预测短数据竞争的高效动态算法

Minjian Zhang ✉伊利诺伊大学厄巴纳-香槟分校，美国

Mahesh Viswanathan ✉伊利诺伊大学厄巴纳-香槟分校，美国

摘要

我们提出并研究了在观测轨迹中检测短竞争的问题。具体而言，对于竞态类型 R，给定轨迹 σ 和窗口大小 w，任务是确定其中是否存在 R-竞态（e_1，e_2），使得从 e_1 开始、到 e_2 结束的子轨迹至多包含 w 个事件。我们提出了一个用于短竞争预测的监控框架，并将该框架应用于先发生关系竞态和同步保持竞态，从而得到了高效的检测算法。我们的先发生关系算法运行时间与 FastTrack 相同，但其空间开销随 log w 增长，而非随 log $|\sigma|$增长。对于同步保持竞态，我们的算法运行速度更快，且比 SyncP 显著节省空间。实验验证了这些短竞争检测算法的有效性：它们运行效率更高、占用内存更少，并且在相同预算下能检测出显著更多的数据竞争，在资源使用和预测能力之间取得了合理平衡。

2012 ACM 学科分类软件及其工程 → 动态分析

关键词和短语并发、预测、数据竞争、先发生关系

开发可靠的并发程序是一项复杂的任务，因此形式化推理原则对于分析这类程序至关重要。设计和验证并发系统之所以困难，源于程序行为固有的非确定性，而这种非确定性是由进程间通信和调度造成的。在开发过程中管理这种非确定性颇具挑战，往往会导致容易出错的软件。此外，并发相关缺陷以难以手动复现而闻名，这凸显了采用自动检测技术提高软件开发效率的必要性。

当多线程程序中的一个线程访问某个共享内存位置，而另一个线程在没有适当同步的情况下修改该位置时，就会发生数据竞争（或简称竞态）。数据竞争往往表明程序存在严重缺陷 [27] ，并且与数据损坏、编译失败 [33, 8, 21], 以及严重的系统故障 [48, 9]有关。因此，大量研究致力于检测和预防多线程程序中的数据竞争。

尽管已有用于数据竞争检测的静态技术，但由于动态竞态检测方法具有更强的可扩展性和精确性，因此应用更为广泛。动态数据竞争检测的目标是分析观测到的程序轨迹，判断其中是否包含生成该轨迹的程序发生数据竞争的证据。这种方法的一个关键目标是具备预测能力——不仅识别观测轨迹中的数据竞争，还要发现程序可能出现的备选执行中的竞态。这种预测能力提升了动态数据竞争检测的实际效用。然而，预测性要求也使该问题在计算上颇具挑战性——动态数据竞争检测是 NP 完全且 W[1]-难的，因此属于固定参数难解问题 [30, 23]。

难以求解这一挑战推动了针对特定类型的预测性数据竞争的研究 [24, 41, 44, 22, 5, 29, 36, 34, 37, 31, 43]。这些方法旨在兼顾预测能力与计算上的可处理性。竞争定义通常依赖于偏序 [24, 44, 22, 29] ，或对所考虑的备选执行类型施加约束 [41, 34, 31, 43]。

即使某种竞态类型是可处理的，也就是说存在多项式时间算法，实际

arXiv:2603.03141v1 [cs.PL] 3 Mar 2026

开销仍可能高得难以承受，从而限制可分析的执行轨迹规模。即使是采用单遍线性时间流式处理的竞争检测算法，下界也表明，它们所需的内存与轨迹大小成正比 [22, 31], ，而这可能相当庞大。为了解决这一问题，窗口化 [44, 41] 被提出作为一种降低动态数据竞争检测开销的技术。这种方法将轨迹划分为长度固定为 w（例如）的子轨迹，算法仅在每个子轨迹内搜索数据竞争。通过将分析限制在有界长度的片段中，窗口化有助于控制计算开销，从而使分析更长的轨迹成为可能。窗口化对预测分析施加了一项关键限制。要确认预测性竞争，通常需要考察两个冲突事件之外的上下文，以及它们之间的事件。如果上下文中的某些关键事件位于窗口之外，那么即使两个竞争事件本身在窗口内，任何算法也无法可靠地确认该竞态。因此，我们关注一个比窗口化更一般的问题。我们不再将数据竞争检测限制在预先定义的子轨迹内，而是研究轨迹中是否存在短距离数据竞争——也就是相关事件之间的距离有界的数据竞争。我们的方法为窗口添加了一个常数空间摘要，用以精确捕获确认预测性竞争所需的窗口外上下文。

除了降低计算开销之外，采用窗口化以及我们检测短程竞争的方法，另一个关键动机在于：在实践中，存在数据竞争的程序经常会表现出短程竞争。此外，高效检测此类短程竞争的算法也能提升随机化竞态检测方法的效果 [45]。我们提出了一个用于在流式轨迹上检测短程竞争的监控框架：随着轨迹在线处理，监控器会维护一个固定大小的窗口以及相关历史的紧凑摘要，并在窗口向前滑动时增量更新它们。随后，我们针对两种广泛使用的竞态概念——先发生关系和同步保持竞态——实例化该框架。先发生竞争 [24] 是历史最悠久、应用最广泛的数据竞态定义。它基于先发生偏序定义，也是 ThreadSanitizer 等工业工具的基础 [42]。同步保持竞态 [31] 是先发生关系的一种泛化，具有更强的预测能力。这两种竞赛类型都可以使用线性时间、单遍算法检测，不过，任何用于检测同步保持竞态的线性时间算法都必须使用线性空间 [31]。我们探讨了这些竞争定义能否以更低开销、更高效地检测短程竞争。我们的结果总结于表1。

R-竞态	竞态检测	短竞争
先发生关系	时间： $O(NT)$ 空间： $O((T+V+L)T \log N)$	时间： $O(NT)$ 空间： $O(w+(\min(w, V+L)+T)T \log w)$
同步保持	时间： $O(NT^2+AVT^3)$ 空间： $O(VLT^3+NT \log N)$	时间： $O(NT(\min(w, T))+AT(\min(w, VT^2)))$ 空间： $O(\min(w,VT)LT^2+(w+V+L)T \log N)$

表1 基于以下参数的结果概述。N 是轨迹的大小，w 是短程竞争的长度，T 是线程数，V 是变量数，L 是锁数，A 是轨迹中的获取事件数。

我们的算法通过维护一个包含最近 w 个事件的滑动窗口，并检查窗口内是否存在数据竞争，来检测长度为 w 的短先发生竞争。这一策略的正确性取决于这样一个观察：一对事件 (e_1, e_2) 是否按先发生关系排序，仅取决于它们之间的事件。我们的方法基于 FastTrack [18],（目前已知最高效的先发生检测器），但对其进行了修改，以避免朴素窗口化的局限。仅将搜索范围限制在最近 w 个事件，同时保持标准的向量时钟语义，并不能改善渐近空间复杂度，因为时间戳仍随完整轨迹长度 N 增长（例如，每线程执行锁操作的最大次数），

因此产生 log N 依赖。相比之下，我们的算法通过将时间重新锚定到窗口，并压缩元数据，消除了这种依赖，使时钟分量的数量及其数值大小都受 w 限制（从而得到 log w 而非 log N）。该算法的运行时间与 FastTrack 相当，但占用的空间更少：FastTrack 的向量时钟分量可能随轨迹长度增长，而我们的分量通过精细的记账方法和新颖的数据结构被限制在 w 以内。尽管这些结果在理论上意义重大，但考虑到 FastTrack 已经具备出色的可扩展性，其实际影响可能有限。

检测短同步保持竞争还面临着必须克服的额外挑战。与先发生于竞争不同，判断一对事件（e_1，e_2）是否构成同步保持竞争，不仅取决于 e_1 和 e_2 之间的事件，还取决于整个轨迹中此前的内存访问和同步事件。这正是任何单遍流式算法检测同步保持竞争的问题都存在一个线性空间下界的原因。因此，仅使用包含最近 w 个事件的简单滑动窗口，无法实现可靠的检测算法。我们的关键洞见是，除了跟踪滑动窗口本身，还可以维护当前窗口之外一组选定事件的元数据，从而保证可靠性。这一发现具有更广泛的意义，表明有望针对其他竞争定义，开发高效的短程竞争检测算法。通过精心设计数据结构，我们开发出一种算法，能够可靠地检测短同步保持竞争，同时以渐近更快的速度运行并显著减少空间用量（见表1）。

我们的实验结果凸显了短程竞争检测算法的潜力。虽然对于先发生竞争而言，其开销降低幅度不如同步保持竞态那么显著，但实验表明，在许多情况下，检测短程竞争能够缩短运行时间。此外，即使窗口较小，FastTrack 检测到的许多竞态仍能被报告出来，这彰显了这些算法强大的预测能力。我们认为，短程竞争检测算法很可能对同步保持竞态尤为重要，而实验结果也印证了这一点。实验表明，将竞争长度限制在较短范围内时，我们的算法几乎在所有方面都能带来显著改进——包括吞吐量、竞态预测能力、运行时间和内存使用量——从而大幅提升可扩展性，同时保持强大的预测能力。

本文其余部分安排如下。第1节介绍基本符号、竞争定义以及本文要解决的问题。第2节介绍针对该问题的监控框架。第3节讨论先发生竞争的具体实现，第4节讨论同步保持竞态的具体实现。第5节介绍实验，最后在第7节作总结。

1 预备知识

1.1 程序执行轨迹

并发程序中的事件 e 表示为一个元组 (t, o)，其中 $\text{thr}(e) = t$ 表示负责该事件的线程标识符，而 $\text{op}(e) = o$ 表示操作类型。在本研究中，我们将操作类型 o 限定为以下几种：(a) 对内存位置 x进行读写访问（即 $o = \texttt{r}(x)$ 或 $o = \texttt{w}(x)$）；或 (b) 获取/释放锁 ℓ（即 $o = \texttt{acq}(\ell)$ 或$o = \texttt{rel}(\ell)$）。轨迹是由事件组成的序列。我们用 E_σ 表示轨迹σ 中的事件集合，用 E_σ^t 表示由特定线程 t 执行的事件。我们约定序列中的第一个事件的索引为 0。对于长度为 n 的轨迹 $\sigma = e_0e_1e_2....e_{n-1}$，第 i 个事件 e_i 也记作 $\sigma[i]$，其索引记作 $\text{index}_\sigma(e_i) = i$。对于事件 e，如果存在索引 i 使得 $\sigma[i] = e$，我们就称 $e \in \sigma$。子轨迹$\sigma[i, j] = e_ie_{i+1}\cdots e_j$ 是 σ中从索引 i 到索引 j、长度为 $j-i+1$ 的子序列。当

$j \le i$，我们约定 $\sigma[i,j]$ 为空序列 ε。

对于每个读事件 $e = \mathrm{r}(x)$，令 $\mathrm{lw}_\sigma(e)$ 表示在 σ 中发生于 e 之前、位于 x 上的最后一次写入事件。若不存在这样的事件，则 $\mathrm{lw}_\sigma(e) = \perp$。对于每个获取事件和释放事件，令 $\mathrm{match}_\sigma(e)$ 表示与之配对的释放 / 获取事件。类似地，若不存在这样的事件，则 $\mathrm{match}_\sigma(e) = \perp$。

如果轨迹 w 符合锁语义，则称其为良构轨迹；也就是说，没有线程会获取由另一线程持有的锁，也不会释放其未持有的锁。如果子轨迹 w' 是某个良构轨迹的子轨迹，则称其为良构子轨迹。根据定义，每个良构轨迹都是良构子轨迹。如果子轨迹中每个释放事件所对应的获取事件都存在，则它是良构轨迹。

对于子轨迹 σ 中的一对事件 e_1，e_2，若 $\mathrm{index}_\sigma(e_1) \le \mathrm{index}_\sigma(e_2)$，即 e_1 在 σ 中先于 e_2 发生，则记作 $e_1 \le^\sigma_{\mathsf{tr}} e_2$。如果 $\mathrm{thr}(e_1) = \mathrm{thr}(e_2)$，则记作 $e_1 \le^\sigma_{\mathsf{TO}} e_2$。相应的非自反序分别为 $<^\sigma_{\mathsf{tr}}$ 和 $<^\sigma_{\mathsf{TO}}$。若两个内存访问事件e_1，e_2 访问同一内存位置（设为）x，且 e_1，e_2 中至少一个是写事件，并且 $\mathrm{thr}(e_1) \neq \mathrm{thr}(e_2)$，则称它们冲突。

1.2 数据竞争

当一个线程在没有适当同步的情况下访问共享内存位置，而另一个线程正在修改其内容时，就会发生数据竞争。在预测性竞争检测中，目标是判断一条观测轨迹是否见证了该轨迹本身或其某种备选“重排”中的数据竞争。这依赖于识别产生该观测轨迹的程序所具有的备选轨迹。因此，要定义一条轨迹是否具有预测性竞争，我们需要正确重排序 这一概念，它刻画了产生观测行为的程序必须展现的一组轨迹。

正确重排 对于给定的良构轨迹 σ，若一条良构轨迹 σ' 满足以下条件，则称其为 σ 的正确重排：(1) $E_{\sigma'} \subseteq E_\sigma$；(2) 对于任意一对事件 e_1, e_2，若 $e_1 \le^\sigma_{\mathsf{TO}} e_2$ 且$e_2 \in E_{\sigma'}$，则 $e_1 \in E_{\sigma'}$ 且 $e_1 \le^{\sigma'}_{\mathsf{TO}} e_2$；(3) 对于每个读事件 $e \in E_{\sigma'}$，$\mathrm{lw}_\sigma(e) = \mathrm{lw}_{\sigma'}(e)$。正确排序识别出生成轨迹 σ 的程序的备选执行。σ 的正确重排 σ' 具有一致的线程序，并确保每个读事件观察到同一写事件的效果。值得注意的是，任意轨迹 σ 都是其自身的正确重排。

预测性竞态动态竞态预测问题要回答的是：给定一条轨迹，是否存在一种正确重排，能够见证一次数据竞争。

要定义轨迹中是否存在数据竞争，我们需要引入事件可启用的概念。若良构轨迹 σ 的事件 e 在 σ 的一种正确重排 σ' 中满足 $e \notin E_{\sigma'}$，且对每个 $e' \le^\sigma_{\mathsf{TO}} e$，都有 $e' \in E_{\sigma'}$，则称该事件在此重排中可启用。接着，如果存在 σ 的一种正确重排 σ'，使得冲突事件 e_1、$e_2 \in E_\sigma$ 在 σ'中均可启用，即 e_1 且e_2，则称这对事件在 σ 中构成预测性数据竞争。

众所周知，预测性竞态检测问题是 NP 完全的，且属于固定参数难解问题 [30]。因此，许多创新方法致力于识别特定类别的预测性竞态，以在效率与预测能力之间取得平衡。这些方法大致分为两类：一类基于偏序 [18, 29, 22]，另一类通过限制正确重排的类别 [31, 43, 34]。下面我们分别介绍这两类中的一个重要示例。

先发生数据竞态 轨迹 σ 的先发生偏序 $\le^\sigma_{\mathsf{HB}}$ 是 E_σ 上的最小偏序，并满足对每一对事件 (e_1, e_2)：

1. 如果 $e_1 \leq^{\sigma}_{\mathsf{TO}} e_2$，那么 $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$，并且2. 如果 $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$， $e_1 = \mathtt{rel}$（ℓ）和 $e_2 = \mathtt{acq}$（ℓ）对于某个锁 ℓ，那么 $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$。
若存在一对冲突事件e_1 与 e_2，使得既非 $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$ 也非 $e_2 \leq^{\sigma}_{\mathsf{HB}} e_1$，则称子轨迹 σ 存在先发生于竞争。

先发生竞争是可靠竞态，即如果轨迹 σ 存在先发生于竞争，则它存在预测性竞争。然而，有必要指出一个细微之处——尽管先发生竞争是可靠的，并能表明存在预测性竞争，但并非每一个先发生于竞争都是预测性竞争。先发生数据竞态是最流行的数据竞争类型之一，也是ThreadSanitizer [42]等广泛使用的工业工具的基础。它拥有一种高效的基于向量时钟的算法，这是其广受欢迎的原因之一。下面我们介绍在许多动态数据竞争检测算法中发挥重要作用的向量时钟和向量时间戳。

向量时间戳与时钟向量时间戳和向量时钟是广泛用于刻画并推理分布式或多线程环境中事件之间因果关系的基本概念。

一个向量时间戳 是从线程$_\sigma$ 到 $\mathbb{N}$ 的映射，为每个线程分配一个自然数。它通常表示为长度为 T 的向量，其中 $\mathrm{T} = |$线程$_\sigma|$ 是 σ中的线程数。在竞态检测算法中，向量时间戳通常与事件相关联；事件 e 的向量时间戳 V_e 表示相对于 $\leq^{\sigma}_{\mathsf{TO}}$ 在因果关系上先于 e的事件所构成的向下闭集；其中，每个条目 V_e（t）存储属于 V_e所表示集合的线程 t 的最后一个事件的索引。例如，向量时间戳是一种高效表示所有先于另一个事件发生的事件集合的方式 e。两个向量时间戳 V_1 和 V_2 的并 记作 $V_1 \sqcup V_2$，是对时间戳逐项取最大值， $\lambda t\mathrm{max}(V_1(t)\ V_2(t))$。向量时间戳上的逐点序定义了一个偏序。因此， $V_1 \sqsubseteq V_2$ 当且仅当 $\forall t$（V_1（t） $\leq V_2$（t））。我们用 $\perp$ 表示默认向量时间戳，其中每个条目的值均为 -1。

向量时钟是一种存储向量时间戳的变量。竞态检测算法会在观察轨迹的过程中，使用向量时钟存储重要事件的时间戳。它们在动态数据竞争检测算法中发挥着至关重要的作用。

本文将重点讨论的第二类数据竞争是同步保持数据竞争，它是先发生竞争的推广形式。通过将关注范围限制在同步保持的正确重排序上，可以得到这类竞争。

同步保持竞态 若 σ' 对 σ 的一种正确重排满足以下条件，则称其为 σ 的同步保持正确重排：对于 $E_{\sigma'}$ 中同一把锁上的任意两个获取事件，即 $e_1 = \mathtt{acq}(\ell)$ 和 $e_2 = \mathtt{acq}(\ell)$，$e_1 \leq^{\sigma'}{}_{\mathsf{tr}}\ e_2$ 当且仅当 $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$。如果一对冲突事件 e_1, e_2 在某个良构轨迹 σ 中存在一种同步保持正确排序 σ'，使得 e_1 和 e_2 均可执行，则称其为同步保持竞争。根据定义，每个同步保持竞争都是预测性竞争。

▶**命题 1** ([31])。 如果良构轨迹 σ 中存在先发生于竞争，那么 σ 中也存在同步保持竞争。

现在，我们从理想或向下闭集的角度，对同步保持竞态给出另一种刻画。这种刻画与使用向量时钟表示向下闭集相结合，构成了线性时间算法的基础，该算法用于检测同步保持竞态 [31]。

同步保持闭包 对于给定的良构轨迹 σ，一组事件 $S \subseteq E_\sigma$ 是线程序和最后写封闭或 *TL*封闭 如果

1. S 相对于 $\leq^{\sigma}_{\mathsf{TO}}$是向下封闭的。2. 对于每个读事件 $e \in S$，如果 $\mathsf{lw}_{\sigma}(e) \neq \bot$，则 $\mathsf{lw}_{\sigma}(e) \in S$。

集合 S 的 TL 闭包记作 $\mathsf{TLClosure}_{\sigma}(S)$，它是包含$S$的最小 TL 封闭集。

此外，如果满足以下条件，则称 S 为 同步保持闭合：

1. S 是 TL 封闭的，并且2. 对于任意一对获取事件 a_1，$a_2 \in S$，如果 a_1，a_2 访问同一把锁，且 $a_1 <^{\sigma}_{\mathsf{tr}} a_2$，那么 匹配$_{\sigma}(a_1) \in S$。

集合 S 的同步保持闭包记作 $\mathsf{SPClosure}_{\sigma}(S)$，它是包含 S 的最小同步保持闭合集合。

令 $\mathsf{prev}_{\sigma}(e)$ 表示 σ 中由线程 $\mathsf{thr}(e)$ 在 e之前执行的最后一个事件。那么，对于事件 $e \in E_{\sigma}$，σ 的每一种同步保持正确排序都必须包含 $\mathsf{SPClosure}_{\sigma}(\mathsf{prev}_{\sigma}(e))$ 中的所有事件，以确保 e 已启用。因此，两个冲突事件 e_1 和 e_2处于数据竞争中，当且仅当 $\mathsf{SPClosure}_{\sigma}(\mathsf{prev}_{\sigma}(e_1), \mathsf{prev}_{\sigma}(e_2))$ 不包含 e_1 也不包含 e_2 [31]；对于事件 f_1、f_2，我们用 $\mathsf{SPClosure}_{\sigma}(f_1, f_2)$ 表示集合 $\mathsf{SPClosure}_{\sigma}(\{f_1, f_2\})$。我们将用 $\mathsf{SPIdeal}_{\sigma}(e_1, e_2)$ 表示集合 $\mathsf{SPClosure}_{\sigma}(\mathsf{prev}_{\sigma}(e_1), \mathsf{prev}_{\sigma}(e_2))$。

1.3 预测性竞态检测

预测性竞态检测问题是检查观测轨迹 σ 是否存在预测性竞态。如上所述，已知预测性竞态检测问题是难解的，且是固定参数难解的 [30]。因此，研究主要聚焦于开发算法，以检查是否存在特定类型的预测性竞态，例如先发生关系竞态或同步保持竞态。设 R 是一种预测性竞态类型。R竞态检测问题如下所示。

R-竞态检测问题 给定一条轨迹 σ，判断是否存在一对冲突事件 e_1，$e_2 \in E_{\sigma}$ 使得 (e_1，e_2) 是一个 R-竞态。

对于先发生关系和同步保持这两类竞态检测问题，都存在单遍线性时间算法。虽然这两类竞态都存在基于自动机 [14, 3]和基于向量时钟 [20,18, 31] 的算法，但在实践中能够扩展到大型轨迹的算法是向量时钟算法。先发生关系和同步保持对应的向量时钟算法的细节将在第3.1节和第4.1节中介绍。即使某个竞态检测问题存在一个线性时间算法，许多情况下开销仍可能很高，尤其是在处理超大型轨迹时。例如，任何能够检测是否存在同步保持竞态的单遍流式算法，都必须使用线性空间（即与轨迹长度成正比的空间） [31]。

窗口化 [44, 41] 已被提出作为应对这一挑战的一种可行方法。在窗口化中，轨迹被划分为长度为（例如）w 的子轨迹，然后算法只在每个子轨迹中搜索竞态。将子轨迹长度限制在一定范围内，可确保竞态检测算法的开销有界。

本文对窗口化方法进行了推广：我们不再局限于固定的子轨迹，而是判断轨迹是否包含短程竞争——即冲突事件之间的距离有界的竞态。在实践中，存在数据竞争的程序通常会出现短程竞争，而检测这类竞争也是竞态检测随机化算法的基础 [45]。

短距离 R竞态检测问题 为了对短程竞争进行形式化推理，我们首先定义跨度的概念：对于索引 $0 \leq i \leq j < |\sigma|$，将 (i, j) 的跨度定义为包含端点的子轨迹 $\sigma[i, j]$ 的长度。因此，$\mathsf{span}(i, j) \triangleq j - i + 1.$。特别地，$\mathsf{span}(0, 1) = 2$。

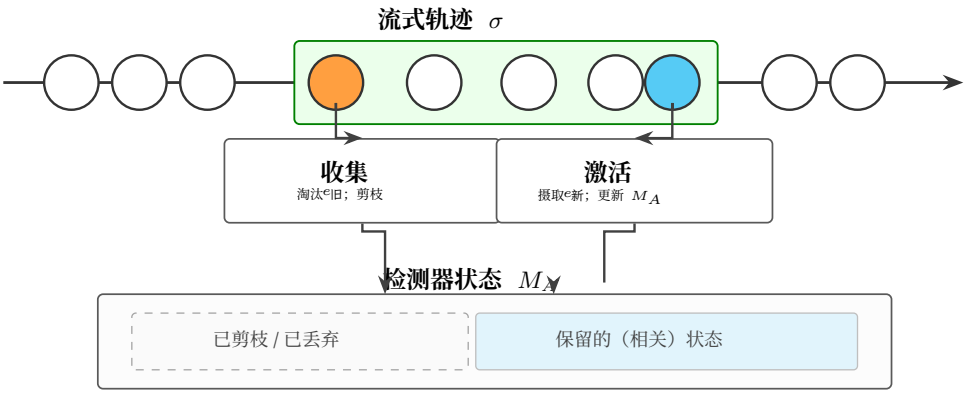


图 1监控框架的高层机制。大小为 w 的窗口沿事件流滑动，以枚举候选项。窗口前移时，collect（左边界）会逐出 e_{old}，并剪除与历史相关、但对未来 Short-R 检查无关的状态。activate（右边界）会摄入 e_{new} 并更新底层检测器状态 M_A，从而将内存占用限制在 $O(w)$。

那么，短距离 R-数据竞争检测问题是：给定轨迹 σ 和 $w \in \mathbb{N}$，判断是否存在冲突事件 $e_i, e_j \in E_\sigma$，其中 $i < j$，使得 e_i 和 e_j 构成一个 R-竞态，且 $\text{span}(i, j) \leq w$。

▶ 注2（判定与报告）。尽管上述问题以判定形式表述，我们的算法旨在实现更强的报告目标：在线扫描轨迹 σ 时，识别并可报告由距离不超过 w 的冲突对所见证的 Short-R 竞争。只需在报告首个竞态后停止（或者等价地，检查报告集合是否为空），即可将判定问题作为一种特殊情况得到。我们采用报告目标，因为动态竞争检测器（包括先发生检测器）通常会扫描完整条轨迹，并配置为输出所关注的竞态。

2 短程竞争监控框架

Short-R 竞态是指由冲突对（e_i, e_j）见证的 R-竞态，且其跨度 $\text{span}(i, j) \leq w$（在完整轨迹 σ 中）。我们仅将大小为 w 的滑动窗口用作在线枚举机制：任何满足 $\text{span}(i, j) \leq w$ 的事件对都必须同时出现在窗口中，因此，只关注窗口内共同出现的事件对，不会遗漏任何 Short-R 见证。关键在于，尽管见证对 是局部的，谓词“（e_i, e_j）构成 R-竞态”可能取决于窗口之外的执行上下文。因此，挑战在于仅保留足够的窗口外上下文，以便对所有未来的窗口局部候选项评估与完整轨迹中相同的 R 判定，同时使内存占用受 w 的函数所界定。

我们提出一种通用监控框架，在有界内存下运行针对 R-竞态的可靠且完备的流式检测器 A_R。随着事件到达，该框架维护一个包含最近 w 个事件的窗口，并运行底层检测器 A_R，同时动态更新并清除其依赖历史的状态，以确保空间有界。抽象而言，窗口仅用于枚举跨度满足 $\text{span}(i, j) \leq w$ 的候选见证对，而检测器状态则在线剪枝，一旦某些信息对未来的 Short-R 检查不再有用，便立即将其丢弃。

对于每个传入事件 e，框架会执行两项操作：
(1) 激活(e)。将 e 插入窗口，并推进底层检测器状态，就像处理完整轨迹一样。具体而言，activate 执行 A_R通常的更新逻辑，并检查涉及 e 与当前仍存活的窗口内事件（即跨度至多为 w 的事件）之间的数据竞争。

(2) 收集(e_{old})。 当窗口超过大小 w 时，驱逐最早的事件 e_{old}，并回收所有不再相关 于任何未来 Short-R 见证的检测器状态。驱逐时，collect 会剪除由 A_R维护的、依赖于历史的状态：它会丢弃不再影响任何未来 Short-R 检查的部分，同时保留 A_R状态中剩余的部分，作为必要的窗口外上下文。

该框架面向短程竞争检测的报告式表述，并要求对于 Short-R 竞争具有可靠性和完备性：它绝不会报告虚假的 Short-R 竞争，并会报告由跨度为 $\leq$w 的冲突对（e_i, e_j）见证的每一个 Short-R 竞争，其中 span(i, j)。

图 1 展示了我们用于短程竞争检测的监控框架的高层结构。随着事件在线处理，我们维护一个大小为 w 的滑动窗口，以枚举短见证候选项，同时运行底层的流式竞态检测器 A 并维护其内部状态 M_A。这两个操作对应于窗口的两个边界：当新事件从右侧进入时，activate 会更新 M_A；而 collect 则从左侧移除最旧的事件，并剪除 M_A 中不会影响任何未来窗口内检查的部分。剩余的残余状态为后续候选项提供必要的窗口外上下文。

有两个方面使实例化并非易事。首先，必须明确当事件离开窗口时， A_R 状态中的哪些部分可以安全丢弃，以及哪些部分必须作为紧凑摘要保留下来，以便为所有未来的窗口局部候选项保留完整轨迹判定；否则，空间占用仍会随 N 增长。其次，必须实现 collect，使得在确保内存有界的同时，不会渐进地增加 A_R的运行时间。我们针对先发生关系和同步保持竞态的实例化通过推导特定于竞态的紧凑摘要，并将边界维护整合到A_R现有的更新逻辑中，同时不改变其渐进时间复杂度，从而解决了这两个挑战。

在本文余下部分中，我们将该框架分别用于先发生关系竞态和同步保持竞态（通过 SyncP），从而得到空间使用量取决于 w 而非 N 的检测器。这些结果汇总于表1，我们的实验也证明了它们的有效性。

3 先发生关系竞态

先发生关系（HB）竞态是历史最悠久、应用最广泛的数据竞争概念。在预测分析框架下，HB 竞态可视为最简单的预测性竞争。HB 仍然是并发测试的标准概念——并发测试会检查一次具体执行中是否存在竞态——也是 ThreadSanitizer (TSan) [42] 等广泛部署工具的基础。一种经典且高效的可靠 HB 竞态检测算法是 FastTrack [18], , 它使用向量时钟计算 HB 关系。FastTrack 的效率已经很高，能够实现线性时间和对数空间性能。尽管如此，研究短 HB 竞态在理论上仍颇有意义：由于 HB 是占主导地位的竞态定义，阐明其在短见证等限制条件下的复杂度，不仅能提供有原则的基线，也能加深我们对竞态检测中局部性约束的理解。

在实践中，已有一些针对 HB 竞争的窗口化方法，旨在使 FastTrack 保持易于处理。例如，TSan 等工具会为每个位置维护一个较小的有界历史记录（“窗口”），保存近期访问；只有当冲突对落在该窗口内时才报告竞态，从而限制每次访问所需的元数据检查。不过，这些技术本质上主要是启发式方法：它们既不会改变 HB 竞态检测的渐近复杂度，也没有被表述为由显式资源上界参数化的算法。受此观察启发，我们从理论角度研究短 *HB* 竞争问题，并设计一种

其复杂度由窗口大小 w 参数化的算法。对于见证间距不超过 w 个事件的所有 HB 竞态，我们的算法都是可靠且完备的，从而明确呈现了现有工具中仅隐含存在的、有原则的成本—覆盖率权衡。

我们首先注意到，先发生关系（HB）竞争具有局部性：对于轨迹 σ中的任意两个事件，它们是否构成 HB 竞争，仅取决于二者之间发生的事件。

▶ **命题 3** (先于关系竞争的局部性)。 设 $\sigma = e_0 e_1 \dots e_n$ 是一条良构轨迹，并假设其中包含一个冲突对 (e_i, e_j)，该冲突对在 σ 中构成 *HB* 竞争。那么，子轨迹 $\sigma[i:j]$ 中的 (e_i, e_j) 也构成 *HB* 竞争。

这一观察表明，可以通过滑动窗口算法解决短先于关系竞争检测问题：该算法在长度为 w 的子轨迹中搜索先于关系竞争。

因此，我们先回顾 FastTrack 算法，之后将对其进行调整，使其融入我们的框架，从而在检测短先于关系竞争的同时降低渐近空间需求。接下来，我们固定待分析轨迹的以下参数：N 表示轨迹长度，T 表示线程数，V 表示变量数，L 表示锁数，w 表示第 3.2 节中所考虑的短竞争的最大长度。

3.1 FastTrack 算法

FastTrack 算法为每个事件计算一个向量时间戳。将事件 e 的局部时间 定义为：在与 e 相同的线程中， σ 之前发生的事件数量。

$$\mathsf{L}^{\sigma}_{\mathsf{FT}}(e) = |\{f \mid f <^{\sigma}_{\mathsf{TO}} e\}| \tag{1}$$

事件 e 的因果时间戳 随后定义为

$$\mathsf{C}^{\sigma}_{\mathsf{FT}}(e) = \lambda t \cdot \max\{\mathsf{L}^{\sigma}_{\mathsf{FT}}(f) \mid \mathsf{thr}(f) = t, f \leq^{\sigma}_{\mathsf{HB}} e\} \tag{2}$$

直观地说，局部时间确保每个事件都有不同的时间。因果时间戳的关键性质是： $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$ 当且仅当 $\mathsf{C}^{\sigma}_{\mathsf{FT}}(e_1) \sqsubseteq \mathsf{C}^{\sigma}_{\mathsf{FT}}(e_2)$。

算法 1 用于检测 HB 竞态的向量时钟算法

```
1  函数 initialize
2      对于每个 t ∈线程，执行
3          C_t ← ⊥
4      遍历 ℓ ∈锁 执行
5          C_ℓ ← ⊥
6      遍历 x ∈内存 执行
7          e_x^w ← null@0;  C_x^r ← ⊥
8  处理程序 read(t, x)
9      C_t ← C_t[t ↦ C_t(t) + 1]
10     如果 e_x^w ⋢ C_t 则声明竞态
11     C_x^r ← C_x^r[t ↦ C_t(t)]
12 处理程序 write(t, x)
13     C_t ← C_t[t ↦ C_t(t) + 1]
14     如果 C_x^r ⋢ C_t 或 e_x^w ⋢ C_t 则声明 竞态
15     e_x^w ← t@C_t[t]
16 处理程序 acquire(t, ℓ)
17     C_t ← C_t[t ↦ C_t(t) + 1]
18     C_t ← C_t ⊔ C_ℓ
19 处理程序 release(t, ℓ)
20     C_t ← C_t[t ↦ C_t(t) + 1]
21     C_ℓ ← C_t
```

FastTrack（算法 1）会随着轨迹的展开，动态计算每个事件的 $\mathsf{C}^{\sigma}_{\mathsf{FT}}(e)$，从而无需为每个事件都维护一个向量时钟。相反，它只维护少量向量时钟和变量。对于每个变量 x，它都会维护一个纪元 e^{w}_{x}，用于存储对 x 的最后一次写入的局部时间；具体而言，纪元的形式为 $t@c$，其中 t 是线程标识符，c 是其局部时间。给定一个向量时钟 $\mathbb{C}$，记号 $\mathsf{e}^{w}_{x} \sqsubseteq \mathbb{C}$ 成立，当且仅当 $c \leq \mathbb{C}[t]$。

此外，它还维护向量时钟，用于存储每个线程在每个变量上最近一次读操作的局部时间（$\mathbb{C}^{\mathsf{r}}_{x}$）、每把锁最近一次释放的因果时间（$\mathbb{C}_{\ell}$），以及每个线程最近一次事件的因果时间（$\mathbb{C}_{t}$）。

▶ 备注 4。FastTrack 算法通常会采用一种优化：仅在释放事件之后递增局部时钟。这种优化不会改变该算法的渐近复杂度。

分析 FastTrack 分配 T +V +L 个向量时钟。由于每个向量时钟包含 T 个条目，且每个条目的值最大可达轨迹长度 N，因此该算法使用 O((T +V +L)T log N) 空间。对于每个读或写事件，FastTrack 都会将当前事件的时间戳与上一次访问的时间戳进行比较，并在必要时报告数据竞争。对于每个获取和释放事件，它会执行向量合并和复制操作，以传播因果信息。因此，该算法的运行时间为 O(NT)，其中每项操作（释放、获取和写）所需的代价为 O(T)。

3.2 检测短先发生竞争

我们现在以 FastTrack 作为底层流式检测器 A_{HB}，针对短距离先发生关系竞态实例化我们的监控框架。根据先于关系竞争的局部性（命题 3），带有 $j - i \leq$w 的冲突对 (e_i, e_j) 是否构成 HB 竞争，仅取决于子轨迹 $\sigma[i{:}j]$。因此，大小为 w 的滑动窗口仅用于枚举候选事件对。

剩下的挑战纯粹是算法层面的：我们应如何设计 activate(·) 与 collect(·) 算子，使该框架同时改善时间与空间复杂度？一种朴素方法是维护最近 w 个事件的窗口，并在每次滑动后对窗口内容从头重新运行（或重新模拟） A_{HB} 。尽管与在完整轨迹上运行 A_{HB} 相比，这降低了空间 占用，但它引入了难以承受的运行时间 开销（本质上是额外的 Θ(w) 倍因子）。

我们的解决方案则将 activate(·) 设计为使用标准 FastTrack 更新规则处理每个传入事件，同时维持检测器状态仅反映当前窗口中事件这一不变式。因此，主要技术难点在于 collect(·)：当事件从窗口中过期时，我们回收与事件关联的访问及同步元数据，并更新任何仍可能引用它的剩余状态，从而使检测器的空间使用量取决于 w 而非 N，且不增加 FastTrack 的渐近运行时间。

从抽象层面看，FastTrack 维护一种状态，用于概括截至目前观察到的关键事件。具体而言，对于每个内存位置，它都会在该位置的时钟状态中存储最近的读写访问元数据（按线程区分）；对于每个线程，它都会在线程的向量时钟中存储该线程截至目前观察到的、来自其他每个线程的最近一次释放。

在我们的短暂竞争框架中，激活(e)会分配（或复用）与 e 关联的元数据记录，并将其安装到 FastTrack 状态中，以便窗口中的后续事件可以引用它。对偶地，当 e 过期时，收集(e)必须（i）移除所有对

e的记录在当前检测器状态中的引用，并（ii）回收该记录，同时以与实际 引用数量成正比的时间完成此操作。

对于读和写事件，这很简单：它们的记录仅通过每个位置的“最近访问”字段被引用，因此 collect(e) 可以在对应条目仍指向 e 时将其清除，然后回收该记录。

释放事件则更为微妙。单个释放记录可能传播到许多线程的向量时钟中，因此，当该释放事件离开窗口时，可能已有大量向量时钟条目引用它。朴素地执行 collect(e)，就必须扫描所有线程的时钟，以查找并清除这类引用。为避免这种开销，我们为 activate(·) 配备了一种转置的簿记结构：每当处理某个事件导致某个向量时钟条目开始引用释放记录 r 时，我们还会在针对 r 的索引中登记这一事实。因此，在窗口边界调用 collect(r) 时，它可以准确枚举当前引用 r 的向量时钟条目，原地更新这些条目，然后安全地回收 r 的元数据，而无需进行任何全局遍历。

以上讨论给出了 activate(·) 和 collect(·) 的高层设计。不过，在这一层面上仍有两个问题尚未解决。首先，我们还没有论证其可靠性：回收过期事件的元数据后，所维护的状态仍必须等价于在当前窗口上运行 FastTrack，这样报告的每个短程竞争才都是真正的 HB 竞争。其次，我们还没有达到预期的空间 界限：即便我们只存储窗口驻留事件的元数据，FastTrack 仍使用全局 局部时间，其位宽会随 N 增长。

通过窗口等价性保证可靠性。 对于 HB 竞态，局部性（命题 3）意味着，报告短程竞争只取决于当前窗口，因此我们的正确性论证归结为维护窗口等价性 不变式：每次更新后，所维护的状态在语义上等价于（至多对相对窗口时间戳进行重命名后）仅在当前窗口上运行 FastTrack。唯一非平凡之处在于确保 collect(·) 能够回收因窗口过期而失效的元数据，同时不破坏这一不变式；这正是上述转置记账机制所保证的，并且它避免了任何全局遍历，从而保留了 FastTrack 的渐近更新时间。

消除剩余的 log N 依赖。 现在我们转向互补的空间目标。在原版 FastTrack 中，每个向量时钟分量都存储一个全局局部时间，该时间最多可增长到 N，因此每个分量需要 Θ(log N) 位。由于短距离先发生关系竞态完全由当前窗口决定，我们可以改用相对窗口时间概念，并随着窗口推进回收标识符，从而将每个分量的开销降至 Θ(log w) 位。更具体地说，我们将分配给每个事件的时间戳替换为它们在窗口中的索引。为了有效维护窗口，我们将其实现为循环数组。设窗口索引为$\{0, 1, \ldots, \mathrm{w}-1\}$，我们将窗口表示为一个循环数组 $W[0..\mathrm{w}-1]$，其头指针为 $h \in \{0, 1, \ldots, \mathrm{w}-1\}$。

形式化地，设 $\mathbb{W} = (W, h)$ 是一个长度为 w 的窗口，其中 $h \in \{0, 1, \ldots, \mathrm{w}-1\}$ 是头位置。对于 $i, j \in \{0, 1, \ldots, \mathrm{w}-1\}$，定义 $i \leq_{\mathbb{W}} j$ 当且仅当以下条件之一成立：(a) $j = h$，(b) $j < h < i$，或 (c) $h < \min(i, j)$ 或 $h > \max(i, j)$ 且 $i < j$。向量时间戳 ($\sqsubseteq_{\mathbb{W}}$) 之间的排序采用通常的逐点排序，其中分量上的排序为 $\leq_{\mathbb{W}}$。

所有这些思想都已纳入我们的算法，如算法 2 所示。与 FastTrack（算法 1）相比，我们着重指出本算法的显著变化和细微差异。该算法使用大小为 w 的循环数组 $\mathbb{W}$来表示窗口。激活函数接收一个新事件 e 并更新 h。它首先对即将从窗口中丢弃的事件调用收集函数，然后在将 e 加入窗口之前，对新事件调用相应的**处理程序** 。 S_i 表示

其条目与存储在 $W[i]$处的释放事件相关联的线程和锁的集合，如上所述。直观而言，**后处理程序** 会撤销**处理程序**中已发生的操作，因此不会影响该算法的渐近运行时间。

▶**定理 5**。 算法 *2* 正确解决了短先行发生竞态检测问题。其运行时间为 $O(\mathsf{NT})$，使用的空间为 $O(\mathsf{w} + (\min(\mathsf{w}, \mathsf{V} + \mathsf{L}) + \mathsf{T})\mathsf{T}\log \mathsf{w})$。

复现包的附录中给出了定理 5 的证明。

▶**示例 6** （短轨迹上的循环窗口）。 考虑轨迹

$$e_1 = [t_1 : \mathtt{w}(y)],\ e_2 = [t_1 : \mathtt{w}(x)],\ e_3 = [t_2 : \mathtt{w}(x)],\ e_4 = [t_2 : \mathtt{w}(y)].$$

存在两个 HB 竞态：(e_1, e_4)，跨度为 4；以及 (e_2, e_3)，跨度为 2。设窗口大小为 $\mathsf{w} = 2$，循环头指针为 $h \in \{0, 1\}$，初始化为 -1。

处理 e_1，e_2 会在不发生驱逐的情况下填充窗口，并记录 $\mathsf{e}_y^w = t_1@0$ 和 $\mathsf{e}_x^w = t_1@1$。当 e_3 到达时，e_1 被驱逐并收集，将 e_y^w 重置为 null。由于 e_2 与 e_3 之间不存在先发生关系边，FastTrack 写检查失败，检测器报告 HB 竞态 (e_2, e_3)。

接下来，处理 e_4 会逐出 e_2。由于 e_x^w 不再指向被逐出的写入，因此不会检测到其他竞态，而 (e_1, e_4) 因窗口边界而被漏检。

▶**示例 7** （循环窗口比较）。考虑轨迹

$$\ldots\ e_1 = [t_1 : \mathtt{w}(x)],\ e_2 = [t_1 : \mathtt{rel}(\ell)],\ e_3 = [t_2 : \mathtt{acq}(\ell)],\ e_4 = [t_2 : \mathtt{w}(x)].$$

假设循环窗口大小为 4。假设处理 e_1 时，框架将其存储在最后一个单元格中，即索引 3 处，并将 e_2，e_3，e_4 存储在前三个单元格中。同步边 $e_2 \leq^{\sigma}_{\mathsf{HB}} e_3$ 将 t_2 传播到 t_1最新事件的当前窗口索引，即$\mathbb{C}_{t_2}[t_1] = 0$ （ e_2的索引）。处理 e_4时，头指针为 $h = 2$。尽管 x 上的最后一次写入存储为 $\mathsf{e}_x^w = t_1@3$ 和 $\mathbb{C}_{t_2}[t_1] = 0$，框架不会在$x$上报告竞态：在由 h导出的窗口感知顺序 $\sqsubseteq_{\mathbb{W}}$ 下，循环缓冲区中的索引 0 比索引 3 更新，这意味着 $3 <_{\mathbb{W}} 0$，因此竞态检查仍然可靠。

4 同步保持竞态

同步保持竞态是一类通用的数据竞争。如前所述，它涵盖了文献中研究的其他竞态概念，包括先发生关系（HB）竞争（见命题 1）和可调度先行发生（SHB）竞态 [29]：任何包含 HB（或 SHB）竞态的轨迹也包含同步保持竞态，但反之则不成立。与只依据程序顺序和同步顺序进行推理的 HB 不同，同步保持竞态还会依据读自关系所导致的可行替代执行进行推理，从而提升预测能力。

这种额外的预测能力需要付出计算代价。尽管同步保持竞态存在流式线性时间算法（例如 SyncP [31] 和基于自动机的方法 [3]），但任何此类算法在最坏情况下都必须使用线性空间 （ [31] 的定理 3.2）。直观来说，空间开销膨胀是不可避免的，因为探索可行的替代执行需要维护因果关系信息，而这可能涉及长度无界的依赖链。具体而言，这意味着同步保持竞态具有非局部性，与 HB 竞态形成鲜明对比。

算法 2 检测长度为 w 的短先发生关系竞态

```
1  函数 initialize
       // Convention: all per-location/per-lock maps (e.g., C^r_x, e^w_x, C_ℓ) default to null if unset.
2      W ← [空]^w
3      h ← −1
4      对每个 t ∈线程 执行
5          C_t ← ⊥
6  函数运行(σ)
7      对于每个传入事件 e 于 σ 执行
8          激活(e)
9  函数激活(e)
10     h ← (h + 1) 模 w
11     e_out ← W[h]
12     如果 e_out ≠空 则
13         收集(e_out)
14     调用处理程序 on e
15     W[h] ← e
16 函数收集(e_out)
       // Dispatch to the corresponding post-handler of the evicted event
17     如果 e_out 是一个读(t, x) 事件 那么
18         调用 后处理程序 read(t, x)
19     如果 e_out 是一个写(t, x) 事件 那么
20         调用后处理程序 write(t, x)
21     如果 e_out 是释放(t, ℓ) 事件 则
22         调用后处理程序 release(t, ℓ)
23 处理程序 read(t, x)
24     如果 e^w_x ⋢_W C_t 则声明竞态
25     如果 C^r_x =null 然后 C^r_x ← ⊥
26     C^r_x ← C^r_x[t ↦ h]
27 处理程序 write(t, x)
28     如果 C^r_x ⋢_W C_t 或 e^w_x ⋢_W C_t 则声明 race
29     e^w_x ← t@h
30 处理程序 acquire(t, ℓ)
31     对于每个 t* ∈线程 _σ do
32         如果 C_ℓ[t*] >_W C_t[t*] 那么
33             S_{C_ℓ[t*]}.添加 (t)
34             S_{C_t[t*]}.移除 (t)
35             C_t[t*] ← C_ℓ[t*]
36         S_{C_ℓ[t*]}.移除 (ℓ)
37     C_ℓ ←空
38 处理程序 release(t, ℓ)
39     S_{C_t[t]}.移除(t)
40     S_h ← ∅
41     S_h.添加(t)
42     C_t ← C_t[t ↦ h]
43     C_ℓ ← ⊥
44     对每个 t* ∈线程_σ 执行
45         S_{C_t[t*]}.添加(ℓ)
46         C_ℓ[t*] ← C_t[t*]
47 后处理程序 read(t, x)
48     如果 C^r_x[t] = h 则 C^r_x[t] ← −1
49     如果驱逐后，对 x 没有读访问在 W 中则 C^r_x[t] ←为空
50
51 后处理程序 write(t, x)
52     如果 e^w_x = t@h 则 e^w_x ←为空
53 后处理程序 release(t, ℓ)
54     对每个 t* ∈ S_h 执行
55         C_{t*}[t] ← −1
56     对 ℓ* ∈ S_h 中的每个元素执行
57         C_{ℓ*}[t] ← −1
58     如果 C_ℓ[t] = h，那么对 t* ∈线程_σ 中的每个元素执行 do
59         S_{C_ℓ[t*]}.移除(ℓ)
60     C_ℓ ←空
61
62     S_h ←空
```

▶**命题 8**。 存在一个以 e_1 开始、以e_2结束的良构子轨迹 σ，以及良构轨迹 σ_1 和 σ_2，使得 σ 是 σ_1 和 σ_2的子轨迹，但(e_1, e_2) 在 σ_2 中是同步保持竞态，而在 σ_1中不是。

证明。 考虑子轨迹

$$\sigma = [t_1\colon e_1 = \mathtt{w}(x)][t_3\colon \mathtt{r}(x)][t_3\colon \mathtt{rel}(\ell)][t_2\colon \mathtt{acq}(\ell)][t_2\colon \mathtt{rel}(\ell)][t_2\colon e_2 = \mathtt{w}(x)].$$

现在考虑这些良构轨迹

$$\sigma_1 = [t_3\colon \mathtt{acq}(\ell)]\,[t_1\colon \mathtt{w}(y)]\,[t_2\colon \mathtt{r}(y)] \cdot \sigma, \qquad \sigma_2 = [t_3\colon \mathtt{acq}(\ell)] \cdot \sigma.$$

我们有 SPIdeal$_{\sigma_1}(e_1, e_2) \cap \{e_1, e_2\} = \{e_1\}$，因此 (e_1, e_2) 在 σ_1 中不是同步保持竞争。相比之下，SPIdeal$_{\sigma_2}(e_1, e_2) \cap \{e_1, e_2\} = \varnothing$，这意味着 (e_1, e_2) 在 σ_2是同步保持竞争。◀

因此，仅靠滑动窗口算法不足以检测短程同步保持竞态：即使两个事件在同一窗口内共同出现，通常也无法仅根据窗口内容判断它们在完整执行中是否构成同步保持竞态。本节将说明，同步保持竞态仍可通过高效的短竞争检测器来检测：我们将滑动窗口与紧凑的参数有界摘要结合起来，后者恰好保留了评估窗口局部候选项所需的窗口外信息。我们还将进一步证明，最先进的流式检测器 SyncP 可以顺利融入我们的框架，从而为 Short-SyncP 竞争提供一种可靠且完备的算法，其空间使用量仅取决于固定参数（而非 N）。我们首先概述 SyncP 算法。

4.1 SyncP 算法：以状态为中心的视角

在现有的同步保持竞态检测技术中，SyncP [31] 最具可扩展性：它支持流式检测，并能在线性时间内处理执行轨迹。基于自动机的方法 [3], 虽然优雅且富有表现力，但依赖于状态空间构造，其规模会随相关参数呈指数级增长，因此在实践中无法扩展到大型执行。鉴于此，我们选择 SyncP 作为底层检测器 A_{SyncP}，并将其纳入我们的框架。

我们以满足框架实例化所需的详细程度，对 SyncP 算法作一份自包含的概述。完整的描述和所有证明见 [31]。回顾第 1.2 节，冲突对 (e_1, e_2) 是同步保持竞争，当且仅当 SPIdeal$_\sigma(e_1, e_2) \cap \{e_1, e_2\} = \varnothing$。因此，随着轨迹以流的形式输入，SyncP 必须（隐式地）为冲突对计算 SPIdeal$_\sigma(e_1, e_2)$。其关键结构性质是，SyncP 操作的集合（包括 TL 闭包$_\sigma(\cdot)$ 和 SPIdeal$_\sigma(\cdot, \cdot)$）关于轨迹顺序 $\leq^{\sigma}_{\mathsf{TO}}$向下封闭，因此可以用向量时间戳紧凑地表示。

在线维护 TL 闭包$_\sigma(e)$。 随着事件到来，SyncP 会为每个已处理事件 e维护其 TL 闭包$_\sigma(e)$ 的向量时间戳。此过程通过一小组向量时钟增量完成，具体而言：对于每个内存位置 x，都维护一个时钟 $\mathbb{LW}_x$，用于存储表示 TL 闭包$_\sigma(\ell)$的向量时间戳，其中 ℓ 是迄今为止观察到的、对x执行的最近一次写操作。

直观地说，对于读事件 $e = \mathtt{r}(x)$，可行性要求 e “看到”对 x 的某次写，因此 TL 闭包$_\sigma(e)$ 必须包含最近的写摘要，即其时间戳在逐点意义下 $\sqsupseteq \mathbb{LW}_x$。（程序顺序和同步的更新也通过并适当的向量时间戳以类似方式处理；见 [31]。）

使用 SPIdeal$_\sigma(e_1, e_2)$ **谓词进行竞态检测**。 要判定 (e_1, e_2) 是否构成竞态，SyncP 必须计算 SPIdeal$_\sigma(e_1, e_2)$，其通过不动点构造定义（见第1节第2小节）。在操作层面，该不动点由反复发现同线程获取对并插入匹配释放来驱动。因此，SyncP 按轨迹顺序存储两类事件摘要：

- **临界区摘要**。 对于每个线程 t 和锁 ℓ，一个列表 $\mathsf{CS}_{t,\ell}$ 其条目形式为 (TL 闭包$_\sigma(a)$, TL 闭包$_\sigma(\text{match}_\sigma(a))$)，其中 a 遍历线程ℓ上由 t触发的获取事件。

- 访问摘要。 对于每个线程 t、访问类型 $a \in \{\mathtt{r},\ \mathtt{w}\}$和位置 x，一个列表 $\mathrm{Acc}_{t,\ a,\ x}$其中包含条目 (TL 闭包$_\sigma$(prev$_\sigma(e)$)，TL 闭包$_\sigma(e)$)，每个条目对应线程 t在 x 上的类型为a 的访问事件 e 。这些列表使 SyncP 能够将新观察到的访问与相关的先前冲突访问进行检查。

增量性与单调性。SyncP 能够进行流式、线性时间处理，一个关键原因是它可以共享不同冲突对的不动点计算。SyncP 通过维护以下内容来利用单调性：(i) 缓存的向量时间戳，用于概括针对每种冲突模式最近计算出的理想；(ii) 指向 CS 列表和 Acc 列表的只向前移动的指针，从而使每个条目只需处理有限次。（此处省略完整的簿记细节；参见 [31]。）

复杂度。总体而言，SyncP 以 O(NT^2 +AVT^3) 的时间处理长度为 σ 的轨迹，并使用 O (VLT^3 + NT log N)[1] 的空间。具体而言，NT log N 这一项反映了 SyncP 维护与事件关联的向量时间戳，其分量存储全局局部时间（因此每个分量需要 Θ(log N) 位），且事件数量为线性级别。这种固有的线性空间依赖促使我们提出短竞争框架，旨在使所维护的状态取决于短竞争参数 w，而非完整轨迹长度 N。

4.2 检测短同步保持竞态

检测同步保持竞态的任何流式算法都需要线性空间下界 [31]，这表明，使用次线性空间检测短同步保持竞争的高效算法可能大有裨益。与 HB 情形一样，我们在实例化该框架时的目标是保持——并在理想情况下进一步提升——底层 SyncP 检测器的运行时间，同时消除其对轨迹长度 N 的空间依赖。然而，正如命题 8 所示，同步保持竞态具有非局部性，判断窗口局部冲突对是否构成竞态可能需要窗口之外的信息。因此，我们的目标并非消除对 N 的所有 依赖；相反，首要目标是去除对 N 的线性 依赖，使窗口和固定参数之外的辅助空间至多为次线性。

我们首先注意到，同步保持竞态在良构轨迹上定义明确：具体而言，不存在未匹配释放，因此基于 SPIdeal$_\cdot(\cdot,\ \cdot)$ 的竞争谓词没有歧义。

▶ **命题 9**。 设 σ 为一条良构轨迹，且$(e_1、\ e_2)$ 是 σ中的一个冲突对。对于任意良构轨迹 σ'，若 σ 是 σ'的子轨迹，则$(e_1、\ e_2)$ 是σ' 中的同步保持竞争，当且仅当$(e_1、\ e_2)$ 是 σ中的同步保持竞争。

因此，一个直接的基线做法是维护一个无界窗口：对于每个未匹配的获取事件，保留其后的所有事件，直到该获取事件由其对应释放匹配（也就是说，保留自最早的未匹配获取事件以来的后缀）。然而，这并不能节省空间：如果某个获取事件长时间未匹配，保留的后缀可能随轨迹长度线性增长，导致空间使用量基本与原始 SyncP 相同。

尽管如此，这一基线仍为我们的设计提供了有用的指导原则：对于当前位于窗口中的事件，我们旨在恰好维护 SyncP

[1] 文献 [31] 中报告的 SyncP 空间界存在一个小错误（作者已确认）。此处给出的是修正后的界。

在执行中对应的良构后缀上运行时会为这些事件维护的信息。换言之，我们设计的状态在限定于窗口驻留事件时，与 SyncP 在相同事件上的状态在语义上等价，因此我们能够像 SyncP 在完整轨迹中那样，准确判断窗口局部候选事件对。高层次的直觉是，由读自边所诱导的因果链——这是进行同步保持竞争推理所必需的——可以在线以压缩形式作为向量时间戳来维护。

因此，我们不需要保留基线算法在未匹配获取之后会保留的无界后缀中的所有内存访问事件；相反，这些事件的影响由向量时间戳概括。

这一思路体现在 SyncP 的设计中：SyncP 通过更新向量时钟来评估基于 SPIdeal.(·，·) 的谓词，从而检测竞态：
如上所述，它会为每个事件维护其闭包 TL 闭包$_\sigma$(·) 的向量时间戳表示，并利用这些闭包驱动竞态检查中 SPIdeal$_\sigma$(·，·) 所依赖的不动点计算。TL 闭包$_\sigma$(·) 的维护方式类似于 FastTrack 中的线程向量时钟：SyncP 对每种固定参数组合（例如每个线程／位置）只保留 $O(1)$ 个向量时钟，每个向量时钟分量存储一个全局局部时间，因此需要 $\Theta(\log N)$ 位。这意味着维护 TL 闭包$_\sigma$(·) 并非低效的主要来源——尤其不是导致 SyncP 线性空间占用的原因——因此，我们可以不作修改地将这一组件纳入框架。此外，保留 TL 闭包$_\sigma$(·) 恰好能提供我们所需的因果摘要（以向量时间戳表示的向下封闭因果闭包），用于处理超出当前窗口的依赖链。

因此，唯一必须显式存储的窗口外事件，是那些信息无法被压缩掉的事件，即其匹配释放尚未被观察到的获取事件。直观而言，保留这些未匹配的获取操作对于“补全”窗口是必要的：它使我们能够将窗口内容解释为仿佛构成了一条良构轨迹，从而使窗口内执行的竞争检查与完整执行中具有相同的语义。我们现在已经解决了如何处理 非局部性。为了获得真正的改进，我们仍须设计框架算子，使所得监控器（i）正确 （对 Short-SyncP 竞争既可靠又完备），且（ii）高效，在不增加 SyncP 渐近运行时间的前提下消除对 N 的线性空间依赖。为此，我们首先确定 SyncP 状态中哪些部分是按事件索引的，并且必须在窗口边界处分配/回收。

事件关联视图（由我们的框架使用）。 就我们的目的而言，将 SyncP 状态视为一组与事件动态关联的记录是方便的。这一视角清楚地表明状态的哪些部分随事件“进入”，哪些部分仍被后续计算引用，这对于在我们的短竞争框架中定义边界操作至关重要。

令 e 为一个已处理事件。
1. 如果线程 t 执行 $e = \mathtt{r}(t, x)$（分别地， $e = \mathtt{w}(t, x)$），那么 e 会贡献一个访问摘要条目（TL 闭包$_\sigma$(prev$_\sigma$(e)),TL 闭包$_\sigma$(e)），并将其追加到 Acc$_{t,\mathtt{r},x}$ （分别地，Acc$_{t,\mathtt{w},x}$）。此外，如果e 是 t 在 x上执行的同类访问中最新的一个，那么后续冲突处理可能会引用链表头以及相应的缓存理想，以检测与其他线程之间的冲突。**2.** 如果 $e = \mathtt{acq}(t, \ell)$，那么 e 会在 CS$_{t,\ell}$ 中贡献一个临界区条目；在匹配的释放到达之前，该条目的第二个分量暂时为空。

算法 3 检测长度为 w 的短同步保持竞态（使用 SyncP 的框架实例化）。

```
1  函数 initialize
       // Convention: all per-location/per-lock/per-tuple maps default to null if unset.
2      W ← [空]^w
3      h ← 0
       // Spilled open-acquire record per lock: stores the unique open entry TL 闭包_σ(acq, 空) if any.
4      对每个 ℓ ∈锁 执行
5          打开_ℓ ←null
       // Initialize the underlying SyncP states for every fixed-parameter
6  函数运行(σ)
7      对每个传入事件 e 在 σ 执行
8          激活(e)
9  函数激活(e)
10     h ← (h + 1) 模 w
11     e_out ← W[h]
12     如果 e_out ≠为空 则
13         收集(e_out)
       // Replay: if a release enters but its matching acquire is outside the window, complete the window.
14     如果 e 是一个 rel(t, ℓ) 事件 ∧打开_ℓ ≠为空 则
           重放(e)
15
       // Allocate event-local records needed by SyncP for this event.
16     分配(e)
       // Underlying detector step: invoke the standard SyncP handler (unchanged).
17     调用SyncP 处理程序于 e
18     W[h] ← e
19 函数spill(e_out)
       // spill only when an acquire is evicted while still unmatched.
20     如果 e_out 是一个获取(t, ℓ) 事件
       ∧ 匹配_σ(e_out) ∉ W 则
       Open_ℓ ← TL 闭包_σ(e_out,null)。 // spill the open entry TL 闭包_σ(e_out, null)
       fromCS_t, ℓ into打开_ℓ
21 函数重放(e)
       // Reinsert the spilled open CS-entry from 打开_ℓ back intoCS_t, ℓ.
       // At the moment a release is processed with its matching acquire out of window, CS_t, ℓ has no window-resident open entry.
22     CS_t, ℓ ←打开_ℓ
23     打开_ℓ ←空
24 函数收集(e_out)
       // (1) Spill unmatched acquires if needed (to preserve well-formedness of future windows).
25     spill(e_out)
       // (2) Reclaim event-local records and detach any references from monotone bookkeeping.
26     释放(e_out)
27 函数分配(e)
       // Allocate only the event-local records that SyncP would create for e (nodes inAcc orCS).
       // All persistent components (e.g., ℂ_t, 𝕃𝕎_x, cached ideals, forward-only pointers) are updated by SyncP.handler.
28 函数释放(e_out)
       // Free all event-local records associated with e_out that remain in-window (access/CS nodes).
       // Also detach any references from cached ideals/pointers so that no dangling pointers remain.
```

3. 如果 $e = \texttt{rel}(t, \ell)$，则 SyncP 完成相应的临界区条目，其方式是填入 TL 闭包σ(匹配 $\sigma(e)$)，并且后续的理想计算可以通过仅前向指针引用此已完成的条目。

此外，SyncP 还维护与固定参数相关联的信息，而不是按单个事件维护。具体而言，它为每个线程 t、每个内存位置 x 以及每个锁 l保存一个向量时钟。与事件局部记录不同，这些按线程、位置和锁划分的结构在我们的框架中是持久的，因为它们对于判定短竞态见证是必要的，因此不会由我们的窗口边界操作创建或回收。

总之，SyncP 可以理解为维护一组事件关联记录（访问记录和临界区条目），以及少量单调簿记信息（缓存的理想和前向指针），这些簿记会逐步消耗这些记录。

算法 3 给出了我们框架的高层视图。该框架首先初始化一个大小为 w 的窗口，以及 SyncP 中为固定参数预留的数据结构。

如上所述，包装函数 `allocation()` 和 `deallocation()` 会分配/释放事件局部记录。为了确保 `deallocation()` 安全，我们在 collect 的实现中使用了两个内部记账辅助函数：spill 和 replay。直观地说，spill 会提取未匹配获取操作的信息；否则，这些信息在驱逐时只能通过事件局部列表 $\text{CS}_{t,\ \ell}$ 访问。随后，spill 将这些信息存储在单独的变量 Open_{ℓ}中。这样，`deallocation()`就能从 $\text{CS}_{t,\ \ell}$ 中分离并释放指针，同时保留稍后所需的未匹配获取信息，从而使标准 SyncP 逻辑能够照常运行。之后，当匹配的释放进入窗口时，replay 会将 spill 出的信息从 Open_{ℓ} 重新注入 $\text{CS}_{t,\ \ell}$，使标准 SyncP 处理程序能够照常运行并保持可靠。

对于每个传入事件 e_{new}，激活算子都会推进滑动窗口，插入 e_{new}，识别被驱逐的事件 e_{old}，并在窗口边界调用 collect(e_{old})。

随后，监控器会对 e_{new} 调用标准 SyncP 处理程序，仅使用驻留在窗口中的记录以及固定参数记录。

算法 3 的运行时间与 SyncP 相同（当 w 足够小时甚至更快），同时消除了 SyncP 所需的、对轨迹长度的线性空间依赖。证明见附录。

▶**定理 10.** 算法 *3* 能正确解决短同步保持竞态检测问题。其运行时间为 $O(\mathsf{NT}(\min(\mathsf{w}, \mathsf{T})) + \mathsf{AT}(\min(\mathsf{w}, \mathsf{VT}^2)))$，空间复杂度为 $O(\min(\mathsf{w}, \mathsf{VT})\mathsf{LT}^2 + (\mathsf{w} + \mathsf{V} + \mathsf{L})\mathsf{T}\log \mathsf{N})$。

5 实验

概述。 本节将介绍我们对两种短程竞争算法进行的实验评估，实验采用了用于竞态预测的标准基准测试套件。由于 FastTrack 的效率已经很高，我们对短程 FastTrack 的研究主要考察其空间复杂度方面的理论改进能否转化为实际性能提升。相比之下，SyncP 算法虽然是预测同步保持竞争的最先进方案，但众所周知，即使面对中等规模或较小的基准测试（轨迹长度低于 1M），它也难以处理。因此，我们预计短程 SyncP 算法将显著提升预测同步保持竞态的实用性。因此，本节**主要**着重分析并理解短程 SyncP 的表现。我们先总结运行短程 FastTrack 算法所得的结果，再介绍短程 SyncP 算法的结果。

设置。 实验在一台配备 36GB 内存的 MacBook Pro 上进行

并搭载 M3 芯片。为确保实验在不同环境中的可复现性和一致性，我们使用 Docker 容器封装了一个基于 Linux 的环境。我们在 RAPID 的基础上使用 Java 实现了论文中提出的两种算法。RAPID 是一个用于分析执行跟踪日志的离线动态分析框架。它内置了经过优化的 FastTrack 实现，也是唯一公开提供 SyncP 实现的框架。

基准测试。我们的实验使用来自 [43], 的执行轨迹，**总计包含 153 个程序**：（i）**30 个 Java 程序**，取自多个基准测试套件，包括 IBM Contest [15], DaCapo [6], SIR [12], Java Grande 论坛 [40], 以及若干独立基准测试；（ii）**123 个 OpenMP/高性能计算基准测试**，取自 OmpSCR [13],DataRaceBench [25], DataRaceOnAccelerator [39], NAS 并行基准测试 [4], CORAL [2],ECP代理应用程序 [26], 和 Mantevo 套件。所有执行轨迹仅记录对共享变量的访问，以及通过锁对象上的获取/释放事件进行的同步。

执行轨迹特征。我们的基准测试套件总共包含 153 条执行轨迹。为更好地展示结果，我们过滤掉了 59 条事件数少于 1,000 万的执行轨迹。在剩余的 92 条执行轨迹中，最长的一条包含 9 亿个事件，而执行轨迹大小的平均值和中位数分别为 1.23 亿和 7,000 万个事件。为便于展示，我们将执行轨迹分为以下几类：OMPRacer（12 条）、DRACC（8 条）、DataRaceBench（30 条）、HPCBench（34 条）、CoMD（6 条）和其他（4 条）。

配置与短程竞争长度的选择。我们在 RAPID 的 FastTrack 和 SyncP 实现上实现了算法 2 和算法 3，分别称为 ShortFastTrack 和 ShortSyncP。每种工具都在 28GB 内存限制和 1 小时时间限制下运行。

我们将 RAPID 的 FastTrack 实现作为 ShortFastTrack 的基线。不过，RAPID 的 SyncP 实现效率极低，即使资源限制大幅提高（4 小时和 400GB [43]），在本研究的大多数基准测试上仍会超时。相比之下，我们的 ShortSyncP 实现进行了大量优化，因此即使在资源限制更严格的情况下，窗口大小设为轨迹长度的 100% 时，也能完成 37 个包含超过 1000 万个事件的基准测试（以及经过筛选、事件数少于 1000 万的基准测试）。因此，我们使用竞争长度为轨迹长度 100% 的 ShortSyncP 作为评估 ShortSyncP 的基线。本节余下部分将把这一基线称为 SyncP。对于 ShortFastTrack，我们对每条基准轨迹分别采用两种短程竞争上限——3000 和 30000（3K、30K），并对每种配置重复运行三次。选择这两种竞争长度有两个原因。首先，先前一项通过采样子轨迹来检测竞态的研究 [45],表明，30K 的上限普遍适用于检测先发生竞争，但该研究没有评估 3K 这样更小的上限；因此，我们纳入 3K 以评估其效果。其次，这些上限使我们能够展示一种潜在的空间节省方式：将向量时钟条目存储在 Java 基本类型 `short` 中，而非 `int`。

相比之下，对于 ShortSyncP，我们以更大的竞争长度运行每条轨迹——100 万和 1000 万（1M 和 10M）——每种配置重复三次。我们选择这些轨迹长度同样出于两个原因：（i）先前的工作 [31] 报告称，同步保持竞态的跨度可以很长；（ii）经过我们的优化后，SyncP 能够在不出现内存溢出（OOM）错误的情况下完成所有最高达 10M 个事件的基准测试，这表明以该长度运行 ShortSyncP 是可行的。这也是我们建议开发者选择竞争长度的方式。

可靠性与完备性。正如第2节所述，我们的算法和

实现满足以下保证。可靠性：它们报告的每个竞态都是真正的 HB 竞争或同步保持竞争。完备性：对于任意跨度上界 w，它们都能检测到跨度至多为 w的所有短程竞争。

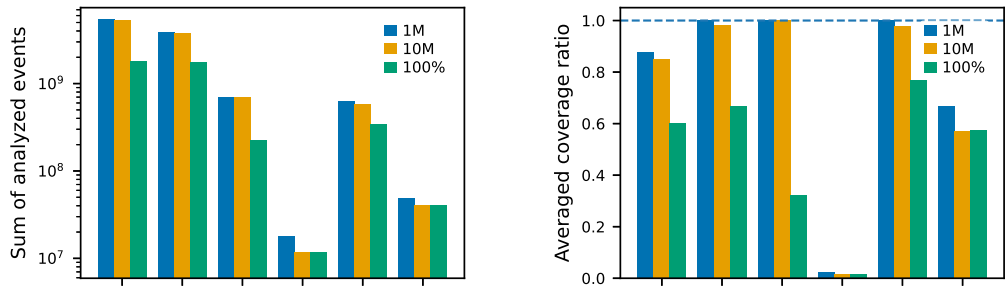
ShortFastTrack。

基准测试组	配置	平均时间（毫秒）	竞争事件	竞争变量	最大堆内存 (MiB)
DataRaceBench	3K	5,377,487	64,546,596	46,166	16,799.5
	30K	6,152,631	172,952,750	51,569	24,198.3
	100%	6,169,788	172,990,978	55,077	17,552.2
HPCBench	3K	2,844,152	4,741,389	43,049	33,199
	30K	3,123,535	34,584,174	84,672	36,828
	100%	4,633,655	775,349,288	9,289,456	36,230
CoMD	3K	551,748	2,148,054	8,878	3,210.9
	30K	618,002	15,941,852	17,855	3,808
	100%	857,893	603,504,406	67,114	3,426.5
DRACC	3K	629,986	401,225,627	2,172	4,793
	30K	696,393	401,225,627	2,172	7,167.7
	100%	609,849	401,225,627	2,172	4,356.7
OMPRacer	3K	460,955	13,239,193	21,888	8,900.1
	30K	517,229	64,256,697	261,779	9,526.5
	100%	761,538	332,176,279	1,280,965	9,821.8
其他	3K	114,734	56,748	3,443	2,111.9
	30K	118,589	329,138	8,402	3,063
	100%	122,657	13,967,485	9,382	2,258.7

表3按基准测试组划分的竞争长度配置（10K、30K、100%）结果；100%为FastTrack实现。每个指标（运行时间、竞争事件、竞争变量、最大堆内存）均为该组内各基准测试之和，且先对每个基准测试的多次运行取平均后再计算。我们仅纳入事件数至少为10M的执行轨迹。

表3展示了在基准测试套件上运行ShortFastTrack的结果。所有配置均在时间限制内成功完成每一条基准轨迹。我们通过运行时间、所报告的竞争事件数量和位置，以及最大堆使用量来衡量各配置的效率。对于每条轨迹，我们对三次重复运行的结果取平均，然后按组汇总数值。尽管算法 2与FastTrack具有相同的渐近运行时间，但由于管理从窗口中进行事件移除会带来额外开销，我们预计ShortFastTrack会更慢。然而，如表3所示，情况并非总是如此。在DRACC基准测试中，ShortFastTrack相对于FastTrack并未表现出明显改进。相比之下，在所有其他基准测试套件中，较小的轨迹长度3K在运行时间上取得了明显收益。然而，我们也观察到，此类收益总是伴随着所报告的竞态事件和竞态变量的减少。

我们针对 ShortFastTrack 所做的实验，并未提供有力证据表明它能在各项基准测试中降低内存使用量。在若干情况下，较大的轨迹长度配置（30K）所使用的堆内存甚至多于 FastTrack。造成这种情况的一个因素是，FastTrack 的内存空间会随固定参数（线程数、锁数和变量数）线性增长，而只随执行长度对数增长——更准确地说，是随每线程最大值



(a)按组统计的分析事件总数（对数刻度）。

(b)按组统计的分析事件数与基线之比的平均值（线性刻度）。

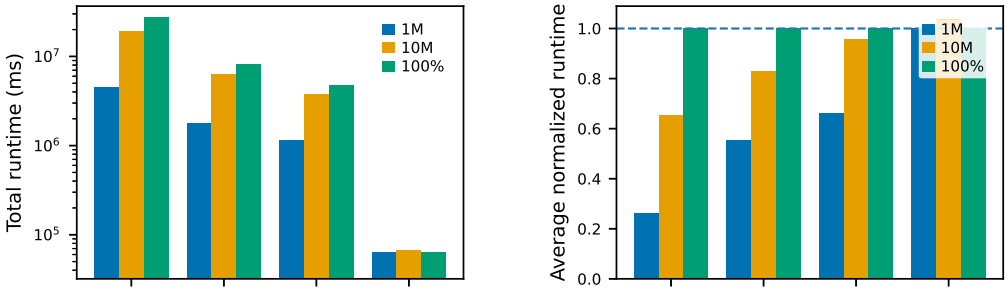
图 2各基准测试组的事件数。(a)：分析的事件总数（对数刻度）。(b)：各基准测试组中，分析事件数相对于各轨迹长度的归一化值的平均值。

锁操作数对数增长。因此，除非这些固定参数较大，或轨迹中每个线程的锁操作很多，否则窗口化带来的节省微乎其微。此外，维护窗口所需的簿记开销可能超过其收益；我们目前的实现尚未针对空间进行充分优化，这可能会进一步放大这一影响。值得指出的是，如果我们将其与一种把向量时钟条目存储为 `long` 而非`int`的 FastTrack 实现进行比较，节省的内存会更加显著。不过，在我们的实验中，最长的轨迹包含 900M 个事件，完全在 32 位 `int`的范围内，因此差异不那么明显。

ShortSyncP. RAPID 的 SyncP 实现是目前唯一公开可用的 SyncP 实现，但效率很低。正如 Shi 等人 [43], 所报告的，它未能覆盖足够多的事件来探索竞态条件，甚至在许多事件数不足一百万的轨迹上，未能完成其数据结构的初始化。我们的算法 3（ShortSyncP）实现包含多项优化；在窗口大小为 100% 时，它成功分析的轨迹比例远高于 RAPID 的 SyncP。不过，它仍未能完成大多数轨迹的分析。

轨迹覆盖率。 因此，我们首先量化 ShortSyncP 在轨迹覆盖率（已分析事件数）方面取得的提升。图 2 左侧面板报告了各基准测试组中分析的事件数，右侧面板则显示平均覆盖率比值，该比值定义为各轨迹的（已分析事件数 ÷轨迹长度）的平均值。窗口大小为 1M 和 10M 的 ShortSyncP 比 SyncP 分析了更多事件，并在大多数基准测试中实现了显著更高的覆盖率。两种窗口大小的覆盖率相近，因为在我们的资源限制下，两者通常都能在许多基准测试中完成全部分析。然而，对于 DRACC 组，窗口化对覆盖率的提升甚微；我们的调查表明，这是由于 SyncP 在以同步事件为主的轨迹上存在固有复杂度。总体而言，我们得出结论：ShortSyncP 显著提高了覆盖率。

运行时间。 我们的分析表明，ShortSyncP 改善了运行时间复杂度；现在我们评估其实际影响。图3报告了每个基准测试组的总运行时间和平均归一化运行时间（相对于 SyncP）。为确保公平，我们排除了（i）SyncP 因内存耗尽而提前终止的 42 条执行轨迹，以避免由提前终止造成的偏差；以及（ii）所有配置均超时的 14 条执行轨迹（包括全部 8 条 DRACC 执行轨迹）。不同于



(a)各组总运行时间（毫秒）。 **(b)**各组平均归一化运行时间。

图 3（a）：总运行时间（对数）。（b）：相对于基线的平均归一化运行时间。

（a）平均竞争事件比例

基准测试组	1M	10M	100%	n
DataRaceBench	54.47x	165.61x	1.00x	30
HPCBench	37.40x	1279.43x	1.00x	34
CoMD	3.55x	3.68x	1.00x	6
DRACC	1.32x	1.00x	1.00x	8
OMPRacer	1.84x	1.75x	1.00x	12
其他	1.17x	0.98x	1.00x	4

(b)平均竞态位置比率

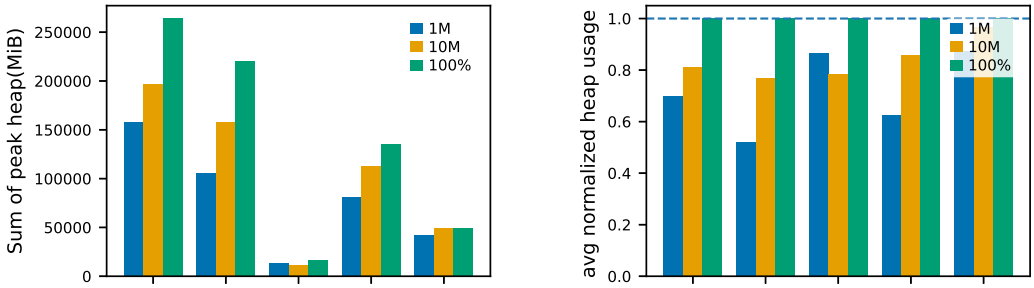
基准测试组	1M	10M	100%	n
DataRaceBench	7.23x	7.21x	1.00x	30
HPCBench	36.93x	1278.72x	1.00x	34
CoMD	1.00x	1.00x	1.00x	6
DRACC	1.00x	1.00x	1.00x	8
OMPRacer	0.98x	1.03倍	1.00倍	12
其他	1.15倍	0.99倍	1.00倍	4

表 4 每组平均归一化竞态事件数和竞态位置数（占 100% 的比例）。

覆盖率结果，1M 配置的开销远低于 10M 配置和 SyncP。在 DataRaceBench 和 HPCBench 上，其平均归一化运行时间分别为 SyncP 的 26% 和 55%。10M 配置的表现也优于 SyncP，但开销高于 1M 配置。

竞态预测能力。接下来，我们考察更高的覆盖率是否会带来更高的竞态暴露。由于不同基准测试中的竞态绝对数量差异很大（部分执行轨迹中多达数百万次），我们在表 4 中报告两项指标的平均归一化检测率：(1) 检测到的竞态对；(2) 检测到的竞态位置。对于每条执行轨迹，我们记录各配置报告的数量，以同一条执行轨迹上基线的数量对其进行归一化，然后对同一组内各执行轨迹的比率取平均值。值得注意的是，10M 配置的检测能力显著更强：在 HPCBench 中，两项指标均最高可达 1000 倍；在 DataRaceBench 中（它包含了我们大部分的执行轨迹），竞态对和竞态位置分别达到 165 倍和 7.2 倍。1M 配置相比基线也有显著提升，但总体上仍不及 10M。因此，尽管 1M 配置通常也能实现更高的覆盖率，更大的窗口仍能带来更高的暴露度，这表明额外的时间上下文对于发现竞态至关重要。

内存使用量。最后，我们评估内存使用量。在 92 个基准测试中，SyncP 基线在 42 条执行轨迹上触发了内存溢出（OOM），而 ShortSyncP 的两种配置（1M、10M）均未报告内存溢出。这已经表明，ShortSyncP 的内存效率显著优于 SyncP。对于其余 50 条执行轨迹，我们通过记录垃圾回收行为并记录每次运行的最大堆使用量来量化内存使用情况。如



(a) 平均最大堆使用量之和。 **(b)** 平均归一化最大堆使用量。

图 4(a)：平均最大堆使用量之和。(b)：分析事件的平均值相对于基线最大堆使用量的归一化值。

图 4 所示，即使在内存压力不大的执行轨迹上，ShortSyncP 也能大幅降低峰值堆使用量。

6 相关工作

静态与动态竞态检测 数据竞争是一类普遍存在且众所周知难以检测的并发错误。竞态通常定义为一对在先发生关系下无序的冲突事件；竞态的存在不仅意味着行为具有非确定性，也表明存在更严重的问题，包括数据损坏 [8] 和不健全的编译器优化。大量研究提出了具有重要影响的静态与动态竞态检测方法。静态分析[32, 7, 17, 1, 46] 旨在通过基于类型的系统，或对数据和控制流进行过程间分析，验证不存在数据竞争。由于验证并发程序在计算上十分困难，实用的静态分析器通常要么报告大量误报，要么会漏掉大量数据竞争。值得注意的是，RacerD [7] 及其后续版本 [19] 展现出更高的实用精度，大幅减少了错误警报。动态分析基于具体执行进行，因此检测范围受限于运行时实际执行的行为。除了检查锁规范的基于锁的分析 [38] 之外，动态分析主要关注给定执行中是否存在竞态。在这一领域，先发生关系（HB）仍是公认的标准概念，为大量研究 [14, 20, 18, 35, 16] 和 ThreadSanitizer 等工业工具提供了基础。

预测性动态竞争检测提升动态分析的一种关键方法是预测性竞态检测：给定一次执行，推断对相同事件进行某种可行的替代重排序是否会暴露出竞态，而不是仅根据观察到的顺序进行推理。我们的工作聚焦于先发生关系（HB）竞争 [24] 和同步保持竞态 [31], ，它们体现了两种互补范式：基于偏序的探索与计算正确重排。HB 已被进一步完善为可调度发生先于（SHB） [29]„ 它保证报告的每个竞态都能在轨迹的正确重排中实现。因果优先关系（CP） [44] 和弱因果优先关系（WCP） [22] 则在保留易处理算法的同时进一步放宽了 HB——CP 的算法复杂度为多项式时间，WCP 的为线性时间。类似地，M2 [34] 和 OSR [43] 定义了可通过受约束的正确重排发现的竞争类别，

与 SyncP 类似。

窗口化 窗口化是一种自然的方法，可在实践中降低数据竞争预测的开销。在窗口化中，数据竞争预测问题旨在检测所有长度固定为 [44, 41] 的子轨迹中的数据竞争。然而，正如本文所示，竞争定义在子轨迹上并不总是定义良好，这可能导致漏掉某些短程竞争，或仅在特定条件下检测到数据竞争，从而使预测能力不再仅取决于事件间距。本文讨论的短竞争检测问题将窗口化加以推广，旨在检测固定数量的事件内发生的数据竞争，同时保证底层竞争定义的正确性。滑动窗口算法 [11] 将窗口化范式进一步扩展，考察某个性质是否在流中最近的 w 个事件上成立。我们关于检测短程竞争的工作可视为将这一思路专门应用于数据竞争这一性质，关键区别在于，我们允许窗口外的某些事件影响窗口内是否检测到数据竞争。

基于采样的竞态检测 基于采样的竞态检测通过仅分析事件的一个子集来降低开销。我们的 ShortFastTrack 算法与 [45], 的思路相近，后者通过采样固定长度的子轨迹来检测先发生竞争。 [45]侧重于选择要分析哪些子轨迹，而我们的工作研究了对每条子轨迹都进行采样的情形，并刻画了分析这些子轨迹的固有复杂度。从概念上说，短竞态问题是子轨迹采样方法的一个特例。

尽管关于基于采样的竞态检测的文献十分丰富，但其研究对象主要是先发生竞争。相比之下，针对其他预测性概念的采样受到的关注很少，原因恰恰在于为这些更丰富的关系设计采样方案在技术上十分困难：它们施加了非局部约束（例如同步保持以及见证重排的可行性），因此，简单的子采样可能会破坏可靠性，而有原则的方案则可能带来难以承受的开销。在此背景下，我们检测短同步保持竞争的方法为这一方向的未来研究奠定了基础。

7 结论

我们提出并分析了在观测轨迹中检测短程竞争这一问题。我们介绍了用于检测短程先发生关系竞争和同步保持竞争的算法；与针对这些竞争的经典算法相比，这些算法的运行时间相近或更短，且所需空间更少。实验突显了这些短程竞争检测算法的有效性，表明它们能够显著降低开销，同时仍能识别出许多数据竞争。未来，探索适用于其他类型竞争的短程竞争检测算法将很有价值。此外，研究此类算法对随机化竞争检测的影响也将是一个有趣的后续方向，因为后者通常依赖于识别短程竞争。

参考文献

1 Abadi, M., Flanagan, C., Freund, S.N.: 安全锁定类型：Java 静态数据竞争检测。ACM Trans. Program. Lang. Syst. 28(2), 207–255（2006年3月） 2 高级模拟与计算，劳伦斯利弗莫尔国家实验室：CORAL 基准测试（2022年）， https://asc.llnl.gov/coral-benchmarks，访问日期：2022-04-11 3 Ang, Z., Mathur, U.: 基于强迹前缀的预测性监控（2024）， https://arxiv.org/abs/2405.10499 4 Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: NAS 并行基准测试——概要与初步结果。载于：1991年 ACM/IEEE 超级计算会议论文集。第158–165页。超级计算 '91，计算机协会，美国纽约州纽约（1991）。doi:10.1145/125826.125925 5 Biswas, S., Cao, M., Zhang, M., Bond, M.D., Wood, B.P.: 面向生产运行的轻量级数据竞争检测。载于：第26届国际编译器构造会议论文集。第11–21页。CC 2017，计算机协会，美国纽约州纽约（2017）。doi:10.1145/3033019.3033020, https://doi.org/10.1145/3033019.3033020 6 Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: DaCapo 基准测试：Java 基准测试开发与分析。载于：第21届 ACM SIGPLAN 面向对象编程系统、语言与应用会议论文集。第169–190页。OOPSLA '06，ACM，美国纽约州纽约（2006）。doi:10.1145/1167473.1167488, http://doi.acm.org/10.1145/1167473.1167488 7 Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD：组合式静态数据竞争检测。ACM 程序设计语言会议录 2(OOPSLA)（2018年10月）。doi:10.1145/3276514, https://doi.org/10.1145/3276514 8 Boehm, H.J.: 如何错误编译存在"良性"数据竞争的程序。载于：第3届 USENIX 并行计算热点主题会议论文集。第3页。HotPar'11，USENIX 协会，美国（2011） 9 Boehm, H.J.: 立场论文：非确定性不可避免，但数据竞争是纯粹的祸害。载于：2012年 ACM 多核与众核可扩展性同步放宽研讨会论文集。第9–14页。RACES '12，计算机协会，美国纽约州纽约（2012）。doi:10.1145/2414729.2414732, https://doi.org/10.1145/2414729.2414732 10 Bond, M.D., Coons, K.E., McKinley, K.S.: Pacer：数据竞争的比例检测。载于：第31届 ACM SIGPLAN 编程语言设计与实现会议论文集。第255–268页。PLDI '10，ACM，美国纽约州纽约（2010）。doi:10.1145/1806596.1806626, http://doi.acm.org/10.1145/1806596.1806626 11 Datar, M., Gionis, A., Indyk, P., Motwani, R.: 在滑动窗口上维护流统计量。SIAM 计算杂志 31(6), 1794—1813（2002）12 Do, H., Elbaum, S., Rothermel, G.: 支持采用测试技术进行受控实验：一种基础设施及其潜在影响。实证软件工程：国际期刊 10, 405–435（2005）13 Dorta, A., Rodriguez, C., de Sande, F.: OpenMP 源代码仓库。载于：第13届 Euromicro 并行、分布式与网络处理会议。第244–250页（2005）。doi:10.1109/EMPDP.2005.41 14 Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks：数据竞争与事务感知型 Java 运行时。载于：第28届 ACM SIGPLAN 编程语言设计与实现会议论文集。第245–255页。PLDI '07，ACM，美国纽约州纽约（2007）。doi:10.1145/1250734.1250762, http://doi.acm.org/10.1145/1250734.1250762

15 Farchi, E., Nir, Y., Ur, S.: 并发错误模式及其测试方法。见：第17届国际并行与分布式处理研讨会论文集。第286.2–页。IPDPS '03，IEEE计算机学会，华盛顿特区，美国（2003年） 16 Fidge, C.: 分布式计算系统中的逻辑时间。《Computer》24(8)，28–33（1991年8月）。doi:10.1109/2.84874, http://dx.doi.org/10.1109/2.8487417 Flanagan, C., Freund, S.N.: 面向Java的基于类型的数据竞争检测。见：ACM SIGPLAN 2000年编程语言设计与实现大会论文集。第219–232页。PLDI '00，ACM，纽约，NY，美国（2000年）。doi:10.1145/349299.349328, http://doi.acm.org/10.1145/349299.34932818 Flanagan, C., Freund, S.N.: FastTrack：高效且精确的动态数据竞争检测。见：第30届ACM SIGPLAN编程语言设计与实现大会论文集。第121–133页。PLDI '09，ACM，纽约，NY，美国（2009年）。doi:10.1145/1542476.1542490, http://doi.acm.org/10.1145/1542476.154249019 Gorogiannis, N., O'Hearn, P.W., Sergey, I.: 静态数据竞争检测器的真阳性定理。ACM 程序设计语言会议录 3(POPL)（2019年1月）。doi:10.1145/3290370, https://doi.org/10.1145/329037020 Itzkovitz, A., Schuster, A., Zeev-Ben-Mordehai, O.: 迈向在分布式共享内存系统中集成数据竞争检测。J. Parallel Distrib. Comput. 59(2)，180–203（1999年11月）。doi:10.1006/jpdc.1999.1574, http://dx.doi.org/10.1006/jpdc.1999.157421 Kasikci, B., Zamfir, C., Candea, G.: RaceMob：众包数据竞争检测。见：第二十四届ACM操作系统原理研讨会论文集。第406–422页。SOSP '13，ACM，纽约，NY，美国（2013年） 22 Kini, D., Mathur, U., Viswanathan, M.: 线性时间内的动态数据竞争预测。见：第38届ACM SIGPLAN编程语言设计与实现大会论文集。第157–170页。PLDI 2017，ACM，纽约，NY，美国（2017年）。doi:10.1145/3062341.3062374, http://doi.acm.org/10.1145/3062341.306237423 Kulkarni, R., Mathur, U., Pavlogiannis, A.: 通过细粒度视角进行动态数据竞争检测。见：国际并发理论会议论文集。第16:1–16:23页（2021年） 24 Lamport, L.: 分布式系统中的时间、时钟与事件排序。Commun. ACM 21(7)，558–565（1978年7月） 25 Liao, C., Lin, P.H., Asplund, J., Schordan, M., Karlin, I.: DataRaceBench：用于系统化评估数据竞争检测工具的基准测试套件。见：高性能计算、网络、存储与分析国际会议论文集。SC '17，计算机协会，纽约，NY，美国（2017年）。doi:10.1145/3126908.3126958 26 劳伦斯利弗莫尔国家实验室：ECP代理应用程序（2022年）， https://proxyapps.exascaleproject.org，访问日期：2021年4月1日 27 Lu, S., Park, S., Seo, E., Zhou, Y.: 从错误中学习：对现实世界并发错误特征的综合研究。见：第13届编程语言与操作系统体系结构支持国际会议论文集。第329–339页。ASPLOS XIII，ACM，纽约，NY，美国（2008年）。doi:10.1145/1346281.1346323, http://doi.acm.org/10.1145/1346281.134632328 Marino, D., Musuvathi, M., Narayanasamy, S.: LiteRace：面向轻量级数据竞争检测的有效采样。见：第30届ACM SIGPLAN编程语言设计与实现大会论文集。第134–143页。PLDI '09，ACM，纽约，NY，美国（2009年）。doi:10.1145/1542476.1542491, http://doi.acm.org/10.1145/1542476.154249129 Mathur, U., Kini, D., Viswanathan, M.: 第一次竞态发生后会怎样？增强基于先发生关系的动态数据竞争检测的预测能力。ACM 程序设计语言会议录 2(OOPSLA)，145:1–145:29（2018年10月）。doi:10.1145/3276515, http://doi.acm.org/10.1145/3276515

30 Mathur, U., Pavlogiannis, A., Viswanathan, M.: 动态数据竞争预测的复杂度。见：第35届ACM/IEEE计算机科学逻辑年度研讨会论文集。第713–727页。LICS '20，计算机协会，纽约，NY，美国（2020年）。doi:10.1145/3373718.3394783, https://doi.org/10.1145/3373718.3394783 31 Mathur, U., Pavlogiannis, A., Viswanathan, M.: 保持同步的数据竞争的最优预测。ACM 程序设计语言会议录 5(POPL)（2021年1月）。doi:10.1145/3434317,https://doi.org/10.1145/3434317 32 Naik, M., Aiken, A., Whaley, J.: 面向Java的有效静态数据竞争检测。见：第27届ACM SIGPLAN编程语言设计与实现大会论文集。第308–319页。PLDI '06，ACM，纽约，NY，美国（2006年）。doi:10.1145/1133981.1134018,http://doi.acm.org/10.1145/1133981.1134018 33 Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., Calder, B.: 利用重放分析自动分类良性与有害数据竞争。见：第28届ACM SIGPLAN编程语言设计与实现大会论文集。第22–31页。PLDI '07，计算机协会，纽约，NY，美国（2007年）。doi:10.1145/1250734.1250738, https://doi.org/10.1145/1250734.1250738 34 Pavlogiannis, A.: 快速、可靠且实质上完备的动态竞争预测。ACM 程序设计语言会议录 4(POPL)（2019年12月）。doi:10.1145/3371085, https://doi.org/10.1145/3371085 35 Pozniansky, E., Schuster, A.: 多线程C++程序中的高效即时数据竞争检测。见：第九届ACM SIGPLAN并行编程原理与实践研讨会论文集。第179–190页。PPoPP '03，ACM，纽约，NY，美国（2003年）。doi:10.1145/781498.781529, http://doi.acm.org/10.1145/781498.781529 36 Roemer, J., Genç, K., Bond, M.D.: 高覆盖率、无界的可靠预测式竞争检测。见：第39届ACM SIGPLAN编程语言设计与实现大会论文集。第374–389页。PLDI 2018，ACM，纽约，NY，美国（2018年）。doi:10.1145/3192366.3192385, http://doi.acm.org/10.1145/3192366.3192385 37 Roemer, J., Genç, K., Bond, M.D.: SmartTrack：高效的预测性竞争检测。见：第41届ACM SIGPLAN编程语言设计与实现大会论文集。第747–762页。PLDI 2020，计算机协会，纽约，NY，美国（2020年）。doi:10.1145/3385412.3385993, https://doi.org/10.1145/3385412.3385993 38 Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser：面向多线程程序的动态数据竞争检测器。ACM Trans. Comput. Syst. 15(4)，391–411（1997年11月）。doi:10.1145/265924.265927, https://doi.org/10.1145/265924.265927 39 Schmitz, A., Protze, J., Yu, L., Schwitanski, S., Müller, M.S.: DataRaceOnAccelerator——用于评估面向加速器的正确性工具的微基准测试套件。见：Schwardmann, U., Boehme, C., B. Heras, D., Cardellini, V., Jeannot, E., Salis, A., Schifanella, C., Manumachu, R.R., Schwamborn, D., Ricci, L., Sangyoon, O., Gruber, T., Antonelli, L., Scott, S.L.（编）Euro-Par 2019：并行处理研讨会。第245–257页。Springer International Publishing，Cham（2020年）。doi:10.1007/978-3-030-48340-1_19 40 Sen, K., Roşu, G., Agha, G.: 通过对执行过程的广义预测分析检测多线程程序中的错误。见：第7届IFIP WG 6.1开放面向对象分布式系统形式化方法国际会议论文集。第211–226页。FMOODS'05，Springer-Verlag，柏林，海德堡（2005年） 41 Şerbănuţă, T.F., Chen, F., Roşu, G.: 面向顺序一致系统的极大因果模型。见：运行时验证国际会议。第136–150页。Springer（2012） 42 Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer：实践中的数据竞争检测。见：二进制插桩与应用研讨会论文集。第62–71页。WBIA '09，ACM，纽约，NY，美国（2009年） 43 Shi, Z., Mathur, U., Pavlogiannis, A.: 同步反转数据竞争的乐观预测。见：IEEE/ACM第46届软件国际会议论文集

工程。ICSE '24，计算机协会，纽约，纽约州，美国（2024）。doi:10.1145/3597503.3639099， https://doi.org/10.1145/3597503.3639099 Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.：多项式时间内可靠的预测式竞争检测。载于：第39届 ACM SIGPLAN-SIGACT 编程语言原理年会论文集。第 387–400 页。POPL '12，ACM，纽约，纽约州，美国（2012）。doi:10.1145/2103656.2103702， http://doi.acm.org/10.1145/2103656.2103702 Thokair, M.A., Zhang, M., Mathur, U., Viswanathan, M.：使用 o(1) 个样本的动态数据竞争检测。ACM 程序设计语言会议录 **7**(POPL)（2023年1月）。doi:10.1145/3571238， https://doi.org/10.1145/3571238 Voung, J.W., Jhala, R., Lerner, S.：RELAY：对数百万行代码进行静态数据竞争检测。载于：第六届欧洲软件工程会议与 ACM SIGSOFT 软件工程基础研讨会联合会议论文集。第 205–214 页。ESEC-FSE '07，ACM，纽约，纽约州，美国（2007） Zhang, M., Lim, D.W.S., Al Thokair, M., Mathur, U., Viswanathan, M.：面向基于采样的竞争检测的高效时间戳记录。ACM 程序设计语言会议录 **9**(PLDI)（2025年6月）。doi:10.1145/3729252， https://doi.org/10.1145/3729252 Zhivich, M., Cunningham, R.K.：软件错误的真实代价。IEEE 安全与隐私 **7**(2), 87–90（2009年3月）。doi:10.1109/MSP.2009.56， https://doi.org/10.1109/MSP.2009.56

定理 5 的证明概要

我们首先证明，算法 2 能正确检测所有跨度至多为 k 个事件的先发生竞争。该算法的关键不变式是，由线程和锁维护的每个向量时钟中的每个条目都记录最近一次释放事件的窗口索引。形式化地说，对于任意线程向量时钟 $\mathbb{C}_t$，在由线程 t处理的任意事件 e 处，条目 $\mathbb{C}_t[t']$ 存储的是线程 t' 执行的、先发生于 e的最近一次释放。如果窗口内不存在此类释放，则条目 $\mathbb{C}_t[t']$ 设为 -1。同样的逻辑也适用于锁向量时钟 $\mathbb{C}_\ell$。

类似地，对于每个读时钟或写纪元，该条目对应于窗口内读或写事件的窗口索引。具体而言，e_x^w 存储对内存位置 x的最后一次写入的窗口索引。如果窗口中不再有此类写事件，则将 e_x^w 设为 -1。这一约定也适用于读向量时钟 $\mathbb{C}_x^{\mathrm{r}}$中的所有条目。

窗口比较器 $\leq_{\mathbb{W}}$ 根据头位置 h比较两个窗口索引 a 和 b，当以下条件之一成立时，有 $a \leq_{\mathbb{W}} b$：(a) $a = -1$，或 (b) $a \leq b \leq h$，或 (c) $h < a \leq b$，或 (d) $b \leq h < a$。结合每个向量时钟的更新方式，这确保当且仅当窗口中存在一对构成先发生竞争的冲突事件时，$\mathbb{C}_x^{\mathrm{r}} \not\sqsubseteq_{\mathbb{W}} \mathbb{C}_t$ 或 $\mathsf{e}_x^w \not\sqsubseteq_{\mathbb{W}} \mathbb{C}_t$ 成立。由此完成正确性证明。

现在我们证明运行时间复杂度和空间复杂度分析是正确的。显然，获取、释放、读和写事件的四个处理程序，其运行时间复杂度与 FastTrack 的对应处理程序相同。读和写事件的后处理程序也在常数时间内运行。尽管释放的后处理程序最多可以访问 T +L 个条目，但请注意，释放事件的 **后处理程序** 只需“撤销”先前事件的影响，而这些事件使该事件的局部时间得以存储。我们将 **后处理程序** 中每一步的开销记在复制了被丢弃释放事件局部时间的获取事件上。由此可知，将获取事件的 **处理程序**的耗时设为 O(2T)，便足以计入释放事件的所有 **后处理程序**所花费的时间。因此，总运行时间为 O(NT)，与 FastTrack 的运行时间相同。该算法使用的内存如下：使用 O(w) 空间来存储窗口中的事件。每个线程都有一个向量时钟，与变量和锁关联的向量时钟数量最多为 w，并且向量时间戳的每个条目最多为 w。接下来，释放事件对应集合 S_i 中的每个元素，都对应于某个与线程或锁关联的向量时钟分量。因此，所有释放事件的 S_i 大小之和也至多为 O((min(w,L) + T)T)。综合以上各项可知，总空间为 O(w +(min(w, V +L) +T)T log w)。

B 定理 10 的证明概要

我们首先论证算法 3 能正确解决短同步保持竞态检测问题。为此，只需证明：当算法正在处理来自流的事件 e_2 时，对于距 e_2不超过k 个事件的每个冲突事件 e_1，算法 3 都能有效判断窗口中的每个事件 e 是否满足$e \in \mathrm{SPIdeal}_\sigma(e_1, e_2)$。由于算法 3 有效模拟了 SyncP 算法，我们首先请读者参阅技术报告 [31]，了解详细背景。回顾一下，算法 3 会为每个事件 e 计算 TL 闭包$_\sigma(e)$，并将其保留在相应的列表中，直到 e 离开窗口。正如 [31], 中所述，给定 TL 闭包$_\sigma(e_1)$和 TL 闭包$_\sigma(e_2)$，计算 SPIdeal$_\sigma(e_1, e_2)$需要在不动点计算过程中逐步确定是否应纳入每个释放事件 e。具体做法是将 match$_\sigma(e)$ 与其之后发生的获取事件进行比较。

在我们的情形中，需要判断窗口内的每个释放事件是否应被纳入。如果某个释放事件 e 的匹配获取 $\text{match}_\sigma(e)$仍在窗口内，则 SyncP 会自然处理这种情况，因为算法 3 会保留 $\text{match}_\sigma(e)$的信息以及其后的所有获取事件；由于 e的存在，这些获取事件也必然位于窗口内。当 $\text{match}_\sigma(e)$移出窗口时，算法 3 会通过函数 spill 保留必要信息，从而能够调用 SyncP 妥善处理这种情况。这确保了该方法的可靠性。

现在，为了证明算法 3 与 SyncP 的运行时间相同（或者当 k 足够小时，运行时间更短），我们需要说明这四个额外函数不会增加渐近复杂度。显然，spill 和 replay 函数的运行时间为常数，因为它们只会从链表中移除头节点，将该节点存放到别处，然后再将其放回链表（只有在链表为空时才会发生）。函数 `allocation` (e) 会分配 SyncP 在处理 e 时会访问和操作的数据结构。对于每个分配的向量时钟，SyncP 至少会执行一次 O(T) 操作，因此，分配大小为 T 的向量时钟不会改变运行时间。显然，对于所有其他数据结构，分配开销都是常数，因此也不会改变运行时间。 `deallocation` 的论证与算法 2 类似，该函数只是“撤销”SyncP 之前执行的操作，确保不会引入额外开销。最后，我们注意到，可以通过统计链表中每个节点被访问的次数，来分析 SyncP 在执行 σ 时的运行时间。按照与 [31]，中分析类似的论证，我们可以证明每个节点至多被访问 w 次。当 w 足够小时，算法 3 的运行时间更短。当 w 较大（大于 VT^2）时，对于存储内存访问事件信息的节点，算法 3 对它们的访问频率至多与 SyncP 相同，访问次数由 k 限定。然而，最坏情况下，算法 3 访问获取节点的总开销可能是 SyncP 的两倍。这种情况发生在获取节点通过 replay 恢复，并被视为新插入的节点时。尽管如此，这不会影响整体渐近复杂度。

空间复杂度可以按如下方式分析：算法 3 为每个锁和内存位置维护一个向量时钟，因此空间复杂度为 O((L +V)T)。对于大小为 w 的窗口，它会为每个事件维护一个向量时钟，共 w 个，即 O(wT)。窗口中的每个事件 e 最多可以贡献 T^2L 个指针；由于指针总数最多为 VLT^3 ，因此还需乘上一个因子 O(min(w，VT)LT^2)。