面向高效并发栈的分片消除与合并

Ajay SinghFORTH ICS
ajay.singh1
@uwaterloo.ca

Nikos
MetaxakisFORTH ICS
csdp1437@csd.uoc.gr

Panagiota
FatourouFORTH ICS
faturu@ics.forth.gr

arXiv:2601.04523v1 [cs.DC] 8 Jan 2026

摘要

我们提出了一种新的阻塞线性化栈实现，它利用分片和取值并递增，与所有现有并发栈相比，性能显著更佳。该实现基于一种新颖的消除机制和一种新的合并方法，并将二者高效结合以获得高性能。我们的实现提升了并行性，并降低了访问共享栈时的竞争。实验表明，在大多数工作负载下，所提出的栈实现的性能最高可比所有现有并发栈提升 2×。在支持大量线程的系统中，以及在高竞争场景下，该实现尤其高效。

关键词： 并发栈、消除、软件合并、并发数据结构

1 引言

栈是各种应用以及操作系统和系统软件中使用的基础数据结构。它们支持对元素进行压入、弹出和查看栈顶操作，以后进先出（LIFO）的方式管理元素。并发栈被广泛用作并发池 [13], 垃圾回收中的共享空闲列表 [1, 29], 以及并发图算法的构建模块 [17]。多个线程可能会同时尝试访问栈，这通常需要对指向栈顶元素的共享指针栈顶进行原子访问，从而产生大量缓存失效流量。因此，朴素的并发栈实现不仅无法实现加速，还可能导致性能急剧下降。为缓解这一串行瓶颈，文献中提出了多种技术，包括消除 [13,、第 11],软件合并 [6, 8, 11, 15, 19], 时间戳 [3], 以及这些技术的组合，因为消除可以构建在其他技术之上。

消除利用了这样一个观察：两个语义相反的操作（例如压入和弹出）可以有效地相互抵消，使数据结构保持在仿佛两者都未发生过的状态。这样，操作无需访问共享数据结构即可完成，从而减少缓存失效流量。例如，消除-退避（EB）栈 [12], 采用单个消除数组，线程通过该数组成对交换信息，以确定彼此的操作是否可以消除。它利用自适应机制动态确定消除数组的大小，从而实现良好的消除度（即平均而言有较多操作被消除），进而减少访问共享栈时的竞争。然而，其性能已被证明不如基于软件合并 [11] 和基于时间戳的栈 [3]的栈实现。主要开销来自线程为判断是否能够进行消除而必须执行的多次比较并交换操作。

软件合并 [6, 8, 11, 15, 19] 旨在降低同步开销，并减少因访问共享栈顶指针而产生的缓存失效流量。具体而言，在任一时刻，由一个线程（称为组合器）获取全局锁后，代表其他线程执行一批操作，同时也执行自己的操作。未担任组合器的线程只需自旋等待组合器处理其操作并报告返回值。所有线程都会在共享列表上发布自己的操作，组合器遍历该列表，将已发布的操作应用于所实现的数据结构，报告返回值，然后释放锁。这种方法降低了栈顶指针处（一个竞争热点）的同步开销，从而减少缓存失效流量。然而，正如我们的实验所示，合并有时会不必要地降低并行性，并且在高并发度下也会造成瓶颈。

事实证明，使用 fetch&add 是设计基础性高效并发数据结构（尤其是先进先出队列）的最先进方法 [5, 7, 9–11, 16] [16, 21]。例如，发表于 PPoPP'13 [16], 的 LCRQ，最近被证明仍是速度最快的并发队列 [22], , 其设计基于一个简单的想法：使用取值并递增来实现两个计数器，一个用于为入队操作分配不同的序列号，另一个用于出队操作。这样，入队操作以序列号 i 插入队列的元素，就会由具有相同序列号的出队操作消除



本作品采用知识共享署名 4.0 国际许可协议授权。*PPoPP '26*，澳大利亚新南威尔士州悉尼©2026年版权归所有者/作者所有。ACM
ISBN
979-8-4007-2310-0/2026/01https://doi.org/10.1145/3774934.3786458

序列号。这是设计 FIFO 队列时顺理成章的思路，因为入队线程从队列的一端插入元素，而出队线程则从另一端移除元素。

然而，使用取值并递增来设计简单高效的并发栈面临诸多挑战，因为要维持 LIFO 属性，线程必须在同一端执行压入或弹出操作。因此，与队列不同，取值并递增尚未在栈设计中得到有效应用。

本文提出了一种新的栈实现，称为SEC（分片消除与合并），详见第3节。SEC 通过将消除与合并整合到统一设计中，高效地利用取值并递增来实现高性能。这些机制共享多个组件，从而能够高效实现，避免重复开销。因此，该算法无缝融合消除与合并，并为阻塞线性化栈引入了一种新颖设计。我们的实验评估（第6节）研究了SEC 栈在各种设置下的性能，结果表明，新栈的表现始终优于最先进的并发栈。此外，消除和合并技术本身也具有独立的研究价值，可应用于其他场景，例如设计高效的并发双端队列或相关数据结构。

SEC 采用分片消除与合并，以避免先前消除与合并技术的性能瓶颈和局限性。它利用多个聚合器，每个聚合器负责一部分线程。每个聚合器中的线程将压入和弹出操作组成批次。在每个批次内，线程协作以消除并合并操作。最终，一个批次中只剩下全部为压入操作或全部为弹出操作的情况，随后这些操作将被应用于栈。

这种批次级消除显著减少了最终需要访问共享栈的线程数量，从而减轻了栈顶端指针处的竞争。在批次内进行合并，可进一步减少栈顶端指针处的竞争。我们的实验，以及近期发表于 PPoPP'24 [21], 、用于在软件中实现 fetch&add 的聚合漏斗技术，都证明了将线程划分到多个聚合器中的优势；这种做法类似于分片，可降低在大型多核机器上竞争热点的开销。

在 SEC 中，为每个以入栈操作为主的批次设置一个独立的组合器，可以提高并行性，因为这样组合器便能并发地创建要追加到共享栈中的节点子栈。这也减少了算法执行昂贵同步原语（例如比较并交换）的次数，因为每个组合器只需执行一次比较并交换，便可尝试将整个子栈追加到共享栈中。类似地，对于以弹出操作为主的批次，其组合器会尝试仅更新一次 *top* 指针（通过一次比较并交换），从栈中移除整个最顶层节点链，以便将这些节点返回给其批次中未被消除的弹出操作。

因此，我们的方法能够实现更高的并行性。少量组合器并发工作，这一点不同于经典的软件合并方法，例如 CC-Synch [6] 或扁平合并 [11],，后者由单个组合器按顺序执行栈上的所有操作。

SEC 的另一项创新在于，它通过在每个批次中有效地使用两个计数器（以取值并递增的方式实现），大幅简化并加速了消除和合并机制。线程只需递增其中一个计数器，即可在批次中声明其操作。之后，这些计数器会用于确定哪些操作以及多少操作将被消除，哪些操作将应用于共享栈。因此，与以往算法相比，SEC 的消除和合并成本大幅降低。例如，与传统 EB 算法 [12], 每对压入和弹出操作最多需要执行三次 CAS 操作不同，本文提出的算法只需对两个独立的共享计数器执行两次取值并递增操作，即可支持消除。这显著降低了竞争。使用计数器还使该算法能够通过一种简单的方法，确定未被消除的弹出操作的返回值。

通过这些创新，我们的方法在不对不同操作的性能造成不成比例影响的情况下实现了更高的吞吐量，并且在高线程数下显著优于现有最先进的算法。SEC 是阻塞式的，因为其消除和合并机制涉及等待。我们的实验表明，在低竞争情况下，SEC 并无明显优势；但在线程数较高时，它显著优于所有竞争算法。

本文贡献 包括：

- 一种基于线程分片并高效利用取值并递增的轻量级消除机制。该机制本身具有独立的研究价值，也可用于设计其他并发数据结构（例如双端队列）。
- 一种将消除机制与高效合并方法相结合的技术，在共享栈上执行压入和弹出操作时，可实现更高的并行性。这些机制显著降低了访问共享栈时的竞争。
- 一种新颖的 C++ 并发栈实现，将所提出的消除和合并技术结合起来，使二者共享多个组件，从而避免重复承担某些开销。
- 全面的实验分析表明，在大型非一致性内存访问系统中，所提出的栈实现于许多情况下优于所有先前的并发栈。

2 相关工作

文献中的若干技术 [4, 11, 12, 20, 23] 旨在降低访问栈顶指针时由竞争瓶颈造成的开销。这些技术利用指数退避、消除、合并、时间戳或这些方法的组合，来减少在栈顶同步操作所产生的高昂开销。

指数退避 [13, 第 7 节4] 是一种广为人知的技术，在多种场景中得到广泛应用。在这种方法中，观察到竞争的线程会随机退避一段时间，让竞争线程有机会完成其操作。退避时间会根据观察到的竞争情况进行调整。SEC 采用了一种简单的退避方案，但其目的有所不同。在 SEC 中，特定线程（称为冻结线程）会短暂退避，以提高分配给每个批次的操作数量，从而可能提高消除度和合并度。

文献 [12] 提出了将消除作为栈的一种替代退避方案。EB [12] 是一种无锁栈，通过单个消除数组将消除作为退避机制。EB 的消除机制开销相当大。线程必须执行三次比较并交换操作，才能协作消除一对操作。SEC 不将消除用作退避，而是在访问共享栈之前主动尝试消除操作。其消除机制开销较低，仅需执行两次取值并递增操作，即可支持消除一对操作。

软件合并 [6, 8, 11, 15, 19] 是一种同步技术，其中单个合并线程获取全局锁，并代表其他多个线程执行它们的请求，从而显著降低同步开销和缓存失效 [11]。软件合并已用于实现共享栈 [6, 8, 11, 15]。SEC 通过使用多个聚合器和批次改进了这些栈，从而提高并行性并减少竞争。

时间戳被用于 [4] ，它利用线性化允许并发操作重新排序这一特性，解决栈中单一热点的问题。在执行相应的入栈操作期间，每个元素都会关联一个时间戳。这样，线程无需对单个栈顶指针进行同步，就能压入元素。然而，这会增加弹出和查看操作的成本，因为为确保正确性，弹出和查看操作必须扫描带有时间戳的元素，并从栈中返回时间戳最高的元素。在我们的一些实验中，线程会执行查看操作，以研究这种开销转移。事实上，在读密集型工作负载下，TSI [4] 的性能不如 SEC。

SEC 借鉴了 [21]，中聚合漏斗方法通过嵌套分区分散竞争的思路，该方法实现了一种软件 Fetch&Add 原语。与聚合漏斗类似，SEC 使用多个聚合器并在每个聚合器中使用批次，但通过引入 a) 一种新颖的批次冻结机制、b) 一种新的轻量级消除方案、c) 一种支持两类操作（压入和弹出）的组合策略，以及 d) 一种向未被消除的弹出操作返回值的高效机制，扩展了这一设计，使其能够实现栈。因此，它大幅拓展了 [21] 的技术，并将其应用于设计一种并发栈，从而提升并行性、减少竞争，并在大型多核机器上实现良好的扩展性。

3 算法

我们首先简要概述该算法的工作原理。在 SEC 中，线程以组为单位进行协调，以减少竞争并提高局部性。这是通过利用多个聚合器实现的。每个线程都被分配到一个固定的聚合器结构中。线程在所分配的聚合器中进行协调。SEC 使用聚合器的方式，为消除方案和组合机制提供了高效的实现。

执行操作 op 的线程的首要目标，是确定 op 是否可以被消除。为此，聚合器中线程调用的操作被划分为多个批次。同一批次中的操作之间会发生消除。这需要一种冻结机制来确定属于每个批次的操作，同时还需要在批次内部设置一个数组，以便一个入栈操作和一个弹出操作能够相遇并交换信息。如果确实发生了交换，则这对入栈和弹出操作就会被消除。

批次中未被消除的操作（它们的类型全部相同，即要么全部是入栈操作，要么全部是弹出操作）会在 SEC 使用的共享栈上执行。为减少竞争，在任一时刻，每个批次中至多只能有一个组合器线程存在。组合器的职责是代表发起这些操作的线程执行批次中未被消除的操作；这些线程会进行本地自旋，等待组合器通知它们操作已得到处理。为了在多个批次之间并行执行入栈操作，并减少对共享栈状态执行昂贵同步原语（例如比较并交换）的次数，每个批次的组合器都会创建一个本地子栈，其中包含它希望压入共享栈的元素。随后，它通过执行比较并交换，尝试将本地子栈追加到共享栈中。类似地，希望从栈中弹出多个操作的组合器会尝试一次性弹出所有这些操作，方法是以原子方式更改栈顶指针，使其指向适当的后续栈节点。

章节 3.1 详细概述了该算法。章节3.2 介绍了SEC 伪代码的详细信息。

3.1 算法描述

• **聚合器和批次**。 该算法使用 K 聚合器 和共享栈。每个线程都会被分配到一个

特定的聚合器。我们用 P 表示分配给每个聚合器的最大线程数。聚合器中的线程彼此竞争。分配给某个聚合器的线程发起的操作 A 会被划分为批次。这样，一个聚合器会使用多个批次（如果执行是无限的，则可能使用无限多个批次）。批次 是一种对象，使线程能够同步并协作，以执行消除和合并。每个聚合器 A 都实现为一个结构体，其中包含一个指向其当前活动批次的指针。

• **消除与冻结**。 在属于聚合器 A的批次 B 的操作中，如果压入（弹出）操作的数量少于弹出（压入）操作的数量，那么 B 的所有压入（弹出）操作都将被弹出（压入）操作消除。最终，该批次中只剩下一种操作类型，并将其应用于共享栈。为了实现消除，分配给 A 的线程需要一种机制，以确定属于 A当前活动批次 B的活动操作集合 S 。确定 S之后，活动线程会在 S中尽可能多地消除操作。

线程通过冻结 机制来决定哪些操作属于 B 。具体而言，B 存储两个计数器，分别是推入计数 和弹出计数。为了执行压入（或弹出）操作，线程通过对推入计数 （或弹出计数）执行取值并递增，在 B 中宣布该操作。因此，推入计数 和弹出计数 分别表示分配给 A 的线程在 B 是 A的活动批次时所宣布 的压入和弹出操作数量。线程在所访问计数器中读取到的值，就是其活动操作的序列号 。相同的序列号只能分配给两种不同类型的操作。

在 B 中宣布第一个压入操作和第一个弹出操作的线程会相互竞争，决定谁将成为 f_B 的冻结器 线程。线程 f_B 将负责冻结 B，也就是说，它会指出一个时间点，在此之后宣布的任何操作都不属于B。序列号为 i 、将其操作宣布到 B 中的线程，会将它打算压入的值记录在数组的第i个元素中，该数组称为消除数组（存储在 B中）。随后，它会自旋等待 f_B 冻结 B。线程在获得序列号后，会立即将要压入的值记录到消除数组中。这样做有以下优点。首先，这可以避免执行弹出操作的线程等待有值可供交换。其次，这可以避免批次合并器不得不等待该值。

B 将属于该批次的实际压入和弹出操作数量存入计数器*pushCountAtFreeze* 和*popCoun-tAtFreeze* ，也就是冻结时已记录在*pushCount* 和*popCount* 中的压入和弹出操作数量。此外， B 使用一个名为*isFreezerDecided*的布尔变量来选择 f_B。*isFreezerDecided* 的初始值为 FALSE。争相成为 B的冻结器的线程尝试使用测试并设置将*isFreezerDecided* 设为 TRUE。获胜的线程将负责 f_B，把*pushCount* 和*popCount* 存入*pushCountAtFreeze* 和*popCountAtFreeze*。

然后， f_B 通过将 A的批次指针改为指向一个新的批次 B'，用一个新批次替换聚合器 A 当前的工作批次。任何新到达的线程都将被分配到 B' 或后续批次。通过更改此指针， f_B 还会通知已将操作宣布到 B 中的其他线程停止自旋。其中，在冻结之后才宣布操作、因此不属于 B的线程，会尝试被分配到 B' 或后续批次。其余线程将尝试通过消除数组消除其操作。考虑其中任意一个线程，设 op 为它的操作，其序列号为 i。如果另一个属于 B 、序列号为 i的操作存在，那么只有它与 op 类型相反时才能进行消除。否则，op 必须在共享栈上执行。操作被消除的线程可以立即开始执行新操作。

• **合并**。在一个批次中， B 其操作未被消除的线程中，有一个线程充当 B 的组合器，负责将 B 的所有未消除操作应用到共享栈上。这样，只有少量线程会并发访问共享栈，从而减少竞争并提升性能。

B的合并线程被选为执行 B第一个未消除操作的线程。每个批次只有一个组合器。为了并行执行不同批次的组合器所执行的入栈操作，每个组合器都会创建一个（本地）子栈，其中包含其批次中所有未消除的入栈操作（记为 m）。随后，组合器尝试通过对栈顶指针执行比较并交换，将整个子栈以原子方式追加到共享栈上。如果比较并交换成功，则所有 m 操作只需通过一次比较并交换即可完成，从而提升性能。

类似地，需要执行 m 弹出操作的组合器会尝试使用比较并交换，以原子方式移除 m 个栈节点，并将栈顶指针更改为指向其后的第 m个节点。所有操作未被消除且不充当组合器的线程，都会进行本地自旋，等待其批次的组合器通知它们，相关操作已应用到共享栈上。组合器会使用一个名为 批次已应用标志 的布尔变量，将其存储到 B中，以通知等待线程停止自旋；同时，组合器会向这些线程返回一个子栈，以便它们找到各自的返回值。具体而言，操作序列号为 i 的线程 q 会读取子栈的第 i个节点；如果该节点存在，就返回其值，否则 q 返回 EMPTY。

为提升性能，冻结线程 f_B 会在冻结前进行短暂的退避 B （在第 29 行之前），以提高 *SEC* 的 消除度，即平均消除的操作数，同时提高其 合并度，即组合器将

▷ 共享栈相关变量1: **结构体**节点 int 值节点* 下一个2: 节点* 栈顶3: int 线程局部 ID ▷ 线程局部 ID▷ 冻结、消除和合并相关变量4: **结构体**批次 int 推入计数, 弹出计数 int 冻结时推入计数 , 冻结时弹出计数 节点* 消除数组 [P]5: 节点* 子栈顶指针 bool isFreezerDecided bool 批次已应用标志6: **结构体**聚合器 批次* 批次7: 聚合器 聚合器[K]

图 1。 关键数据结构和变量

服务。实验表明，这带来了更高的性能。

3.2 伪代码与详细信息

数据结构和变量。 图 1 展示了 SEC 中使用的数据结构和变量。共享栈通过由 节点 （第 1 行）组成的 链表实现，每个节点存储一个整数 值 和一个指向下一个节点的指针。共享栈顶 指针（第 2 行）指向栈的最顶端节点。

该算法使用一个数组 聚合器 的 K 对象，（第6行）每个对象都存储一个指针，指向批次其当前活动的批次。一个批次（第4行）存储四个计数器：推入计数、弹出计数、冻结时推入计数和冻结时弹出计数，以及布尔变量*isFreezerDecided*和批次已应用标志。它还存储消除数组，供该批次中的操作相互消除。

Push 的伪代码。执行 push 操作的线程 q （算法 1）首先在第 2 行根据自己的 tid 获取分配给它的聚合器。在 SEC 中，线程会均匀分布到各个聚合器上（当然也可以采用更复杂的方案 [21]）。例如，有两个聚合器和十个线程时，第一个聚合器负责前五个线程，第二个则负责其余五个。第 3 行， q 分配一个新节点，其中包含待压入的值，且 next 字段设为 null。随后，线程反复执行以下操作（第 4–26 行的循环）：将指向聚合器当前活动批次的指针 B 取到其局部变量 myBatch 中；对 B的 pushCount 执行取值并递增操作（第 6 行），以此声明该操作，并将返回值（作为 q的序列号）存入 mySeqNum；q 使用自己的序列号确定要访问 B的消除数组中的哪个槽位（第 7 行）。

第 8–13 行对应冻结机制。线程q 检查自己是否在以下线程中收到序列号 0，这些线程会递增线程的 B推入计数（第 8 行 if 语句的第一个条件）。如果 q的序列号确实为 0， q 将执行第 8 行的测试并设置操作，以避免与在B中执行第一个已声明弹出的线程发生潜在竞争；该线程也可能尝试成为冻结器。如果测试并设置成功（即返回 0）， q 会将 B的 isFreezerDecided 标志设为 TRUE，并承担冻结器的角色 f_B。如果 q收到的序列号不是 0，它就会自旋（第 12 行），等待 f_B 通过将 B 的批次指针改为指向一个新批次来完成冻结阶段。

冻结器线程 f_B 执行 FreezeBatch 函数（第 28-31行），将 *pushCount* 和 *popCount*的副本分别存入 *pushCountAtFreeze* 和 *popCountAtFreeze*。线程 f_B 还会分配一个新的批次 B'，并将组合器的 *batch* 指针设置为指向 B'（第 31行）。这会通知其他线程冻结已完成。

在第 14行， q 通过将其操作 op 的序列号与 B 中记录的值进行比较来检查 op 是否属于该批次，该值已被记录在 f_B 中B的*pushCountAtFreeze* 中。如果 op 的序列号大于或等于 *pushCountAtFreeze*，则意味着 q 在 f_B 读取 *pushCount*之后增加了 *pushCount* ，因此它不属于 B，必须使用后续批次重试应用其操作（第 4行的 while 循环）。

其入栈操作 op 属于 B 的线程 q 会执行第 14 行 if 语句的主体。在第 15 行， q 检查 op 是否可以被消除。如果 op 的序列号小于属于 B的弹出操作数量，则 op 可以被消除，因此 q 会在第 24 行直接返回 TRUE。否则，op 无法被消除，因此 q 会执行第 17-22 行。在第 16 行， q 通过比较我的序列号与冻结时弹出计数来检查自己是否应成为组合器。如果属于 B 的入栈操作多于弹出操作，则我的序列号等于冻结时弹出计数的线程会成为组合器；也就是说，组合器是在尚未被消除的操作中执行序列号最小的入栈操作的线程。组合器执行 PushToStack 例程（第 17 行），将未被消除的入栈操作应用到共享栈上。其余线程在第 20 行自旋，等待组合器通过将批次已应用标志设置为 TRUE（第 18 行）来通知它们其操作已被应用。

该PushToStack 例程在任一时刻只由一个线程执行，即当时的组合器。PushToStack以指向 B 的指针和组合器操作的序列号作为参数。根据组合器的选取方式，序列号介于压入操作的序列号介于我的序列号和冻结时推入计数之间的所有操作都必须应用于共享栈。组合器会创建一个子栈，其中包含这些操作各自创建的节点（第37-43行）。随后，将子栈的最后一个节点链接到共享栈当前的顶端节点（第41-42行）。具体做法是，使用变量 *bot* 记录子栈最底端的节点（第36行），并使用变量*top* 记录其最顶端的节点（第42行）。在创建子栈的过程中，

算法 1推送算法。

```
1: 函数 压入(int value): void
2:     myAgg ← 聚合[线程 ID/K]
3:     节点* myNode ← 新节点(value, null)
4:     当真 时执行
5:         我的批次 ← 我的聚合批次
6:         我的序列号 ← 取值并递增(myBatch.pushCount)
7:         myBatch.eliminationArray[我的序列号] ←我的节点
8:         如果 (我的序列号 == 0&& !T&S(myBatch.isFreezerDecided)) 那么          ▷ 冻结块
9:             FreezeBatch(myAgg, myBatch)
10:        else
11:            当myBatch == myAgg.batch执行 nop
12:            结束当                                  ▷ 非冻结线程等待冻结器冻结工作批次
13:        结束如果
14:        如果 mySeqNum < myBatch.pushCountAtFreeze则                          ▷ 包含测试。
15:            如果 mySeqNum ≥myBatch.popCountAtFreeze 则                       ▷ 消除测试。
16:                如果 mySeqNum ==myBatch.popCountAtFreeze 则                  ▷ 组合器测试
17:                    PushToStack(myBatch, 我的序列号)
18:                    myBatch.批次已应用标志 ← TRUE
19:                else
20:                    while!myBatch.批次已应用标志 do nop
21:                    结束 while
22:                结束 if
23:            结束 if
24:            返回 TRUE
25:        结束 if
26:    结束 while
27: 结束函数
28: 函数FreezeBatch（聚合器 myAgg, 批次* myBatch)
29:    myBatch.冻结时弹出计数 ← myBatch.弹出计数
30:    myBatch.冻结时推入计数 ← myBatch.推入计数
31:    myAgg.批次 ← 创建新批次(0,0,0,0,0, ⊥， 假, FALSE)
32: 结束函数
33: 函数PushToStack(Batch* myBatch, int 我的序列号)
34:    节点 *top=⊥， *bot=⊥
35:    整数 i ← 我的序列号
36:    底部 ← myBatch.eliminationArray[我的序列号]
37:    当 ++i < 批次.冻结时推入计数执行                                          ▷ 从 myBatch 中的推入操作准备一个子栈
38:        当!myBatch.eliminationArray[i] 执行 空操作
39:        结束循环
40:        临时节点 ← myBatch.eliminationArray[i]
41:        临时节点.next ← 栈顶
42:        栈顶 ← 临时节点
43:    结束 while
44:    while TRUE 执行                                                          ▷ 将子栈添加到共享栈
45:        节点* 临时栈顶 ← 栈顶
46:        底部.下一个 ← 临时栈顶
47:        如果 比较并交换(栈顶, 临时栈顶, 顶部) 则
48:            返回
49:        结束 if
50:    结束 while
51: 结束函数
```

算法 2 弹出算法

```
52: 函数弹出( ): int
53:     myAgg ←聚合[tid/K]
54:     当 为真 时执行
55:         myBatch ← myAgg.batch
56:         我的序列号 ← 取值并递增(myBatch.popCount)
57:         如果 (我的序列号 == 0 && !T&S(myBatch.isFreezerDecided)) 那么                ▷ 冻结块
58:             FreezeBatch(myAgg, myBatch)
59:         else
60:             当myBatch == myAgg.batch执行nop
61:             结束当
62:         结束如果
63:         如果 mySeqNum < myBatch.popCountAtFreeze那么                                ▷ 包含测试
64:             如果 mySeqNum < myBatch.pushCountAtFreeze那么                           ▷ 消除测试。
65:                 当!myBatch.eliminationArray[mySeqNum] 执行 nop
66:                 结束当
67:                 返回 myBatch.eliminationArray[mySeqNum].value
68:             结束如果
69:             如果 mySeqNum ==myBatch.pushCountAtFreeze 则
70:                 PopFromStack(myBatch, 我的序列号)
71:                 myBatch.批次已应用标志 ← 真
72:             else
73:                 当!myBatch.批次已应用标志 执行 空操作
74:                 结束循环
75:             结束条件判断
76:             返回GetValue(myBatch,我的序列号 - myBatch.pushCountAtFreeze)
77:         结束 if
78:     结束 while
79: 结束函数
80: 函数 PopFromStack(Batch * myBatch, int我的序列号)
81:     int i ← 我的序列号
82:     当 TRUE 时执行
83:         节点 *top ← 栈顶
84:         节点 *bot ← 栈顶
85:         当 ++i < 批次.冻结时弹出计数 执行
86:             如果 bot == 空 则跳出循环
87:             结束如果
88:             底部 ← 底部.下一个
89:         结束循环
90:         如果 比较并交换(栈顶, top, 底部) 则 跳出
91:         结束条件
92:     结束 while
93:     myBatch.子栈顶指针 ← top
94: 结束函数
95: 函数GetValue(Batch *myBatch, int 我的序列号)
96:     temp ← myBatch.子栈顶指针
97:     重复 我的序列号 次次执行
98:         如果 temp == 为 null 则返回EMPTY
99:         结束 if
100:        temp ← temp.next
101:    结束 for
102:    返回 temp.value
103: 函数结束
```

组合器可能需要等待（第38行），直到被分配了序列号的线程 i 将其节点记录到消除数组中。写入后，该节点会在第40行被读取，并在第41行链接到子栈的栈顶。随后，栈顶会在第42行更新为这个新链接的节点。

组合器会反复执行第44 行至第50 行，直到成功更新栈顶。这包括读取栈顶的当前值，将子栈最底部的节点链接到该值，并尝试通过一次比较并交换将栈顶设置为指向子栈最顶端节点的指针。如果比较并交换成功，函数就会返回；否则会重试。

Pop伪代码。弹出操作的公告和冻结部分与压入操作类似。具体而言，执行 q 弹出操作的线程会获取分配给它的聚合器 A （第53行）以及 A当前的活动批次 B （第55行）。然后，它对B的弹出计数执行取值并递增操作（第6行），以公告其操作，并将返回值存入我的序列号，作为其序列号。第57-62 行对应冻结代码，与压入操作的冻结代码相同。

在第 63 行， q 通过将其序列号与 B 的值进行比较，检查它是否属于 冻结时弹出计数。如果 q 的序列号大于或等于该值， q 就必须在后续批次中重试。如果 q 属于 B，它会在第 64 行检查其操作op 是否可以消除。如果 op 可以消除， q 就会在第 65 行自旋，直到将消除该操作的 压入操作把它的节点写入消除数组[我的序列号]
节点可用后， q 会完成 op 的执行，并返回节点中记录的值。

如果 op 无法消除， q 会检查自己是否应成为组合器（第 16 行）。如果属于 的 B 弹出操作比 压入操作多，那么在尚未消除的弹出操作中，序列号最小的那个操作所在线程将成为组合器。组合器执行 PopFromStack 例程，将未消除的弹出操作应用到共享栈上。其余线程等待组合器通知它们操作已应用。

组合器会在共享栈上执行序列号介于我的序列号和冻结时弹出计数 之间的所有弹出操作。在 PopFromStack 中，组合器首先将栈顶 的当前值记录到局部变量*top*中，然后使用局部变量*bot* 遍历与未消除的弹出操作数量相同的栈节点，前提是栈中有足够的节点。接着，组合器执行比较并交换，将栈顶 更新为指向与*bot*相同的节点。如果比较并交换成功，函数就会返回；否则，组合器会重试上述步骤。

为了查找响应，执行未消除弹出操作的线程 q 会调用 GetValue。在 GetValue 中，线程遍历已移除的子栈，根据各自的序列号查找返回值。序列号为 $SeqNum+i$ 的线程（其中 $SeqNum$ 是组合器的序列号），如果已移除的子栈中存在第 i个节点，则接收该节点；否则返回 null。

查看栈顶。 我们省略了查看操作的伪代码，因为它只是读取栈顶，与 Treiber 栈 [28]类似。

正确性和内存回收的讨论见补充材料。

4 内存回收

这里简要介绍如何将内存回收算法集成到我们的栈实现中。在 SEC 中，我们采用 Brown 实现的基于纪元的内存回收算法（DEBRA）来回收批次和栈节点 [2]。其他内存回收算法 [1, 14, 18, 24–27]也可以用相同的方式应用。在弹出操作中，当未消除的等待线程从组合器返回的子栈中取回其节点时，该共享栈节点会被交由内存回收算法回收。批次则会在以下任一情况下被回收：消除后批次为空（因为其中所有操作都已被消除），此时由冻结器将其交由回收；或者其中所有操作都已应用于共享栈，此时由组合器将其交由回收。

5 正确性与进展

我们的 SEC 算法是一种线性化的栈实现。详细的正确性证明见附录。

考虑属于聚合器 A 的批次 B 中的操作 op。令 q 为发起 op 的线程。

如果 op 被消除，令 op′ 为消除 op 的操作。回想一下，op 和 op' 是类型相反且序列号相同的操作。此外，op 和 op' 必须读取了同一个批次。设 op 为压入操作，op' 为弹出操作（反向情况同理）。令 t 和 t' 分别为 op 和 op' 对 B.pushCount 和 B.popCount 执行取值并递增、以宣布其操作的时刻。操作 op 在第 7 行将其值写入由我的序列号表示的槽位，而 op' 在第 67 行读取 op 返回的值。我们将 op 和 op′ 线性化于 t 和 t' 中较晚的时刻，即 op 和 op′ 彼此消除的时刻。

现在假设 操作 未被消除。那么，存在一个组合器线程 c_B 用于 B，它执行了 操作 (c_B 可能是 q，也可能是执行属于B的其他操作的线程）。如果 操作 是一次 压入 操作，我们将其线性化于c_B 成功执行第 比较并交换 行 47 的时刻，该行位于 PushToStack 中。类似地，如果 操作 是一次 弹出 操作，我们将其线性化于 c_B 成功执行第 比较并交换 行 90 的时刻，该行位于 PopFrom-Stack 中。如果多个操作在线性化时刻相同，则按序列号打破平局；序列号较小的操作先线性化。

性质 5.1。 SEC栈是阻塞的

在SEC中，线程会等待冻结器和组合器线程执行各自的操作。此外，在消除过程中，执行弹出的线程可能会等待压入操作（第65行）。因此，该算法是阻塞的。

6 实验评估

实验设置。实验分别在一台英特尔 Emerald Rapids（**Emerald**）机器和一台 Ice Lake-SP（**IceLake**）机器上进行。Emerald 配备 2 个 NUMA节点，每个节点有 12 个核心，并支持双路超线程，共有 56 个硬件线程。IceLake 配备 4 个 NUMA节点，每个节点有 12 个核心，同样支持双路超线程，共有 96 个硬件线程。SEC 使用 C 实现++。我们的基准测试使用 C++14和-O3 优化进行编译，在 Ubuntu 24.04 上运行，使用 mi- malloc 分配器和基于纪元的回收算法 Debra [2]，以回收SEC中的内存。

我们还在一台配备英特尔 Sapphire Rapids 处理器、拥有 192 个线程的机器上进行了实验。IceLake 和 Sapphire Rapids 的图表见第 D 节和 E节；所有机器上的趋势都类似。

测试算法。我们测试了 SEC，并将其性能与五种知名栈实现进行比较，分别是 Treiber栈（TRB） [28]、消除退避栈（EB） [12]、基于扁平合并的栈（FC） [11]、基于 CC-Synch 的栈（CC） [6]，以及文献 [4] 中基于时间戳的栈（TSI）的区间变体（这是文献 [4] 中介绍的性能最佳变体）。

FC 和 CC 的实现取自 CC [6]，随附的基准测试框架，EB 和 TSI 的实现则取自文献 [4] 中的基准测试。我们采用了 TSI 基准测试的默认设置。我们还测试了其他设置（例如其他延迟值），但这些设置下算法的性能并未优于默认设置。

方法。每次实验持续 5 秒，使用的栈初始预填充了 1000 个节点。同样，改变预填充大小对性能没有显著影响。执行期间，线程会随机执行压入、弹出或查看栈顶操作，所用数值均匀分布于指定范围内。我们报告在五次运行中取平均的吞吐量（每秒百万次操作），并针对不同线程数测试了三种工作负载类型：读密集型（90% 查看栈顶、5% 压入、5% 弹出；即10% 更新）、混合型（50% 查看栈顶、25% 压入、25% 弹出；即50% 更新）和更新密集型（50% 压入、50% 弹出；即100% 更新）。我们测试了启用和未启用线程固定的情况，发现性能没有显著差异，因此报告未启用线程固定时的结果。在操作之间加入随机工作，也没有改变我们观察到的性能趋势。

在 Emerald 和 IceLake 上，线程数分别超过 56 和 96 后，系统会出现超额订阅。除非另有说明，我们在 SEC 实验中使用两个聚合器，并将线程均匀分配到这两个聚合器上，因为这种配置能带来最佳性能。在所有 SEC 实验中，吞吐量的波动均低于 5%。

分析。图 2 展示了 SEC在上述工作负载下的吞吐量，并与其他竞争栈进行了对比。

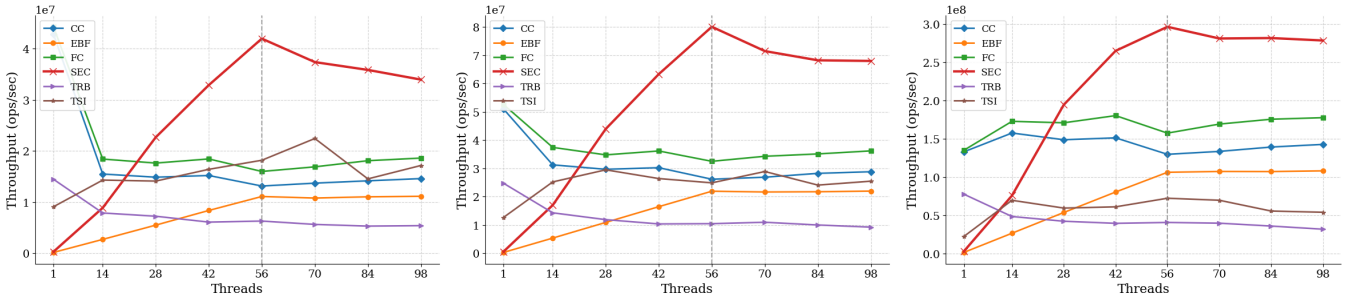
在 Emerald（图2a）中，SEC 在所有工作负载下的吞吐量均比 5× 高出 1.8 至 2 倍。 FC 和 CC 在所有工作负载下都是如此。 FC和CC 通过按顺序执行整个压入和弹出操作来限制并行性。这降低了共享栈顶指针处产生的同步开销，但很快就会成为瓶颈。相比之下，SEC中的组合器会并发执行它们所服务操作的（部分）内容，仅在访问栈顶指针时才进行串行化。因此，它们执行的临界区更短。这提高了并行性，进而带来更好的性能。

从图 2a中可以看到，SEC 的表现优于 TSI，在线程数较大时尤其明显。与相比，SEC ，随着更新率降低（50% 和 10%），TSI的性能会下降。这是由于弹出 和查看栈顶操作在 TSI中开销较高。为实现无需同步的TSI将开销转移到了弹出 和 查看栈顶 操作上。在 100% 更新率（50% 压入和 50% 出栈操作）时，压入和 弹出 操作的均衡组合，使得无需同步的压入 操作在 TSI 中能够抵消 弹出 操作的成本，从而提升性能。相比之下，在 50% 和 10% 的更新率下，弹出 和 查看栈顶 操作的占比更高，增加了开销并降低了性能。

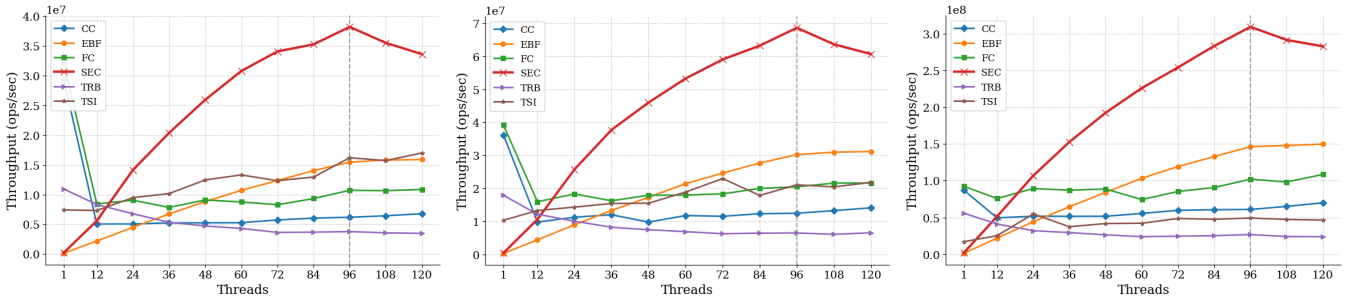
在 IceLake（图 2b）上，SEC 的速度最高可达 3× 比 TSI快 2.2 倍，并且在所有工作负载下都具有更好的扩展性。TSI使用 x86 专用的 RDTSCP 指令来定义高效消除所需的时间间隔。为此，它在时间间隔的起始和结束之间引入延迟，这会增加延迟并对TSI的性能产生负面影响。

在其他算法中，只有 EB 能够良好扩展。这得益于其基于消除的退避机制。然而，其速度仍比 SEC 慢至 2.6× 倍。其吞吐量较低，源于它采用的消除机制比 SEC 更繁重。EB 需要执行三次比较并交换才能实现消除，而 SEC 则采用基于取值并递增的更快消除机制，并通过分片降低竞争。此外，SEC 会主动采用其消除机制，而 EB 仅将消除作为后备方案。另外，EB 对消除数组的使用方式可能导致本可消除的操作对未能消除，从而限制了该方法的消除度。例如，某个压入操作可能在消除数组的某个槽位上等待，却未能实现消除，尽管此时可能有并发的弹出操作，只是它们选择了数组中的其他槽位。相反，在 SEC 中，*PushCounter* 和 *PopCounter* 的使用确保了每个批次内的消除度达到最优。

图 3 聚焦于 Emerald 机器上的仅推送和仅弹出工作负载。IceLake 上的实验显示出类似的性能趋势，详见第 D 节。这些实验展示了测试算法在不进行消除时的吞吐量，并突出说明了以下技术：



(a) Emerald 上的吞吐量。系统在 56 个线程后超额订阅。



(b) IceLake 上的吞吐量。系统在 96 个线程后超额订阅。

图 2。 吞吐量。（左）100% 更新。（中）50% 更新。（右）10% 更新。纵轴：每秒操作数（百万）。横轴：线程数。使用的聚合器数量为两个。

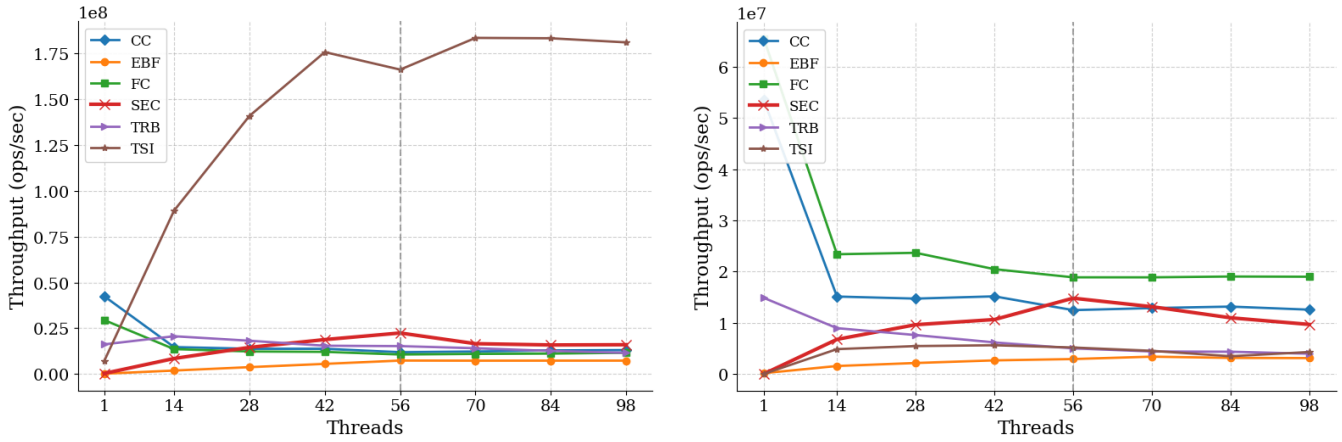


图 3。 Emerald 上仅推送和仅弹出工作负载的吞吐量。（左）仅推送。（右）仅弹出。纵轴：每秒操作数（百万）。横轴：线程数。使用的聚合器数量为两个。

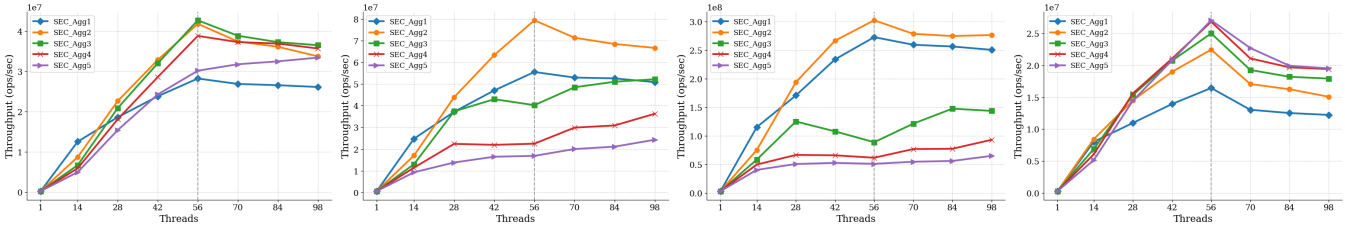


图 4。 比较 Emerald 上 SEC 在不同聚合器数量下的吞吐量。从左到右依次为：100% 更新、50% 更新、10% 更新、100% 仅推送。纵轴：吞吐量。横轴：线程数。使用 1 个聚合器的 SEC 标记为 SEC_Agg1。

给栈操作带来不对称的开销。例如，TSI 的压入操作很快，但弹出操作的开销较高；而包括 SEC 在内的其他技术，在压入和弹出操作上的性能相近。具体而言，在仅入栈基准测试中，TSI 的速度最高可比 SEC 快 6×；而在仅出栈基准测试中，SEC 的速度最高可比 TSI 快 3×。在仅推送工作负载中，TSI 的速度更快，因为其压入操作通过将节点插入线程本地池，避免了在共享栈顶指针处进行同步。然而，这种设计将开销转移到了弹出和查看栈顶操作上。在仅出栈工作负载中，这一点尤为明显：由于弹出操作的开销过高，TSI 的性能显著下降。

图 4 展示了聚合器数量对 SEC 性能的影响，其中我们将聚合器数量从 1 改为 5。我们将包含 m个聚合器的配置记作“Agg$m$”。例如，SEC 使用 1 个聚合器时，标记为 SEC_Agg1。

在仅入栈工作负载中（图4最右侧的图），聚合器数量越多越好（两个或更多），因为这样能更好地分散线程间的竞争：先通过聚合器分散，再通过各聚合器内的批次分散。因此，冻结和合并的开销会分散到多个线程组中，从而提升性能。请注意，在此工作负载下无法进行消除。

对于更新率为 100% 的工作负载（图4最左侧的图），两个到四个聚合器较为理想，因为在这种情况下，线程既能将竞争分散到多个聚合器中，又能利用合并和消除。另一方面，使用五个聚合器会过度分散线程间的竞争，从而降低线程利用消除和合并的机会。对于一个聚合器，情况则恰恰相反：由于线程竞争而产生的冻结和合并开销会集中在单个聚合器上。对于更新率为 50% 和 10% 的工作负载（图4中间的图），一个或两个聚合器最为合适，既能让线程分散竞争，又能利用合并和消除。

我们选择使用两个聚合器来运行所有工作负载，因为事实证明，在大多数情况下，这是最佳设置。

7 结论

我们介绍了 SEC（分片消除与合并），这是一种线性化并发栈，通过嵌套分片将消除与合并统一起来。SEC 将线程划分为聚合器，再进一步划分为批次，从而减少共享栈上的竞争，并支持多个组合器并行执行，有别于传统的扁平合并方法。SEC 高效运行的关键在于轻量地使用取值并递增计数器，这简化了消除与合并，并在常见情况下免去了代价高昂的同步操作。与 EB、CCSynch 和扁平合并等先前算法相比，这种方法的开销较低，同时保持了良好的性能。

我们的评估表明，SEC 在各种工作负载和机器上始终优于最先进的栈。除了栈之外，新颖的分片消除和高效的合并也具有独立的研究价值，并可应用于其他并发数据结构，例如双端队列。

致谢

本研究由希腊教育、宗教事务与体育部的 SUB 1.1 计划——研究卓越合作项目（项目：HARSH，代码：YΠ 3 TA-0560901）资助，并通过希腊国家复苏与韧性计划 Greece 2.0 实施，资金来自欧盟的 NextGenerationEU。作者感谢匿名审稿人提出的宝贵意见。

参考文献

[1] Daniel Anderson、Guy E Blelloch 和 Yuanhao Wei。2021。具有常数时间开销的并发延迟引用计数。载于第42届ACM SIGPLAN编程语言设计与实现国际会议论文集。526–541。[2] Trevor Alexander Brown。2015。为无锁数据结构回收内存：一定有更好的办法。载于2015年ACM分布式计算原理研讨会论文集。261–270。[3] Mike Dodds、Andreas Haas 和 Christoph M. Kirsch。2015。一种可扩展且正确的时间戳栈。载于第42届ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（印度孟买）（POPL '15）。计算机协会，纽约州纽约，美国，233–246。doi:10.1145/2676726.2676963[4] Mike Dodds、Andreas Haas 和 Christoph M Kirsch。2015。一种可扩展且正确的时间戳栈。ACM SIGPLAN Notices 50，1（2015），233–246。[5] Panagiota Fatourou 和 Nikolaos D. Kallimanis。2011。一种高效的等待自由通用构造。载于第二十三届ACM算法与体系结构并行性研讨会论文集（美国加利福尼亚州圣何塞）（SPAA '11）。计算机协会，纽约州纽约，美国，325–334。https: //doi.org/10 1145 1989493 1989549。/。Panagiota Fatourou 和 [6] Nikolaos D Kallimanis。2012。重新审视结合式同步技术。载于第17届ACM SIGPLAN并行编程原理与实践研讨会论文集。257–266。[7] Panagiota Fatourou 和 Nikolaos D Kallimanis。2014。高效的等待自由同步。《计算系统理论》55，3（2014），475–520。[8] Panagiota Fatourou 和 Nikolaos D. Kallimanis。2017。锁振荡：提升并发数据结构的性能。载于第21届分布式系统原理国际会议 OPODIS 2017，葡萄牙里斯本，2017年12月18日至20日（LIPIcs，第95卷），James Aspnes、Alysson Bessani、Pascal Felber 和 João Leitão（编）。达格施图尔城堡—莱布尼茨信息学中心，8:1–8:17。doi:10.4230/ LIPICS.OPODIS.2017.8[9] Panagiota Fatourou、Nikolaos D. Kallimanis 和 Eleftherios Kosmas。2022。持久化中软件合并的性能优势。载于第27届ACM SIGPLAN并行编程原理与实践研讨会论文集（韩国首尔）（PPoPP '22）。计算机协会，纽约州纽约，美国，337–352。

doi:10.1145/3503221.3508426
[10] Eric Freudenthal 和 Allan Gottlieb。1991。使用取值并递增进行进程协调。ACM SIGOPS《操作系统评论》25，特刊（1991），260–268。[11] Danny Hendler、Itai Incze、Nir Shavit 和 Moran Tzafrir。2010。扁平组合与同步和并行性的权衡。载于第二十二届ACM算法与体系结构并行性研讨会论文集。355–364。[12] Danny Hendler、Nir Shavit 和 Lena Yerushalmi。2004。一种可扩展的无锁栈算法。载于第十六届ACM算法与体系结构并行性研讨会论文集。206–215。[13] Maurice Herlihy、Nir Shavit、Victor Luchangco 和 Michael Spear。2020。《多处理器编程艺术》。Newnes。[14] Daewoo Kim、Trevor Brown 和 Ajay Singh。2024。你的纪元是否过于漫长？批量释放可能有害。载于第29届ACM SIGPLAN并行编程原理与实践年会论文集。30–41。[15] David Klaftenegger、Konstantinos Sagonas 和 Kjell Winblad。2018。队列委派锁。IEEE 并行与分布式系统汇刊 29，3（2018），687–704。doi:10.1109/TPDS.2017.2767046[16] Adam Morrison 和 Yehuda Afek。2013。面向 x86 处理器的快速并发队列。载于第 18 届ACM SIGPLAN并行编程原理与实践研讨会论文集。103–112。[17] Donald Nguyen、Andrew Lenharth 和 Keshav Pingali。2013。一种轻量级图分析基础设施。载于第二十四届ACM操作系统原理研讨会论文集。456–471。[18] Ruslan Nikolaev 和 Binoy Ravindran。2024。一系列快速且节省内存的锁式与无等待内存回收方法。ACM《编程语言论文集》8，编程语言设计与实现会议（2024），2174–2198。[19] Yoshihiro Oyama、Kenjiro Taura 和 Akinori Yonezawa。1999。高效执行具有同步瓶颈的并行程序。载于并行与分布式符号及非规则应用国际研讨会论文集（PDSIA '99）。182–204。[20] Yaqiong Peng 和 Zhiyu Hao。2018。FA-Stack：一种具有等待自由进展保证的快速基于数组的栈。IEEE 并行与分布式系统汇刊 29，4（2018），843–857。doi:10.1109/TPDS.2017.2770121[21] Younghun Roh、Yuanhao Wei、Eric Ruppert、Panagiota Fatourou、Siddhartha Jayanti 和 Julian Shun。2025。用于加速取值并递增和队列操作的聚合漏斗。载于第30届ACM SIGPLAN并行编程原理与实践年会论文集。99–114。[22] Raed Romanov 和 Nikita Koval。2023。最先进的LCRQ并发队列算法不需要CAS2。载于第28届ACM SIGPLAN并行编程原理与实践年会论文集（加拿大蒙特利尔，魁北克省）（PPoPP '23）。计算机协会，纽约州纽约，美国，14–26。doi:10.1145/3572848.3577485[23] Nir Shavit 和 Asaph Zemach。2000。组合漏斗。《并行与分布式计算杂志》60，11（2000年11月），1355–1387。doi:10.1006/jpdc.2000. 1621[24] Gali Sheffi、Maurice Herlihy 和 Erez Petrank。2021。VBR：基于版本的内存回收。载于第33届ACM算法与体系结构并行性研讨会论文集（线上活动，美国）（SPAA '21）。计算机协会，纽约州纽约，美国，443–445。doi:10.1145/3409964.3461817[25] Ajay Singh 和 Trevor Brown。2025。Ping上发布：在并发数据结构的内存回收中发布预留的更佳方式。载于第30届ACM SIGPLAN并行编程原理与实践年会论文集。128–141。[26] Ajay Singh、Trevor Brown 和 Ali Mashtizadeh。2021。Nbr：基于中和的内存回收。载于第26届ACM SIGPLAN并行编程原理与实践研讨会论文集。175–190。
[27] Ajay Singh、Trevor Alexander Brown 和 Ali José Mashtizadeh。2023 年。通过中和实现简单、快速且广泛适用的并发内存回收。*IEEE* 并行与分布式系统汇刊 35，2（2023），203–220。[28] R Kent Treiber 等。1986 年。系统编程：应对并行性。国际商业机器公司，Thomas J. Watson 研究中心……[29] Albert Mingkun Yang 和 Tobias Wrigstad。2022 年。深入探究 ZGC：OpenJDK 中的现代垃圾收集器。*ACM* 编程语言与系统汇刊 *(TOPLAS)* 44，4（2022），1–34。

A 工件说明

本节将分步指导如何在 Docker 容器中运行我们的制品。该制品扩展了广受欢迎的 Brown 的 setbench 基准测试，公开提供于：https://gitlab.com/trbot86/setbench。

工件可通过以下链接获取：

- Zenodo（最新版本 v4，含 Docker 镜像）：
  https://zenodo.org/records/18109078。
- GitHub（仓库不含 Docker 镜像）：
  https://github.com/ConcurrentDistributedLab/combXAgg

如果您希望直接使用该制品，而不使用 Docker 容器，请参阅源代码中随附的 README 文件。

以下说明将帮助您加载并运行从 Zenodo 链接下载的制品中提供的 Docker 镜像。Docker 容器启动后，您可以按照随附的 README 文件编译并运行基准测试中的实验。

加载并运行所提供 Docker 镜像的步骤：

注意：Su do 权限可能 be 需要 d 以执行以 h下说明。

1. 在系统上安装最新版本的 Docker。我们使用 Ubuntu 24.04 上的 Docker 版本 28.3.0（构建版本 38b7060）测试了该工件。Docker 安装说明请参阅 https://docs.docker.com/engine/install/ubuntu/。或者，也可以参阅本 README 末尾的“安装 Docker”部分。要检查计算机上 Docker 的版本，请使用：`$ docker -v`2. 从 Zenodo 下载该工件，网址：https://zenodo.org/records/18109078。3. 解压下载的文件夹，并使用*sec_setbench/* 命令进入该目录。 *cd* 4. 找到名为*sec_setbench.tar.gz*的 Docker 镜像文件，它位于 sec_setbench/ 目录中。使用以下命令加载下载的 Docker 镜像：`$sudodocker load < sec_setbench.tar.gz`5. 验证镜像是否已加载：`$sudodocker images`6. 使用已加载的镜像启动 Docker 容器：`$ sudo docker run -it --rmsec_setbench`7. 调用 *ls* 以查看该工件中的若干文件/文件夹：Dockerfile、README.md、common、ds、install.sh、lib、microbench、sec_experiments、tools。现在，要编译并运行实验，可以按照 README 文件中的说明进行操作。

B 正确性与进展

我们首先陈述根据伪代码得出的一系列观察结论，随后证明 SEC 是线性化的。现在考虑一个属于聚合器 A 的批次 B 中的操作 op，并令 q 表示发起 op 的线程。

观察 B.1。 A 的每个批次都恰好有一个冻结器。

这可直接从伪代码得出。最多只有两个线程会竞争成为 冻结器：一个正在执行 压入，另一个正在执行 弹出。这两个操作的序列号都必须为 0，也就是说，它们分别是该批次中各自类型的第一个操作。只有一个操作能成功执行其 测试并设置 (第 8 行和第 57 行)，并随后调用 `freezebatch`。因此，每个批次恰好有一个冻结器。

观察 B.2。 组合器会对栈应用全部压入操作，或全部弹出操作。

这是由线程从左到右在批次的 消除数组 上声明操作的方式决定的。可能出现三种情况：

1. 一个批次中的入栈操作多于弹出操作。在这种情况下，消除后只剩下入栈操作。执行第一个未被消除的入栈操作的线程将成为组合器。

2. 一个批次中的弹出操作多于入栈操作。在这种情况下，消除后只剩下弹出操作。执行第一个未被消除的弹出操作的线程将成为组合器。
3. 一个批次包含数量相等的入栈操作和弹出操作。在这种情况下，消除后该批次为空，不再有操作需要执行。具体而言，在压入操作中，线程在第 15 行检查处失败；在弹出操作中，线程在第 64行检查处失败。因此，不会调用组合器。

观察 B.3。 A 的每个批次恰好有一个组合器。

线程按冻结、消除和合并的顺序执行这些操作。仅当消除后批次内仍有操作时，才需要进行合并。这些剩余操作必须全部是压入操作或全部是弹出操作。

不失一般性，考虑一个仅包含压入操作的批次。合并步骤必须由执行该批次中第一个未被消除的压入操作的线程完成。假设使用$fetch\&increment$递增的 *pushCount* 变量不会发生回绕。在一个批次内， $fetch\&increment$保证不会有两个线程获得相同的序列号。在冻结期间，*pushCountAtFreeze* 会被赋予一个唯一的序列号。

因此，在第 16 行的组合测试中，只有一个线程的序列号能够与批次的 冻结时弹出计数匹配。因此，只有一个线程能够进入算法 1 第 16 行的组合代码块。

对于 弹出操作，论证是对称的：只有一个线程的序列号能够与批次的 冻结时推入计数匹配，因此只有一个线程能够进入算法 2 第 69 行的组合代码块。

引理 B.4. 每个未消除操作的线性化点都位于其执行区间内 *l.*

证明。 设 op 是一个未消除操作。则存在一个合并线程 c_B，为批次 B 应用 op。（组合器c_B 可能是发起该操作的线程，也可能是在 B 中执行弹出操作的另一个线程 q。）

如果 op 是一个 压入操作，我们将其线性化于 c_B 在 PushToStack 的第 47 行成功执行 比较并交换之处。类似地，如果 op 是一个 弹出操作，我们将其线性化于 c_B 在 PopFromStack 的第 90 行成功执行 比较并交换之处。注意， c_B 可以通过一次 比较并交换压入或弹出多个操作。所有此类操作都按照其序列号的顺序进行线性化，序列号较小的操作先进行线性化。□

引理 B.5. 已消除的弹出操作的线性化点紧接在与其对应的已消除压入操作的线性化点之前，且这两个线性化点都位于弹出的执行区间内。

证明。 考虑一对已消除的压入和弹出操作。这两个操作都在线 67处线性化，此时弹出会在其执行区间内与对应的压入交换值。没有其他操作能够在线性化顺序中插入这两个已消除的操作之间。□

不变式 B.6. 在任何时刻 t，*stackTop* 都表示按照线性化点的先后顺序依次执行所有在 t之前线性化的操作后，栈所处的状态。

我们通过归纳证明该不变式 在共享栈上执行操作所进行的一系列线性化步骤上成立 k.

归纳基： 初始时，栈为空，也就是说没有任何操作修改过它。如果存在一系列已消除的操作，它们必定彼此抵消，因此净效果是栈仍为空。所以该不变式显然成立。

归纳假设：假设在某一步骤 i之后，不变量仍然成立，该步骤由某个操作 op 引起。**归纳：**
考虑该操作（$i+1$）$_{(}th$）。

- 如果这是一个被消除的操作，它不会修改栈，并且会与 i_{th} 操作相互抵消，使栈保持不变。此外，操作 i 和 $i+1$之间不可能存在其他操作。
- 如果这是一个组合器执行比较并交换以添加未消除的入栈操作，那么组合器会按照批次中操作的顺序，将相应的节点依次添加到栈顶。
- 如果这是一个组合器执行比较并交换以执行 k 弹出操作，那么组合器会按照批次中弹出操作的顺序，依次从栈顶移除 k 个节点。

在所有情况下，步骤 $i+1$ 期间执行的操作集合都能保持不变量。因此，根据归纳法，
不变量对所有步骤均成立。

引理 B.7。 每个操作的响应都与其线性化点一致。

只有查看栈顶和弹出操作会产生响应，因此我们只考虑这两种情况。读取栈顶 时，查看操作即被线性化。它们的响应是共享栈顶端节点的值。

被消除的压入或弹出操作会一同线性化：弹出操作紧接着与之匹配的压入操作线性化，两者之间不会有其他操作线性化。弹出操作会返回与之匹配的压入操作所交换的值。在这两种情况下，该论断都显然成立。

接下来，考虑一批 弹出 操作， pop_1， pop_2，$\cdots$， pop_k，这些操作通过对 栈顶 执行成功的比较并交换而按顺序线性化。执行 pop_1 的线程就是组合器，它执行比较并交换，从共享栈中移除顶部的 k 个节点。 pop_i 的响应值是从栈中移除的 i_{th} 节点的值，其中 $1 > i \leq k$（参见 `GetValue()`）。根据不变量 B.6，这些响应遵循线性化顺序。

由不变量 B.6 和引理 B.7 可得以下定理。

定理 B.8。 SEC是一个线性化栈实现。

C 在56线程机器 Emerald 上的额外实验

在表1中，我们研究了SEC的批处理、消除和合并度。不同列显示了不同更新率下的测量结果。可以看到，批处理程度和消除度随着更新率的提高而增加，而合并度保持不变。例如，在更新率为 100% 的情况下，批次的平均大小（不同线程数下）为 41，是所有设置中最大的；其中批次内 78% 的操作被消除。因此，SEC 的主要性能优势来自高效的批处理和消除。

	SEC		
工作负载 →	**100% 更新**	**50% 更新**	**10% 更新**
批处理程度	17.8	17.2	14
%消除	79%	79%	77%
%合并	21%	21%	23%

表1. SEC 中 EXP1 在穿梭机上的批处理程度（一次执行期间批次的平均大小）、消除百分比（每个批次中被消除的操作所占百分比）和合并百分比（每个批次中未被消除的操作所占百分比）。列标签 100 表示工作负载包含 100% 更新，依此类推。

D 96线程机器上的实验：IceLake

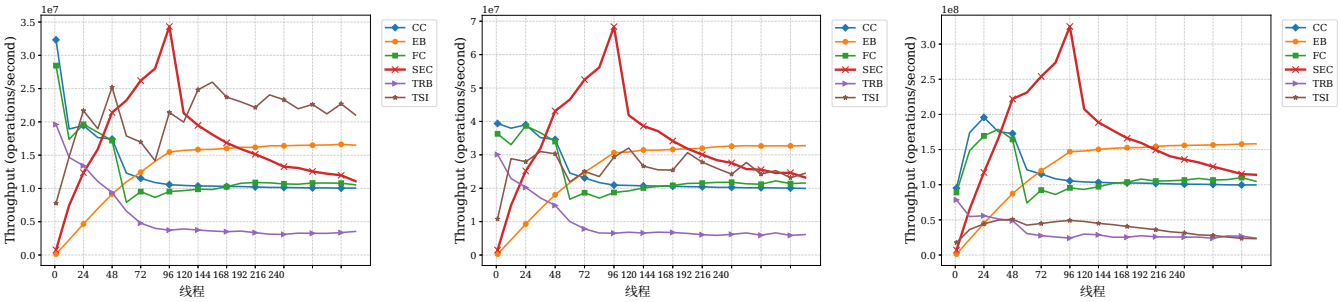


图 5. 不同线程数下的吞吐量。（左）100% 更新。（中）50% 更新。（右）10% 更新。纵轴：每秒操作数（百万）。横轴：线程数。

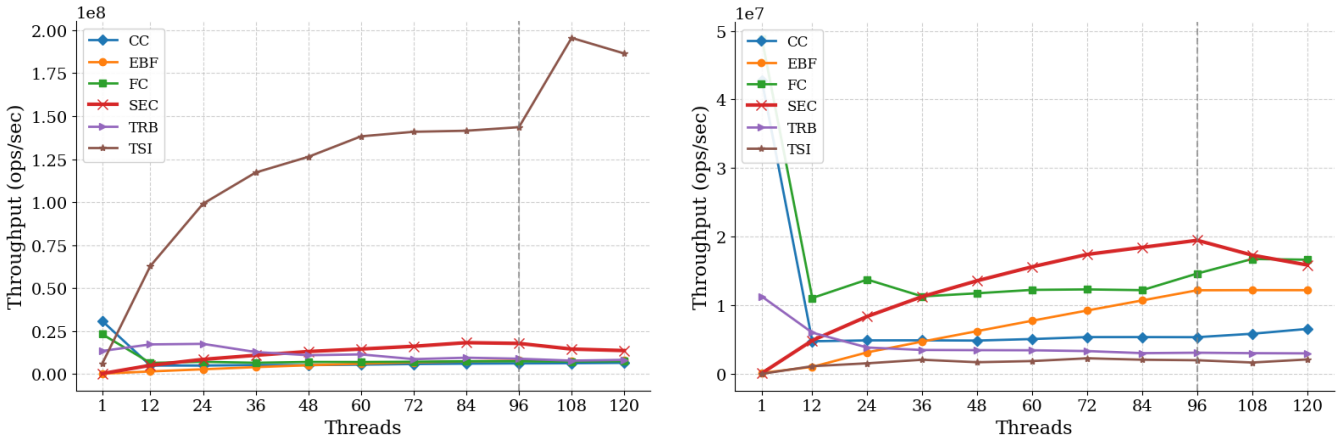


图 6. 仅推送和仅弹出工作负载的吞吐量。（左）仅推送。（右）仅弹出。纵轴：每秒操作数（百万）。横轴：线程数。使用的聚合器数量为两个。

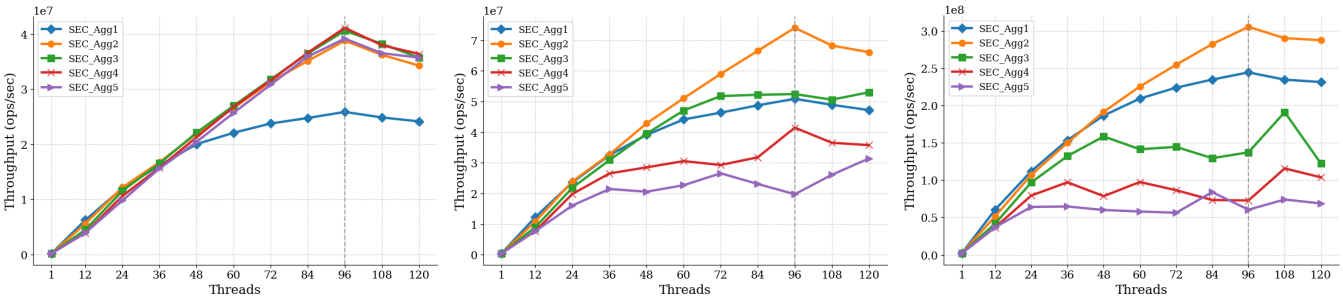


图 7. 使用聚合器进行自身比较：不同线程数下的吞吐量。（左）100% 更新。（中）50% 更新。（右）10% 更新。纵轴：每秒操作数（百万）。横轴：线程数。

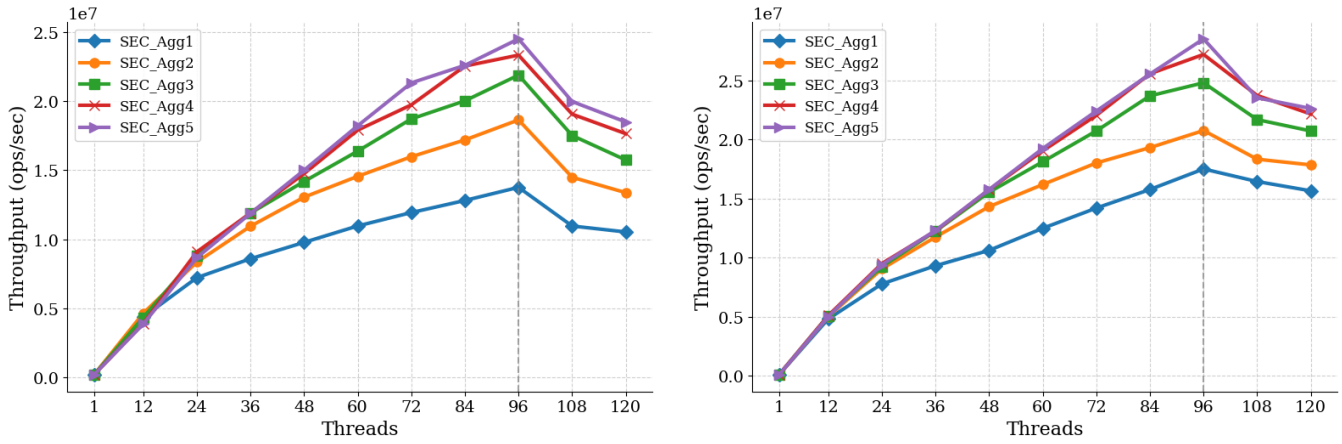


图8。使用聚合器进行仅推送和仅弹出工作负载的自身比较。不同线程数下的吞吐量。（左）仅推送。（右）仅弹出。Y 轴：每秒数百万次操作的吞吐量。X 轴：线程数。

	SEC		
工作负载 →	**100% 更新**	**50% 更新**	**10% 更新**
批处理程度	40	31	28
%消除	85%	85%	84%
%合并	15%	15%	16%

表 2. 批处理程度（一次执行期间批次的平均大小）、SEC 中图 5 所示实验的消除百分比（每批次消除的操作所占百分比）和合并百分比（每批次未被消除的操作所占百分比）。列标签 100 表示工作负载中 100% 为更新操作，其他情况依此类推。

E 在192线程机器上的实验：Sapphire

我们还在一台英特尔 Sapphire Rapids（**Sapphire**）上进行了实验。该机器有8个NUMA节点，支持双路超线程，每个NUMA节点上运行24个线程，因此总共支持192个硬件线程。该机器的配置如下：210 MiB三级缓存、3.5 GHz频率、380 GB内存。

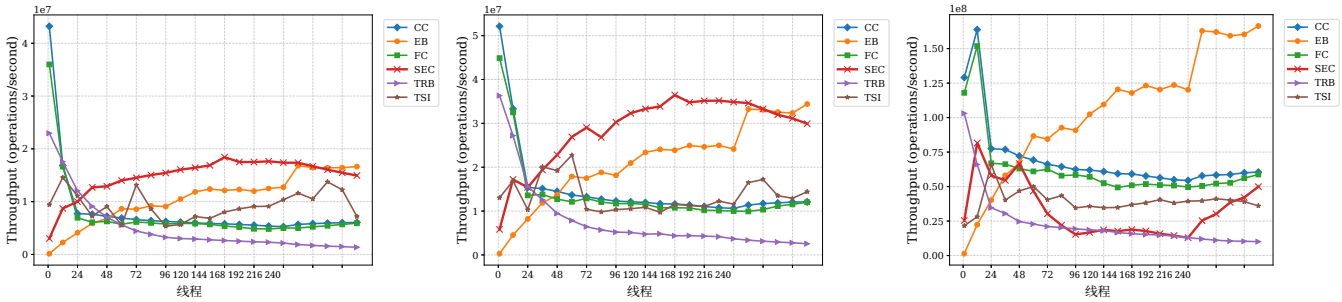


图9. 吞吐量。（左）100% 更新。（中）50% 更新。（右）10%更新。纵轴：每秒百万次操作的吞吐量。横轴：线程数。使用的聚合器数量为两个。

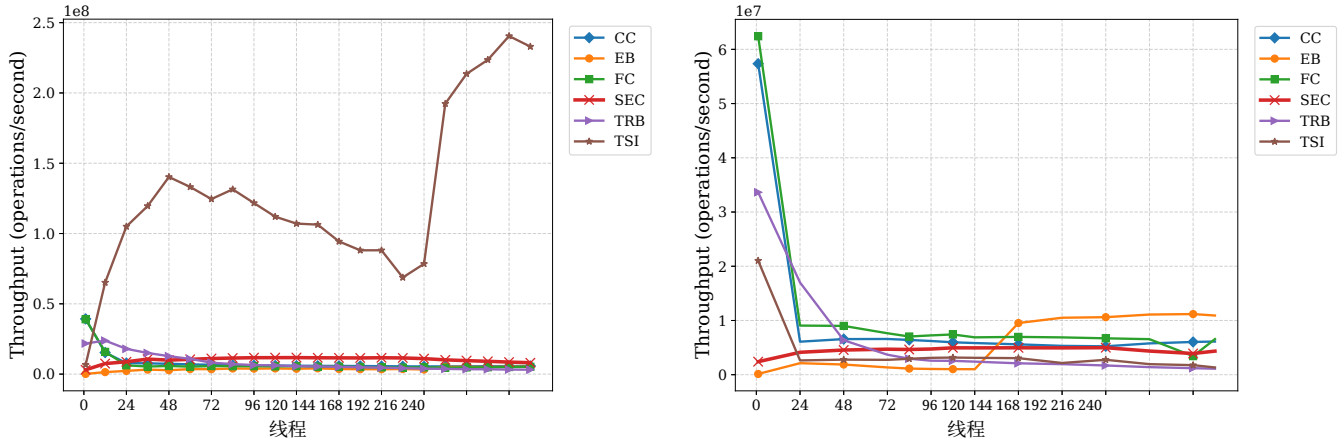


图 10. 仅推送和仅弹出工作负载的吞吐量。（左）仅推送。（右）仅弹出。纵轴：吞吐量，单位为每秒百万次操作。横轴：线程数。使用的聚合器数量为 2。

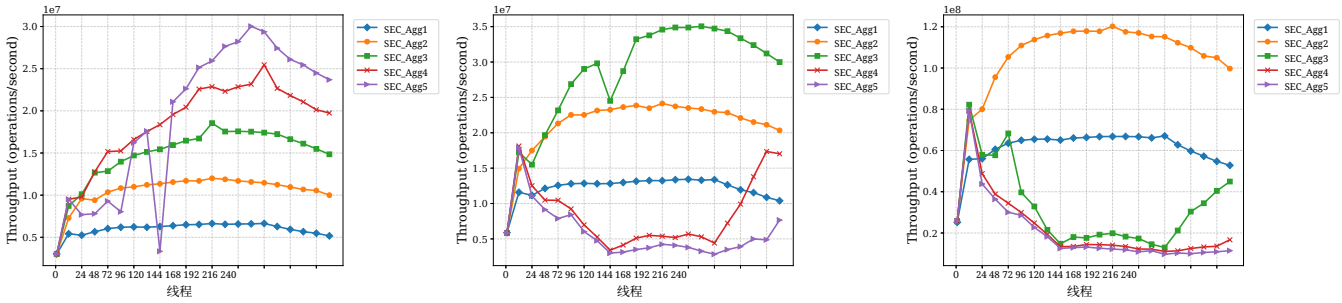


图 11. 比较 DCE+ 在不同聚合器数量下的吞吐量。从左到右依次为：100% 更新、50% 更新、10% 更新、100% 仅推送。纵轴：吞吐量。横轴：线程数。使用 1 个聚合器的 DCE+ 标记为 DCE+_Agg1。

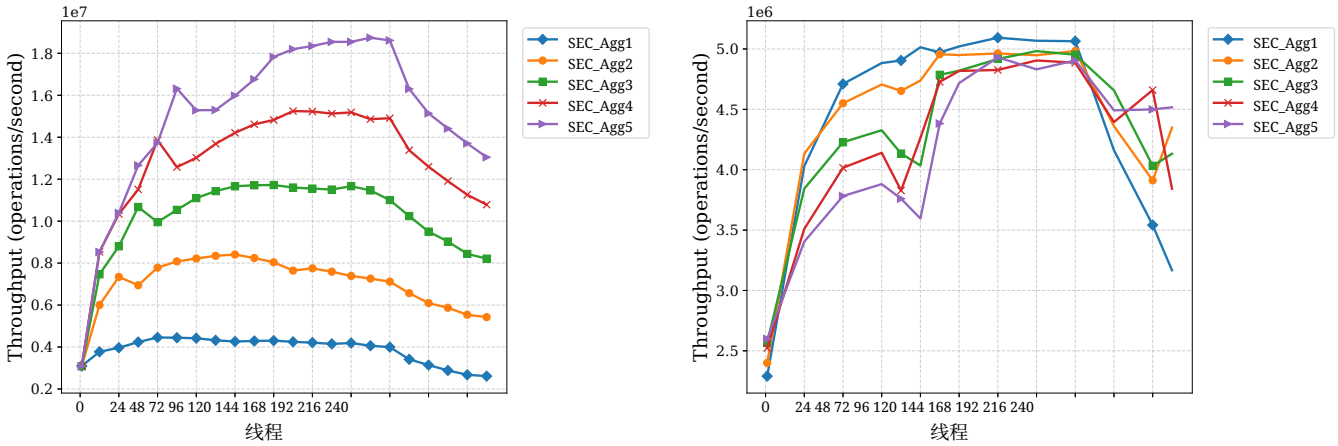


图 12. 仅推送和仅弹出工作负载下与聚合器的自身比较。不同线程数下的吞吐量。（左）仅推送。（右）仅弹出。纵轴：每秒操作数（百万）。横轴：线程数。

	SEC		
工作负载 →	**100% 更新**	**50%更新**	**10%更新**
批处理程度	24	22	17
%消除	77%	75%	70%
%合并	23%	25%	30%

表3. EXP1 的 SEC 中的批处理程度（执行期间批次的平均大小）、消除百分比（每批次中被消除的操作百分比）和合并百分比（每批次中未被消除的操作百分比）。列标签 100 表示工作负载包含 100% 更新，以此类推。