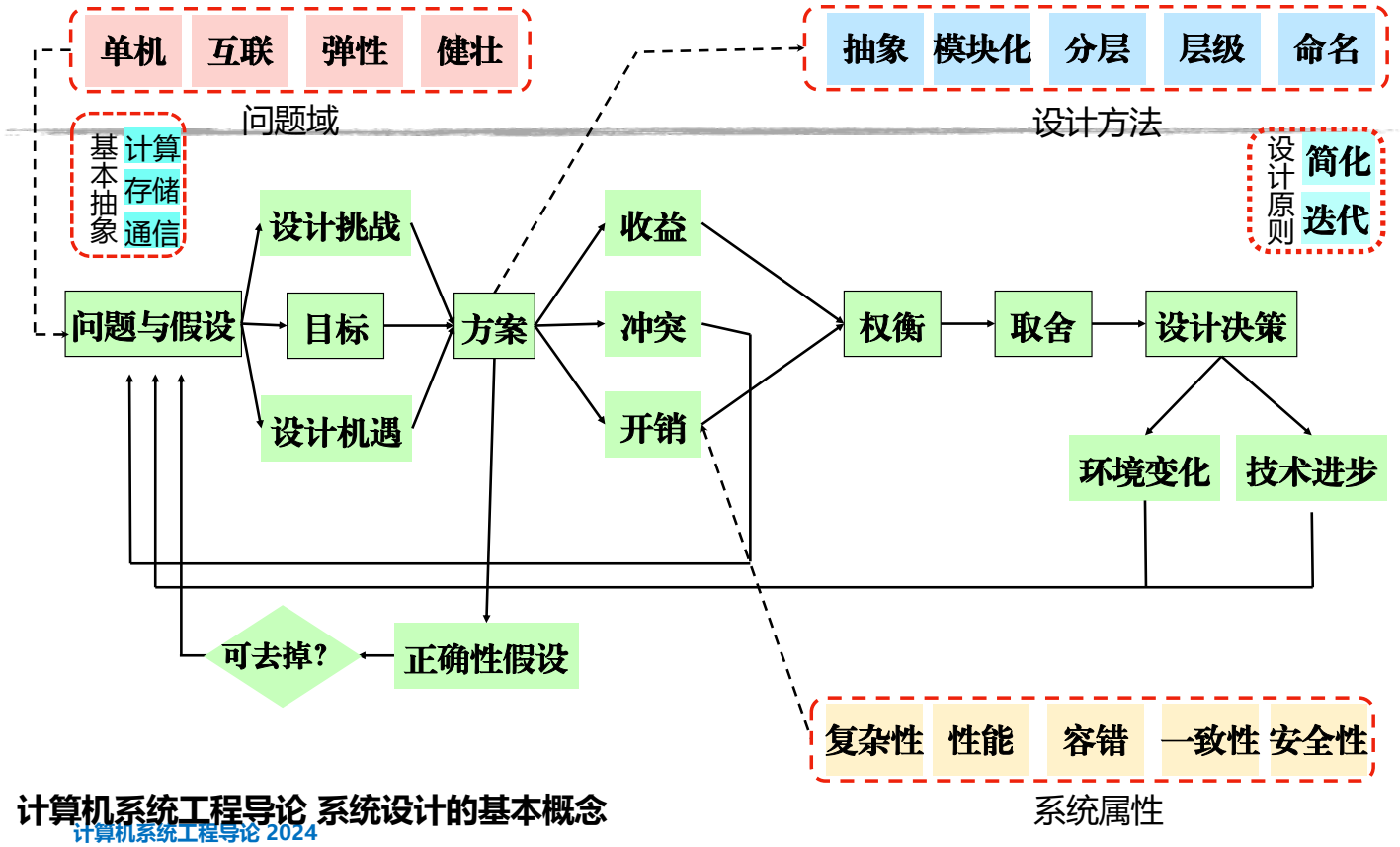


计算机系统工程导论 系统设计的基本概念

12. 性能

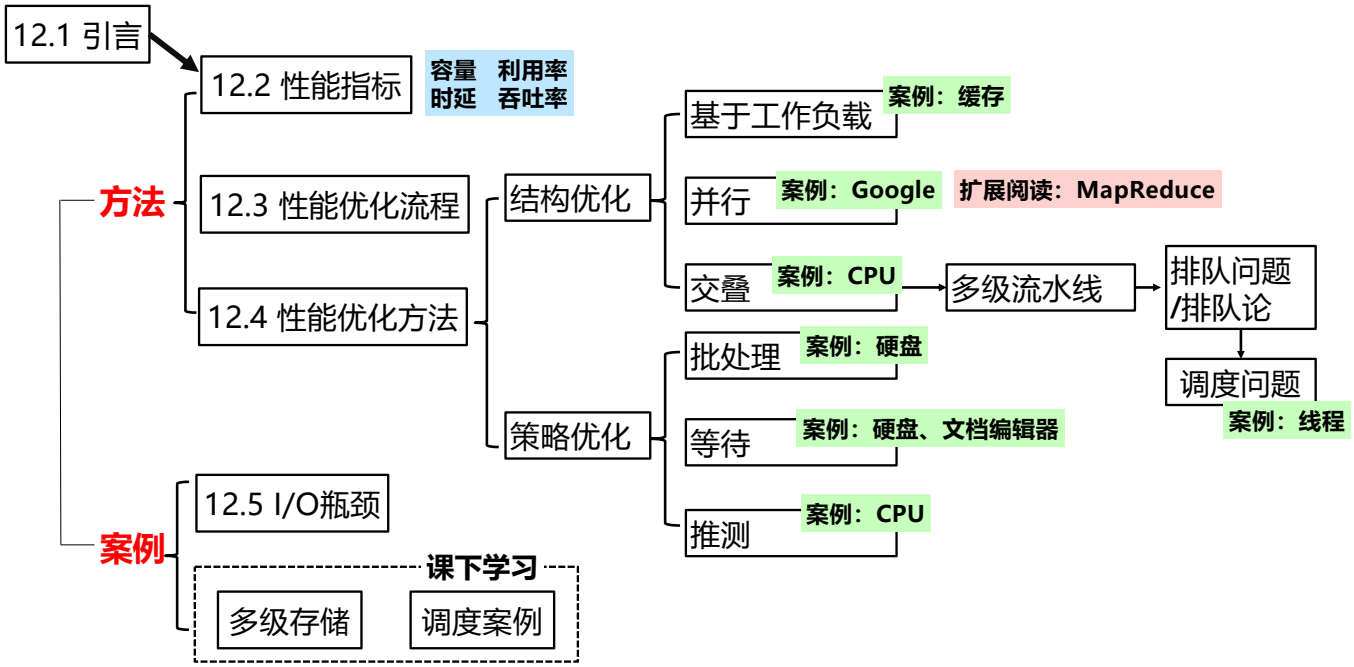
[redacted]

本章相关的参考文献

- LAMPSON B W. Hints for computer system design[J]. **ACM SIGOPS** Operating Systems Review, 1983, 17(5): 33-48. DOI: 10.1145/800217.806614.
- DEAN J, GHEMAWAT S. MapReduce: Simplified Data Processing on Large Clusters[C] //Proceedings of the 6th conference on Symposium on **Opearting Systems Design & Implementation**, 2004,51(1) : 107-113. DOI: 10.1145/1327452.1327492.

  Also in Communications of the ACM 2008, 51(1), 1–10.
- PATT Y N, GANGER G R. Metadata update performance in file systems[C] //Operating Systems Design and Implementation. USENIX Association, 1994: 5–15. DOI: 10.5555/1267638.1267643.
- **BRIN S, PAGE L**. The Anatomy of a Large-Scale Hypertextual Web Search Engine[J]. Computer Networks and ISDN Systems, 1998, 30(1-7):107-117. DOI: 10.1016/S0169-7552(98)00110-X.
- JAIN R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling[M]. New York, USA: Wiley-Interscience, 1991.

主要内容



12.1 引言

性能是什么

设计目标之一

- 计算机系统显式或隐含了性能目标

我们每天都在说性能（不限于计算机），
你能说出它的含义吗?
能够举出几个例子吗?

- 含义是：_____。
  - 提示：它是一个分数

- 例如：_____。

投票(匿名) 最多可选1项 设置

是否在《计算机系统基础》课程中学习过，或自学过 “性能”？

A 没有

B 1个课时以下

C 1-2个课时

D 2个课时以上

提交

投票(匿名) 最多可选1项 设置

通过何种方式学习或了解到性能?

A 仅仅在程序设计时考虑到性能，但未学习性能相关方法

B 在选修的课程中专门学习了性能方法

C 主要通过自学看书学习了性能方法

D 不仅看书，还做了CSAPP的性能Lab

提交

投票 最多可选3项 设置

假如你和小伙伴要去参加“互联网+”大赛，你觉得哪3个指标可能是你们最在意的?

A	尽量兼容各个平台
B	尽量具备各种（花哨的）功能
C	可靠性（不出错）
D	结果的正确率（假如有智能识别算法）
E	可维护性（修改方便）
F	程序易于测试（快速检验程序是否符合预期）
G	程序的模块化设计
H	界面友好、易用
I	尽量高性能（运行快、容量大……）
J	投入的经济成本和时间成本
K	deadline

提交

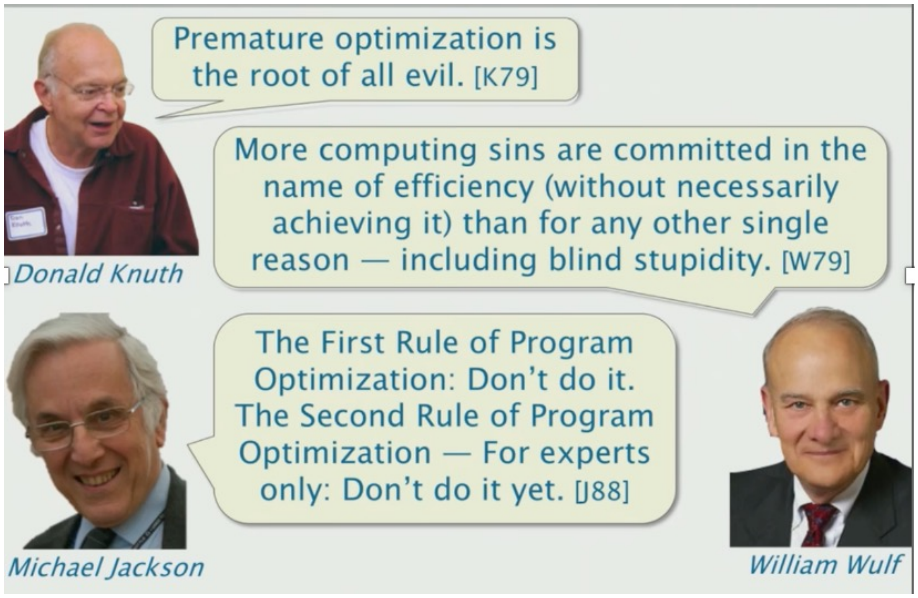
性能的意义是什么?

性能是计算机的货币

IBM System/360		DEC PDP-11		Apple II	
					
Launched:	1964	Launched:	1970	Launched:	1977
Clock rate:	33 KHz	Clock rate:	1.25 MHz	Clock rate:	1 MHz
Data path:	32 bits	Data path:	16 bits	Data path:	8 bits
Memory:	524 Kbytes	Memory:	56 Kbytes	Memory:	48 Kbytes
Cost:	$5,000/month	Cost:	$20,000	Cost:	$1,395

性能是免费的午餐吗?

性能是有代价的

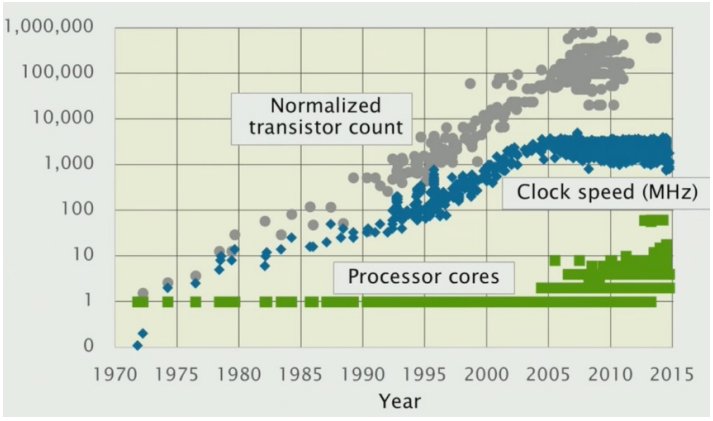


我们可以从摩尔定律、多线程获得性能提升。

为什么还要以复杂性、并发冲突为代价，收获一点点性能?

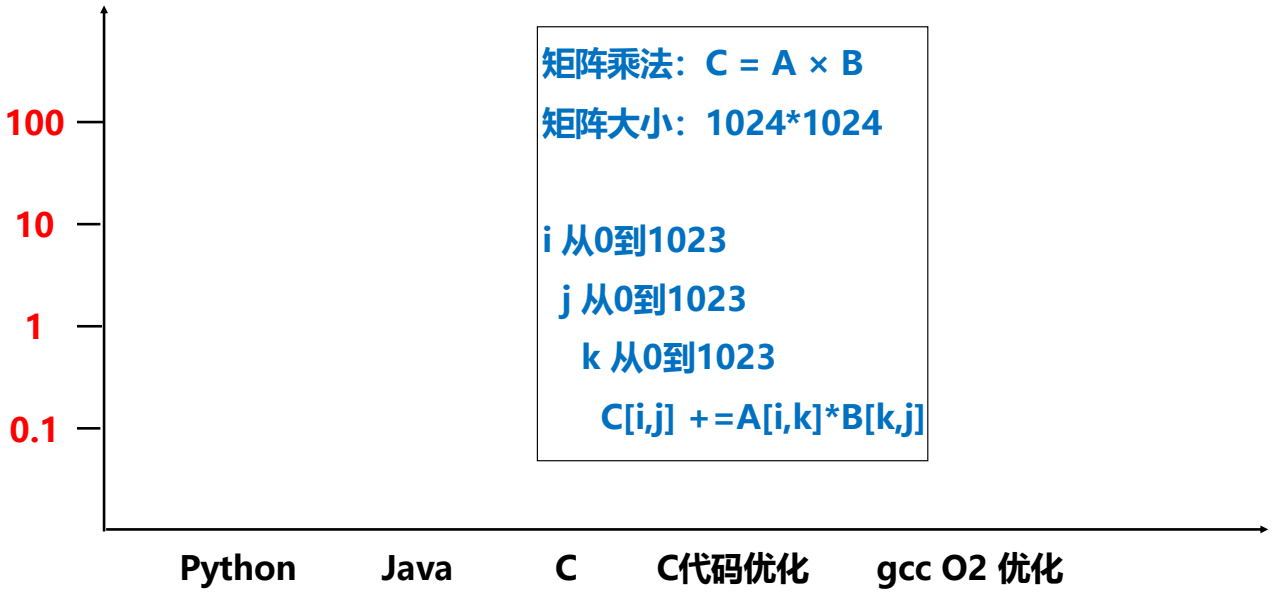
性能工程的意义

1. 摩尔定律不再有效



2. 性能工程不只是一点点提升

性能优化示例



12.1.1 性能工程的关键问题：瓶颈

什么是瓶颈?

- 瓶颈（bottleneck）= 系统中最慢的环节
- 未经性能优化的系统，通常都存在瓶颈

性能工程的关键

- 消除瓶颈
- 尽可能保持模块化、避免复杂性增长

性能瓶颈是怎么产生的?

① 各种限制导致技术维度发展不均衡

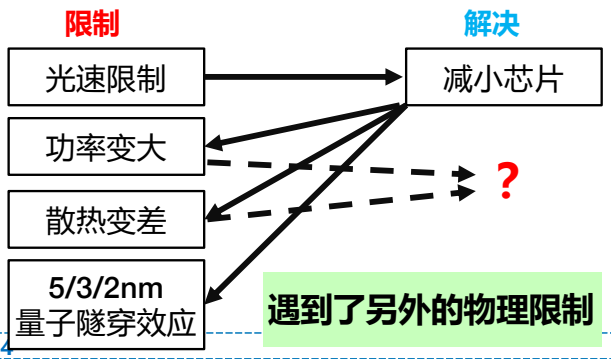
② 共享使性能变成了一个多目标问题

① 各种限制导致技术维度发展不均衡

物理

- 光速限制（计算、访存、通信：后者无法匹配前者的速度）
- 功率限制、散热限制

绕过限制时，经常遇到冲突，例：



① 各种限制导致技术维度发展不均衡

技术

- 目前图同构最好的算法时间复杂度是$O(2^n)$
- 目前排序最好的算法时间复杂度是 O(nlogn)
- 目前大数分解最好的算法时间复杂度是$O(2^n)$

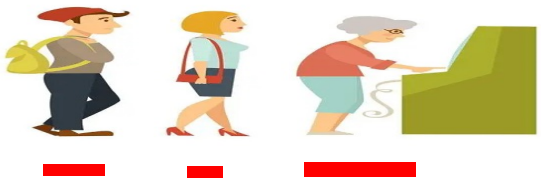
数据处理模块等待计算模块
插入模块等待查询模块
通信模块等待加密模块

② 共享使性能变成了一个多目标问题

很多组件共享同一资源（存储、通信、计算等）

共享的多种目标：

- 总体利用率最高
- 平均等待最短
- ……



无论采取何种目标，优先级低的组件的请求成为瓶颈

12.1.2 性能设计中的矛盾

① 是否投入成本进行性能设计？

$$\frac{dtech}{dt}$$

② 性能设计带来的问题如何解决？

性能 vs. 复杂性

①是否投入成本进行性能设计？

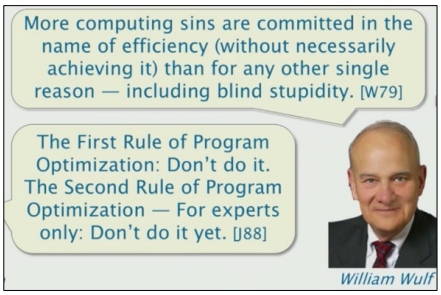
难题：如何判断性能优化在技术进步下是否有价值？

- 反例：高投入产出优化版本，产品发布时瓶颈已被技术消除
- 瓶颈是内在的还是技术依赖的？
  - 内在的：系统设计导致的。优化有价值！
  - 技术依赖的：技术限制导致的
- 当无法区分瓶颈的属性时：

设计经验 1：〔迟疑时，诉诸暴力〕

设计经验 1：迟疑时，诉诸暴力

- 拿不准就用暴力解决
  - 选择简单的算法，等待复杂的算法成熟和稳定
  - 选择简单硬件堆叠，等待技术发展带来新硬件
    - Thompson&Ritchie：定长UNIX进程表，快速线性查询
    - 暴力搜索的高性能象棋程序战胜人类选手
    - 暴力堆叠的大模型围棋程序战胜人类选手
    - 新的暴力方法建立在当下问题的解决之上



②性能设计带来的问题如何解决?

主要问题：复杂性提升

采取KIS原则（Keep It Simple）和两种设计方法：

1. 抽象方法
   - 通过抽象可以隐藏实现、保持模块化
2. 接口保持方法
   - 组件性能多样性（如存储），通过接口保持，保持简单性

单选题 1分 设置

假设你需要开发一个程序，程序的首要目标是尽可能减少执行时间，则你首选：

A **用Java开发**

B **用Python开发**

C **用C开发**

D **用LIsp开发**

提交

12.1 总结

性能是一定成本/时间下的收益

性能工程的意义：收益客观，弥补摩尔定律

**体会性能优化的乐趣

12.1.1 性能工程的关键是去除瓶颈

性能瓶颈因限制导致的不平衡而发生

性能瓶颈因共享导致的多目标而变得复杂

12.1.2 性能设计中的矛盾

进行性能设计的投入是否有意义?

性能设计带来的问题如何解决?

12.2 性能指标

性能指标

计算机系统工作场景

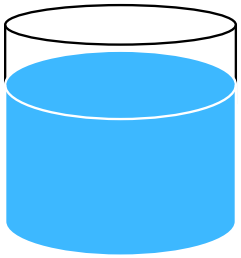
- C/S架构：模块化，请求/服务，时间与数量

主要性能指标

① 容量（capacity）：　　提供多少?
② 利用率（utilization）：　有无浪费?
③ 时延（latency）：　　等待多久?
④ 吞吐率（throughput）：　处理多快?

① 容量

容量 = 服务提供的资源的大小或数目



例：

a) *CPU容量：

b) 硬盘容量：

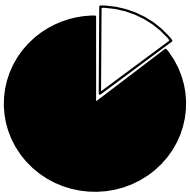
c) Web服务器容量：

d) 网络链路容量：

② 利用率

利用率：服务资源用于给定工作负载的比例

什么是工作负载？
是否有相对性？



例：OS vs. APP

例：

a) CPU利用率：

b) 硬盘利用率：

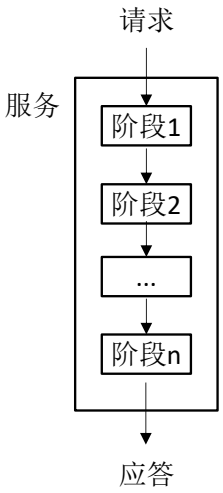
c) Web服务器利用率：

d) 网络链路的利用率：

③ 时延

时延：开始和结束之间的时间差

是否有相对性？ Server vs. Client



例：

- Web服务器：
- 网络：
- 过程（函数）：

串行任务/流水线（pipeline）时延：

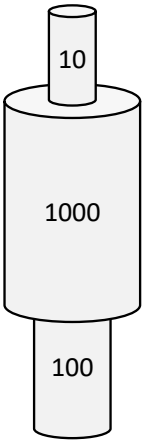
- A+B 的时延 ≥ 时延A + 时延B

④ 吞吐率

吞吐率：完成有用工作（useful work）的速率

什么是有用工作？

例：链路层和网络层对比



例：

- Web服务器：
- 网络：

串行任务吞吐率：

- A+B的吞吐率 ≤ minimum { 吞吐率A, 吞吐率B }
- 限制了快速部件的利用率

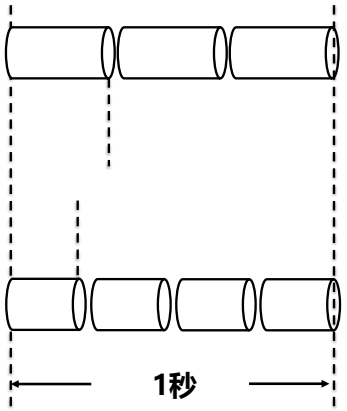
吞吐率与时延的关系

串行处理时:

- (最大)吞吐率= 1 / (处理)时延

并行处理时:

- 没有直接的关系



对于提高吞吐率，给我们什么启发?

减少时延是否可以?

如果时延不能减少呢? 多部件并行是否可以?

进一步思考：吞吐率和时延，哪个更容易提升?

吞吐率——购买更多服务器、带宽……

时延——你不能改变物理定律：光速、CPU频率……

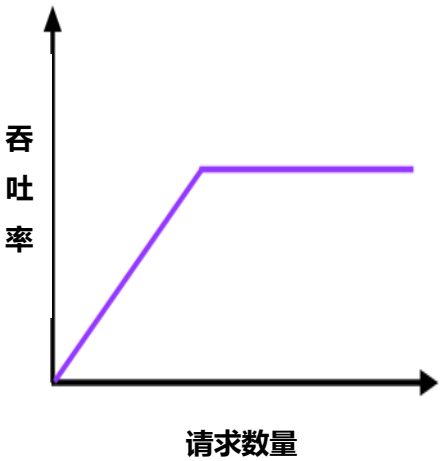
吞吐率与时延的关系

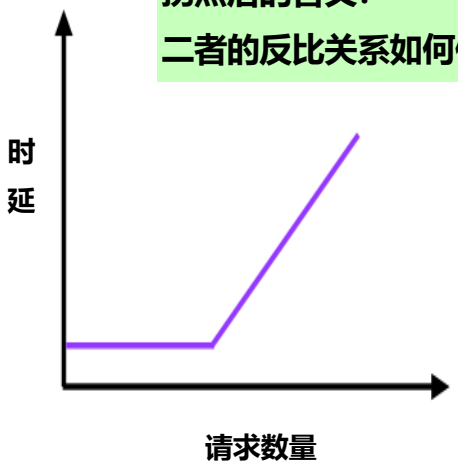
观察这两个图所表示的现象，思考与讨论

拐点前的含义?

拐点后的含义?

二者的反比关系如何体现?





12.2 总结

4个主要性能指标：容量、利用率、时延、吞吐率

串行结构：

时延大于组件时延的总和

吞吐率小于最慢组件的吞吐率

时延与最大吞吐率成反比

排队时延在到达最大吞吐率时开始产生

并行结构：

时延与吞吐率无关

12.3 性能优化流程

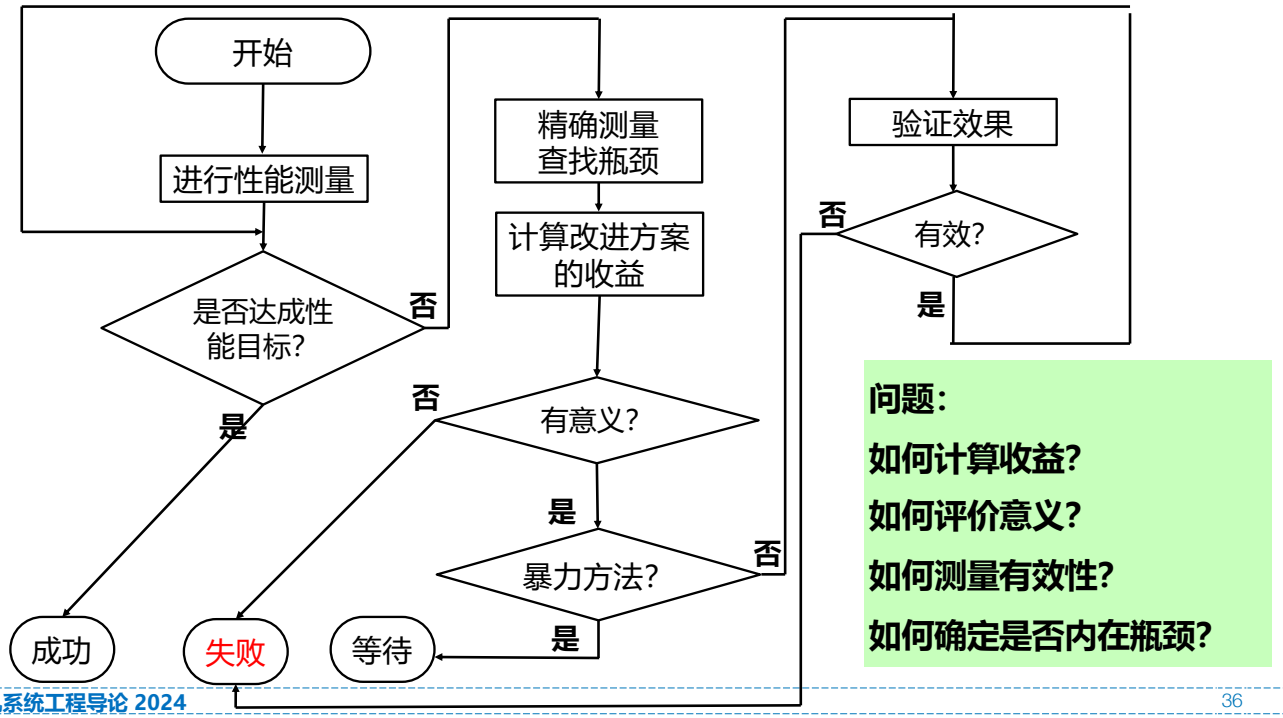
性能优化流程

系统化方法：

1. 通过测量、分析、计算，找到瓶颈
2. 提升瓶颈模块的性能
3. 迭代

- 牢记收益递减法则（law of diminishing returns）
- 如果收益递减、难以达成目标，重新审视系统的设计！

迭代优化流程图



12.4 优化设计方法

12.4.1 结构优化

减少时延的思路

单个部件/请求的时延难以减少

- 物理：一束信号从青岛传输到旧金山所需的传播时间
- 技术：从哈希表(红黑树)里查询一条记录的计算次数
- 经济：价格限定下的10T存储的访问时延

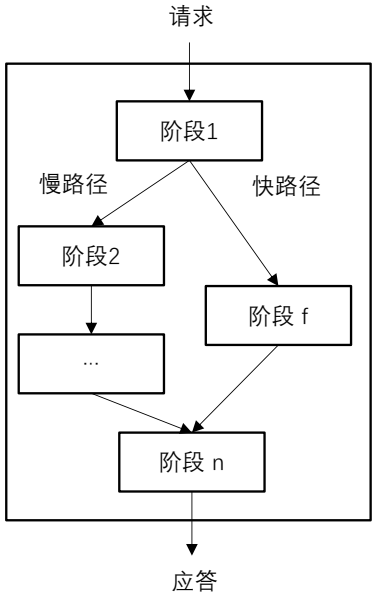
多个部件/请求的平均时延可以减少

- 利用资源的(不平均)分配
- 并行服务
- 交叠隐藏

① 基于工作负载

快路径/慢路径（基于非均匀性）

- 为常见的操作提供快路径
  - 典型代表：高速缓存（cache）

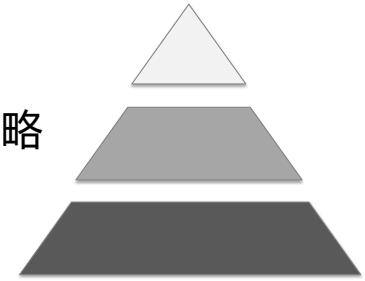


设计经验2:〔优化高频用例〕

设计经验2：优化高频用例

- 优化高频用例（缓存）
  - 计算机存储：多级存储
    - 缓存1ns，内存100ns，90%的命中率，平均时延是多少?

1*90%+100*10%=10.9ns

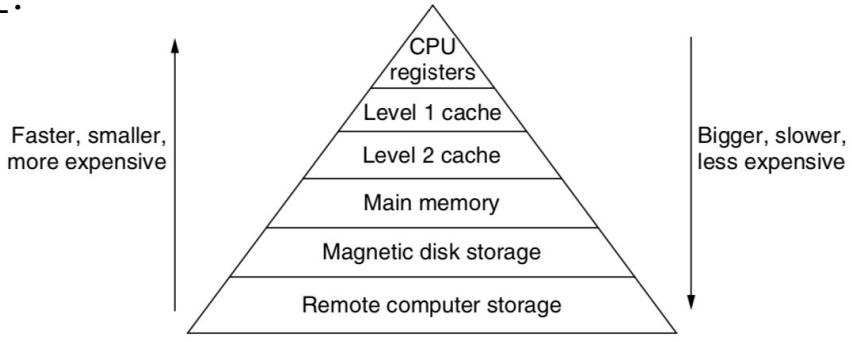


  - 设计提示：有存储抽象的地方都可以使用该策略
    - 域名：cache in DNS
    - 浏览器：页面内容、地址解析结果
    - 网站：CDN

例：基于虚拟内存的多级存储管理

多级存储

- 容量和时延的相关性：



两种管理方式

- 程序员手工 / 系统自动（自动管理子系统）
  - 平均性能优良的优秀自动管理算法
  - 保证软件的硬件无关性、降低编程复杂性、提高透明性

但是，有无缺点?

收益

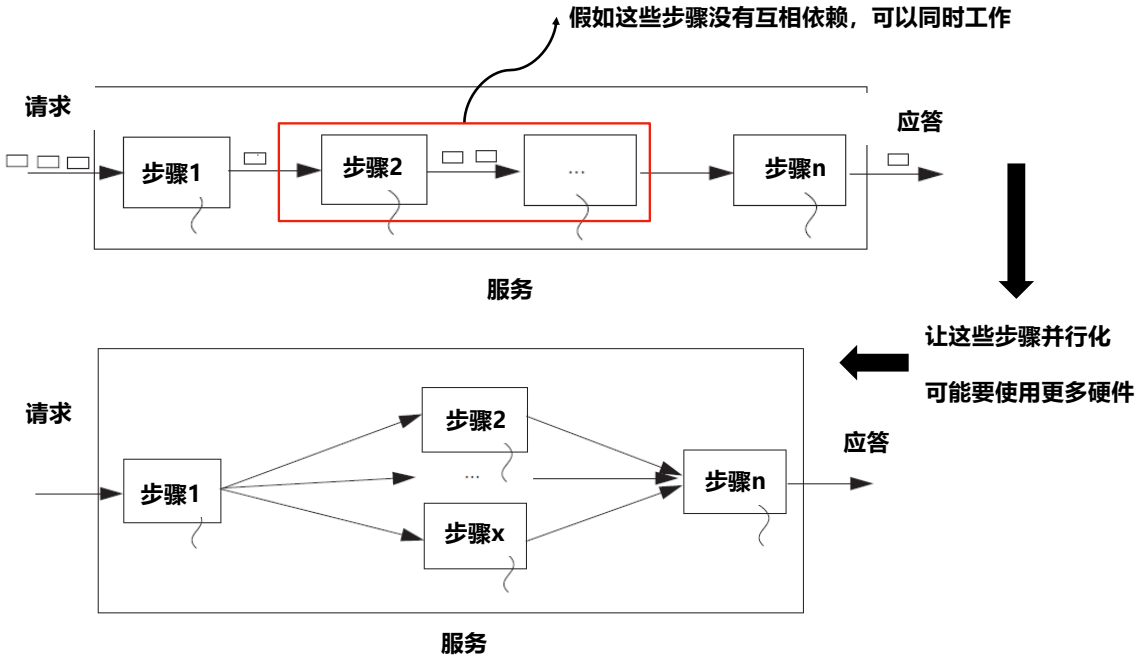
平均时延下降

- 平均时延是各类情况时延的频率加权平均

是通用的吗？以进程缓存为例：

- 代码
- 数据
- 总结：分布规律决定是否适用

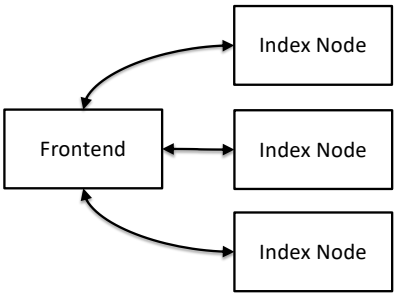
② 并行



② 并行

什么情况下可采用并行模块?

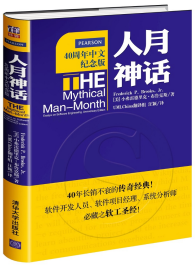
- 是否可将一个任务拆分为可并行的任务?
- 示例: Google搜索引擎



- 访问缓存（cache），能否拆分?

并行化的提升效果

理想：吞吐率1→n，时延1→1/n



1个人10个月的工作 $\stackrel{?}{=}$ 10个人1个月

实际：小于设想

- 不完全并行化: 子任务依赖、资源共享
- 并行化的开销: 模块间通信交换中间结果
- 并行不平衡: 存在快/慢路径

✍ 扩展阅读

Brin S, Page L. The anatomy of a large-scale hypertextual web search engine[J]. Computer networks and ISDN systems, 1998, 30(1-7): 107-117.

- 该论文首先在www'98会议发表（google引用23667次，2024.2）
- 介绍了Google早期的搜索引擎性能优化工作
  - web索引被分片存于不同主机；
  - 前端将查询发给所有主机，回收结果并排序，生成网页并给浏览器
- 思考：哪些是未并行/并行部分？哪些是并行开销？

并行化的挑战

※ 课后阅读 ※

1. **很多应用难以并行化**
2. **实现并行化的复杂性**
   - 死锁/争用

Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.（先发表于OSDI'04，google引用24792）

- 大型集群/大规模数据的并行计算的开发与管理实例
- 贡献：对程序开发者隐藏了背后的并发、容错等

困难重重，但核数提升已是处理器提升的主要路线

③ 交叠

如果只有串行资源（例如处理器），那么前两个策略不适用

- （以处理器执行线程为例）

  指令1→指令2→……→指令n

1. 快路径/慢路径策略不适用
2. 步骤之间有依赖
   - 程序 = <指令→指令2→……> 因资源串行，不适用
3. 步骤拆分成子步骤，子步骤之间也有**强**依赖
   - 指令 = <取指→译码→执行→访存→写回> 也无法并行

③ 交叠

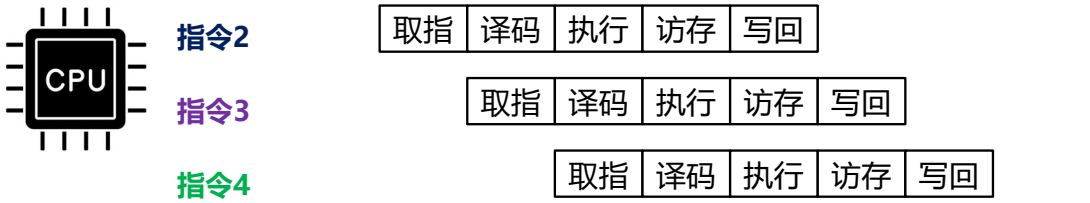
思考：不同步骤的子步骤也是前后严格依赖的吗？

拆分：取指→译码→执行→访存→写回

不严格依赖：第一个子步骤不依赖上一步骤的最后一个子步骤

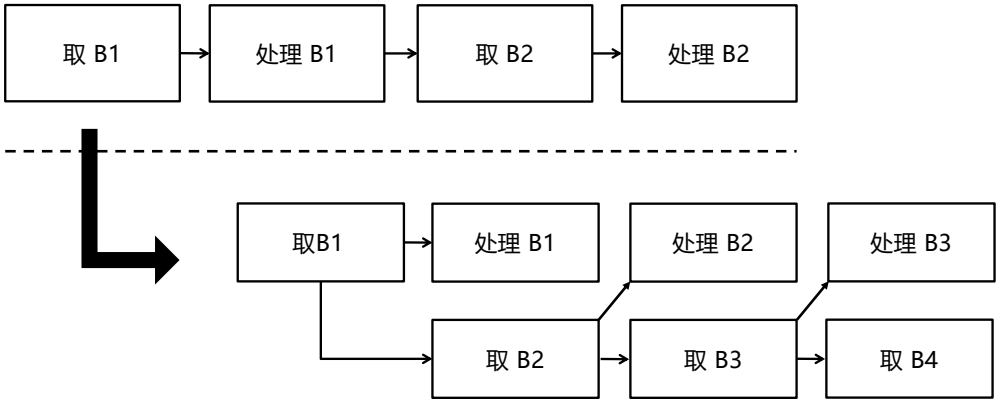
新方法：交叠（overlapping，或称流水线pipeline）

优点：不需增加服务能力



交叠的特点：隐藏时延，提升吞吐率

1. 如可拆分为并行子步骤，减少时延 → 提升吞吐率
2. **交叠**拆分为串行子步骤，隐藏时延 → 提升吞吐率
   - 解耦 → 设计流水线



设计经验3: 时延难以减少？隐藏它

- “One cannot bribe God”（David Clark）
  - 光速：青岛→乌鲁木齐 ≈ 10毫秒
  - 处理器的主频也已受到光速制约
- 运用交叠隐藏时延
  - **处理器**的多级流水线
  - **分布式计算**的传输与计算交叠
  - **浏览器**对页面分时加载（显示与传输交叠）

David Dana Clark

Born	April 7, 1944 (age 79)
Nationality	American
Known for	Clark-Wilson model, Multics
Awards	SIGCOMM Award Telluride Tech Festival Award of Technology IEEE Richard W. Hamming Medal (1998)
Scientific career	
Fields	Computer Science
Institutions	Internet Architecture Board National Research Council MIT Humboldt University of Berlin

交叠面临的挑战

1.子模块的工作均衡问题

- 有的模块跟不上进度
- 缓解：子模块间的有界缓冲区
  - 适用何种情况？（why）
  - 新问题：排队的研究

交叠面临的挑战

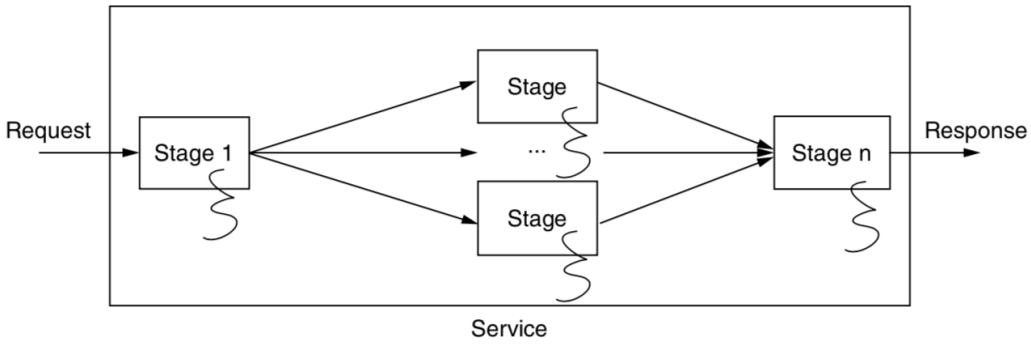
2. 需要有多个并发的请求

- 如何并发?
  - 多客户端
  - 单客户端异步并发
- 异步面临的挑战
  - 不同时间的请求/应答如何匹配

交叠提供的设计机会

交错技术（interleaving)

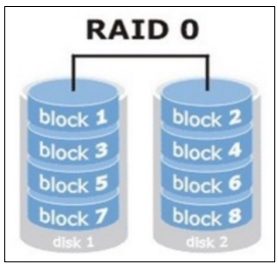
- 流水线和并行的综合运用
- 典型应用场景：存储



交错技术的应用

RAID 0

- 将存储块（block）依次存于不同的盘上
- 读盘时，RAID卡并行读多个硬盘，到RAID缓存中



多通道内存

- 多条指令或数据存于不同内存条，CPU并行读取

排队问题

排队的原因：

- 请求超过服务容量

排队的系统状态：

- 吞吐率最低的模块过载运行（常伴随其他模块空转）

排队时延的特点

- 与请求数量有关
- 随时间变化

排队论

排队分析的数学与计算方法

- 数学(110) - 运筹学(74) - 排队论(25)
  - 《GB13745 学科分类与代码》
- 为管理决策提供科学依据的应用数学
- 以概率论、随机过程等为基础

排队论起源于对电话排队的研究：

Erlang A K. The theory of probabilities and telephone conversations[J]. Nyt. Tidsskr. Mat. Ser. B, 1909, 20: 33-39.

时延分析：最简单的M/M/1排队模型

- 请求抵达的分布：~ P（λ）
  - 请求是独立、随机到达的 → 请求的间隔时间呈指数分布
- 服务时间分布：~ P（μ）
  - 指数分布、独立的服务时间

1. λ比μ大会怎么样?
2. 单位时间到来的请求数量呈什么分布?
3. 随着利用率的提升，队伍有什么变化?

- 服务数量：单服务
- 服务策略：先来先服务（FCFS）
- 排队和请求的最大容量均无上限

例 CPU指令 - 内存

CPU执行指令 1ns，访存 10 ns

- CPU应当提前9条指令发出访存请求，避免等待
- 如果每条指令都访问内存，内存成为瓶颈
  - 改进：内存管理器10个并发 → OK!
- 如果平均每2条指令访问一次内存
  - 改进：内存管理器5并发 → OK?

例 CPU指令 - 内存

问题：如果平均每2条指令访问一次内存，内存5并发，OK?

- 答：取决于应用程序模式

  a. 每2条指令1访存：队列长度固定、时延固定

  b. 每10条指令5访存：队列长度有上限（5）、时延波动

  c. 如果随机分布：队列长度无上限、平均时延ρ/(1-ρ)

是否超出设想？设计者应当理解非平衡模式（non-uniform）带来的问题！

12.4.2 调度问题 (Scheduling)

调度问题

排队时，不同请求的顺序问题。

- 硬盘访问：如何最小化时延？最优排序?
- 服务请求：FIFO？考虑请求的耗时大/小?
  - 提高吞吐率可能导致某个用户的极高时延
- …

这些问题构成了调度问题。

问题随技术发展而改变但本质不变。

调度的本质

计算机系统是请求使用资源的实体的集合

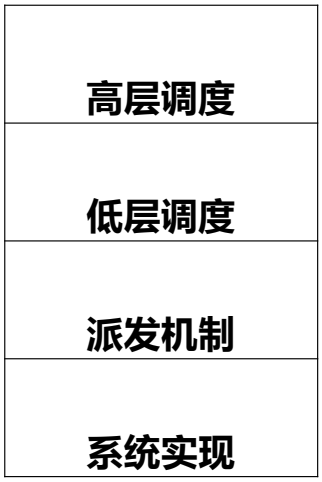
- 实体：线程、虚存地址、用户、客户端
- 资源：CPU周期、物理内存、硬盘空间、网络容量

调度是派发策略和派发机制的集合，将资源分配给实体

- 基于性能目标和最低保障进行排序，并实施
- 实现：调度器（scheduler）

调度器设计的3个难题

1. **高层目标与低层调度的鸿沟**
2. **策略与派发机制的鸿沟**
3. **派发机制与实现的鸿沟**



① 高层目标与低层调度的鸿沟

高层的调度策略难以被低层模块了解

例：web服务

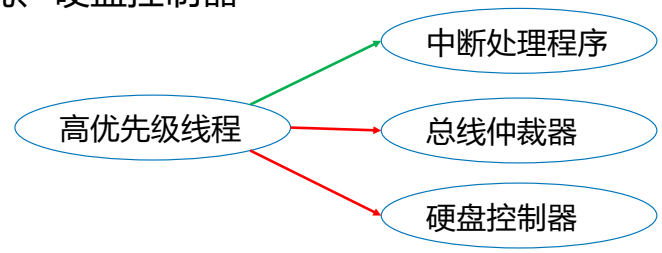
- 策略：有价值的用户优先服务
  - 如何分析请求来自有价值的用户？
  - 请求在模块之间传递时会丢失用户信息

② 调度策略与派发机制的鸿沟

正确的策略难以派发到所有模块，导致低效率的情况

例：操作系统内核

- 策略：高优先级线程先执行
  - 给予线程高优先级，但是无法派发到总线仲裁器
  - 也无法派发到：共享文件系统、硬盘控制器

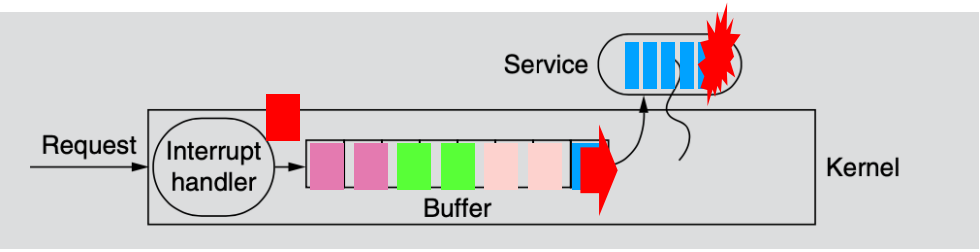


③ 派发机制与实现的鸿沟

如果派发机制与实现不匹配，会出现活锁

例：receive livelock

- 当 吞吐率1000遇到10000并发，但如采用中断实现，利用率为0
- 思考：你能想出什么解决方法？·



困难与实际

困难

- 解决调度问题，需要设计复杂的策略
  - 例如：航线调度

实际

- 大多数计算机系统中，资源不昂贵，可采用简单方案

调度器的设计挑战

调度器主要功能：调度、派发

- 调度策略：谁先谁后
- 派发机制：执行换入换出

设计思路

- 调度策略需要动态变化
- 但派发机制不需要改变
- 如何提高调度器的设计灵活性？
  - 间接层！

设计经验4: 将实现从策略中分离

- 把决策留给模块的client
  - 灵活替换
  - 便于移植（仅改写实现部分）
- 前提
  - 接口需定义良好
- 缺点
  - 性能略受影响，复杂度略上升
- 成功案例：在多级存储管理中，缺页处理机制和替换策略是分开实现的

常用的调度指标

对于请求

1. 周转时间（Turnaround time）：请求抵达到执行完毕的时间
2. 响应时间（Response time）：请求抵达到开始产生响应的时间
3. **等待时间（Waiting time）：请求抵达到被开始处理的时间**

服务器的视角
体现了排队延迟

- 调度指标：
  - 平均周转时间
  - 平均响应时间
  - 平均等待时间

常见的调度策略

1. **先来先服务（First-Come, First-Served）**
2. **最短工作优先（Shortest-Job-First）**
   - 改进：最短工作优先完成
3. **时间片轮转（Round-Robin）**
4. **优先级调度（Priority Scheduling）**
5. **实时调度（Real-Time Schedulers）**

结合多线程之间的数据共享同步、缓存亲和性等，调度策略变得更加复杂

- 问题复杂性无止境

12.4.3 策略优化

性能优化策略

应对性能瓶颈的优化策略

1. 批处理（batching）
2. 等待（dallying）
3. 推测（seculation）

① 批处理

通过批量完成任务，从而减少平均时延

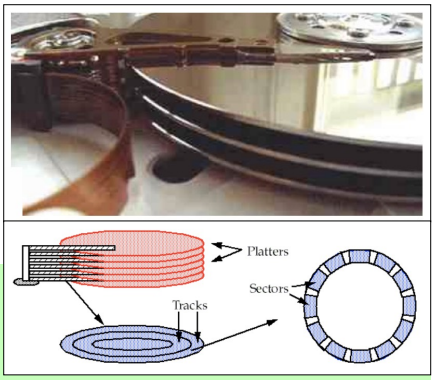
- 假设每个操作时延由f 和v 组成，f 操作可以共用
  - （设想无人车往返菜鸟取n个快递，f：路程，v：取）
  - 批处理前：n个操作的时延： $n \times (f+v)$
  - 批处理后：n个操作的时延： $f + (n \times v)$

① 批处理

系统案例：假如有10个进程申请读写硬盘

1000÷（7200/60）÷2

- 每次读写硬盘，旋转半周4ms，其他8ms
- 10*（4+8）=120 ms
- 合并可共用的(大)固定时延：
  - 旋转1周2*4 = 8ms
  - 总共时间：8+10*8 = 88 ms



- 更多设计机会：
  - a. 两次写操作有可能合并（覆盖）
  - b. 重排序，优化寻道操作，减少时延
  - ✷ 批处理的机会能否创造？（思考现实案例）

② 等待

推迟操作，等待该操作过时或创造批处理的机会

系统案例：写硬盘

- 在可以访问的时候推迟访问，等待批处理的机会
- 等待写过时又叫“写吸收（write absorption）”

系统案例：临时文件

- 最后的批处理写入

有没有风险？？
有什么风险？？

分析

对时延的影响分析

- 分情况思考：
  - I/O密集爆发
  - 均匀分布
  - 稀疏分布

等待面临how-long的权衡与挑战

- 与系统和应用相关，这个策略的优化效果是不确定的

③ 推测

在有依赖时，利用空闲资源，提前推测执行，降低时延

系统案例：CPU流水线

- 假如下一条指令是jg、je、jl等跳转指令
  1. 放弃并发？流水线的其他部分空转
  2. 冒险尝试？可能会预测错误
- 分支预测是CPU的重要技术
  - Intel分支预测正确率 > 90%　（试估算性能提升）

进一步思考：程序员写什么样的代码有利于预测？

③ 推测

其他系统案例：

1. 文件编辑器：预取文件块
2. CPU缓存：预取数据和指令

推测面临的挑战

1. **如何提升预测的命中率?**
   - 分支预测
   - 值预测?
2. **如果预测失败，如何回退（undo）？**

设计引入的新挑战

批处理、等待、推测有代价吗?

- 引入脆弱性，增加复杂性
- 习题

多选题 1分 设置

交叠可以减少每个请求的时延

只要资源的利用率不是100%，那么队列的长度就存在上限。

A **正确**

B **错误**

C **正确**

D **错误**

提交

习题： Toastac-25

- Louis P. Hacker在一次拍卖会上以14.99美元的价格购买了二手的Therac-25（曾发生过多起事故的放疗仪）。稍作修改后，他把Therac-25连接到家庭网络，作为可以通过计算机控制的高速烤面包机，可以在不到2毫秒的时间内烤好一片面包。
- 他使用RPC来控制Toastac-25（改造后的Therac-25），每一个烤面包的请求会在服务器上启动一个新线程，烤好面包后返回一个确认信息（或返回错误代码“故障54”），然后退出。每个服务器线程运行以下过程：

```
procedure SERVER()
    ACQUIRE(message_buffer_lock)
    DECODE(message)
    ACQUIRE(accelerator_buffer_lock)
    RELEASE(message_buffer_lock)
    COOK_TOAST()
    ACQUIRE(message_buffer_—lock)
    message<-"ack"
    SEND (message)
    RELEASE(accelerator_buffer_lock)
    RELEASE(message_buffer_lock)
```


习题： Toastac-25

问题1 ： 烤面包机第一次大量使用时就停止烤面包了，出了什么问题呢?

A. 两个服务器线程可能出现死锁，因为一个持有message_buffer_lock并且想要获取accelerator_buffer_lock，而另一个持有accelerator_buffer_lock并且想要获取message_buffer_lock。

B. 两个服务器线程可能出现死锁，因为其中一个持有accelerator_buffer_lock和message_buffer_lock。

C. Toastac-25发生死锁是因为COOK_TOAST不是一个原子操作。

D. 不充分的锁定会导致线程之间交错不恰当。

习题： Toastac-25

在Louis修复了多线程服务器之后，Toastac得到了比以往更多的使用。然而，当Toastac同时有许多请求（即有许多线程）时，系统性能严重下降，远远超出了Louis的预期。性能分析表明，锁的竞争并不是问题所在。

问题2： 可能是哪里出了问题?

A. Toastac系统花费所有时间用于在线程之间切换上下文

B. Toastac系统花费所有时间用于等待请求到达

C. Toastac系统发热，导致烤面包的时间更长

D. Toastac系统花费所有时间用于释放锁

习题： Toastac-25

问题3：

- 升级成超级计算机可以解决这个问题，但为时已晚——Louis痴迷于性能，他从RPC切换到了一种异步协议，该协议在一次请求之后的2毫秒内收到其他请求时，会将它们分组成单个信息。他注意到在他的网络上，传输时间非常长，这种方法加快了某些工作负载的速度，但没有加快其他工作负载。请描述一个加快的工作负载和一个没有加快的工作负载（示例：每10毫秒一个请求）。

问题4：

- 你作为一个设计工程顾问，需要对Louis从RPC转向异步客户端/服务端的决定进行评价，你如何看待他的做法？注意，Toastac有时会出现"故障54"而无法正常烤面包。

12.5 性能优化案例
硬盘I/O瓶颈优化

回顾：硬盘I/O

背景知识：

- 硬盘访问时间取决于：
  1. 磁头移动 → 寻道时延（毫秒）
  2. 磁盘转动 → 选择时延（毫秒）
     - 3600-4500-5400-7200-10000 （发展缓慢）
  3. 传输数据 → 总线传输时延 = 传输量/总线带宽
- 假设：选择时延8毫秒/转，其他时延8毫秒/次

硬盘参数示例

Table 1 Drive specifications summary for 3TB, 2TB, 1.5TB, 1TB and 750GB models

Drive Specification*	ST3000DM001 ST2000DM001	ST1500DM003	ST1000DM003 ST750DM003
Formatted capacity (512 bytes/sector)**	3000GB (3TB); 2000GB (2TB)	1500GB (1.5TB)	1000GB (1TB); 750GB
Guaranteed sectors	5,860,533,168 3,907,029,168	2,930,277,168	1,953,525,168; 1,465,149,168
Heads	6	4	2
Disks	3	2	1
Bytes per sector (4K physical emulated at 512-byte sectors)	4096	4096	4096
Default sectors per track	63	63	63
Default read/write heads	16	16	16
Default cylinders	16,383	16,383	16,383
Recording density (max)	1807kFCI	1807kFCI	1807kFCI
Track density (avg)	352ktracks/in	352ktracks/in	352ktracks/in
Areal density (avg)	625Gb/in^2	625Gb/in^2	625Gb/in^2
Spindle speed	7200 RPM	7200 RPM	7200 RPM
Internal data transfer rate (max)	2147Mb/s	2147Mb/s	2147 Mb/s
Average data rate, read/write (MB/s)	156MB/s	156MB/s	156MB/s
Maximum sustained data rate, OD read (MB/s)	210MB/s	210MB/s	210MB/s
I/O data-transfer rate (max)	600MB/s	600MB/s	600 MB/s
Cache buffer	64MB	64MB	64MB
Average seek, read (typical) Average seek, write (typical)	<8.5ms typical <9.5ms typical	<8.5ms (read) <9.5ms (write)	<8.5ms (read) <9.5ms (write)
Non-recoverable read errors	1 per 10^{14} bits read	1 per 10^{14} bits read	1 per 10^{14} bits read
Weight (typical)	620g / 1.36 lb	535g / 1.18 lb	400g / 0.88 lb
Average latency	4.16ms	4.16ms	4.16ms
Power-on to ready (max)	<17.0s	<17.0s	<10.0s
Standby to ready (max)	<17.0s	<17.0s	<10.0s

吞吐率问题

吞吐率上限：

- 理论：读盘 = (容量÷磁头数÷16383) × (转速/60) ≈ 3.75GB/s
- 实际： SATA（串行ATA）=（150~600）MB/s

瓶颈在哪里？

I/O瓶颈

吞吐率与一次访问的数据量有关

- 4k字节总吞吐率：4k ÷ 12毫秒 → 300k字节/秒
- 远小于最大吞吐率

假设不能减少时延，如何隐藏时延?

程序示例

程序功能：数值计算 （读 →算 → 写）

- 假设：未采用优化策略
- 参数：每个track 1.5M 字节，转速7200/分
- 耗时：每个循环约25ms = 读12+算1+写12

```
1    in ← OPEN ("in", READ)              // open "in" for reading
2    out ← OPEN ("out", WRITE)           // open "out" for reading
3
4    while not ENDOFFILE (in) do
5        block ← READ (in, 4096)         // read 4 kilobyte block from in
6        block ← COMPUTE (block)         // compute for 1 millisecond
7        WRITE (out, block, 4096)        // write 4 kilobyte block to out
8    CLOSE (in)
9    CLOSE (out)
```

如何优化?

1. **基于工作负载?**
   - 无重复访问，缓存技术无效
2. **并行?**
   - 考虑单个硬盘，没有并行的机会
3. **交叠?**
   - 计算和I/O是不同部件，有一定可能

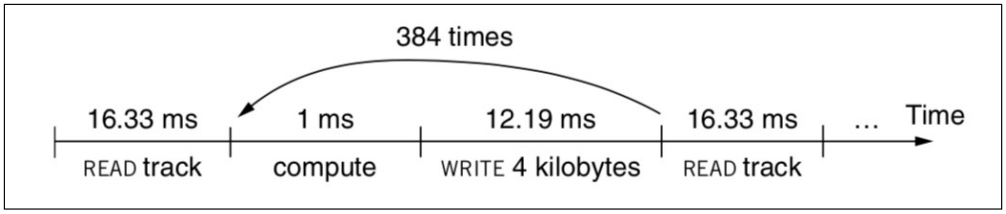
3种结构优化方法

先考虑策略优化 → **3种策略优化方法**

策略优化-1

推测（speculation）？假如成功：

- 改进文件系统：文件连续分布，每次读取整个track
- 读取1个track：12ms+4ms=16ms
- 如读到1.5M均有用，则可供384次循环读。384次计算与写耗时：5000ms。
  - 每循环约13ms。省了多少？48%

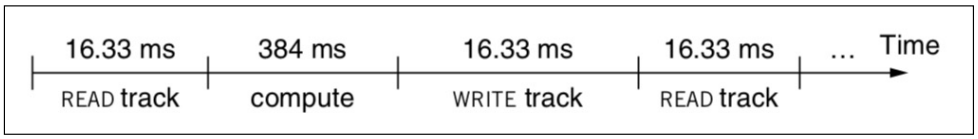


策略优化-2

批处理：

- 384次写合并
- 优化后，平均每次循环的时间为1ms

批处理的机会通过等待来创造



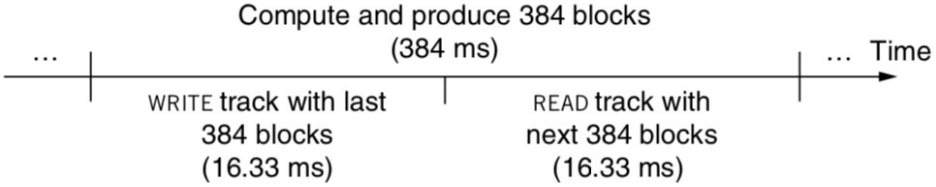
策略优化-3

推测：再贪心一点

- 如果预取下一个track
- 读写被计算时间完全覆盖
- 计算成为瓶颈

优化是否需要其他系统的配合？创造合适条件？

以上优化依赖于：文件顺序存储，计算是顺序访问文件的



如果访问不是顺序的怎么办？文件系统批处理写入时将其重排序（reorder）

其他优化

RAID：多盘并行、读写交叠

新技术：闪存（Flash Disk）

- （设计经验1）

思考：前述的优化是免费的午餐吗？

不及时存盘，系统崩溃怎么办？

1. 可靠性：数据丢失
2. 原子性：不可分隔的任务完成一半
3. 一致性：数据处于错误状态

后续课的内容：可靠性、原子性、一致性

设计经验

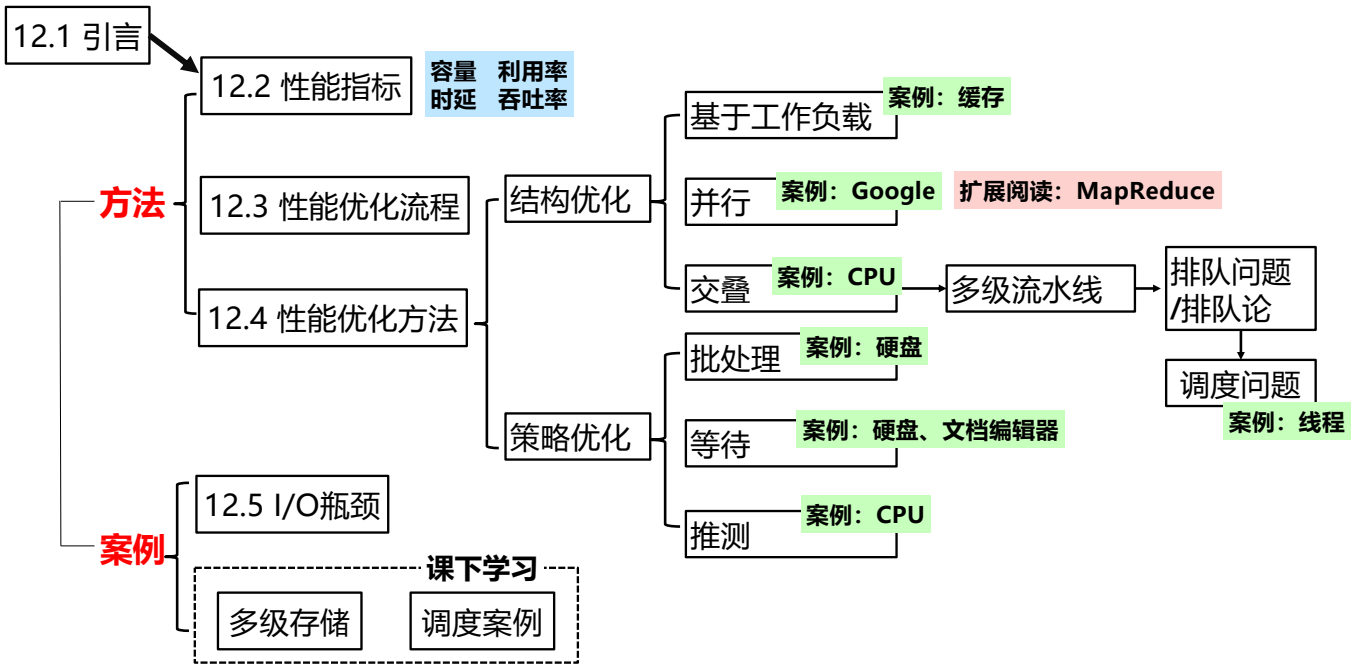
迟疑时，诉诸暴力

优化高频用例

时延难以减少？隐藏它

将实现从策略中分离

总结



重要术语中英文对照

瓶颈（bottleneck）

容量（capacity）

利用率（utilizition）

延迟（latency）

吞吐率/吞吐量（throughout）

高速缓存（cache）

并发（concurrency）

交叠 （overlap/interleaving）

交叠 （overlap/interleaving）

批处理（batching）

等待（dallying）

推测（speculation）

直写（write-through）

易失（volatile）

写吸收（write absorption）

回退（undo）

文献阅读与作业

- 论文MapReduce（第4、7节可略过）
  - 论文发表在两年一次的USENIX OSDI '04上。
  - 阅读1-3节，能够理解和解释Figure 1 (the "Execution overview")。
  - 阅读5-6节，了解其在实际中的性能。尝试回答这个初级问题：
    - straggler是如何影响性能的?

文献阅读与作业

阅读论文时，请思考以下问题

1. MapReduce的编程模式是受限的，收益值得接受这种限制吗?
2. MapReduce能处理哪些错误，怎么处理的?

作业：请回答以下问题

1. 工程师提出MapReduce的编程模型和实现，他们的性能目标是什么?
2. Google是怎么通过实现去满足这些目标的?
3. MapReduce为什么选择这样实现，而没有走其他技术道路?

本章相关的参考文献

- LAMPSON B W. Hints for computer system design[J]. ACM SIGOPS Operating Systems Review, 1983, 17(5): 33-48. DOI: 10.1145/800217.806614.
- DEAN J, GHEMAWAT S. MapReduce: Simplified Data Processing on Large Clusters[C] //Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, 2004,51(1) : 107-113. DOI: 10.1145/1327452.1327492.

  Also in Communications of the ACM 2008, 51(1), 1–10.
- PATT Y N, GANGER G R. Metadata update performance in file systems[C] //Operating Systems Design and Implementation. USENIX Association, 1994: 5–15. DOI: 10.5555/1267638.1267643.
- JAIN R. The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling[M]. New York, USA: Wiley-Interscience, 1991.
- BRIN S, PAGE L. The Anatomy of a Large-Scale Hypertextual Web Search Engine[J]. Computer Networks and ISDN Systems, 1998, 30(1-7):107-117. DOI: 10.1016/S0169-7552(98)00110-X.

应对过载

- 过载不可避免
  - 短时间过载：排队
  - 长时间过载
    - 提高服务容量
    - 降低负载（shed load）

降低负载

- 有限缓冲区方法：缓冲区满 → 等待，将瓶颈倒推给开始 → 停止接收请求
  - 如果请求者等待（交互）输出，进入自管理
  - 停止还是推迟?
- 配额（quota）方法：如设置活动进程、打开文件等的上限
- 换出一些不能胜任的负载