学校代号	10532	学　　号	XXXXXX
分 类 号	TP391	密　　级	普通

湖南大學
HUNAN UNIVERSITY

博士学位论文

多核系统并发哈希研究

学位申请人姓名	XXX
培　养　单　位	信息科学与工程学院
导师姓名及职称	XX 教授
学　科　专　业	计算机科学与技术
研　究　方　向	多核体系结构、高性能计算
论文提交日期	二〇一 x 年 x 月 xx 日

学校代号: 10532

学　　号: XXXXXX

密　　级: 普通

湖南大学博士学位论文

多核系统并发哈希研究

学位申请人姓名: XXX

培　养　单　位: 信息科学与工程学院

导师姓名及职称: XX 教授

专　业　名　称: 计算机科学与技术

论文提交日期: 二〇一 x 年 x 月 xx 日

论文答辩日期: 二〇一 X 年 x 月 xx 日

答辩委员会主席: 待定

Concurrent Hashing on Multi-core Systems

By

XXX XX

M.S. (Hunan University)2013

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of engineering

in

Computer Science and Technology

in the

Graduate School

of

Hunan University

Supervisor

Professor XX XX

December, 2017

湖 南 大 学

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写的成果作品。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律后果由本人承担。

作者签名：_____　　签字日期：_____

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权湖南大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。本学位论文属于

□ 不保密　□ 保密（____ 年）

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

目　录

学位论文原创性声明和学位论文版权使用授权书 ····· **I**
插图索引 ····· **V**
附表索引 ····· **VII**
摘　要 ····· **2**
Abstract ····· **4**
第 1 章　绪　论 ····· **6**
1.1　多核系统概述 ····· 6
1.1.1　多核体系结构 ····· 6
1.1.2　多核系统的缓存一致性 ····· 10
1.2　课题研究背景及意义 ····· 12
1.2.1　并发哈希表的性能评估 ····· 12
1.2.2　并发哈希表的设计 ····· 13
1.2.3　哈希表与布隆过滤器 ····· 15
1.3　本文主要工作 ····· 16
1.4　本文组织结构 ····· 17
第 2 章　并发哈希表的相关研究 ····· **18**
2.1　哈希表概述 ····· 18
2.1.1　相关概念 ····· 18
2.1.2　哈希函数 ····· 18
2.1.3　哈希冲突处理 ····· 19
2.2　基于软件技术的同步方法研究 ····· 21
2.2.1　阻塞技术 ····· 21
2.2.2　屏障技术 ····· 23
2.2.3　无阻塞技术 ····· 23
2.3　NUMA 架构内存管理相关研究 ····· 24
2.3.1　线程与处理器内核的关联 ····· 24
2.3.2　NUMA 系统的非对称互连 ····· 25
2.4　事务内存相关研究 ····· 26
2.4.1　实现事务内存的相关技术 ····· 26
2.4.2　基于事务内存的并发数据结构 ····· 27
2.5　布隆过滤器 ····· 27

2.6 本章小结 28
第 3 章 基于多核系统的并发哈希表的评估与分析 30
3.1 实现并发哈希表的同步方法比较 30
3.2 典型的并发哈希表 31
3.2.1 缓存行哈希表 31
3.2.2 Cuckoo 哈希表 33
3.2.3 Hopscotch 哈希表 34
3.2.4 基于 RCU 机制的哈希表 35
3.2.5 基于 Intel TBB 的哈希表 35
3.3 统一的跨平台并发哈希表测试框架的设计 36
3.3.1 参数说明 36
3.3.2 测试逻辑 37
3.3.3 延迟测量工具 38
3.4 并发哈希表的评估与分析 38
3.4.1 测试平台与配置 39
3.4.2 线程扩展性 41
3.4.3 更新比重对性能的影响 45
3.4.4 缓存与主存 47
3.4.5 操作延迟 48
3.4.6 线程绑定方案的影响 50
3.4.7 同步 53
3.4.8 内存消耗 54
3.5 本章小结 56
第 4 章 基于硬件事务内存的缓存行哈希表的实现 57
4.1 事务内存的基本概念 57
4.2 Intel 事务同步扩展 58
4.2.1 Intel 硬件锁省略 58
4.2.2 Intel 限制性事务内存 59
4.2.3 RTM 的 Lemming 效应 60
4.3 软件辅助的硬件锁省略技术 62
4.3.1 软件辅助的锁删除方法 62
4.3.2 软件辅助的冲突检测方法 65
4.4 基于 HTM 的并发哈希表的设计 66
4.4.1 问题引入 66
4.4.2 缓存行哈希原型描述 68

4.4.3 基于 RTM 的锁方法描述 ····· 70
4.4.4 基于 HTM 的缓存行哈希表 ····· 72
4.5 性能评价 ····· 75
4.5.1 测试平台和参数设置 ····· 75
4.5.2 基于 HTM 的粗粒度锁实现与细粒度锁 CLHT-lb 的比较 ····· 75
4.5.3 不同的全局锁方案之间的比较 ····· 76
4.5.4 基于 HTM 的细粒度锁实现与传统细粒度方法的比较 ····· 78
4.5.5 影响 HTM 性能的因素分析 ····· 79
4.6 本章小结 ····· 80
第 5 章 并发 Cuckoo 过滤器的设计 ····· 82
5.1 布隆过滤器 ····· 83
5.1.1 基本原理 ····· 83
5.1.2 误判率估计 ····· 84
5.1.3 最优哈希函数个数 ····· 85
5.1.4 最优位数组长度 ····· 86
5.2 Cuckoo 过滤器的设计 ····· 87
5.2.1 Cuckoo 过滤器的数据结构 ····· 88
5.2.2 Cuckoo 过滤器的参数 ····· 90
5.3 并发 Cuckoo 过滤器 ····· 94
5.3.1 自旋锁 ····· 94
5.3.2 加锁与解锁 ····· 97
5.3.3 并发访问接口 ····· 99
5.4 性能优化 ····· 104
5.4.1 空间性能优化 ····· 104
5.4.2 并发性能优化 ····· 106
5.5 性能评估 ····· 110
5.5.1 实验配置和性能指标 ····· 110
5.5.2 空间效率和误判率 ····· 110
5.5.3 扩展性 ····· 111
5.5.4 更新比重对性能的影响 ····· 112
5.5.5 $Retries$ 设置对性能的影响 ····· 113
5.6 本章小结 ····· 114
总结与展望 ····· 115
参考文献 ····· 117

插图索引

图 1.1 SMP 架构多处理器的基本结构示意图 ····· 8
图 1.2 DSM 架构多处理器结构示意图 ····· 9

图 3.1 4 路组相联 Cuckoo 哈希表 ····· 34
图 3.2 Hopscotch 插入操作示例 ····· 35
图 3.3 CHTBench 测试流程图 ····· 37
图 3.4 AMD Opteron 内存结点拓扑结构 ····· 40
图 3.5 Intel Phi 7120p 体系结构 ····· 42
图 3.6 并发哈希表的吞吐量随线程数量的变化曲线，哈希表的密度为 50%，u 为 10%，i 为一百万。 ····· 43
图 3.7 数据服从 zipf 分布时吞吐量随线程变化情况 ····· 44
图 3.8 哈希表初始化元素个数一定的前提下，吞吐量随更新比重变化趋势 ···· 46
图 3.9 不同初始化大小对应的吞吐量抽样直方图 ····· 47
图 3.10 E5-2630 平台上 6 种操作对应的延迟随线程数量变化曲线 ····· 49
图 3.11 E5-2630 平台上使用平衡型线程绑定方案时线程与核的映射拓扑结构 · 50
图 3.12 三种线程绑定方式性能对比 ····· 51

图 4.1 错误示例 ····· 63
图 4.2 串行执行五个操作的哈希表 ····· 67
图 4.3 MCS 锁的工作原理 ····· 71
图 4.4 传统细粒度锁方法和基于 RTM 的粗粒度锁方法之间的性能比较 ····· 76
图 4.5 使用不同全局锁方法之间的性能比较 ····· 77
图 4.6 紧凑型线程绑定方案运行结果 ····· 77
图 4.7 使用/不使用 HTM 的细粒度锁方法之间的性能比较 ····· 78

图 5.1 布隆过滤器 ····· 83

图 5.2 Cuckoo 哈希表元素插入过程 ····· 89
图 5.3 （2，4）组相联 Cuckoo 过滤器 ····· 89
图 5.4 负载因子与指纹信息长度变化关系 ····· 92
图 5.5 哈希桶容量与空间效率的关系 ····· 94
图 5.6 CLH 锁结构 ····· 97
图 5.7 使用不同软件优化方法的 CCF 线程扩展性 ····· 112
图 5.8 并发 Cuckoo 过滤器的线程扩展性 ····· 112
图 5.9 不同 $retries$ 值之间的性能差异 ····· 113

附表索引

表 3.1 用于评估的五种并发哈希表实现 ····· 31
表 3.2 测试目标平台的硬件和系统特征 ····· 39
表 3.3 CLHT-lb 内存带宽随线程数量变化情况 ····· 42
表 3.4 平均缓存未命中数量随更新比重变化情况 ····· 46
表 3.5 E5-2630 平台上的内存使用情况 ····· 55

表 4.1 RTM 的中止状态定义 ····· 60
表 4.2 Intel PMU 收集的不同的软件优化方法的运行时数据 ····· 79
表 4.3 slr-scm-mcs 方案的中止率随线程变化情况 ····· 80
表 4.4 slr-scm-mcs 方案的中止率随更新比重变化情况 ····· 80

表 5.1 相关符号及其含义 ····· 90
表 5.2 HashTable 类的成员函数列表 ····· 100
表 5.3 空间效率、误判率和构造速率 ····· 111

摘 要

随着多处理器技术的日臻成熟以及集成到单个处理器上的处理器核心数量的日趋增加，计算机的运算能力的瓶颈不断被打破，同时这也为设计具有高可扩展性的并发数据结构以及开发基于多核架构的高性能软件系统提出了挑战。并发哈希表是一种重要的并发数据结构，因其处理元素的开销为常数时间的特性被广泛应用于多核架构的软件系统开发。在并发哈希表的设计、优化以及应用中，处理器的体系结构，缓存一致性协议，内存带宽，内存访问延迟以及多线程的同步机制都对其性能产生重大影响。本文针对基于多核系统的并发哈希表做了如下工作:

首先，设计了用于并发哈希表的测试、评估的统一测试框架 CHTBench。CHTBench 是目前第一个用于并发哈希表性能比较与评估的，能保证测试结果公平性与客观性的测试框架。它提供统一的测试接口，具有可配置的线程与核的映射关系，能够测试不同规模的数据集以及数据集中更新操作的比重等。此外，CHTBench 使用 $sspfd$ 进行延迟的测算，综合考察不同并发哈希表线程扩展性，查询和更新吞吐量等宏观指标。结合其它的工具，如 $likwid$ 对缓存命中率，内存带宽，跨内存节点通信开销等微观指标进行分析以及 $sspfd$ 对各种操作类型的延迟进行比较。使用 CHTBench 可以为并发哈希表的评估和比较提供相对公平的测试环境。

第二，基于 CHTBench 测试框架对现有的几种具有代表性的并发哈希表在四个不同的多核系统上进行了深入的剖析。对并发哈希表的评估选用的性能指标涵盖宏观和微观两个层面，包括吞吐量的线程扩展性，延迟，分层内存结构对性能的影响，不同线程与核映射方式之间的性能差异，同步机制的性能评估以及内存消耗。通过对上述指标的分析，找出现有并发哈希表设计方法中存在的问题以及可能的性能瓶颈，分析总结了 8 条设计、优化并发哈希表的最佳实践原则，为将来进一步设计基于多核系统的、具有高可扩展性的并发哈希表奠定理论和实践基础。根据对比现有文献以及相关研究工作，我们对于并发哈希表的评估所用的方法和涉及的评估指标是迄今为止最全面的。

第三，使用 Intel 限制性事务内存 (RTM) 实现了基于硬件事务内存的缓存行哈希表。使用全局锁实现了基于 RTM 的缓存行哈希表。实验评估结果表明，当数据集的规模大于最后一级缓存容量时，基于 RTM 的缓存行哈希表的性能是使用传统的细粒度锁方法实现的缓存行哈希表的 120%。使用硬件事务内存进行并发哈希表的设计真正做到了粗粒度锁方法的简便与细粒度锁方法的高性能的有机结合。同时，为了消除 Intel TSX 的 Lemming 效应对性能的影响，设计了两种软件辅助方法：软件辅助的锁省略方法 (SLR) 和软件辅助的冲突管理方法 (SCM)。实

验结果表明，这两种方法对基于 RTM 的缓存行哈希表性能的提升比使用 Intel 官方推荐的 RTM Retry 机制更明显。

最后，使用不完整键 Cuckoo 哈希方法设计并实现了支持多线程并发的 Cuckoo 过滤器。这是迄今为止第一款支持多线程并发的过滤器实现，同时也是第一款基于硬件事务内存的过滤器。原始的布隆过滤器不支持删除操作，实现删除操作需要引入额外的空间开销，通过使用不完整键 Cuckoo 哈希方法，通过哈希函数生成指纹，利用 Cuckoo 哈希表多路组相连能够存储多个相同的指纹信息的特点实现了删除操作，不额外增加过滤器的空间开销。使用基于 Intel RTM 的读写锁实现了多线程并发的 Cuckoo 过滤器。实验评估表明，并发 Cuckoo 过滤器的初始化速度是使用单个线程初始化的 10 倍，查询操作的性能是单个线程的 38 倍，处理更新占 10% 的数据集的性能是单个线程的 11 倍。此外，还对 retry 的最大值如何选取以及使用不同软件优化方法的 Cuckoo 过滤器版本进行了线程扩展性的比较。

关键词：多核系统；缓存一致性；并发哈希表；布隆过滤器；同步；非一致性内存架构；硬件事务内存

Abstract

As the development of multi-processor techniques and the increasing of the number of cores integrated into a single CPU chip the bottleneck of computing power of the processor is constantly being broken. It also makes designing highly scalable concurrency data structures and developing high-performance software systems based on multi-core architectures more complicated. Concurrent hash table(CHT) is an important concurrent data structure and it is widely used to implement software systems on multi-core architectures as its queries run in amortized constant time. The hardware architecture, cache coherence protocol, memory bandwidth, memory access latency and multi-thread synchronization mechanism impact the designing, optimization and application of CHT significantly. The works of this paper including:

First, we present a framework, named CHTBench, which provides a fair testing environment and unified interface for the experiments by hiding the discrepancies of hardware platforms, synthesized workloads, concurrency models, and compiler configurations. In this way, we can guarantee the experimental results generated from our framework are fairly comparable between different CHTs. An open source library, $sspfd$, is integrated into CHTBench to measure the access latency of different kinds of operations. With CHTBench it's easy to compare the performance of CHTs. And it can also combine with other system tools such as $likwid$ to measure cache hit/miss rate, memory bandwidth overheads and cross-socket node communication overheads from micro perspective.

Second, we dissected 5 state of the art CHTs on CHTBench. The evaluations are explored from a wide range of perspectives including thread scalability, throughput, latency, memory hierarchy impact, low-level synchronization primitives, and memory usage. The inter-correlations between relevant metrics are also discussed when necessary. The experiments are conducted on four major hardware platforms including Intel Many Integrated Core(MIC) architecture and three representative Non-uniform Memory Architecture(NUMA) systems. We ported CHTs to the MIC platform, and to our knowledge, this is the first extensive study of concurrent hash tables on Intel MIC architecture. According to the experimental results, Œeight principles of best practices in designing and optimizing concurrent hash tables are summarized which lay the theoretical and practical foundation for further design of high scalability concurrent hash tables based on multi-core systems in the future.

Then, inspired by the fine-grained Cache Line Hash Table, we implemented a concur-

rent cache-line hash table with hardware transactional memory(HTM-CLHT). The size of a HTM-CLHT bucket is padding to the size of cache line(64 bytes in our testbed). The HTM-CLHT takes the whole hash table as the critical section which can provides optimistic concurrenty control by allowing threads to run in parallel with minimal interference. When running workloads which large than the capacity of LLC, the performance of HTM-CLHT is twenty percent better than using traditional fine-grained locks. HTM-CLHT achieves the goal that using a simple, coarse-grained locking method, obtaining high performance which matched with sophisticated synchronization methods such as fine-grained locking and non-blocking. In order to eliminate the Lemming effect of Intel TSX, we presented two software-assisted techniques, lock removal(SLR) and conflict management(SCM). Both of these methods improve the performance of the HTM-based concurrent hash table more significantly than the RTM Retry mechanism recommended by Intel.

At last, we presented a concurrent Cuckoo Filter based on partial-key Cuckoo Hashing. To our knowledge, this is the only concurrent filter so far, and its also the only one which based on hardware transactional memory(HTM). The standard Bloom Filter do not support delete elements from filter, while other extended versions of Bloom Filter support the deletion of filters with high space overheads. The structure of Cuckoo Hashing is in a set-associative way, each element can map to several slots. Taking this feature of Cuckoo hash table, we extract the fingerprints of the members of a set and store them in a hash table. And items with same fingerprint are fine, i.e. the filter can store several identical fingerprint. To delete an item from Cuckoo Filter, the fingerprint of this item is deleted from filter. If there is another item has the same fingerprint, we can still find this item with a copy of this fingerprint. This can delete an element very easy and straightforward without any additional space overheads. According to our experiment results, when running with read-only workloads, the max throughput is 38 times of the throughput of a single thread, and running workloads with ten percent update operations, the max throughput is 11 times of the throughput running workload with a single thread.

Key Words: Multi-core System; Cache Coherence; Concurrent Hash Table; Synchronization; Non-uniform Memory Architecture; Hardware Transactional Memory

第 1 章　绪　论

1.1　多核系统概述

20 世纪 90 年代，随着商业化微处理器的生产成本的降低，计算机处理器设计人员开始寻求性能强于单个微处理器的用于构建服务器和超级计算机的多处理器。伴随着单处理器上性能的增幅减少以及对计算机功耗的关注，促成了人们热衷于研究指令级并行 (ILP) 技术，这致使计算机体系结构进入新的时代——一个多处理器在低端到高端市场扮演主要角色的时代。

1.1.1　多核体系结构

多处理技术的重要性体现在以下几个方面:

- 2000 年至 2005 年期间，研究人员在寻找和利用更高的指令级并行期间发现，功耗和硅成本的增长速度远超性能增长的速度，更高的指令级并行理论意义大于实际意义。除了指令级并行之外，另一条为人熟知的可能比基础技术具有更高性能的方法是使用多处理 (Multiprocessing) 技术。
- 云计算和软件即服务 (saas) 对高端服务器的需求越来越大;
- 互联网上的海量数据刺激数据密集型应用的增长;
- 有观点认为提升桌面电脑性能相比之下不再那么重要 (至少在图形处理方面)，要么是因为当前的桌面电脑满足性能需求，要么是因为高强度的计算密集型和数据密集型应用可以通过云计算完成。
- 人们对于如何有效的使用多处理器的认识的加深。

处理器是指计算机系统中的中央处理单元 (CPU)。它由多级指令和数据缓存、指令译码器和不同类型的算术和逻辑运算单元构成。多处理器是指由多个紧密耦合的处理器构成的计算机系统，多个处理器的协调和使用受同一个操作系统控制，并通过共享地址空间共享内存[1]。

计算机系统为了增强性能，降低功耗以及更加有效的同时处理多任务，将两个或两个以上用于读取和执行程序指令的独立处理单元 (通常被称为核心) 集成到 CPU 芯片上，这种集成了多个计算核心的芯片被称为**片上多核处理器 (Chip Multi-core Processor)**[2]。每个核心都具有单独的一级缓存和执行单元，同一块处理器芯片上的所有核心共享二级缓存。这种设计意味着虽然处理器有一个相比之下容量更大的缓存池，但每个核心拥有访问速度更快的内存空间和算术/逻辑运算单元。因此，单个处理器可以在不同的核心上同时运行多个指令，这种处理方

式被称为芯片级多处理 (Chip-level Multiprocessing)。

更细一步的划分，一个核心可以支持多个线程，这种同时执行的线程被称为**同步多线程** (Simultaneous Multithreading，简称 SMT)，也叫同时多线程。尽管多个线程运行在相同的核心内，但线程之间是完全隔离开的。同步多线程是多线程的两个主要实现之一，另一个是时间多线程 (也称超线程)。在同步多线程中，多于一个线程的指令可以在任何指定的流水线阶段中同时执行。实现同步多线程技术在基本的处理器架构上进行修改：一是增加了在一个周期中从多个线程获取指令的能力；二是设置一个更大的寄存器文件用于保存来自多个线程的数据。核心支持的并发线程的数量可以由芯片设计者决定。Intel 超线程技术 (Hyper-threading) 就是 SMT 实现的一个典型技术[3]。最新款英特尔 Core vPro 处理器系列[4]、Core 处理器系列[5]、Core M 处理器系列和 Xeon 处理器系列[6] 都采用了英特尔超线程技术。常见模式是每个 CPU 核心支持两个并发线程，Sun 公司 2004 年推出的第一款 SPARC 架构的多核处理器 UltraSPARC T1 Niagara 每个核心支持 4 个同步线程[7]，后续推出的 Niagara2 每个核心支持 8 个同步线程。

为了充分利用具有 n 个处理器的多指令多数据（MIMD）多处理器的优势，系统中必须要有至少 n 个线程或进程。单个进程中的独立线程通常是由程序员标识出来或者通过操作系统创建。分配给线程的计算量（称为粒度）在考虑如何有效利用线程级并行性方面很重要，线程级并行与指令级并行本质区别在于线程级并行性由高层软件系统或程序员标识，并且线程由数百到数百万条可并行执行的指令组成。线程也可以利用数据级的并行性，但是它的开销可能高于单指令多数据（SIMD）处理器或 GPU[8]。线程的高开销意味着要充分的发挥并行性，它的粒度必须足够大。

共享内存多处理器 (Shared Memroy Multiprocessor) 是一种典型的多处理器系统架构。共享内存多处理器在智能移动终端、服务器上的普及带来了并发编程技术上的重大变化。随着支持多线程的芯片的成本的降低以及单处理器无法突破现有的性能瓶颈，配备多处理器的计算机设备会变得越来越普遍。系统中包含的处理器的数量不同，决定了处理器间的内存组织方式和互联策略的差异，因此，按照处理器的内存组织方式可以将现有的共享内存多处理器分为两类：第一类为**对称多处理器 (Symmetric (shared-memory) Multiprocessors，简称 SMPs)**，又称为集中式共享内存多处理器。这种结构的处理器特点是具有少量的处理器，通常为 8 个或者更少。对于具有如此小的处理器数量的多处理器，所有处理器可以共享一个单一的集中存储器，所有处理器都有相同的访问权限，“对称”因此得名。在每一块多核芯片上，核心之间的内存都采用共享的方式。当前连接的多核芯片的数量大于 1 时，会为每一块芯片分配单独的内存，此时的内存是分布式的。SMP 体系结构的处理器通常也称**统一的内存访问 (Uniform Memory Access，简称 UMA) 多处理器**，这是因为所有的处理器都具有相同的内存延迟。图 1.1所示为 SMP 体系

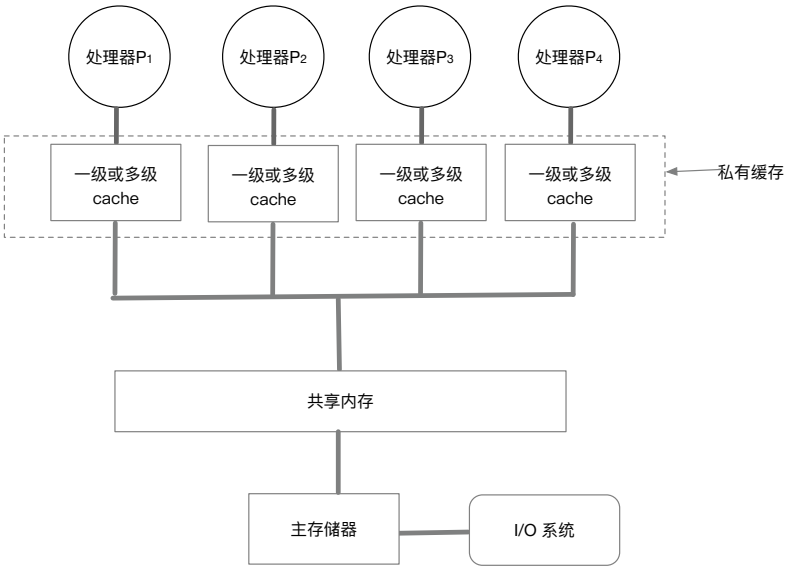


图 1.1　SMP 架构多处理器的基本结构示意图

结构的多处理器结构图。处理器的缓存子系统共享相同的物理内存，通常情况下具有一级共享内存，每个处理器内的核心具有一级或多级的私有缓存。该架构的重要属性是，所有的处理器具有相同的内存访问时间跟延迟。所有的处理器都必须通过一条总线实现同步与内存访问，因此，统一内存访问架构的多处理器系统的最大性能瓶颈是内存。当系统的处理器个数大于 32 时，因为总线争用将十分激烈，严重影响到多核系统性能。因此，使用单总线连接处理器和内存模块的方式不可取。

当系统内的处理器数量较多时，使用 SMP 架构会对系统性能造成影响，因此处理器数量较多的系统通常采用第二种架构：**分布式共享内存 (Distributed Shared Memory，简称 DSM)** 架构。DSM 是与 SMP 相对的一种体系结构，这种架构的多处理器共享逻辑地址空间，但是物理内存是分布式的。图 1.2所示为 DSM 体系结构多处理器的结构简图。为了支持更大的处理器数量，系统的内存必须是分布式的。否则，内存系统无法在保持内存访问延迟较低的前提下满足大量处理器的内存带宽需求。随着处理器性能的快速增加以及随之增长的内存带宽的需求，分布式内存要求的多处理器的规模将持续缩小。将内存分布在不同的结点上不仅增加了内存带宽，同时也满足了处理器访问内存低延迟的要求。由于访问时间取决于数据在本地内存结点还是远程内存结点上，因此，DSM 多处理器通常也被称为**非一致内存体系结构 (Non Uniform Memory Architecture，简称 NUMA)** 多处理器。DSM 的主要缺点是处理器之间的数据传输更加复杂，若要充分利用分布式存储器提供的更高的内存带宽需要花费更多的精力进行软件设计。

在 SMP 和 DSM 两种体系结构的多处理器中，线程之间的通信都是通过共享地址空间进行的，这意味着只要处理器具有必要的访问权限，任何处理器都可以在任何内存位置上进行内存引用。与 SMP 和 DSM 相关的术语**共享内存**是指地址

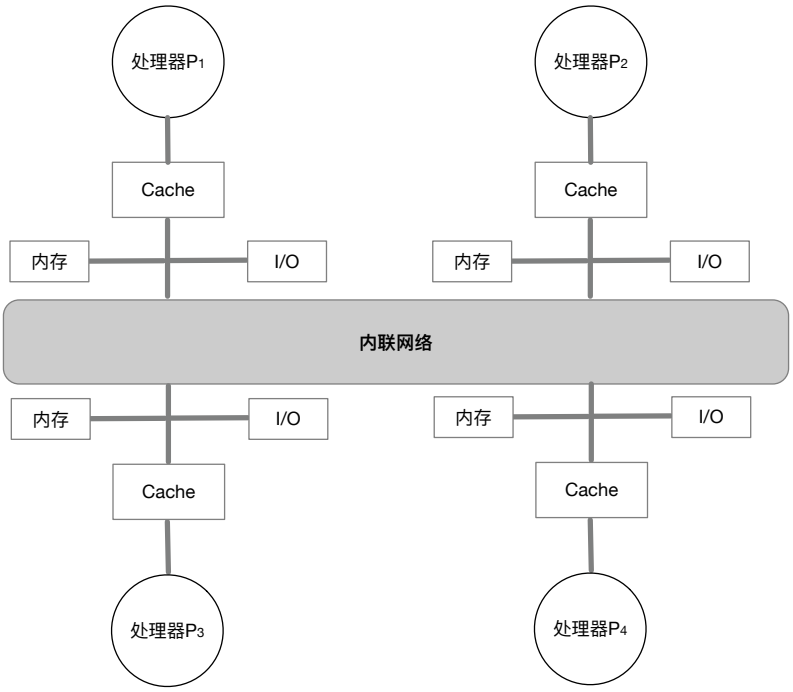


图 1.2　DSM 架构多处理器结构示意图

空间是共享的。

负载均衡是多处理器系统的一个显著特点。如果内核上的进程分布不均衡，即便是系统中的处理器数量再多，对于性能的提升也是微乎其微的。对称多处理器系统实现负载均衡有两种途径：第一种方法是将“准备就绪”的进程插入到一个可以共用进程队列中，每个处理器上的调度器都能访问这个进程队列。当某个处理器中的调度器被激活时，它将从“准备就绪”的进程队列中选取一个进程进入处理器进行运算。对称多处理系统通过使用共用的“准备就绪”的进程队列实现负载自动均衡。当某个处理器为空闲状态时，它的调度器会从队列中选取一个进程，并开始在该处理器上运行。这种方法能够实现处理器负载的自动均衡，但是共用的进程队列实现较为困难。

因此，为 SMP 设计的现代操作系统通常为每个处理器分配一个进程队列。这种操作系统使用显示的负载均衡机制，通过这个机制，过载的处理器上的等待列表中的进程会被移动到另外的负载较少的处理器的进程队列中。例如，SMP Linux 系统每间隔 200 毫秒就激活一次负载均衡机制[9]。这种方法也存在一个问题，当每个核心都具有私有缓存时，将进程迁移到不同的处理器上的代价高昂。因此一些操作系统 (如 Linux) 提供一个系统调用来指定与处理器绑定的进程，这与处理器负载无关。

多核处理器和多处理器的区别与联系: 多核处理器是指一个 CPU 内包含若干个计算单元；而多处理器是指在计算机内存在多个相同的 CPU。以双核处理器为例，双核心的设置类似于在同一台计算机上安装两个独立的处理器，但是由于两个核心实际上位于同一块芯片上，它们之间的信息交换的速度要快于同一台计算

机上的两个独立的处理器之间的通信速度。多处理器的 CPU 可以是由多个普通的只具有一个核心的 CPU 构成，也可以是由多个多核 CPU 构成。

在本文的研究中，使用的五个实验平台的其中四个是基于 NUMA 架构的多处理器系统，另外一个是 Intel 的基于 Many Integrated Core(MIC) 架构的众核处理器。

1.1.2 多核系统的缓存一致性

多核系统的体系结构决定了它的缓存系统的复杂性。在大多数现代多核系统上，每个处理器核心都具有独立的一级私有缓存，同一个 CPU 内的多个核心共享二级缓存。当系统只有一个核心工作时，数据的一致性是能够保障的，一旦系统上有多个核心同时工作，就会出现缓存一致性问题。试想当某个 CPU c_1 缓存行中对应的内存内容被另一个 CPU c_2 做了修改，而 c_1 没有收到任何关于该内容被修改的通知，那么 c_1 在下一次读取该数据的时候获取的是一个错误的值，从而导致运算结果偏离预期。这种后果往往是灾难性的。因此，在拥有多组缓存的情况下，保持这些缓存数据的同步至关重要，这需要设计一种各个缓存都能遵守的协议来实现同步。

注意到造成缓存不一致的根源问题在于系统拥有多组缓存，而不是因为系统拥有多个处理器核心。一种直观的解决方案是让多个处理器核心共用一组缓存，即只设计一块一级缓存。在任何一个指令周期内，只有一个处理器核心能够通过一级缓存进行内存操作，完成其相应的指令。这在逻辑上没有任何问题，唯一的问题是效率太低。所有的核心都需要排队等待使用一级缓存。最终问题还是要回到使用多组缓存上来，但要设法让多组缓存之间的行为看上去就像只有一组缓存在工作一样。缓存一致性协议便在这样的背景下应运而生。

缓存一致性协议有多种，常见的计算机设备上使用的是基于“窥探”（snooping）的协议。它的基本思想是所有内存传输都发生在一条共享的总线上，而这条总线对所有的处理器都是可见的，缓存不仅在做内存传输时才和总线打交到，而是不停地在窥探总线上发生的数据交换，跟踪其它缓存在做什么。这种协议的好处是延迟低。但是这种总线型的设计方式不适合大规模的多核处理器系统。在大规模的多处理器系统上使用的是“基于目录的”（directory-based）协议。使用这种机制的缓存一致性协议的弊端是延迟较高，但是具有较好的可扩展性。

现代的多核处理器使用的缓存一致性协议为 MESI 协议及其衍生协议。MESI 协议得名于该协议约定的四种缓存行状态：已修改（Modified），独占（Exclusive），共享（Shared）和失效（Invalid）的首字母。为了叙述的方便起见，本文将以相反的顺序对这四种状态进行描述：

- I 是指该缓存行要么已经被替换出了缓存，要么是该缓存行上存储的内容已经过时。为了达到缓存的目的，处理器读取缓存行数据时，被标记为 I 的缓存行会被忽略。也就是说，一旦缓存行被标记为失效，等同于该缓存

行未被加载进缓存内。

- S 表明该缓存行存储的内容是与主存内容一致的一份副本，处于 S 状态的缓存行只能被读取，不能进行写入。多组缓存可以同时拥有来自同一内存地址的共享缓存行。
- E 状态的缓存行与 S 状态的一样，也是和主存内容一致的一份副本。区别在于如果一个处理器持有了某个 E 状态的缓存行，那么其它的处理器就不能同时持有它，这就是名字“独占”的由来。这意味着如果其它处理器原本也持有同一缓存行，则该处理器持有的缓存行马上会转变成 I 状态。
- M 状态的缓存行，属于脏（dirty）缓存行，表明它已经被持有它的处理器修改了。如果一个缓存行处于 M 状态，那么它在其它处理器缓存中的副本马上就会转变成 I 状态，这个规律与 E 状态对副本的处理一样。此外，已修改的缓存行如果被替换出缓存或被标记为 I 状态，那么先要将它的内容回写到主存中。

将上述四种状态和单核系统中回写模式的缓存对比，发现状态 I、S 和 M 都有对应的概念：失效/未载入、干净以及脏的缓存行。这里唯一没有对应的是 E 状态，代表某个处理器的独占访问权限。这个状态解决了“在对某块内存进行修改之前通知其它处理器”的问题。当处理器想对某个缓存行进行写入时，如果它没有独占权，它必须先发送一条请求独占的请求给总线，然后广播其它处理器，使它们拥有的同一缓存行的副本失效。反之，如果其它处理器想读取这个缓存行，M 或者 E 状态的缓存行必须先回到 S 状态。如果是 M 状态的缓存行，还需要先将更新的内容回写到主存中。

在进行多核系统的软件开发时，需要理解两点：

第一，在多核系统中，读取某个缓存行，实际上会牵涉到和其他处理器的通讯，并且可能导致它们发生内存传输。写某个缓存行需要多个步骤：在写入数据之前，首先要获得独占权，以及所请求的缓存行的当前内容的拷贝。

第二，尽管系统为了保障缓存一致性做了许多复杂的工作，但是最终的结果还是能够保证的。它遵循 MESI 定律[10]——在所有的脏缓存行（M 状态）被回写后，任意缓存级别的所有缓存行中的内容，和它们对应的内存中的内容一致。此外，在任意时刻，当某个位置的内存被一个处理器加载入独占缓存行时（E 状态），那它就不会再出现在其他任何处理器的缓存中。

在原生的 MESI 协议的基础上，衍生出了 MOSEI，MESIF 等变种。MOSEI 协议的“O”（Owned）状态与 E 状态类似，也是保证缓存一致性的手段，但它直接共享脏的缓存行的内容，而不需要先把脏缓存行回写到内存中。这种协议用于 AMD Opteron 处理器上。MESIF 协议扩展了一个“F”（Forward）状态，它指定某个处理器专门处理针对某个缓存行的读操作。当多个处理器同时拥有某个 S 状态的缓存行时，只有被指定的那个处理器（即对应的缓存行为 F 状态）才能对读操作

做出回应。这种设计可以降低总线的数据流量。Intel 的大部分多核处理器采用 MESIF 协议。

1.2 课题研究背景及意义

并发哈希表 (Concurrent Hash Table，CHT) 是一种允许在同一时刻有多个读者或写者线程访问共享对象的哈希表。其提供与串行哈希表一样的访问接口，但是 CHT 能够更有效的发挥多核处理器的性能。在并发编程模型里基于锁和无锁是两种常用的同步控制方式，用于确保多个线程有序的对内存进行访问。为了保证线程安全性，基于锁的并发哈希表对临界区进行加锁操作。锁的实现有粗、细两种不同的粒度。粗粒度锁在实现上相对简单，但是采用粗粒度锁往往临界区特别长（极端情况是使用全局锁），增加发生冲突的机率，阻碍对计算机资源的高效利用。使用细粒度锁的好处是允许多个线程对数据的不同区段进行并发的读/写，有效的降低了发生数据冲突的概率。锁的粒度越细，越有利于提升整体性能，但是在实现和正确性验证上也需要耗费更多的精力。另一种相对的并发编程范式是无锁。无锁化编程是用计算机原语来代替显示锁的一种并发编程范式。此外还有使用非阻塞方法实现的并发哈希表[11–13]。

1.2.1 并发哈希表的性能评估

对于并发哈希算法性能的评估主要集中在：吞吐量、哈希表的线程扩展性、操作的延迟、内存使用情况、哈希表的空间利用率以及能耗等方面。吞吐量是指在单位时间内哈希表执行插入、删除、查询操作的总量，单位为 Mops/s 或者 ops/ms。线程扩展性是指多核系统上的应用的性能随着参与运算的核心的数量的增加而保持不减的能力。它是用来衡量多核软件系统性能的一个重要指标。应用程序的线程扩展性是指随着程序启用的物理线程的数量的增加，应用的吞吐量保持不减的能力。理想情况下程序的性能应该与物理线程的数量呈线性关系。延迟是指平均每执行一次哈希表操作需要耗费的 CPU 时钟周期数。内存使用情况是指处理同等规模的数据是需要消耗的内存。哈希表的空间利用率是指哈希表在不进行扩容操作的前提下，性能达到饱和时，哈希表内实体的数量与哈希表存储能力的比值，在有些文献中哈希表的空间利用率也被称作负载因子 (load factor)。

在现有的相关研究成果中[11,12,14–16]，对哈希表进行评估时所使用的哈希表的设计方法、优化方案、内存分配与管理、测试环境、测试数据、性能评估指标等方面都存在出入，并且对于微观层面的分析不够深入。设计方法上的差异体现在哈希函数的选取；采用什么样的冲突处理方法处理发生冲突的数据；采用哪种同步方法实现多个线程的并发访问。

Y.Liu 等人设计了一系列可动态调整哈希表大小的无阻塞并发哈希表[11]。他

们的算法是基于 Java 的，都使用 java.util.concurrent 包进行优化。他们对并发哈希表进行评估是通过将他们的算法与使用了相同的冲突处理技术和类似同步方法的 SplitOrder 哈希表[13]进行比较，测试在两种不同的多核处理器架构上 (X86 和 SPRAC) 展开。因为这类哈希表支持动态调整哈希表的大小，所以对于哈希表的空间利用率的评估意义不大。而且还可能受到来自 Java 虚拟机的影响。

Z.Metreveli 等人设计了一种缓存分区哈希表——CPHash[16]，CPHash 将查找/插入请求通过消息传递机制传输到指定的缓存分区内 (分区的大小设置为缓存行的大小)。这种设计的目的有两个：一是使用消息传递机制替代传统的锁；二是采用批处理的方式避免过于频繁的缓存行切换。所以对 CPHash 的性能评估侧重其与基于锁的同类哈希表的比较，突出消息传递机制取代锁方法的重要意义。

X.Li 等人在传统的 Cuckoo 哈希表的基础上实现了支持多读多写的并发 Cuckoo 哈希表[14]。由于其独特的组相连设计，它的空间利用率是现有的并发哈希表中最好的之一。Cuckoo 哈希表对锁方法、内存消耗、Cuckoo 查找路径都进行了优化，所以对于它的评估主要集中在吞吐量、内存消耗、哈希表的空间利用率等方面。

T.David 等人对并发数据结构的评估方法相对来说是目前现有工作中最为全面的[12]。他们通过对现有的一些并发数据结构的评估，总结了 4 条“异步并发 (ASCY)”编程范式。遵循这 4 条 ASCY 模式设计了缓存行哈希表 (CLHT)。缓存行哈希表的核心思想是对哈希表内的元素进行操作时尽量减少缓存行切换。所以，对 CLHT 的评估重点在哈希表的线程扩展性、操作延迟、吞吐量等方面，没有针对硬件平台的特点进行优化。

从上述几种并发哈希表的评估方法中可以看出，现有的并发哈希表的评估方法都紧密围绕其设计方法展开的，这样的评估方法缺乏客观性：放大了自身方法优于其它方法的点，而选择性的隐藏自身方法中的性能瓶颈与问题。这对并发哈希表的设计、优化、应用都是不利的。用户无法直观判断在其应用中选择哪种并发哈希表能够获得更高的性能。

因此，设计一个统一的测试框架，为并发哈希表的评估提供公平的测试环境，排除因硬件特性、线程分配、内存管理、数据分布、数据集的差别等因素的影响，充分从吞吐量、线程扩展性、内存消耗、同步方法、延迟、实现的难易程度等方面予以考察，确定设计和使用并发哈希表的最佳实践意义重大。

1.2.2 并发哈希表的设计

设计具有线程扩展性和充分利用处理器多核心优势的并发哈希表是一项富有挑战性的工作。即使是在特定的平台上实现一个达到预期性能要求的具有可扩展性的并发哈希表也是具有相当难度的课题，更不用说实现具有可移植的扩展性的并发哈希表。在某种体系结构上所采用的优化方法可能会在其他的体系结构上失

去作用[17,18]。比如，针对 NUMA 架构的多处理器使用的优化方法对于 SMP 架构的系统就没有任何意义[18]。针对某种硬件特性使用的优化方案在不支持这种硬件指令的系统上甚至都无法实现正常编译。比如使用 RTM 优化的并发哈希表在不支持 RTM 的机器上无法编译。再者，如果某个并发哈希表是针对特定类型的工作集而进行的优化，那么工作集轻微的变化将造成性能的不稳定或者极速下降。如使用 RCU 机制设计的并发哈希表，众所周知，RCU 机制适合用于处理读占绝大多数的数据集，但是对于数据集中包含有更新操作的场景，基于 RCU 的并发哈希表的线程扩展性很差，吞吐量低于运行在单处理器系统上达到的吞吐量[19]。

本文的目标之一是探究并发哈希表的性能瓶颈。确定这些瓶颈有助于实现并发哈希表的*可移植的扩展性*，即在不同的平台、工作负载和性能指标上都具有扩展性。乍一看，这个目标可能看起来很模糊，因为它提出了一个根本性的问题：在综合考虑数据结构、体系结构、性能指标和工作负载的前提条件下，我们可以期望获得具有哪种程度的扩展性?

研究表明，对于现代的多核处理器而言，由对共享数据的写操作引发的一致性流量是抑制并发软件可扩展性的最大障碍。然而，受并发数据结构固有语义的限制，并发数据结构中不得不存在一定比例的写操作，这些写操作是无法省略或者被其他操作替换的；通常在这些数据结构的串行版本中 (即不支持多个进程共享的同种数据结构) 也有相同的写操作。假设将这样的串行数据结构部署到多核系统上并且由多线程共享该数据结构，显然会得到错误的 (如非线性化[20]) 执行结果。然而，这种异步执行的程序的性能指明了如何对这些数据结构进行设计确保实现正确的同步。

在追求高性能并发数据结构的过程中，同步控制的软/硬件支持同样是一大挑战。在多核多处理器平台上，数据的一致性由硬件缓存一致性协议保障。缓存一致性协议维护读、写和原子指令 (比如 CAS 和 FAI) 之间的状态转换。比如，一致性协议可以选择不同的写和无效转换，如读后写，写后写，读无效，写无效。状态转换影响缓存间的流量，从而影响实际工作的可用缓存带宽。现代计算机处理器通常使用 MESI 缓存一致性协议和它的衍生协议，如 AMD 的 Opteron 处理器使用从 MESI 中演化出的 MOESI 协议，O(Owned) 是 MESI 中 S 和 M 的一个合体，表示缓存行被修改，和内存中的数据不一致，不过其它的核心可以有这份数据的拷贝，状态为 S；Intel 酷睿 i7 处理器使用从 MESI 中演化出的 MESIF 协议，F(Forward) 从 Share 中演化而来，缓存行如果处于 Forward 状态，它可以把数据直接传给其它核心的 Cache，而 Share 则不支持该功能。

硬件事务内存是基于缓存一致性协议的、用于进行同步控制的指令集扩展。当前的技术比较成熟，应用较为广泛的硬件事务内存主要是 Intel 的事务同步扩展指令集 (TSX)[21]，它包括硬件锁省略 (HLE) 和限制性事务内存 (RTM) 两套不同的指令集扩展。使用 Intel TSX 构造并发哈希表存在的最大问题就是如何避免

lemming 效应对事务内存性能的抑制作用。

而软件上的同步机制按照发生数据冲突时线程的行为分为阻塞、非阻塞两大类。阻塞是指在运行多线程 (多进程) 程序时，当某个线程 (进程) 在执行临界区时出现延迟，从而导致其它等待进入该临界区的线程 (进程) 全部延迟的情形；非阻塞是相对阻塞而言的概念。具体细分下来阻塞技术又包括锁方法、屏障技术，非阻塞技术又包括无锁 (lock-free)、无等待 (wait-free) 等方法。读-复制更新 (Read-copy Update，RCU) 是一套无锁化编程机制，由于其安全性高、性能稳定等原因被广泛用于 Linux 操作系统内核中[22–24]。

1.2.3 哈希表与布隆过滤器

在数据库、缓存、路由器和存储系统中，经常需要进行*近似集合成员关系查询*以确定某个元素是否属于某个集合的成员。布隆过滤器是一种空间效率很高的随机数据结构，它利用位数组很简洁的表示一个集合，并能判断一个元素是否属于这个集合。数据量较小的情况下，使用哈希表、集合、位数组等方法都能完美解决元素查询问题。但是在大数据的背景下对拥有巨量的元素信息的应用场景而言，如果按照常规的方法存储元素的完整信息，所需的存储空间将给存储系统造成巨大的负担。这种情况下布隆过滤器有了用武之地。布隆过滤器具有极高的空间利用率，用来解决海量数据的索引问题最合适。它的核心思想是：利用多个不同的哈希函数来解决集合元素查询时产生的冲突问题。布隆过滤器被广泛应用于概率路由表中减少内存空间的需求[25]；用于加速 IP 地址的最长前缀匹配[26]；用于提升网络状态管理和监控[27,28]；用于网络数据包组播转发信息的编码[29]，以及其他的网络应用[30]。

然而，布隆过滤器存在两个缺陷：一是由于元素信息使用位数组保存，所以不能支持元素的删除操作；二是因为哈希函数存在碰撞导致布隆过滤器在进行元素查询时存在一定的误报率。虽然在后续的研究成果当中，出现了支持元素删除操作的布隆过滤器的升级版本，比如 Counting 布隆过滤器[31]、d-left counting 布隆过滤器[32] 以及 quotient 过滤器[33]。但是这些过滤器要么是牺牲了空间效率，要么是牺牲了性能。为了保持相同的误判率，counting 布隆过滤器需要使用 3-4 倍的存储空间用于表示一个元素，d-left counting 布隆过滤器则需要 1.5 倍的存储空间，而 quotient 过滤器获得同等的空间效率它的查询性能要大打折扣。

此外，在当前多核处理器越来越受重视以及互联网数据呈爆炸式增长的背景下，目前的文献中还没有尝试设计支持多线程并发的布隆过滤器的研究。考虑到布隆过滤器的核心功能是通过哈希函数计算元素存储位置的索引值，这与构建并发哈希表存在相通之处，可以重用哈希表的同步方法、哈希方法、哈希表结构。另外，Cuckoo 哈希方法的多路组相连的特性能够确保哈希表在空间利用率达到 95% 左右时，仍然能够保证具有较好的性能。所以，设计基于 Cuckoo 哈希方法的

Cuckoo 过滤器是能解决当前的布隆过滤器方法中需要耗费大量存储空间实现删除操作或者需要牺牲查询性能换取对删除操作的支持的问题。

综上所述，研究并发哈希表在多核系统上的软/硬件同步方法，对于并发哈希表的优化与设计具有重要的理论意义。研究新的硬件同步机制对于构建并发数据结构以及提高现有并发数据结构的性能、简化并发数据结构的设计、实现新的并发数据结构具有重要的实用价值。

1.3 本文主要工作

本文主要着眼主流多核处理器架构上的并发哈希表的优化、设计与应用研究。

首先，设计了一个在多核系统架构上对并发哈希表进行测试的框架——CHTBench，CHTBench 为并发哈希表提供公平测试环境和统一的测试接口。使用 CHTBench 进行测试可以兼容不同的硬件平台、工作集、并发模型以及编译选项配置，在进行性能评估的比较时，排除上述因素的干扰，确保测试结果的公平性。对并发哈希表的线程扩展性、吞吐量、延迟、工作集的大小与读写比例、多核系统的内存分层结构、底层同步原语、内存消耗以及线程与核的映射关系等 8 个维度进行比较分析，在必要时还对存在关联的指标进行深入分析。实验平台涵盖了主流的 SMP 架构和 DSM 架构多核处理器以及众核架构 (MIC) 多处理器系统。其中，本文中将并发哈希表这种数据结构移植到 Xeon Phi 平台上进行同步性能评估的工作，该项工作是已知的最先将并发哈希表的研究扩展到 MIC 架构上的研究成果。此外，根据每一项评估指标的实验结果提出了与该指标相关的性能陷阱和优化误区，并提出了在设计并发哈希表时存在相互矛盾的性能指标的情况下，如何进行折衷与优化，以对并发哈希表的优化、设计提供指导意见。

第二，根据对并发哈希表的评估结果进行分析得到的最佳设计实践原则，设计了基于硬件事务内存的缓存行哈希表。原始的缓存行哈希表的并发操作使用的是细粒度锁实现的，使用链式法解决哈希冲突问题，缓存行哈希表具有高吞吐量，良好的线程扩展性和低操作延迟的特点。考虑到细粒度锁方法实现上和进行正确性验证（比如经典的 ABA 问题）上的复杂性，本文实现了基于硬件事务内缓存行哈希表。在处理规模大于 L3 缓存容量的数据集时，使用硬件事务内存实现的全局锁的缓存行哈希表的性能是使用传统细粒度锁版本的 120%。此外，为了降低 Lemming 效应对 Intel TSX 性能的副作用，提出了两种软件辅助方法：软件辅助的锁省略方法 (SLR) 和软件辅助的冲突管理方法 (SCM)。

最后，设计了支持多线程并发的 Cuckoo 过滤器。布隆过滤器是一种判断元素是否属于集合的数据结构，它允许一定的假阳性率换取存储空间的极大节省，是哈希方法的具体应用，但是标准的布隆过滤器有两个缺陷：不支持并发，不支持删除操作。本文使用基于 HTM 的读写锁实现了支持多线程并发的 Cuckoo 过滤

器，并以极低的开销实现了 Cuckoo 过滤器的删除操作。通过理论分析表明布隆过滤器的空间效率要优于 Cuckoo 过滤器，但是通过实验评估表明，这种差距产生的影响很微弱。相比于其他支持删除操作的布隆过滤器的变体而言，Cuckoo 过滤器的空间效率的优势十分突出。实验表明，并发的 Cuckoo 过滤器的构造速度是使用单个线程运行的 10 倍，进行元素查找的速度是使用单线程的 38 倍。

1.4 本文组织结构

本文分五个章节展开，各章内容安排如下：

第一章概述多核系统架构的基本概念以及多核处理器在计算机软、硬件技术发展中的重要意义，以此为基础综述 NUMA 架构多处理器的内存管理、基于多核架构的并发哈希表的研究的现状与缺陷、硬件事务内存对基于多核架构的并发数据结构的作用以及布隆过滤器的原理与缺陷等问题。指出当前基于多核系统架构的并发哈希表的优化、设计与应用中存在的问题及其研究的意义。最后归纳本文的主要工作以及论文组织结构。

第二章介绍与本文密切相关的并发哈希表的概念、设计方法与相关研究成果。介绍用于实现并发数据结构的同步方法，包括锁算法、屏障方法、非阻塞方法和硬件事务内存。然后对 NUMA 架构下的内存管理研究成果进行了概述。最后，对哈希表的一个重要应用——布隆过滤器的相关研究成果进行介绍。

第三章选取 5 种基于不同设计方法的具有代表性的并发哈希表进行评估。首先着重介绍了用于在多核系统上对并发哈希表进行测试的测试框架 CHTBench，然后对所选的 5 种并发哈希表的设计方法、数据结构特点以及同步原理进行了描述，随后根据这 5 种并发哈希表在 CHTBench 上的运行结果总结了 8 条设计、优化和应用并发哈希表的最佳实践原则，最后是对这一章的小结。

第四章根据前一章总结的 8 条最佳实践原则以及硬件事务内存在实现并发数据结构上的优势，设计了基于硬件事务内存的并发缓存行哈希表。然后介绍了用于克服 Lemming 效应对 Intel TSX 的软件辅助方法——软件辅助的锁省略技术和软件辅助的冲突管理技术。接下来是对基于硬件事务内存的缓存行哈希表的评估，评估同样是在 CHTBench 框架上展开。

第五章介绍支持删除操作的并发 Cuckoo 过滤器。首先对传统的布隆过滤器实现最优误判率、空间利用率的参数进行分析；然后对基于不完整键 Cuckoo 哈希方法的进行介绍，并对达到最优空间效率和误判率的相关参数进行了分析；接着描述并发 Cuckoo 过滤器的插入、查询和删除操作的实现；最后对 Cuckoo 过滤器的性能进行评估，并根据评估的结果提出了一些优化 Cuckoo 过滤器性能的方法。

第 2 章　并发哈希表的相关研究

2.1　哈希表概述

2.1.1　相关概念

哈希表、树、链表等都属于搜索数据结构。搜索数据结构由元素集合以及访问和操作这些元素的接口构成。如果搜索数据结构能够被多个处理器共享，我们则称该数据结构为**并发搜索数据结构** (CSDS)。哈希表（hash table），又名散列表，是一种应用广泛的搜索数据结构，它通过键值对（key-value）实现对关联数据的高效存取。键值对之间的映射关系称为**哈希函数**。一般的哈希表都提供了 *Insert(),Delete()* 和 *Lookup()* 三种操作的接口。哈希表的操作分为**读**操作和**写**操作，其中读操作指哈希表的查询操作，写操作包括在哈希表中插入和删除元素。哈希表和树型数据结构相比的最大的优势是哈希表的查询复杂度可以到常数级。存放值的存储空间称为**哈希桶**（bucket）或者**哈希槽**（slot）。哈希表中存放的元素的数量与哈希桶数量的比值称为**负载因子**（load factor)。哈希表使用哈希函数计算得到一个索引值，该索引值表明键对应的值在桶数组中的位置。关于哈希函数，有一个最理想的原则：将每一个 key 映射到单独的哈希桶内。但是，当数据集规模很大时，能够完美的实践上述原则的哈希函数并不存在。因此在实际的映射过程中往往会出现多个 key 对应相同的索引值，这种情况称为**碰撞**（collision）。既然碰撞难以避免，那么我们能做的就是在设计哈希表的时候尽量地选择好的哈希函数。一个好哈希函数的基本需求是输出的哈希值比较均匀。这样可以使发生碰撞的概率最小化，同时使得各个 bucket 中碰撞的条目比较平均。有国外的研究人员对已有的哈希函数做过比较[34]，结论是 MurmurHash3[35] 和 CityHash[36] 当时最出色的哈希函数。

2.1.2　哈希函数

哈希函数是将任意大小的数据转换成特定大小的函数，转换后的数据称为哈希值或哈希编码。哈希函数是实现哈希表和布隆过滤器的基础。根据其应用场景可以将哈希函数划分成加密和非加密两类。非加密的哈希函数通过数学运算将字符串转化成整型数输出。哈希函数的一个重要特点是它的输出能够在可能的输出域内尽量的保证均匀分布，尤其是当具有比较相近的输入时，这种特性尤为可贵。与加密哈希函数所不同的是，非加密哈希函数无法承担阻止攻击者利用碰撞进行攻击的任务。非加密哈希函数的运算速度要比加密哈希函数快。哈希表通常

采用非加密哈希函数建立元素与哈希表的对应关系。

Bob Jenkins 长期从事哈希函数的研究，他在 1997 年对哈希函数的研究中提出了被后来研究人员称为 Jenkins 的哈希函数[37]，在接下来的研究中，他对其研究成果进行了扩展，提出了名为 lookup2 和 lookup3 的哈希函数[38]。lookup3 哈希函数被有关学者认为是第一款“现代的”哈希函数。2008 年，Austin Appleby 发布了名为 MurmurHash 的哈希函数[35]。最新的 Murmurhash 具有两倍于 lookup3 的性能。由于其卓越的运算速度和统计特性，MurmurHash 得到广泛应用。2011 年，发布了两款高性能的哈希函数：一款是 Google 发布的 CityHash[36]；另一款是由 Jenkins 提出的 SpookyHash[39]。这两款哈希函数都是基于 MurmurHash，其性能的提升在很大程度上得益于更高的指令集并行。这两款哈希函数都有两倍于 MurmurHash 的处理速度，CityHash 的速度源于 SSE 4.2 中的 CRC32 指令。SpookyHash 产生 128 的输出结果，而 CityHash 的输出结果更为灵活，可生成 64 位、128 位和 256 位的哈希值。

2.1.3 哈希冲突处理

处理碰撞的方法大致可以分为两类：一类是**开放寻址法** (open-addressing)；一类是**开链法** (separate chaining)。

开放寻址法，所有元素都存放在哈希桶数组内，当需要在哈希表中插入新元素时，将对哈希桶进行扫描，从被直接映射到的哈希桶开始，按照某种探测序列进行扫描，直到找到空闲的哈希桶为止。当需要查找某个元素时，需要以同样的探测序列进行查找，直到找到所需的元素，或者最终发现元素不在表中为止。常用于开放寻址法的探测序列有线性探测、二次探测、双重哈希以及 Cuckoo 哈希。

线性探测。在线性探测中，进行探测的初始位置由 $h(k)$ 确定，地址增量为 i，从当前位置开始，若为空，则插入元素，若非空，则探测距离当前位置 i 个单位的位置继续，直到找到空闲的位置或者遍历完整个哈希表为止。为了确保能够遍历到整个哈希表，i 通常是哈希表容量 m 的相对质数。利用线性探测计算 key 的位置的公式如式 2.1所示:

$$h(k,i) = (h(k) + i)\ mod\ m \tag{2.1}$$

当地址增量 i 等于 1 时，探测的位置是连续的。线性探测的性能取决于查找 key 的位置时进行的探测次数，而探测次数又取决于哈希表的负载因子。哈希表的负载因子越高，查找一个 key 的位置时所需的探测次数就越多。Knuth 的论证表明[40]，使用线性探测法完成一次 key 的位置探测所需的平均探测次数约为 $1/2(1 + \frac{1}{1-\alpha})$。

二次探测与线性探测的相关性很高。二次探测计算 key 的位置的公式如

式 2.2所示：

$$h(k,i) = (h(k) + c_1 i + c_2 i^2) \; mod \; m \tag{2.2}$$

式中 c_1 和 c_2 均为常数。Heileman 等人的研究表明，当哈希表的大小超过缓存的容量时，使用线性探测的性能要好于使用二次探测[41]。虽然线性探测通常需要比二次探测执行更多的探测尝试，但是这些尝试具有更高的缓存命中率，从而在整个哈希表的大小超过了缓存容量时表现出更好的整体性能。另一方面，当哈希表的负载因子较高时，使用二次探测能够获得更好的性能，当然这只是相对而言的，当 α 接近 1 时，使用二次探测的性能也不理想。同样 Knuth 对使用二次探测完成一次 key 的位置的探测所需的平均探测次数进行了估计，约为 $1 - ln(1-\alpha) - \frac{\alpha}{2}$。

双重哈希，顾名思义就是使用两个哈希函数 h_1 和 h_2 计算需要探测的初始索引值。计算公式如式 2.3所示：

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \; mod \; m \tag{2.3}$$

由于 h_2 提供可变的地址增量，所以双重哈希很好的解决了线性探测和二次探测中的“聚集”问题。此外，多重碰撞的设置产生比线性或二次探测更为均匀的键值分布。双重哈希方法的性能也会随着哈希表负载因子的增加而下降。使用双重哈希完成一次查找所需的平均探测次数为 $-\frac{1}{\alpha}ln(1-\alpha)$。

Cuckoo 哈希的特性是：在哈希表中插入新的 key 时，会将原来存储于该位置上的 key 移动到其它的位置上去。Cuckoo 哈希一般设置两个哈希函数 $h_1(k)$ 和 $h_2(k)$。当插入新的 key 时，可以任意地选择在 $h_1(k)$ 和 $h_2(k)$ 位置上进行插入。如果插入的位置上已经存有了 k_1，则将 k 覆盖 k_1，然后将 k_1 存入到 $h_1(k)$ 和 $h_2(k)$ 中空闲的位置。如果 $h_1(k)$ 和 $h_2(k)$ 中都没有空闲位置，则继续上述过程，直到所有的 key 都找到了合适的位置或者执行替换的次数达到上限[42,43]。Pagh 等人的研究表明，要保持 Cuckoo 哈希的最优性能，需要确保 $\alpha \leqslant 0.5$[42]；Erlingsson 等人提出了一种广义的 Cuckoo 哈希技术[43]，能够使哈希表的负载因子达到 99% 时仍然具有较好的性能；Ross 等人通过应用单指令多数据流 (SIMD) 指令挖掘并行性并消除探测函数内的分支指令实现了对广义 Cuckoo 哈希方法的性能优化[44]。

上述的几种常用的开放寻址法中，性能都与哈希表的负载因子相关。负载因子越高，完成一次查找所需的探测次数就越多，性能也随之受到影响。实际上，即便是再好的哈希函数，在负载因子大于 0.7 之后性能会急剧下降。

另一类常用的冲突处理方法称为开链法。开链法引入额外的数据结构，比如链表，用于解决哈希冲突问题。每一个哈希桶都是独立的，所有经过哈希之后具有相同索引值的元素都放在同一个哈希桶中，这些元素通过链表进行管理。所以，使用开链法的哈希表能够存储的元素的个数大于哈希桶的数目，也就是说它的负

载因子可以大于1。对哈希表执行操作的时间等于找到相应的哈希桶的时间加上对链表进行操作的时间。虽说使用开链法的哈希表的负载因子可以大于1，并不意味着链入同一个哈希桶中的元素的个数可以是无限制的，如果某一个位置冲突过多的话，插入的时间复杂度将退化为O(N)，这种退化将引起缓存未命中率的骤然升高[45]。因此，每个哈希桶内元素的个数应在3个以内。

两种方法各有优劣，开放寻址在解决当前冲突的情况下可能会导致新的冲突，而开链不会产生这种问题。另一方面开链的局部性较之开放寻址法要差，在程序运行过程中可能引起操作系统的缺页中断，从而导致系统颠簸。

哈希表被广泛实现系统层和应用层软件，被集成到编程语言如Java，Python等，还可以用来实现关联数组，数据库索引，缓存，集合等。哈希表的高效性使其具有重要的研究价值和应用价值。

2.2 基于软件技术的同步方法研究

在并发哈希的设计中，除了选用的哈希函数、所采用的冲突处理方案以及数据结构上的差异造成性能上的差别之外，另一个重要的因素是选取的同步方案。

2.2.1 阻塞技术

(1) 锁方法

在并发数据结构中锁用于保证多个线程对数据结构的互斥访问，以避免线程间发生“错误的”交错，从而产生预期之外的结果。设计锁算法的关键问题是：当线程t_1试图申请线程t_2已经占有的锁时，t_1要采取的行动。在单处理器系统上，出现这种情况时唯一明智的处理方式是将处理器让给t_1。但是在多处理器系统上，因为锁可能在不长的时间内会被另一个处理器上执行的线程释放，所以使t_1反复尝试获取锁有助于提升性能。这种使线程不断的尝试获取锁的技术称为**自旋锁(spinlock)**。在线程执行期间很难预测该线程会持有锁多长时间，所以很难在阻塞技术和自旋锁之间做出选择。

简单的自旋锁重复使用同步原语，如比较和交换(swap-and-change，CAS)，以原子方式将锁从无主状态转换到被占有状态。如果锁的设计不够仔细，自旋锁会引起激烈的锁竞争，从而对性能造成严重影响。一种简单的降低锁竞争的方法是引入指数退避(exponential backoff)机制[46]。使用这种方法获取锁失败的线程在进行重试之前会等待一段时间；失败的次数越多，等待的时间越长，此时线程会及时的“自行分散”，由此降低了线程对锁的争用，同时也减少了由于尝试获取锁失败引起的流量开销。

使用指数退避机制的锁的缺陷是锁可以处于无主状态，而尝试获取该锁的线程都已经执行退避策略，需要等待较长的时间，因此在这段时间内所有的线程都

不会前向执行。解决这个问题方法是使所有申请锁的线程存储在一个队列中，锁被释放后将锁的所有权传递给下一个正在排队的线程。基于这种方法实现的锁被称为**队列锁 (quenelocks)**。Anderson[47] 和 Graunke[48] 提出了基于阵列的队列锁方法。之后 M.Crummey 和 T.Scott[49] 对他们的方法进行改进实现了基于列表的 MCS 队列锁，以及由 Craig 和 E.Hagersten 等人[50,51] 提出的 CLH 队列锁。

使用 CLH 锁的线程形成一个虚拟的节点链表，每个节点都包含一个 $done$ 标志；某个线程只有当列表中它的前继节点的 $done$ 标志被设置为 $true$ 后才进入临界区。为了获得锁，线程创建一个节点，将它的 $done$ 标志设置为 $false$，表示它还没有释放临界区，并且使用同步原语 (如 CAS) 将它自己的节点放在列表的尾部，同时确定其前继节点。随后，该线程在其前继节点的 $done$ 标志上自旋。值得一提的是，每个线程的自旋过程发生在不同的内存位置，因此，在基于缓存的体系结构中，当某个线程设置其 $done$ 标志以通知队列中的下一个线程可以进入临界区时，所有其他正在自旋的线程的 $done$ 标志不会被修改，这些线程将继续在本地缓存行上自旋，而不会产生额外的内存流量。这在很大程度上减少了争用，提升了扩展性。但是，如果使用这种锁算法的程序运行在 NUMA 平台上，某些线程不得不在远程内存结点上进行自旋，这样无疑会消耗更多的内存流量。使用 MCS 队列锁[49] 通过将线程自旋的位置限定在该线程自身节点的 $done$ 标志来解决 NUMA 平台上的问题。

此外，为了迎合特定的数据结构的数据读取特性在后续的演化中出现了一些标准锁方法的变体。队列锁算法中出现了一种具有“可中止的”特性的版本，它允许正在申请锁的线程放弃等待，比如在实时性要求较高的应用中延迟超过极限值时[52,53]，或者线程需要从死锁中进行恢复时。M.Scott 等人提出了抢占安全锁 (preemption-safe locks)[54]，它通过确保队列中被抢占的线程不会阻止锁被授予另一个正在运行的线程，从而试图减少锁抢占对性能造成的负面影响。

许多数据结构有并发读取的需求，因此，这样的读写锁只允许线程对临界区内的数据进行读取而不能修改，如果当前临界区没有写者线程进行操作，则允许多个读者线程并发访问。M.Crummey 和 T.Scott 等人提出的读写队列锁算法是基于 MCS 队列锁并且使用读计数器和指向写者节点的指针实现的[55]。Krieger 等人[56] 提出了一种通过设置队列节点的双链表的队列锁，这种方法的每个节点都有自己的简单“迷你锁”，读者通过获取其相邻节点的迷你锁并重定向双链表的指针来将自己从 queuelock 列表中移除。

锁对并发数据结构重要性不言而喻。选取锁的标准要遵循两个原则：一是要与应用场景结合；二是要能提供充分的扩展性。

2.2.2 屏障技术

屏障是这样一种机制，所有的提前执行到代码中指定的某些位置的线程悬停，只有当所有线程都到达这个点时才允许它们继续执行。当访问数据结构或应用程序需要划分成若干个不相互重叠的执行阶段时，就需要使用内存障碍。例如，并行垃圾收集器的标记和扫描阶段。此外，在本文中设计的统一的并发哈希表测试框架就用到了内存屏障。并发哈希表在测试之前需要进行初始化，也就是在哈希表内插入一些元素，让哈希表的密度达到预定的值。初始化的过程使用多线程共同完成，由于不需要删除元素，所以这个过程在数据量大的测试集中采用多线程会节约初始化时间。此时如果有现成提前完成了初始化的任务，它会触碰到内存屏障，必须要等待所有参与初始化的线程全部完成之后再往下执行。

实现屏障的一个简单直观的方法是使用计数器，计数器的值被初始化为参与运算的线程的数量。具体过程是：每个线程在到达屏障后递减计数器，然后自旋，等待计数器变为零，然后继续往下执行。这种直观的实现方式可能引起两个方面的问题：

- 当使用相同的 pass/stop 技术实现了多个串行屏障时，当有线程到达第二个屏障而在第一个屏障内还有一些线程没有完成时，会出现死锁；
- 由于所有的线程反复的查询全局变量的状态，导致通信流量大，从而对程序的可扩展性造成影响。

针对上述问题，实现了专门的屏障技术，可以让线程在不同的位置自旋[57–60]。或者也可以使用 Dijkstra 和 Scholten 风格的发散计算树来实现屏障[61]。在这种方法中，每个线程都是二叉树中一个节点的所有者。线程等候它的子节点的到达，然后通知该线程的父节点以表明自己的子节点已到达。一旦所有线程都已到达，树的根节点通过向下发送释放消息释放所有线程。除了通过软件技术实现的屏障之外，还有通过硬件实现了上述的屏障功能[62]。

2.2.3 无阻塞技术

如前文所述，使用无阻塞编程是为了克服使用锁方法带来的若干问题。无阻塞技术包含几类条件——无等待[63,64]，无锁[64] 和无阻碍[65]。三类条件由强到弱排列顺序依次为无等待强于无锁，无锁强于无阻碍。但是这三类条件都强于使用诸如锁之类的阻塞结构。虽然更强的前向条件是可取的，但是通常情况下实现较弱的保障条件更加容易、效率更高，并且易于设计和进行正确性验证。所以，在实际的应用中，研究人员往往通过采取退避策略[46] 或使用更复杂的竞争管理技术[66] 来补偿较弱的前向条件。

除了少数特别情况，非阻塞算法使用硬件必须提供的原子读-修改-写原语，其中最值得注意的是比较和交换指令 (CAS)。使用无阻塞方法实现的并发数据结构都是使用这些原语的标准接口实现的 (在一般情况下，即使是使用了读-修改-写

原子原语，临界区也是阻塞的)。

无等待算法具有最强的非阻塞前向保证条件，它保证所有 CPU 在连续处理有效工作时，没有运算会被其他运算所阻塞。如果每个操作完成所需的执行步骤是有限的，则认为该算法是无等待的算法。在性能成本不是太高的前提下这个属性对于实时系统具有非常重要的意义。早在上个世纪九十年代，Herlihy 等人就证明了所有的算法都可以实现无等待版本[67]，并且已经证明了很多被称为通用结构的串行代码转换。但是转化后的性能与设计初衷南辕北辙。有研究人员对实现无等待算法的难度进行了评估。比如，文献[68] 的研究表明使用 CAS、LL/SC 等原子条件原语很难在不增加内存消耗和损失线性的线程扩展性的前提下实现一般的数据结构的无饥饿算法。

2011 年以后，学术界和工业界对无等待算法的研究才开始重视起来。2011 年，Kogan 和 Petrank 提出了一种基于 CAS 原语的无等待队列[69]，这种无等待队列只需要普通的硬件支持即可实现。这种无锁队列是对 Michael 和 Scott[70] 提出的一种被广泛应用于实际的队列的扩展。2012 年 Kogan 和 Petrank[71] 等人又提出了一种提高无等待算法处理速度的方法并且使用这种方法实现的无等待队列的性能比无锁方法实现的相同的队列性能更好。2014 年，Timnat 和 Petrank[72] 提出了一种将无锁数据结构自动转化成无等待数据结构的机制。至此，无等待实现可以用于多种数据结构中。

无锁算法允许个别线程处于饥饿状态，但能够确保系统吞吐量。如果所有线程运行了足够长时间后，至少有一个线程能获得前向执行，那么这个算法是无锁的。所有的无等待算法都是无锁的。如果程序的某个或某几个线程被挂起，那么无锁算法能够保证剩下的线程能够顺利的执行。

2.3 NUMA 架构内存管理相关研究

2.3.1 线程与处理器内核的关联

当前，NUMA 系统上主要使用操作系统的调度程序将应用线程分配给处理器内核。调度程序考虑系统状态和不同的策略目标（比如“平衡核心负载”或“整合核心上的线程或使内核保持为休眠状态”），然后匹配应用线程和相应的物理核心。特定线程会在其分配的核心上执行一段时间，之后被交换到核心之外进行等待，因为其它线程也需要执行。如果另一核心可用，调度程序将选择迁移该线程，以确保及时执行并实现其策略目标。

将线程从一个核心迁移到另一核心会导致 NUMA 共享内存架构出现问题，因为它会断开线程与其本地内存分配之间的关联。也就是说，线程可能启动时在节点 N_1 上分配内存，因为它运行在 N_1 的核心上。但是当该线程后来迁移至 N_2

的核心上时，之前该线程在 N_1 上保存的数据变成了远程数据，内存访问时间大幅增加。

线程与处理器核心关联：处理器关联指线程/进程与特定处理器资源实例相关联的持续性（无论其它实例的可用性如何）。通过使用系统 API，或修改操作系统数据结构（比如关联掩码），特定核心或核心集可与应用线程相关联。然后在制定有关线程寿命的决策时，调度程序会关注这种关联方式。例如，线程可能配置成仅在处理器 P_1 的 0-3 号核心上运行。调度程序将在核心 0-3 之间进行选择，不会考虑将线程迁移至其他节点。

执行处理器关联可确保内存分配对有需要的线程保持局部性。不过，实行线程与处理器内核之间的关联也存在缺点。一般来说，如果本可以使用更好的资源管理方式，处理器关联将会限制调度程序的选择，并产生资源争用现象，从而对系统性能造成不利影响。除了阻止调度程序将等待线程分配给未利用的内核外，处理器关联的局限性还会对应用本身产生不利影响，因为其他节点上的额外执行时间无法弥补速度较慢的内存访问。

在进行 NUMA 系统上的并发哈希表的设计时，必须慎重考虑处理器关联方法是否与其数据结构的特点和共享系统环境相适应。值得注意的是，除显式关联外，部分系统提供的处理器关联 API 还支持向调度程序提供优先级“提示”和关联“建议”。相比于强制执行显示的线程绑定策略，使用此类建议能够确保在通用案例中实现最佳性能，并在高资源竞争环境下避免限制调度选择[73]。

2.3.2 NUMA 系统的非对称互连

NUMA 架构的多处理器系统的最显著的特点就是它具有非一致的内存访问时间。因此，线程和内存的分布对 NUMA 系统的性能起着至关重要的作用。NUMA 系统的这个特性在操作系统领域衍生出了许多的 NUMA 感知算法。这些算法要么侧重为线程分配最近的内存节点上的内存空间[74–76]，或是将内存页分散在系统中以避免内存控制器和互连链路的过载[76]，或是将共享相同数据的线程放置在同一个内存节点上[77,78] 以避免内存控制器的争用[78–80]，或是将可能产生缓存或内存带宽争用的线程分布在不同的内存节点上。

尽管上述的研究注意 NUMA 架构上的线程和数据的分布方式以发挥 NUMA 架构的性能，但是这些研究似乎都没有考虑到在未来可能会盛行的一个 NUMA 系统的重要属性：不对称互连。现代操作系统旨在减少用于线程间和线程到内存通信的跳数。运行 CPU 之间的负载平衡时，Linux 首先使用同一节点上的 CPU，然后是相隔一跳的 CPU，最后是相隔两跳或更多跳数距离的 CPU。这些技术假定节点之间的互连是对称的：即对通过直接链路连接的任何节点对而言，链路具有相同的带宽和相同的等待时间。然而在现代的 NUMA 系统中情况并非如此。也就是说，当节点通过不同带宽的链路连接时，不仅要考虑线程和数据是否放置在相

同或不同的节点上，而且还要考虑这些节点是如何连接的。B.Lepers[81] 等人通过研究 NUMA 系统的非对称性对 x86 系统的影响，发现在同一个节点上的线程和数据分布相同但节点间连接不同的情况下，性能可能会相差 2 倍以上。他们对于节点之间的互连有了新的认识，认为在不对称互连的特性下最好的互连方式是在总内存带宽最大的节点之间进行连接，而不是选择具有最小跳数的节点之间进行互连。基于这个观点，他们实现了基于 Linux 系统的动态线程和内存分配算法[81]。

2.4 事务内存相关研究

锁在设计并发数据结构中的关键作用是它允许线程对多个内存单元进行原子的修改，因此没有哪一个线程能够读取这些位置上的任何中间值。事务内存机制是一种允许用户自定义的将访问多个内存单元的代码片段作为一个原子步[82]。这种机制对于简化并发数据结构的设计具有重要的理论和实际意义。单纯的从算法实现上而言，编写代码时不再需要考虑哪些内存访问需要持有锁，并且还能有效的防止死锁问题。

用于实现并发数据结构的事务机制的灵感来源于数据库领域的事务的概念。虽然两者在概念上相通，但是在共享内存单元上支持事务不同于实现存储在磁盘上的数据元素的事务访问。因此，在这种情况下，可以对于共享内存单元的事务访问的支持可以使用更加轻量级方法实现。

2.4.1 实现事务内存的相关技术

Kung 和 Robinson 等人提出的乐观并发控制 (OOC)[83] 是一种用于实现并发数据结构的事务机制。OOC 的基本原理是在事务结束时短暂的持有全局锁。但是使用这种方法全局锁是一个性能瓶颈，对线程扩展性造成负面影响。理想情况下，事务性的访问的实现不应该依赖锁，并且在访问不相交的内存单元的事务之间不需要同步。

多处理器上的事务化支持首先是由 Herlihy 等人[84] 提出的，同时他们还提出了一种基于硬件的事务内存的方法。之后，这种基于硬件的事务内存的方法被 Rajwar 和 Goodman 等人扩展到包含硬件锁省略技术[85,86]。在他们的方法中使用硬件方法将临界区自动转换为一条事务，通过这种方法使两个或多个实际上不相互冲突的临界区可以并行的执行。

事务内存发展到今天，已经有了软、硬件两种实现：基于软件的事务内存 (STM) 和基于硬件的事务内存 (HTM)。不论是软件事务内存[87–91] 还是硬件事务内存[92–95] 都得到了充分的研究与长足的发展。随着 IBM z 系列[96] 和 p 系列[97] 处理器的出现标志着硬件事务内存从理论研究上升到实际研究，支持事务内存的 Intel Haswell 处理器[98] 的问世，标志着硬件事务内存从实验研究走向市场化。

Intel 的事务同步扩展指令集（TSX）是目前少数几款支持硬件事务内存的商用处理器之一（IBM 的 power CPU 也支持硬件事务内存）。目前硬件事务内存还存在一些问题，比如 Intel 的 Lemming 效应[99]；再比如硬件事务内存不能保证每次事务的执行都能成功提交，为了保证程序能够正确、顺利的执行，需要为程序设置回退路径处理事务不能成功提交的状况，而这个回退路径的实现往往是通过传统的锁方法实现的。事务内存能够轻易的用于实现并发数据结构，当能够克服上述问题时，使用硬件事务内存将是设计并发数据结构的首选同步机制。

2.4.2 基于事务内存的并发数据结构

随着支持 HTM 的多核处理器的问世，基于 HTM 的并发数据结构也得到了深入的研究。复旦大学的陈海波等实现了一系列基于 HTM 的并发树型数据结构[100–102]。之后，基于他们对于并发树型数据结构的研究发现由于数据冲突引起的事务中止对基于 HTM 的并发数据结构造成严重影响。基于这点考虑提出了 Eunomia[103]。Eunomia 是一种集成了若干条用于减轻事务中止的设计模式，主要用于优化搜索树这一类的数据结构。Z.Wang 等人设计了基于 HTM 的 skip list[104]，通过在 RTM 模拟器和真实的支持 RTM 的处理器上对比细粒度锁、无锁方法和硬件事务内存等同步方式的比较，总结了若干条提升硬件事务内存性能的规律。Afek[99] 等人设计了两种用于缓解 Intel TSX 的 Lemming 效应的软件优化方法，并将他们的方法与传统的 CLH、MCS 锁以及单纯的使用 HTM 的性能进行了比较。本文基于 HTM 的并发哈希表的设计以及并发 Cuckoo 过滤器的设计都借鉴了他们提出的软件辅助方法进行优化。

2.5 布隆过滤器

在数据库、缓存、路由器和存储系统中通常需要判定一个元素是否存在在某个集合内，这种判断允许一定的误报率。进行这种成员关系判定布隆过滤器 (bloom filter) 是使用得最多的一种数据结构[105]。布隆过滤器最初用于拼写检查和数据库检索，随着计算机处理海量数据的压力与日俱增，布隆过滤器的研究再一次焕发新春[106]。布隆过滤器因其高效的内存效率而备受关注。前 Google 研究员吴军[107] 在其《数学之美》一书中指出，使用布隆过滤器的存储效率大约是使用哈希表处理同等规模数据的 4 到 8 倍。

1970 年 B.Bloom[105] 提出了一种用于处理拼写检查的过滤器，由于其超高的空间效率和处理速度受到广泛关注，后来者为了纪念 B.Bloom 的突出贡献，将其创造的这种数据结构命名为 Bloom 过滤器。布隆过滤器支持对元素的插入和查询操作。误判率，记作 ϵ 和空间效率是判断布隆过滤器性能好坏的两个重要的指标。布隆过滤器对元素的查询返回两种状态：一是“绝对不存在”；二是“可能存在”（这

种可能存在的概率为 $1-\epsilon$)。误判率要求越低，用于表示每个元素所需的比特位越多。

为了弥补标准布隆过滤器不支持元素删除的缺陷，Counting 布隆过滤器[108]对标准的布隆过滤器进行了扩展。Counting 布隆过滤器的原理很简单，就是将原来的比特数组扩展成计数器数组，当插入某元素时，将对应位置上的计数器加 1，删除元素时，对应位置上的计数器减 1。一般的，为了防止算数溢出，计数器的大小为 4 比特或者 4 的倍数比特，所以实现 Counting 布隆过滤器需要至少 4 倍于标准布隆过滤器的存储开销。

Blocked 布隆过滤器[109] 重点在于优化查询效率，同样不支持删除操作。这种过滤器是由若干个小型的布隆过滤器构成的，每个布隆过滤器的大小为一个缓存行的大小。这一点与缓存行哈希表[12] 的处理方式类似。每进行一次查询操作，最多造成一次缓存未命中的结果，极大的提高了处理速度。Blocked 布隆过滤器的缺陷是由于各个小型布隆过滤器之间的负载不均衡造成误判率相对较高。

d-left Counting 布隆过滤器[32] 使用 *d-left hashing*[110] 将元素转化为指纹信息存储在哈希表中。删除元素时，找到对应的指纹信息进行删除。相比于 Counting 布隆过滤器，它的实现更简洁，而且空间性能也得到了很大提升，空间开销只有 Counting 布隆过滤器的 50%，处理同等规模的数据，所需的存储空间约为标准布隆过滤器的 1.5 到 2 倍。

Quotient 过滤器[33] 同样是一种通过哈希表存储指纹信息以实现元素删除的过滤器。它使用类似于线性探测的方法定位目标指纹信息的位置，这种方法具有更好的空间局部性。实现这种方法的代价是需要 10%-25% 的额外空间用于对哈希表内的实体进行编码。此外，编码后的实体到达目标元素所在的位置时需要解码成实体序列，哈希表的密度越高，实体序列越长。因此，这种过滤器的性能在哈希表的密度高于 75% 后急剧下降。

除了上述一些具有代表性的布隆过滤器的设计方法之外，还有一些用于优化特定应用场景的布隆过滤器，如拆分型布隆过滤器[111]，分档布隆过滤器[112]，以及将布隆过滤器扩展到多维空间的布隆过滤器[113] 等。

2.6 本章小结

由于并发哈希表对元素的查找和更新时间为常数级别的特性，因此这种数据结构被广泛的应用于多核架构上的软件系统的开发。学术界和工业界对于哈希表的研究重点也从单核处理器转移到多核处理器上。本章的主要目的在于对与本文研究内容相关的问题的研究现状和亟待解决的问题进行归纳，以期为后面章节的研究提供理论依据。本章首先对哈希表的基本概念进行了简单介绍，并对决定哈希表个性的哈希函数，哈希冲突处理技术进行了介绍；然后对用于实现并发哈希

表的软件同步方法进行简单的描述和比较；其次，介绍了 NUMA 架构与本文相关的特性——非对称互连和线程绑定；再次，介绍了事务内存的发展，实现事务内存的软硬件技术并对事务内存在构建并发数据中的相关研究进行了归纳；最后对与哈希方法密切相关的布隆过滤器技术进行了介绍。

第 3 章　基于多核系统的并发哈希表的评估与分析

多核处理器技术的发展为处理更为复杂的数据创造了可能的同时，也为如何通过设计或者优化新的并发数据结构来充分发挥多核系统的性能提出了挑战。

在这一章中，首先对与本章内容密切相关的基本概念进行介绍，随后对现有的几类主要的哈希表进行描述，介绍相关参数的意义以及配置方法，最后给出对现有几个具有代表性的并发哈希表进行评估结果，以及通过对实验结果进行分析得出的并发哈希表设计和应用的最佳实践建议。

3.1　实现并发哈希表的同步方法比较

为了保证多个线程能够有序地对内存进行访问，有基于锁 (lock-based)、无锁化编程 (lock-free) 和事务内存 (transaction memory) 几种主要的并发编程模型。

为了保证线程安全性，基于锁的并发哈希表对临界区进行加锁操作。被锁保护的内存区间称为**临界区** (critical section)。根据临界区的长短，基于锁的方法又可以划分为粗粒度 (coarse-grained) 锁和细粒度 (fine-grained) 锁。

一般的粗粒度锁在实现上相对简单，它使用少量的锁将受保护的数据结构分成几个区间，极端的情况是整个数据结构用一个全局锁。这就使得临界区往往特别长，造成数据冲突的概率极高，不利于计算机资源的高效利用。细粒度锁方法是使用锁对数据结构中的基本单元进行保护。比如哈希表中每个哈希桶设置一个锁字段对哈希桶进行保护，这样只有当不同的线程同时访问同一个哈希桶时才会造成冲突。细粒度锁方法的好处是允许多个线程对数据的不同区段进行并发的读/写，大大提高了处理能力和资源利用率。锁的粒度越细，越有利于提升整体性能，但是同时设计基于细粒度锁的数据结构的复杂度更高，并且需要耗费大量精力进行正确性验证。同时使用大量的锁占用的存储空间也不容忽视。

无锁化编程是相对于基于锁的编程范式而言的同步方法。无锁化编程是用计算机原语来代替显示锁的一种并发编程范式。同样的，无锁并发哈希也得到广泛的研究[11,16,19]。使用无锁化编程设计并发数据结构能够获得良好的线程扩展性和性能。但是，使用这种设计方法的复杂度更甚于细粒度锁实现。

事务内存的概念是沿用数据库事务处理的概念，数据库事务秉承 ACID 原则，即事务具有原子性，一致性，隔离性和持久性。事务内存也遵循 ACID 原则：

- 原子性：事务代码要么全部执行，要么全部不执行，不存在事务停滞在中间的某个状态，一旦因为某些原因需要中止，则事务回滚到事务开始执行的状态；

- 一致性：事务内存不会破坏数据的完整性和执行的事务代码的逻辑性；
- 隔离性：并行执行的多个事务代码区域互不干扰；
- 持久性：一旦事务代码成功执行完成提交，对系统状态所做的更改就会生效并且不会回滚，直到有新的事务执行结果对其进行修改。

事务内存的初衷是让用户能够在多核系统上用粗粒度锁的实现方式进行软件开发，并且获取接近甚至超过细粒度锁或无锁化编程锁获得的性能。目前，软件事务内存[87–89]和硬件事务内存[21,92,114]都得到了较好的实现。但是，在使用硬件事务内存时还需要使用软件辅助方案，具体的细节将在下一章中详细介绍。

3.2 典型的并发哈希表

对串行哈希表的研究已日臻成熟，但是在主流处理器生产商相继推出多核处理器之后，传统的串行哈希表已经无法充分利用多核系统的计算资源，也无法满足多核架构上的性能需求，在这样的背景下，相关研究人员开始着手高性能的并发哈希表的研究与设计。并发哈希表继承了串行哈希表的快速索引，高效插入和删除元素的特性。学术界和工业界的研究人员针对不同的应用场景提出并实现了一些具有特色的并发哈希表。这些并发哈希表有基于用户态Read-copy Update(urcu)机制实现[19]，有被应用于memcached系统的多读单写Cuckoo哈希[115]，有应用于企业级应用的Threading Building Blocks (TBB)[116]，有集成到编程语言内的Concurrent_HashMap[117]，有使用消息传递机制来代替锁的CPhash[16]。在哈希表密度非常高的情况下仍然可以保持较好性能的Hopscotch哈希[15]，以及遵循最小缓存行切换原则而设计的缓存行哈希(CLHT)[12]等。

表 3.1 用于评估的五种并发哈希表实现

序号	算法名称	设计思想	语言
1	Cache Line Hash Table (CLHT)	Minimizes cache line transfers[12]	C
2	Hopscotch Hashing (Hopscotch)	Combines the features of cuckoo, linear probing and chaining[15]	C++
3	Concurrent Cuckoo Hashing (Cuckoo)	A concurrent cuckoo hashing supports multi-reader/multi-wirter[14]	C++
4	User-Level Read-copy Update (URCU)	lock-free, trades update performance for read-side performance[19]	C
5	Threading Building Block (TBB)	Based on separate chaining, scales well for read-heavy workload[116]	C++

3.2.1 缓存行哈希表

T.David等人提出了一种“异步并发(ASCY)”的思想[12]，他们提倡遵循异步并发的四条编程模式来进行CSDS的设计。四条ASCY的编程模式内容如下：

- **ASCY**$_1$：CSDS的搜索操作不应该包含任何等待、重试或者写操作；
- **ASCY**$_2$：更新操作的解析阶段不应该包含任何重试或者等待，除非有清除当前内容的需要，否则不应包含任何写操作；

- **ASCY**$_3$：当某个更新操作在解析阶段失败之后（比如，需要删除某个元素时没有在哈希表内发现该元素或者在插入元素时发现该元素已经存在于哈希表内）不应当执行任何写操作，除非有必要对解析阶段产生的数据进行清理；
- **ASCY**$_4$：在成功的更新操作中进行内存写操作的次数和写入的区域应当尽量与标准串行实现方法所消耗的次数和区域相近。

频繁的缓存行切换对并发哈希表的性能是灾难性的。明确了这一事实后，T.David 等人在四条异步并发模式的基础之上设计了缓存行哈希表 (CLHT)[12]。CLHT 的核心思想在于“并发算法想要获得良好的可移植性和可扩展性，它对于共享状态的内存的访问就需要像串行化那样是异步进行的。”CLHT 首要的设计准则就是“尽最大可能的缓存行的切换”，在这一准则下设计出的 CLHT 展现出极佳的性能。为了确保大部分操作都能在一次缓存行切换内完成，CLHT 的哈希桶被精心的设计成与缓存行相同大小（64 Bytes，一般处理的缓存行大小都为 64 Bytes）。CLHT 的冲突处理使用的是开链法，它的哈希桶通过指针链接。因此，CLHT 的哈希桶被隔离成 8 个字（8 Bytes 为一个字长），其中一个字用于进行并发控制，6 个字用于存储三组键/值对，另外的一个字用于指向其他哈希桶。直观的，完成一次更新操作（比如，在哈希表中插入新的元素），至少需要执行一次对共享状态的修改。然而，根据 ASCY 指出的原则，查询操作不应该包含任何的写操作。因此，CLHT 的查询操作需要对跟当前键对应的哈希桶进行解析并且不经过任何同步就返回结果。为了实现就地更新，对哈希桶的解析不单纯的是对键进行遍历，还要同时获取每个键/值对的快照。该原子快照确保搜索操作在找到目标键之后，与该键对应的值被返回但不会涉及并发修改。

CLHT 有基于锁（CLHT-lb）和无锁（CLHT-lf）两个版本，本文对两个版本都进行了评估。CLHT-lb 采用细粒度锁 (每个哈希桶都有一个锁字段) 完成对读者和写者的同步控制。查询操作遍历键/值对，如果匹配，则返回值。更新操作首先需要执行一次查询以确定该操作可以继续执行（如果插入元素时发现桶内已存在相同元素，或者删除元素时发现桶内没有该元素，则不进行下面的操作），如果可以继续执行，则持有该哈希桶的锁，直到完成相应的更新操作，完成后释放锁。如果当前映射到的哈希桶内已经没有足够的空间插入新的元素，则会选择使用指针字段链入一个新的哈希桶，或者触发哈希表扩容操作（resize）。

而对于 CLHT-lf，为了保持其插入键/值对时的原子性，设计了一个 *snapshot_t* 的对象。*snapshot_t* 的大小为 8 字节，它包括一个 4 字节的版本号和一个 4 字节的 map。*snapshot_t* 提供在 map 内原子的读取/修改索引值的接口。版本号被用于同步并发修改，map 则用于确认对应哈希桶的索引值的状态是有效，失效还是正在被插入元素。简而言之，CLHT-lf 的原子性的过程可以大致描述如下：首先，在进入原子区间之前读取 *snapshot_t* 对象的值；然后，使用比较并交换 (CAS) 原子指

令对 map 内的目标索引值进行读取/更改。举个例子，如果其它线程执行的并发插入已经完成了，那么当前的线程就会使 CAS 失败，因为两次的版本号不一致。最后，可以通过 map 内的置位情况来判断给定的键/值对是处于有效、失效还是正在被插入三种状态中的哪一种。

3.2.2 Cuckoo 哈希表

与 CLHT 所不同的是，Cuckoo 哈希方法使用的开放寻址法解决哈希值冲突的问题。使用的冲突处理方式不同注定了它们在数据结构上的差异。Cuckoo 哈希表中，所有的键/值对都被存放到一个大数组内，没有指针也不使用链表。为了处理哈希冲突，它使用了两种技术：

- 第一，元素可以插入到桶数组内的两个位置 (设置了两个哈希函数)，如果其中一个位置被占用，则尝试插入到另外的位置上；
- 第二，哈希桶设计采用多路组相连的方式，也就是说，对于每个桶都有 B 个“槽位 (slot)”可供元素插入。

查找键 k 时，分别由哈希函数 f_1 和 f_2 计算得到 k 的两个可供存储的位置 b_1 和 b_2，然后在 b_1 和 b_2 中检查 k 是否存在。图 3.1给出了一个使用 2 个哈希函数，4 路组相联的 Cuckoo 哈希表。采用这种设计带来的好处是：只需要检查 $2*4$ 个键就能完成一次查询操作。因此查询操作非常快速并且完成一次查询所要查找的位置是可预测的。而在哈希表内插入新元素时，如果 b_1 和 b_2 中任意一个哈希桶内有空闲的位置，那么就将该元素存到这个桶内；如果两个对应的桶都没有空闲位置，则随机的从待插入的哈希桶内选择一个元素踢出去，然后新的元素插入到被踢出去的元素的位置上。被踢出的元素则重新计算候选位置，可能会踢出其它的元素以供自己插入，如此往复，直到没有元素被踢出或者达到预先设定的最大踢出次数为止。如果最终仍有键没有找到空闲的位置进行插入，则说明哈希表的负载因子接近极限了，此时会考虑对哈希表进行扩容操作。一般的做法是将哈希表的容量扩大一倍，然后从扩容前的表中将数据拷贝到新创建的哈希表中。在执行插入操作的过程中，被踢出的元素序列被称作一条 Cuckoo 路径。当表的负载因子升高时，Cuckoo 路径的长度会增加，每执行一次插入所需要的随机读/写的次数也会增加，Cuckoo 的更新性能也因此受到影响而下降。

Cuckoo hashing 方法最早是由 R.Pagh 等人在 2004 年提出的[118]，其原始版本并不支持多线程并发。之后由 X.Li[115] 和 B.Fan[14] 等人分别实现了支持多读单写和多读多写的并发 Cuckoo 哈希表。本文中使用的 B.Fan 等人实现的多读多写 Cuckoo 哈希表。

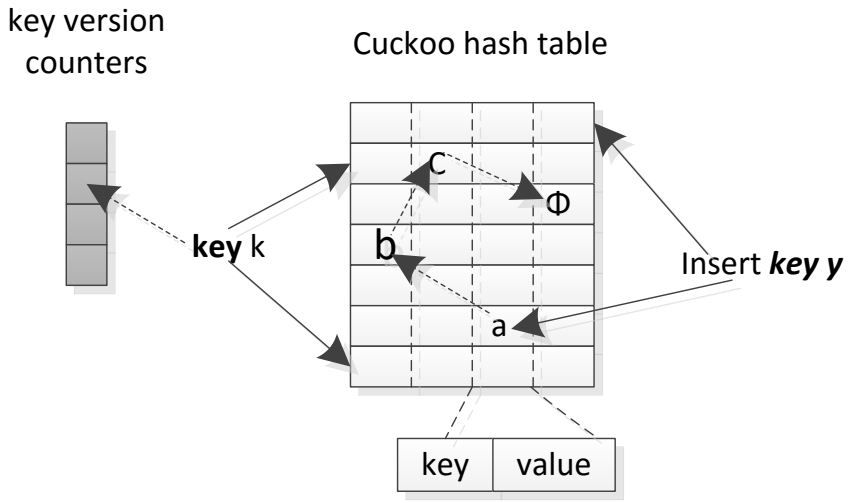


图 3.1　4 路组相联 Cuckoo 哈希表

3.2.3　Hopscotch 哈希表

与 Cuckoo 类似，Hopscotch 也是采用的开放寻址法解决哈希冲突问题。它在设计时综合考虑了 Cuckoo 哈希，线性探测法和链式法的特点，并将三者的优点结合起来。Hopscotch 哈希表由哈希桶数组构成。它的核心概念是，对于任意的哈希表内的元素，它周围的所有哈希桶都称为其邻居哈希桶。邻居哈希桶具有一个重要的特性：在邻居桶内查找元素所需的开销与在被映射的桶中查找元素所需的开销相同或者非常接近。这个特性是专门为处理插入操作而设计的。

图3.2实例描述 Hopscotch 插入元素的过程。元素被映射进的哈希桶总是在经过哈希函数计算得到的哈希桶内或者在其相邻的 H − 1 个哈希桶内，其中 H 是设定的常量（一般 H 为 32 或者 64 位，一个标准机器字长），可以根据需求进行调整。每一个哈希桶包含一个字节的’跳‘信息和一张 H 比特的位图，位图指示在当前哈希桶的下 H-1 个哈希桶内包含该元素的桶的偏移量。通过查看‘跳’信息能够得知哪些元素属于同一个哈希桶，然后只需扫描常数级别的哈希桶的数量快速找到对应的元素。从任意相邻桶内找出某一特定元素的开销等同或者非常接近于直接从该元素所属的桶内进行查找的开销。

总之，Hopscotch 的设计思想就是将空闲的槽位向着目标哈希桶移动，或者像 Cuckoo 哈希那样将元素从目标桶中移除然后重新为其寻找合适的位置。Hopscotch 实现了串行和并发两个版本[15]，本文中用到的支持多线程并发的 Hopscotch。

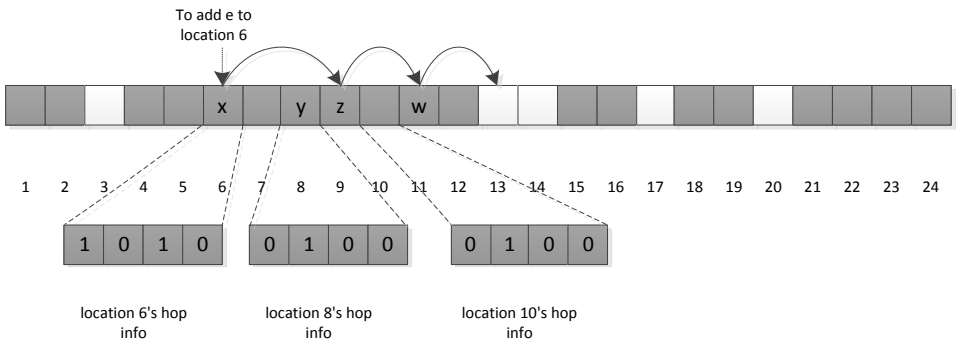


图 3.2　Hopscotch 插入操作示例

3.2.4　基于 RCU 机制的哈希表

读-复制更新，Read-copy Update(RCU) 并不单纯是一种并发哈希表的设计方案，它最初是一种用于优化 Linux 系统内核的共享数据访问的一种并发模型[119]。RCU 是一种无锁化编程模型。对于被 RCU 机制保护的共享数据结构，读者线程可以不需要锁自由地对其进行访问，但是写者线程在对该共享数据结构进行修改前必须保留该数据结构的副本。当修改完成之后，通过适当的回调函数将指向旧数据的指针修改成指向新的数据（从旧数据拷贝的数据）。RCU 的设计中有两个关键的时间区间：一个是 Reader Lock 时间；一个时 Grace Period。Reader Lock 时间是读者线程持有锁到释放锁之间的时间，Grace Period 是指写者线程从修改临界区的数据结构开始，到所有的读者线程都经过了至少一个 Reader Lock 时间的时间段。假设 Grace Period 开始的时刻记作 T_{start}，Grace Period 的持续时间记作 t_{gp} 和 Reader Lock 的持续时间记作 t_{rl}，则 t_{gp} 和 t_{rl} 之间存在两种不同的关系：

- 如果 t_{rl} 横跨 T_{start}，则 t_{gp} 必然在 t_{rl} 结束之后结束；
- 如果 t_{rl} 开始于 $_{start}$ 之后，则 t_{gp} 可能在位于 t_{rl} 时间段内的任意一个时刻结束，也可能在 t_{rl} 结束之后结束。

在写者线程删除数据时，无论两者处于什么样的关系，在经历了 Grace Period 之后所有的读者线程都不可能获取到在 T_{start} 之前的旧数据，所以删除操作是安全的。

基于 RCU 的并发哈希表的最大特点是牺牲更新性能来换取读性能。基于 RCU 的编程模型主要用于 Linux 系统内核，为了便于将 RCU 集成到我们的测试框架下进行测试我们使用用户态的 RCU 实现 (URCU)[19] 。本文使用的 URCU v-0.8.8 版本[120]。

3.2.5　基于 Intel TBB 的哈希表

Intel 推出的线程构建模块 (Threading Building Blocks, TBB) 是一种针对多核平台的基于任务的并行编程模型[116]。它实现的 concurrent_hash_map 允许多个线程并发的访问。concurrent_hash_map 的键是无序的。每一个键在哈希表内最多只有

一个元素与之相对应，但不排除有其它的元素尝试插入到相同位置。类 *const_accessor* 和 *accessor* 统称为访问控制器 (accessor)。访问控制器允许多个线程并发的访问哈希表内的键/值对。访问控制器充当指向键/值对的智能指针。它持有作用于键/值对上的隐式锁，直到该实例被销毁或者访问控制器调用解锁函数才会释放。

TBB 实现的并发哈希表处理冲突采用的是典型的链式法。键被哈希到包含了由若干元素形成的链表的哈希桶内。基于 TBB 的并发哈希表继承了链式法的所有优缺点。适合处理读占多数的数据集。本文将 TBB v4.2 集成到 CHTBench 中进行测试。

3.3 统一的跨平台并发哈希表测试框架的设计

前文在进行并发哈希表介绍时提到，文献中提出的并发哈希表通常在设计（采用什么样的同步编程模型，基于锁还是无锁），实现（采用什么样的硬件优化方案）以及评估方法（总和的测试集合与测试方法）上都存在差异。这些差异使得用户在横向上很难直观的进行比较，也很难判断究竟哪种因素限制了并发哈希表的性能。在这部分内容里，将介绍一种用于评估比较并发哈希表的测试框架——**CHTBench**。CHTBench 能为参与评估的并发哈希表提供一个公平的测试环境，排除编译器、数据分布、线程调度方式、键值大小等因素的干扰。

3.3.1 参数说明

表 3.1列出的哈希表使用 C/C++ 编写，为设计测试框架提供了便利。可执行程序使用 gcc 4.8 编译生成，编译优化选项为‘－ O3’。为了简便起见，所有的键值对均为 64 位整形数。*n* 表示需要创建的线程数量，创建 *n* 个线程用于并发的执行 *Insert*，*Delete* 和 *Lookup* 操作。四个测试平台所能创建的最大线程数量见表 3.2。*c* 为范围在 1 到 100 的随机数，它用于控制执行查询和更新操作的比重，确保执行的操作的比重跟预期一致。*d* 表示一次测试运行的时间，单位为毫秒，它用于控制 *while* 循环什么时候结束。创建的所有线程都将执行包含 *u*% 更新操作的工作集，*u* 表示更新操作占总的操作数量的百分比。更新操作包含插入操作和删除操作，如没有特别说明，二者所占的比重是相同的。因此，总的操作数中查询操作所占的比重为 100 - *u*%。*i* 为预先填充进哈希表的元素的个数，一般的 *i* 为 2 的幂次方。*r* 表示范围在 1 到 2*i* 的值，它表示所产生的键的位置。设置成最大值为 2*i* 的目的是为了确保有 50% 的操作是失败的操作。

3.3.2 测试逻辑

多线程的创建和销毁都使用 pthread 库提供的接口。在 3.4.6 中详细介绍了三种不同的线程绑定方案，编译时可通过命令行配置采用哪种方案。编译时只需要输入相应的关键字就能使用对应的线程绑定方案进行编译。哈希表的初始化在执行所有操作之前，初始化设置了两种方式，一是使用单个线程进行初始化；一是所有被创建的线程都参与初始化；在使用多线程完成初始化时，初始化任务被均匀分配到每个线程。为了避免某些线程提前完成初始化转而执行其他操作，设置了内存屏障。这样只有当所有的线程都完成了初始化任务，才会开始下一阶段的任务。

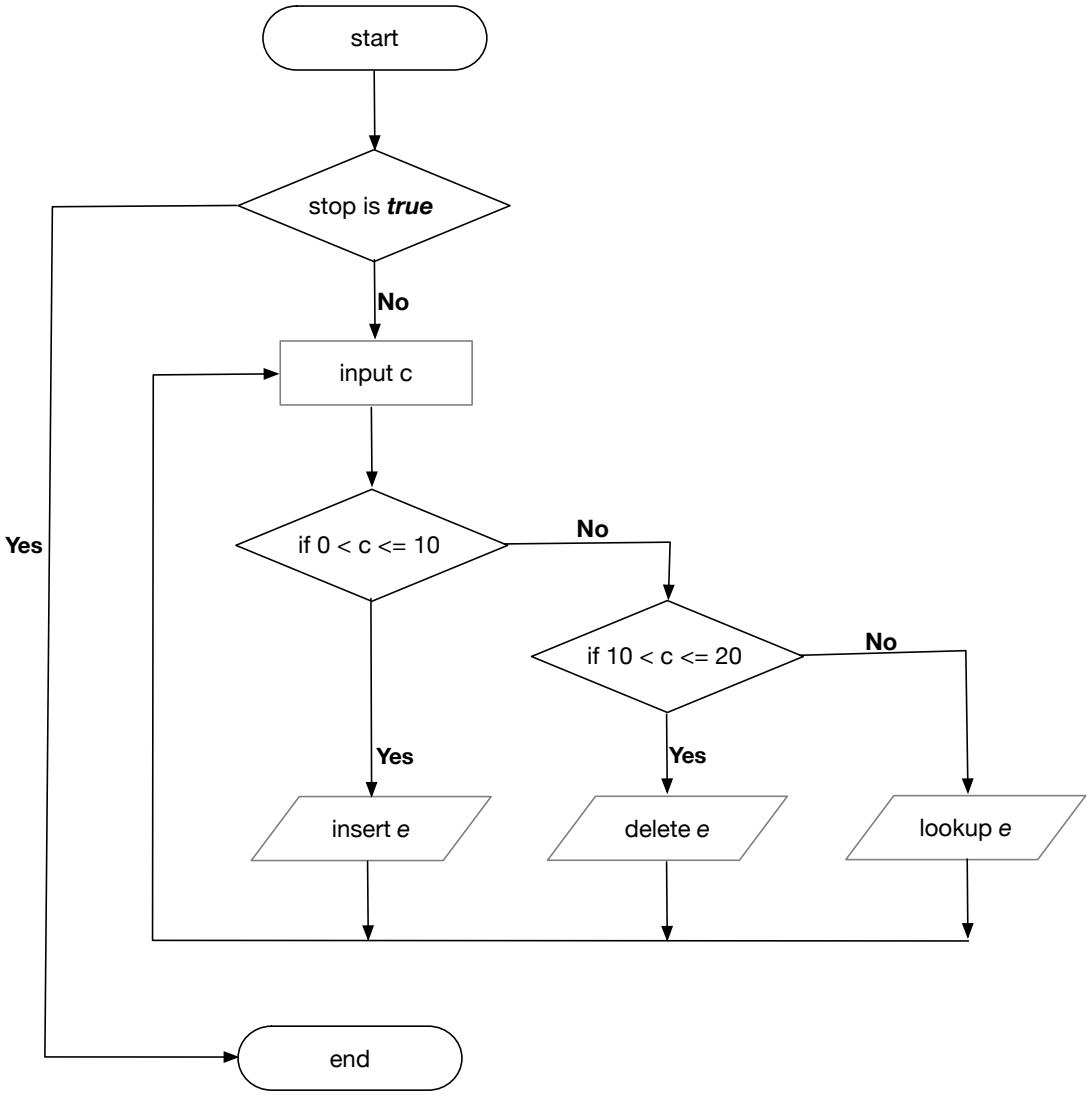


图 3.3 CHTBench 测试流程图

如图 3.3所示，进入测试函数后，先确定此次执行哪种操作。这个由随机值 c 控制。下面通过一个简单的例子说明参数 c 的用法。假设测试集中更新所占比重为 20%，则插入和删除各占 10%，而查询操作所占的比重为 80%，如果 $0 < c \leqslant 10$ 时，执行插入操作；$10 < c \leqslant 20$ 执行删除操作；而 c 为其他值时则执行查询操作。

为了统计执行的总的操作次数，以及各项操作执行成功和失败的次数，每执行完一次操作，相应的计数器加 1。最终通过统计相关计数器的值再除以执行时间得到吞吐量。

3.3.3 延迟测量工具

在并发哈希表中，线程申请锁的延迟，或是对某些数据执行原子操作的延迟将对其性能造成直接的影响。因此，在并发哈希表的评估中有一项重要指标——延迟。CHTBench 测试框架集成了一个简易的解析器：$sspfd$[121]。$sspfd$ 能够以极细的粒度（单个 CPU 时钟周期）分类记录各项操作的执行时间，当应用程序有延迟统计的需求时，$sspfd$ 对统计结果进行统计性的分析（比如计算标准差，绝对偏差，值聚类等），最后将统计结果输出。

使用 $sspfd$ 测量应用的延迟时，将 SSPFD_DO_TIMINGS 设置为 1，设置为其它值则表示关闭 $sspfd$ 的功能。$sspfd$ 提供了一系列访问接口，具体如下：

1. **SSPFDINIT(num_stores, num_entries, id)**：初始化函数，参数 num_stores 表示要解析的操作类型数量，$num\_entries$ 表示统计每种操作的延迟需要的样本数量，样本越多测量结果越准确，id 表示用于输出打印结果的线程的编号；
2. **SSPFDI(store)**：获取当前时间戳作为启动时间戳 t_{start}，开始编号为 $store$ 的操作的测算；
3. **SSPFDO(store, entry)**：获取当前时间戳作为终止时间戳 t_{end}，将 $t_{end} - t_{start}$ 的值存储到 $entry$ 中。
4. **SSPFDSTATS(store, num_ops, statsp)**：为第 $store$ 类型的操作的前 num_ops 个值生成统计信息，将结果存储到 $sspfd\_stats\_t$ 结构体的 $statsp$ 指针中；
5. **SSPFDPRINT(statsp)**：打印 $statsp$ 指针中的统计信息；
6. **SSPFDPRINTV(store, num_print)**：打印 $store$ 的前 num_print 个测量信息；
7. **SSPFDP(store, num_vals)**：为 $store$ 的前 num_vals 个值生成统计信息并打印；
8. **SSPFDPN(store, num_vals, num_print)**：在 7 的基础上额外打印这类操作的前 num_print 个测量值；

3.4 并发哈希表的评估与分析

并发哈希表（CHT）是一种允许在同一时刻有多个读者或写者访问共享对象的哈希表。其提供与串行哈希表一样的访问接口，但是并发哈希表能够更有效的发挥多核处理器的性能。并发哈希表的性能不仅依赖于应用本身的需求而且还依赖于底层硬件特性。所以，对并发哈希表的剖析不能单纯的停留在吞吐量、延迟

等直观的指标上，而是要综合考虑微观和宏观，底层和上层等多个层面的影响。进一步说，选用通用的测试评估指标在一个统一的测试框架下进行测试，处理不同的工作集时没有任何一种并发哈希表能全面的体现其优势。另一方面，对于用户而言，能够预先知道某种并发哈希表的性能障碍将对其挑选合适的并发哈希表提供帮助。不幸的是，这种用户关切的问题在现有研究中鲜有人提及。由于缺乏一个统一的测试框架，用户也很难通过自行测试进行比较而选出理想的 CHT 应用到其软件系统中。总之，对并发哈希表进行全面深入的剖析对并发哈希表的使用、设计以及优化都具有重要意义。基于以上考虑，我们从现有的并发哈希表中挑选出表 3.1中列举的五种哈希表进行深入的分析。

具体的，将挑选的并发哈希表放入 CHTBench 的框架内进行测试，对并发哈希表的评估与分析将围绕并发哈希表的线程扩展性、更新比重对性能的影响、初始化的哈希表元素的规模对性能的影响、延迟、线程绑定方案、同步机制以及内存消耗等七个方面展开。

3.4.1 测试平台与配置

为了体现对并发哈希表的评估具有一般性，测试在四台基于不同体系架构的多核处理器上展开。它们分别是 AMD Opteron 6172，Intel Xeon E5-2630，Xeon E7-4850，以及 Intel Xeon Phi 7120p。表 3.2 给出了四个平台的硬件和系统特征。测试过程中为了避免因操作系统的差异引入的干扰因素，所有测试平台都安装的是 Ubuntu 14.04 LTS 操作系统。下面分别对四台机器的硬件特征逐一进行介绍。

表 3.2 测试目标平台的硬件和系统特征

名称	AMD Opteron	Intel E5-2630	Intel E7-4850	Intel MIC
系统	Magny Cours	Ivy Bridge-EP	Haswell-EX	Knights Coner
处理器	Opteron 6172	Xeon E5-2630	Xeon E7-4850	Phi 7120P
核/线程数量	24/48	16/32	48/96	61/244
时钟频率 (GHz)	2.1	2.4	2.3	1.238
L1 缓存 (KB)	64/64 I/D	32/32 I/D	32/32 I/D	32/32 I/D
L2 缓存 (KB)	512	256	256	512
LLC 缓存 (MB)	2x6	20	24	NULL
互联通道	6.4 GT/s HT	2x QPI	3x QPI	NULL
最大内存带宽 (GB/s)	42.7	51.2	68	352
主存（GiB）	128	56	128	16

AMD Opteron. 该 48 核的 AMD Opteron 机器包含 4 个 Opteron 多芯片模块（MCMs）。该系统总共具有八个内存节点: 每一个 MCM 分为 2 个片区，每个片区

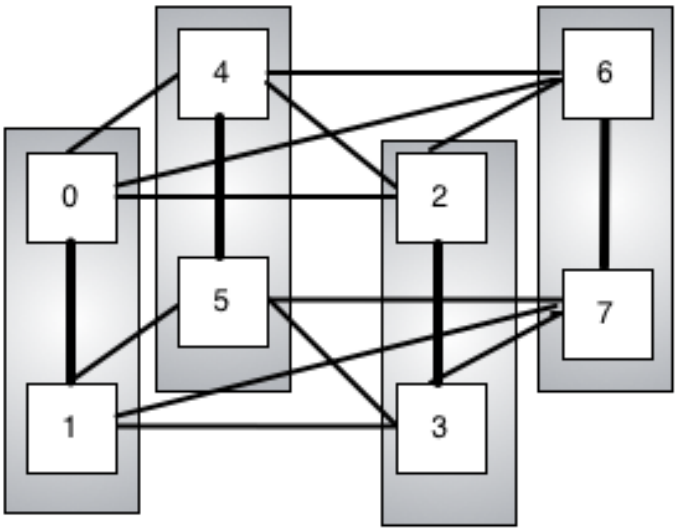


图 3.4　AMD Opteron 内存结点拓扑结构

拥有 6 个核，各个片区使用独立的内存控制器进行控制。该系统的拓扑结构如图 3.4 所示。位于同一个多芯片模块内的两个片区之间的距离定义为 1 跳（1-hop），位于不同多芯片模块上的两个片区距离为 2 跳（2-hop）。多芯片模块内通信开销要低于多芯片模块间的通信开销，并且模块内的两个片区之间的共享带宽比位于两个不同模块内的片区间的共享带宽要高。它的 CPU 的时钟频率为 2.1 GHz，三级缓存的容量分别为 64 KB，512 KB 和 4 MB（每一个片区的私有缓存容量）。总的运行内存为 128 GB。Opteron 的缓存具有 write-back 和 non-inclusive[122] 的特性。然而，存储的层次结构并不具备严格的排他性，也就是说在 LLC 中命中的数据会被推送到 L1 中，至于该数据会不会在 LLC 中被删除取决于硬件实现。Opteron 采用 MESI 的扩展性协议 MOESI 作为缓存一致性协议。其中‘O’表示占有（Owned）状态，它表示缓存行的内容被修改，与内存中的数据不一致，不过在其他的核上可能有这份数据的副本。

Intel Xeon E5-2630. Intel Xeon E5-2630 有两个 socket，每个 socket 包含 8 个物理核（16 个硬件线程），它的运行内存为 16 GB，时钟频率为 2.4 GHz，三级缓存的容量为别为 32 KB，256 KB 和 20 MB。该系统具有 4 个内存通道和两条 6.4 GT/s 的快速互联通道（QuickPath Interconnect, QPI）。E5-2630 的最大内存带宽可达 59 GB 每秒。它的缓存是 inclusive 的，就是说每一个新的缓存行的填充都将在三级缓存中同步[123]。最后一级缓存是 write-back 的，当出现由空间不足或者一致性问题引起处于‘M’状态的缓存行被驱逐时，数据被写入到内存中。

Intel Xeon E7-4850：该平台由 4 个 socket 组成，每个 socket 内有 12 个物理核（24 硬件线程），它的运行内存为 128 GB，时钟频率为 2.3 GHz。三级缓存的容量分别为 32 KB，256 KB 和 24 MB。它具有 4 个内存通道和 3 个 QPI。该机器的最大内存带宽为 68 GB 每秒。它采用的缓存一致性协议与 E5-2630 相同。

Intel Xeon Phi 7120p：该机器在同一片芯片上集成了 64 个有序核心。每一个核心均支持最多四个硬件线程，因此该机器的最大硬件线程数量可达 244 个之多，单个核心的时钟频率为 1.23 GHz。Intel Xeon Phi 7120p 的内存分层结构类似于传统的多核系统。图 3.5 所示为其体系结构。Phi 上的内存为所有核所共享，所有的核心均可对其进行访问，该机器的内存大小为 16GB。每一个核心都有一个大小为 32KB 的一级数据缓存和 32KB 的一级指令缓存，以及一个大小为 512KB 的二级私有缓存，片上二级缓存的总量为 31MB。Phi 实现了一个扩展的 MESI 协议，该协议的新颖性在于其将 Shared 状态扩展为一个基于目录的缓存一致性协议，这个协议简称为 GOLS(Globally, Owned, Locally Shared)，该协议的好处是能够共享被修改的缓存行，并且避免地址总线上的广播风暴。每一次的缓存都通过查看 GOLS 协议来确定缓存行的状态。Phi 的全局一致性的维护是通过纪录有每一个缓存行一致性状态的分布式目录标签（Distributed Tag Directories, DTDs）完成的。每一个缓存行的地址都通过一个哈希函数映射到 DTD 上，这样做的好处是保证负载均匀分布。

3.4.2 线程扩展性

线程扩展性是指在多核平台上随着被创建的线程数量的增加，应用的吞吐量保持不减的趋势。线程扩展性直观的体现在吞吐量上，吞吐量也是很常用的性能评价指标之一。吞吐量的计算方式：执行操作的总数除以总时间。单位用百万次操作每秒（Mops/s）表示。

(1) 总体的扩展性评估

本次实验中，采用紧凑的线程绑定方案（具体见 3.4.6 ），每次测试的持续时间为 5 秒，实验的结果如图 3.6 所示，每一组参数配置都运行五次，最终结果取五次结果的平均值。

从图 3.6 的曲线变化可以观察到，在这样的参数配置下，URCU 在多个平台上的吞吐量曲线几乎与 x 轴重合，也就是说在更新比重为 10% 的情况下，URCU 的性能很糟糕。而 CLHT（由于 CLHT-lb 和 CLHT-lf 之间的趋势相同，数值接近，故不单独进行讨论）则在四个平台上均展现出最佳的性能，这得益于它的设计出发点：尽可能的减少缓存行的切换次数。Hopscotch 的吞吐量在 E5-2630 和 E7-4850 两台机器上达到峰值时对应的线程数分别为 16 和 24。值得注意的是，16 和 24 正是这两台机器单个内存结点（socket）支持的最大线程的数量。同样的它在 AMD Opteron 上的峰值出现在线程数为 6 这个点（6 恰好是该机器同一片区支持的最大线程数）。综合 Hopscotch 在三个 NUMA 架构平台上的表现，可以推断 Hopscotch 在降低跨内存结点通信开销方面存在缺陷，它在单内存结点的多核计算机平台上应该有不错的线程扩展性。换句话说就是 Hopscotch 在利用 NUMA 系

统特性上所做的优化力度不够，更多的细节描述见 3.4.6 节。

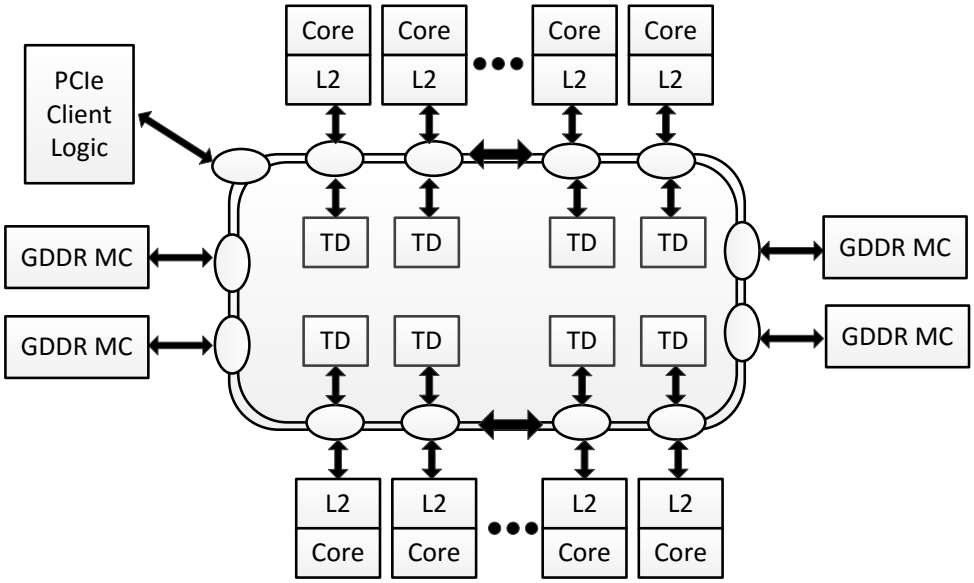


图 3.5　Intel Phi 7120p 体系结构

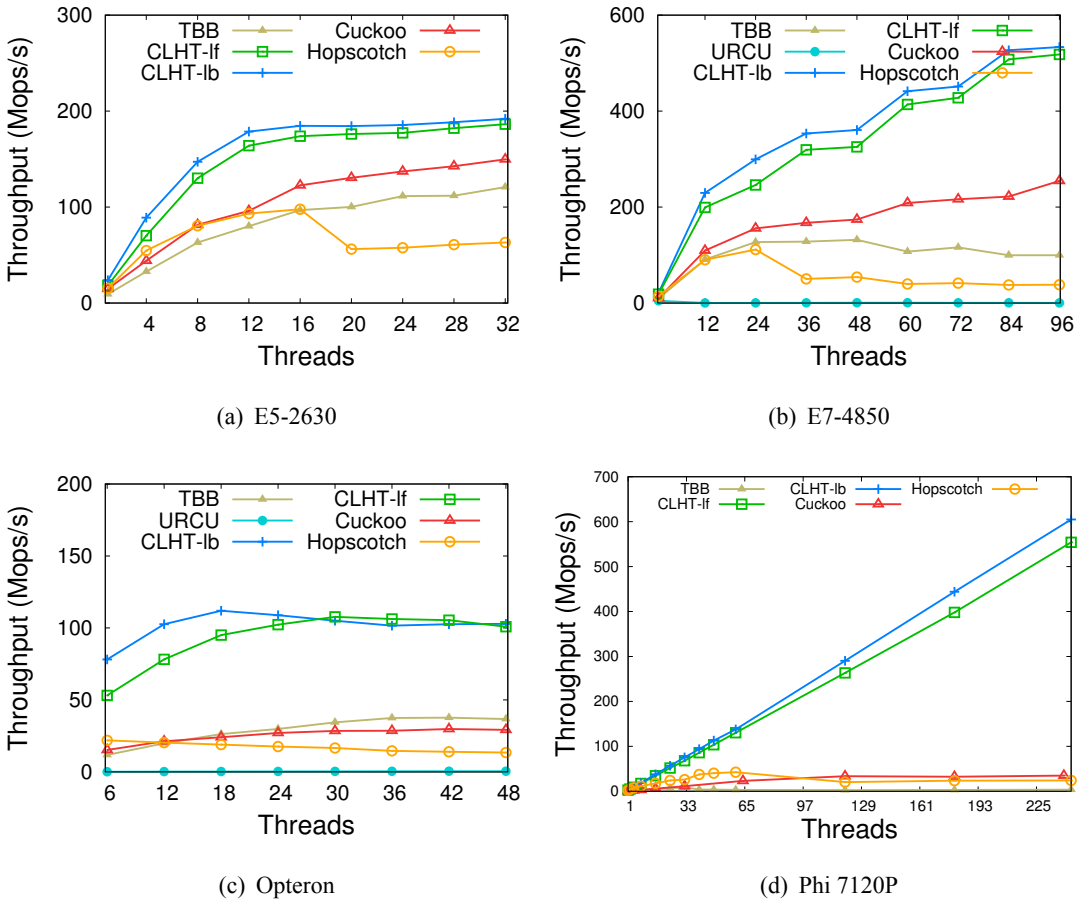
考虑到在图 3.6（a）中当线程数为 16 时存在一处拐点，故将该平台上的性能曲线单独分成两个阶段进行说明。

阶段 1: 在这个阶段创建的线程数量在 1 到 16 这个范围内，超线程没有被启用（紧凑型线程绑定方案）。很明显 CLHT 表现出最佳的吞吐量增长速率，每多创建一个线程，它的吞吐量大约增加 15 Mops/s。CLHT-lb 的性能优于 CLHT-lf 版本。对 Hopscotch 而言，每多创建一个线程，吞吐量大约增加 5.5Mops/s。在该阶段 Cuckoo 的性能比 TBB 要差。

阶段 2: 在这个阶段，创建的线程数量大于 16 个，线程分布在两个不同的 CPU 节点上而且超线程在这个阶段也被启用参与运算。可能受到内存带宽上限的影响，CLHT 的吞吐量保持微弱的增长趋势。表 3.3 列出了在 Intel Xeon E5-2630 平台上 CLHT-lb 的内存带宽随线程数量变化的情况。该表中数据表明在该阶段 CLHT-lb 运行时内存带宽没有发生变化。而 Hopscotch 性能的陡然下降揭示它在处理跨 CPU 节点通信上的性能开销远大于增加额外的线程带来的性能增益。在这个阶段 Cuckoo 和 TBB 展现出不错的增长速度。

表 3.3　CLHT-lb 内存带宽随线程数量变化情况

Threads	1	4	8	12	16	20	24	28	32
内存带宽（单位：GB/s）	1.5	5.8	11.2	12.8	12.8	12.8	12.8	12.8	12.8



(a) E5-2630　(b) E7-4850

(c) Opteron　(d) Phi 7120P

图 3.6　并发哈希表的吞吐量随线程数量的变化曲线，哈希表的密度为 50%，*u* 为 10%，*i* 为一百万。

图 3.6（b）刻画的是 E7-4850 平台上的吞吐量曲线。除 CLHT 之外，其它几种并发哈希表的性能曲线走势与 E5-2630 相似。前面介绍了在 E5-2630 平台上由于受到内存带宽的限制，CLHT 的吞吐量在第二阶段基本没有增长，而在 E7-4850 平台上 CLHT 全程保持稳定的增长趋势。

图 3.6（c）给出的是 AMD 机器上的运行结果。在该机器上 TBB 的线程扩展性在 5 个并发哈希表中仅次于 CLHT。Hopscotch 只有当线程分布在同一个片区内时（n<=6）才具备最佳性能，创建更多的线程并不能提升整体性能。另外，对于 Cuckoo 和 CLHT，当创建的线程数量超过某个特定的值时，会引起吞吐量的下降，这种引起吞吐量下降的原因来自两个方面：一是受到内存带宽的限制；一是来自有限的资源被多个线程激烈竞争。

Xeon Phi 7120P 上的实验结果如图 3.6（d）所示。CLHT 的两个版本在该平台上获得了线性的线程扩展性，每多创建一个线程，相应的吞吐量会增加大约 2.5Mops/s。CLHT 在该平台上获得线性扩展性归因于两个方面：其一，每一个哈希桶的大小等于缓存行的大小，这样大大降低了缓存行切换的次数。并且数据对齐有利于避免多线程伪共享问题，多线程伪共享（false-sharing）是影响 Xeon Phi 性能的主要因素之一。其二，CLHT 采用细粒度锁在一定程度上抑制了线程间的

竞争。然而，并不是所有的并发哈希表在 Phi 7120 上都能表现的如 CLHT 一般呈现线性扩展性，相比于其它几个平台 Cuckoo 和 TBB 在该平台上的性能非常差，创建更多的线程也无法保证性能的提升。详细的原因将在 3.4.3 和3.4.7节中阐述。

(2) 数据分布方式对线程扩展性的影响

在前面的实验中，键值对是用随机方法产生的，服从均匀分布。但是在实际的应用场景下，数据一般不会呈现完美的均匀分布，某些数据出现的频度存在两极分化，因此，人们会更加关注不同的数据分布形式下各哈希表的性能表现。在这一部分内容中，我们将对服从 zipf 分布的数据集进行测试。在本次测试中，参数配置比如哈希表密度，更新比重以及哈希表的初始化元素个数均与线程扩展性一节的参数设置保持一致。图 3.7为运行服从 zipf 分布数据集的性能曲线（在几个平台上的性能变化趋势相一致，故我们只给出了在 E5-2630 上的实验结果）。与图 3.6（a）中的实验结果进行比较，Cuckoo，CLHT，Hopscotch 以及 TBB 的吞吐量分别下降 44%，51%，53% 和 30%。性能下降的原因在于 zipf 分布使得数据访问更加倾向于频度较高的键值对，造成多数的操作集中访问少数频度较高的数据内容。这加剧了多个线程访问相同键的竞争程度，对片上互联通道和同步造成压力从而引发更多的缓存一致性流量。

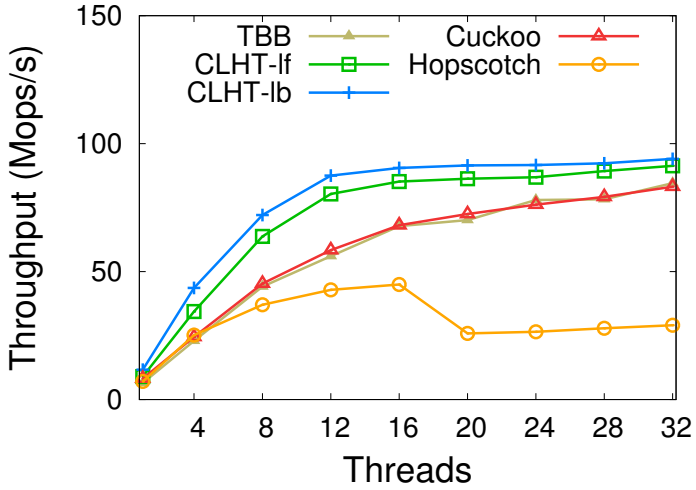


图 3.7 数据服从 zipf 分布时吞吐量随线程变化情况

(3) 哈希表的扩容

不断的往哈希表中插入新的元素会引起哈希表密度的增加，将导致插入元素的时间成本增加。在最坏的情况下，会因为找不到空闲位置并尝试多次之后导致插入操作失败，这无疑会对哈希表的性能造成巨大的损耗。面对这种情形，大部分哈希表在设计的时候允许哈希表在达到一定密度后自行进行扩容。具体的工作流程是，创建一张新的哈希表，新表的容量一般为旧表容量的两倍，然后将旧表中的元素拷贝到新表中，这不可避免的引入额外的时间和空间开销。为了探究并

发哈希表进行扩容时的开销情况，设计了如下一组实验。在本次测试中，由于Hopscotch不支持哈希表的扩容，故没有比较，哈希表的初始化密度设为90%，更新比重调整为40%，其中插入占35%，删除占5%，这样设计的目的在于确保插入表中元素的个数大于被删除的元素个数，从而保证在运行过程中哈希表中元素的密度在不断的增加，增加触发哈希表扩容的概率。实验结果表明，Cuckoo触发扩容操作的概率低于TBB和CLHT，我们认为这得益于Cuckoo在查找cuckoo路径上所做的优化。在触发哈希表扩容操作的情况下，吞吐量大概下降5%左右。

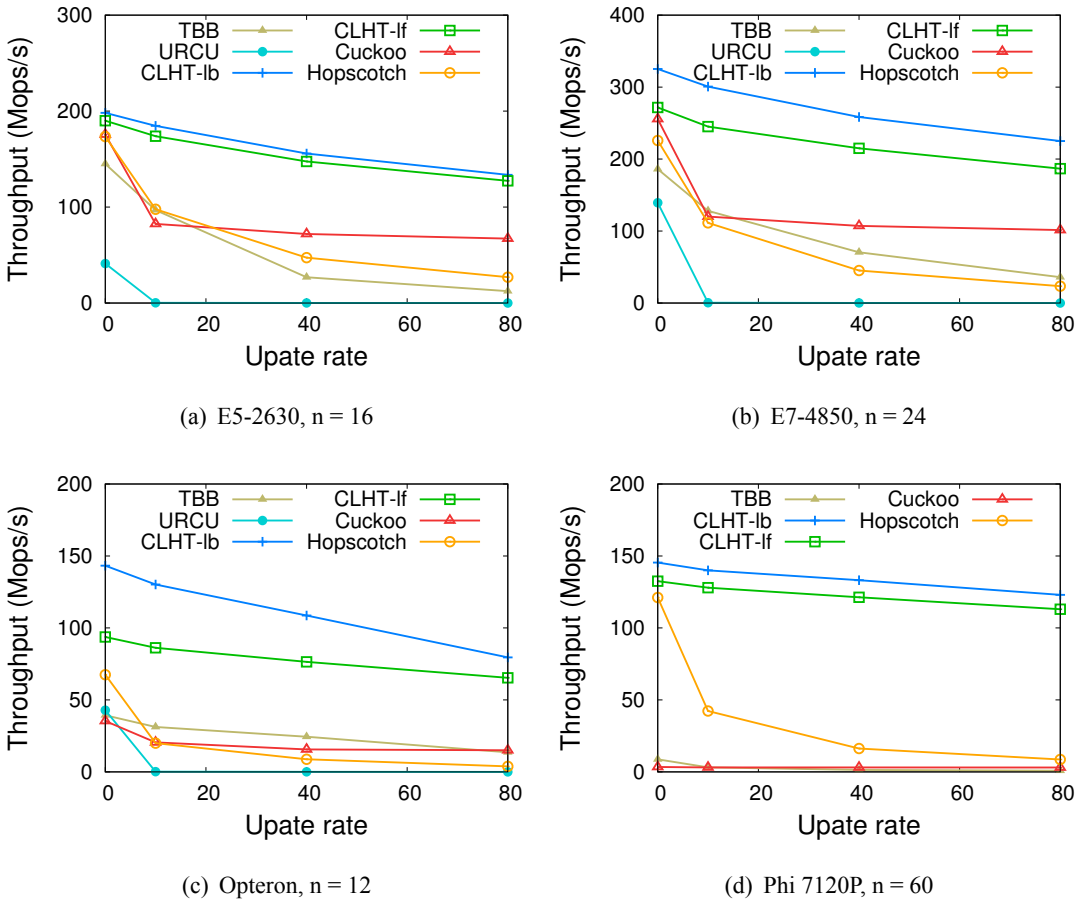
分析：创建更多的线程并不总是意味着吞吐量的增加。一方面，创建更多的线程可能导致内存子系统达到饱和，从而导致性能不再上升甚至将削弱总体性能。另一方面，NUMA系统在缺乏适当仲裁机制的情况下更高的并发度将引起高速互联通道的竞争居高不下，从而达不到最优性能。与传统的多核架构相比，Intel MIC架构在并发哈希的扩展性方面体现出重大差异。在这种新的体系架构上设计并发哈希表需要充分考虑这种体系架构的特征。处理倾向于高频度数据的访问的工作负载将引起更高的缓存一致性流量，这将需要更复杂的同步算法来缓解性能下降的问题。

3.4.3 更新比重对性能的影响

并发哈希表继承了其串行版本的优点：在处理读操作占多数的工作负载上优于具备同类功能的其它数据结构。在本节中，通过调整工作负载中更新操作的比重来观察各个并发哈希表性能的变化情况。哈希表中初始化元素的个数设置为一百万。在三个NUMA架构平台上，选取 n 为平台单个socket支持的最大线程数。Phi 7120P上 n 是选取一个能够展示线程扩展性的值。这样，使用紧凑型线程绑定策略时可以避免跨socket流量的干扰。

如图3.8所示，所有的并发哈希表的吞吐量峰值都出现在更新比重为0处，也就是说并发哈希表对处理只读的工作负载具有最佳性能。但是一旦运行的工作负载中包含了更新操作，性能会有很明显的下降。其中URCU对于更新操作最敏感，更新比重从0调整到10%，URCU的吞吐量下降到原来的1/270。下降幅度大的还有Hopscotch。该现象同样解释了Hopscotch由于很高的同步开销导致的严重的扩展性问题（其它的并发哈希表比如TBB和URCU也有同样的问题）。在更新比重增加时，Cuckoo表现出相对稳定的性能，它的下降速率在几个并发哈希表中是最低的，表明Cuckoo更能应对具有高更新比重的工作负载。

为了更好的说明工作负载中的更新比重发生变化引起性能变化的原因，本次实验引入一个新的微观指标：缓存未命中数与总的操作数量的比值。缓存未命中数用VTune Amplifier进行测量，测量结果为线程数16时的结果，具体见表3.4。通过表3.4的数据表明：工作负载中更新比重越高，每操作对应的缓存未命中数也越高。CLHT在低更新比重和高更新比重下都表现良好，这从表3.4中也可以得



(a) E5-2630, n = 16

(b) E7-4850, n = 24

(c) Opteron, n = 12

(d) Phi 7120P, n = 60

图 3.8　哈希表初始化元素个数一定的前提下，吞吐量随更新比重变化趋势

到体现，CLHT 对应的最大值与最小值之间的差距只有 30%。而 URCU 对应的值从 0 到 10% 这个阶段扩大了 14 倍，从 0 到 80% 更是扩大了 127 倍之多。糟糕的情况更不止如此，URCU 的 CPU 利用率非常低，大约保持在 3% 左右，而同等参数配置下其它并发哈希表的 CPU 利用率达到 50%。进一步探究发现，创建更多的线程并不会引起缓存未命中数量的增加，这表明工作负载中更新比重的变化是引起吞吐量变化的主要原因。

表 3.4　平均缓存未命中数量随更新比重变化情况

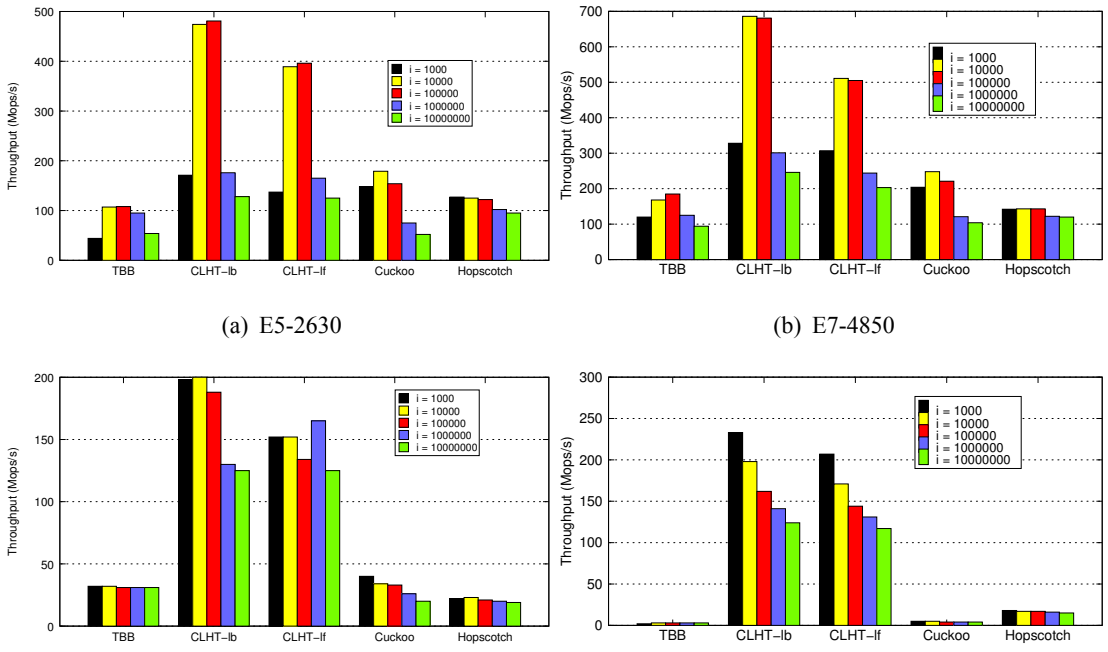
更新比重（%）	TBB	URCU	CLHT-lf	CLHT-lb	Cuckoo	Hopscotch
0	28.6	85.2	23.6	31.7	33.5	22.9
10	46.3	1169.2	24.6	34.9	44.2	28.1
40	99.2	4046	27	37.3	57.2	62.7
80	159.1	10788	30.7	40.6	66.2	115.5

分析：频繁的缓存行切换是并发哈希表更新性能最大的敌人。更新操作致使缓存行失效的原因来自于两方面：一方面是写入本地内存节点的流量；另一方面

是被缓存一致性协议强制进行的跨 socket 消息，比如通过片上高速互联通道如 Intel QPI 和 AMD HyperTransport 等传递的信息。一些设计用于处理读为主的工作负载的并发哈希表，哪怕是工作负载中包含很小部分的更新操作，它的性能也会大打折扣。写友好型并发哈希表通常会进行精心设计用以控制缓存内的关键数据，比如共享变量等。

3.4.4 缓存与主存

考虑到哈希表属于内存密集型应用，所以在这一部分我们探究内存分层结构对并发哈希表性能的影响。图 3.9中的柱状图表示吞吐量随着哈希表初始化元素个数的变化情况。可以观察到吞吐量会随着初始化元素的个数有较大的波动，具体表现是：在三个传统的多核平台上，吞吐量呈现先增后减的趋势；而在众核机器上，吞吐量随着初始化元素个数的增加而减少。在多核机器上产生这种现象的原因是初始化元素越多，所需的内存空间越大，当初始化元素较少时，所需的内存空间较少，可以一次性被缓存所容纳（以 E5-2630 为例，初始化一万个元素所需的内存空间约为 15 MB，而该机器的 LLC 的容量为 20 MB），而当初始化元素较多时，所需的内存空间远超系统缓存容量，在进行数据读取/写入操作时具有较高的延迟。



(a) E5-2630　(b) E7-4850

(c) Opteron　(d) Phi 7120P

图 3.9　不同初始化大小对应的吞吐量抽样直方图

从图 3.9 (a)-(c) 中观察到，当工作集的规模小于缓存的容量时，吞吐量是随着初始化元素个数的增加而增加的。我们通过 likwid-perfctr 这个工具对最后一级缓存与主存之间的带宽与流量进行监测确认了这一结果。一旦工作集的规模超过最

后一级缓存的容量，缓存未命中率、访问延迟等都升高。此外，CLHT 由于其在缓存使用机制上的独特设计，它的工作集规模小于缓存容量时的性能要比其它方法更好。通过本次实验，我们还观察到一个有意思的现象（图 3.9中没有体现该现象），当被创建的线程个数增加时，在一级缓存中存在显著的数据竞争情况。具体情况是，当初始化元素个数为 1000 时，并发哈希的吞吐量随着线程数量的增加而降低。这个现象的原因在于线程越多，不同线程同时访问相同的共享数据的概率越高，从而导致较高的同步开销。

Hopscotch 在本次实验中属于例外情况，它的吞吐量几乎不随初始化元素个数的变化而变化。进一步的探究发现在 Hopscotch 的实现中，哈希表的内存是通过预先开辟固定大小的内存空间建立的。这种方法的好处是获得了性能的稳定，缺点是丧失灵活性，在处理较小规模的工作负载时对内存空间的浪费较严重。

对高并发度的应用而言，无论是基于锁还是无锁的并发哈希表的实现，都依赖于有效的内存管理机制。而这些内存管理机制通常是建立在第三方程序库的基础之上，或者来自由系统提供的动态内存分配器。需要说明的是，有些并发哈希表在试图运行更大的工作负载（初始化值大于 1 亿）时失败了这也可能是由内存管理上的缺陷造成的。

前文中指出，Phi 7120P 的内存分层结构与其它主流多核体系架构存在较大差异。它只包含两级缓存，并且其核心与内存控制器之间是通过双向环状总线进行连接的。图 3.9 （d）为 Phi 7120P 上的实验结果，在该平台上 CLHT 的吞吐量随着初始化元素个数的增加而降低，而其它几种哈希表因为数值太小在图中无法明确的体现这一趋势，通过比较实验数据发现同样符合这个趋势。

除了图 3.9 中展示的数据规模之外，还对运行超过 GB 级别的数据进行了测试。性能曲线与初始化值为一百万时的同类测试类似，只是在吞吐量上要打折扣。

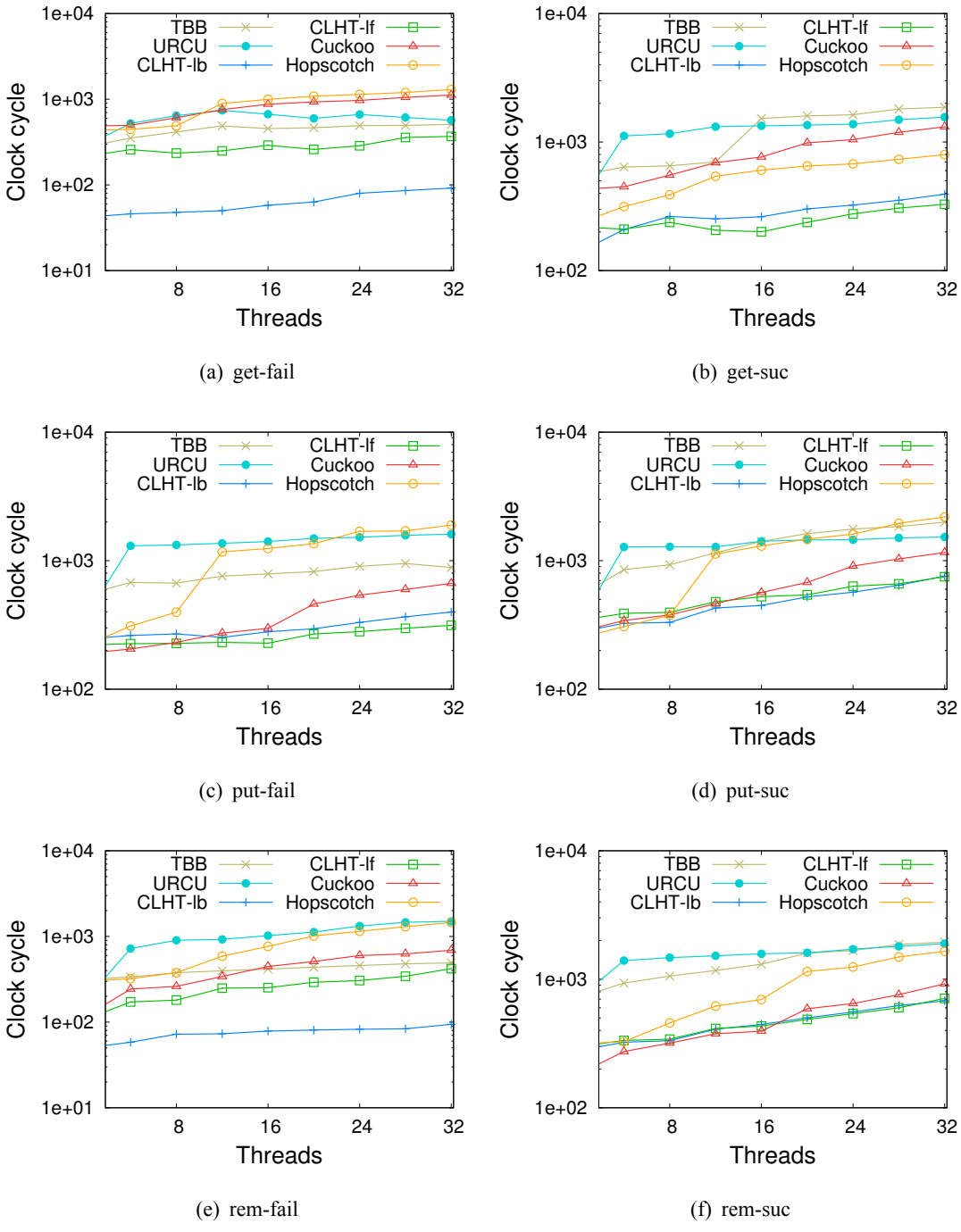
分析：缓存对加速基于多核系统的并发哈希表的性能具有非常重要的作用。对缓存采用细粒度的方法进行控制的目的在于获得可预测的结果，比如为常驻缓存的工作集进行线程调度有利于避免额外的同步开销。有效利用缓存行替换机制有助于提高缓存命中率。静态内存分配的方式对于并发哈希这种内存密集型应用来说并不是最佳选择，而对超大规模数据集使用动态内存分配方式需要进一步的优化方案予以辅助。

3.4.5 操作延迟

前文从宏观的吞吐量出发对并发哈希表进行了评估，在这一节当中我们从微观的每个操作执行的运行时延迟的角度进行分析。对于延迟敏感型应用而言，理解不同算法设计的延迟变化造成的影响至关重要。

本次实验中，更新比重设置为 10%，哈希表密度为 0.5，哈希表的初始化元素个数为一百万。通过实验收集哈希表操作所耗费的 CPU 时钟周期数进行统计。我

们对哈希表的操作进行如下定义：如果某个操作完成了对其预期目标对象的访问，我们称该次操作为一次成功的操作；否则我们称之为一次失败的操作。根据如上定义，得到六种不同的操作类型，它们分别记做：get-suc，get-fail，put-suc，put-fail，rem-suc 和 rem-fail。所需的时钟周期数使用开源工具 sspfd[121] 收集，然后通过均匀抽样方法计算每种操作对应的平均延迟。



(a) get-fail　(b) get-suc

(c) put-fail　(d) put-suc

(e) rem-fail　(f) rem-suc

图 3.10　E5-2630 平台上 6 种操作对应的延迟随线程数量变化曲线

图 3.10给出了我们在 E5-2630 平台上的实验结果。随着所创建的线程数量的增加，资源的竞争（包括内存带宽，缓存以及内存控制器）和同步开销逐步增加，

这些都将不可避免的导致延迟的增加。通过比较发现，CLHT（两个版本）在大部分情况下的延迟都低于其它并发哈希方法，这归因于它在设计上严格遵循四条异步并发模式。过多的缓存行切换需要更周密的缓存一致性协议予以支持，从而导致哈希表操作延迟的增加。在几种并发哈希表中，URCU 的操作延迟最高，原因是 URCU 在设计上有一个称为“RCU 优雅时间”的等待期。Hopscotch 的延迟对跨 socket 通信相当敏感（对应图中线程数量大于 8 时的曲线变化）。并且在 Hopscotch 进行删除操作时它的时间戳会被修改，用以确保同一时刻对该内存地址的查询操作失效，这一点类似于 CLHT 的原子快照方法。Hopscotch 和 CLHT 的不同之处在于 Hopscotch 需要存储共享变量，这将引起额外的缓存一致性开销。另外，因为 Hopscotch 的锁在一进入解析阶段就已经获得，所以 Hopscotch 的更新操作对应的解析阶段包含了等待过程。这些设计都违背了异步并发模式的第二条原则：“除非对数据结构进行清零操作，否则在更新操作的解析阶段不要执行任何写操作，也不要有任何的等待过程或者重试操作。”在最坏的情况下，Hopscotch 的延迟比 URCU 的延迟还要高（图 3.10（c）和（d））。

Cuckoo 从直观的计数器的数据来看，它的搜索操作耗费的平均时钟周期比更新操作的更高。出现这种现象的原因是 Cuckoo 需要频繁的从较长的 cuckoo 路径中搜索元素。在并发度较低的情况下，Cuckoo 处理 put-fail 和 rem-suc 两种操作的延迟较低。同样在线程数低于 6 的情况下，Hopscotch 处理 put-suc 操作的延迟较低。TBB 在处理 fail 类型的操作时的延迟相对稳定，但是在处理 suc 类型操作时与 URCU 持平，甚至在某些情况下比 URCU 表现更糟糕。

分析： David 等人在其研究中提出的异步并发 (ASCY) 模式有助于获得良好的扩展性[12]。这一点在 CLHT 上得到很好的体现。在实际应用中，ASCY 的特定模式能够帮助开发人员在实现并发哈希表时避开潜在的陷阱。比如，在对 Hopscotch 进行分析时，我们发现其在进行删除操作时对共享变量所做的修改，以及在更新操作的解析阶段的等待都与 ASCY 的原则相悖。将删除操作替换成其它类型的操作进行测试发现其吞吐量有明显的增加。

3.4.6 线程绑定方案的影响

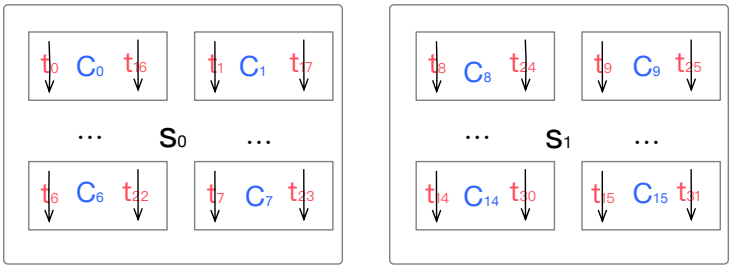


图 3.11 E5-2630 平台上使用平衡型线程绑定方案时线程与核的映射拓扑结构

在多核计算机系统上线程到核之间的映射关系称为线程绑定。线程绑定对于

并发编程至关重要[124]。在这一部分内容中，将以 E5-2630 为例，探究三种不同的线程绑定方案在四个多（众）核系统上的表现有何区别。图 3.11所示为 E5-2630 平台上使用平衡型线程绑定方案时线程与核的映射拓扑结构。该拓扑结构使用开源工具 *likwid*[125] 的 *likwid-topology* 得到。该平台具有两个 socket，每个 socket 集成了八个物理核，每个物理核最多支持两个硬件线程。

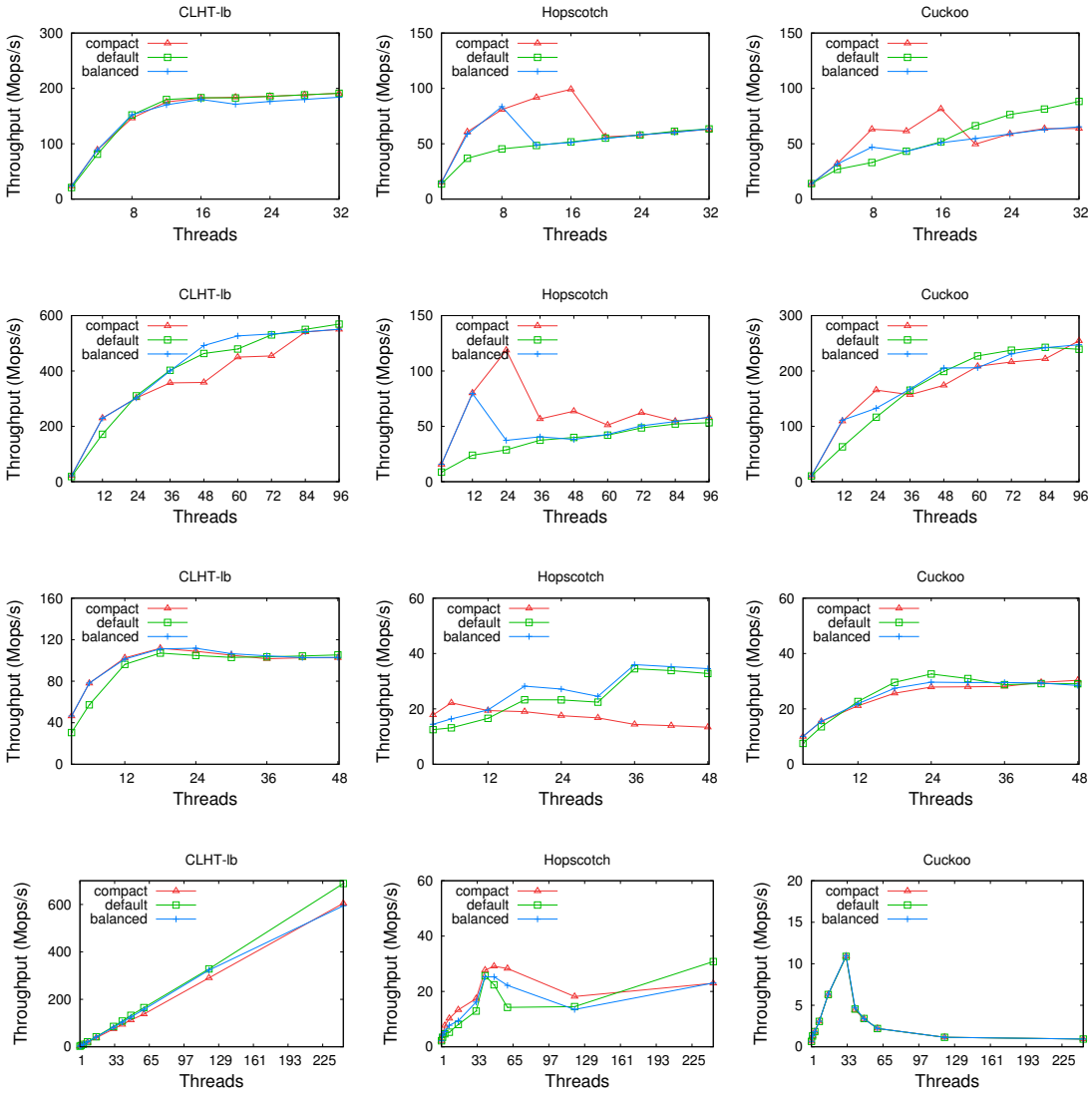


图 3.12　三种线程绑定方式性能对比

在进行实验分析前，首先简要介绍实验中用到的三种不同方案：默认绑定方式和显示绑定方式，其中显示绑定方式又分为紧凑型绑定和平衡型绑定两种。

默认方式：线程通过操作系统调度器自动进行线程与核心的关联。操作系统调度器尽可能地实现多个核心之间的负载均衡。这种绑定策略的好处是允许线程在执行过程中在不同的核心之间迁移。

紧凑型：尽可能的将连续的线程映射在拓扑结构上距离最近的核心上。如果需要创建 24 个线程，根据紧凑型绑定策略首先将第一个 socket 内的核心映射满，线程 t_0 到 t_{15} 被映射到 s_0 内的 c_0 到 c_7 上，然后余下的线程将映射到位于 s_1 上的

c_8 到 c_{11} 上。这种绑定方式的好处是在线程间提供较高的数据重用率。

平衡型：紧凑型线程绑定方案具有在单个 socket 内共享缓存的优势，它的劣势在于紧凑型的线程绑定容易引起负载不均衡。因此，平衡型绑定策略将线程均衡地绑定到位于不同 socket 的核心上。下面用一个实验测试中的实例进行说明：假设运行某次测试需要创建 16 个线程参与运算，正好可以将 16 个线程分别映射到 c_0 到 c_{15}；如果需要创建的线程数量超过了系统核心的数量，这时才会启用超线程。使用平衡型绑定策略的好处有两方面：一是保持负载均衡；二是，在创建的线程数量少于系统具有的物理核心的数量时，可以避免启用超线程产生的干扰。

CLHT 的吞吐量受线程绑定方式影响较小，但 Hopscotch 和 Cuckoo 的吞吐量的波动却很大。具体的以 E5-2630 为例，在该平台上以默认方式的性能曲线为基准，线程数量为 8 时，使用紧凑型和平衡型绑定策略性能提升了大约一倍。进一步分析，在紧凑型线程绑定策略下，线程数达到 16 时，吞吐量比采用默认绑定方式提升了 2 倍。在 E7-4850 机器上体现了相同的趋势，故不再单独分析。

Hopscotch 吞吐量显著的加速比来自于它的静态内存分配机制。它在创建哈希表的同时就已经预先进行了内存分配，受 Linux 内核 first touch 内存分配策略的控制，预先分配的后果是导致所分配到的内存全部位于同一个 NUMA 结点上。所以，我们使用 libnuma 对其它并发哈希表进行优化，使其在进行动态内存分配时均衡的兼顾所有 NUMA 架构下的内存节点。然而，当线程数从 16 向 20 变化（E5-2630），从 24 向 36 变化（E7-4850）过程中，吞吐量大幅度下跌，这说明跨 socket 通信开销对性能的副作用远大于此阶段增加参与运算的线程数量带来的性能增益。

而在 AMD 平台上，各哈希表的性能表现对于线程绑定方式不敏感，这个现象可以从两个方面进行解释。首先，Intel Xeon 最后一级缓存的 inclusive 特性提供了较强的局部性，这种局部性提高了 socket 内通信的效率。AMD Opteron 的不完整目录协议也会使跨 socket 间的流量失效。因此，socket 内的性能与跨 socket 情况下的性能基本一致。由于负载的均衡分布使得该平台对跨 socket 通信不敏感，所以当线程分布在多个 socket 上时，默认绑定方法和平衡型绑定方法的性能要优于紧凑型绑定方式。

在 Phi 7120P 平台上，CLHT 无论采用哪种绑定方式，都具有极为相近的性能曲线（吞吐量随着所创线程数量的增加呈线性增长）。而 Cuckoo 采用三种不同绑定方式所获得的吞吐量不存在显著差别（三条性能曲线基本重合）。这一点与之前在其它三个 NUMA 平台上观察到的结果存在较大差别，在 NUMA 平台上 Cuckoo 采用默认绑定方式具有最佳性能。当线程数在［1，150］这个区间变动时，Hopscotch 在 Phi 平台上使用紧凑型绑定策略获得最佳性能。然而，在线程数超过 150 个之后，CLHT 和 Hopscotch 在这个阶段使用默认绑定策略的性能要优于其它

两种策略。使用显示线程绑定策略的缺点是：受到二级缓存容量的限制，会造成更多的缓存未命和过高的内存访问延迟。而在 Xeon Phi 上，默认绑定方式由于依赖于操作系统的调度器进行线程绑定并且允许线程在不同的核之间进行动态迁移，能够有效地降低资源竞争并且更好的利用缓存子系统的特性。另外，Xeon Phi 的表现类似于对成多处理机（SMP）系统，所有核心到主存储器之间的距离是相等的。将线程分配在不同的物理核上引起的开销类似于 NUMA 系统上跨 socket 的开销，但由于 Phi 上快速双向环状互联通道的设计使得跨核开销要远小于跨 socket 开销。

分析：实验表明没有哪一种绑定策略能在所测试的多个硬件平台上具有压倒性优势，也没有哪一种策略能够让所有的并发哈希表发挥最佳性能，所以在比较三种不同的线程绑定方案过程中没有一个统一的结论。比如，在 Intel Phi 平台上，CLHT 使用默认绑定方式具有最佳性能，但是在其它平台上则是其它绑定方式下具有最佳性能。再就是 Cuckoo 在 Phi 平台上使用三种不同绑定方案的性能相当接近，没有哪一种具有明显的优势。因此，要探究并发哈希表的异常表现，需要结合并发哈希表的设计模式，线程绑定方案，创建的线程数量以及对应的硬件平台的特征进行考虑。这对开发人员无疑是一种负担，最理想的解决方案是在运行过程中通过对硬件特征和工作负载的监控进行自适应的动态切换线程绑定策略，以达到获得最佳性能的目的。

3.4.7 同步

在并发读写的过程中，为了保障性能，多个线程对共享数据的访问需要用到同步方案对并发线程进行协调。因此，设计并发哈希表时的另一个重要环节是同步方式的选取。对线程间的同步处理不当会成为阻碍扩展性的最大障碍。在这一部分中，将讨论并发哈希表中几种主要的同步机制以及运用不同的同步机制对性能会产生什么样的影响。

回顾图 3.8的性能曲线，几种并发哈希表中 CLHT 的性能最突出，尤其是在 Phi 7120P 平台上体现的最为明显。CLHT 获得突出线程扩展性的原因除了其在数据结构设计上的巧妙之处外，它所采用的同步机制的贡献也不容忽视。CLHT-lb 使用原子快照对读者线程和写者线程进行同步。具体地，在读者线程间，它使用由原子操作 FAI（Fetch-and-Increment）实现的自旋锁对每个哈希桶进行保护。采用这种方式的好处是，即便在并发度很高的情况下，竞争仍然保持一个较低的水平。简单锁方法加上低竞争有利于改善扩展性[18]。无锁的 CLHT 版本同样使用快照的方式进行同步，该快照的大小为 8 字节，它能在单个操作内被读取、存储或者 CAS（Compare-and-Swap）。在哈希桶内有一个版本计数器用于同步并发写线程，同时还有一张表示有效、失效或者正在被执行插入操作三种状态的位图。以上就是 CLHT 获得高性能的原因。

在处理只包含读操作的工作负载时，Hopscotch 的性能要优于 CLHT。而一旦工作负载中包含了更新操作，Hopscotch 的性能就会显著地下降。并且这种下降幅度随着工作负载内更新操作比例的增加而增大。这种现象也说明选取同步方案的重要性。Hopscotch 使用 TTAS 锁对写线程进行协调。写线程之间的同步开销要高于读线程间的开销。为了减轻同步开销，在写线程和读线程之间使用时间戳。Hopscotch 在设计上特意使锁的数量等同于所创建的线程的数量，这与 CLHT 相比无疑会引起更高的竞争。更糟糕的是，Hopscotch 使用的 TTAS 锁本身的扩展性也存在问题。以上对 Hopscotch 严重的性能下降现象就同步方面作出了解释。

Cuckoo 的实现也是基于细粒度锁。它使用锁对读－写和写－写两种访问模式进行同步。Cuckoo 使用的是 striped spinlock，锁的实现使用 CAS（Compare-and-Swap）原语。TBB 使用细粒度锁对每个哈希桶进行保护，这一点与基于锁的 CLHT 的设计思想相似，但二者的区别是 TBB 每个哈希桶内包含的键/值对的数量要远多于 CLHT。

如图 3.6（d）和图 3.8（d）所示，并发哈希表在 Xeon Phi 平台上性能曲线与其它三个 NUMA 平台存在明显差异。Xeon Phi 使用的是扩展的 MESI 缓存一致性协议，该协议用 GOLS（Globally Owned Locally Shared）模拟共享状态以允许对处于修改状态的缓存行实现共享，对处于 GOLS 状态的缓存行进行写操作同样会引起无效的通信。过高的 coherence traffic 很容易使 Phi 的环状互联通道达到饱和。从实验结果可以推断，CLHT 的设计同样很好地适用于 Xeon Phi 的体系架构。而 Cuckoo 在该同台上的性能很差，即便是处理只读的工作负载，其性能也不理想。其原因在于每一次查询操作，与给定哈希值相关联的哈希桶都被上了锁，这严重的限制了并发度，从而影响了其在该平台上的性能。

分析：同步在并发哈希表的性能中扮演者至关重要的角色。然而，并不存在一种放之四海兼准的通用的同步方案，设计一种高效的同步机制需要综合考虑来自底层原语到体系结构特征再到高层锁实现乃至并发模型的影响。正如我们通过实验分析的那样在 MIC 架构平台上，除了 CLHT 之外，其它并发哈希表反常的低性能表明获得可移植的高性能任重道远。在实践当中，判断同步机制是否合适，需要全面的理解上述因素的多方面影响。例如，在设计锁算法时，需要知道在特定的平台上怎样选择更合适的原子原语，而这又取决于缓存一致性，以获得更好的性能。更进一步说，有些锁在高竞争环境下表现良好，而有些锁在低竞争情形下更具有竞争力。因此，实现自适应锁来利用不同锁算法的优点将是一项有价值的研究工作。

3.4.8 内存消耗

在物理内存有限的场景下，对于 CHT 这种内存密集型应用而言，除了对于性能方面的需求之外，另一个重要的指标是内存的消耗。在运行时间一定的情况下，

获得相同的吞吐量所消耗的内存越小，该应用就越占据优势。下面对四种动态分配内存的并发哈希表的内存使用情况进行分析。

表 3.5 给出各个并发哈希表运行时内存使用情况，创建 16 个线程用于运行含有 10% 更新操作的数据集，数据集初始化元素个数从 10^3 到 10^8 变化，单位为 MB。运行时内存通过 Linux 系统的系统监测工具测量。在初始化元素个数相同的前提下，Cuckoo 在处理规模较大的工作集时内存效率更高，URCU 仅次于 Cuckoo，并且处理小规模数据集时具有更高的内存效率。CLHT 和 TBB 在同等规模数据集下所消耗的内存是 Cuckoo 和 URCU 的几倍。

表 3.5　E5-2630 平台上的内存使用情况

i	TBB	CLHT	URCU	Cuckoo	Hopscotch
10^3	0.6	1	**0.6**	63	608.6
10^4	2.2	15.4	**1.8**	63	608.6
10^5	14.1	34.4	**7.8**	63	608.6
10^6	80	78.4	**43.9**	101	608.6
10^7	857	1024	645	**633**	608.6
10^8	5.5 GiB	8 GiB	5 GiB	**2.3** GiB	

产生这种差异的原因是什么呢？接下来的内容将从数据结构的设计的角度对这个问题进行说明。

Cuckoo 内存效率高的原因来自两个方面。首先，Cuckoo 组相联的设计节约了空间，提高了空间使用效率。其次，使用版本计数器代替指针连接其它哈希桶，在处理键值对很小的条目时具有极高的内存效率。CLHT 和 TBB 都使用细粒度锁对每个哈希桶内的键值对进行保护。这种机制的的好处是处理速度快，但是在处理大规模键值对的工作集时，指针将消耗大量的内存。所以，使用细粒度锁机制是一种通过牺牲空间换取性能的优化方式。

分析：一方面，相同配置下较低的内存开销的应用能够处理更大规模的工作集，并且降低了操作系统发生页切换的概率，降低页切换概率有利于降低来自平台对性能的影响。另一方面，内存效率依赖于内部的数据管理机制，而数据管理机制的选择也会在一定程度上对性能造成影响。链式结构的并发哈希表需要额外的指针对哈希桶进行链接，尤其是以 CLHT 最为突出，它的哈希桶的容量被设计成与缓存行的大小一致，这就需要在每个缓存行都预留一个指针以便与其它的哈希桶进行链接，而 Cuckoo 的组相联的设计减少了这种指针的使用，降低了额外的内存开销。另外，锁的粒度越细，所消耗的内存空间也越多。总而言之，对数据的组织结构和同步的粒度进行优化有助于提高性能，其代价是更高的内存开销。

3.5 本章小结

并发哈希表在现代多核软件系统中占据至关重要的地位。为了解决特定硬件环境与错综复杂的算法设计中的实际问题，相关研究人员提出了一系列的并发哈希表的设计方法。用户如果需要从众多的相关算法中选择一个适合自己应用的算法就需要深入透彻的去了解现有并发哈希表的优缺点。然而，事实并非如此，并发哈希表要么是根据特定的应用软件设计的，要么是针对特定的硬件平台进行优化，缺乏统一的评测基准。这对并发哈希表的设计、优化和应用都造成了不便。

基于上述考虑，首先设计一套用于测试评估并发哈希表的统一的测试框架（CHTBench），为并发哈希表的评估提供一个相对公平的环境，避开因工作负载、键值对生成方式等因素的干扰。随后使用 CHTBench 对现有文献中影响力较高的 5 种并发哈希表进行测试评估。评估和比较的指标辐射面从宏观的吞吐量到微观的延迟，从片上缓存到主存，从复杂的同步机制再到可移植优化等，并在每一项指标的评估之后做了针对性的分析。

本章对并发哈希表的比较和评估工作是迄今为止覆盖面最广，指标最多，最有深度的。文中设计 CHTBench 和测试方法对并发哈希将来的设计研究和应用推广具有重要价值。

第 4 章　基于硬件事务内存的缓存行哈希表的实现

4.1　事务内存的基本概念

事务内存（TM）是一种简化并发编程的并发控制范式，其源自于数据库管理系统中的事务概念。它的核心思想是将一段代码标记为一条事务。在数据库管理系统中，事务必须满足 ACID 性质，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。原子性指的是事务中的动作要么全部执行，要么一个都不执行；一致性指的是任何时刻，数据库必须处于一致性状态，即必须满足某些预先设定的条件；隔离性是指一个事务不能看见其它未提交事务所涉及到的内部对象的状态；而持久性则是指一个已提交的事务对数据库系统的改变必须是永久的。

事务内存继承了事务的 ACID 性质。事务内存的原子性是指：事务代码区间的内容要么全部执行，要么全部不执行，不存在事务代码停滞在中间的任何一条语句，如果发生意外而中止了事务的执行，则系统状态会回滚到事务开始执行时的状态继续执行；事务内存的一致性是指是指：事务要么执行完成并使外界看到其造成的系统状态的变化，要么执行失败并使所有相关状态保持不变。如果有多个事务同时运行，那么从这些事务之外的角度来进行观察，它们对系统状态做出的改变始终是一个接着一个发生的，中间不会有任何交叉。例如，在（对同一个账户）两个独立且并发的存款和取款事务完成之后，账户余额应该是两个操作所产生的累加效果（取钱是对账户加上一个负数）。事务内存的隔离性是指：并发执行的多个事务代码区间彼此之间是严格隔离的，本事务无法了解其他事务的局部变更结果，所有事务造成系统状态的变化只有在成功提交之后才对外部可见；事务内存的持久性是指事务代码执行完成并且成功提交之后，对于系统状态的变更是持久有效的，并不会回退到起始状态，直到有新的事务执行的代码改变这个状态。

事务内存具有两个重要的性质，即事务内存要确保对临界区代码片段的执行具有原子性和隔离性。满足原子性和隔离性的事务内存可以安全的并行执行，可以取代现有的令人头疼的锁机制与原子原语。基于锁的并发编程是一种悲观的并发模式，它假设获得锁的线程一定会访问共享数据，从而会造成其他线程的阻塞。而任意两条访问受锁保护的变量的事务能够并发的执行，并且只有在其中一条事务尝试修改共享数据时才会触发回滚机制。

事务内存被认为是最有希望解决多核处理器编程问题的并发编程方案之一。

它最吸引人的特点是程序员只需在本地对共享数据的访问行为进行预估，并让底层系统确保正确的并发执行。该模型有望提供细粒度锁机制那样的线程扩展性并能避免一般的锁机制中常见的陷阱，比如死锁等问题。

本章主要研究硬件事务内存对构建并发哈希表的作用。

4.2 Intel 事务同步扩展

2012 年 2 月，Intel 发布了其支持硬件事务内存的事务同步扩展 (Transactional Synchronization Extensions, TSX) 指令集，并在其 2013 年 6 月发布的 Haswell 微体系结构的微处理器上实现了对 TSX 的支持。标志着硬件事务内存的全面商用化。

TSX 对使用锁省略方式编写的多线程并发软件具有明显的加速作用。TSX 提供一组扩展指令集，允许编程人员指定事务同步的代码区域。有了事务同步，硬件就能动态的决定是否需要将进入临界区的线程串行化执行。这个特性有助于提升程序的并发性。在 Intel TSX 的最底层，编程人员指定的代码区域（也称为事务区域）可以事务性地执行。只有确认事务成功提交后，在执行事务区域期间对于系统状态的变更操作才会对其他逻辑核可见。完成一次事务执行并提交状态变更信息的过程称为一次原子提交。这样，编程人员只需使用粗粒度的锁编程就能实现细粒度锁方法才能获得的性能。如果多个线程同时访问受同一个锁保护的临界区而彼此之间又不存在任何数据冲突，那么这些线程就能并发的执行。

Intel TSX 提供两种不同的指令前缀：一种称为硬件锁省略 (Hardware Lock Elision，HLE)；一种称为限制性事务内存 (Restricted Transactional Memory，RTM)。

4.2.1 Intel 硬件锁省略

HLE 提供与传统指令集兼容的指令集接口供编程人员编写事务化的程序，还提供两条全新的指令前缀 XACQUIRE 和 XRELEASE。

使用 HLE 时，将 XACQUIRE 前缀放置在用于获取保护临界区的锁的指令之前。处理器将 XACQUIRE 视为省略写相关的锁获取操作的提示。即使是获取锁之后在该锁对应的临界区内执行写相应的操作，处理器将锁的地址添加到事务区域的读取集（read set）中，不会向该锁发起任何写入请求。之后逻辑核进入事务执行状态。如果锁在 XACQUIRE 前缀指令之前是可用的，那么后面的其他所有的核都将该锁视作是可用的。因为当前正在事务化执行的逻辑核不会将锁的地址写入到它的写入集（write set）中，也不对其执行外部可见的写操作，其他的逻辑核仍然可以读取该锁而不会造成数据冲突。这就允许其他的逻辑核也进入临界区，并发的对该临界区进行操作。处理器将自动检测事务执行期间发生的任何数据冲突，有必要时中止事务操作。

XRELEASE 前缀放置在用于释放保护临界区的锁的指令的前面。这涉及到对

锁的写的问题。如果 XRELEASE 正在试图将锁的值恢复到 XACQUIRE 发起锁请求时所操作的同一个锁的值，那么处理器将省略所有与即将释放的锁相关的外部锁请求，并且该锁的地址不会被添加到写入集中。然后处理器尝试提交事务执行结果。

使用 HLE 时，有多个线程同时执行到被同一个锁保护的临界区时，如果它们执行的操作不会相互之间产生数据冲突的话，这些线程可以并发执行。如果无法事务地执行事务代码区域，处理器将不使用省略而以普通的方式执行代码区域。HLE 确保软件有与基于传统锁方法相同的前向执行的能力。即在不支持 HLE 的硬件平台上，使用 HLE 指令前缀的程序也能得到正确的执行结果。不支持 HLE 的硬件会忽略掉 XACQUIRE 和 XRELEASE 前缀提示，也不会执行任何省略操作。更为重要的是，HLE 很好的与现有的基于锁的编程模型兼容。在基于锁的编程模型中不正确的使用 XACQUIRE 和 XRELEASE 前缀提示不会引起功能错误，但它可能会暴露代码中已经存在的潜在的漏洞。对于一次成功的 HLE 执行，锁和临界区代码都必须遵循特定的指导原则[98]。

4.2.2 Intel 限制性事务内存

RTM 为事务的执行提供了更为灵活的接口。它提供三条新的指令——XBEGIN，XEND 和 XABORT——供用户启动，提交和中止一次事务执行。

XBEGIN 指令用于指定事务代码区域的起始位置，XEND 指令表示事务代码区域的结束。XBEGIN 指令携带一个操作数，该操作数用于表示它到回滚指令地址的相对偏移量。如果 RTM 代码区域无法成功的事务化执行，将回退到该操作数指向的地址继续执行。

处理器有很多原因中止 RTM 的事务执行。硬件自动检测事务中止条件，退回到回滚指令处，从 XBEGIN 开始的指令时的体系结构状态和 EAX 寄存器更新的描述事务中止的状态重新开始执行。

XABORT 指令允许用户显示的中止 RTM 代码区域的执行。它携带一个 8 位的直接参数，这个参数被加载到 EAX 寄存器中并将在发生 RTM 中止后对软件可见。具体参数的意义见表 4.1 。

RTM 指令没有任何与它们相关联的数据存储位置。虽然硬件无法保证 RTM 区域总是能够成功地提交事务，但大多数遵循 Intel 技术手册中的指导原则的事务都有可能成功地提交。使用 RTM 进行编程时，用户必须设置回退路径，用以确保在事务中止后程序的前向执行。回退路径的设置可以是传统的锁方法或非阻塞方法，比如，通过简单的锁申请然后以非事务的方式执行指定的代码区域。此外，在特定的实现中发生中止的事务可能会在接下来的实现中完成事务的执行。因此，用户必须确保事务性区域代码的路径和回退路径上的代码序列在功能上是完备的。

表 4.1 RTM 的中止状态定义

EAX 寄存器比特位	说明
0	由 XABORT 指令引起的中止
1	该事务有可能在 Retry 之后成功
2	该次中止是因数据冲突而引起的
3	该次中止是因内部的缓存溢出引起的
4	该次中止是因调试断点引起的
5	该次中止是因事务嵌套引起的
23:6	保留位
31:24	XABORT 参数（只有位 0 被置位时有效）

使用 Intel RTM 必须获得相应的硬件支持，在不支持 RTM 的平台上无法完成编译。这是 RTM 有别于 HLE 的地方之一。

4.2.3 RTM 的 Lemming 效应

Lemming 效应是指正在运行中的事务之间陷入一种相互阻挠对方进入事务执行的状态。具体地，任意获得锁的线程会对锁变量进行修改，同时强制其他当前正在该锁上执行锁省略过程的线程中止。这些被中止的线程随后会尝试重新事务的执行或者尝试获取锁。当存在多个线程都处于这一过程时，这些线程会陷入一种持续阻挠对方回退到回退路径上执行的状态，使得锁省略过程长时间处于停滞状态。通俗的讲，Lemming 效应就是 RTM 事务在运行过程中由于事务中止而引起的串行开销。

在 RTM 中，受锁保护的代码段执行时通过启动一条事务并“虚”持有锁，但实际上并没有真正持有该锁。也就是说，锁被读取，如果它的状态被解锁，那么它就被放置在被锁住的事务读集中，而不影响锁的状态，这仍然保持解锁状态。然而，当事务中止时（例如发生冲突），它会执行回退然后以非事务性的执行方式获取锁，并将锁写入。被中止的线程获取的全局可见的锁与 RTM 事务时试探性加载的锁发生冲突，由于在锁的位置上发生了冲突，所以会导致所有这些锁被中止。此外，当新的线程到达临界区时发现当前锁已经被其他线程持有时，新的线程不会启动它们的事务。在公平锁的情况下，获取锁时产生的冲突会使得线程无法并行的执行，这种情况一直会持续到经历一个没有线程试图访问锁的静默期为止。这种引起不必要的串行化从而限制并发度的现象称为 *Lemming* 效应[126]。

如果是在中止率很低的情况下，*Lemming* 效应的影响可以忽略不计。当中止率升高时，*Lemming* 效应的影响会越来越严重。从本质上说，*Lemming* 效应使得事务从中止中恢复过来的成本更高，并且引起不必要的串行化执行。

在事务的执行过程中，常常由于冲突而导致事务中止。事务申请获取的锁当前被其他线程占有从而被迫中止，但是持有该锁的线程可能很快执行完成并释放锁。因此，大部分被中止的事务经过一次或者多次重试之后可以成功获取锁，完成提交。Intel 在 TSX 开发者手册中提倡使用 RTM 进行锁省略编程时适当的进行 RTM 重试有助于提升性能。使用 RTM 编写锁省略代码时，如下行为会引起 *Lemming* 效应的加剧：

第一，过快的进行 RTM 重试。当检测到发生中止时立即进行重试。考虑这么一种情况，设定最大重试次数为 5 次（重试 5 次仍然无法成功提交则放弃事务执行转而使用常规方式申请锁执行），假设现在有 t_1 和 t_2 两个线程在并行的执行事务代码，t_1 长时间的持有锁。这时 t_2 尝试申请锁，但发现当前锁不空闲，于是立即开始重试，重试次数很快达到 5 次，此时 t_1 仍然没有释放锁，t_2 跳转到回退路径上执行，然后使用标准方式申请锁完成执行。如果有更多的线程并行的处于上述状态，它们将陷入 *Lemming* 效应的魔咒。这些重试达到最大次数进入回退路径执行的线程排队等待锁执行，长时间无法再一次进入到事务执行。对性能造成损耗。

第二，设定的 RTM 重试次数过大。一般地，在使用 RTM 进行锁省略编程时通常能够提升性能。而且，对于不同的工作负载最大重试次数是弹性变化的。但是，当设定的重试次数过大时，在处理其他工作负载时会有加剧 *Lemming* 效应的风险。同样以 t_1 和 t_2 为例，假设 t_1 和 t_2 都尝试事务执行，它们进入临界区，一定的时间后同时造成对方中止。然后，它们进行重试，又一次遇到上述情况，如此循环若干次。摆脱这种纠缠不清的状态的方法是其中一方跳转到回退路径上执行。但是，如果重试次数过高，它们在进入回退路径之前会浪费较长的时间。

第三，不设定回退路径，不停的进行重试。除了冲突可能造成事务中止之外，还有其他原因（比如执行 TSX 不兼容的指令，动态链接库，页脏位以及其他异常等）也会造成中止，TSX 不能保证每次事务执行都能成功。不设置回退路径可能导致程序挂起。

第四，没有将锁变量放入读集合内。锁省略依赖于其读取集（read set）中的锁变量，用来确保事务地执行和真实的锁持有者之间的完全同步。这要求在事务执行过程中至少读取一次锁变量。如果该锁只使用单一的锁变量，那这个过程将由 RTM 自动完成，但是当锁有多个锁变量的时候，即便是有 RTM 的情况下，这个过程也可能会出错。

第五，在事务执行完成时使用 *xtest* 代替锁的状态检测。

4.3 软件辅助的硬件锁省略技术

Haswell 的硬件事务内存使用非常简单的“请求者至上”（requestor wins）的冲突解决策略[127]。具体地是指在事务执行的过程中，如果有一致性消息（读或者写）到达写入集中的缓存行，或者由于写入而导致的驱逐到达读取集中的缓存行，则该事务被中止。这种策略存在的问题是，它既不能避免线程“饥饿”，又容易引起“活锁”[128]。Y.Afek 通过实验证明事务容易出现伪中止（suprious aborts）现象[99]，这种异常的中止无法用数据冲突或者读/写集的溢出来进行解释。这种异常的中止表明即便是运行完美的无冲突的工作负载，也可能出现由于 Lemming 效应而引起的性能损耗。

在使用硬件事务内存设计并发哈希表时，为了减轻 *Lemming* 效应对性能造成的负面影响，Intel 的开发者手册中建议对被中止的事务进行重试[98]。然而，从上一节中介绍了几种可能加剧 *Lemming* 效应的原因，过快、过度的使用 RTM 重试或者不对 RTM 重试的次数做出限定同样会引起 *Lemming* 效应。Y.Afek 等人通过实验也表明这种简单的技术方案并不能完全解决这个问题[99]，尤其是当 *Lemming* 效应很严重时，比如使用公平锁或者高并发度情况下，单纯的依靠 RTM 重试机制不能有效缓解 *Lemming* 效应的副作用。为了有效地抑制 *Lemming* 效应造成的性能下降问题，他们提出了两种软件辅助方案：软件辅助的锁删除方法和软件辅助的冲突检测方法。

4.3.1 软件辅助的锁删除方法

上一节对 RTM 的 Lemming 效应进行介绍时提到，RTM 事务携带锁在读取集中运行会引起 Lemming 效应。也就是说，尽管 RTM 事务只是“虚”持锁，但是这个锁仍然会在某个线程中止时制造事务中止的链式反应。有一种考虑是，如果硬件已经通过一致性协议检测到了冲突，为何不直接避开锁启动事务？单纯的实现锁的删除会造成两个方面的问题：其一，多个线程会由于数据冲突而彼此中止对方执行的事务，从而造成没有任何线程能够前向执行，线程可能会陷入“活锁”（live-lock）状态；其二，如果有线程持有锁以非事务的方式执行，那么与该线程并发执行的事务线程的事务会观察到不一致的状态（比如由非事务线程写入的中间状态会被事务线程观察到）。这部分内容将尝试实现锁删除方法，以牺牲不透明度（opacity）为代价（允许事务线程观察到非事务线程写入的中间状态），确保不会出现“活锁”，并避免锁删除带来的两个方面的问题。

解决由于线程饥饿或者活锁引起的线程前向问题，可以通过设置前向保证机制——在事务线程提交时，线程回退，并使用锁完成提交。但是，简单的让被中止的线程通过获取锁避免饥饿的做法会导致不正确的执行结果。这是由于运行在临界区内的线程的非原子性造成的。由这样的线程执行的内存更新操作是全局可

见的，使得并发执行的事务线程可以读取到不一致的状态。下面结合一个具体实例予以说明。

错误示例：图 4.1中的代码片段 C_1 和 C_2 受同一个锁 L 保护。假设线程 t_1 不访问锁 L，事务地执行 C_1，并且读取到 X 的值为 0。此时，另一个线程 t_2，非事务性地执行代码 C_2，申请锁 L，并将 $Y = 1$ 写入。紧接着，线程 t_1 从内存中读取变量 Y。由于变量 Y 不在 t_1 的读集中，所以线程 t_2 对变量 Y 的写入操作与线程 t_1 读取变量 Y 的操作不存在数据冲突，此时 t_1 读取到的变量 Y 的值为 1。随后 t_1 执行完毕并提交。因此，t_1 观察到不一致的状态：$X = 0$ 且 $Y = 1$。（在上面的例子中，如果 t_2 事务性地运行，t_1 要么能够读取到 t_2 的全部写入结果，要么全部读取不到，不存在只读取到其中的某一个）。

代码片段C1:		代码片段C2:	
lock	(L)	lock	(L)
load	(X)	store	(Y, 1)
load	(Y)	store	(X, 1)
unlock	(L)	unlock	(L)

图 4.1　错误示例

算法 4.1: SLR 的加锁方法

```
1  Function lock()
2      retries = 0
3      speculative path:
4      XBEGIN (line 6)                    /* 如果发生中止，跳转到第 6 行重试 */
5      return
6      fallback path:
7      retries = retries+ 1
8      if retries < MAX_RETRIES then
9          goto Line 3
10     else
11         lock.lock()                    /* 执行标准锁请求 */
12     end
```

为了消除 Lemming 效应对硬件事务内存的影响，引入第一种软件辅助技术——软件辅助的锁删除技术，Software-assisted lock removal(SLR)。SLR 可以视为对 Rajwar 和 Goodman[86] 硬件锁删除技术的软硬件混合实现。Rajwar 和 Goodman[86] 观察到，只要事务内存能够为冲突的事务提供前向保证，那么就可以

不需要加锁操作就能执行具有相同临界区的事务（也就是通过启动事务代替申请锁，提交事务代替释放锁）。但是这种方法需要依赖于硬件冲突管理策略能够保证线程无饥饿，从而避免出现活锁问题（存在数据冲突的事务陷入彼此中止对方的死循环）。而 HasWell 的事务内存采用的“请求者至上”的冲突管理策略既不能保证线程无饥饿，又不能避免活锁。SLR 使用如下方法解决线程活锁问题：它使用硬件事务内存事务地执行临界区，在事务执行过程中不需要访问锁，直到准备提交时才会读取锁。提交时，如果当前锁没有被其他线程持有，则获取该锁并完成提交；否则，该事务被中止并进行重试（retries）。如果重试了若干次之后仍然未能成功提交，则放弃事务地执行，改用非事务的方式通过直接获取锁完成本次操作。在使用 SLR 时，线程获取锁的行为既不会与正在运行中的事务产生冲突，也不会阻止其它到达的线程推测性地（speculatively）启动其事务。但是这个方法有一个缺点，它会损失一部分不透明性（opacity），即推测性事务可能会与持有锁的事务并行的运行，此时推测性事务可能会观察到不一致的状态（这会导致推测性事务提交失败并中止）。R. Guerraoui[129] 的研究表明，由于事务的沙箱（sand boxing）机制，这种不透明性在大多数情况下是安全的。

SLR 的加锁和解锁过程如算法 4.1 和 4.2 所示。在算法 4.1 中，第 3 行为事务代码区域的起始位置。如果在事务执行时发生中止，则跳转到第 6 行的回退路径上执行，在回退路径内，首先对中止的事务进行重试，如果重试的次数超过预先限定的门限值 MAX_RETRIES，则放弃事务执行，转而用普通的方式通过申请锁完成本次执行（第 10 行）。

算法 4.2: SLR 的解锁方法

```
1  Function unlock()
2      if XTEST() then
                                        /* 如果本次为事务执行，返回 TURE */
3          if lock is TRUE then
4              XABORT                                  /* 中止事务执行 */
5          else
6              XEND
7          end
8      else
9          lock.unlock()                          /* 以标准解锁方式解锁 */
10     end
```

4.3.2 软件辅助的冲突检测方法

算法 4.3 和 4.4 描述了 SCM 的加锁和解锁的过程。

算法 4.3: SCM 的加锁方法

```
1  Function lock()
2      retries ← 0 ;
3      primary path:
4      XBEGIN (line 6)                    /* 如果发生中止，跳转到第 6 行重试 */
5      条件成熟时调用 HLE 或者 SLR 的 lock() 方法;
6      return
7      serializing path:
8      if aux_lock_owner is FALSE then
9          retries ← retries+1;
10     else
11         aux_lock.lock()                          /* 辅助锁申请锁 */
12         aux_lock_owner ← TURE
13     end
14     if retries < MAX_RETRIES then
15         goto Line 3
16     else
17         main_lock.lock()
18     end
```

软件辅助的冲突管理技术 (Software-assisted Conflict Management, SCM)。SLR 方法虽然能够很好的抑制 *Lemming* 效应，但是这个方法有一个缺点，它会损失一部分不透明性（opacity)。SCM 是另一种抑制 *Lemming* 效应的软件辅助方法，它通过使用简单的冲突管理技术，允许不冲突的线程继续运行其试探性地基于 RTM 的事务，而不受产生冲突的线程的影响。为此，在锁的实现中添加一个串行化的路径，其中一个被中止的线程必须获得一个独立的辅助锁（不使用锁省略）以便重新加入与其他线程的推测执行。使用这种方法，发生冲突的线程之间被串行化，而不影响其他线程的并行执行。如果线程在多次冲突中失败，那么它必须放弃推测执行转而申请原始锁。

SCM 方法使用了两种锁，一种为主锁 (*main_lock*)，一种为辅助锁 (*aux_lock*)。主锁使用的是基于 RTM/SLR 机制中的一种，而辅助锁是一种使用传统的锁方法实现的。只有在以标准的非事务执行方式执行代码区域时才会申请使用辅助锁（算法 4.3 第 9、10 行）。辅助锁会将所有陷入冲突的线程召集到一起，然后将顺序

地执行。当某一事务被中止时，被中止的线程以非事务性的方式申请辅助锁，随后重新加入到对原始临界区的推测执行队列中。这种为了重新加入到推测执行队列而申请辅助锁的过程称为串行化的路径 (serializing path)。线程在进入串行化路径之前会对事务进行重试。

算法 4.4: SCM 的解锁方法

```
1  Function unlock()
2      if XTEST() then
3          调用 HLE 或者 SLR 的 unlock() 方法 XEND
4      else
5          main_lock.unlock()
6      end
7      if aux_lock_owner = TRUE then
8          aux_lock.unlock()
9          aux_lock_owner ← FALSE
10     end
```

使用 SLR 方法时，当线程放弃事务执行转而以非事务的方式申请锁时会引起的程序的前向问题，结合使用 SCM 可以解决这一问题。

上述的两种软件辅助方法已经使用标准的 pthread 锁接口封装到动态库中。可以直接在程序中使用上述方法而无需对代码进行调整，也无需重新编译。

4.4 基于 HTM 的并发哈希表的设计

4.4.1 问题引入

哈希表用于在线性时间内对键值对建立映射关系。插入和查询是哈希表的两种核心操作，当然哈希表的扩容和删除操作也很重要，但是这里为了简要的引出所要解决的问题，暂时不考虑哈希表的扩容和删除操作。

考虑这么一种场景，有一个简单的如图 4.2所示哈希表，它顺序地执行五次操作，分别是插入、插入、插入、查询、插入。现在变更需求，希望将这个哈希表设计成并发的，会有怎样的问题呢?

设计高并发的哈希表并不是一件简单的事。目前有很多方法，比如基于锁的方法，使用原子指令实现的无锁化编程等都可以用于构建高并发度的哈希表。但是这些方法无疑都会涉及到更加复杂的数据结构的设计，同时也增加了程序的复杂性。

使用粗粒度锁方法（使用少量的锁）实现并发是最简单的方法。最直观的是

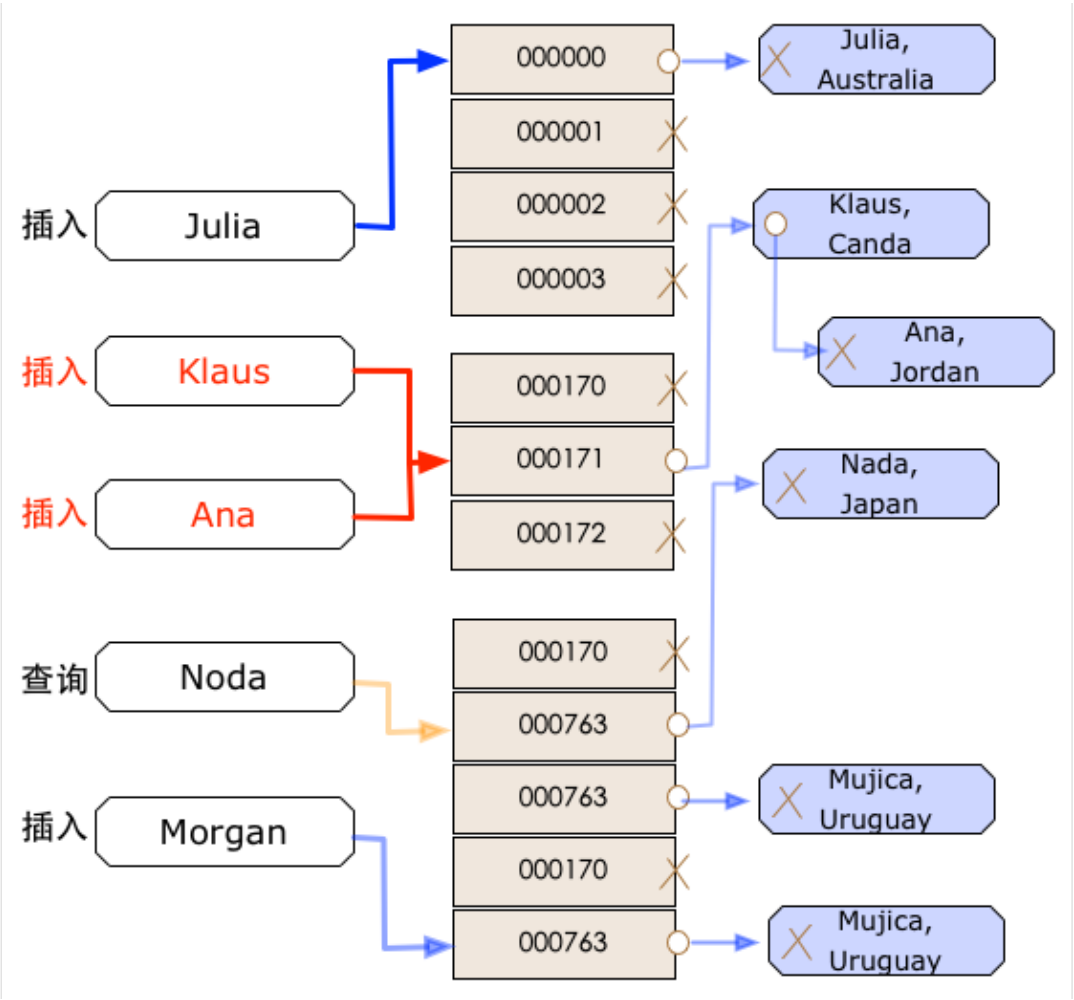


图 4.2　串行执行五个操作的哈希表

使用全局锁，即将整个哈希表作为临界区，只用一个锁进行同步控制。在这个方法中，每一次对哈希表的操作都是先申请锁，然后执行相应操作，再释放锁。当锁被某个线程持有时，其他线程无法获得锁，因此也不允许对哈希表进行其他操作。显然，这种方法的缺陷是致命的，它的效率非常低。

与粗粒度锁对应的方法是细粒度的锁方法，即缩短临界区的长度，增加锁的数量。具体的方法是将哈希表分成更小的区间，每个区间用一个锁保护，一般的做法是对每个哈希桶设置一个锁。使用这种方法的好处是，允许多个线程在不存在数据竞争的前提下并发的对哈希表进行操作。相比于之前采用的整个哈希表用一把锁的情形，显然细粒度锁方法的效率要更高。但是细粒度锁方法同样也存在缺陷，它引入了不必要的延迟，有更高的内存消耗，并且使得数据结构的设计更加复杂，且需要花费大量精力进行正确性验证。

粗粒度锁方法易用，便于理解，易于调试，唯一的缺陷是在多线程环境下对性能造成阻碍。在多处理器上这个缺陷是致命的，不可调和的。细粒度锁方法能够较好的发挥多核处理器的性能，但是细粒度锁方法的实现颇为复杂。那是否有一种方法既具备粗粒度锁方法简单易用，便于理解，易于调试的特点，又能够保

证获得使用细粒度锁方法那样的性能呢？Intel TSX 实现了细粒度锁或者无锁化编程的性能与粗粒度锁方法的简单易用等特点的结合，对于并发数据结构的设计具有重要意义。

图 4.2 中的五个操作，只有两个插入操作存在冲突。其他的三个操作都是互不干扰的。因此，使用事务内存可以实现硬件锁省略。换句话说，使用这种方法获得的性能与执行没有加锁和解锁过程的代码获得的性能非常接近。造成性能相近的关键原因是这些操作受到硬件事务内存的保护，该硬件取代了锁的功能，实现对临界区的保护。

被映射到同一地址的两个操作由于存在冲突，所以需要交错执行。但是，在多线程环境中，两个操作有可能被两个线程同时执行。Intel TSX 将确认在这种情况下确实需要加锁以保证代码的正确执行，同时也会因为加锁操作而引入额外的开销。实际上，出现这种情形时，相互冲突的任务将执行到事务代码里面，一直到处理器检测到这个冲突。这时，双方都会中止事务代码的执行。最常用的解决这个问题的方法是让每个任务继续以非事务的方式（常规的获取锁，释放锁的方式）执行。这就意味着，总有一个任务先获得锁而进入临界区完成操作，而另一个任务则延迟直到先进入临界区的任务执行完毕。

4.4.2 缓存行哈希原型描述

根据第三章中对现有并发哈希表的全面评估与分析的结果，发现缓存行哈希表在多个平台上无论是线程扩展性、性能或者延迟这些方面都具有突出的表现。因此，在设计基于硬件事务内存的并发哈希表时，参考了缓存行哈希表的数据结构的设计思路。

在前一章中详细介绍了缓存行哈希表 (第3.2.1节) 的设计思想、原理以及使用的同步方法。这里只对与本章内容密切相关的问题做一个简单的回顾。缓存哈希表的哈希桶被设计成具有与缓存行相同的大小（64 字节），这样设计的好处是在发生缓存未命中时，最多只需要一次缓存行切换操作就能匹配到正确的数据。这样可以避免由于过多的缓存行切换次数对系统性能造成的损耗。每一个哈希桶中包含了一个 8 字节的同步控制字段，用于存放锁 (用于实现基于锁的版本) 或原子快照 (用于实现无锁版本的同步控制)。基于锁的缓存行哈希表采用细粒度锁（每个哈希桶都设置有一个锁字段）完成对读者线程和写者线程的同步控制。查询操作遍历键/值对，如果匹配，则返回值。算法 4.5 描述了基于锁的缓存行哈希表的插入操作的过程。更新操作（插入或者删除元素）首先需要执行一次查询以确定该操作可以继续执行（如果插入元素时发现桶内已存在相同元素或者删除元素时发现桶内没有该元素则不进行下面的操作），如果可以继续执行，则持有该桶的锁，直到完成相应的更新操作，完成后释放锁。如果当前映射到的哈希桶内已经没有足够的空间插入新的元素，则会选择使用指针字段链入一个新的哈希桶，或

算法 4.5: CLHT-lb 的插入方法

```
1  Function Insert(hashtable,key,val)
2      初始化;
3      lock ← &bucket → lock;
4      empty ← NULL;
5      empty_v ← NULL;
6      LOCK_ACQ(lock);
7      while True do
8          for j = 0 to ENTRIES_PER_BUCKET − 1 do
9              if &bucket → key[j] == key then
10                 LOCK_RLS(lock);
11                 return false;
12             end
13             else if empty == NULL and bucket.key[j] == 0 then
14                 empty ← &bucket → key[j];
15                 empty_v ← &bucket → val[j];
16             end
17         end
18         if bucket → next == NULL then
19             if empty == NULL then
20                 bucket → next ← clht_bucket_create();
21                 bucket → next → key[0] ← key;
22                 bucket → next → val[0] ← val;
23             else
24                 empty_v ← val;
25                 empty ← key;
26             end
27             LOCK_RLS(lock);
28             return true;
29         end
30         bucket ← bucket → next;
31     end
```

者触发哈希表扩张操作 (resize)。

通过评估的结果还发现缓存行哈希表对 NUMA 架构的多核系统友好，能够很好的克服访问远程内存节点时过高的延迟开销的问题。

但是，缓存行哈希表同样存在一些不足之处:

第一，较之 Cuckoo、Hopscotch 等哈希表，缓存行哈希表需要消耗更多的内存。这主要体现在两个方面:

- 基于链表的实现方法占用更多的内存空间。使用链表组织哈希桶的结构可以保证查询速度以及哈希表的空间利用率，但是每个桶都需要有 8 字节用于存放指针;
- 以缓存行大小为粒度的锁实现消耗更多内存。临界区的长度等于一个缓存行的大小，每个哈希桶都包含 8 字节的信息用于并发控制（锁版本该位置存放锁，无锁版本该位置存放的是原子快照）。

第二，细粒度锁方法虽然能够保证线程扩展性以及稳定的性能，但其实现复杂度过高并且难以进行正确性验证。

硬件事务内存为多线程应用的同步提供硬件指令支持，使用硬件事务内存既可以实现无锁化编程，又能用于实现锁。它以粗粒度的锁实现达到或者接近使用细粒度锁方法所获得的性能。本文参考缓存行哈希表的设计思想，使用 Intel RTM 实现了基于硬件事务内存的并发哈希表。并运用 SLR 和 SCM 两种技术用于优化 RTM 实现的锁以获取更高的性能。

4.4.3 基于 RTM 的锁方法描述

在介绍实现基于 RTM 的缓存行哈希表所用的同步方法前，先对 MCS 锁的基本原理进行介绍，并就采用 MCS 锁的原因予以说明。

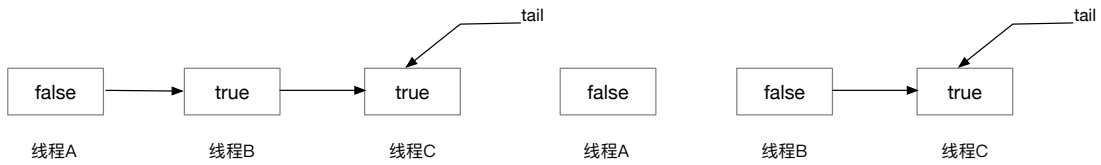
实现基于硬件事务内存的缓存行哈希表使用的是 MCS 自旋锁[49]，MCS 锁的名字来源于其发明人——John Mellor-Crummey 和 Michael Scott 名字的首字母，这个名字并不涉及锁本身的具体意义。MCS 锁是一个链表形式的锁，每一个线程是链表中的一个节点，MCS 锁用一个 tail 指针维护链表中的最后一个节点，每一个节点用一个布尔值 $locked$ 表示自己是否被锁定，以及一个 next 指针表示在链表中的下一个节点。

- **加锁:** 线程获取锁时，用 SWAP 操作将 tail 指针更新为指向自己；如果 tail 之前的值为 NULL，则表示没有线程在等待，该线程可以直接获得锁进入临界区。如果 tail 的值不为 NULL，说明前面有线程在等待，那么该线程插入到链表尾端，并将自己的 locked 标志设置为 true，然后在 locked 上自旋，等待其它线程将 locked 设置为 false，然后退出循环结束等待，进入临界区;
- **解锁:** 线程 A 释放锁时，首先判断自己的 next 节点是否为 NULL，如果为 NULL，则说明后面没有线程在等待，那么 A 就是链表中的最后一个线程，

随后执行 CAS 操作，将 tail 指针指向 A，将 tail 指针设置为 NULL 并返回，如果 CAS 执行成功了，表示成功释放了锁，可以直接返回；如果 CAS 操作失败，说明在执行 CAS 操作之前有其它线程 B 先一步插入到链表的尾端并修改了 tail 指针，B 会将 A 的 next 指针指向自己，则 A 需要等待 next 指针不为 NULL。随后 A 将 B 的 locked 标志修改为 false，B 获取锁，进入临界区。

MCS 锁的加锁和释放锁的过程如图 4.3所示。图 4.3（a）描述的是多个线程等待获取锁的情形，一共有 A、B、C 三个线程，其中线程 A 的 locked 标志为 false，表明线程 A 正在临界区内执行，B 和 C 先后试图获取锁，由于此时 A 正在临界区内，所以 B 和 C 依次插入到链表内，它们的 locked 标志都为 true。tail 指向当前链表内的最末端的节点，图中对应节点 C。图 4.3（b）描述的是线程 A 释放锁的过程，A 释放锁时，A 将它的 next 指针指向的节点 B 的 locked 标志设置为 false，B 退出结束自旋，进入临界区。A 将自己从链表中脱离。

选择 MCS 锁的原因：其一，基于 test-and-set（TAS）的自旋锁在多核系统上的扩展性较差。每一次 TAS 操作都会写入高速缓存，导致所有线程的对应的缓存行都失效，当线程数量较多时，就会产生大量的缓存一致性流量。MCS 锁的优点在于每一个线程只在自己的 locked 变量上自旋，因此自旋完全可以在对应的核心的 L1 缓存中完成，不会引起缓存一致性流量和 NUMA 系统上的跨节点流量，也不会产生内存访问。只有在释放锁时需要对 next 指向的线程的 locked 变量进行一次写入操作。另外，MCS 锁采用的是链表，是一种公平锁，线程按照进入链表的先后顺序排队使用锁。



(a) 加锁　　(b) 解锁

图 4.3　MCS 锁的工作原理

加锁过程: 算法 4.6描述的是结合了 SLR 和 SCM 两种软件辅助方法的 MCS 锁。该算法中，进行 RTM 重试的次数设定为 10，用 $MAX\_RETRIES$ 表示。首先将 $lock$ 设置成 MCS 锁（第 2 行），并将变量 $reason$ 初始化为 0（第 3 行）；算法设计了两条执行路径，一条为 speculative 路径（4-6 行），用于事务执行；一条为 fallback 路径（7-32 行），用于触发了事务中止时执行。在 speculative 路径内，指令前缀 XBEGIN 表示事务代码的起始位置，它携带两个参数：fallback_path，指明回退路径的地址，reason 用于记录当前事务被中止的原因。如果事务顺利执行并且成功提交，则通过 XEND 指令前缀结束该事务；如果由于某种原因触发事务中止

时，线程携带触发中止的原因跳转到fallback路径上执行。进入fallback路径后（第7行），首先将变量$retries$的值加1（第8行），再将$retries$与$MAX_RETRIES$进行比较，如果进行RTM重试的次数还没有达到上限，则跳转到第4行的speculative路径上，重新尝试运行事务（第10行）。如果进行重试的次数已经超过了上限值，并且此时线程持有的是辅助锁，则将线程添加到辅助锁队列内（12-14行）；否则就以标准的锁申请方式申请获得辅助锁（16-25行），并使线程进入忙等待状态（第21行）。如果事务被中止的原因是TXN_MAY_SUCCEED，进入辅助锁队列内的线程还会进行一轮重试（26-29行）。最后，设置主锁为”True”，以标准方式申请持有主锁。

解锁过程: 算法4.7描述了与算法4.6对应的解锁过程。在执行解锁操作时，首先使用XTEST指令判断当前执行的内容是否属于事务执行，如果为事务执行（第3行），则进一步判断当前执行的事务代码内是否存在锁嵌套（第4行），如果当前恰好有事务获取了锁正在执行提交操作，则显式的调用XABORT指令中止该事务（第5行），这样可以避免因锁嵌套引起的lemming效应；然后使用XEND指令终止当前正在执行的事务代码。如果当前线程正处于辅助锁队列内（第8行），则需要使对应的辅助锁标志（第9行）无效，并将辅助锁的重试次数置为0（第10行），随后将辅助锁释放（11-18行）。如果当前线程不是事务线程（即它以非事务的方式对临界区进行操作），则表明该线程持有的是主锁，则按照标准的锁释放流程释放锁，将锁的状态置为”False”（22-31行）。最后，将变量$retries$置为0。

4.4.4 基于HTM的缓存行哈希表

原始的缓存行哈希表的实现有基于锁和无锁两个版本。本文设计基于HTM的缓存行哈希表的灵感主要来自其基于锁的版本。缓存行哈希表采用的是细粒度锁方法，这种方法的好处是使临界区足够短，能有效的减少数据冲突。但是这种方法实现较为复杂，而且锁的数量过多，维护锁的开销较高。考虑到硬件事务内存具有简化多线程同步的特点，本文利用算法4.6中描述的MCS锁实现了基于HTM的缓存行哈希表。当线程执行对哈希表内元素的操作时，线程首先“虚”持该锁（线程表面上持有锁，但实际上该锁并没有真正意义上被占有，如果有其它线程完成了事务需要进行提交时，仍然可以优先持有该锁），然后事务的执行相应的操作，待执行完毕准备提交时才真正意义上的申请锁，如果当前锁为空闲状态，则占有锁，完成提交；如果提交时锁被其他线程占用，则中止提交，然后进行重试。

基于HTM的缓存行哈希表使用的是全局锁，它将整个哈希表作为临界区的方法本质上是一种粗粒度锁实现，但是通过与原始的细粒度锁实现的缓存行哈希表比较发现，无论是线程扩展性还是吞吐量都具有极强的竞争力，甚至获得了优于使用传统细粒度锁方法实现的缓存行哈希表的性能。

算法 4.6: 结合 SLR 和 SCM 的锁方法

```
1  Function lock_mutex_lock(*mutex)
2      *lock ← *mutex;
3      reason ← 0;
4      speculative_path:
5      XBEGIN(fallback_path, reason);
6      return 0;
7      fallback_path:
8      retries ← retries + 1;
9      if retries < MAX_RETRIES then
10         goto speculative_path;
11     end
12     *prev ← NULL;
13     if thread_handle == lock → aux_lock_owner then
14         lock → aux_retries + +
15     else
           /* 以标准方式申请辅助锁                                    */
16         my_aux_node.locked ← true;
17         prev ← __sync_lock_test_and_set(&lock → aux_lock, &my_aux_node);
18         if prev! = NULL then
19             prev → next ← &my_aux_node;
20             while my_aux_node.locked do
21                 cpu_relax();
22             end
23             lock → aux_lock_owner ← thread_handle;
24             lock → aux_retries ← 1;
25         end
26         if reason&TXN_MAY_SUCCEED is true then
27             if lock → aux_retries‘MAX_RETRIES then
28                 goto speculative_path;
29             end
30         end
           /* 以标准方式申请锁，这个过程与申请辅助锁一样，省略描述      */
31
32     end
33     return 0;
```

算法 4.7: 结合 SLR 和 SCM 的解锁方法

```
1  Function lock_mutex_unlock(*mutex)
2      *lock ← *mutex;
3      if XTEST() is true then
4          if lock → lock != 0 then
5              XABORT(1);
6          end
7          XEND();
8          if thread_handle == lock → aux_lock_owner then
9              lock → aux_lock_owner ← INVALID_THREAD_HANDLER;
10             lock → aux_retries ← 0;
11             if my_aux_node.next == NULL then
12                 if __sync_bool_compare_and_swap(&val,&val,NULL) is true then
13                     return 0;
14                 end
15                 while my_aux_node.next == NULL do
16                     cpu_relax();
17                 end
18             end
19             my_aux_node.next ← NULL;
20             last → locked ← false;
21         end
22     else
           /* 使用标准的方式进行解锁                                    */
23         if my_node.next == NULL then
24             if __sync_bool_compare_and_swap(&val,&val,NULL) is true then
25                 goto unlock_aux_lock;
26             end
               /* 接下来重复第 15 至 17 行 while 循环                  */
27         end
28         my_aux_node.next ← NULL;
29         last → locked ← false;
30         unlock_main_lock:              /* 这里重复第 9 至 20 的过程 */
31     end
32     retries ← 0;
33     return 0;
```

为了进一步探究 RTM 对细粒度并发哈希表性能的影响，实现了基于 RTM 的细粒度锁版本的缓存行哈希表。锁的粒度与原始实现一致，唯一不同的是用基于 RTM 的 MCS 锁代替原始实现使用的锁方法，具体的性能表现在第 4.5一节的性能评估中展示。

4.5 性能评价

4.5.1 测试平台和参数设置

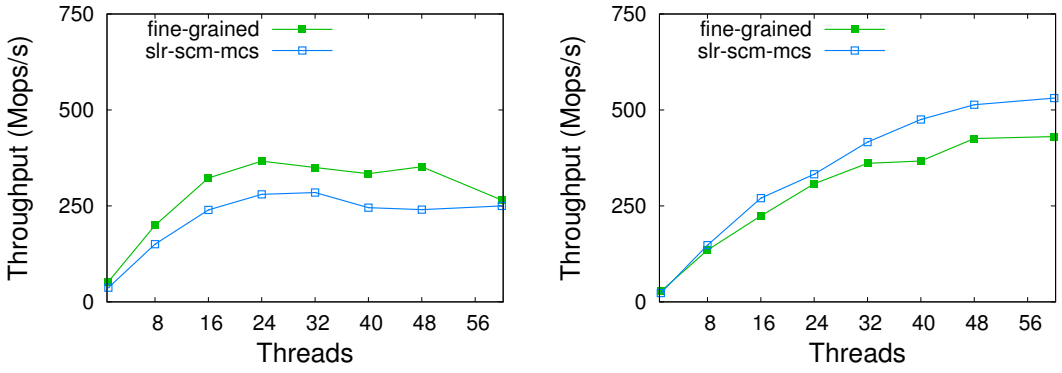
进行实验测试的平台为 Linux 工作站。该工作站配备有两个 Intel Xeon Broadwell EP/EN/EX 处理器，总共有 32 个物理核（64 个逻辑核），内存总容量为 64 GB。CPU 的时钟频率为 2.1GHz，三级缓存的容量分别为 64 KB，256 KB 以及 40 MB。该工作站安装的是 Ubuntu 16.04 LTS 操作系统。

本次实验的源代码使用 GCC-4.8.0 编译生成可执行文件。下文中如非特别说明，所有的可执行文件的编译都是使用默认的线程绑定方案（见 3.4.6 ），即由操作系统完成线程与核之间的映射。为了简便起见，实验中所用的键/值对的大小均为 64 位，所有的查找、插入以及删除请求都是按照预先设定的分布方式通过伪随机数方法生成。为了便于描述，创建的线程数量用参数 n 表示。所有被创建的 n 个线程都执行相同的工作负载，该工作负载中包含的更新操作请求的比重用 u% 表示，则查询操作所占的比重为 100- u%（实验中的默认更新比重为 10%）。更新操作的一半为插入操作，剩下的一半为删除操作。实验中用到的其它参数如下：d 表示一次测试运行的时间，单位为毫秒。i 表示预先填充至哈希表中的元素的个数，r 表示键的范围，r 的范围为 0 到 2i。

4.5.2 基于 HTM 的粗粒度锁实现与细粒度锁 CLHT-lb 的比较

本次实验中，设定了大小为 1000 和 1 百万两个不同的初始化值，设定为 1000 的目的是使工作集的大小恰好可以被每个核的私有缓存所容纳，而设定为 1 百万的目的是使得所运行的工作集的规模超过最后一级缓存的大小，以便测试主存与缓存之间进行数据交换的性能。每个工作集中更新操作所占比重为 10%，每次测试运行时间为 5000 毫秒。实验的最终结果取五次运行结果的平均值，最终得到的性能曲线如图 4.4 所示。fine-grained 代表基于传统的细粒度锁实现的 CLHT-lb 版本；slr-scm-mcs 代表结合了 SLR 和 SCM 方法优化的基于 HTM 的缓存行哈希表实现。在初始化元素个数设定为 1 百万时，两个版本都展现良好的线程扩展性，吞吐量随着参与运算的核的数量的增加而增加，其中 slr-scm-mcs 性能要略好于 fine-grained（约高出 20% 左右）。然而，当初始化元素的规模小于私有缓存的容量时，fine-grained 的性能要比 slr-scm-mcs 的性能要好。出现这种现象的原因在于：

在这个级别的数据集下，缓存行哈希表会遭遇严重的数据冲突，而数据冲突的加剧会造成事务执行频繁的进行重试，并且触发事务中止的概率也更高，事务一旦中止，就必须跳转到回退路径处继续执行。而从实现的复杂度的角度来分析，基于 HTM 的缓存行哈希表的实现方式要比传统的细粒度锁方式要更加简单。在构建并发数据结构时，它只需要使用一个全局锁对临界区进行保护，极大的减少了锁的数量，并且完美的摆脱了使用全局锁会抑制多线程性能的问题。另外，使用细粒度锁方法需要的内存空间要多于使用 HTM 时所需的内存空间。如果需要创建 1024∗1024 个哈希桶，需要额外的分配 8 MB 的内存用于存储同步变量。



(a) $i = 10^3$　　(b) $i = 10^6$

图 4.4　传统细粒度锁方法和基于 RTM 的粗粒度锁方法之间的性能比较

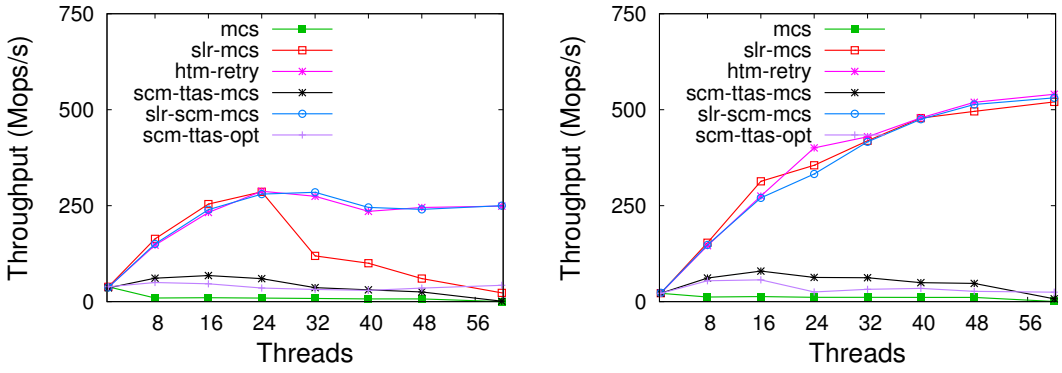
分析：根据实验结果，有两点结论：第一，在处理大规模工作集时，使用 HTM 构建并发哈希表的好处体现在两个方面：一是获得的性能和扩展性具有一定的竞争力；二是它能降低内存开销并达到简化并行编程的目的。第二，当工作集的大小小于片上缓存的容量时，此时由于更加激烈的数据冲突引发频繁的事务中止影响了基于 HTM 的全局锁性能。

4.5.3　不同的全局锁方案之间的比较

在这一小节，将进一步评估使用不同的锁实现方式作为全局锁时，对应的缓存行哈希表之间的性能差异。为此实现了 6 种锁方法，它们分别是：（1）标准的 MCS 锁——没有使用任何优化的 MCS 锁方法；（2）用 SLR 优化的基于 RTM 的 MCS 锁（slr-mcs）；（3）基于 HTM 的事务重试方法（HTM-retry）；不使用 SLR 和 SCM 进行优化，仅参照 Intel 技术手册上推荐的方法对发生中止的事务进行重试，重试的次数设定为 10。（4）使用 SCM 优化的基于 RTM 的 ttas 锁；（5）采用 SLR 和 SCM 两种软件辅助方法共同优化的 MCS 锁（slr-scm-mcs）；（6）基于乐观的 SCM 方案的 ttas 锁，线程在经过 10 次重试事务执行后仍无法成功提交时，线程会以事务性的方式获取锁，完成本次操作。

图 4.5 展示了运行初始化大小为 1000 和 1 百万两个不同规模的数据集的性能

曲线。正如预期那样，使用传统的未经优化的 MCS 锁方法不论是线程扩展性还

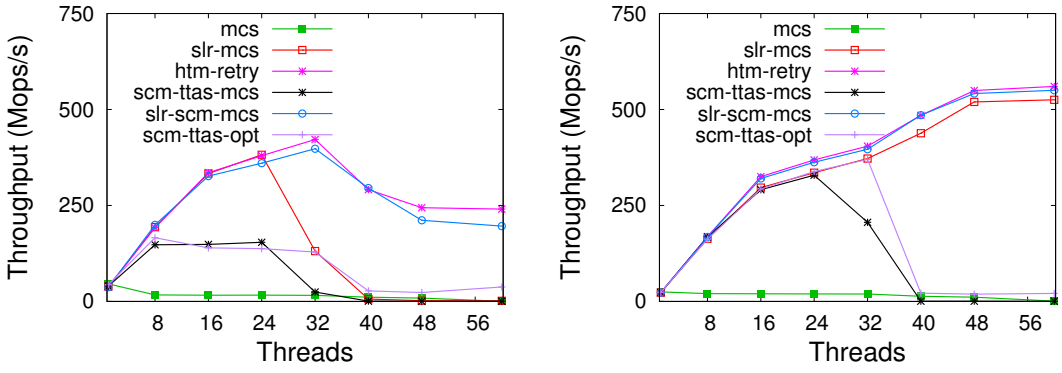


(a) $i = 10^3$　　(b) $i = 10^6$

图 4.5　使用不同全局锁方法之间的性能比较

是吞吐量都是几种方案中最差的。观察图 4.5（a）和（b）的性能曲线发现，在处理较小规模的数据集（初始化元素个数为 1000）时，slr-mcs，slr-scm-mcs 和 HTM-retries 在 n 为 24 时达到吞吐量峰值，之后吞吐量随着 n 的增加而减少。而其它两种方法，scm-ttas-mcs 和 scm-ttas-opt 对应的性能略比传统的 MCS 锁好一些。在处理小规模数据时，缓存行哈希表性能在线程数量达到一定数值之后下降的原因是：这种情况下触发数据竞争的概率成倍增加。当数据集的规模超过最后一级缓存的容量时，slr-mcs, slr-scm-mcs 以及 HTM-retry 三种方法都展现了近线性的线程扩展性。

通过对实验数据的分析还发现基于 RTM 实现的全局锁方法的缓存行哈希表的性能受线程绑定方式（三种线程绑定方式见 3.4.6 ）的影响，即采用不同的线程绑定方法时存在性能上的差异。这里采用紧凑型线程绑定方式。图 4.6 中描述了本次实验的结果。通过与图 4.5 的比较，有两点发现：其一，slr-scm-mcs, slr-mcs



(a) $i = 10^3$　　(b) $i = 10^6$

图 4.6　紧凑型线程绑定方案运行结果

和 HTM-retry 三种方案的受线程绑定方式的影响较小，并且比其它几种方案的性

能更加稳定。其二，scm-ttas-mcs 和 scm-ttas-opt 在单个 socket 内表现出较好的扩展性，然而当线程数量超过单个 socket 能够提供的最大线程数量时，它们的性能下降严重。这种现象正好印证了基于 TAS 的锁在多核系统上扩展性差的结论。

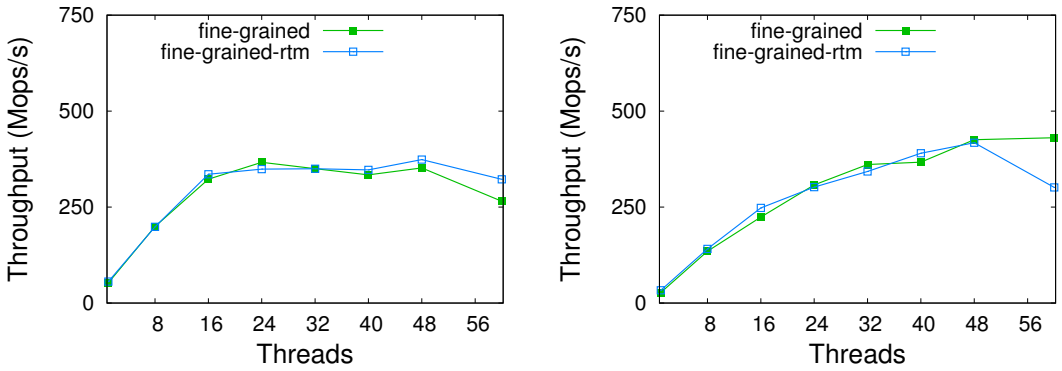
在 4.5.5 中，将通过测算总的事务量、事务中止率以及发起锁请求的次数三个指标具体分析引起不同线程绑定方式之间性能差异的原因。

分析：由这部分的实验结果得到的结论分为三个方面：第一，MCS 和 HTM-retry 同样是使用全局锁，后者是基于硬件事务内存实现的方法，但未使用任何软件优化方案，两者之间的性能差异巨大，验证了硬件事务内存有助于提高多核处理器性能的论断；第二，使用基于 HTM 实现的锁方法，能够在性能上有多大的提升还取决于所使用的软件优化方案；第三，slr-scm-mcs 和 HTM-retry 两种方案在扩展性上较其它方法更具有竞争力。

4.5.4 基于 HTM 的细粒度锁实现与传统细粒度方法的比较

使用基于 RTM 的全局锁构建并发哈希表有利于性能的提升。还有一个令人关心的问题，如果使用基于 HTM 实现的细粒度锁构建并发哈希表会对性能提升有帮助吗？下面将通过一组实验对比数据来回答这个问题。

参照实现粗粒度 RTM 锁的缓存行哈希表的设计方法，增加锁的数量，缩短临界区的长度，使用基于 RTM 的 MCS 锁实现了缓存行哈希表的细粒度锁版本，并将其与原始实现的细粒度锁版本在 CHTBench 测试框架上运行同等规模的数据集，对两者的性能和线程扩展性进行了比较。结果如图 4.7 所示。设置两组不同的初始化数量，一组用以说明哈希表的规模小于高速缓存的容量时的性能（$i = 10^3$），一组用以说明哈希表的规模远超过高速缓存容量时的性能（$i = 10^6$）。



(a) $i = 10^3$　　(b) $i = 10^6$

图 4.7　使用/不使用 HTM 的细粒度锁方法之间的性能比较

通过比较得知，$n < 48$ 时，两个版本间的性能差异并不明显。这种性能差异不明显的现象说明：如果采用细粒度锁方法，使用 RTM 进行优化以期获得更高性能的做法意义不大。但是当 $n > 48$ 时，图 4.7（a）和（b）对应的曲线有较大的

差别。具体地，当数据量减小时（图 4.7 （a)），使用 RTM 的性能要略高于不使用 RTM。在这种数据量小而并发度高的情况下，两种不同的实现方式都面临严重的数据竞争问题，但是由于 RTM-MCS 锁设置了 Retry 机制，故而在这种情况下它具有比原始的细粒度锁更好的稳定性。而在图 4.7 （b）中，情况恰好相反，不使用 RTM 的版本具有更好的线程扩展性。

分析: 经过实验比较，得到的结论是：在传统的细粒度锁方法能够提供良好的线程扩展性和性能的前提之下，使用 RTM 进行优化既达不到简化并发控制和降低内存开销的目的，又对整体吞吐量的提升没有助益。

4.5.5 影响 HTM 性能的因素分析

图 4.5 为五种不同的基于 RTM 的同步方案的性能曲线。为了探究造成它们性能上的差异的原因，借助 Intel 的性能计数监视器（PMU）收集了一些微观的运行时指标。在接下来的内容中只挑选最能说明观察到的实验现象的指标予以说明。表 4.2 中列举的数据对应的参数为 n =32，i 分别为 1 百万/1000，u =10%。表中第二列（c_2）数据表示执行过程中请求锁的次数，第三列（c_3）为发起的总的事务量，第 4 列（c_4）表示被中止的事务的数量，最后一列为中止率，通过 c_4/c_3 计算得到。事务中止率 r 由中止的事务量除以执行期间发起的总的事务量计算得到，总的事务量为中止的事务量与提交的事务量之和。从表 4.2 的数据表明不同的软件优化方案对应的中止的事务量差别不大，这说明中止的事务量并不是最主要的性能瓶颈，而执行过程中申请锁的次数以及发起的总的事务量对性能有直接的影响。申请锁的次数和提交的事务量越多，对应的吞吐量越高。比如，slr-mcs，scm-ttas-mcs 和 HTM-retry 对应的申请锁的次数和提交的事务量明显高于其它两种方案，图 4.5 所示的这三种方法的性能曲线比另外两种更加理想，也正好印证了上述论断。

表 4.2 Intel PMU 收集的不同的软件优化方法的运行时数据

	申请锁的次数（百万）	总的事务量（百万）	中止的事务量	中止率（%）
HTM-retry	170/160	290/360	53/67	18.3/18.5
slr-mcs	160/60	290/360	52/67	18.6/18.6
slr-scm-mcs	160/160	300/350	52/66	17.8/19.0
scm-ttas-mcs	48/12	110/130	51/62	48.6/48.4
scm-ttas-opt	110/48	220/180	53/65	23.0/35.4

另外一种现象是吞吐量随着中止率的升高而下降。通过运行不同的参数组合来探究中止率、创建的线程的数量、更新比重以及哈希表初始化元素个数几者之间的关系。表 4.3 用以说明线程数与中止率之间的关系。表 4.3 的第二列和第三列

分别表示发起的事务总量和被中止的事务量。不论是发起的事务总量还是被中止的事务量都随着线程数量的增加而增加，然而被中止的事务量的增长速率远低于发起的事务量的增长速率。换言之，随着越来越多的线程被创建用来参与运算，被提交的事务量的增长速度远大于被中止的事务量的增长速度。这个趋势与图 4.7 中的线程扩展性曲线相符合。

表 4.3　slr-scm-mcs 方案的中止率随线程变化情况

n	总的事务量（百万）	中止的事务量	中止率（%）
2	156/156	70/66	43.6/42.3
8	228/239	72/59	31.6/21.5
16	303/298	73/64	24.1/19.5
32	382/349	80/67	20.9/19.2
40	399/299	77/74	19.3/24.8
48	495/310	87/78	17.6/25.2
64	522/316	90/89	17.2/26.0

表 4.4 中的数据用以说明中止率随数据集中更新比重的变化情况。线程数为 32，表的第 2 到第 4 列表示的内容与表 4.2 相同，每一列表中记录了两组数据，分别代表初始化值为 1 百万和 1000 时的测试结果。这一次，运行的测试集中更新比重分别为 0，10% 和 80%。处理纯读数据集时，对应的中止率是最低的。这是因为在纯读的场景下，没有写入内存的请求，这样发生数据冲突的概率很小，从而引起的事务中止也相对较少。随着数据集中更新比重的增加，中止率会上升，性能自然受到影响而下降。

表 4.4　slr-scm-mcs 方案的中止率随更新比重变化情况

n	总的事务量（百万）	中止的事务量	中止率（%）
0	346/554	73/60	21.1/10.8
10	320/352	70/66	21.9/18.8
80	282/273	74/70	26.4/25.6

4.6　本章小结

本章首先介绍了事务内存的起源、概念，并对 Intel TSX 提供的两种事务同步扩展指令集——HLE 和 RTM——的使用方法和特性进行了描述。提出了针对 Intel TSX 的两种软件优化方法：SLR 和 SCM。

然后，从对哈希表顺序执行五次操作的实例入手，由浅入深的探讨了当前主流的三种多线程同步编程模式：粗粒度锁，细粒度锁和硬件事务内存，在构建并发哈希表上的优势和劣势。粗粒度锁方法实现简单，易于理解，且容易保障正确性，但是使用这种方法获得的性能不理想。细粒度锁方法能够发挥出多核系统的性能优势，并且有很好的线程扩展性，但是细粒度锁增加了设计并发哈希表的难度，并且需要花费大量精力在保证正确性上。事务内存继承前两者的优势于一身，实现简单，易于验证正确性，又能够获得接近甚至超越细粒度锁方案的性能，是理想的并发编程范式。

接着，介绍了与本章内容密切相关的 MCS 锁的原理。参考基于细粒度锁的缓存行哈希表的设计方法，结合 SLR 和 SCM 两种软件辅助方法实现了基于硬件事务内存的 MCS 锁，并使用这种锁实现了基于 RTM 的缓存行哈希表，并就二者的性能进行了比较，证明了基于 RTM 的缓存行哈希表能够保证良好的线程扩展性和性能，并且在数据集规模较大的场景中，性能比使用细粒度锁方法时更加优秀。

实现了 6 种不同的粗粒度并发哈希，用于进一步的探究使用哪一种方法对基于 HTM 的并发哈希表的优化效果更好。并结合实验结果，论证了在细粒度锁方案已经能够保证良好的性能和线程扩展性的前提下，使用细粒度的基于 HTM 的锁对于性能的提升没有意义。

最后，对影响硬件事务内存性能发挥的因素进行了分析。结合在测试过程中统计的发起的事务总量、被中止的事务量以及中止率等数据，对实验评估中的一些现象进行解释说明。

对基于硬件事务内存的缓存行哈希表的评估结果表明，本章提出的方法达到了预期目标。对于并发哈希表的设计具有重要的意义。

第 5 章　并发 Cuckoo 过滤器的设计

查找或判断一个元素是否存在于一个指定集合中，是计算机科学中一个基本问题。通常会采用线性表（数组或链表）、树（二叉树、堆、红黑树、B+/B-/B* 树）等数据结构存储所有元素，对数据元素进行排序和查找。这里的查找时间复杂度通常都是 O(N) 或 O(log(N))。当集合元素的规模非常庞大时，不仅会降低查找的效率，同时对内存空间的需求也非常大。

在网络安全领域有一个简单的应用场景：判断 URL 是否链接到存在安全隐患的网站。用户在浏览器内输入 URL，浏览器需要判断该 URL 是否是恶意的，它将该 URL 与本地缓存的已知恶意 URL 进行匹配，如果匹配失败，则说明该 URL 是安全的链接，可以正常访问；否则，说明该 URL 可能存在安全隐患。此时，提交请求给远程客户端进行验证，并警告用户该 URL 存在风险。在这个应用场景中，如果缓存的 URL 数量很少，那么使用上述的线性表、树等都可以达到较高的查找效率，同时对内存空间的要求也不高。假设现在需要缓存的 URL 的数量为 10 亿条（这在当前是很常见的一个数量级），每条 URL 的大小为 8 个字节，那么存储所有的 URL 大约需要 8GB 的内存。使用哈希表是一种可能的解决方案。哈希表的查询时间复杂度为 O(1)，可以节省查找的时间，但是没有降低对内存的需求。

事实上，除非有特别的需求，否则判断元素是否在一个指定集合内，并不需要把所有元素的原始信息都保存下来，而只需要保存该元素的“存在状态”，存储存在状态比存储元素本身更能节省空间，它只需要几个 bit 就能表示一个元素。使用哈希函数可以将元素映射成位数组中的一个点，采用 k 个哈希函数将元素映射成 k 个点。这样，经过映射之后，查找元素是否存在时只需读取特定的几个位点的值就能判断某个元素是否存在于集合当中，如果 k 个位置都为 1，则说明该元素可能存在，如果有 1 个位置上为 0，则可以肯定该元素不存在。这样不仅可以大大缩减内存空间，查找速度也非常快。这就是布隆过滤器 (Bloom Filter) 的基本思想。它的名字源自其发明者 Burton.H Bloom[105]。布隆过滤器最初应用于拼写检查和数据库系统。但是，随着互联网的爆炸式发展，海量数据中快速检索目标数据的需求掀起了布隆过滤器的应用与研究的热潮，涌现出新的许多新的布隆过滤器应用和变种[25,28,32,33]。

5.1 布隆过滤器

5.1.1 基本原理

布隆过滤器使用位数组表示元素集合 S，并使用 k 个哈希函数 ($h_1, h_2, ..., h_k$) 来对元素进行位映射。初始状态下的布隆过滤器是一个包含 m 位的位数组，每一位都置 0，图 5.1(a) 所示为 m = 12 的布隆过滤器。当需要将集合 S 中的 n 个元素 x_1，x_2, ... x_n 用位数组表示时，对该元素分别使用 k 个相互独立的哈希函数进行计算，得到位数组上 k 个位置的索引值，随后将映射到位数组的相应位置 1。值得注意的是，如果一个位置被多次置为 1，只有第一次的置位是有效的。图 5.1(b) 表示 k = 3 时，将元素映射到位数组的过程，其中元素 x_1 和 x_2 都对第 5 位置位。图 5.1(c) 表示判断元素 y_i(i = 1,2, ..., n) 是否属于集合。与插入过程类似，同样先对 y 进行 k 次哈希，如果计算得到的索引值对应的位上有任何一位为 0 则表示 y 元素绝对不存在于集合中，只有当所有映射位均为 1 时才表示该元素**有可能**存在于集合当中。

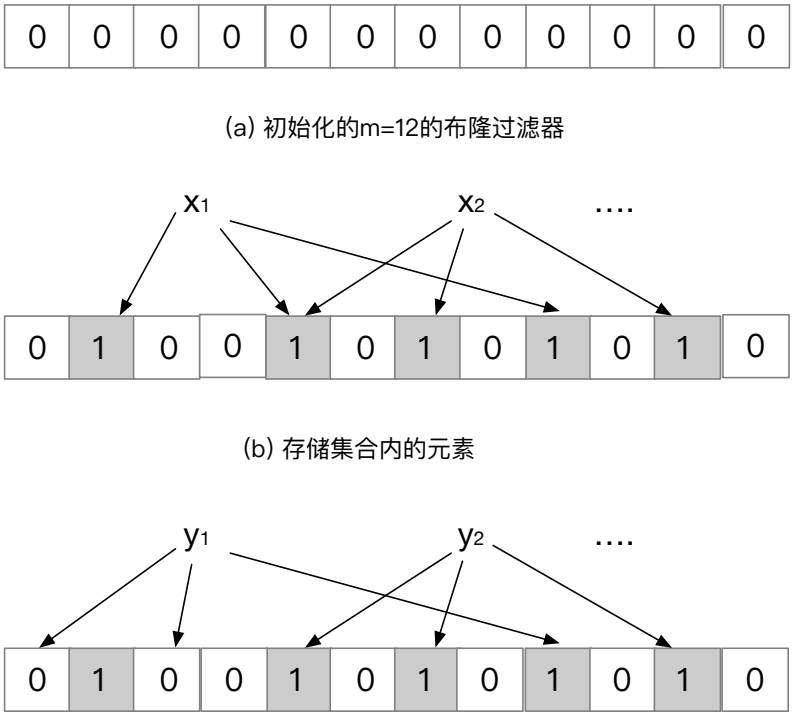


图 5.1 布隆过滤器

换句话说，如果布隆过滤器判断一个元素不在集合中，那肯定就不存在；而如果判断存在，则不一定存在。下文将对这种不一定存在的原因进行说明。这种不能确定元素一定存在的问题是由哈希函数可能发生碰撞的特性所决定的。由此可见，布隆过滤器的高效是以一定的误报为代价的，它通过容忍一定的错误发生

的概率换取存储空间的极大节省。布隆过滤器不适合那些“零错误”的应用场合。

标准的布隆过滤器不支持删除操作。考虑在图 5.1(b) 中删除元素 x_2，意味着将位数组内的第 5、7、11 位置 0，此时如果查询元素 x_1 时会得到该元素不存在的结果，因为它对应的第 5 位上的值在删除 x_2 时被置 0 了。

标准布隆过滤器的实现中有几个重要参数：误判率ϵ、负载因子α、哈希函数的个数 k、位数组大小 m 和集合中元素的个数 n。误判率可通过调整位数组大小或者哈希函数个数进行控制。下面将对这些参数进行分析，以确定实现标准布隆过滤器的最佳参数组合。

5.1.2 误判率估计

在进行正式的误判率估计前先明确几个定义：

定义 5.1 假阳性（false positives）也叫误判，是指当前元素不在集合内，但由于哈希冲突的缘故存在其它元素被映射到部分相同 bit 位上，从而有一定的概率导致在判定该元素时认定其对应的所有位置都为 1，从而判定其在集合内，造成一次误判。这个概率本文称为误判率，误判率用 ϵ 表示。

定义 5.2 假阴性（false negatives），也叫漏报，是指在位数组内删除某个元素时，导致其他元素对应的比特位被置 0，造成本来存在的元素被漏报成不存在。

定义 5.3 负载因子是指集合中存储的元素的个数 n 和布隆过滤器位数组的长度 m 之间的比值，它用 α 表示，其中 $\alpha = n/m$。当 α 为 0 时，表示布隆过滤器为空，α 为 1 表示布隆过滤器满载。

下面进行误判率 ϵ 的计算公式的推导。首先假设布隆过滤器中使用的 k 个哈希函数的计算结果都是均匀分布的，即每个元素都等概率地被哈希到 m 个 bit 位上的任何一个，与其他元素被哈希到的位置无关。则对某一特定 bit 位在一个元素由特定哈希函数插入时没有被置位为 1 的概率 p_1 为：

$$p_1 = 1 - 1/m \tag{5.1}$$

则 k 个哈希函数中都没有一个对其置位的概率 p_2：

$$p_2 = {p_1}^k \tag{5.2}$$

如果插入 n（$n \leqslant m$）个元素，但都未对其置位的概率 p_3:

$$p_3 = {p_2}^n = {p_1}^{kn} \tag{5.3}$$

则此位被置位的概率 p_4 为:

$$p_4 = 1 - (1 - 1/m)^{kn} \tag{5.4}$$

在进行元素查询时，如果待查询元素对应的 k 个位全部置位为 1，则可判定其在集合中。因此将某元素误判的概率为:

$$\epsilon = \left(1 - (1 - 1/m)^{kn}\right)^k \tag{5.5}$$

由 $(1+x)^{1/x} \approx e$ 可知，当 m 很大时，满足 $-1/m \to 0$，可将公式5.5转化为:

$$\epsilon = \left(1 - (1 - 1/m)^{-m\frac{-kn}{m}}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k \tag{5.6}$$

由公式5.6可以初步断定 ϵ 由元素的个数 n 和位数组的长度 m 决定，增大 n 或者减小 m 都会导致 ϵ 的升高。m 和 n 的比值就是布隆过滤器的空间开销。这种计算方法不严格，因为前面假设哈希函数和散列后值的分布是相互独立的。但是，这个假设随着 m 和 n 的增大误判率更接近真实的误判率。Mitzenmacher 证明无假设情况下的误判率的期望值相同[130]。

5.1.3 最优哈希函数个数

哈希函数的选择对于布隆过滤器的性能以及空间利用率都有至关重要的作用。对于选取什么样的哈希函数已经在前文中有过介绍，这里不再赘述。而对于哈希函数的个数，直观的认为越多越好。实际上，哈希函数越多，用于表达集合中每一个元素所需要的位数就越多，这与布隆过滤器用较低的误判率换取空间的高效利用的初衷相悖。那么哈希函数的个数 k 应该满足什么条件才能发挥最佳性能呢?

下面推导对给定的 α，k 满足什么条件可以使 ϵ 最小化。

令:

$$f(k) = \left(1 - e^{-nk/m}\right)^k \tag{5.7}$$

取 $b = e^{n/m}$，得:

$$f(k) = \left(1 - b^{-k}\right)^k \tag{5.8}$$

两边取对数得:

$$lnf(k) = kln\left(1 - b^{-k}\right) \tag{5.9}$$

函数两边对 k 求导得:

$$1/f(k)\cdot f'(k)=ln(1-b^{-k})+k\frac{b^{-k}\cdot lnb}{1-b^{-k}} \tag{5.10}$$

对式 5.10右边求最值。
令:

$$\begin{aligned}\frac{1}{f(k)}\cdot f'(k)=ln(1-b^{-k})+k\cdot\frac{b^{-k}\cdot lnb}{1-b^{-k}}=0\\ \Rightarrow(1-b^{-k})ln(1-b^{-k})=-k\cdot b^{-k}\cdot lnb\\ \Rightarrow 1-b^{-k}=b^{-k}\\ \Rightarrow e^{-\frac{kn}{m}}=\frac{1}{2}\\ \Rightarrow k=ln2\cdot m/n\approx 0.7m/n\end{aligned} \tag{5.11}$$

因此，对于固定的 α，当 $k=0.7m/n$ 时具有最低的误判率，此时 ϵ 等于:

$$(1-1/2)^k=2^{-ln2\cdot\frac{m}{n}}\approx 0.62m/n \tag{5.12}$$

5.1.4 最优位数组长度

对于给定的误判率上限，布隆过滤器至少需要多少位才能表示全集中任意的 x 个元素的集合? 假设全集中元素的个数为 n，最大误判率为 ϵ。以此为前提展开对位数组大小应满足的条件进行分析。

假设 S_n 为全集中任取 n 个元素的集合，B 是用于表示 S_n 的位数组。那么对于集合 S_n 中任意一个元素 x，在 B 中查询 x 都能得到肯定的结果，即 B 能够接受 x。显然，由于布隆过滤器允许一定的误判率，B 能够接受的不仅仅是 S_n 中的元素，它还允许最多 $\epsilon\cdot(u-n)$ 个误判。因此，对于一个确定的位数组来说，它能够接受总共 $n+\epsilon\cdot(u-n)$ 个元素。在 $n+\epsilon\cdot(u-n)$ 个元素中，B 真正表示的只有其中 n 个，所以一个确定的位数组可以表示:

$$\binom{n+\epsilon\cdot(u-n)}{n} \tag{5.13}$$

个集合。m 位的位数组一共有 2^m 个不同的组合，可以进一步推导 m 位的位数组可以表示:

$$2^m\binom{n+\epsilon*(u-n)}{n} \tag{5.14}$$

个集合。全集中包含 n 个元素的子集总共有:

$$\binom{u}{n} \tag{5.15}$$

个。因此，要让布隆过滤器的位数组大小能够满足所有包含 n 个元素的子集，必须满足:

$$2^m \cdot \binom{n+\epsilon\cdot(u-n)}{n} \geqslant \binom{u}{n} \tag{5.16}$$

即 m 需要满足:

$$m \geqslant log_2(\binom{u}{n}/\binom{n+\epsilon\cdot(u-n)}{n}) \geqslant log_2(\binom{u}{n}/\binom{\epsilon u}{n}) \approx log_2\epsilon^{-n} = nlog_2(1/\epsilon) \tag{5.17}$$

式 5.17中近似相等有个重要的前提条件：n 远小于 $\epsilon\cdot u$，这个前提在处理实际问题中通常是容易被满足的。根据式 5.17中的不等式，得到如下结论：在误判率上限为 ϵ 的情况下，m 至少要等于 $nlog_2 1/\epsilon$ 才能表示任意 n 个元素的集合。

在本章 5.1.3中推导出哈希函数的个数 k 等于 $k = 0.7m/n$ 时可以得到最小误判率，此时的误判率为 $(\frac{1}{2})^{0.7m/n}$。令 $(\frac{1}{2})^{0.7m/n} \leqslant \epsilon$，可以进一步推导:

$$m \geqslant n\frac{log_2(1/\epsilon)}{ln2} = nlog_2e * log_2(1/\epsilon) \approx 1.44n\cdot log_2(1/\epsilon) \tag{5.18}$$

式 5.18说明当 k 取到最优值时，要保证误判率不超过 ϵ，m 至少要取到最小值的 1.44 倍。可以验证，当给定 $\epsilon = 0.01$ 时，存储每个元素需要 9.6 比特。而 $\epsilon = 0.001$ 时，每个元素需要额外的增加 4.8 比特。所以，在实际的应用中，对于误判的容忍度不同，要求误判率越低，则存储每个元素需要的比特位越多，相同位数组长度能存储的元素个数就越少。

5.2 Cuckoo 过滤器的设计

传统的布隆过滤器的空间效率高，对插入和查询元素的处理也相当快。但是它仍然有不足之处——不支持元素的删除操作。除非能实现没有碰撞的哈希函数，否则布隆过滤器对元素是否存在的误判难以避免。研究人员能做的就是尽量选择均匀的哈希函数，并且借助一些数据结构的特性有效地对碰撞进行处理。在上一节介绍了误判率每缩小到原来的十分之一，至少要增加 4.8 个比特位用于表示一个元素。另外，现有的实现布隆过滤器元素删除操作的方法使用计数器等方法，将每个比特位都扩张成一个计数值，删除元素时，将对应位置的计数值减 1，这种方法增加了空间开销。所以，布隆过滤器实现更低的误判率和实现删除操作

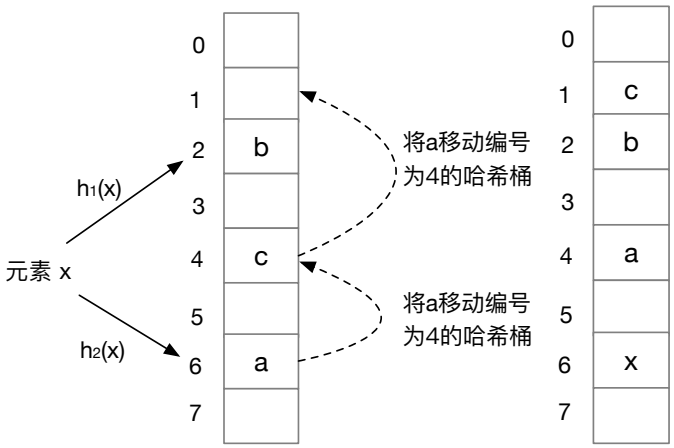
都需要牺牲一定的空间效率。

为了解决上述问题，Bin Fan 等人利用 Cuckoo 哈希方法的特性实现了 Cuckoo 过滤器[131]。它实现了在布隆过滤器内动态的插入和删除元素，而不会造成漏报。为了实现动态的插入和删除元素，Cuckoo 过滤器采用的是一种称为**不完整键值 (partial-key)** Cuckoo 哈希的技术。Cuckoo 过滤器是标准 Cuckoo 哈希表在集合元素查询算法领域的应用。

5.2.1 Cuckoo 过滤器的数据结构

Cuckoo 过滤器在基本数据结构上与 Cuckoo 哈希表类似，因此在介绍 Cuckoo 过滤器的构造过程之前，先简单回顾一下 Cuckoo 哈希表的构造过程。Cuckoo 哈希设置了两个哈希函数，分别记作 $h_1(x)$ 和 $h_2(x)$，在插入元素时，每个元素有两个备选的哈希桶。在进行元素查询操作时，检查对应的两个哈希桶内是否存储有该元素。图 5.2（a）所示为在一个含有 8 个哈希桶的 Cuckoo 哈希表内插入新元素 x 的过程，通过哈希函数计算之后，元素 x 可以被存储到编号为 2 或者 6 的哈希桶内。如果元素 x 对应的两个哈希桶都有空闲位置，则随机选取一个位置将 x 插入，然后结束该次插入操作。如果两个对应的哈希桶都已经饱和，则将按照图 5.2（a）所示的过程选择一个备选哈希桶（如编号为 6 的哈希桶），然后将该哈希桶内的元素（对应元素 a）踢出，将 a 移动到 $h_1(a)$ 或 $h_2(a)$ 对应的哈希桶内（如果 a 当前存储的桶编号为 $h_1(a)$，则 a 将移动到桶编号为 $h_2(a)$ 的哈希桶内，反之亦然。本例中是从编号为 6 的哈希桶移动到编号为 4 的哈希桶），若编号为 4 的哈希桶也没有空闲位置，则继续将元素 c 移动到它对应的另一个备选哈希桶内（过程与移动元素 a 一样）。一直重复上述过程，直到所有的元素都找到合适的存储位置，或者进行元素移动的次数达到了预先设定的上限值为止（一般地上限值设置为 500）。插入元素 x 之后，Cuckoo 哈希表如图 5.2（b）所示。如果在某次插入元素时，经过了 500 次的元素移动仍然有元素没有找到合适的存储位置，则判定哈希表饱和。

Cuckoo 过滤器构造过程与 Cuckoo 哈希表类似：首先创建一个空的 Cuckoo 哈希表；然后将集合内的元素插入到哈希表内。如图 5.1所示，标准的布隆过滤器是一个大数组，它表示元素时需要使用 k 个哈希函数将元素映射到过滤器的 k 个位置上，对相应的位置置 1；判定元素是否存在时，也是通过使用 k 个哈希函数分别进行运算得到 k 个索引值，然后判断对应的 k 个位置是否全为 1。而 Cuckoo 过滤器很好的利用了哈希表的特性，只需使用一个哈希函数将元素表示成一个固定长度的哈希值，然后按照哈希表插入元素的方法插入即可。这个固定长度的哈希值在 Cuckoo 过滤器中被称为指纹，指纹的长度用 f 表示。其插入元素的过程与标准的 Cuckoo 哈希表类似，区别在于元素以指纹的形式存储。元素以指纹信息的形式存储在过滤器内，有利于提高空间利用率，但是同时也会产生一个问题：



(a) 插入 x 之前的哈希表　　(b) 插入 x 之后的哈希表

图 5.2　Cuckoo 哈希表元素插入过程

如果发生如图 5.2(a) 所示的现象，需要为新插入的元素腾出空间而移动旧元素时，无法根据旧元素的指纹信息得到它的备选哈希桶的偏移地址。解决这个问题的方法使用的是一种称之为不完整键 Cuckoo 哈希（partial-key cuckoo hashing）的方法，关于该方法的详细介绍在 (1)小节中给出。值得说明的是，计算元素指纹信息所用的哈希函数 $f(x)$ 与计算元素哈希桶索引值所用的两个哈希函数是不相同的。构造 Cuckoo 过滤器的过程如图 5.3所示。为了提高空间利用率和元素查询的速度，Cuckoo 过滤器采用（2，4）组相连结构，即使用两个哈希函数，每个哈希桶能容纳四个元素。

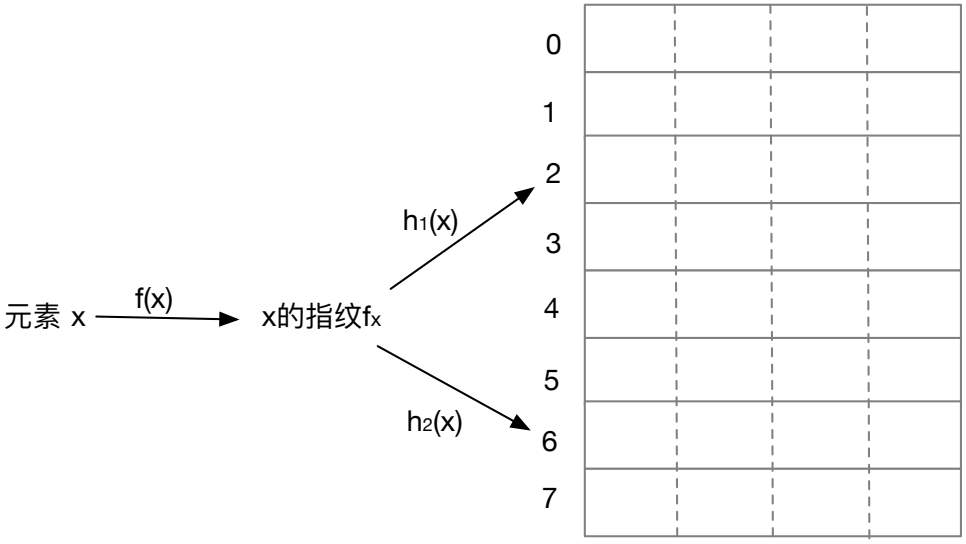


图 5.3　（2，4）组相联 Cuckoo 过滤器

5.2.2 Cuckoo 过滤器的参数

这部分重点对 Cuckoo 过滤器有重大影响的几个参数的理论计算过程进行分析。方便起见，表 5.1列出了本小节所用到的一些关键参数及其代表的含义。

表 5.1 相关符号及其含义

参数	含义
ϵ	误判率
f	指纹信息的长度 (单位：bit)
α	负载因子 ($0 \leqslant \alpha \leqslant 1$)
b	每个哈希桶内包含的实体的数量
m	哈希桶的数量
n	元素的数量
C	表达一个元素所需的平均位数 (单位：bit)

(1) 指纹信息的长度

这部分将通过理论分析指出，使用不完整键 Cuckoo 哈希方法将元素以指纹的形式存储在 Cuckoo 过滤器内的方法会导致指纹信息的长度下界值会随着过滤器的规模增加而缓慢增长。这一点与标准的布隆过滤器不同，标准布隆过滤器每个元素使用多少位进行表示（k 值的设定）只取决于预定的误判率。直观的看，这似乎是 Cuckoo 过滤器的劣势，但是实验结果表明，这一点对 Cuckoo 过滤器性能的影响微乎其微。实验结果表明，当每个哈希桶能够存储 4 条长度大于等于 6 bits 的指纹信息时，存储 40 亿个元素的哈希表的空间占用率达到 95% 以上。

最小指纹信息长度：在 Cuckoo 过滤器中，对于一个给定的元素，根据其当前位置和指纹信息使用不完整键哈希方法可以计算出它的备选哈希桶的索引值。这样，每个元素的候选哈希桶都不是独立的。比如，某一元素可以存放在桶 i_1 或 i_2 中，对于长度为 f 比特的指纹信息，根据公式 5.24，i_2 可能的索引值最多有 2^f 种可能。若指纹信息的长度为一个字节，对给定的 i_1，i_2 最多只能偏离 i_1 256 个位置。对于具有 m 个桶的哈希表而言，当 $2^f \leqslant m$ 时，i_2 能够选择的哈希桶的范围只是整个 m 个哈希桶的一个很小的子集。

直观上看如果指纹信息的长度足够长，不完整 Cuckoo 哈希仍然能够接近标准 Cuckoo 哈希的冲突处理能力。然而，如果哈希表非常大，而此时指纹信息的长度相对来说要远远小于哈希表的大小，这样容易引起更多的哈希碰撞，从而导致元素插入失败的概率升高。当 Cuckoo 过滤器需要处理大量元素，而 ϵ 设定一个中等偏低的值时，可能发生上述情形。接下来，将通过分析确定插入失败的概率下限。

首先分析对于给定的 u 个元素，它们恰好映射到相同的两个哈希桶内的概率 p_1。假设第一个元素 x 位于桶 i_1 内，并且指纹为 t_x。如果其他的 $u-1$ 个元素具有与 x 相同的桶索引值，它们必然满足以下两个条件：（1）它们的指纹都为 t_x，出现的概率为 $\frac{1}{2^f}$；（2）它们第一个桶索引值为 i_1 或者 $i_1 \bigoplus h(t_x)$，出现的概率为 $\frac{2}{m}$。因此，u 个元素映射到相同的两个桶内的概率 $p_1 = (\frac{2}{m} * \frac{1}{2^f})^{u-1}$

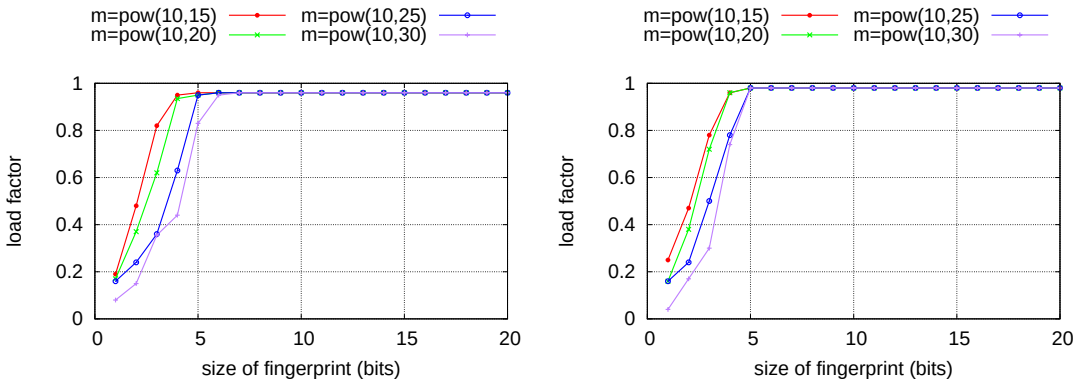
现在考虑构建 Cuckoo 过滤器的随机插入 n 个元素的构建过程。假设初始化的哈希表桶数组满足 $m = cn$，每个哈希桶容纳的元素个数为 b，其中 c 为常数。当出现 $u = 2b+1$ 个元素被映射到相同的两个桶内时，表明插入失败。出现这种情形的概率为插入失败的概率下界。由于从 n 个元素中包含 $2b+1$ 个元素的子集有 $\binom{n}{2b+1}$ 个，$2b+1$ 个元素在插入过程中发生碰撞的期望值为：

$$\binom{n}{2b+1}(\frac{2}{2^f m})^{2b} = \binom{n}{2b+1}(\frac{2}{2^f cn})^{2b} = \Omega(\frac{n}{4^{bf}}) \tag{5.19}$$

因此，由式 5.19可以做出结论，4^{bf} 必须满足 $\Omega(n)$ 才能避免异常的插入失败的概率。指纹信息的长度最好为 $f = \Omega(log(\frac{n}{b}))$ 比特。在第 5.1.4节中指出标准的布隆过滤器用于表示每个元素所需的比特数为常数（近似等于 $ln(\frac{1}{\epsilon})$）。而 Cuckoo 过滤器指纹信息所需的长度为 $\Omega(logn)$ 这个级别，这个结果看上去似乎不是特别理想。这会不会引起扩展性问题呢？实验表明桶容量 b 在下界约束中的起决定作用：只要将 b 控制在合理的大小，指纹信息的长度仍然可以保持较小的值。

实验评估：图 5.4所示为负载因子 α 与指纹信息长度 f，哈希桶容量 b 以及哈希表总的哈希桶数量 m 的数量关系曲线。x 轴表示指纹信息的长度（$1 \leqslant f \leqslant 20$），$y$ 轴表示负载因子（$0 \leqslant \alpha \leqslant 1$）。图 5.4(a)、(b) 所示分别为 $b = 4, 8$ 时指纹信息长度与负载因子的变化关系。实验时，插入的元素为 64 位的随机值。认定哈希表“饱和”的条件是：当执行某次插入操作时，进行了多达 500 次的替换操作之后仍没有找到空闲的位置接纳被踢出的元素。当哈希表达到”饱和“之后，停止本次测试并记录此时哈希表的负载因子。每组参数运行十次，最终结果取十次的平均值。

如图 5.4所示，在所有的参数组合中，$b = 4$ 的过滤器的哈希表的利用率能够达到 95%，而 $b = 8$ 的过滤器在指纹信息大小足够长的前提下哈希表利用率可以达到 98%。利用率达到这些值后，增加指纹信息的长度对哈希表利用率的提升几乎没有帮助（但是可以降低误判率）。通过前文的理论分析表明，当过滤器的规模增大时，所需的 f 的最小值会发生变化。此外，通过比较图 5.4(a) 和图 5.4(b) 发现达到高哈希表利用率的 f 的最小值随着哈希桶容量 b 的增加而减小，这个规律同样也与前文的理论分析的结果吻合。在本次实验中，使用两个完全独立的哈希函数，当 $b = 4$ 且 $m = 2^{30}$ 时，哈希表最多能存储多达 40 亿个元素，而当指纹信息的长度大于 6 比特时，α 接近” 最优负载因子”。



(a) $b=4$ (b) $b=8$

图 5.4　负载因子与指纹信息长度变化关系

启示： 通过结合公式 5.19对 f 的下界约束推导的结果与图 5.4中的实验结果可以总结出 Cuckoo 过滤器的一条非常重要的结论。在理论上 Cuckoo 过滤器的空间效率要比标准布隆过滤器“差”——$\Omega(log\ n)$ 与常数的区别。对于布隆过滤器而言，不论哈希表存储的元素个数是一千、一百万还是数十亿，达到百分之一的误判率大约都只需要 10 个比特位来表示每个元素。而为了保持相同的空间效率 Cuckoo 过滤器需要使用更多的比特位表示每条指纹信息。同样的由理论推算过程可知，f 为 $\Omega(log\ n)$ 比特，如果 b 足够大，则 f 的值增长非常缓慢，在实际的应用中，可以将其视为常数。图 5.4的结果表明 6 比特的指纹信息足够存储数十亿个元素，并且能够达到很高的哈希表利用率。

(2)　空间效率

对 Cuckoo 过滤器内的元素进行增、删、查操作与每个哈希桶内包含多少实体无关。但是，为 Cuckoo 过滤器选择正确的参数对于空间效率具有重要意义。这一部分着重介绍如何选取合适的参数优化 Cuckoo 过滤器的空间效率。

空间效率是通过计算在完整的过滤器中用于表示每个元素所用的平均比特数来衡量的。用哈希表长度（比特）除以过滤器有效存储的元素的个数就是表示每个元素所用的平均比特数。尽管每个实体可以存储一条指纹信息，但是并不是所有的实体都已经存入了指纹信息——过滤器的哈希表内一定有空闲的实体。所以，每个元素实际上需要的比特数大于指纹信息的长度。如果每条指纹信息的长度为 f 比特，哈希表的负载因子为 α，C 的单位为：bits/item，每个元素的空间开销 C 可以用式 5.20表示。

$$C=\frac{\text{哈希表的比特数}}{\text{存储的元素个数}}=\frac{f\cdot\text{实体的数量}}{\alpha\cdot\text{实体的数量}}=\frac{f}{\alpha}\text{bits.} \tag{5.20}$$

在前文中有过介绍，指纹信息的长度和负载因子都与哈希桶的大小有关。下

面研究在给定的误判率 ϵ 前提下，如何通过设置哈希桶的容量 b 使 C 达到最小。

保持 Cuckoo 过滤器的总比特数为常量，调整哈希桶的容量会产生两方面的影响：**第一，哈希桶容纳的实体数量越多，哈希表的空间利用率越高。**对于使用两个哈希函数的 Cuckoo 过滤器，当桶能容纳的实体数 $b=1$ 时，哈希表的负载因子 α 为 50%，而当 $b=2,4,8$ 时，α 分别为 84%，95% 和 98%。

第二，哈希桶的容量越大，维持相同误判率需要的指纹信息的长度越长。哈希桶的容量越大，进行查询时需要检查更多的实体，发现相同指纹信息的概率也会增加。在最坏情况下，查询一个并不存在的元素需要探测两个分别包含了 b 个实体的桶（当然并不是所有的哈希桶内都填满了实体，这里只是考虑最糟糕的情况；当哈希表的负载因子达到 95% 时，已经很接近极限情况）。对于每一个实体而言，一次查询与存储的指纹相匹配并且返回成功匹配被误判的概率最多为 $1/2^f$。在进行 $2b$ 次这样的比较之后，误判率的上界为：

$$1-(1-1/2^f)^{2b} \approx 2b/2^f \tag{5.21}$$

该上界约束与哈希桶的容量 b 成正比。为了保证预定的误判率 ϵ 不变，必须确保 $2b/2^f \leqslant \epsilon$，保证这个条件的最小指纹信息长度为：

$$f \geqslant \lceil log_2(2b/\epsilon) \rceil = \lceil log_2(1/\epsilon) + log_2(2b) \rceil \tag{5.22}$$

由式 5.20和5.22可以推算存储每个元素的空间开销 C 受下面条件的限制：

$$C \geqslant \lceil log_2(1/\epsilon) + log_2(2b) \rceil / \alpha \tag{5.23}$$

α 随着 b 的增加而增加。当 $b=4$ 时，$\alpha=0.95$，$1/\alpha \approx 1.05$。此时 $C = 1.05log_2(1/\epsilon) + 1.05*3$。式 5.23表明，当负载因子一定时，Cuckoo 过滤器的空间开销要低于布隆过滤器 ($1.44log_2(1/\epsilon)$)。

为了确定最优的哈希桶容量 b，下面将通过实验比较参数 b 为不同的值时的空间效率。用不完整键 Cuckoo 哈希方法构造具有不同的指纹信息长度的哈希表，分别记录对应的平均空间开销和误判率。结果如图 5.5所示。使空间效率最好的 b 的取值依赖于预定的误判率 ϵ：当 $\epsilon > 0.002$ 时，$b=2$ 对应的平均空间开销要略好于 $b=4$ 对应的空间开销；而当 $10^{-5} < \epsilon \leqslant 0.002$ 时，$b=4$ 具有最小的空间开销。

综上所述，Cuckoo 过滤器的默认参数配置为 (2，4)，即每个元素有两个候选的哈希桶，每个哈希桶最多能够容纳 4 条指纹信息。选取这组参数作为默认配置的原因有两个方面：一是实际的应用一般都要求误判率满足：$10^{-5} < \epsilon \leqslant 0.002$[30]；二是这个参数组合能够提供最优综合性能。

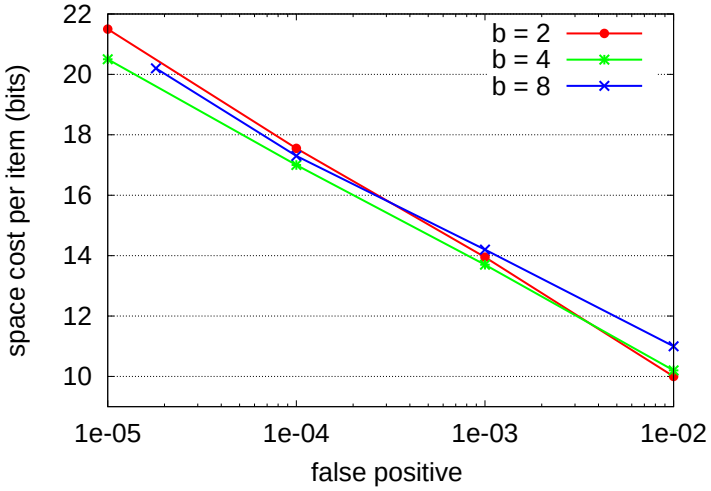


图 5.5　哈希桶容量与空间效率的关系

5.3　并发 Cuckoo 过滤器

无论是标准的布隆过滤器还是现有的其他布隆过滤器的扩展版本，都具有在单核平台上快速处理元素的能力以及高效的空间利用率。单核处理器的计算能力已达到瓶颈，相对而言多核计算机的计算资源和计算能力更加充裕。在当前数据呈现爆炸式增长的背景下，海量数据处理压力越来越大，基于单核处理器的过滤器逐渐显得捉襟见肘。因此，设计基于多核系统并发过滤器对于海量数据处理是具有重大实际意义的工作。然而，当前的相关研究中并没有一款支持并发查询和更新的过滤器。

并发控制机制是在多核平台上设计并发数据结构的关键环节，它对线程扩展性和整体性能起决定作用。在前面对并发哈希表的评估与分析中通过比较各个并发哈希表的线程同步方式发现，不当使用共享变量以及使用 TAS 锁不利于并发哈希表的线程扩展性。本文设计的并发 Cuckoo 过滤器的基础数据结构本质上仍然是哈希表，不同的是 Cuckoo 过滤器根据处理集合成员查询问题的特征，改用存储不完整键值替换标准 Cuckoo 哈希表中存储完整的键值信息的做法，换取存储空间的极大节省。因此，前文通过实验评估得出的结论仍然适用于并发 Cuckoo 过滤器。这一部分主要介绍并发 Cuckoo 过滤器的实现过程和性能评估结果。

5.3.1　自旋锁

在设计并发数据结构时，如果临界区非常小（比如，只包含几条或者几十条指令），则可以考虑使用时旋锁（spinlock）。因为使用普通的互斥锁会涉及到操作系统的调度。自旋锁的工作方式是让竞争的线程不断地读取一个变量（locked）的状态，判断是否满足进入临界区的条件。在前文的第 4.4.3小节中，介绍了一种名为 MCS 的自旋锁。这里再介绍几种常用的自旋锁——TAS 锁，TTAS 锁和 CLH 锁，并解释选择 MCS 锁作为实现并发 Cuckoo 过滤器的原因。

TAS（test-and-set）锁是一种简单的自旋锁。其实现代码如算法 5.1所示。test_and_set() 操作用于早期的多处理器架构上解决同步问题，它利用 CPU 提供的指令（如 X86 的 xchg，LOCK 指令前缀等）对内存地址（或者字节）进行操作。该内存地址上存储的是一个布尔变量，值为 $true$ 或 $false$。TAS 指令原子地完成以下操作：写入 true 到这个地址，并返回这个地址上存储的旧值。从表面上看，TAS 指令是实现自旋锁的理想选择。当 lock 的值为 $false$ 时，表明当前锁处于空闲状态，可以被占有，当值为 $true$ 时，则表明当前锁正在被其它线程占有。$spin\_lock()$ 方法不断的重复执行 TAS 指令，直到返回 $false$。释放锁时，调用 $spin\_unlock()$，将 $lock$ 的值设置为 $false$。

算法 5.1: TAS 锁

```
1 Function spin_lock (lock)
2 |   while (test_and_set(lock, true));
3 Function spin_unlock (lock)
4 |   atomic_set(lock, false);
```

实际上在 SMP 系统上，TAS 锁的性能不理想。原因有两个：其一，TAS 中，每次执行锁的 get_and_set() 方法，会独占总线以发送锁的状态变量被修改的消息，导致其它处理器执行 get_and_set() 时都需要等待，出现串行执行。其二，每一次执行 TAS 都必须读-写 $lock$ 变量，这涉及到多个 CPU 之间的缓存一致性问题。比如，当 CPU P_1 读取 $lock$ 时，如果 $lock$ 不在 P_1 的缓存内，则需要从主存中读入。考虑到 P_1 需要修改 $lock$，所以将 $lock$ 从主存读入缓存后，需要将其它 CPU 上缓存的副本失效（Invalidate）。存在多个 CPU 竞争同一个 TAS 锁时，每一次 TAS 操作都需要完成以上操作，在系统总线上会产生大量的一致性流量。此外，在释放锁时，由于总线一直被执行 test_and_set() 方法的处理器占用，导致释放锁被延迟。因此，随着 CPU 数目的增多，性能衰减得很快。

TTAS（test-and-test-and-set）锁是在 TAS 锁的基础上进行改进的一种自旋锁。当线程 A 持有锁时，TTAS 锁算法的行为如下：当线程 B 第一次读取锁时，发生缓存未命中，强制使线程 B 在加载变量 $lock$ 到它的缓存时阻塞。只要 A 持有锁，B 不断的重复读取 $lock$ 变量，总是能在它的缓存中命中。这样 B 不会造成任何的总线流量，也不会导致其它线程的内存访问延迟。算法 5.2简单描述了其实现方式。

但是 TTAS 锁在释放锁时会出现问题：持有锁的线程通过修改 $lock$ 变量的值为 $false$ 释放锁，这会导致正在进行自旋的线程的缓存副本立即失效。每个在 $lock$ 上自旋的线程都发生一次缓存未命中，重新从主存中读入新的值，然后调用 test_and_set() 方法获取锁。第一个成功获取锁的线程会使其它线程的缓存行失效，这些线程必须重新读入 $lock$ 变量，这将引起总线上的流量风暴（storm of bus

traffic）。

算法 5.2: TTAS 锁

```
1 Function spin_lock (lock)
2     while test_and_set(lock, true) do
3         while (lock  != false);
4     end
```

CLH（Craig, Landin, and Hagersten locks）[50,51]：也是一种自旋锁，与 MCS 锁一样，都是以其发明人名字的首字母命名，名字本身不涉及它的实现原理。CLH 锁能确保无饥饿性，提供先来先服务的公平性。它是一种基于链表的可扩展的、高性能、公平的自旋锁，申请线程只在本地变量上自旋，不断轮询前驱的状态，如果发现前驱节点释放了锁，就结束自旋进入临界区。CLH 队列中的节点 QNode 中含有一个 locked 字段，该字段若为 $true$ 表示该线程需要获取锁，且不释放锁，为 $false$ 表示线程释放了锁。结点之间是通过隐形的链表相连，之所以叫隐形的链表是因为这些结点之间没有明显的 next 指针，而是通过 myPred 所指向的节点的变化情况来影响 myNode 的行为。CLH 锁上有一个 tail 指针，始终指向队列的最后一个节点。

当一个线程需要获取锁时，会创建一个新的 QNode，将其中的 locked 设置为 true 表示需要获取锁，然后线程修改 tail 指针，使自己插入到队列的尾部，同时获取一个指向其前趋的引用 myPred，然后该线程就在前趋结点的 locked 字段上自旋，直到前趋结点释放锁。当一个线程需要释放锁时，将当前结点的 locked 域设置为 $false$，同时回收前趋结点。如图 5.6所示，线程 A 需要获取锁，其 myNode 域为 true，此时 tail 指向线程 A 的结点，随后线程 B 插入到线程 A 后面，tail 指向线程 B 的节点。然后线程 A 和 B 都在它的 myPred 域上旋转，一旦它的 myPred 结点的 locked 字段变为 false，它就可以获取锁执行。图中线程 A 的 myPred 的 locked 域为 $false$，表明 A 的前驱节点释放了锁，并且将 A 的 locked 域设置为 $false$，线程 A 结束自旋进入临界区。

CLH 队列锁的优点是空间复杂度低（如果有 n 个线程，L 个锁，每个线程每次只获取一个锁，那么需要的存储空间是 O（L+n），n 个线程有 n 个 myNode，L 个锁有 L 个 tail）。它的缺点是：在 NUMA 架构下，每个线程都有自己的内存，如果前趋节点的内存位置比较远，自旋判断前趋节点的 locked 域，性能将大打折扣，但是在 SMP 系统结构下 CLH 的性能还是可以保障的。

综合比较上述几种自旋锁的优缺点，在实现并发 Cuckoo 过滤器时决定采用 MCS 锁。

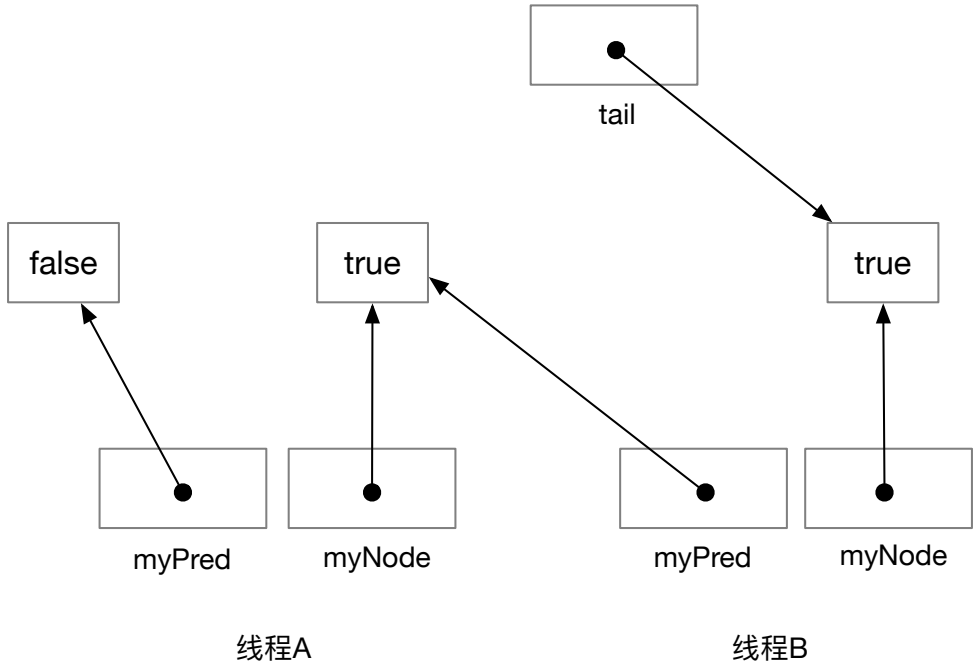


图 5.6　CLH 锁结构

5.3.2　加锁与解锁

接下来的内容介绍如何通过基于 Intel RTM 的 MCS 锁实现多线程并发的 Cuckoo 过滤器。实现 Cuckoo 过滤器的多线程并发使用的是读写锁，算法 5.3给出了基于 Intel RTM 的 MCS 锁算法实现。使用 RTM Retry 机制来消除 TSX Lemming 效应对性能的抑制。其中结构体 $spe\_mcs\_lock\_t$ 是一个对齐的 mcs 锁字段，而结构体 $locklib\_mutex\_t$ 是一个互斥锁字段，它包含了 mcs 锁字段、一个 $uint8\_t$ 类型的 $mode$ 变量以及一个用于缓存行对齐的字符串的结构体。函数 $locklib\_mutex\_lock()$ 具有两个参数，一个为互斥量 $mutex$，一个为整型数 $mode$。$mode$ 的值分别对应 0、1，0 表示当前持有锁的线程为读者线程，1 表示当前持有锁的线程为写者线程（执行的是删除或插入操作）。对临界区的保护设置了 $speculative\_path$，$fallback\_path$ 两条执行路径：一条为事务化推测执行路径（第 16 行）；一条为回退路径（第 21 行），即当事务执行失败后，临界区申请标准锁完成本次操作。在事务化推测执行期间，线程首先不会申请获取锁，直到执行完成准备提交时再申请获取锁，如果申请的锁被占用，则该线程所执行的事务被中止，线程对系统状态所做的更改都失效，系统回退到初始状态并跳转到回退路径执行（事务内存的原子性）。

跳转到回退路径之后，如果当前重试的次数没有达到预先设定的门限值，将继续尝试进行事务化推测执行（第 23-26 行）。如果当前重试次数已经达到了门限值，则使用标准的锁方法完成操作（第 29-35 行）。

由于硬件事务内存不能保证每次事务化执行都能成功提交对系统状态的更改。为了避免进程悬停，在使用硬件事务内存进行锁省略编程时设置回退路径是

算法 5.3: 基于 Intel RTM 的 MCS 锁算法

```
1  Function locklib_mutex_lock (locklib_mutex_t *mutex, uint8_t mode)
2      spec_mcs_lock_t *lock = (spec_mcs_lock_t *) mutex;
3      reason = 0 ;
4      speculative_path:
5      XBEGIN(fallback_path, reason);
6      if lock is locked then
7          XABORT(1);
8      end
9      return 0;
10
11     fallback_path:  retries = retries + 1;
12     while lock is locked do
13         cpu_relax();
14     end
15     if retries < MAX_RETRIES then
16         goto speculative_path;
17     end
       /* 以标准方式申请辅助锁                                          */
18     my_node.locked = true;
19     qnode_t = __sync_lock_TAS(&lock->lock, &my_node);
20     if prev != NULL then
21         prev->next = &my_node;
22         while my_node.locked is true do
23             cpu_relax();
24         end
25     end
26     mutex->mode = mode;
27     return 0;
```

有效的保障程序顺利执行的手段。一般的，事务代码在经过一定次数的重试之后成功提交的概率远远大于失败的概率，所以执行回退路径对性能的影响是可控的。

算法 5.4: 基于 Intel RTM 的 MCS 解锁算法

```
1  Function locklib_mutex_unlock(locklib_mutex_t *mutex)
2      spec_mcs_lock_t *lock = (spec_mcs_lock_t *) mutex;
3      if XTEST() then
4          XEND();
5      else
           /* 使用标准的方式进行解锁                                    */
6          qnode_t *last = my_node.next;
7          if last == NULL then
8              if true == __sync_bool_CAS(&lock->lock, &my_node, NULL) then
9                  return 0;
10             end
11             while (last = my_node.next) == NULL do
12                 cpu_relax();
13             end
14         end
15         my_node.next = NULL;
16         last->locked = false;
17     end
18     retries = 0;
19     return 0;
```

算法 5.4展示了对应的解锁过程。与加锁过程的两条路径相对应，释放锁的过程也分为两个阶段：首先使用 *XTEST* 判断当前执行的操作是否为事务执行，若为事务执行，使用 *XEND* 结束；若判断此次操作申请的锁是通过标准的方式获取的，则按照标准锁的释放过程释放锁。最后将 *retries* 变量清零。

5.3.3 并发访问接口

并发 Cuckoo 过滤器的开发和测试使用 C++。在这一部分中，将对并发 Cuckoo 过滤器 **HashTable** 类的主要 API 进行介绍。表 5.2 列出了 Cuckoo 过滤器的 *HashTable* 类的主要 API。序号 3-5 对应的 API 主要用于统计哈希表的信息，用于最终计算哈希表的负载因子，内存消耗等；序号 6 对应的 API 输出指纹信息的长

表 5.2　HashTable 类的成员函数列表

序号	API	描述
1	explicit **HashTable(num)**	构造函数
2	~ **HashTable()**	析构函数
3	size_t **NumBuckets()**	返回哈希桶数量
4	size_t **SizeInBytes()**	返回哈希表大小 (单位:Bytes)
5	size_t **SizeInTags()**	返回哈希表容纳的指纹的数量
6	string **Info()**	返回哈希表的容量信息
7	uint32_t **ReadTags(i, j)**	读取指纹信息
8	void **WriteTags(i, j, t)**	修改指纹信息
9	bool **ConFindTagInBuckets(i1, i2, tag)**	并发查找接口
10	bool **ConDeleteTagFromBucket1(i, tag)**	并发删除接口
11	ReturnCode **ConInsertTagToBucket(i, tag, kickout, &oldtag)**	并发插入接口

度、每个哈希桶能容纳的指纹信息的数量、哈希桶的数量、哈希表的最大指纹容量等信息；序号 7、8 对应的 API 分别用于读取和修改指定的指纹信息；序号 9-11 对应的 API 实现在哈希表内并发的读取和修改元素。

下面对 Cuckoo 过滤器的三种元素操作的多线程并发实现进行描述。

(1)　插入操作

在标准 Cuckoo 哈希表中，在哈希表中插入新的元素时需要以某种方式读取原本存储于哈希表内的元素，以便在发生踢出原始元素时确定将该元素安置在哪个位置上。但是，对于只存储了指纹的 Cuckoo 过滤器而言，它无法根据原始元素的键计算出旧键迁移到哪个位置。这里引入不完整键 Cuckoo 哈希算法来解决无法根据指纹信息定位旧键迁移位置的问题。对任意的元素 x，使用公式 5.24计算其两个备选哈希桶的索引值:

$$\begin{aligned} h_1(x) &= hash(x) \\ h_2(x) &= h_1(x) \bigoplus hash(\text{x 的指纹信息}) \end{aligned} \tag{5.24}$$

式 5.24的异或操作有一个重要的特性：$h_1(x)$ 可以通过 $h_2(x)$ 和指纹信息用同样的公式推算出来。即就是说，替换编号为 i 的哈希桶中的旧键（不论 i 对应的是 h_1 还是 h_2），都可以通过当前桶编号 i 以及存储在该桶内的指纹直接计算出旧键的备选哈希桶的编号 j，计算方式为：$j = i \bigoplus hash$(指纹信息)。

因此，在进行插入操作时，不用检索目标哈希桶内存储的元素的完整信息，只需要存储在哈希表中的指纹信息。

另外，为了使众元素在哈希表中均匀分布，在与索引值进行异或运算之前，指纹信息已经进行过哈希运算。当指纹信息远小于哈希表的大小时，如果直接使

用索引值与未经哈希的指纹信息进行异或运算来计算接纳被踢出元素的哈希桶的索引值，那么接纳被踢出的元素哈希桶与原来存储该元素的哈希桶在位置上很近，这样会导致插入的元素过于集中，增加 Cuckoo 路径的长度。下面的例子有助于理解上述问题。使用 8 比特的指纹信息时，接纳从 i 中被踢出的元素的哈希桶的位置距离 i 的最大距离为 $2^8 = 256$。这是因为在进行异或运算时，会选取索引值的低 8 位进行运算，而高 8 位保持不变。对指纹信息进行哈希能够确保这些元素尽量分散在哈希表的不同位置，从而可以避免哈希碰撞，提高哈希表的空间利用率。 算法 5.5给出了使用不完整键 Cuckoo 哈希方法对 Cuckoo 过滤器动态插入元素的过程。使用基于 Intel RTM 的 MCS 锁对 Cuckoo 过滤器进行保护。在事务化推测执行期间，线程并没有真正持有锁，除非它有提交需求。所以，使用粗粒度锁对整个哈希表进行保护不会影响线程扩展性。执行插入操作时，首先计算出元素 x 的指纹信息 f，然后计算 x 的哈希桶索引值 i_1、i_2。插入具有相同指纹信息的元素在 Cuckoo 过滤器内是合法的。然后在索引值 i_1、i_2 对应的任意哈希桶内查找是否有空闲位置，若有，则存入；若两个桶内都没有空闲位置，则在 i_1、i_2 中随机的选取一个哈希桶，随机的踢出该桶内的一个元素，然后将新插入的元素存储到空出来的位置。被踢出的元素将移动到其备选的哈希桶内，如果备选桶内也没有空闲位置，则重复替换过程，一直到所有的元素都找到存储位置或者替换的次数达到上限值为止。如果替换次数达到上限值，可以认为哈希表已经达到“饱和”，需要考虑重建更大的哈希表。

指纹信息的长度小于 h_1 和 h_2 的长度造成的后果有两个方面：**第一**，通过公式 5.24计算出的 (h_1, h_2) 的组合的总数会远远小于使用完整的哈希值计算得到的组合的数量，这将导致哈希碰撞更严重；**第二**，允许插入两个具有相同指纹的元素 x 和 y，在一个哈希桶内可能出现多个相同的指纹信息是合法的。但是如果相同的指纹的数量超过 2b（b 为哈希桶的大小）时，存储其指纹信息的哈希桶会过载。解决哈希桶过载的途径有多种。

- 第一，不实现过滤器删除元素操作，这样每条指纹信息都只需要存储一份副本。这是最简单、直观的方法，但这显然与 Cuckoo 过滤器的设计初衷不符。
- 第二，引入适当的空间开销在哈希桶内加入计数器，插入/删除元素时计数器适当的自增/自减。
- 第三，将原始的键存储在其他位置（可以是访存速度较慢的外存上），这样可以在插入元素时查看该记录，避免插入重复的元素。但是如果哈希桶内已经存在匹配的实体，则插入的速度相对较慢。

算法 5.5: Cuckoo 过滤器插入操作

```
1  # define UPDATE_LOCK    locklib_mutex_lock(mutex, 1)
2  # define UPDATE_UNLOCK    locklib_mutex_unlock(mutex)
3  Function Insert(x)
4  f = fingerprint(x)
5  i1 = hash(x)
6  i2 = i1 ⊕ hash(f)
7  UPDATE_LOCK
8  if 桶 i1 或 i2 内有空闲实体; then
9      将 f 存入 i1 或 i2;
10     UPDATE_UNLOCK
11     return true
12 end
   /* 当前桶内没有空闲位置                                          */
13 i = rand(i1, i2)
14 for n = 0 to MaxNumKicks − 1 do
15     从 bucket[i] 中随机的选择一个实体 e;
16     将新插入元素的指纹信息与 e 的指纹信息交换;
17     i = i ⊕ hash(f)
18     if bucket[i] 有空闲实体; then
19         将 f 存储到 bucket[i];
20         UPDATE_UNLOCK
21         return true
22     end
23 end
24 UPDATE_UNLOCK
   /* 哈希表饱和                                                    */
25 return false
```

(2) 删除操作

在标准的布隆过滤器中删除元素需要对整个过滤器进行重建，引起惊人的性能开销。所以标准布隆过滤器不支持删除操作。而将布隆过滤器的比特位扩展成计数值的方法需要耗费 3-4 倍的空间开销。Cuckoo 过滤器可以直接从过滤器中移除相关元素的指纹完成删除操作。

安全的删除元素有一个前提条件：被删除的 x 必须是已经插入到了过滤器中的元素。否则的话，有可能错删过滤器内具有相同指纹信息的其它元素。这条原则不仅是对 Cuckoo 过滤器，同样对其他支持删除操作的过滤器也适用。有关 Cuckoo 过滤器的删除过程如算法 5.6所示。执行删除操作时，同样要先得到待删除元素的指纹信息，然后通过哈希函数和指纹信息得到存储该元素的哈希桶的索引值。然后在对应的哈希桶内找到该元素的指纹信息，完成删除操作。

算法 5.6: Cuckoo 过滤器的删除操作

```
1  # define UPDATE_LOCK     locklib_mutex_lock(mutex, 1)
2  # define UPDATE_UNLOCK     locklib_mutex_unlock(mutex)
3  Function Delete(x)
4  f = fingerprint(x)
5  i1 = hash(x);
6  i2 = i1 ⊕ hash(f)
7  UPDATE_LOCK
8  if bucket[i1] 或 bucket[i2] 中含有 f then
9      从当前 bucket 内删除 f 的一个副本;
10     UPDATE_UNLOCK
11     return true
12 end
13 UPDATE_UNLOCK
14 return false
```

相比当前其它支持删除操作的布隆过滤器的扩展版本，比如 *d-left* 计数过滤器，熵过滤器的实现，Cuckoo 过滤器的删除操作十分简单。删除元素时，Cuckoo 过滤器首先根据索引值在桶内进行查找；如果在任意的桶内有匹配的指纹信息，则删除该桶内的指纹信息的一个副本。

在删除某个元素后，不需要对这个实体进行清理。这样可以避免在同一个桶内存有两个具有相同指纹信息的元素时的“误删”。假设元素 x 和 y 都映射到了桶 i_1 内，并且具有相同的指纹信息 f。因为 $i_2 = i_1 \bigoplus hash(f)$，所以它们同样能够保存在桶 i_2 内。在删除 x 时，不用考虑删除的指纹信息的副本是在插入 x 还是 y 是

添加的。删除 x 后，在桶 i_1、i_2 中 y 仍然可以被查询到。

值得注意的是，在上面的例子中删除元素 x 之后，过滤器的误判行为仍然存在。过滤器中的 y 会在查询 x 时发生误判，因为两者具有相同的桶索引值和指纹信息。误判行为仍然在近似集合元素查询数据结构接受的范围之内，误判率也仍然满足 ϵ 的上界约束条件。

(3) 查询操作

算法 5.7: Cuckoo 过滤器查询操作

```
1  # define FIND_LOCK     locklib_mutex_lock(mutex, 0)
2  # define FIND_UNLOCK     locklib_mutex_unlock(mutex)
3  Function Lookup(x)
4  f = fingerprint(x)
5  i1 = hash(x);
6  i2 = i1 ⊕ hash(f)
7  FIND_LOCK
8  if bucket[i1] 或 bucket[i2] 中含有 f; then
9  |   FIND_UNLOCK
10 |   return true
11 end
12 FIND_UNLOCK
13 return false
```

Cuckoo 过滤器的查询操作相对简单。算法 5.7对这个过程做了简单描述。对于给定的元素 x，首先计算其指纹并根据公式 5.24计算出它的两个哈希桶的索引值。然后遍历这些哈希桶：如果在任何一个桶内找到了 x 的指纹，则返回 $true$。否则返回 $false$。值得注意的是，只要哈希桶不发生溢出，就能确保不存在任何误判。

5.4 性能优化

5.4.1 空间性能优化

布隆过滤器因其高效的空间利用率被用于海量数据索引。因此在设计布隆过滤器时，在寻求使用更少的比特位表示每个元素的同时保持误判率在可承受的范围之内是十分有意义的工作。

如果 Cuckoo 过滤器不需要实现元素的删除操作，它的空间效率还可以进一步提升。具体的，对哈希桶容量为 4 时的 Cuckoo 过滤器使用一种半排序方法。之

算法 5.8: Cuckoo 过滤器的并发查询过程

```
1  # define FIND_lock     locklib_mutex_lock(mutex, 0)
2  # define FIND_unlock     locklib_mutex_unlock(mutex)
3  Function ConFindTagInBuckets(const i1, const i2, const tag)
4      char *p1 = bucktets_[i1].bits_, *p2 = bucktets_[i2].bits_
5      int v1 = *(*)p1, v2 = *(*)p2
6      FIND_LOCK
7      if f == 4  &&  k == 4 then
8          bool  ret ← hasvalue4(v1, tag) || hasvalue4(v2, tag)
9          FIND_UNLOCK
10         return ret
11     end
12     else if f == 8 && k == 4 then
13         bool ret ← hasvalue8(v1, tag) || hasvalue8(v2, tag)
14         FIND_UNLOCK
15         return ret
16     else if f == 12 && k == 4 then
17         bool ret ← hasvalue12(v1, tag) || hasvalue12(v2, tag)
18         FIND_UNLOCK
19         return ret
20     else if f == 16 && k == 4 then
21         bool ret ← hasvalue16(v1, tag) || hasvalue16(v2, tag)
22         FIND_UNLOCK
23         return ret
24     else
25         for j = 0 to kTagsPerBucket − 1 do
26             if (ReadTag(i1, j) == tag) || (ReadTag(i2, j) == tag) then
27                 FIND_UNLOCK;
28                 return true
29         end
30     end
31     FIND_UNLOCK;
32     return false;
```

所以称之为半排序是因为这种排序算法只适用于哈希桶容量为 4 时的 Cuckoo 过滤器。使用半排序技术能够为每一个元素的存储节省 1 比特的空间。对哈希桶进行半排序是基于一个重要的事实：哈希桶内存储的指纹信息的顺序与 Cuckoo 过滤器元素查询的结果无关。在这个前提下，可以首先对哈希桶内的指纹信息进行排序，然后将排序后的指纹信息进行编码压缩。这个方法与 F.Bonomi 等人提出的“半排序哈希桶”技术类似[32]。

下面举例说明这种半排序压缩方法如何达到节省空间的目的。假设每个哈希桶的容量 $b = 4$ 并且每条指纹信息的长度 f 为 4 比特。则未经压缩的哈希桶的大小为 $4 * 4 = 16$ 比特。然而，如果对哈希桶内所有的 4 比特的指纹信息进行排序之后（没有存储元素的实体视为存储了变量”0”），经过排序之后总共只有 3876 种可能的组合结果。再将这 3876 种可能的桶-值组合存储到一个额外的表内，然后用指向这个表的索引替换掉原始哈希桶，这样原来的哈希桶就只需要使用 12 比特的索引结构而不是原来的 16 比特表示，平均每条指纹信息节省 1 比特的存储空间。

注意到使用这种预先排列出所有可能组合进行查找的方法需要额外的编码/解码表和间接索引。因此，为了保证查询效率，必须保证编码/解码表足够小以便适用高速缓存的容量。基于上述考虑，半排序方法只适用于哈希桶容量为 4 的 Cuckoo 过滤器中。另外，当指纹信息的长度大于 4 比特时，只对它的 4 个比特位进行编码，余下的比特位将直接单独的存储在原来的哈希桶内。

5.4.2 并发性能优化

在访问共享数据时，为了保证程序的一致性，会损耗程序的性能和扩展性。因此，有效的减少对共享数据的访问有助于提升程序的性能。

事务内存执行事务代码时，为了保证一致性和正确性，需要维护一个读取集（read set）和一个写入集（write set）。在进行提交时，事务内存必须验证事务的读取集是一个快照，并且需要对照该快照原子地更新写入集[132,133]。事务执行时的读取集和写入集统称为事务的“足迹”（footprint）[134]。对于硬件事务内存，事务的足迹的大小必须小于缓存行的容量。

在软件事务内存中，有一种称为一致性不敏感的编程模型（consistency oblivious programming, COP）[135]，即允许符合特定条件的并发代码部分在不进行一致性验证的情况下执行。COP 可以精简事务足迹的大小，从而消除一部分数据冲突和伪中止（spurious abort），有效的提升基于软件事务内存的并发数据结构的性能和扩展性。但是，使用 COP 时，必须在涉及对共享数据进行修改的位置添加检查点，用以验证算法在正确的轨道上执行，如果算法执行过程出现异常，则会以更加保守的方式从检查点处重新开始执行，这需要付出更加高昂的一致性开销。当前支持硬件事务内存的系统，如 Intel HasWell 和 IBM Power HTM，都不允

许软件以任何形式访问或者修改事务的读取集。所以，这种通过设置检查点的方式无法用于支持硬件事务内存的环境中。

受软件事务内存 COP 编程模型的启发，H.Avni 等人[134] 设计了一种可以用于硬件事务内存的 COP 模版。他们的方法可以概括为：使用 COP 对基于硬件事务内存的并发数据结构进行优化时，可以考虑对事务进行预处理，将其转化成 COP 事务。这种针对硬件事务内存的 COP 编程模型将原始事务分成两部分：一部分为只读前缀，一部分为更新后缀。前缀执行时不包含任何同步操作，并产生一个可能不一致的输出。后缀启动包含两个对象的硬件事务：验证前缀生成的是一个可接受的结果，并执行 COP 事务所需的所有更新操作。

下面结合算法伪代码（算法5.9）说明 COP 编程模型是如何突破硬件事务内存访问限制，精简事务的读取集和写入集的。假设函数 Γ(Gamma) 是某个数据结构的一个串行化操作（如哈希表的插入、删除、查询）。为了使 Γ 能够适应 COP 模型，需要先提取 Γ 的最长只读前缀到 ΓReadOnlyPrefix()。然后 ΓReadOnlyPrefix() 不使用任何同步操作计算得到 ΓOutput（第 2 行）。由于可能与其它并发操作存在冲突，所以 ΓOutput 可能是不一致的甚至是错误的。得到 ΓOutput 之后，启动事务（第 6 行），并调用 ΓVerify($Gamma$Output) 函数进行验证（第 8 行）。如果 ΓOutput 是不一致的，则调用 XABORT 中止该事务（第 9 行）。如果是一致的，则继续执行 ΓComplete(ΓOutput)（第 10 行）。ΓComplete(ΓOutput) 会执行所有需要执行的更新操作。

在尝试事务提交时（第 16 行），首先需要检查全局锁是否空闲（第 13 行）。如果锁被占有，当前事务被中止，并返回一个特殊编码（第 14 行）。在事务启动时可以对锁进行采样，并在锁被其它线程占用的情况下中止事务。但是，出现这种情况会被判定为进行了一次重试，会导致错误的回退。所以，通过第 19 行的代码，可以避免重新计算新的 Read Only Prefix（ROP），并且能够避免被判定为一次重试。从第 22 行开始处理事务执行失败需要进行重试的情况。如果事务中止的原因是容量溢出，则不再重试该事务，因为它仍然面临执行失败的结果。如果事务是被显示的中止的，比如 ΓVerify 函数调用了 XABORT 指令，则必须重新运行 ΓReadOnlyPrefix() 函数重新计算正确的 ΓOutput；否则，事务中止是由冲突引起的，说明 ΓOutput 的结果是正确的，事务很有可能在经过一次或者几次重试之后成功提交。如果重试的次数达到上限值，则获取锁，并运行 Γ 函数的串行化版本。

有了 COP 模版之后，实现 Cuckoo 过滤器的 COP 版本，只需要将 Cuckoo 过滤器的 Insert、Delete 和 Lookup 方法（统一用 Φ 方法描述）转化成对应的 ΦReadOnlyPrefix()，ΦVerify() 和 ΦComplete()。算法 5.10所示为 Cuckoo 过滤器的 Insert 方法的 COP 版本的实现。

使用 COP 编程模型对基于硬件事务内存的并发数据结构的性能提升有一定的帮助，但是这种方法增加了实现的复杂度。

算法 5.9: COP 模版

```
1  retries ← 0;
2  ΓOutput ← ΓReadOnlyPrefix();
3  while locked do
4  |  await;
5  end
6  status ← XBEGIN;
7  if status == XBEGIN_STARTED then
8  |  if ΓVerify(ΓOutput) fail then
9  |  |  XABORT;  //如果失败，调用 XABORT 中止事务
10 |  else
11 |  |  ΓComplete(ΓOutput);  //执行更新操作
12 |  end
13 |  if locked then
14 |  |  XABORT(WAS_LOCKED);
15 |  end
16 |  XEND;
17 else
18 |  if is_explicit(status, WAS_LOCKED) then
19 |  |  goto line 6;  //重用 Output
20 |  end
21 |  retries++;
22 |  if status ≠ XABORT_CAPACITY then
23 |  |  if retries < MAX_RETRIES then
24 |  |  |  if is_explicit(status, BAD_ROP) then
25 |  |  |  |  goto line 2;  //重新计算 ROP
26 |  |  |  else
27 |  |  |  |  goto line 6;  //重用 ΓOutput
28 |  |  |  end
29 |  |  end
30 |  end
31 |  lock();
32 |  Γ();
33 |  unlock();
34 end
```

算法 5.10: 基于 COP 编程模型的 Cuckoo 过滤器的 Insert 方法

```
1  Function CF_Insert(CF *s, K, V)
2      retries ← 0;
3      retry:
4      *index = CFInsertReadOnlyPrefix(t,K);
5      retry_verify:
6      while locked do
7          pause();
8      end
9      status ← XBEGIN();
10     if status == XBEGIN_STARTED then
11         CFInsertVerify(index, K);
12         CFInsertComplete(t, K, V, index);
13         if locked then
14             XABORT(WAS_LOCKED);
15         end
16     else
17         if is_explicit(status, WAS_LOCKED) then
18             goto retry_verify;
19         end
20         if retries++ ⩽ MAX_RETRIES then
21             if is_explicit(status, BAD_ROP) then
22                 goto retry;
23             end
24             if can_retry(status) then
25                 goto retry_verify;
26             end
27         end
28     end
29     //回退路径 lock();
30     index = CFInsertReadOnlyPrefix(t, K);
31     CFInsertComplete(t, K, V, index);
32     unlock();
```

5.5 性能评估

本文在 Cuckoo 过滤器的基础上[131]，实现了基于 Intel RTM 的并发 Cuckoo 过滤器，并结合一些软件优化方法进行进一步的优化。下文为了便于进行性能比较和描述，将并发 Cuckoo 过滤器记作“CCF”。

5.5.1 实验配置和性能指标

实验配置：所有插入过滤器的元素都是预先采用随机数生成器生成的 64 位的整型数。由于生成相同数值的概率极低，所以没有对具有相同值的元素进行排查。每一次对过滤器的操作请求，过滤器首先使用 CityHash[36] 生成对应元素的 64 位哈希值。每一组数据都是运行 10 次的结果取平均值得到。所有实验都是在具有两个 Intel Xeon Broadwell EP/EN/EX 处理器的机器上运行。该机器总共有 32 个物理核（64 个逻辑核），内存总容量为 64 GB。CPU 的时钟频率为 2.1 GHz，三级缓存的容量分别为 64 KB，256 KB 以及 40 MB。该工作站安装的是 Ubuntu 16.04 LTS 操作系统。

性能指标：为了充分评估 CCF 的性能、空间效率、误判率以及与 CCF 并发性相关的线程扩展性、吞吐量，本文确定了以下几个性能指标：

- 空间效率：用于表示一个元素所需的平均比特数，计算方式是过滤器插入的元素的总的比特位数除以过滤器存储的元素数量，用 bits/item 表示。
- 误判率：在过滤器的负载因子不再变化之后，通过在过滤器内查询不存在的元素，统计返回值为“True”的查询的次数 $false\_count$，然后用 $false\_count$ 除以执行的查询的次数，得到误判率；
- 构造速率：往空的过滤器中不断插入元素直到过滤器“饱和”（判断过滤器“饱和”的依据是当执行某次插入操作返回失败）时为止，总的插入元素的个数除以从空到“饱和”花费的时间，单位为 million items/sec。
- 吞吐量：平均每秒所能执行的过滤器元素操作的数量；吞吐量跟工作集、过滤器内元素的密度以及线程数量相关。
- 线程扩展性：吞吐量随着线程数量增长的能力，好的扩展性体现在随着参与运算的线程的增加，对应的吞吐量也增加。

5.5.2 空间效率和误判率

首先对比 CF 和 CCF 的空间效率和误判率。过滤器的大小设置成 192 MB（为 $m = 2^{25}$ 个能存储 4 个大小为 12 比特的指纹信息的哈希桶的哈希表的大小）。对于 CCF，执行测试的线程数设置成机器支持的最大线程数（64），每个线程承担的插入任务是均等的，当线程的某次插入操作失败时，该线程的插入任务终止。当所有的线程完成插入任务或者被终止时，统计插入元素的个数，最终插入元素

的个数为各个线程执行的插入次数的总和。同样 CF 停止插入元素的条件也是某次插入返回失败。误判率计算方法是，在过滤器内查询一个确定不存在的元素，如果某次查询操作返回该元素存在，则记录一次误判。本次实验中，抽样查询次数为一百万次。最终误判率通过记录的误判数除以总的查询的次数。表 5.3列出了 CF 和 CCF 的相关测试结果。

在指纹信息长度、过滤器容量一定的前提下，CCF 的误判率是串行 Cuckoo 过滤器的一半；此外，二者最明显的差距体现在构造速率上，CCF 的构造速度大约是串行 Cuckoo 过滤器的十余倍。但是，CCF 的空间利用率较低，约为串行过滤器的 50%。

表 5.3　空间效率、误判率和构造速率

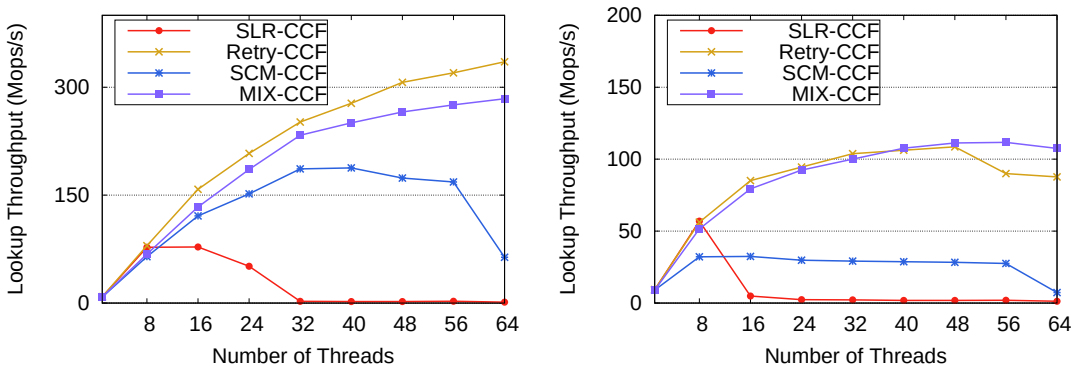
过滤器类型	插入元素个数	单个元素占用的比特位数	ϵ (%)	构造速度
CF	128.07 (M)	12.58 (bits)	0.2	5.32 (M items/sec)
CCF	66.9 (M)	24.07 (bits)	0.1	658.06 (M items/sec)

5.5.3　扩展性

这一部分将对 CCF 的线程扩展性进行评估。

首先综合评估了使用不同软件优化方案的 RTM 版本。这些软件方法分别是：基于 RTM Retry 的版本，用“Retry-CCF“标识，其中 $retry$ 的最大次数设置为 10（对于 $retries$ 次数的设定将有专门的内容进行说明）；基于 SLR 的版本，用”SLR-CCF“标识；基于 SCM 的版本，用”SCM-CCF 标识“；以及结合了 SLR 和 SCM 的版本，用”MIX-CCF“标识。图 5.7展示了这四个不同版本的线程扩展性。图 5.7（a）对应处理只读数据集的运行结果，图 5.7（b）对应处理更新占 10% 的数据集的运行结果。过滤器的参数说明：过滤器的哈希桶数量为 2^{24} 个，数据结构使用（2，4）路组相连的 Cuckoo 哈希表，即使用 2 个哈希函数，每个哈希桶能容纳 4 条指纹信息，因此过滤器的最大指纹信息存储数量为 2^{26}，指纹信息的长度为 12 比特，过滤器的负载因子固定为 50%（之所以不设置成更高的参数，是基于数据集中包含一定比例的插入操作），即初始化的过滤器内存储了 2^{25} 个指纹信息。此时过滤器的大小为 96 MB，远大于测试机器的最后一级缓存的容量（40 MB）。

图 5.7（a）的结果表明 Retry-CCF 的扩展性最好，其次是 MIX-CCF，单独使用 SLR 或者 SCM 的扩展性都不理想；图 5.7（b）中 Retry-CCF 和 MIX-CCF 的吞吐量和扩展性不相伯仲。这与第 4.5.3节中基于 HTM 的并发哈希表存在差异。



(a) $u = 0$ (b) $u = 10$

图 5.7　使用不同软件优化方法的 CCF 线程扩展性

5.5.4　更新比重对性能的影响

Cuckoo 过滤器的核心技术是不完整键 Cuckoo 哈希，是建立在 Cuckoo 哈希表上的应用。在对哈希表的综合评估中（第3.4.3）发现数据集中更新比重所占的比例对性能的影响较大。因此，这里通过几组数据对运行不同数据集的 CCF 的性能进行比较。上述对使用不同优化方法的版本进行比较发现基于 Retry-CCF 机制的性能略胜一筹。因此，进行这部分测试使用 Retry-CCF。过滤器各项参数的设置与收集图 5.7数据的设置相同。不同的是测试所用的数据集包括三个，分别为包含了 40% 和 10% 更新操作以及只进行查询操作的数据集。运行的结果如图 5.8所示。

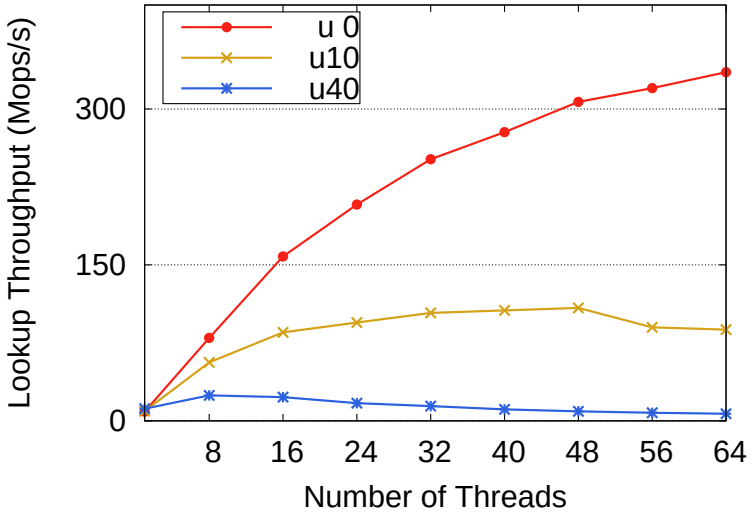


图 5.8　并发 Cuckoo 过滤器的线程扩展性

由图 5.8观察到，并发 Cuckoo 过滤器的只读性能的扩展性最好，吞吐量随着线程数量的增加而上升，它的最高吞吐量出现在线程数为 64 处（实验机器支持的最大线程数），是运行单个线程的 Cuckoo 过滤器的 38 倍。而如果测试集中包含一定比例的更新操作时，情况会发生变化。这种变化直观的体现在吞吐量上，比

如数据集包含 10% 的更新操作时，最高吞吐量为使用单个线程的 11 倍，而数据集包含 40% 更新操作时，最高吞吐量仅为单个线程的 2.5 倍。另外，线程的扩展性也受到数据集中更新操作数量的影响，具体的趋势是数据集的更新操作的比重越高，扩展性就越差。如在只读的数据集中，一直到线程数达到最大，CCF 都展现出良好的扩展性，但在 $u = 10\%$ 时，吞吐量在线程数大于 48 后出现下降；同样的，$u = 40\%$ 时，吞吐量在线程数大于 16 就开始下降。

进一步结合本文第 3.4.2中结论，设计或者应用并发哈希表不能一味追求高并发度，要综合考虑工作集的特点。

5.5.5 *Retries* 设置对性能的影响

在基于 RTM Retry 的版本中，*retries* 值的设置将直接对性能造成影响。如果值太小，等待提交的线程在当前持有事务锁的线程释放锁之前就已经转为申请标准锁或者被存放到等待线程队列中，增加了串行化执行的比例，造成性能下降；如果值过大，某些由系统问题（如页错误、不兼容的指令等）引起的事务中止耗费在 Retry 过程的时间过长，导致运行效率低。因此，选用恰当的 *retries* 值对性能和效率都至关重要。

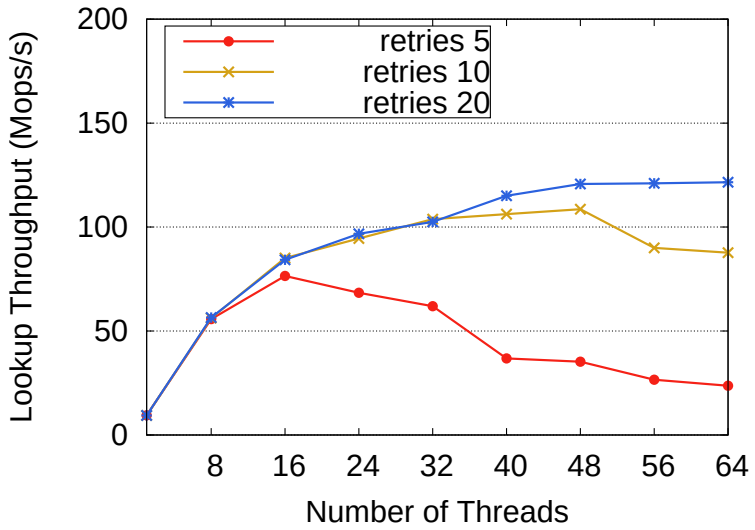


图 5.9 不同 *retries* 值之间的性能差异

接下来通过一组实验数据说明基于 RTM Retry 的版本中参数 *retries* 的设置对并发 Cuckoo 过滤器性能的影响。本组实验中测试了 $retries = 5, 10, 20$ 三个值。测试所用的数据集包含 10% 的更新操作。测试结果如图 5.9所示。从图中的曲线趋势可以观察到，当 $retries = 5$ 时，CCF 的吞吐量在线程数大于 16 之后开始下降，并且并发度越高，吞吐量越低。而 *retries* 设置成更大的值时，线程扩展性有明显的改善，$retries = 20$ 的扩展性略微比 $retries = 10$ 的好，但是它所花费的执行时间也更长。因此，综合考虑扩展性和效率两方面的因素，在本文的相关测试中将

$retries$ 设置为 10。

5.6 本章小结

本章首先介绍了标准布隆过滤器的原理，对布隆过滤器的误判率，最优哈希函数个数，位数组长度和空间效率等进行了理论推导。最终得到位数组长度 m，元素个数 n 和误判率 ϵ 之间需要满足：$\frac{m}{n} \geqslant 1.44log_2(1/\epsilon)$ 才能使性能达到最优。在哈希函数个数一定的情况下，误判率每缩小到原来的十分之一，每个元素平均需要增加 4.8 比特的存储空间。随后，介绍了基于不完整键 Cuckoo 哈希方法的 Cuckoo 过滤器，对 Cuckoo 过滤器的设计原理，指纹信息的长度，空间效率，哈希桶的容量等方面都进行了深入的理论分析。确定了 Cuckoo 过滤器的空间效率等于指纹信息的长度 f 与过滤器的负载因子 α 的比值。而 f 与 α 都与哈希桶容量 b 有关，故以此为出发点，进一步的对哈希桶的容量对空间效率的影响进行了分析。接下来，使用基于 Intel RTM 实现的读写锁实现了支持多线程并发的 Cuckoo 过滤器。对其加锁和解锁算法进行了描述，对并发操作的方法进行详细介绍。最后对并发 Cuckoo 过滤器的性能进行了全面的评估。评估涵盖了对其的线程扩展性、构造速率、误判率、空间效率的评估，评估了与硬件事务内存特性相关的参数的影响以及使用不同软件优化方法对性能的影响。

实验评估的结果表明，我们的并发 Cuckoo 过滤器在多核处理器上的效率和性能都与预期的结果相符，在处理只读数据集时最高吞吐量是使用单个线程运行的 38 倍，数据集中包含少量更新操作时，吞吐量略低，但也达到单个线程处理的吞吐量的 11 倍。此外还发现在支持硬件事务内存的环境里使用简单的优化方案带来的性能上的改善效果更明显。

总结与展望

1. 本文工作总结

计算机系统的数据处理依赖CPU，应用程序的设计从一定意义上说就是程序设计者想方设法“榨取”CPU的处理能力的过程。数据结构就是“榨取”CPU处理能力的手段之一，数据结构的效率与性能对应用程序至关重要。哈希表是一种经典的数据结构，因其常数时间的元素处理特性得到广泛应用。多核CPU的问世对设计新的、具有高可扩展性的数据结构提出了挑战，这些挑战主要包括如何处理高并发度下的数据竞争，如何充分利用多核处理器的计算资源等。近几年，人们开始关注并发哈希表的设计、优化与应用。本文围绕多核系统上的并发哈希表的设计与应用主要做了以下工作：

1. 针对当前基于多核系统的并发哈希表缺乏统一、公平测试框架的问题，设计了基于C/C++语言的用于并发哈希表测试、评估的框架CHTBench。CHTBench能够为进行测试的并发哈希表提供一致的运行环境，包括内存管理、线程管理、编译、数据集等。可以用于评估并发哈希表的线程扩展性、读写性能、同步机制的有效性以及内存效率等方面的内容。
2. 选取五种近几年最具有代表性的并发哈希表，使用CHTBench框架在4个不同的多核系统上进行了全面评估：线程扩展性、吞吐量、运行时延迟、内存分层结构的影响、底层同步原语和内存消耗等进行比较分析。在必要的情况下，还对存在关联的指标进行了深入分析。总结了8条应用并发哈希表应当遵循的原则以及需要规避的陷阱。实验平台涵盖NUMA（Non-uniform Memory Architecture）和非NUMA架构系统。本文成功的将并发哈希算法移植到Intel MIC平台上进行测试，这是首次将并发哈希表的研究扩展到Intel MIC架构上，在MIC架构上以并发哈希表为例，探讨了该架构下的若干同步问题。
3. 现有的并发哈希表基本都是使用经典的锁算法或者非阻塞算法实现的多线程并发，这些经典方法效率高，对性能有保障，但是实现复杂、变种繁多，且难以保证正确性。利用硬件事务内存在设计并发数据结构上的天然优势，设计了基于硬件事务内存的并发哈希表。并针对Intel事务同步扩展的Lemming效应，提出了两种减轻Lemming效应的软件优化方法，实验评估表明，这两种软件优化方法比Intel官方推荐的Retry机制效果更好。通过在CHTBench框架上的测试表明基于硬件事务内存的并发哈希表处理大规模数据集时的性能比使用传统的细粒度锁方法获得的性能提升了20%。
4. 经典的布隆过滤器以一定的误判率换取极高的空间效率，但是它不支持删

除操作，后续研究在布隆过滤器的基础上实现了删除元素功能，但是以更高的空间开销为代价；此外，目前没有一款支持多线程并发的布隆过滤器。Cuckoo 过滤器使用不完整键 Cuckoo 哈希算法实现了删除功能，且具有极低的空间开销。在 CUckoo 过滤器的基础上，使用 Intel RTM 实现了支持多线程并发的 Cuckoo 过滤器 (CCF)。CCF 的最高吞吐量是处理同等数据规模的单线程 Cuckoo 过滤器的 38 倍。

2. 下一步工作展望

本文针对多核系统的并发哈希表的研究虽然取得了一些进展，然而在计算机软硬件技术高速发展的今天这些研究成果堪称冰山一角。在本文的研究中还存在一些值得深入探究的问题:

1. 随着芯片制作工艺的提升，使得单芯片上能够集成的核心数不断增加，这种单芯片上的核心数量的变化又促进多处理器架构的调整。在 Intel MIC 架构上对并发哈希表进行移植和评估的过程中发现在传统多核处理器架构上性能和扩展性都不错的并发哈希表在 MIC 架构上表现不佳。这种吞吐量随处理器核心数小幅度变化的现象说明传统的用于实现并发哈希表的同步机制不适用于 MIC 架构。因此，研究区别于传统多核处理器架构的多线程同步机制、内存管理机制，设计基于 Intel MIC 架构的具有高可扩展性的并发哈希表是下一步研究中的重点问题。
2. 支持硬件事务内存的多核处理器的问世为设计并发哈希表提供了新的同步机制。但是事务执行并不能保证每次执行都能成功，所以为了避免线程悬停，必须设计事务代码的回退路径，这个回退路径的执行方式可以多样，但是归根结底无法避开传统的同步机制，如锁或者非阻塞方法。这会丧失事务内存灵活易用的特性。因此，研究独立于传统同步机制的硬件事务内存实现方法具有重大意义。
3. 虽然本文实现了首款支持多线程并发的 Cuckoo 过滤器，从实验评估的结果看，Cuckoo 过滤器处理更新操作的效率不高，空间利用率与其串行版本相比也较差，存在进一步优化的空间，为了与当前海量数据处理低延迟的需求相适应，下一步将考虑优化 CCF 的元素插入和删除性能，提升它的空间效率。
4. 最后，没有一种万能的数据结构和同步机制能够用于处理所有的数据集类型，实现程序根据数据集的特点自适应的调用锁方法也是值得关注和深入研究的问题。

综上所述，本文针对并发哈希表在多核系统上的设计、优化和应用的研究都取得了一定的研究成果，对于文中未尽事宜将作为本人在下一阶段的研究内容。

参考文献

[1] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach. Elsevier, 2011

[2] Geer D. Chip makers turn to multicore processors. Computer, 2005, 38(5):11–13

[3] Marr D, Binns F, Hill D, et al. Hyper-threading technology in the netburst® microarchitecture. 14th Hot Chips, 2002.

[4] Samson E C, Machiroutu S V, Chang J Y, et al. Interface Material Selection and a Thermal Management Technique in Second-Generation Platforms Built on Intel® Centrino™ Mobile Technology. Intel Technology Journal, 2005, 9(1)

[5] Lempel O. 2nd Generation Intel® Core Processor Family: Intel® Core i7, i5 and i3. In: Proc of Hot Chips 23 Symposium (HCS), 2011 IEEE. IEEE, 2011, 1–48

[6] Chang J, Huang M, Shoemaker J, et al. The 65-nm 16-MB shared on-die L3 cache for the dual-core Intel Xeon processor 7100 series. IEEE Journal of Solid-State Circuits, 2007, 42(4):846–852

[7] Kongetira P, Aingaran K, Olukotun K. Niagara: A 32-way multithreaded sparc processor. IEEE micro, 2005, 25(2):21–29

[8] Shi L, Chen H, Sun J, et al. vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Transactions on Computers, 2012, 61(6):804–816

[9] Bolla R, Bruschi R. An effective forwarding architecture for SMP Linux routers. In: Proc of Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International. IEEE, 2008, 210–216

[10] Giesen F. Cache coherency primer. Website. https://fgiesen.wordpress.com/2014/07/07/cache-coherency/

[11] Liu Y, Zhang K, Spear M. Dynamic-sized nonblocking hash tables. In: Proc of Proceedings of the 2014 ACM symposium on Principles of distributed computing. ACM, 2014, 242–251

[12] David T, Guerraoui R, Trigonakis V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. SIGARCH Comput. Archit. News, 2015, 43(1):631–644

[13] Shalev O, Shavit N. Split-ordered lists: Lock-free extensible hash tables. Journal of the ACM (JACM), 2006, 53(3):379–405

[14] Li X, Andersen D G, Kaminsky M, et al. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In: Proc of Proceedings of the Ninth European Conference on Computer Systems. New York, NY, USA: ACM, 2014, 27:1–27:14

[15] Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing. In: Proc of Distributed Computing. Springer, 2008: 350–364

[16] Metreveli Z, Zeldovich N, Kaashoek M F. Cphash: A cache-partitioned hash table. In: Proc of ACM SIGPLAN Notices, volume 47. ACM, 2012, 319–320

[17] Baumann A, Barham P, Dagand P E, et al. The multikernel: a new OS architecture for scalable multicore systems. In: Proc of Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009, 29–44

[18] David T, Guerraoui R, Trigonakis V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In: Proc of Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. New York, NY, USA: ACM, 2013, 33–48

[19] Desnoyers M, McKenney P E, Stern A S, et al. User-level implementations of read-copy update. Parallel and Distributed Systems, IEEE Transactions on, 2012, 23(2):375–382

[20] Herlihy M P, Wing J M. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 1990, 12(3):463–492

[21] Intel. Intel® transactional synchronization extensions. Website. https://software.intel.com/en-us/node/524022

[22] McKenney P E. Kernel korner: using RCU in the Linux 2.5 kernel. Linux Journal, 2003, 2003(114):11

[23] McKenney P E, Sarma D, Soni M. Scaling dcache with RCU. Linux Journal, 2004, 2004(117):3

[24] McKenney P E, Boyd-Wickizer S, Walpole J. RCU usage in the linux kernel: One decade later. Technical report, 2013.

[25] Yu M, Fabrikant A, Rexford J. BUFFALO: Bloom filter forwarding architecture for large organizations. In: Proc of Proceedings of the 5th international conference on Emerging networking experiments and technologies. ACM, 2009, 313–324

[26] Dharmapurikar S, Krishnamurthy P, Taylor D E. Longest prefix matching using bloom filters. In: Proc of Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications. ACM, 2003, 201–212

[27] Bonomi F, Mitzenmacher M, Panigrah R, et al. Beyond bloom filters: from approximate membership checks to approximate state machines. In: Proc of ACM SIGCOMM Computer Communication Review, volume 36. ACM, 2006, 315–326

[28] Song H, Dharmapurikar S, Turner J, et al. Fast hash table lookup using extended bloom filter: an aid to network processing. ACM SIGCOMM Computer Communication Review, 2005, 35(4):181–192

[29] Jokela P, Zahemszky A, Esteve Rothenberg C, et al. LIPSIN: line speed publish/subscribe inter-networking. ACM SIGCOMM Computer Communication Review, 2009, 39(4):195–206

[30] Broder A, Mitzenmacher M. Network applications of bloom filters: A survey. Internet mathematics, 2004, 1(4):485–509

[31] Fan L, Cao P, Almeida J, et al. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking (TON), 2000, 8(3):281–293

[32] Bonomi F, Mitzenmacher M, Panigrahy R, et al. An improved construction for counting bloom filters.

In: Proc of ESA, volume 6. Springer, 2006, 684–695

[33] Bender M A, Farach-Colton M, Johnson R, et al. Don't thrash: how to cache your hash on flash. Proceedings of the VLDB Endowment, 2012, 5(11):1627–1637

[34] Haberman J. State of the hash functions. Website. http://blog.reverberate.org/2012/01/state-of-hash-functions-2012.html

[35] Appleby A. MurmurHash. Website. https://sites.google.com/site/murmurhash/

[36] Geoff Pike J A. Cityhash. Website. https://opensource.googleblog.com/2011/04/introducing-cityhash.html

[37] Jenkins B. Hash functions. Dr Dobbs Journal, 1997, 22(9):107–+

[38] Jenkins B. Function for producing 32bit hashes for hash table lookup, 2006

[39] Jenkins B. Spookyhash: a 128-bit noncryptographic hash, 2012

[40] Knuth D E. The art of computer programming: sorting and searching, volume 3. Pearson Education, 1998

[41] Heileman G L, Luo W. How Caching Affects Hashing.. In: Proc of ALENEX/ANALCO. 2005, 141–154

[42] Pagh R, Rodler F F. Cuckoo hashing. Journal of Algorithms, 2004, 51(2):122–144

[43] Erlingsson U, Manasse M, McSherry F. A cool and practical alternative to traditional hash tables. In: Proc of Proc. 7th Workshop on Distributed Data and Structures (WDAS'06). 2006

[44] Ross K A. Efficient hash probes on modern processors. In: Proc of Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on. IEEE, 2007, 1297–1301

[45] Black J R, Martel C U, Qi H. Graph and Hashing Algorithms for Modern Architectures: Design and Performance.. In: Proc of Algorithm Engineering. 1998, 37–48

[46] Agarwal A, Cherian M. Adaptive backoff synchronization techniques, volume 17. ACM, 1989

[47] Anderson T E. Performance implications of spin-waiting alternatives for shared-memory multiprocessors. In: Proc of Proceedings of the 1989 International Conference on Parallel Processing. Publ by IEEE, 1989

[48] Graunke G, Thakkar S. Synchronization algorithms for shared-memory multiprocessors. Computer, 1990, 23(6):60–69

[49] Mellor-Crummey J M, Scott M L. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS), 1991, 9(1):21–65

[50] Craig T. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993.(ftp tr/1993/02/UW-CSE-93-02-02. PS. Z from cs. washington. edu), 1993

[51] Magnusson P, Landin A, Hagersten E. Queue locks on cache coherent multiprocessors. In: Proc of Parallel Processing Symposium, 1994. Proceedings., Eighth International. IEEE, 1994, 165–171

[52] Scott M L. Non-blocking timeout in scalable queue-based spin locks. In: Proc of Proceedings of the

twenty-first annual symposium on Principles of distributed computing. ACM, 2002, 31–40

[53] Scott M L, Scherer W N. Scalable queue-based spin locks with timeout. In: Proc of ACM SIGPLAN Notices, volume 36. ACM, 2001, 44–52

[54] Michael M M, Scott M L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. journal of parallel and distributed computing, 1998, 51(1):1–26

[55] Mellor-Crummey J M, Scott M L. Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proc of ACM SIGPLAN Notices, volume 26. ACM, 1991, 106–113

[56] Krieger O, Stumm M, Unrau R, et al. A fair fast scalable rea, der-writer lock. In: Proc of Parallel Processing, 1993. ICPP 1993. International Conference on, volume 2. IEEE, 1993, 201–204

[57] Brooks E D. The butterfly barrier. International Journal of Parallel Programming, 1986, 15(4):295–307

[58] Hensgen D, Finkel R, Manber U. Two algorithms for barrier synchronization. International Journal of Parallel Programming, 1988, 17(1):1–17

[59] Mellor-Crummey J, Scott M. Fast, Contention-Free Combining Tree Barriers. 1992.

[60] Tseng Y L, Huang K H, Lai B C C. Scalable mutli-layer barrier synchronization on NoC. In: Proc of VLSI Design, Automation and Test (VLSI-DAT), 2016 International Symposium on. IEEE, 2016, 1–4

[61] Dijkstra E W, Scholten C S. Termination detection for diffusing computations. Information Processing Letters, 1980, 11(1):1–4

[62] Solihin Y. Fundamentals of Parallel Multicore Architecture. CRC Press, 2015

[63] Lamport L. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 1974, 17(8):453–455

[64] Herlihy M. Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 1991, 13(1):124–149

[65] Herlihy M, Luchangco V, Moir M. Obstruction-free synchronization: Double-ended queues as an example. In: Proc of Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on. IEEE, 2003, 522–529

[66] Herlihy M, Luchangco V, Moir M, et al. Software transactional memory for dynamic-sized data structures. In: Proc of Proceedings of the twenty-second annual symposium on Principles of distributed computing. ACM, 2003, 92–101

[67] Herlihy M P. Impossibility and universality results for wait-free synchronization. In: Proc of Proceedings of the seventh annual ACM Symposium on Principles of distributed computing. ACM, 1988, 276–290

[68] Fich F, Hendler D, Shavit N. On the inherent weakness of conditional synchronization primitives. In: Proc of Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing. ACM, 2004, 80–87

[69] Kogan A, Petrank E. Wait-free queues with multiple enqueuers and dequeuers. In: Proc of ACM

SIGPLAN Notices, volume 46. ACM, 2011, 223–234

[70] Michael M M, Scott M L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc of Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. ACM, 1996, 267–275

[71] Kogan A, Petrank E. A methodology for creating fast wait-free data structures. In: Proc of ACM SIGPLAN Notices, volume 47. ACM, 2012, 141–150

[72] Timnat S, Petrank E. A practical wait-free simulation for lock-free data structures. In: Proc of ACM SIGPLAN Notices, volume 49. ACM, 2014, 357–368

[73] Majo Z, Gross T R. A library for portable and composable data locality optimizations for numa systems. ACM Transactions on Parallel Computing (TOPC), 2017, 3(4):20

[74] Brecht T. On the importance of parallel application placement in NUMA multiprocessors. In: Proc of Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV). 1993, 1–18

[75] Lachaize R, Lepers B, Quéma V. MemProf: a memory profiler for NUMA multicore systems. In: Proc of ATC-USENIX Annual Technical Conference. 2012

[76] Dashti M, Fedorova A, Funston J, et al. Traffic management: a holistic approach to memory placement on NUMA systems. In: Proc of ACM SIGPLAN Notices, volume 48. ACM, 2013, 381–394

[77] Tam D, Azimi R, Stumm M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc of ACM SIGOPS Operating Systems Review, volume 41. ACM, 2007, 47–58

[78] Tang L, Mars J, Zhang X, et al. Optimizing Google's warehouse scale computers: The NUMA experience. In: Proc of High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on. IEEE, 2013, 188–197

[79] Bull J M, Johnson C. Data distribution, migration and replication on a cc-NUMA architecture. In: Proc of Proceedings of the fourth European workshop on OpenMP. 2002

[80] Blagodurov S, Zhuravlev S, Fedorova A, et al. A case for NUMA-aware contention management on multicore systems. In: Proc of Proceedings of the 19th international conference on Parallel architectures and compilation techniques. ACM, 2010, 557–558

[81] Lepers B, Quéma V, Fedorova A. Thread and Memory Placement on NUMA Systems: Asymmetry Matters.. In: Proc of USENIX Annual Technical Conference. 2015, 277–289

[82] Moir M, Shavit N. Concurrent Data Structures., 2004

[83] Kung H T, Robinson J T. On optimistic methods for concurrency control. ACM Transactions on Database Systems (TODS), 1981, 6(2):213–226

[84] Herlihy M, Moss J E B. Transactional memory: Architectural support for lock-free data structures, volume 21. ACM, 1993

[85] Rajwar R, Goodman J R. Speculative lock elision: Enabling highly concurrent multithreaded execution. In: Proc of Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture. IEEE Computer Society, 2001, 294–305

[86] Rajwar R, Goodman J R. Transactional lock-free execution of lock-based programs. In: Proc of ACM SIGOPS Operating Systems Review, volume 36. ACM, 2002, 5–17

[87] Spear M F. Lightweight, robust adaptivity for software transactional memory. In: Proc of Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures. ACM, 2010, 273–283

[88] Saha B, Adl-Tabatabai A R, Hudson R L, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proc of Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM, 2006, 187–197

[89] Shavit N, Touitou D. Software transactional memory. Distributed Computing, 1997, 10(2):99–116

[90] 林菲. 软件事务内存的动态竞争管理策略. 计算机工程与设计, 2010, (7):1510–1512

[91] 王睿伯. 面向 NUMA 结构的软件事务内存关键技术研究: [Dissertation]. 国防科学技术大学, 2007

[92] Yen L, Bobba J, Marty M R, et al. LogTM-SE: Decoupling hardware transactional memory from caches. In: Proc of High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on. IEEE, 2007, 261–272

[93] Moore K E, Bobba J, Moravan M J, et al. LogTM: log-based transactional memory.. In: Proc of HPCA, volume 6. 2006, 254–265

[94] Dalessandro L, Carouge F, White S, et al. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. ACM SIGARCH Computer Architecture News, 2011, 39(1):39–52

[95] 王肇国. 基于硬件事务内存的内存计算系统可扩展性研究: [Dissertation]. 复旦大学, 2014

[96] Cain H W, Michael M M, Frey B, et al. Robust architectural support for transactional memory in the power architecture. In: Proc of International Symposium on Computer Architecture. 2013, 225–236

[97] Wang A, Gaudet M, Wu P, et al. Evaluation of Blue Gene/Q hardware support for transactional memories. 2012. 127–136

[98] Intel R. Intel R 64 and IA-32 Architectures. Software Developer's Manual. 2015.

[99] Afek Y, Levy A, Morrison A. Software-improved hardware lock elision. 2014, 212–221

[100] Wang Z, Qian H, Li J, et al. Using restricted transactional memory to build a scalable in-memory database. In: Proc of Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014, 26

[101] Wei X, Shi J, Chen Y, et al. Fast in-memory transaction processing using RDMA and HTM. In: Proc of Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015, 87–104

[102] Chen Y, Wei X, Shi J, et al. Fast and general distributed transactions using RDMA and HTM. In: Proc of Proceedings of the Eleventh European Conference on Computer Systems. ACM, 2016, 26

[103] Wang X, Zhang W, Wang Z, et al. Eunomia: Scaling Concurrent Search Trees under Contention Using HTM. In: Proc of Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2017, 385–399

[104] Wang Z, Qian H, Chen H, et al. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In: Proc of Proceedings of the 4th Asia-Pacific Workshop on Systems. ACM, 2013, 3

[105] Bloom B H. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 1970, 13(7):422–426

[106] 谢鲲, 文吉刚, 张大方, et al. 布鲁姆过滤器查询算法. 软件学报, 2009, 20(1):96–108

[107] 吴军. 数学之美, volume 5. 人民邮电出版社, 2012

[108] Fan L, Cao P, Almeida J, et al. Summary cache: A scalable wide-area web cache sharing protocol. In: Proc of ACM SIGCOMM Computer Communication Review, volume 28. ACM, 1998, 254–265

[109] Putze F, Sanders P, Singler J. Cache-, hash-and space-efficient bloom filters. Experimental Algorithms, 2007. 108–121

[110] Mitzenmacher M D, Vocking B. The asymptotics of selecting the shortest of two, improved. 1999.

[111] 肖明忠, 代亚非, 李晓明. 拆分型 Bloom Filter. 电子学报, 2004, 32(2):241–245

[112] 谢鲲, 闵应骅, 张大方, et al. 分档布鲁姆过滤器的查询算法. 计算机学报, 2007, 30(4):597–607

[113] 谢鲲, 秦拯, 文吉刚, et al. 联合多维布鲁姆过滤器查询算法. 通信学报, 2008, 1

[114] Ware M, Rajamani K, Floyd M, et al. Architecting for power management: The IBM® POWER7™ approach. In: Proc of High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on. IEEE, 2010, 1–11

[115] Fan B, Andersen D G, Kaminsky M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.. In: Proc of NSDI, volume 13. 2013, 385–398

[116] Intel. Threading Building Blocks. Website. https://www.threadingbuildingblocks.org

[117] Oracle. Java 7 EE. Website. https://www.oracle.com

[118] R.Pagh, E.F.Rodler. Cuckoo Hashing. Journal of Algorithms, 2004, 51(2):122–144

[119] McKenney P E, Slingwine J D. Read-copy update: Using execution history to solve concurrency problems. In: Proc of Parallel and Distributed Computing and Systems. 1998, 509–518

[120] liburcu. User-level Read-copy Update. Website. http://liburcu.org

[121] Trigonakis V. SSPFD. Website. https://github.com/trigonak/sspfd

[122] Conway P, Kalyanasundharam N, Donley G, et al. Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE micro, 2010, 30(2)

[123] Intel. Intel 64 and IA-32 architectures software developer's manual. 2016.

[124] Mazouz A, Touati S A A, Barthou D. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In: Proc of High Performance Computing and Simulation (HPCS), 2011 International Conference on. IEEE, 2011, 273–279

[125] LIKWID. Likwid-perfctr. Website. https://code.google.com/p/likwid/wiki/LikwidPerfCtr

[126] Dice D, Herlihy M, Lea D, et al. Applications of the adaptive transactional memory test platform.

Applications of the Adaptive Transactional Memory Test Platform Researchgate, 2008.

[127] Intel 64 and IA-32 Architectures Optimization Reference Manual. Order Number, 2011.

[128] Bobba J, Moore K E, Volos H, et al. Performance pathologies in hardware transactional memory. In: Proc of International Symposium on Computer Architecture. 2007, 81–91

[129] Guerraoui R, Kapalka M. On the correctness of transactional memory. In: Proc of Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008, 175–184

[130] Mitzenmacher M. Compressed bloom filters. IEEE/ACM transactions on networking, 2002, 10(5):604–612

[131] Fan B, Andersen D G, Kaminsky M, et al. Cuckoo filter: Practically better than bloom. In: Proc of Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. ACM, 2014, 75–88

[132] Dalessandro L, Spear M F, Scott M L. NOrec: streamlining STM by abolishing ownership records. In: Proc of ACM Sigplan Notices, volume 45. ACM, 2010, 67–78

[133] Dice D, Shalev O, Shavit N. Transactional locking II. In: Proc of DISC, volume 6. Springer, 2006, 194–208

[134] Avni H, Kuszmaul B C. Improving htm scaling with consistency-oblivious programming. In: Proc of 9th Workshop on Transactional Computing, TRANSACT, volume 14. 2014

[135] Afek Y, Avni H, Shavit N. Towards consistency oblivious programming. Principles of Distributed Systems, 2011. 65–79