Getting to the Root of Concurrent Binary Search Tree Performance

综述报告

1.背景

这篇文章由 Maya Arbel-Raviv, Trevor Brown, Adam Morrison 等人发表在 2018 年的 USENIX Annual Technical Conference 会议上。该会议于 2018 年 7 月 11-13 日在美国波士顿举行，是操作系统、体系结构方面最好的会议之一。

许多系统依赖乐观的并发搜索树(BSTs)来实现多核可扩展性，文章发现并解释了具有相似树结构和搜索实现的 BSTs 之间显著的意想不到的性能差异，进一步得出了避免这种性能下降的规定方法，以及关于 BSTs 设计的算法见解。

2.关于 BSTs

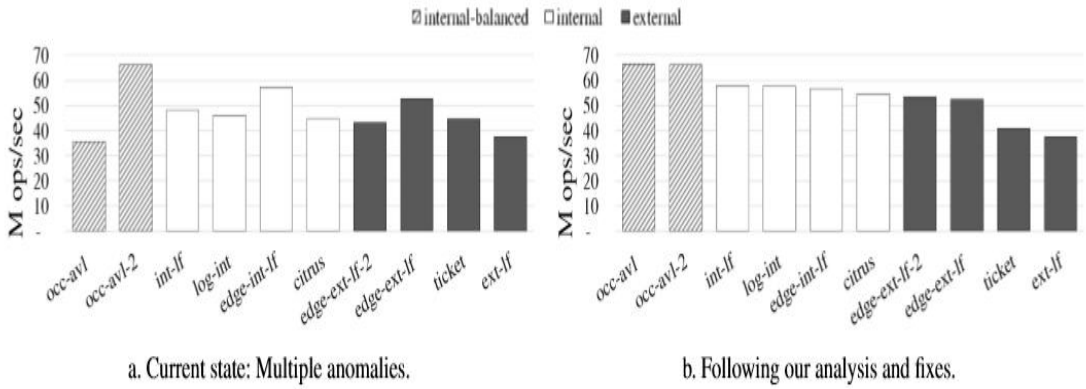
BSTs 原理

乐观的搜索树似乎有一个简单的性能故事，基于这样的观察：要很好地扩展工作负载，必须包含足够的高级并行性(例如，操作不应该都修饰同一个键[1])。因此，乐观搜索树努力避免语义上不冲突的操作之间的同步争用，例如对不同关键字的更新。特别是，乐观树使用只读搜索，不会锁定或以其他方式写入遍历的节点，对共享内存的写入只发生在修改数据结构[2,3]。这种设计被认为是搜索树性能的关键[4,5]。

发现的问题

然而，实现乐观树设计的全部性能优势远不简单，因为它们的性能受到难以识别和解决的系统软件和硬件子系统的微妙交互的影响。为了证明这个问题，文章假设搜索性能将是一个主要因素。（毕竟，大部分时间将花在搜索树上，同步(如果有的话)只发生在搜索结束时。）特别是，我们期望两棵树具有相似的结构，例如具有对数高度的平衡树，表现相似。

然而，实际上，这种期望被证明是错误的。我们在乐观二进制搜索树上测试上述推理，发现具有相似结构和遍历技术的 BSTs 之间存在显著的性能差异。图 1a 描述了这种异常的例子。



a. Current state: Multiple anomalies.　　b. Following our analysis and fixes.

图 1：意外的 BST 性能结果

本文贡献

这项工作的目标是解释和解决这种出乎意料的性能结果。我们在大规模 x86-64 硬件上对 10 种最先进的乐观 BST 实现进行了详细的性能分析，从中我们发现了观察到的异常的根本原因。通过使用微基准，我们发现性能异常是由 BSTs 与系统软件和硬件子系统的多种性能退化交互作用造成的，这些交互作用主要与缓存效应有关。这些缓存效应要么是由于缓存不友好的实现疏忽，要么是由于 BST 和内存分配器之间的交互导致的内存布局问题。为了确定我们的观察结果是否仅仅是微观基准测试的产物，或者类似的问题是否出现在更复杂的软件中，我们将 BST 作为索引结构部署在内存数据库[6,7,8,9]中。我们发现 DBx1000 中也存在类似的异常。最重要的是，我们提出一种隔离 BST 相关的分配的简单方法，而不是使用混合数据，这样可以将 BST 的性能提高 20%，将整个应用程序的性能提高多达 10%。图 1b 展示了我们的部分结果。

面对的挑战

我们的工作强调多核性能的理论和实践之间的差距。正如我们所展示的，理解搜索树的性能并不简单，特别是性能是由于基本算法因素还是实现问题。当我们关注 BSTs 时，我们发现的效果与其他乐观并发数据结构相关，因为它们来自内存分配器和系统设计的一般原则。(但是我们将这种分析留给未来的工作。)因此，我们的研究结果需要进一步的研究，以帮助弥合多核性能的原理和实践之间的差距，简化部署并发数据结构的任务以及对其性能的推理。

3.相关工作

内存布局问题。我们发现的一些现象在[10,11,12]中有报道，但是这些作品没有考虑所有因素的组合及其对 BST 性能的影响。早期的研究提出了编译器和库技术，通过在内存[11,12]中仔细放置对象来提高缓存利用率，但是这些技术没有部署，

因此不清楚它们是否会解决我们考虑的异常。

隔离分配。基于区域的内存管理[13,14]从专用内存池中分配每个对象类型。然而，它的动机是加速内存分配和释放，而不是提高缓存和 TLB 利用率。lattner 和 Adve [15]提出了一种编译器算法，用于将不同的数据结构实例分离成不同的池。他们的方法不会在数据结构中隔离分配，这可能是避免未充分利用缓存集所必需的。

性能理解。几项研究比较了并行数据结构[4,5]的性能，但没有分析性能差异的根本原因。我们的工作是对理解实验评估结果困难的研究的补充，[6,13,31]没有考虑并发数据结构。

4.建议

这项工作中吸取的经验教训可以应用于其他并发数据结构，因为它们源自一般性能原则。在这里，文章试图将这些教训提炼为具体建议。

数据结构设计者和实现者：研究数据结构的内存布局。(1)如果与节点相邻的高速缓存线通常包含其他对象，则节点可能未充分利用高速缓存。(2)使用填充来避免错误共享，特别是在频繁访问的节点和其他程序数据之间，且这种填充应该考虑预取。(3)不分青红皂白地填充所有节点可能会降低性能，因为这会减少 LLC 中适合的节点数量。

使用数据结构的程序员：数据结构导入程序会合并两个内存空间，并可能会产生或消除错误共享或缓存未充分利用的问题。因此，应该(a)检查数据结构和程序的组合内存布局，并解决这些问题，或者(b)使用单独的分配器分离数据结构的内存。

内存分配器设计者和实现者：在内存分配器的额外支持下，上述建议将更容易付诸实践：首先，为分配分离提供一个接口。第二，为内存布局检查提供接口或工具，以允许确定(1)对象到大小类的映射；(2)哪些对象类型经常位于存储器中彼此靠近；以及(3)将对象分配到 LLC 中的高速缓存集中(对于每个对象类型)。这种查询也可能导致用于识别内存布局问题的高质量自动化工具。

5.感谢

这项工作部分由 ISF 资助(2005/17 和 1749/14 赠款)。Maya Arbel-Raviv 部分获得了 Hasso Platner 技术研究所的支持。Trevor Brown 获得了 NSERC 的博士和博士后研究金，并获得了 RGPIN-2015-05080 号赠款。Adam Morrison 得到了 Len Blavatnik 和 Blavat-Nik 家庭基金会的支持。

6.参考文献

[1] HERLIHY, M., AND SHAVIT, N. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA,USA, 2008.

[2] BRONSON, N. G., CASPER, J., CHAFI, H., AND OLUKOTUN, K.A Practical

Concurrent Binary Search Tree. In PPoPP (2010).
[3] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache Craftiness for Fast Multicore Key-value Storage. In EuroSys (2012).
[4] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In ASPLOS (2015).
[5] GRAMOLI, V. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In PPoPP (2015).
[6] AVNI, H., AND BROWN, T. Persistent Hybrid Transactional Memory for Databases. VLDB 10, 4 (Nov. 2016).
[7] LIM, H., KAMINSKY, M., AND ANDERSEN, D. G. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In SIGMOD(2017).
[8] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. VLDB 8, 3 (Nov. 2014).
[9] YU, X., PAVLO, A., SANCHEZ, D., AND DEVADAS, S. TicToc:Time Traveling Optimistic Concurrency Control. In SIGMOD(2016).
[10] AFEK, Y., DICE, D., AND MORRISON, A. Cache Index-Aware Memory Allocation. In ISMM (2011).
[11] [28] LVIN, V. B., NOVARK, G., BERGER, E. D., AND ZORN, B. G.Archipelago: trading address space for reliability and security. In ASPLOS (2008).
[12] WU, C.-J., AND MARTONOSI, M. Characterization and Dynamic Mitigation of Intra-application Cache Interference. In ISPASS(2011).
[13] HANSON, D. R. Fast allocation and deallocation of memory based on object lifetimes. SPE 20, 1 (1990).
[14] ROSS, D. T. The AED Free Storage Package. ACM 10, 8 (Aug.1967).
[15] LATTNER, C., AND ADVE, V. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In PLDI (2005).