橡皮擦：用于多线程程序的动态数据竞争检测器

斯蒂芬野蛮人
华盛顿大学
MICHAEL BURROWS, GREG NELSON 和 PATRICK SOBALVARRO
数字设备公司和
托马斯·安德森
加州大学伯克利分校

多线程编程困难且容易出错。在产生数据争用的同步中容易犯错误，但在调试过程中很难找出此错误。本文介绍了一种名为 Eraser 的新工具，用于动态检测基于锁的多线程程序中的数据竞争。橡皮擦使用二进制重写技术来监视每个共享内存引用，并验证是否遵守一致的锁定行为。我们提出了一些案例研究，包括本科课程和多线程 Web 搜索引擎，这些研究证明了这种方法的有效性。

类别和主题描述符：D.1.3 [编程技术]：并行编程-并行编程；D.2.5 [软件工程]：测试和调试-调试工具；监视器追踪 D.4.1 [操作系统]：过程管理-并发；僵局;多处理/多程序；互斥；同步化

通用术语：算法，实验，可靠性

其他关键字和短语：二进制代码修改，多线程编程，竞争检测

1. 介绍

多线程已成为一种常见的编程技术。大多数商业操作系统都支持线程，并且 Microsoft Word 和 Netscape Navigator 等流行的应用程序都是多线程的。

本文的较早版本出现在 1997 年于法国圣马洛举行的第 16 届 ACM 操作系统原理专题讨论会上。
作者的地址：华盛顿大学计算机科学与工程系的 S. Savage 和 T. Anderson，华盛顿州西雅图市，邮编 352350；邮编：98195；电子邮件：savage@cs.washington.edu;M.Burrows, G.Nelson 和 P.Sobalvarro，数字设备公司系统研究中心，加利福尼亚州帕罗奥多市 Lytton Avenue 130 号，邮政编码 94301。

不幸的是，调试多线程程序可能很困难。同步中的简单错误会产生与时序有关的数据争用，可能需要数周或数月才能进行跟踪。因此，许多程序员拒绝使用线程。John Ousterhout 很好地总结了使用线程的困难。[1]

在本文中，我们描述了一种名为 Eraser 的工具，该工具可以动态检测多线程程序中的数据竞争。我们已经为数字 Unix 实现了 Eraser，并用它来检测许多程序中的数据争用，这些程序从 AltaVista Web 搜索引擎到大学生编写的入门编程练习。

动态竞争检测的先前工作是基于 Lamport 的 beforebefore 关系建立的[Lamport 1978]，并检查同步事件将来自不同线程的冲突内存访问分开。之前发生过算法可以处理多种样式的同步，但是这种普遍性是有代价的。我们已经将 Eraser 专门针对现代多线程程序中使用的基于锁的同步。橡皮擦只是检查所有共享内存访问是否遵循一致的锁定规则。锁定规则是一种编程策略，可确保不存在数据争用。例如，简单的锁定规则是要求线程之间共享的每个变量都必须由互斥锁保护。我们将争辩说，对于许多程序来说，橡皮擦执行锁定规则的方法比基于事前发生的方法更容易，更有效，更彻底地捕捉比赛。据我们所知，Eraser 是第一个应用于多线程生产服务器的动态竞争检测工具。

本文的其余部分安排如下。在回顾了什么是数据竞赛之后，并描述了竞赛检测的先前工作之后，我们首先介绍了橡皮擦使用的 Lockset 算法，然后从高层次开始，然后以足够低的层次揭示了关键的性能关键实现技术。最后，我们描述了将橡皮擦与许多多线程程序一起使用的经验。

橡皮擦与 Mellor-Crummey [1993]所构造的同名工具没有关系，该工具用于检测作为 ParaScope 编程环境一部分的共享内存并行 Fortran 程序中的数据竞争。

1.1 定义

锁是用于互斥的简单同步对象。它是可用的，或由线程拥有。对锁 mu 的操作是 lock () 和 unlock ()。因此，它实质上是用于互斥的二进制信号量，但与信号量的不同之处在于，仅允许锁的所有者释放它。

当两个并发线程访问一个共享变量，并且当

[1]在 1996 年 USENIX 技术会议（1 月 25 日）上的邀请演讲。提供演示幻灯片 通过 http://www.smli.com/people/john.ousterhout/threads.ps （后记） 要么 / PPT (PowerPoint)。

—至少一个访问权限是写操作，

—线程不使用显式机制来防止访问同时进行。

如果程序有潜在的数据争用，则对共享变量的访问冲突的结果将取决于线程执行的交织。尽管程序员在不确定性似乎无害时偶尔会故意允许数据争用，但潜在的数据争用通常是由于无法正确同步而引起的严重错误。

1.2 相关工作

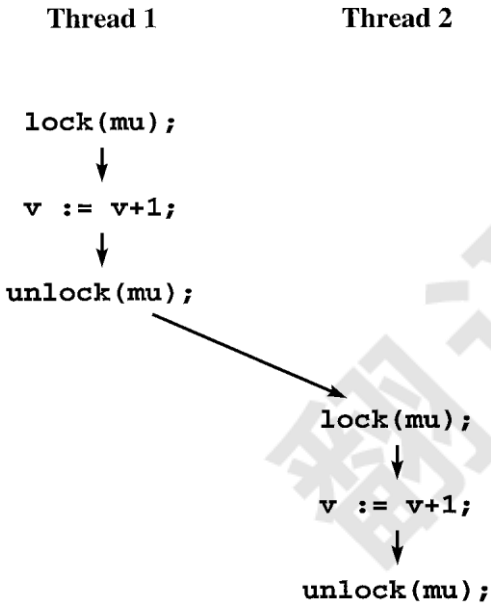
Hoare [1974]提出了一种避免数据争用的早期尝试，它是监视器的开创性概念。监视器是一组共享变量以及允许访问它们的过程，所有这些都与一个匿名锁捆绑在一起，该匿名锁在过程的入口和出口自动获取并释放。监视器中的共享变量在监视器外部超出范围（即，不可见）；因此，只能从持有锁的监视器程序中访问它们。因此，监视器提供了静态的编译时保证，可以对对共享变量的访问进行序列化，从而避免数据争用。如果所有共享变量都是静态全局变量，则监视程序是避免数据争用的有效方法，但对于具有动态分配的共享变量的程序，它们不能防止数据争用，早期用户发现这一限制很重要[Lampson and Redell 1980]。通过将动态检查替换为静态检查，我们的工作旨在允许动态分配共享数据，同时尽可能保留监视器的安全性。

已经做出了一些尝试来创建可以在动态分配的共享数据存在的情况下工作的纯静态（即，编译时）竞争检测系统：例如，Sun 的锁棉绒[SunSoft 1994]和 Modula-3 的扩展静态检查器[Detlefs 等。1997].[2] 但是这些方法似乎是有问题的，因为它们需要对程序语义进行静态推理。

以前在动态竞赛检测中的大多数工作是由科学并行编程社区完成的[Dinning and Schonberg 1990;梅洛●克鲁梅（Mellor-Crummey），1991 年；Netzer 1991;（Perkovic and Keleher 1996），并基于我们现在描述的兰莫特的事前关系。

事前发生顺序是并发执行中所有线程的所有事件的部分顺序。在任何单个线程中，事件都是按照事件发生的顺序进行排序的。在线程之间，事件根据它们访问的同步对象的属性进行排序。如果一个线程访问同步对象，并且下一次访问该对象是由另一个线程进行，则如果同步对象的语义禁止调度，则将第一次访问定义为在第二次之前进行

[2] 另请参阅扩展静态检查主页：http://www.research.digital.com/SRC/esc/Esc.html.



图。1。Lamport 的事件发生-在时间顺序上按顺序在同一线程中对事件进行排序，如果事件之间线程彼此同步，则在不同线程中对事件进行排序。

这两个交互会及时交换。例如，图 1 显示了执行同一代码段的两个线程的一种可能排序。线程 1 执行的三个程序语句按发生前的顺序排列，因为它们在同一线程中顺序执行。线程 2 对 mu 的锁定是按顺序进行的，而线程 1 对 mu 的锁定是在发生之前进行的，因为在其先前的所有者释放它之前无法获取锁定。最后，线程 2 执行的三个语句按发生前的顺序排序，因为它们在该线程中顺序执行。

如果两个线程都访问一个共享变量，并且访问没有按事前发生关系进行排序，则在程序的另一次执行中，较慢的线程运行得更快和/或较快的线程运行得更慢，这两个访问可能同时发生也就是说，无论是否实际发生了数据争用。我们知道的所有以前的动态种族检测工具都基于此观察结果。这些竞争检测器监视每个数据引用和同步操作，并检查对它们所监视的特定执行的事前关系无关的共享变量的冲突访问。

不幸的是，基于事前发生的工具有两个重大缺陷。首先，它们难以高效实现，因为它们需要有关对每个共享内存位置的并发访问的每线程信息。更重要的是，基于 beforebefore 的工具的有效性在很大程度上取决于调度程序产生的交错。

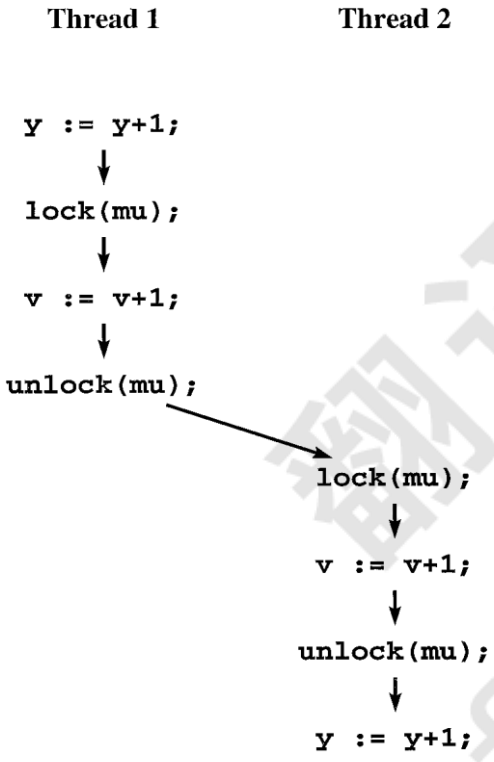


图 2。该程序允许在 y 上进行数据争用，但该错误不会被 before-before 检测到在这个执行交织中。

图 2 显示了一个简单的示例，其中事前发生方法可能会丢失数据争用。尽管对 y 的无保护访问存在潜在的数据争用，但在图中所示的执行中将不会检测到它，因为线程 1 在线程 2 之前拥有锁，因此在这种交织中对 y 的访问按顺序进行-之前。仅当调度程序产生一个交错时，线程 2 的代码片段才出现在线程 1 的代码片段之前，这种基于 beforebefore 的工具才会检测到错误。因此，有效的是，基于 beforebefore 的竞争检测器需要大量的测试用例来测试许多可能的交错。相反，在使用两个代码路径的任何测试用例下，Eraser 都将检测到图 2 中的编程错误，因为无论调度程序产生何种交错，路径都违反 y 的锁定准则。虽然橡皮擦是一种测试工具，因此不能保证程序没有种族，但与基于事前发生的工具相比，它可以检测到更多种族。

Dinning 和 Schonberg 的锁覆盖技术是对大量使用锁的程序的事前方法的一种改进[Dinning and Schonberg 1991]。实际上，描述我们的方法的一种方法是扩展 Dinning 和 Schonberg 的改进，并丢弃潜在的事前改进的手段。

2. 锁集算法

在本节中，我们将介绍 Lockset 算法如何检测种族。讨论的水平很高。下一节将介绍用于有效实现该算法的技术。

Lockset 算法的第一个也是最简单的版本强制执行简单的锁定规则，即每个共享变量都受某个锁保护，从某种意义上来说，任何访问该变量的线程都将持有该锁。橡皮擦通过监视程序执行时的所有读写来检查程序是否遵守此规则。由于 Eraser 无法知道哪些锁旨在保护哪些变量，因此它必须从执行历史中推断保护关系。

对于每个共享变量 v，Eraser 维护 v 的候选锁的集合 C（v）。该集合包含到目前为止为计算保护了 v 的那些锁。也就是说，如果在此之前的计算中，访问 v 的每个线程在访问时都持有 l，则在 C（v）中具有锁 l。初始化新变量 v 时，将其候选集 C（v）视为持有所有可能的锁。访问该变量时，橡皮擦用 C（v）与当前线程持有的一组锁的交集更新 C（v）。此过程称为锁集细化，可确保在 C（v）中包含任何始终保护 v 的锁。如果某个锁 l 始终保护 v，则在完善 C（v）时它将保留在 C（v）中。如果 C（v）为空，则表明没有锁可以始终保护 v。

总而言之，这是 Lockset 算法的第一个版本：

```
令 holds（t）为线程 t 持有的锁的集合。对于每个 v，将 C
 （v）初始化为所有锁的集合。
在线程 t 每次访问 v 时，
  设置 C（v）：= C（v）持有 n 个锁
   （t）;如果 C（v）= {}，则发出警告。
```

图 3 说明了如何通过锁集细化发现潜在的数据竞争。左列包含程序语句，从上到下依次执行。右列反映了执行每个语句后的一组候选锁 C（v）。此示例有两个锁，因此 C（v）开始包含两个锁。按住 mu1 并访问 v 后，C（v）会精炼以包含该锁。之后，仅保留 mu2 即可再次访问 v。单例集{mu1}和{mu2}的交集是空集，正确指示没有锁保护 v。

2.1 改进锁定纪律

到目前为止，我们使用的简单锁定规则太严格了。有三种非常常见的编程实践违反了学科，但没有任何数据争用：

—初始化：共享变量经常在不持有锁的情况下进行初始化。

Program	*locks_held*	*C(v)*
	{}	{mu1,mu2}
lock(mu1);		
	{mu1}	
v := v+1;		
		{mu1}
unlock(mu1);		
	{}	
lock(mu2);		
	{mu2}	
v := v+1;		
		{}
unlock(mu2);		
	{}	

图 3。如果共享变量有时受 mu1 保护，有时受锁 mu2 保护，那么在整个计算过程中，没有锁可以保护它。该图显示了 v 的候选锁 C（v）的集合的逐步细化。当 C（v）变空时，Lockset 算法检测到没有锁保护 v。

—读取共享数据：某些共享变量仅在初始化期间写入，而在其后为只读。无需锁即可安全地访问它们。

—读写锁：读写锁允许多个读取器访问共享变量，但仅允许一个写入器进行访问。

在本节的其余部分中，我们将扩展 Lockset 算法以容纳初始化和读取共享数据，然后进一步扩展它以容纳读写锁。

2.2 初始化和读取共享

如果没有其他线程可能拥有对正在访问的数据的引用，则无需线程锁定其他线程。程序员在初始化新分配的数据时通常会利用此观察结果。为了避免由于这些未锁定的初始化写入而引起的误报，我们将位置候选集的细化推迟到初始化之前。不幸的是，我们没有简单的方法知道初始化何时完成。因此，橡皮擦认为共享变量在第二个线程首次访问时要初始化。只要仅由单个线程访问变量，读和写就不会影响候选集。

由于多个线程无法同时读取共享变量，因此，如果变量是只读的，也不需要保护它。为了支持此类数据的解锁读取共享，我们仅在一个已初始化的变量被多个线程写共享后才报告竞争。

图 4 说明了状态转换，该状态转换控制何时进行锁集细化以及何时报告竞争。首次分配变量时，它将设置为 Virgin 状态，指示该数据是新数据，尚未被任何线程引用。一旦访问数据，它

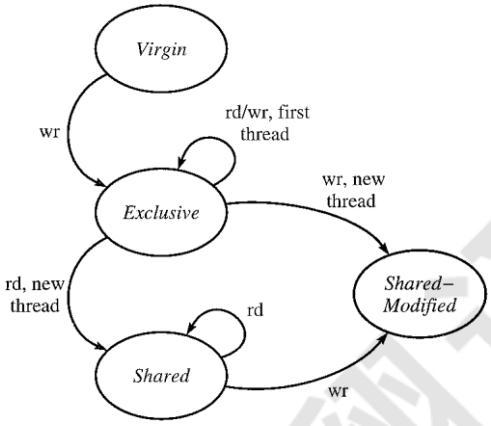


图 4。橡皮擦将所有位置的状态保存在内存中。新分配的位置从维珍州开始。当各种线程读取和写入位置时，其状态根据图中的转换而改变。仅针对处于 SharedModified 状态的位置报告竞争条件。

进入“独占”状态，表示该状态已被访问，但只能由一个线程访问。在这种状态下，同一线程的后续读取和写入操作不会更改变量的状态，也不会更新 C（v）。这解决了初始化问题，因为第一个线程可以初始化变量而不会导致 C（v）细化。当另一个线程访问该变量时，状态将发生变化。读访问将状态更改为“共享”。在共享状态下，C（v）被更新，但是即使 C（v）为空，也不报告数据竞争。由于多个线程可以读取一个变量而不会导致报告争用，因此可以解决共享数据读取问题。正如该算法的原始简单版本所述，来自新线程的写访问将状态从“独占”或“共享”更改为“共享修改状态”，在该状态下 C（v）被更新并报告了竞争。

我们对初始化的支持使 Eraser 的检查比我们希望的更加依赖调度程序。假设线程在没有锁定的情况下分配并初始化了共享变量，并且错误地使该变量在完成初始化之前可供第二个线程访问。然后，如果在第一个线程的最终初始化操作之前发生了第二个线程的任何访问，则 Eraser 将检测到该错误，但否则，Eraser 会丢失该错误。我们认为这不是问题，但是我们无法确定。

2.3 读写锁

许多程序使用单写，多读锁以及简单锁。为了适应这种风格，我们介绍了锁学科的最后改进：我们要求对于每个变量 v，都有一些锁 m 保护 v，这意味着对于 v 的每次写入，m 都处于写模式，而 m 处于某种模式（读或每次写入）。

我们继续使用图 4 的状态转换，但是当变量进入 Shared-Modified 状态时，检查略有不同：

令 holds（t）为线程 t 在任何模式下持有的锁的集合。
令 hold（t）的写锁为线程 t 在写模式下保持的锁的集合。对于每个 v，将 C（v）初始化为所有锁的集合。
在线程 t 每次读取 v 时，
  设置 C（v）：= C（v）持有 n 个锁（t）；如果 C（v）：= {}，则发出警告。
在线程 t 对 v 的每次写入中，
  设置 C（v）：= C（v）持有 n 个写锁（t）；
  如果 C（v）= {}，则发出警告。

就是说，发生写操作时，将纯保留在读取模式下的锁从候选集中删除，因为由写程序持有的此类锁无法防止写程序与某些其他读程序线程之间的数据竞争。

3. 实施橡皮擦

使用 ATOM [Srivastava and Eustace 1994]二进制修改系统在 Alpha 处理器上为数字 Unix 操作系统实现了橡皮擦。橡皮擦将未经修改的程序二进制文件作为输入，并添加工具以生成功能相同的新二进制文件，但包括对橡皮擦运行时的调用以实现 Lockset 算法。

为了维持 C（v），橡皮擦会检测每个加载并存储在程序中。为了维持每个线程 t 的锁保持状态（t），Eraser 会检测每个调用以获取或释放锁，以及管理线程初始化和完成的存根。要为动态分配的数据初始化 C（v），Eraser 会对存储分配器的每个调用进行检测。

橡皮擦将堆或全局数据中的每个 32 位字视为可能的共享变量，因为在我们的平台上，32 位字是最小的内存一致性单元。橡皮擦不会检测加载和存储其地址模式是间接指向堆栈指针的负载，因为它们被假定为堆栈引用，并且共享变量被假定为位于全局位置或堆中。橡皮擦将为通过堆栈指针以外的寄存器访问的堆栈位置保留候选集，但这只是实现的产物，而不是经过精心设计的计划，以支持在线程之间共享堆栈位置的程序。

当报告比赛时，Eraser 指示发现该文件的文件和行号以及所有活动堆栈帧的回溯列表。该报告还包括线程 ID，内存地址，内存访问类型以及重要的寄存器值（例如程序计数器和堆栈指针）。当与程序的源代码结合使用时，我们发现该信息通常足以确定种族的起源。如果仍然不清楚比赛的原因，则用户可以指示橡皮擦记录对特定变量的所有访问，从而导致其候选锁定集发生更改。

3.1 代表候选锁集

天真的锁集实现将为每个内存位置存储候选锁列表，这可能会消耗程序分配的内存很多倍的时间。我们可以利用幸运的事实避免这种花费，该事实是在实践中观察到的不同的锁组数量非常少。实际上，在执行 Lockset 监视算法时，我们从未观察到超过 10,000 个不同的锁集。因此，我们用一个小的整数表示每个锁集，这是表中的锁集索引，该表的条目将锁集规范地表示为锁地址的排序向量。该表中的条目永远不会被释放或修改，因此每个锁集索引在程序的生存期内均保持有效。

由于获取锁，释放锁或通过应用交集操作而创建了新的锁集索引。为了确保每个锁集索引代表一组唯一的锁，我们维护了完整锁向量的哈希表，该哈希表在创建新的锁集索引之前就进行了搜索。橡皮擦还缓存每个相交的结果，因此设置相交的快速情况只是简单的表查找。表中的每个锁定向量都经过排序，因此当缓存失败时，可以通过简单比较两个排序后的向量来执行相交操作的慢速情况。

对于数据段和堆中的每个 32 位字，都有一个对应的影子字，用于包含 30 位锁集索引和 2 位状态条件。在互斥状态下，这 30 位不用于存储锁集索引，而是用于存储具有互斥访问权限的线程的 ID。

所有标准的内存分配例程都可以为程序分配的每个单词分配和初始化一个影子单词。当线程访问内存位置时，橡皮擦通过在该位置的地址中添加固定位移来找到阴影字。图 5 说明了影子存储器和锁集索引表示如何用于将每个共享变量与一组对应的候选锁相关联。

3.2 性能

性能不是我们实施橡皮擦的主要目标，因此有很多优化的机会。使用橡皮擦时，应用程序的速度通常会降低 10 到 30 倍。此时间扩展可以更改线程的调度顺序，并且可以影响对时间敏感的应用程序的行为。我们的经验表明，线程调度中的差异对 Eraser 的结果影响很小。我们对时间紧迫的应用程序缺乏经验，因此它们可能会受益于更有效的监视技术。

我们估计当前实现速度减缓的一半是由于在每个加载和存储指令处进行过程调用的开销所致。通过使用可以内嵌监视代码的 ATOM 版本，可以消除这种开销。1996]。还有很多

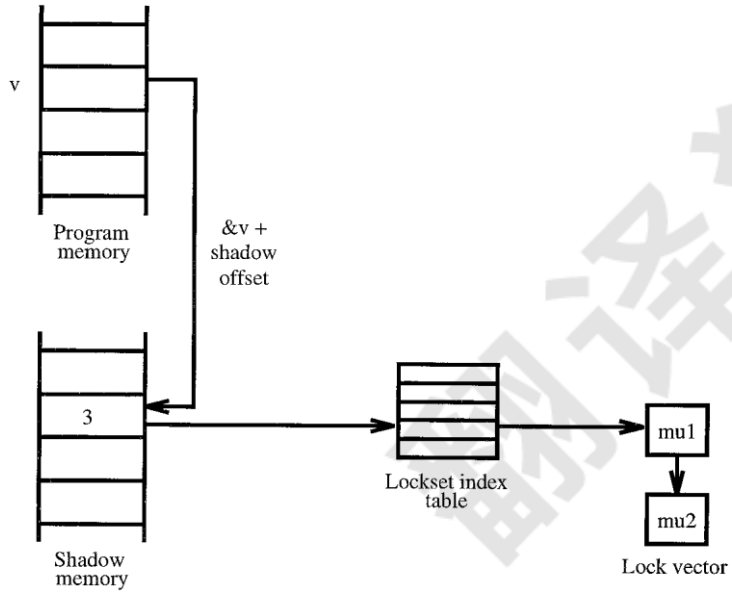


图 5。橡皮擦通过将变量的地址添加到固定的影子内存偏移量来将锁集索引与每个变量关联。索引又从锁集索引表中选择一个锁向量。在这种情况下，共享变量 v 与包含 mu1 和 mu2 的一组锁相关联。

使用静态分析来减少监视代码开销的机会；但是我们还没有探索它们。

尽管性能调整有限，但我们发现 Eraser 的速度足以调试大多数程序，因此符合最基本的性能标准。

3.3 程序注释

不出所料，我们在 Eraser 上的经验表明它会产生错误的警报。我们研究的一部分旨在找到有效的注释，以抑制错误警报，而不会意外丢失有用的警告。这是使像橡皮擦这样的工具有用的关键。如果使用准确和特定的注释抑制了误警报，则在修改程序并测试修改后的程序时，只会产生新的相关警告。

根据我们的经验，错误警报主要分为三大类：

-内存重用：报告了错误警报，因为在不重置影子内存的情况下重用了内存。橡皮擦会检测所有标准 C，C ++和 Unix 内存分配例程。但是，许多程序实现了空闲列表或私有分配器，并且 Eraser 无法知道私有回收的内存是否受到一组新锁的保护。

—专用锁定：报告错误警报，因为在运行时未将信息传递给橡皮擦而进行了锁定。这通常是由多个读取器/单个写入器锁的私有实现引起的，这些实现不属于橡皮擦提供的标准 pthreads 接口。

—良性竞赛：发现了不影响程序正确性的真实数据竞赛。其中一些是故意的，而其他则是偶然的。

对于这些类别中的每一个，我们开发了程序注释，以允许 Eraser 的用户消除虚假报告。对于良性种族，我们添加了

```
EraserIgnoreOn ()
EraseRigRoOOF ()
```

告知比赛检测器它不应在方括号代码中报告任何比赛。为了防止报告内存重用竞赛，我们添加了

```
EraserReuse (地址，大小)
```

指示橡皮擦将与指示的内存范围相对应的影子内存重置为 Virgin 状态。最后，可以通过用以下方式注释私有锁实现的存在：

```
EraserReadLock (锁定)
EraserReadUnlock (锁定)
EraserWriteLock (锁定)
EraserWriteUnlock (锁定)
```

我们发现，这些注释中的少数通常足以消除所有错误警报。

3.4 操作系统内核中的竞赛检测

我们已经开始修改橡皮擦，以检测 SPIN 操作系统中的种族[Bershad 等。1995]。SPIN 的许多功能（例如运行时代码生成和后期代码绑定）使检测过程变得复杂，因此 Eraser 在此环境中尚无法使用。尽管如此，尽管我们没有找到关于数据竞争的结果，但我们已经获得了一些在内核级别实现这种工具的有用经验，在某些方面与用户级别有所不同。

首先，SPIN（与许多操作系统一样）通常会提高处理器中断级别，以相互排斥设备驱动程序和其他中断级别代码访问的共享数据结构。在大多数系统中，将中断级别提高到 n 可确保只有优先级大于 n 的中断才能得到服务，直到降低中断级别为止。可以使用升高然后恢复的中断级别代替锁定，如下所示：

```
级别：= SetInterruptLevel (n) ;
 (*处理数据*)
RestoreInterruptLevel (level) ;
```

但是，与锁不同的是，特定的中断级别可保护所有受较低中断级别保护的数据。通过为每个单独的中断级别分配一个锁，我们将这种差异纳入了 Eraser。当内核将中断级别设置为 n 时，Eraser 将该操作视为已获取了前 n 个中断锁。我们希望该技术能够使我们使用标准锁来检测代码之间的竞争，并使用中断级别来检测代码之间的竞争。

另一个区别是操作系统更多地使用了发布/等待样式同步。最常见的示例是使用信号量在线程和 I / O 设备驱动程序之间同步执行。接收到数据后，设备驱动程序将执行一些最少的处理，然后使用 V 操作向等待 P 操作的线程发出信号，例如，唤醒等待 I / O 完成的线程。如果在设备驱动程序和线程之间共享数据，这可能会导致橡皮擦出现问题。因为信号量不是“所有者”，所以橡皮擦很难推断出它们正在保护哪些数据，从而导致发出错误警报。集成线程和中断处理的系统[Kleiman and Eykholt 1995]对该问题的麻烦可能较小。

4. 经验

我们在许多简单的程序上校准了橡皮擦，这些程序包含常见的同步错误（例如，忘记锁定，使用了错误的锁定等）以及已纠正错误的那些程序的版本。在编写这些测试的程序时，我们意外地引入了一个种族，令人鼓舞的是，Eraser 探测到了它。这些简单的测试对于在 Eraser 中查找错误非常有用。在确信该工具有效之后，我们解决了一些由 Digital Equipment Corporation 系统研究中心的资深研究人员编写的大型多线程服务器：AltaVista 的 HTTP 服务器和索引引擎，Vesta 高速缓存服务器以及 Petal 分布式磁盘系统。我们还将 Eraser 应用于华盛顿大学的本科程序员编写的一些家庭作业问题。

如下所述，Eraser 在四个服务器程序中的三个中以及在许多本科作业中发现了不良的比赛条件。它还产生了虚假警报，我们可以通过注释来消除它。当我们发现比赛状况或错误警报时，我们适当地修改了程序，然后重新运行橡皮擦以查找剩余的问题。此过程的十次迭代通常足以解决程序报告的所有争用。

我们在其上测试过 Eraser 的服务器的程序员并不是从测试 Eraser 甚至使用 Eraser 的锁定规则的计划开始的。橡皮擦在服务器上运行良好的事实证明了

程序员即使在提供更多复杂同步原语的环境中也倾向于遵循简单的锁定规则。

在本节的其余部分，我们将报告每个程序的经验详细信息。

4.1 阿尔塔维斯塔

我们检查了流行的 AltaVista Web 索引服务的两个组件：mhttpd 和 Ni2。[3]

mhttpd 程序是一个轻量级的 HTTP 服务器，旨在支持 AltaVista 承受的极高的服务器负载。每个搜索请求由一个单独的线程处理，并依靠锁定来同步并发请求对共享数据结构的访问。另外，mhttpd 使用几个其他线程来管理后台任务，例如配置和名称缓存管理。该服务器包含大约 5000 行 C 源代码。我们通过从三个单独的 Web 浏览器调用一系列测试脚本来测试了 mhttpd。mhttpd 测试使用了大约 100 个不同的锁，这些锁形成了大约 250 个不同的锁集。

Ni2 索引引擎用于响应索引查询来查找信息。索引数据结构在所有服务器线程之间共享，并显式使用锁来确保更新被序列化。Ni2 基本库包含大约 20,000 行 C 源代码。我们使用名为 ft 的实用程序单独测试了 Ni2，该实用程序使用指定数量的线程（我们使用 10 个线程）提交一系列随机请求。ft 测试使用了大约 900 个锁，形成了大约 3600 个不同的锁组。

我们发现了许多已报告的种族，其中大多数被证明是错误的警报。这些主要是由内存重用引起的，其次是私有锁和良性竞争。在 Ni2 中发现的良性环特别有趣，因为它们证明了有意使用环以减少锁定开销。例如，考虑以下代码片段：

```
如果 (p3ip fp == (NI2 XFILE *) 0) {          //是否设置了文件指针？NI2
   LOCKS LOCK (&p3ip 锁定) ;                 //不？如果 (p3ip fp ==
    ( (NI2 XFILE *) 0) {                     //已设置文件指针
                                             //自上次检查以来？
     P3IP FP= NI2xFOpenT (                   //不？设置文件指针
                p3ip 名称，“ rb”) ;
   }
   NI2 LOCKS UNLOCK (&p3ip 锁定) ;
}
                    ...                      //如果文件没有锁定开销
                                             //指针已设置
```

在此代码段中，在未锁定的情况下测试了 ip fp 字段，这与其他线程创建了数据竞争，其他线程使用

[3]http://altavista.digital.com/

ip 锁定已锁定。为了避免在已设置 ip fp 的常见情况下锁定开销，对竞争进行了有计划的优化编程。即使在竞争中，该程序也是正确的，因为 ip fp 字段在多线程范围内从不会从非零转换为零，并且在该字段测试为零的情况下，程序在锁内重复测试（因此避免了竞争）。两个线程找到字段零，然后都对其进行初始化）。

这种代码非常棘手。例如，在其余过程中访问 p3ip fp 字段似乎很安全（用代码段中的省略号代替了这些行）。但这实际上是一个错误，因为 Alpha 的内存一致性模型允许处理器在没有中间同步的情况下看到内存操作混乱。尽管 Ni2 代码是正确的，但在使用 Eraser 之后，程序员决定对其进行重新编程，以便其正确性参数更简单。

我们还在 Ni2 测试工具程序中发现了一个良性竞争，在该竞争中，多个线程在读写一个称为终止查询的全局变量时发生竞争。此变量初始化为 false，并设置为 true 指示所有线程都应退出。每个线程都会定期轮询变量，并在将其设置为 true 时退出。其他敲定代码也有类似的良性竞争。为了防止比赛检测器报告此类比赛，我们使用了 EraserIgnoreOn / Off () 批注。同样，在定期更新全局配置数据和统计信息时，mhttpd 会忽略锁定。这些确实是同步错误，但是它们的影响相对较小，这也许就是为什么长时间未发现它们的原因。

在 Ni2 库中插入 9 个注释，在 ft 测试工具中插入 5 个注释，在 mhttpd 服务器中插入 10 个注释，从而使报告的种族数从一百多个减少到零。

4.2 Vesta 缓存服务器

Vesta 是先进的软件配置管理系统。[4] 用特殊的功能语言编写的配置描述了用于导出软件当前状态的依赖性和规则。由 C 编译器生成的部分结果（例如“ .o”文件）将缓存在 Vesta 缓存服务器中，并由 Vesta 构建器用来创建特定的配置。缓存服务器由大约 30,000 行 C ++代码组成。我们使用 TestCache 实用工具测试了缓存服务器，该工具发出并发随机请求流。高速缓存服务器使用 10 个线程，获取 26 个不同的锁，并实例化 70 个不同的锁集。在测试缓存服务器时，Eraser 报告了许多竞赛，其中大多数围绕三种数据结构。在维护高速缓存条目中的指纹的代码中检测到第一组种族。因为计算指纹可能很昂贵，所以缓存服务器在缓存条目中维护一个布尔字段，以记录指纹是否有效。仅在需要其真实值和当前值时才计算指纹

[4] http://www.research.digital.com/SRC/vesta/

值无效。不幸的是，布尔值是在没有保护锁的情况下访问的，如下所示：

```
结合 XorFPTag FPVal () {
  如果 (! this3validFP) {                    //指纹被标记为有效吗？
                                             //不？计算指纹 NamesFP (fps, bv,
    this3fp, imap) ;                         // (NamesFP 更改 this3fp)
    this3validFP = true;                     //并将其标记为有效
  }
  返回 this3fp;
}
```

这是一场严重的数据竞赛，因为在没有存储屏障的情况下，Alpha 语义不能保证有效 FP 字段的内容与 fp 字段一致。

另一组竞赛围绕 CacheS 对象中的空闲列表进行。CacheS 对象维护各种日志条目的免费列表。我们的第一个响应是使用 EraserReuse () 批注，其中元素从此空闲列表中分配。但是，这并没有使所有警告消失。刷新日志的调用仍然引起种族。检查发现，每个日志的头部均受锁保护，但不受单个条目的保护。刷新例程锁定日志的开头，将其值存储在堆栈变量中，将日志头设置为 0，然后释放锁定。此后，他们将访问未锁定的单个条目，最终将它们放入空闲列表。这是正确的，因为其他线程在持有日志头锁的情况下访问日志条目，并且线程不维护指向日志的指针。因此，Flush 有效地使数据对调用 Flush 的线程私有。通过将 EraserReuse () 注释移到三个 Flush 例程中，我们消除了这些比赛的报告。

最后，还有一些与 TCP 套接字和 SRPC 对象有关的错误警报，这些警报用于实现服务器端 RPC。缓存服务器使用主服务器线程来等待传入的 RPC 请求。收到请求后，此线程将当前套接字和 RPC 数据结构传递给负责处理其余 RPC 的工作线程。由于主线程和工作线程永远不会同时访问数据结构，因此它们不需要使用锁来序列化访问。对于橡皮擦来说，这似乎违反了锁定规则，被标记为种族。通过一些努力，可以修改 Eraser 以识别此锁定规则，但是我们可以通过两个 EraserReuse () 注释实现相同的效果。

总共 10 条注释和 1 条错误修复程序足以将比赛报告从数百减少到零。

4.3 花瓣

Petal 是一个分布式存储系统，它为客户端提供由服务器和物理磁盘群集实现的巨大虚拟磁盘[Lee and Thekkath 1996]。Petal 实现了分布式共识算法以及故障检测和恢复机制。花瓣服务器是

大约 25,000 行 C 代码，我们在测试中使用了 64 个并发工作线程。我们使用发出随机读写请求的实用程序测试了 Petal。

我们发现了由私有读写器锁定实现引起的许多误报。使用注释很容易抑制这些。我们还在例行程序 GMapCh CheckServerThread（）中检测到真实竞赛。该例程由单个线程运行，并定期检查以确保相邻服务器正在运行。但是，这样做时，它会读取 gmap3state 字段而不保留 gmapState 锁（所有其他线程在写入 gmap3state 之前均会保留）。

我们发现了两个种族，其中包含统计信息的全局变量被修改而没有锁定。这些竞赛是有意为之的，其前提是锁定昂贵且服务器统计信息仅需大致正确。

最后，我们发现了一个无法注释的错误警报。函数 GmapCh Write2（）分叉多个线程，并将每个引用传递给 GmapCh Write2 的堆栈框架的组件。GmapCh Write2（）实现了类似连接的构造，以保持堆栈帧处于活动状态，直到线程返回为止。但是，Eraser 不会为每个新的堆栈帧重新初始化影子内存。因此，对于堆栈帧的不同实例重复使用堆栈存储器会导致错误警报。

4.4 本科课程

作为对我们使用成熟的多线程服务器程序的经验的反击，华盛顿大学的两位同事使用橡皮擦检查了由他们的本科操作系统班级产生的家庭作业中发现的同步错误的种类（个人通信，SE Choi 和 EC 刘易斯，1997 年）。我们在这里报告其结果，以说明橡皮擦如何以不太复杂的代码库运行。

该类是完成四个标准多线程分配所必需的。这些分配大致可分为低级（通过测试和设置构建锁），线程级（构建小型线程包），同步级（构建信号量和互斥体）和应用程序级（生产者/消费者风格）。问题）。每个作业都基于先前作业的实施。我们的同事使用橡皮擦检查了大约 40 个小组的每一项任务。总共约
上交了 100 个可运行的分配（并非所有组都完成了所有分配；一些未编译；一些立即陷入僵局）。在这些“工作”任务中，有 10%具有橡皮擦发现的数据竞争。这些是由于忘记获取锁，在写入过程中获取锁而不是在读取过程中，使用不同的锁在不同时间保护相同的数据结构以及忘记重新获取在循环中释放的锁引起的。

橡皮擦还报告了一个错误警报，该警报由队列触发，该队列通过锁定的头和尾字段（类似于 Vesta 的 CacheS 对象）访问该队列，从而隐式保护了元素。

4.5 有效性和敏感性

由于 Eraser 使用测试方法，因此无法证明程序没有数据争用。但是我们认为，与手动测试和调试相比，橡皮擦的效果很好，并且橡皮擦的测试对调度程序的交错不是很敏感。为了检验这些信念，我们进行了另外两个实验。

我们查阅了 Ni2 的程序历史，并重新引入了以前版本中存在的两个数据竞赛。第一个错误是对用于垃圾收集文件数据结构的引用计数的未锁定访问。另一个竞争是由于未能获得保护在大型过程中间调用的子例程的数据结构所需的附加锁而引起的。这些种族在 Ni2 源代码中已经存在了几个月，然后才被程序作者手动找到并修复。使用橡皮擦，我们中的一个人可以在几分钟内找到两个种族，而没有任何关于种族在哪里或如何引起种族的信息。花了 30 分钟纠正了两个错误并确认没有其他比赛报告。

我们通过重新运行 Ni2 和 Vesta 实验来检查敏感性问题，但仅使用两个并发线程而不是 10。如果橡皮擦对线程交织的差异敏感，那么我们将期望找到一组不同的比赛报告。实际上，我们使用两个线程或 10 在多个运行中发现了相同的比赛报告（尽管有时顺序不同）。

5. 额外的经验

在本节中，我们简要介绍另外两个主题，每个主题都涉及一种动态检查多线程程序中同步错误的形式，我们尝试过并认为这是重要且有希望的，但是我们并未在 Eraser 中实现。

第一个主题是多重锁定保护。一些程序通过多个锁而不是单个锁来保护某些共享变量。在这种情况下，规则是：每个写入变量的线程必须持有所有保护锁，而读取变量的每个线程必须至少持有一个保护锁。仅当两个访问均被读取时，此策略才允许一对同时访问，因此可以防止数据争用。

在某些方面，使用多个保护锁与使用读取器锁和写入器锁相似，但是它并不是为了增加并发性，而是为了避免在包含上调用的程序中避免死锁。

使用早期版本的 Eraser 在多线程 Modula-3 程序中检测竞争状况，我们发现 Lockset 算法报告了 Trestle 程序的错误警报[Manasse and Nelson 1991]，该程序使用多个锁来保护共享位置，因为两个读取器中的每一个

可以同时按住两个不同的锁来访问该位置。作为实验，我们通过修改 Lockset 算法以仅针对写入精简候选集来解决该问题，同时检查了读取和写入双方，如下所示：

```
在线程 t 每次读取 v 时，
  如果 C（v）= {}，则发出警告。在线
程 t 对 v 的每次写入中，
  设置 C（v）：= C（v）持有 n 个锁
  （t）;如果 C（v）= {}，则发出警告。
```

这样可以防止误报，但是此修改有可能导致误报。例如，如果线程 t_1 在持有锁 m_1 的同时读取 v，而线程 t_2 在持有锁 m_2 的同时写入 v，则仅当写入时才会报告违反锁定准则在读取之前。通常，仅当测试用例导致足够的共享变量读取以跟随相应的写入时，修改后的版本才能做好工作。

从理论上讲，可以处理多个保护锁而没有任何假否定的风险，但是所需的数据结构（一组锁集而不是仅一组锁）似乎在复杂性方面的代价与可能的代价不成比例。获得。由于我们对错误否定不满意，并且由于多重保护锁定技术并不普遍，因此当前版本的 Eraser 会忽略该技术，从而为使用该技术的程序产生错误警报。

第二个话题是僵局。如果数据争用是 Scylla，则死锁是 Charybdis。

避免死锁的一种简单规则是在所有锁中选择部分顺序，并对每个线程进行编程，以便每当持有多个锁时，它就以升序获取它们。该规则类似于避免数据竞争的锁定规则：它适合于通过动态监视进行检查，并且生成暴露该规则违规的测试用例要比生成实际导致安全隐患的测试用例要容易得多。僵局。

对于一个独立的实验，我们选择了一个大型的 Trestle 应用程序，该应用程序具有复杂的同步性（formsedit，一个双视图用户界面编辑器），记录了所有锁的获取情况，并进行了测试，以查看锁中是否存在命令。每个线程都尊重。进入 Formsedit 启动后的几秒钟，我们的实验监视器检测到一个锁周期，表明不存在部分顺序。仔细检查周期，发现 formedit 可能存在死锁。我们认为这是一个有希望的结果，并且可以推测，按照这些思路进行死锁检查将对橡皮擦有用。但是需要更多的工作来对偏序规则中的声音和有用的变化进行分类，并开发注释以抑制错误警报。

6. 结论

硬件设计师已学会设计以实现可测试性。使用线程的程序员必须学习相同的知识。编写正确的文字还不够

程序;正确性必须是可证明的，理想情况下是通过静态检查证明的，实际上是通过部分静态检查和严格的动态测试的组合来证明的。

本文介绍了在使用许多不同同步原语的通用并行程序中强制执行简单锁定规则而不是检查竞争的优点，并证明了使用此技术动态检查生产多线程程序的数据竞争是可行的。

操作系统领域的程序员似乎将动态种族检测工具视为深奥且不切实际的。我们的经验使我们相信，它们是避免数据争用的实用且有效的方法，并且动态争用检测应该是多线程程序的任何严格测试工作中的标准过程。随着多线程使用的扩展，除非使用更好的方法消除数据竞争，否则由数据争用引起的不可靠性也会随之增加。我们认为，在 Eraser 中实现的 Lockset 方法很有希望。

致谢

我们要感谢以下个人对这个项目的贡献。崔成恩和克里斯托弗 ● 刘易斯（E. Christoper Lewis）负责所有本科实验。Alan Heydon，Dave Detlefs，Chandu Thekkath 和 Edward Lee 提供了有关 Vesta 和 Petal 的专家建议。Puneet Kumar 致力于早期版本的 Eraser。Cynthia Hibbard，Brian Bershad，Michael Ernst，Paulo Guedes，Wilson Hsieh，Terri Watson 以及 SOSP 和 TOCS 审阅者对本文的早期草案提供了有用的反馈。

参考资料

bershad，bn，savage，s。，pardyak，p。，先生，例如 fiuczynski，m。，becker，d。，eggers，s。，
和钱伯，C。1995。SPIN 操作系统中的可扩展性，安全性和性能。在第 15 届 ACM 操作系统原理研讨会论文集中（科罗拉多州，Copper Mountain，12 月）。ACM，纽约，267-284。

DETLEFS，DL，LEINO，RM，NELSON，G.，和 SAXE，JB 1997。扩展的静态检查。
科技共和国水库加利福尼亚州帕洛阿尔托市数字设备公司系统研究中心报告 149。

丁宁，A。和肖恩伯格，1990 年。接入异常检测监视算法的经验比较。在第二届 ACM SIGPLAN 关于并行编程的原理和实践的研讨会论文集中（西雅图，华盛顿，3 月）。纽约 ACM，1-10。

丁 *宁，A。和肖恩伯格，E.，1991 年。在具有关键部分的程序中检测访问异常。在 ACM / ONR 关于并行和分布式调试的研讨会论文集中。ACM SIGPLAN 不是。26、12（十二月），85-96。*

霍尔（1974）。监视器：一种操作系统结构化概念。公社 ACM 17，10（十月），549 –557。

KLEIMAN，S. and EYKHOLT，J.1995 年。作为线程中断。美国运通公司系统。Rev. 29，2（Apr.），21–26.

LAMPORT，L.1978 年。时间，时钟和分布式系统中事件的顺序。
公社 ACM 21，7（7 月），558 –565。

兰普森（B.Lampson）和丹德尔（REDELL），1980 年。在 Mesa 中具有流程和监视器的经验。公社 ACM 23，2（2 月），104 –117。

李 埃克（LEE EK）和瑟卡斯（THEKKATH），加利福尼亚，1996 年。花瓣：分布式虚拟磁盘。在第七届国际编程语言和操作系统的体系结构支持国际会议论文集（马萨诸塞州剑桥，十月）上。纽约 ACM，84-93。

玛纳瑟斯（MS）和纳尔逊（G. NELSON），1991 年。栈桥参考手册。物件。加利福尼亚州帕洛阿尔托市数字设备公司系统研究中心报告 68。

梅勒-克鲁米（Mellor-CRUMMEY），1991 年。带有嵌套的 forkjoin 并行性的程序的数据竞争的实时检测。在 1991 年超级计算机调试研讨会论文集（新墨西哥州阿尔伯克基，11 月）中。1–16.

梅勒-克鲁梅（Mellor-Crummey），J.，1993 年。编译时支持在共享内存并行程序中进行有效的数据竞争检测。在 ACM / ONR 关于并行和分布式调试的研讨会论文集（加利福尼亚，圣地亚哥，5 月）中。纽约 ACM，129 –139。

NETZER，RHB，1991 年。竞争条件检测，用于调试共享内存并行程序。博士大学论文威斯康星州麦迪逊市威斯康星州麦迪逊分校。

PERKOVIC，D.和 KELEHER，P. 1996。通过一致性保证进行在线数据竞赛检测。在第二届 USENIX 操作系统设计和实现座谈会的会议记录中（华盛顿特区，十月）。USENIX 协会，加利福尼亚州伯克利，47-58。

SCALES，DJ，K.GHARACHORLOO，K.，THETHEKATH，CA，1996。沙斯塔：开销低，仅软件的方法，用于支持细粒度的共享内存。在第七届国际编程语言和操作系统支持国际会议论文集（马萨诸塞州剑桥，十月）上。纽约 ACM，174 –185。

SRIVASTAVA，A。和 EUSTACE，A。1994。ATOM：一种用于构建定制程序分析工具的系统。在 1994 年 ACM SIGPLAN 编程语言设计与实现会议的会议记录中（佛罗里达州奥兰多，6 月）。纽约 ACM，196 –205。

SunSoo.1994 年。锁定皮棉用户指南。SunSoft 手册，Sun Microsystems，Inc.，加利福尼亚州帕洛阿尔托。

1997 年 7 月收到；1997 年 9 月修订;1997 年 9 月接受