, “The Turn Model for Adaptive Routing,” ISCA 1992.

FIG. 2. The possible turns and simple cycles in a two-dimensional mesh.

FIG. 3. The four turns allowed by the xy routing algorithm.

FIG. 4. Six turns that complete the cycles and allow deadlock.

32

健忘性路由: 勇士算法

- 健忘性算法的例子
 - 目标: 均衡网络负载
 - 思路: 随机选择一个中间目的节点, 首先路由到该节点, 接着从该节点路由到最终目的
 - 源-中间节点和中间节点-目的, 可以使用维度序路由
- + 随机的/均衡网络负载
- 非最小(包延迟可能增加)
- 优化:
 - 在高负载时使用
 - 限制中间节点

33

33

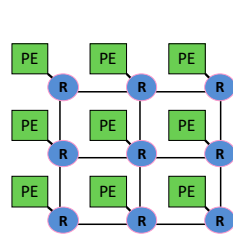
适应性路由

- 最小适应性
 - 路由器根据网络状态(比如, 下游缓冲区的占用情况)来选择高效输出口发送包
 - 高效输出口: 能使包离目的更近的端口
 - + 能感知局部拥塞
 - 追求最小性限制了高连接利用率的获得(负载均衡)
- 非最小(完全)适应性
 - 根据网络状态将包“错误路由”到非高效输出口
 - + 能够获得更好的网络利用率和负载均衡
 - 需要保证避免活锁

34

34

片上网络

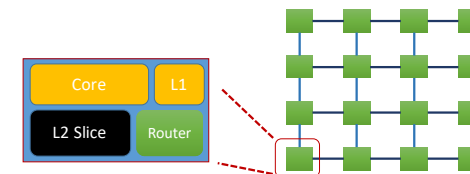
- 
- 连接核、cache、内存控制器等
 - 总线和交叉开关不具有可扩展性
 - 包交换
 - 2D mesh: 最常用的拓扑
 - 主要用来应对cache缺失和访存请求
- R 路由器
- PE 处理单元
(核, L2 Bank, 内存控制器, 等等)

35

35

高效互连的动机

- 在众核芯片中, 片上互连(NoC)消耗了巨大的能量
- | 芯片 | 功耗 |
|-----------------|-----------|
| Intel Terascale | ~28% 芯片功耗 |
| Intel SCC | ~10% |
| MIT RAW | ~36% |



- 最近的一些工作利用无缓冲偏转路由来减小功耗和芯片尺寸

36

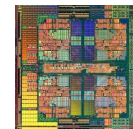
36

多核设计

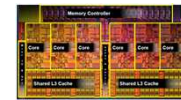
37

片上众核

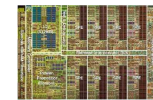
- 比一个大核更简单并且功耗更低
- 片上大规模并行



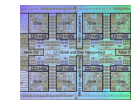
AMD Barcelona
4 cores



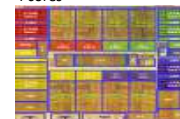
Intel Core i7
8 cores



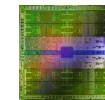
IBM Cell BE
8+1 cores



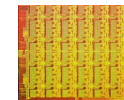
IBM POWER7
8 cores



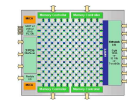
Sun Niagara II
8 cores



Nvidia Fermi
448 "cores"



Intel SCC
48 cores, networked



Tilera TILE Gx
100 cores, networked

38

对于片上多核

- 我们想要:
 - 当我们在N个核上并行化一个应用, 我们能获得N倍在单个核上的性能
- 我们能够得到:
 - Amdahl定律 (串行瓶颈)
 - 并行部分的瓶颈

39

回顾: 并行性

- Amdahl定律
 - f: 程序中可并行的部分
 - N: 处理器个数

$$\text{加速比} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

- 最大加速比受限于串行部分: 串行瓶颈
- 并行部分通常不完美
 - 同步开销 (比如, 对共享数据的更新)
 - 负载不均衡开销 (并行化不完美)
 - 资源共享开销 (N个处理器之间的竞争)

40

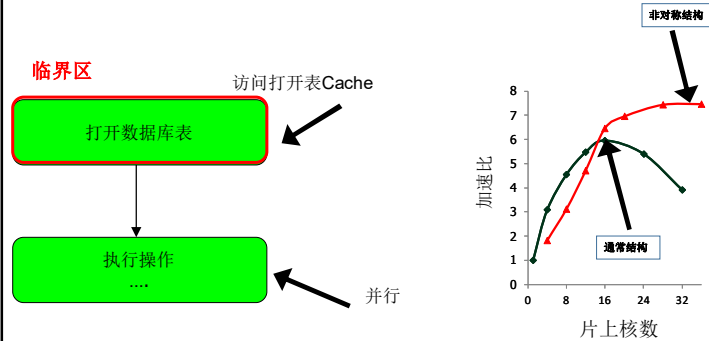
问题: 串行化的代码段

- 很多并行程序无法完全并行化
- 串行化代码段的产生
 - 连续的部分(Amdahl的“串行部分”)
 - 临界区
 - 栅障
 - 流水化程序中的受限阶段
- 串行化的代码段
 - 降低性能
 - 限制扩展性
 - 浪费能源

41

41

MySQL的例子



42

42

不同代码段的需求

- 我们想要:
- 串行代码段 → 一个强有力的“大”核
- 并行代码段 → 许多弱的“小”核
- 这两者互相冲突:
 - 如果你有一个强有力的核, 就不可能同时拥有很多核
 - 一个小孩在能耗和面积消耗方面都远比一个大核更高效

43

43

“大” vs. “小” 核

- | 大核 | 小核 |
|--|---|
| <ul style="list-style-type: none">• 乱序• 宽取指• 更深的流水线• 激进的分支预取器• 很多的功能单元• 内存依赖的投机• | <ul style="list-style-type: none">• 按序• 窄取指• 浅的流水线• 简单的分支预取器• 很少的功能单元 |

大核的功效低:
比如, 4x的面积(功率)只能获得2x的性能

44

大核 vs. 小核

- Grochowski et al., “Best of both Latency and Throughput,” ICCD 2004.

	Large core	Small core
Microarchitecture	Out-of-order, 128-256 entry ROB	In-order
Width	3-4	1
Pipeline depth	20-30	5
Normalized performance	5-8x	1x
Normalized power	20-50x	1x
Normalized energy/instruction	4-6x	1x

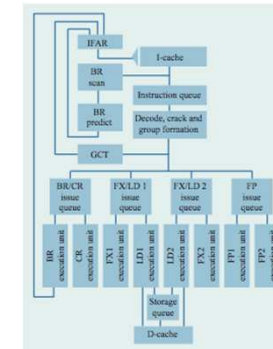
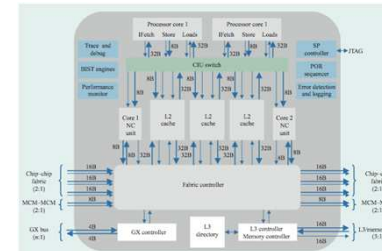
45

45

大核: IBM POWER4

- Tendler et al., “POWER4 system microarchitecture,” IBM J R&D, 2002.

- 另外一种对称多核芯片...
- 但是, 更少、更强有力的核



41

46

IBM POWER4

- 2 个核, 乱序执行
- 每核100-entry的指令窗口
- 8宽度取指、发射和执行
- 大容量, 本地+全局混合分支预测器
- 1.5MB, 8路 L2 cache
- 基于流的激进预取

47

47

IBM POWER5

- Kalla et al., “IBM Power5 Chip: A Dual-Core Multithreaded Processor,” IEEE Micro 2004.

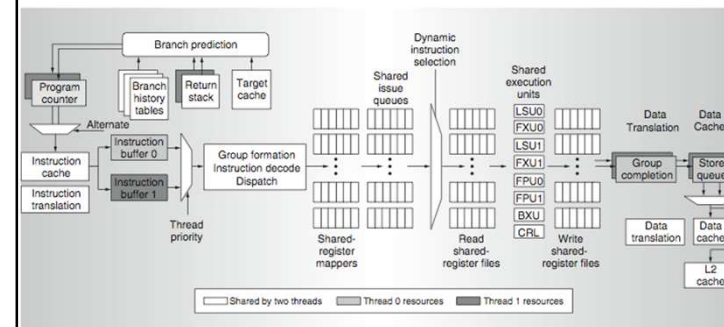


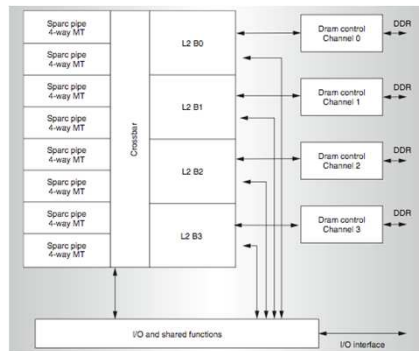
Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

41

48

小核: Sun Niagara (UltraSPARC T1)

- Kongetira et al., “Niagara: A 32-Way Multithreaded SPARC Processor,” IEEE Micro 2005.

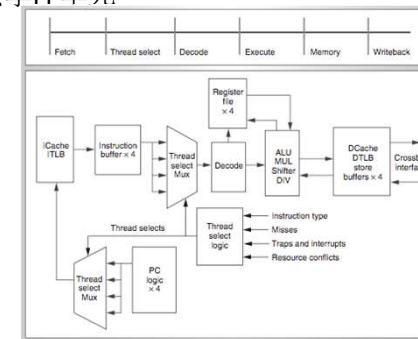


49

49

Niagara的核

- 4路细粒度多线程, 6阶段双发射按序执行
- 循环线程选择(除非发生cache缺失)
- 核间共享FP单元



50

回顾: 需求

- 我们想要:
 - 串行代码段 → 一个强有力的“大”核
 - 并行代码段 → 许多弱的“小”核
- 这两者互相冲突:
 - 如果你有一个强有力的核, 就不可能同时拥有很多核
 - 一个小核在能耗和面积消耗方面都远比一个大核更高效
- 能否两全其美?

51

51

性能 vs. 并行性

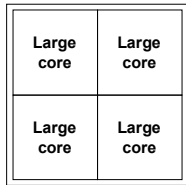
假设:

- 小核用1个面积的预算获得1份性能
- 大核用4个面积的预算获得2份性能

52

52

铺砌大核



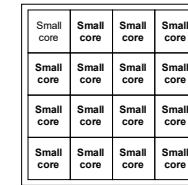
“铺砌大核”

- 铺砌少量大核
- IBM Power 5, AMD Barcelona, Intel Core2Quad, Intel Nehalem
- + 单线程、串行代码段时可获得高性能
- 并行程序段时吞吐率低

53

53

铺砌小核



“铺砌小核”

- 铺砌很多小核
- Sun Niagara, Intel Larrabee, Tiler TILE
- + 并行部分吞吐率高
- 串行部分、单线程性能低

54

54

两全其美?

- 铺砌大核
 - + 单线程、串行代码段时可获得高性能
 - 并行程序段时吞吐率低
- 铺砌小核
 - + 并行部分吞吐率高
 - 串行部分、单线程性能低, 相比现有单线程处理器性能还差
- 思路: 在一个芯片上同时集成大核和小核 → 性能不对称

55

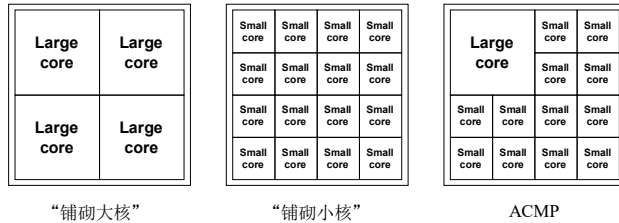
55

非对称多核

56

56

非对称片上多处理器(ACMP)



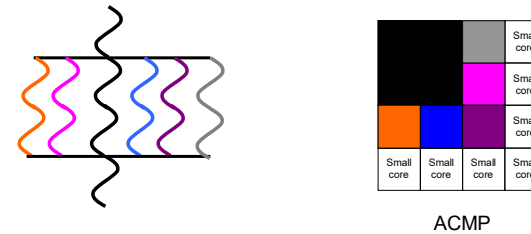
- 提供一个大核和多个小核
- + 利用大核加速串行部分
- + 在小核和大核上执行并行部分以获得高的吞吐率

57

57

加速串行瓶颈

单线程 → 大核



58

58

性能 vs. 并行性

假设:

1. 小核用1个面积的预算获得 1 份性能
2. 大核用4个面积的预算获得2份性能

59

59

ACMP 性能 vs. 并行性

面积预算 = 16 个小核

	“铺砌大核”	“铺砌小核”	ACMP
大核	4	0	1
小核	0	16	12
串行性能	2	1	2
并行吞吐	$2 \times 4 = 8$	$1 \times 16 = 16$	$1 \times 2 + 1 \times 12 = 14$

(6)

60

再审视：并行性

• Amdahl定律

- f: 程序中可并行的部分
- N: 处理器个数

$$\text{加速比} = \frac{1}{1 - f + \frac{f}{N}}$$

- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

• 最大加速比受限于串行部分: 串行瓶颈

• 并行部分通常不完美

- 同步开销 (比如, 对共享数据的更新)
- 负载均衡开销 (并行化不完美)
- 资源共享开销 (N个处理器之间的竞争)

61

61

加速并行瓶颈

- 并行部分中的序列化或不均衡的执行同样可以得益于大核

• 例子:

- 临界区竞争
- 比别的阶段执行时间更长的并行阶段

- 思路: 动态判别会导致序列化执行的代码段并将它们放到大核上执行

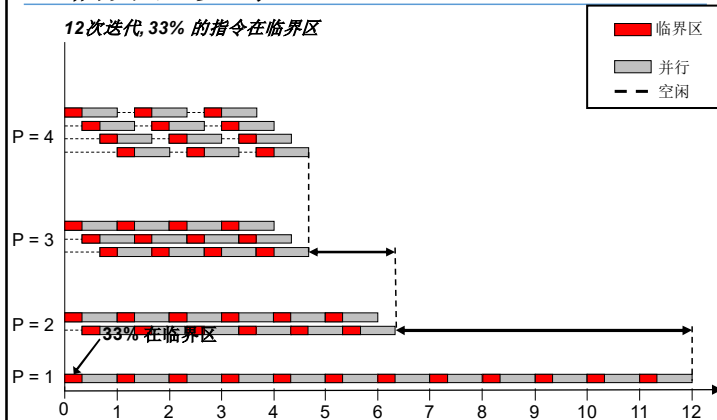
- 加速临界区
- 瓶颈识别和调度

62

62

临界区竞争

12次迭代, 33% 的指令在临界区

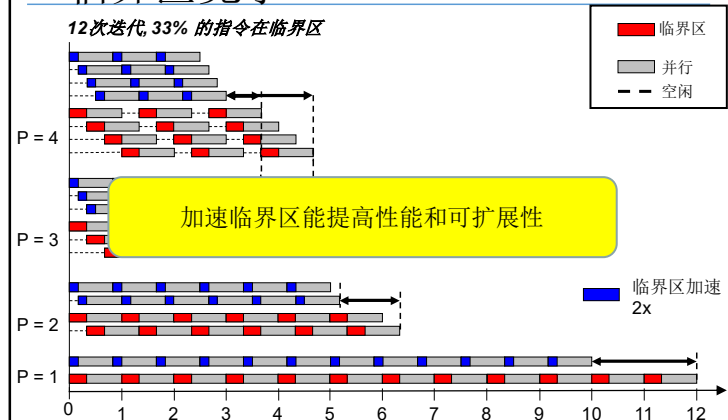


63

63

临界区竞争

12次迭代, 33% 的指令在临界区

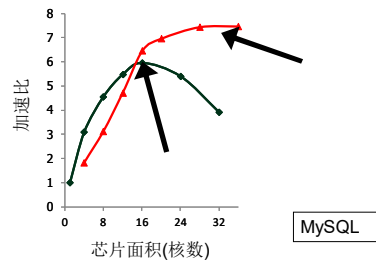


64

64

临界区对可扩展性的影响

- 临界区竞争导致并行程序段中线程的串行执行(序列化)
- 临界区竞争随着线程数增加而增加, 并且限制可扩展性



65

65

利用不对称

- 串行部分的执行时间必须短
- 对程序员来说缩短这些串行段相当困难
 - 领域知识不够
 - 硬件平台的多样性
 - 受限的资源
- 目标: 一种不需要程序员参与的缩小串行瓶颈的机制
- 思路: 在非对称多核平台上通过将串行代码段迁移到强有力的核上来加速串行部分的执行

66

66

一个例子: 加速临界区

- 思路: 在非对称多核体系结构中将临界区迁移到大的、强有力的核上
- 好处:
 - 减少由于锁的争用带来的串行化
 - 减少无法并行化的部分对性能的影响
 - 程序员不需要(过多地)优化并行代码 → 更少的bug, 提高效率
- Suleman et al., “Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009, IEEE Micro Top Picks 2010.
- Suleman et al., “Data Marshaling for Multi-Core Architectures,” ISCA 2010, IEEE Micro Top Picks 2011.

67

67

多线程应用中的瓶颈

一种定义: 任何会被线程竞争的代码段

例如:

- Amdahl的串行段
 - 只有一个线程在执行 → 在关键路径上
- 临界区
 - 保证互斥 → 如果有竞争很可能就在关键路径上
- 栅障
 - 再继续推进之前保证所有线程到达该点 → 最后到达的线程在关键路径上
- 流水线阶段
 - 一个循环迭代的阶段可能在不同线程上执行, 最慢的阶段会使其它阶段等待 → 在关键路径上

68

68

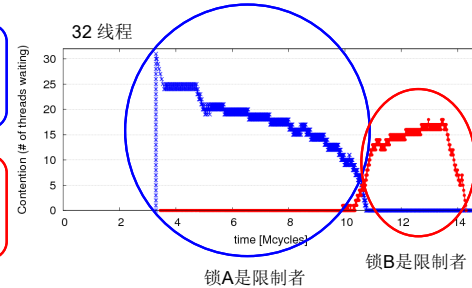
观察: 起作用的瓶颈随时间变化

A=满的链表; B=空的链表

repeat

```
lock A
  Traverse list A
  Remove X from A
unlock A
Compute on X
lock B
  Traverse list B
  Insert X into B
unlock B
```

until A is empty

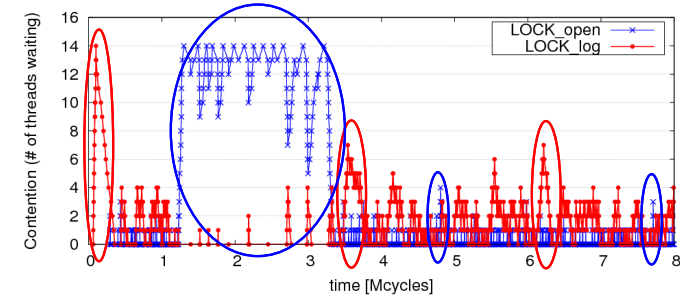


69

69

真实应用中瓶颈产生限制确实是变化的

MySQL运行Sysbench查询, 16 线程



70

70

瓶颈加速的一些研究

- 非对称片上多处理器(ACMP) [Annavaram+, ISCA'05]
[Morad+, Comp. Arch. Letters'06] [Suleman+, Tech. Report'07]
- 临界区加速 (ACS) [Suleman+, ASPLOS'09, Top Picks'10]
- 反馈指引的流水线 (FDP) [Suleman+, PACT'10 and PhD thesis'11]

- 不能加速所有类型的瓶颈
- 不能适应起作用瓶颈 (瓶颈重要性) 的细粒度变化

目标:

一种通用机制能够识别并加速任何类型的正在影响性能的瓶颈

Jose A, et.al "Bottleneck Identification and Scheduling in Multithreaded Applications", ASPLOS 2012.

71

71

瓶颈识别与调度

- 主要观点:
 - 线程等待损害并行性并且可能降低性能
 - 代码是导致大多数线程等待的原因 → 可能的关键路径
- 主要思路:
 - 动态识别导致大多数线程等待的瓶颈
 - 加速它们(使用ACMP中的强有力的核)

Jose A, et.al "Bottleneck Identification and Scheduling in Multithreaded Applications", ASPLOS 2012.

72

72