Detecting and Surviving Data Races using Complementary Schedules

Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy
University of Michigan
kaushikv,pmchen,jflinn,nsatish@umich.edu

摘要

数据竞赛[1]在多线程程序中是错误的一种普遍的来源。在这篇论文中，我们展示在运行中，如何通过执行多个带着补充线程调度[2]的程序的副本来防止程序出现由数据竞赛而产生的错误。补充调度是一个线程副本的集合，他们保证副本只有在数据竞赛发生时才会偏离，而且基本上只有恶性的数据竞赛才会导致偏离。我们的系统，名为Frost，使用补充调度来使至少一个副本会避免会导致错误程序运行和最恶性的数据竞赛的指令顺序。Frost引入了基于收益的竞赛观测，它通过比较副本执行补充调度的状态来检测数据竞赛。我们会展示这种方法比当前的针对非托管代码的动态数据竞赛的检测法要快得多。为了帮助程序在生产中免除故障，Frost也会诊断数据竞赛的故障并采取适当的恢复策略，比如选取一个看上去是对的副本，或者执行更多的副本来获取更多的信息。

Frost通过在一个单核上非抢占式地运行一个部分的所有线程在控制部分的线程调度。为了将程序规模扩展到多核，Frost 并行地运行第三个副本来产生程序将来可能的检查点，这些检查点使Frost能将程序执行分成几个时段，然后即可让它们并行执行。

我们用11个在桌面或者服务器应用环境下的真实的数据竞赛故障在测评Frost。Frost将这些故障全部检测并且度过了。因为Frost运行了3个副本，因为它的效用比是3x。不过，如果有空闲的核心来吸收这些额外的负荷的话，Frost只增加了$3-12\%$的开销。

分类描述

D.4.5 [Operating S ystems]: Reliability; D.4.8[Operating Systems]: Performance

着眼点

设计，表现，可靠性

关键词

数据竞赛检测，数据竞赛生还，单并行

[1]Data race
[2]complementary schedule

1 引言

多核处理器的流行鼓励了多线程程序在多个领域的应用，比如科学计算，网络服务器，桌面应用，和移动终端。不幸的是，多线程软件对于由数据竞赛，即两个线程中的两条指令（至少有一条是写）未经同步操作使用了同一个内存地址，而产生的故障十分脆弱。许多并行的故障都来源于数据竞赛。这些故障在生成中很难发现，并且可能会在运行中导致崩溃或者其他错误。由竞赛导致的错误可能是灾难性的，由2003年东北美的灯火熄灭和Therac-25的辐射超标即可见一斑[23,25]。

研究者们已经就在生产中消除或者检测数据竞赛提出了各种各样的方法，如纪律预言[5,7]，静态竞赛分析[13]，和动态竞赛分析[15,40,42]。尽管有这些努力，数据竞赛还是在折磨生产中的代码，并且是崩溃，错误，入侵的一个主要来源。

为了帮助解决数据竞赛故障的问题，我们提出运行一个程序的多个副本并且强迫两个部分执行补充调度。我们使用补充调度的目的是强迫副本只有在出现潜在的恶性数据竞赛时才产生偏离。我们通过利用在可能的进程调度中的一个最佳点来做到这一点。首先，我们使所有的副本输入相同的输入并使用遵从由同步时间和系统调用所引发的同一个程序顺序限制的线程调度。这保证了不会执行一对竞赛指令的副本不会偏离[37]。接着，在遵从前一个限制的前提下，我们试着让2个副本执行的线程调度尽可能的不同。就是说，我们试着去最大化二条在不同线程没有被同步指令或者系统调用排序过的指令在两个副本中按不同顺序执行的可能性。对于我们研究过的所有的在真是应用中的恶性数据竞赛，这个策略导致了副本偏离。

我们的系统Frost，使用补充调度来打到两个目的：低成本

地检测数据竞赛，和通过在运行时消除恶性数据竞赛的效果来增加程序的可行性。Frost引入了一个新的方法来检测竞赛：基于输出的数据竞赛检测。传统的方法通过分析程序引发的事件来检测数据竞赛，而基于利益的竞赛检测检测的是一个数据竞赛的效果，这点是通过比较不同的执行补充调度的副本的状态得到的。基于输出的竞赛检测获得了更低的能耗，但它可能会检测不到一些竞赛，比如，一些需要中断来引发一个错误的数据竞赛或者对多个无中断的调度都生成同样正确的输出的（完整地论述在3.4节）。不过，在我们对真实程序的测评中，Frost检测出了所有的潜在的那些被传统的检测器检测出的恶性数据竞争。虽然先前的工作[31]比较了多个指令排序的输出来分辨这些数据竞赛是无害的或者有害的，Frost是构造多个调度来检测和生还未知数据竞赛的第一个系统。Frost因此面临必须针对未知的竞赛来构造调度。Frost从早先的分类工作中继承的一个优势是他自动地过滤了大部分被传统动态竞赛检测器认为无害的竞赛。

对生产系统，Frost不但能够检测，还能够检测和生还恶性的数据竞赛。因为一个由数据竞赛产生的并行故障，只会在一个特定的内存读取顺序下，执行补充调度能在很大程度上使得某一个副本躲过了数据竞赛的恶性作用。因此，一旦Frost发现了一个数据竞赛，他分析不同副本的输出，并采取一个最有可能掩盖掉错误的调度，比如分辨并继续执行正确的副本，或者增加副本来获得一个正确的副本。

为了生成补充线程调度，Frost必须严格控制每个副本的执行。为了做到这点，Frost把一个副本的线程时间切片到一个单核上，并只在同步时间点切换线程（使用无中断的调度）。在一个单核上运行无中断的各个线程还有一个好处：它防止了那些在优先权下的故障（如原子冲突），因此增加了可靠性。因为在单核上跑所有线程防止了某个副本扩展到多核，Frost使用单并行[44]，通过在多核上同时运行那个程序的多个时间片来并行化一个单核的执行。

Frost帮助解决多个环境下的数据竞赛问题。在测试中，它能充当一个快速动态数据竞赛检测器的角色，分辨出一个数据竞赛是无害的还是有潜在威胁的。对于一个测试或者处于生产中的有活跃用户的系统，检测和可靠性都是重要的目标。Frost遮盖了许多数据竞赛的错误，同时给开发者提供了可能会引发数据竞赛的程序的报告。

这篇论文做出了下列贡献。首先，它提出补充调度的概念，保证了副本在没有数据竞赛的时候不会偏离，并让副本在恶性数据竞赛很可能会偏离。第二，它展示了一种实际且低延迟的方法去使用补充调度去运行两个副本，这是通过用第三个副本来加速前两个副本的运行来实现的。第三，它展示了如何去分析三个副本的输出来打造一个生还数据竞赛的策略。第四，它介绍了一个新方法来检测数据竞赛，它拥有比传统动态数据竞赛检测更低的能耗。

我们在11个真实的桌面环境和服务器环境下的数据竞赛故障中测评补充线程调度的效率。在每一次测评中，Frost检测并且生还了所有这些故障。Frost的能耗在最坏情况下是3x，但如果有足够的核心或者空闲的CPU周期，它只有$3-12\%$的消耗。

2 补充调度

Frost的核心思想就是使用补充调度执行两个副本来检测和生还数据竞赛。一个数据竞赛由两条访问同一内存空间的指令组成（至少一条为写指令），以至于应用的同步限制能够让这两条指令以任意次序执行。对于恶性的数据竞赛，一个次序会导致程序出错（如果两个次序都导致出错，那么错误的根源不在于同步性的缺乏）。我们称那个会造成错误的顺序为一个故障要求。一个数据竞赛的故障可能有多个故障要求，它们都必须触发才会引发程序崩溃。

作为一个例子，考虑图1(a)中的简单故障。如果线程1在线程2解析指针之前将fifo设置为NULL，程序就将崩溃。而如果线程2先解析了指针，程序就会正确执行。在图中的箭头指出了故障需要。图1(b)指出了一个复杂一些的原子冲突。这个故障有两个故障需要，就是说，两个数据竞赛就必须以某种顺序执行。

为了解释补充调度概念下隐含的准则，我们先考虑一个至多只有一个数据竞赛故障，并且没有任何会导致顺序限制的同步（我们把这种执行序列成为无同步）的执行序列。对于这样的序列，补充调度为数据竞赛的检测和生还提供了坚强的保证。我们在这节分析这些保证，不过，真实的程序不止有这些区域，因此在3.4节我们讨论如何推广这些能被Frost保证的区域。

以补充调度执行两个副本的目标是保证有一个副本躲过了数据竞赛的故障。当每对没有被系统同步排序的指令对a和b，在一个副本中a 在b前，在另一个副本中b在a前时，我们称两个

副本有完美的补充调度。

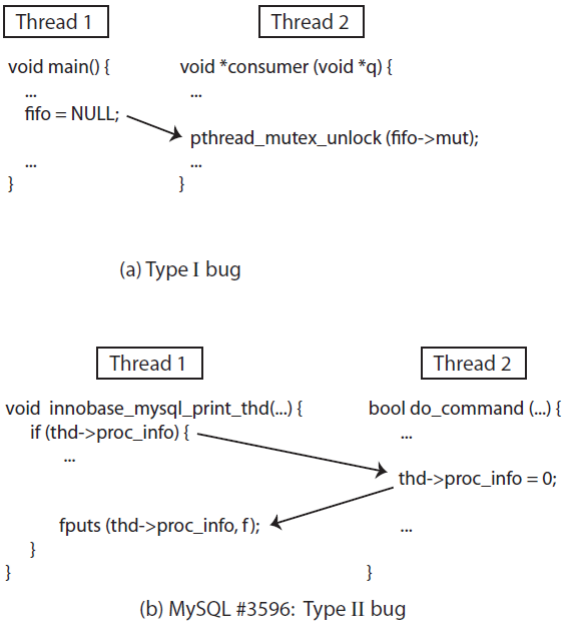


图1：数据竞赛的例子

为了达到完美补充调度，去除中断是必须的。为了理解这点，考虑考虑一个有中断的调度：线程A执行了指令a_0，然后线程B 中断了线程A然后执行了指令b_0，然后线程A继续执行了指令a_1。不可能给这个调度生成一个完美的补充调度。在这个调度中，a_1先于b_0，然后b_0先于a_0。不过，这样就使得a_1先于a_0，违反了线程内的执行顺序。

反过来，没有中断时，给无同步的两个线程的一段指令序列构造一个完美的补充调度是简单的–一个调度只需把A的所有指令放在B之前执行；另一个调度把B的所有指令放在A之前执行。对于超过两个线程，一个调度按某个顺序执行那些线程，另一个调度按相反顺序执行那些线程。

因此，为了保证至少一个副本在无同步的时间段里避免了一个特别的数据竞赛故障，我们执行两个副本，并使用无中断的调度，并且让两个副本的调度顺序是相反的。

上面那个算法为一些普遍的数据竞赛故障提供了更好的性质。比如，考虑图1(b)中的原子冲突。因为故障需要指向两个不同的方向，两个调度会各满足一个限制但不会满足两个。既然故障需要两个要求都被满足，我们提出的算法保证了两个调度都躲过了这个故障。

一般说来，给了n个线程，我们必须任选一个顺序给一个调度，然后把那个顺序反过来给另一个调度。想象一下线程按这个顺序排好序，如果所有的故障需要指向一个方向，那么算法保证了有一个调度不会发生故障。在这篇论文的其他部分，我们会把这类故障称为“类型I”的故障。如果有两个故障需要指向不同方向，那么算法提供了更强的保证：两个副本都不会发生故障，我们把这类故障称为“类型II”。

3 设计和实现

Frost使用补充调度来检测和生还数据竞赛。这一节描述了几个挑战，包括估计有同步指令和一些故障的程序段的理想行为，使用多核执行评价表现，和使用启发式方法来检测正确的和错误的副本。我们将以Frost可能会无法检测和生还一些竞赛的情况和Frost如果去极小化这些情况带来的影响的讨论来结束本节。

3.1 构造补充调度

Frost把程序执行分成了很多时间片，称为小片。在每个小片，它运行了多个副本，并且控制每一个的线程来达到一些性质。

Frost的第一个性质就是每个副本都遵循同一个系统调用和同步指令的顺序。换句话说，一些事件对，如对一个锁的lock和unlock，对一个条件变量的signal和wait，或者对一个管道的read和write，都表示了两个线程中的事件有先后关系；比如，lock之后的事件不会在unlock之前发生。通过确保所有线程对这种事件都有相同的先后顺序，Frost保证了两个副本只有在小片中出现了数据竞赛故障才会在输出或者内存状态中偏离[37]。另外，所有的副本都会遇见发生竞赛的指令对。

在第一个性质满足之后，Frost试图达到的第二个性质就是两个副本将拥有两个尽量互补的调度。根据我们在第2节论述的，这个性质是为了保证补充调度中至少一个副本不会因为某个数据竞赛而崩溃。

Frost必须执行一个副本来观察同步操作和系统调用的先后关系，之后才能在其他副本上加入相同的操作。为了观测那些

顺序，Frost 使用了一个调整过的glibc，并在Linux的终端中给每个线程和同步个体（如锁和条件变量）维护了一个向量钟。向量中的每个值代表了一个线程的虚拟时间。同步时间和系统调用会在本地向量钟中增加相应线程的值。类似unlock的操作会把那个锁的在向量钟中的值设为之前它的值和unlock的线程的向量钟值的较大者。类似lock的操作会把线程的值设为它之前的值和锁的值的较大者。类似的，我们调整了中断的个体使它们包含向量钟并且在相关的系统调用中传播这些讯息。例如，类似map和munmap的系统调用不能交换，Frost 给进程的地址空间加了个向量钟，来获得一个调整地址空间的系统调用的全序。

当第一个副本执行时，Frost把所有系统调用和同步记录到了一个日志了。其他的副本读取这些记录好的值并使用它们来遵循相同的发生先后关系。每个副本维护一个回放向量钟，它记录线程调用同步指令或者系统调用的情况。在回放向量钟相等或者超过第一个副本记录的值之前，线程不能继续执行。这保证了，举个例子，直到另一个线程unlock，一个线程不会从lock返回（如果这两个操作在原副本中有先后关系）。同一时间可以同时执行超过一个副本的小片；不过，其他的副本会稍微落后第一个副本，因为他们不能执行一个同步指令或者系统调用，在第一个副本执行完之前。

给定了一个先后发生顺序，Frost使用下列下列算法来来给两个副本构造调度使他们尽量得互补，但又没有修改原本的程序。Frost在一个副本中给所有线程选择了一个排序，然后给第二副本一个相反的排序。比如，三个线程在一个副本排序为[A,B,C]，那么在另一个副本中就排序为[C,B,A]。Frost在单核上执行每个副本的所有线程因此两个线程不会同时运行。只要一个线程没有在等待去满足一个先后发生关系并且没有跑完当前的小片，它就能继续执行。Frost的终端永远都执行调度中的第一个且能继续执行的线程。一个线程会持续执行知道当前小片结束，它会因先后关系，或者副本线程调度而阻塞。

3.2 使用单并行来扩展

就像我们到现在所描述的那样，补充调度不允许程序通过多核来扩展因为一个副本的所有线程都必须在一个2单核上顺序执行。如果一个副本的多个线程要同时在不同的核心上执行两条指令，这两条指令不会被一个先后发生限制所排序，因此可能会产生竞争。在这时候，这两个副本应该以不同的顺序执行这两条指令。不过，确定顺序，并让另一个副本以相反顺序执行就表明了这两条指令要顺序执行，而不是并行。

Frost使用单并行[44]来取得扩展性。单并行是建立在一个结论上的：至少两个方法来让一个多线程程序在多核上运行。第一个方法，称为线程并行，在不同的核心上跑多线程，是利用并行的一个传统方法。第二个方法，称为小片并行，并行地运行多个时间片。

单并行执行一个程序的线程级并行或者多个小片级并行。它限制每个小片级并行的执行来使他们在一个单核上执行。这个策略允许了小片级的并行来获得单核上运行的优势。我们对单并行原本的使用是在一个叫DoublePlay的系统，它能高效的软件决定性回放[44]。Frost是建立在DoublePlay的基础上修改来的，不过他出于一个不同的目的使用单并行，即使用以相同先后关系的补充调度来执行副本。

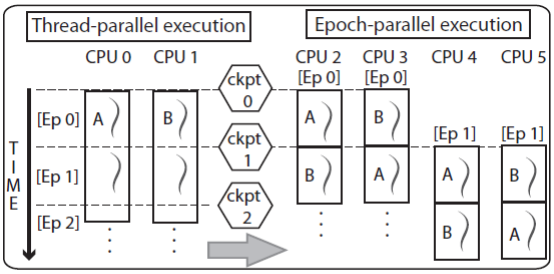


Figure 2: Frost: Overview

图2：Frost的概览

就像在图2中展示的那样，为了并行地执行小片，一个单并行的执行会生成每个小片的起始点。他必须尽早产生这些起始点来在之前的小片执行完之前开始执行之后的小片。因此，线程级的并行在小片级并行之前跑，并且产生未来小片的起始点。多个小片并行地执行，类似处理器的流水线–这使得小片级并行能随着能用的核心数而扩展。

总之，Frost给每个小片跑三个副本：一个线程级并行地副本用来记录每个小片的发生先后关系，并且为另外两个副本的并行运行生成检查点,还有两个小片级并行的副本以补充调度运行。副本使用写入时复制的分享来减少总共的内存开销。Frost使用在线的决定性回放[24]来保证所有副本获得了同一个非决定性的输入来使得所有副本都有相同的发生先后顺序。他在线程级并行的副本执行时，记录了所有系统调用的结

果和同步指令。当小片级并行的副本在之后执行相同的操作时，Frost的终端和glibc库不会重新执行操作，而是直接返回已经记录好的值。信号也在线程级并行的副本中被记录好，并且在小片级并行的同一点上发出。因为Frost记录并且回放了所有除了数据竞赛之外的非决定性因素，只有数据竞赛会使得副本发生偏离。在线回放还有一个好处– 小片级并行不会因为I/O而阻塞，因为线程级并行已经获得了结果。

当副本在一个小片内偏离时，Frost会在一些行动中选择一个。首先，它可能会接受某一个副本执行的结果，我们把这个叫做递交那个副本。如果它选择递交线程级并行的副本，它就是简单的放弃了小片开始的检查点。如果它选择递交了一个小片级并行的副本，并且那个副本的内存和寄存器状态和线程级并行的副本不同，那么流水线上之后的小片都是不正确的。因此，下一个小片开始的检查点是未来程序状态的不正确的一个提示。Frost先放弃递交的副本的开始的检查点。之后，它放弃所有递交的副本之后的所有小片。并且以递交的副本的状态来重新开始执行线程级并行和小片级并行的副本。

Frost也可能选择去执行更多的副本来获得更多关于导致偏离的小片的情况。他从那个小片的开端开始了一个额外的线程/小片级并行的执行，我们称这种这个过程叫做回滚那个小片。Frost之后可以决定提交原来的副本或者新的一个，不过现在只有新的会被递交。

因为副本会给出不同的输出，Frost不会放出任何输出知道它决定了提交哪个副本。他使用了推测性的执行（使用Speculator[32]来完成）来推测输出。当我们决定一个小片应该执行多长时间时，引出了一个正确性，功耗，和输出延迟的权衡。更长的小片会对正确性有好处，就像我们在3.4.3节讨论的那样，和比较小的功耗。更短的小片会有更短的输出延迟。通过使用适应性的小片长度，Frost 平衡了这些限制。对于有关CPU的无外部输出的程序，小片长度会长达1秒。不过，当程序会执行产生外部输出的系统调用，Frost会马上开始一个新的小片。因此，我们测评的服务器程序会有一秒几百个小片。另外，就像我们在3.3.1节论述的，小片长度会随便观测到的数据竞赛的数量而变化–没有数据竞赛的小片会逐渐的增加长度（一次50ms），然而出现数据竞赛的小片会减少小片长度（以最大20的比率）。当Frost决定开始一个小片之后，它会登台1所有线程去到达一个系统调用或者同步操作。然后它给程序设立检查点，并允许线程继续执行。

3.3 分析小片的输出

等到所有三个副本都完成了一个小片之后，Frost终端比较了他们的执行来找到并且生还数据竞赛。因为Frost的控制代码和数据在终端里面，下面的逻辑不会被程序级的故障所损坏。

首先，Frost测定一个副本是否已经崩溃或者进入了一个死循环。我们称这为自证明的故障，因为Frost能宣称这样的一个副本已经故障了，而不用去考虑别的副本的结果。Frost使用干预终端的信号处理机制来检测一个副本是否已经崩溃或者退出。它使用一个基于超时的机制来测定一个副本是否已经进入了一个死循环（我们发现无需进行更加复杂的检测）。

其他类型的故障不是自证明的；比如，一个副本可能会产生不正确的输出或者中间的状态。一个检测这种故障的方法需要一个细致的对正确输出的描述。不过，对于网络服务器或者数据库这种复杂的程序，制作这种描述是非常骇人的。在实际中面对解决这个挑战需要一个不依靠程序语义或者手工制造的描述的检测不正确输出的方法。

Frost通过比较三个副本的输出和程序状态来猜测潜在故障的存在。在执行中，Frost比较了每个副本产生系统调用的顺序和参数。Frost还比较了每个副本在每个小片执行完后的内存和寄存器的状态。为了减少比较内存状态对Frost表现的影响，Frost只会比较被污染的页或者新分配的页。如果他们在小片中的状态或者的输出之一不同，Frost就认为两个副本有不同的输出。

为了检测并且生还数据竞赛，Frost必须猜测两个数据竞赛发生了和哪个副本故障了。Frost首先考虑哪个副本经历了自证明的故障。如果这个副本没有经历一个自证明的故障，Frost考虑这个副本在一个小片后的内存和寄存器状态，和在小片中副本产生的输出。

3个副本的结果有11种组合，在表1的左边一列中展示了出来。F指副本经历了一个自证明的故障；A-C指某个副本产生的状态和输出。我们给相同的状态和输出使用了相同的字母。为了简化说明，我们在这个展示里不区别各种故障的类型。第一个字母展示了线程级并行的结果，另外两个展示了小片级并行的结果。比如，F-AA表示线程级并行的副本有一个自证明的故障，但两个小片级执行没有经历一个自证明的故障并且产生了同样的状态和输出。

另外，两个副本可能会产生同样的输出和最终状态，但因为一个数据竞赛在小片中采取了不同的执行过程。因为Frost的补充调度算法，很可能这个数据竞赛的无害的，就是说两个副本都是正确的。因此，在小片中允许小规模的偏离是一个有用的优化。当会产生一个合理的结果时，Frost 让一个小片级并行的副本执行执行一个不同的系统调用（如果这个调用没有副作用）或者一个不同的同步指令。比如，它允许一个小片级并行的副本去执行没有被线程级并行副本调用的一个nanosleep或者getpid系统调用。它同样允许自取消的指令对，如一个unlock跟着一个lock。当更多的优化可能时，被这样的优化过滤掉的无害的竞赛的数量是比较少的。因此，增加更多的优化可能是不值得。因此，让一个偏离不能被上述的优化处理时，Frost就称两个副本有不同的输出。

Epoch Results	Likely Bug	Survival Strategy
A-AA	None	Commit A
F-FF	Non-Race Bug	Rollback
A-AB/A-BA	Type I	Rollback
A-AF/A-FA	Type I	Commit A
F-FA/F-AF	Type I	Commit A
A-BB	Type II	Commit B
A-BC	Type II	Commit B or C
F-AA	Type II	Commit A
F-AB	Type II	Commit A or B
A-BF/A-FB	Multiple	Rollback
A-FF	Multiple	Rollback

左边一列给出了三个副本可能输出组合；第一个字母说的是线程级并行，另外两个说的是小片并行的结果。F指自证明的错误，A，B，或者C指一个没有自证明故障的副本。我们在副本给出相同输出和状态时，我们用相同的字母。

表一：epoch输出的分类

3.3.1 使用小片的输出来生还数据竞赛

Frost使用Occam的剃须刀来诊断结果：它选择能产生观察到的结果的最简单的解释。特别的，Frost选择需要一个小片内出现最少数据竞赛故障的解释。在相同数量故障的解释中，表1的中间那列展示了Frost和结果联系起来的解释，右边那列展示了Frost 根据那个解释采取的行动。

最简单的可能解释就是那个小片没有数据竞赛故障。因为所有副本遵从同一个发生先后关系的限制，一个没有数据竞赛的小片必然在所有副本中产生相同的结果，所以这个解释只能应用于A-AA和F-FF。对于A-AA小片，Frost认为小片在所有副本中都正确执行并且进行递交。对于F-FF小片，Frost断言小片因为一个非竞赛的故障在三个副本中都失败了。在这个情况下，Frost回滚并且从小片开端重新执行程序，希望这个错误是非决定性的，可能在一个不同的执行中被躲过，比如，这个故障是因为一个不同的发生先后关系而发生的。

第二个最简单的解释就是小片经历一个I类型的数据竞赛故障。单个I类型的故障会产生最后两个不同的输出（一个顺序一个），因此两个小片级并行的执行会不同，因为他们以不同顺序执行发生竞赛的指令。对于I类型的故障，有一个顺序不会满足故障需要，因此会正确执行。另外一个顺序会满足故障需要并且导致一个自证明的故障，不正确的状态，或者不正确的输出。下列组合的小片级并行副本有两个不同的输出（有一个是对的），并且最多只有两个不同：A-AB（A-BA），A-AF（A-FA），和F-AF（F-FA）。

对于结果是A-AF和F-AF的结果，一个没有发生自证明的错误的副本比较可能是正确的，因此Frost提交那个副本。对于产生A-AB 的副本，我们不清楚那个是正确的（可能因为一个无害的竞赛，两个都是对的），因此Frost要收集更多的信息，它开启了新一组的三个副本，从小片的开端开始运行。在这时候，Frost首先试图去寻找一个能防止竞赛发生的发生先后限制；我们假设对于一个经过良好测试的程序，这样的执行很可能是正确的。对于我们已经测试过的数据竞赛，Frost经常在一个或者两个回滚后遇到这样的一个限制。这是因为结果的不同组合（比如，A-AA，Frost可以递交小片并且继续）。如果Frost在每个执行都碰见了数据竞赛，我们就认为最自然的执行可能是正确的，就是说，让Frost选择线程级并行的执行（这是最正常的执行）。注意，因为小片级并行的执行使用了手工的调度，他们不应该被看重。因为，我们不会认为A-BA给A投了两票，B一票，而认为只给A投了一票。

如果结果的一个组合不能被一个类型I的故障解释，下一个简单的解释就是一个类型II故障。一个类型II的可以产生下述结果组合：A-BB，A-BC，F-AA，和F-AB。这里没有一个应该是被一个类型I的故障引发的，因为小片级并行的副本产生了相同的结果（A-BB或者F-AA）或者因为有三种输出（A-BC或者F-AB）。在之后的那个情况下，输出不可能是由一个类型I的故障引发的，不过在第一个情况下，那个结果是不太可能的。任何小片级并行的执行都应该躲避类型II的故障，因为它无中断的执行不符合一种故障的故障需要。比如，原子冲突故障是

一种线程交叉时引发的故障。因为线程不会在小片级并行中被中断，两个副本都会避开这种故障。

我们发现类型II的故障很普遍地会有三个不同的输出（比如，A-BC或者F-AB）。比如，考虑两个线程以一个没有同步形式在记录输出。线程级并行的副本不正确地将输出混合起来（输出A），一个小片级并行的正确的在第二个值之前完整地输出了第一个值（输出B），剩下的小片级并行副本也正确地完整地在第一个值之前输出第二个值（输出C）。在插入或者删除非同步的共享数据结构也会遇到相似的情况。因此当Frost遇见一个A-BC输出，它会递交一个小片级并行的副本。

剩下来的组合（A-BF，A-FB，或者AFF）不能被单个数据竞赛故障所解释。A-BF有多于两个输出，这样就不是一个单个类型I的故障了。A-BF也包含了一个失败的小片级并行的运行，这样就不是单个的类型II的故障了。两个小片级并行的副本都在A-FF失败，这也不可能是单个的类型I或者类型II的故障。我们的结论是这些组合是由一个小片内的多个数据竞赛故障引发的。Frost会回滚到小片开始检查点，然后执行一个比较短的小片长度（希望只遇到一个故障）。

3.3.2 使用小片输出来检测竞赛

使用小片的输出来检测数据竞赛比使用这些输出来生还数据竞赛更加容易。任何在系统调用，同步指令，或者最终状态中出现偏离的输出都表明在小片中出现了一个数据竞赛。因为这三个副本遵从相同的发生先后关系，数据竞赛是唯一的副本偏离的理由。另外，所有的副本也是从同样的初始状态开始的。

因为Frost的数据竞赛检测是基于输出的，不是所有在小片中发生数据竞赛都会被报告。这是一个过滤无害竞赛的一个好方法，有时程序员会有意地加入来提升表现。特别的，一个ad-hoc的同步可能永远不会引发一个内存或者输出的偏离，或者，一个竞赛可能会引发一个暂时的偏离，比如马上就被覆盖的栈中的值。如果Frost尝试了一对指令的不同顺序并且没有报告一个竞赛，那么这个竞赛几乎就是无害的，至少在程序的这次执行中。唯一的例外，在3.4.4节中会讨论，就是多个故障导致了相同但不正确的程序状态或者输出。

既然Frost允许副本在小片中出现些许的偏离，它有时观察到副本在系统调用或者同步指令上有不同，但没有在输出或者最终状态没有出现不同。这种竞赛同样是无害的。Frost会报告这种竞赛的存在，但增加一个注解：没有在程序上产生明显效果。一个开发者能自己选择是否去解决这些竞赛。

因为Frost实在DoublePlay框架上做决定性的记录和回放的，它继承了DoublePlay的重新执行的能力[44]。因此，在报告了一个竞赛的存在之后，Frost也能根据需要重新把导致竞赛的程序给运行一遍，这样就允许一个开发者去选择他最喜欢的去故障工具。比如，我们在DJIT+[34]上面搭建了一个传统的动态数据竞赛检测器，它能回放一个偏离的小片来精准地确认出现竞赛的指令。

3.3.3 采样

一些检测竞赛的工具使用采样来以错过数据竞赛的代价来减少功耗[6,14,29]。我们给Frost增加了类似的选项。当用户确定了一个目标采样频率后，Frost就只给一些小片创造小片级并行的副本；我们称这些小片为采样过的小片。Frost不会给别的小片执行小片级并行的副本，意味着它在这些小片中既不检测也不生还数据竞赛。Frost动态的选择哪些小片来采样，使得总的执行时间和采样频率相符。虽然可以使用更加复杂的机制来选择采样的小片，这个策略能使得Frost对数据竞赛检测和生还的能力减少和采样频率成正比。

3.4 不足

第2节讨论了补充调度能在没有同步且数据竞赛不多于一个的程序段中检测并生还数据竞赛的保证。我们现在讨论当小片不符合这些性质时的不足之处。我们也会讨论Frost是如何缓和这些不足之处的。就像在4.1.2节展示的那样，这些不足没有在实际测试中给Frost造成多大麻烦。

3.4.1 一个小片中的多个故障

尽管我们假定数据竞赛的故障比较少见，一个小片还是可以包含多余一个故障。如果在小片中出现了多个故障，Frost对这个执行的诊断可能是不正确的。这会影响生还和检测数据竞赛。

生还数据竞赛需要至少一个副本正确地执行。给一个小片增加任意多的类型II的故障不会影响生还，因为两个小片级并行的副本都不会因为这种故障而失败。因此，当一个小片有零

个或一个类型I的故障再加上任意多个类型II的故障时，都会有一个副本是正确的。但是，多个类型I的故障的存在会导致两个副本都失败。一般来说，不同的故障会导致程序以各种方式来崩溃。失败的表现（如崩溃或者退出）可能是不同的，或者内存和寄存器状态在失败的时候也会是不同的。因此，Frost仍然可以采取更正措施，如回滚，或者执行额外的副本，特别当失败是自证明的时候。当Frost回滚时，它会减少小片长度来在重新执行时将故障分隔开来。这像是一种搜索。

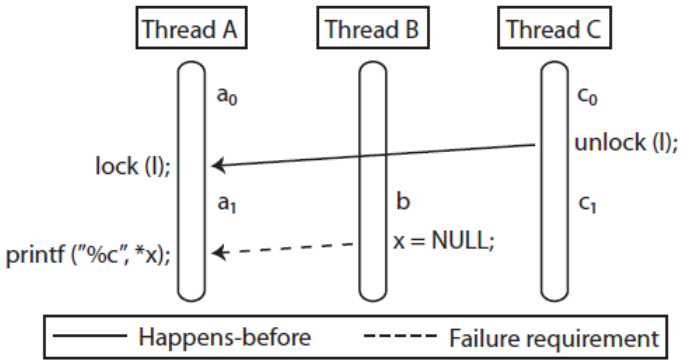
有可能（虽然很少见）两个不同的类型I的故障在程序状态上有相同的作用，这时补充调度的副本会到达相同的最终状态。如果失败不是自证明的，Frost会错误地将小片归类，并且递交了一个错误的状态。

为了检测数据竞赛的目的，多个数据竞赛只在所有竞赛对程序状态有同一个作用时会成为一个问题。不然，副本会偏离，然后Frost 会给这个小片报告一个竞赛。当回放时，开发者就会发现多个数据竞赛。

3.4.2 优先级反转

先后关系限制可能会导致Frost没有生还或者检测到一个小片内的数据竞赛。对于那种只有线程互相交互的小片，Frost的补充调度算法会构造出使所有潜在的会竞赛的指令顺序不同的调度。不会产生竞赛的指令可能会以相同顺序执行，不过，根据定义，这不会影响到数据竞赛的保证。

当多于两个线程在小片中交互时，一个类似于优先级反转的情况会发生，并且防止Frost改变所有不发生竞赛的指令的顺序。比如，考虑图三。这个小片包含了三个线程，一个由于同步的发生先后关系限制，和一个由两个竞赛的指令引发的故障需要。如果Frost 将ABC 的顺序设定给了一个副本，两个副本中的执行顺序就是$\{a_0, b, c_0, a_1, c_1\}$，和$\{c_0, c_1, b, a_0, a_1\}$。所有的代码段对在两个调度中都以不同顺序执行，除了两个例外。在两个副本里，c_0都在a_1之前执行。不过，这个顺序是程序的同步所需的，并且那个同步防止了这些指令出现竞赛。另外，b在两个调度里都在a_1之前执行。如果图中的类型I的故障出现的话，那么两个副本都会失败。如果失败不是自证明的且没有在线程级并行中出现的话，这可能会防止Frost生还或者检测竞赛。



图三：优先级反转的情况

注意Frost可以给三个线程选择另一组性质，比如BAC。基于这个观察，我们设立了一个机制，能够帮助选择线程的特性来防止优先级反转。就在程序执行时，Frost记录每对线程的发生先后限制的数量。它使用一个贪心算法来将两个发生先后限制最多的两个线程安排了相邻的优先级，然后再放入同样限制最多的线程，这么下去。因为优先级反转只会在在优先级中2个非相邻的线程上，只要之前的限制是对之后限制的一个比较好的预计，这种机制就能减少优先级反转的几率。

在某些情况下，一个小片的线程级并行执行可能在小片级并行执行之前完成。在这种情况下，Frost可以发现那个小片特有的发生先后限制，并且以此来选择线程的性质。但我们还没有使用这个优化。

3.4.3 小片的边界

Frost把执行分成小片来使用多核获得扩展性。每个小片代表了一个顺序限制（一个障碍），这个障碍在原本的程序中是不存在的。如果一个错误需要经过了一个小片边界（一条指令在一个小片中，另外一个在下一个小片里），这两条指令的顺序在所有副本中都是固定的。对于类型I的故障，所有副本都会失败或者都会躲过故障。

为了减缓这个不足，Frost采取了两步。首先，它以比较小的频率构造小片。然后，它当所有线程都在执行一个系统调用时创建了一个小片。对于一个类似于原子冲突的数据竞赛，这保证了没有副本会失败，除非程序在一个原子区域执行了一个系统调用。

所有的用来生还恶性竞赛的系统（如Frost），都必须在输出之前提交状态，并且继续执行是以输出为前提的。为了避免

一个恶性竞赛引发的故障，这种系统必须回滚到某个在故障前提交的状态。这个提交的状态可能人工地将提交点上下将指令定序，并且这个顺序限制可能会强迫程序经历一个恶性数据竞赛[26]。

当Frost只被用来检测而不生还数据竞赛时（比如在测试时），可能就没有必要在数据竞赛发生后保持外部输出的稳定了。因此，我们使用了一个优化：当Frost只被用来检测时，外部输出不会引发下一个小片的创立。这个优化只在4.2节被使用。

3.4.4 类型II故障的检测

Frost基于输出的竞赛检测可能不能检测到某些类型II的故障。检测需要任意两个副本在系统调用或者同步指令中不同或者两个副本在小片末尾有不同的内存或者寄存器状态。就像之前提到的，一些无害的竞赛有这个性质–过滤掉这种竞赛基于输出的竞赛检测的一个优势。另外，如果两个竞赛的指令集合相同，一个代码段也会有这种性质。当两个小片级并行的副本都正确并且以同一状态完成小片时，这种情况就很容易发生。不过，在我们到目前的经验来看，类型II的故障总是会在程序状态或者输出中有一些不同。

4 检测

我们的检测回答了以下几个问题：
Frost能多么有效地生还数据竞赛的故障？
Frost能多么有效地检测这样的故障？
Frost的功耗怎么样？

4.1 生还和检测竞赛

4.1.1 方法论

我们用一个4GB的DRAM内存，由两个2.55GHz四核Xeon处理器组成的8核服务器来测评Frost检测和生还数据竞赛的能力。操作系统是CentOS Linux 5.3，Linux 2.6.26终端和2.5.1的GNU库，都被调整过来支持Frost。

我们使用11个真实的并行故障来测评。由重现一个在线的并行故障集合[50]开始（Apache,MySQL和pbzip2），他们是从几个学术资源那里编译过来的[27,49,51]，还有BugZilla的数据库。从12个并行故障中，我们重现了所有9个数据竞赛故障。另外，我们重现了应用pfscan中的一个数据竞赛故障，它曾经在学术讲座上被使用过[51]。最后，在我们的测试中，Frost检测到了glibc中一个之前不知道，可能是恶性的数据竞赛。表2列出了故障并且描述了他们的作用。

对每个故障，我们做了5次测试，在测试中故障会在Frost的监护下起作用。表2的第4列展示副本对包含故障的小片的输出。第5 列展示了Frost以一个没有故障的输出来生还数据竞赛的百分比。下一列展示了Frost根据输出或者状态检测到故障的百分比。最后一列展示Frost要花多久才能从故障中恢复过来，这包括了回滚和执行新副本的代价。

Application	Bug number	Bug manifestation	Outcome	% survived	% detected	Recovery time (sec)
pbzip2	N/A	crash	F-AA	100%	100%	0.01 (0.00)
apache	21287	double free	A-BB or A-AB	100%	100%	0.00 (0.00)
apache	25520	corrupted output	A-BC	100%	100%	0.00 (0.00)
apache	45605	assertion	A-AB	100%	100%	0.00 (0.00)
MySQL	644	crash	A-BC	100%	100%	0.02 (0.01)
MySQL	791	missing output	A-BC	100%	100%	0.00 (0.00)
MySQL	2011	corrupted output	A-BC	100%	100%	0.22 (0.09)
MySQL	3596	crash	F-BC	100%	100%	0.00 (0.00)
MySQL	12848	crash	F-FA	100%	100%	0.29 (0.13)
pfscan	N/A	infinite loop	F-FA	100%	100%	0.00 (0.00)
glibc	12486	assertion	F-AA	100%	100%	0.01 (0.00)

结果时测试的平均值。括号内的值是标准差。

表2: 数据竞赛的检测和生还

4.1.2 结果

这些实验的主要结果就是Frost检测并生还所有11个故障的5次检测。对于这些应用，生还数据竞赛增加了一些执行功耗，主要是因为小片对于如MySQL何Apache的应用来说太短了，并且另外应用中的故障在执行的末尾发生，因此撤销之后的工作几乎没有减少功耗。我们接下来给每个故障提供更多的细节。

pbzip2数据竞赛在一个工作线程对主线程解法的一个指针解引用时可能触发一个SIGSEGV。这是一个类型II的故障因为解引用必须在重新分配之后而且在主线程结束之前。这种失败是自证明的，导致了一个F-AA的小片输出。

Apache的21287号故障是在更新和检查缓存对象时缺乏原子性导致的，一般会造成一个双重解锁。这是一个潜在的故障：这个数据竞赛会导致引用计数上的一个不正确的值，然后会造成一个应用错误。Frost通过检测出发生竞赛的小片末尾的内存偏离检测出了这个故障，这比这个错误的效果展露出来要快得多。早前的检测使得Frost能避免输出被数据竞赛污染的输出。这个故障可能是类型I也可能是类型II的，根据缓存操作的顺序。

Apache的25520号故障是由两个线程同时以不安全的方式修改了一个共享变量导致的原子冲突。这导致了Apache的接入日志里出现了混淆的输出。由于日志被输出前会先被缓存，Frost检测到了一个内存偏离。这个小片被归类到了A-BC，因为这个失败不是自证明的且两个小片级并行的执行给出了不同顺序的日志消息（因为日志操作是并行执行的，所以两个顺序都是对的）。

Apache的45605号故障是在一个调度线程在等待一个条件变量之后没有重新检查条件引发的原子冲突。为了这个故障来生效，这个调度线程必须在一个循环中旋转多次并且接受多个条件。因为需要输出没有在小片解释之前被放出，Frost防止了这个故障。既然accept是一个同步网络指令，两个accept不会同一个小片中出现。因此，Frost把它检测出的故障转换成了一个无害的数据竞赛。即使多个accepts的需要被删除了，Frost把这个故障认为是类型II的竞赛并且生还了。

MySQL的644号故障是一个类型II的原子冲突，导致了一个不正确的循环结束条件。这导致了引发MySQL崩溃的内存泄露。Frost 根据故障小片末尾的一个内存偏离检测到了这个故障。因此，它在内存污染导致不正确输出之前恢复了过来。

MySQL的791号故障是一个类型II的原子冲突，导致了MySQL无法去做日志操作。和Apache的25520号故障类似，Frost看见了故障小片输出为A-BC，不过区别发生在外部输出而不是内部状态。就像对Apache的故障一样，两个小片级并行的副本的输出不同，但都是正确的。

MySQL的2011号故障是一个类型II的多变量原子冲突，当Mysql旋转中继日志时发生。这导致MySQL在错误检测时崩溃了，导致了不正确的行为。Frost以一个A-BC的输出检测到了这个故障。

MySQL的3596号故障是在图1(b)中展示的类型II故障。对NULL指针的解引用产生了一个自证明的故障。两个小片级并行的副本躲过了竞赛并且采用了根据情况正确但偏离的路径。因此Frost看到了小片输出为F-BC。

MySQL的12848号故障在一个缓存重新定大小的操作时展现了一个不正确的缓存中间大小，使得MySQL崩溃了。虽然这个被一个锁保护着，一个锁的获得丢失了，导致了一个不正确顺序的操作，成为了一个类型I的故障。既然崩溃很快就发生了，这个崩溃是自证明的。

一个pfscan中的类型I的故障，导致主线程进入了在工作进程结束时进入了一个旋转循环。Frost将这个旋转循环认定为一个自证明的故障，并把小片分类为F-FA。第三个副本通过选择一个不符合故障需要的竞赛指令的顺序躲过了旋转循环。

当重现已知的故障时，Frost还检测到了一个另外的，在glibc中没有报告的故障。多个线程无同步地并行地更新malloc的数据计数器，导致可能出现不正确的值。当能够调试时，对这些变量的额外检查会引发断言。如果一个数据竞赛引发了不可行的数据时，这些断言会不正确地被触发。我们写了一个测试程序来可靠地触发这个故障。既然断言像数据竞赛一样发生在同一个小片里，这个故障是自证明的。我们已经将这个数据竞赛报告给了glibc的开发者，并且正在等待回应。

总的来说，对于广泛的应用故障，Frost能够检测并且生还所有故障，以最小的恢复时间。对于污染程序状态的延迟故障，Frost检测在程序出现自证明的失败征兆之前检测除了故障，防止了不正确的输出。

App	Bug Number	Harmful Race Detected?		Benign Races	
		Traditional	Frost	Traditional	Frost
pbzip2	N/A	5	5	3	1
apache	21287	0	0	55	2
apache	25520	3	3	61	2
apache	45605	3	3	65	2
MySQL	644	4	4	2899	2
MySQL	791	3	3	808	1
MySQL	2011	0	0	1414	1
MySQL	3596	0	0	658	2
MySQL	12848	0	0	1449	2
pfscan	N/A	5	5	0	0
glibc	12486	6	6	9	3

第3列展示了一个全覆盖的传统动态竞赛检测器在多少次运行中检测出恶性的竞赛，第四列展示了Frost有多少次。最后两列展示了无害数据竞赛的数量。

表3: 对数据竞赛检测范围的比较

4.2 独立的竞赛检测

接下来我们对Frost的数据竞赛检测和传统的发生先后关系的动态数据竞赛检测器进行覆盖范围的比较。4.1.2节展示了Frost检测并生还了我们数据中所有的恶性数据竞赛。不过，在这些实验中，我们只考虑了恶性的竞赛。这个可能会让Frost的检测变得人容易，它可以让副本更可能出现偏离。

在这一节里，我们重复4.1.2节的实验，不过我们不会采取一些特殊手段来让故障生效。就是说，我们简单地执行一序列存在潜在竞赛指令的程序段。比如，我们建立一个基于DJIT+[34]的数据竞赛检测器。虽然很慢，这个数据竞赛检测器能提供全范围的检测；就是说，它会检测出程序运行中的所有错误。既然现在数据竞赛检测器一般用范围换速度（比如，使用采样），我们使用的这个全范围的数据竞赛检测器提供了最强的竞争性。

比较数据竞赛检测器的范围是有挑战性的，因为一般没有保证说两个工具会在程序的不同执行中遇到相同的指令序列和同步操作。幸运的是，因为Frost是建立在DoublePlay的基础上的，我们可以使用DoublePlay来进行决定性地回放。当我们在回放执行中运行一个动态竞赛检测器的时候，能保证线程级并行和小片级并行都见到同样顺序的同步指令。另外，每个线程执行的指令直到第一个数据竞赛也都是一样的。这保证了一个完美的比较。

表3比较了Frost和传统动态数据竞赛检测器的范围。我们以相同的测试时间比较了每个数据；这个表显示了所有运行的统计数据。

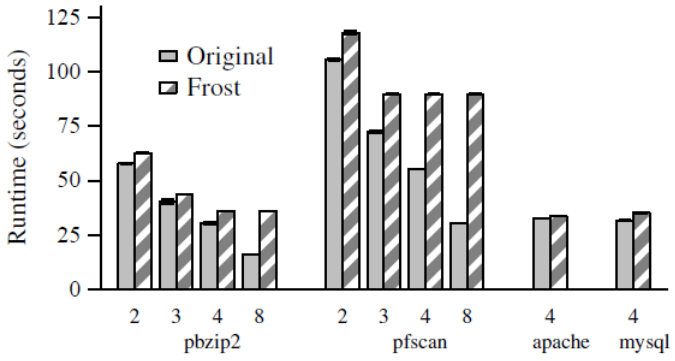
对于每个传统数据竞赛检测器检测到恶性竞赛的运行，Frost也检测到了同一个竞赛。表3的第三列列出了传统数据竞赛检测器发现恶性竞赛的运行次数。第四列列出了Frost发现同样竞赛的运行次数。对于一些恶性竞赛，在我们预设定的测试过程中，Frost和传统数据竞赛检测器都没有检测出来。这是可以预见的，因为动态数据竞赛检测器必须看到没有同步的指令执行，才能报告一个竞赛。

我们同样测评了3.4.2节所描述的排序机制带来的好处。我们去除这个机制来运行Frost时，它没有检测出pbzip2中的恶性竞赛。我们证实了原因是因为优先级反转。

表3的最后两列列出了传统数据竞赛检测器和Frost识别出的无害的数据竞赛。由之前给出的分类法[31]，我们手工分类了79个由传统检测器在pbzip2,Apache,pfscan和glibc数据集中被传统竞赛检测器报告出的79个无害的竞赛。获得了以下结果：(a)用户构造的同步(42个竞赛)：比如，Apache使用自定义的同步，这是传统检测器在没有注释时没有发现的，因此传统检测器错误地把正确的同步操作认为是竞赛，(b)多余的写操作（8个竞赛）：两个线程将同一个值写到一个位置，(c)双重检查（11个竞赛）：一个变量被有意地检查，而没有取得一个锁，然后在否认后又检查了一次，和（d）大致的计算（18个竞赛）：比如，glibc的malloc机制维护了数据并且有一些线程并行地进行记录而缺乏同步机制。我们也对MySQL的644号进行了分类，并且发现用户构造的同步产生了2619个无害的数据竞赛，冗余写占了71，双重检查占了153，大致计算占了156。因为时间和精力的关系，我们没有对其他MySQL数据进行分类。

反过来，Frost报告了更少的无害竞赛。比如，如果一个竞赛导致了一个短暂的偏离（比如一个幂等的写写竞赛），那么如果小片在结束之间又聚合了，Frost 不会标明这个竞赛。Frost也不用担心自定义同步，如果那个同步保证了同步指令在所有副本上有相同的作用。在我们的数据中，Frost只标明了8个无害竞赛（2个双重检测和6个大致计算）。因此，大概有一半的被Frost标明的竞赛是恶性的，不过小于0.25%的被传统检测器标明的是恶性的（大部分无害的竞赛是因为MySQL中的自定义

同步）。



这幅图显示了在一台8核机上Frost如何影响了4个数据的表现。我们展示了2,3,4,8线程的pbzip2和pfscan。Apache和MySQL是输入输出有界的，因此结果在2和8线程是相同的。我们把4线程的结果作为代表样本。结果是5次试验的平均值；错误条有90%的可信度。在有足够的核心去跑多余的线程时，Frost增加了一点点功耗（3%到12%）。当工作线程超过3（pfscan）或者4（pbzip2）时，Frost就无法掩盖运行多余副本的代价。

图4: 运行功耗

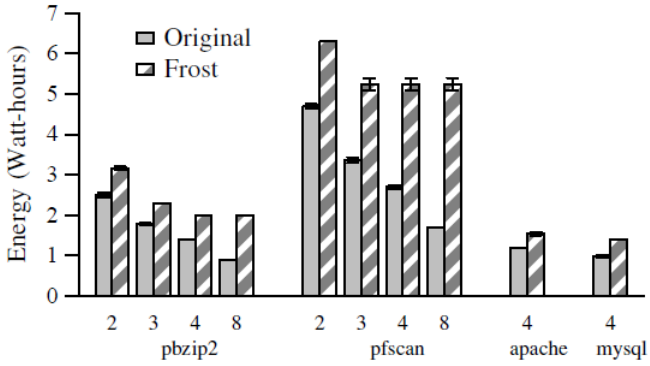
4.3 表现

4.3.1 方法论

我们之前的实验展现了Frost在pbzip2,pfscan,Apache和MySQL中检测和生还数据竞赛的能力。接下来我们测定了Frost引入的功耗。

我们并行地测试了pbzip2，使用它去压缩一个498MB的日志文件。我们使用pfscan去在935MB的日志文件的文件夹中寻找字符串。我们通过执行了150次运行扩展了数据，因此我们可以在保证数据在文件缓存中的时候检测Frost的功耗（不然，我们的数据将只能跑在磁碟上）。我们使用ab(Apache Bench）来测试了Apache，同时给多个用户那里发送了5000个70KB文件的请求，在同一个本地网络下。我们使用版本0.4.12的系统数据测评了MySQL。这个数据使用了多个客户线程来在ISAM数据库中生成2600个数据库询问；2000个是只读的，600个更新。

对于这些程序，工作线程的数量控制了他们能够有效使用的最多的核心数量。对于每个数据，我们调整了工作线程的个数，从2 个到8个。一些数据有一些几乎没有事干的控制线程；我们没有把他们算进去。pbzip2使用了两个多余的线程：一个去读取文件一个来输出；这些线程也没有被算入。所有的结果是5次测试的平均值。



这幅图展示了Frost的能源消耗。我们展示了pbzip2和pfscan的2,3,4,8线程，还有Apache和MySQL的4线程。结果是5次测试的平均值；错误条有90%的可信度。

图5:能源消耗

4.3.2 吞吐量

影响Frost的CPU有界应用的主要因素就是没有使用的核心的可用性。就像在图4所展示的，当只使用2个核心时，Frost给pbzip2 增加了一个合理的8%的功耗，给pfscan增加了12%。Frost的执行功耗低的原因是服务器有空闲的资源来运行额外的副本。

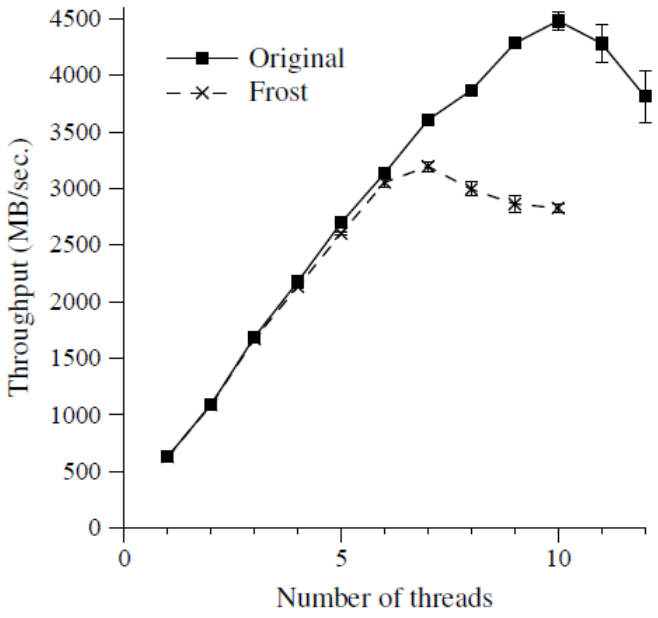
对了检测Frost的功耗如何随着空闲的核心数而改变，我们逐渐增加线程个数到这台8核机器的总容量。Frost对于pfscan的表现提升停止在了3个工作线程，这是可以预见的，因为使用3个有3个线程的副本需要9个核心（比这个电脑有的多一个）。对于pbzip2，Frost 的表现扩展到了4个线程，这是因为应用特性的行为。一个pbzip2中的数据竞赛有时会导致一个包含一个nanosleep的自循环。当这个发生时，一个副本不会消费CPU时间。就像预期的那样，如果这两个CPU有界的程序使用所有8个核心，Frost增加了稍少于200%的功耗。就像所有使用活跃副本的系统一样，Frost不能隐藏运行额外副本的代价，当没有多余的资源能够调用的时候。

相反，我们发现服务器程序，Apache和MySQL不能通过额外的核心来扩展，因此受Frost的功能影响较少。特别的，我们发现Apache的瓶颈在于网络输入输出，MySQL的瓶颈在于硬盘的输入输出。既然Apache和MySQL都不是CPU有界的，从2至8改变线程的个数不会影响源程序或者Frost的执行时间。因为这个原因，我们只展示了4线程的结果。就像在图4中所展

示的那样，Frost只给Apache增加了3%的功耗，而给MySQL增加了11%的功耗。

4.3.3 能源消耗

即使多余的资源会隐藏执行多个副本的表现影响，额外的执行有一个能源消耗。在与能源成比例的硬件上，能源的消耗大约有200%。我们对当前硬件感兴趣，它不是能源成比例的。



这个图展示了pfscan的吞吐量（每秒扫描了多少MB的数据），在有和没有Frost的情况下。我们改变了pfscan的工作线程个数。结果是5次测试的平均值。错误条有百分之50的可信度。

图6：在32核机上的扩展性

我们使用了一个Watts Up? .Net能源计来测量在有Frost和没有Frost时这台8核机消耗的能源。就像在图5中展示的，在2线程时，Frost给pbzip2增加了26%的能源消耗，给pfscan增加了34%。在8个工作线程时，消耗变为了122% 和208%。Frost给Apache增加了28%的能源消耗，给MySQL增加了43%，无论有多少个工作线程。

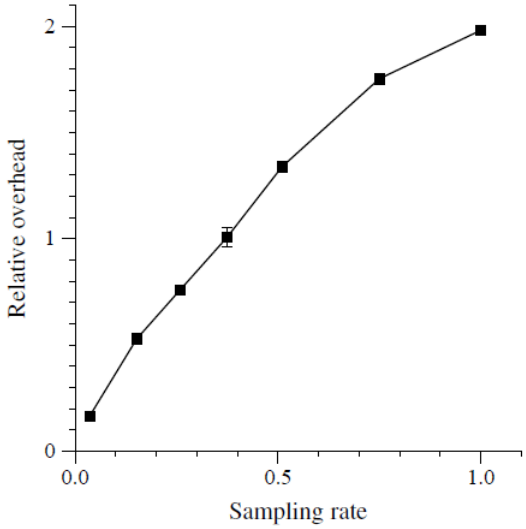
4.3.4 扩展性

因为在有2-3个工作线程时，8核机就是用尽CPU的资源，我们接下来在有4个2.27GHz的8核Xeon X7560处理器和1.8GB内存的32核机上测评的Frost的扩展性。这个服务器运行了之前实验中的同样的程序。我们选择了pfscan来看，因为它展示了一个最高的8核消耗。我们通过增加了100倍的数据扩展了数据。我们报告了pfscan的吞吐量，即每秒扫描的数据量。

就像在图6所展示的那样，pfscan在没有Frost时能够很好地扩展，直到10核。在这时候，我们猜测它已经用尽了这台32核机的所有内存带宽。Frost能在这台机器上比较好的扩展到6核，只有少于4%的功耗。Frost在pfscan上的执行在7核时达到了最大的吞吐量。我们猜测因为额外的副本，他更早地用尽了内存带宽。因为副本都执行相同的工作，缓存稍微减缓了组合副本需要3倍于原本数据的情况。

4.3.5 采样

就像在3.3.3节中讨论的那样，Frost能被设定为只采样一部分程序。采样减少了功耗，但Frost只会在采样到的程序段中检测或者生还数据竞赛。因此，Frost会经历一个检测和生还数据竞赛的比率的降低，这是和采样频率成正比的。



这幅图展示了Frost在不同采样频率的相对功耗。一个0.25的采样频率意味着Frost只在4个小片中的一个检测和生还数据竞赛。结果是5次测试的平均值；错误条有百分之90的可信度。

图7：采样在相对功耗上的效果

我们在8核机上重新跑了我们之前实验中使用的8个工作线程的CPU有界的pfscan。我们改变了采样频率并且测量了Frost增加的额外相对功耗。图7展示了，对于3.5%的采样

频率，Frost只增加了17%的相对功耗。就在采样频率增加时，Frost的相对功耗几乎是线性增加到了当没有采样时的200%。

4.4 讨论

总的来说，Frost检测到了一个全范围的数据竞赛检测器检测到的全部恶性数据竞赛，并且在我们的实验中生还了这些竞赛。虽然这些结果是喜人的，我们相信还是有一些情况Frost会无法顺利解决，就像在3.4节中描述的那样。当程序有多余的资源来执行多余的副本时，Frost的功耗为$3-12\%$。当没有空闲的资源能够使用时，执行多余副本的代价就无法被掩盖了。Frost能随着核心的数量比较好地扩展，虽然它会经历如内存带宽之类的限制，只要那个资源是一个瓶颈。

Frost测定的功耗比它的基础DoublePlay[44]所报告的要稍微少一点，因为有做代码优化。单并行的执行，就是Frost和DoublePlay都在采用的，会有比较高的功耗，导致内存中的页被很快地污染，比如，到目前我们发现的DoublePlay的最快情况功耗是SPLASH-2的ocean数据（对于空闲核心来说是121%）；我们期望Frost在有空闲核心时有差不多的功耗，而在没有空闲核心时有3倍的功耗。

对于一个不同没有改动过的程序，这些检测出的功耗比传统动态数据竞赛检测其要少得多。就像使用多副本的其他系统一样，Frost在可靠性和利用性上提供了一个权衡。在软件的生命周期里，一个人可以以不同的优先级来运行Frost；比如，我们可以软件在刚刚放出或者更新时使用Frost去检测和生还数据竞赛，然后去除Frost或者让Frost对一些小片进行采样，以此来减少功耗，因为这个程序已经可以认为是没有竞赛的了。另外，既然一般很难扩展很多的工作（比如，输入输出有界的），在生产中一般我们都能有空闲的核心，这样Frost就能掩盖它自己的功耗。我们同样可以只在有空闲的核心时使用额外的副本。

5 相关工作

因为Frost又能检测又能生还数据竞赛，我们两类的相关工作都会讨论。

5.1 数据竞赛生还

使用副本来生还错误能够追溯到计算机的早期[45,28]。在活跃（状态机）的复制中，副本并行执行并且可能用来检测错误并且给哪个结果是正确的进行投票[41]。在被动（主备份）的复制中，一个副本一直被使用，直到检测到了一个错误，然后另一个副本从一个已知的正确状态开始执行[8]。被动的副本引发了更少了运行时间功耗，不过不能通过比较副本来检测错误。Frost是使用活跃副本的。

在1985年，Jim Gray发现当暂时的硬件故障能够被重新执行指令来处理（一种类型的被动副本），一些软件错误可能可以被用同样的方式处理。研究者通过很多种方式扩展了这个想法，比如连续地从老的状态来重试[47]，预防延迟错误的前瞻性的重启[20]，缩小需要被重启的系统部分[9]，并且减少运行多个副本的消耗[19]。

一种在基于副本的系统中去增加生还几率的常见技巧就是去使副本出现偏离，以防所有副本同时失败。很多种分离都能被加入，比如修改内存布局[4,17,36]，修改指令集[2,22]，或者甚至运行多个独立的程序版本[1]。我们的注意力集中在保证至少一个正确的副本能够正确运行[11,39]。

基于副本的系统中最类似于Frost就是那些使用不同的事件来改变调度的系统了，比如改变信息或者信号的顺序[36,47]或者改变进程的优先级[36]。Frost对这个领域做的贡献就是补充调度，它描述了如何使用补充调度去检测数据竞赛，并且展示了如何使用没有中断的调度和单并行来构造补充调度。

控制线程调度的思想也被用来探索可能的线程交叠，在模式检查和程序检测中[18,30]。这些早期工作的目标是探索所有可能的行为，以此来寻找故障。相反的，Frost的主要目标是保证至少一个线程调度以不会错误的顺序执行了会产生竞赛的指令。这改变了用来产生调度的算法，并且让Frost只使用两个互补的调度，而不是更多的调度。类似Frost，CHESS[30]使用无中断的调度来对线程调度进行比较严格的控制。不过，因为CHESS只是用来做测试的，它不需要像Frost一样并行化无中断的执行。

之前的研究已经发现了几个不需要活跃副本的生还并行故障的会造成死锁的方法[21,46]。Frost不同于这些方法，因为它目标一类不同的由于数据竞赛的并行故障。最近的研究提出要主动没有检测过的线程交叠来减少出现并行故障的概率，而不是检测到错误才想办法如何去恢复。不过这个方法会有高能耗[12]或者需要处理器的支持[51]。其他的研究发现一些并行故障可能用消除中断或者提供顺序语义[3]来消除。其他的系统[48]通过躲避引发错误行为的线程来躲避已知的故障。不

像Frost，它们不能第一次就生还未知的故障。

5.2 数据竞赛检测

除了生还数据竞赛的能力，Frost也能作为动态数据竞赛的检测工具，能在生产或者测试环境中使用。数据竞赛检测器能够在很多个维度上被比较，包括能耗，覆盖面（检测出多少数据竞赛），准确性（检测出多少无害的竞赛）和信息量（提供了多少关于竞赛的信息）。

静态的竞赛检测器（如[13])试着去证明一个程序没有数据竞赛；他们没有引入运行时间上的消耗，但是报告了很多假的阳性（减少了正确率），这是由静态分析的不足造成的，对于结构性较差的语言，如C，更是如此。另一方面，动态竞赛检测器只希望某次运行出现数据竞赛时将其检测出来；他们必须观察潜在的会发生竞赛的指令对。早期的动态数据竞赛检测器一般是基于两个基本的技术：发生先后分析[23,42]和锁集分析[40]。这两种技术都分析了程序发出的同步和内存操作，以此来决定一个数据竞赛是否发生了。因为内存操作发生地很频繁，动态数据竞赛检测器在历史上将程序的速度降低了一个数量级。在最近的研究中，Flanagan和Freund[15]比较了几个标准的动态数据竞赛检测器并且展示了他们最好的JAVA实现是原本实现的8.5倍慢。对于一个结构性比较差的经过优化的不在虚拟机中跑的程序（比如C或者C++）可能会有更加高的能耗，最近发布的工业竞赛检测器（因特尔[38]和谷歌[43]）就是典型，会引入30倍的功耗。

动态竞赛检测器可以实用语言或者运行时的特性去减少功耗。RaceTrack[52]在微软CLR上跑了一个CPU密集的数据，慢了有2.6-3.2倍，但它没有检查包含本地代码的竞赛，这就减少了数量客观的方法。RaceTrack同样利用了已经检查程序的面向对象的特性，使用了一个聪明的修改策略，它先检测了对象颗粒的竞赛，然后检查在对象的域颗粒的竞赛。对于对象颗粒的检测可能会出现很多误报，因此在一个比较低的优先级被报告出来。不过，除非一个特定的指令对对一个对象竞赛了两次，RackTrack不能很自信地报告这个竞赛。功耗也能够通过一个分离的静态分析阶段[10]来减少。不过这些优化不能给一些不安全的程序应用。

如果有空闲的核心来并行化副本执行，Frost会有$3-12\%$的减速，当没有空闲的核心，会慢3倍。对于不在虚拟机中运行的一般程序，在这比所有之前的动态数据竞赛检测器都要好很多。虽然Frost可能会错过一些被高能耗的数据竞赛检测器检测出的竞赛，在实际中Frost还是把所有被其他检测器检测出的竞赛都检测了出来。

为了减少能耗，一些近期的竞赛检测器使用采样来和覆盖面做交换。PACER[6]使用随机采样，因此有和采样率几乎一样的覆盖面。在3%的采样频率下，PACER跑CPU密集的程序有1.6-2.1倍慢。不过，在这个频率下，PACER只报告了$2-20\%$的动态竞赛。LiteRace[29]使用了一个机制（适应性地将采样频率提高）来提高期望的竞赛发现，但这个机制可能会使得某些竞赛不容易被发现（在经常执行的函数中不经常出现的指令序列）。LiteRace运行CPU密集的程序时，慢了2.4倍，找到了所有竞赛的70%，和稀有竞赛的50%。

采样对于大部分数据竞赛技巧来说是垂直的。Frost通过只检查一部分的小片来实施采样。在一个稍大于3%的采样频率下，对于一个CPU有界的数据Frost的功耗只是17%。同样可以使用和LiteRace类似的机制，但程序被Frost的小片颗粒给复杂化了。虽然LiteRace在函数颗粒性上看的很紧，Frost只能保证小片颗粒。不过，Frost能从它的线程级并行中获益，比如造选择查看那个小片之前，可以通过检查小片级并行的执行来查看冷代码的百分比。

可以通过使用自定义硬件[35]来更多地减少动态数据竞赛检测的功耗。Data Collider[14]给现存的硬件进行了重编程（查看点），获得了一种全新的动态竞赛检测技术。Data Collider通过暂停当前访问内存的线程来对内存访问进行采样，并且使用其他线程使用的内存地址来标识没有同步的内存访问。在现存处理器上的硬件检查点的缺乏（在他们的实验中有4个）限制了可以同时采样的内存地址。Data Collider因此可以获得非常低的功耗（一般小于10%），但可能无法取得比较好的覆盖面，因为采样率太低了。Data Collider是否会随着核心的增长而扩展也不得而知，因为每个核心的检查点不会增长，而采样一个地址需要给每个核心一个IPI去设置一个检查点。

大部分的数据竞赛不是故障。先前的工作已经展示比较不同的冲突的内存访问指令的顺序产生的输出可以被用来分类一个竞赛是无害的或者潜在恶性的[31]。这能被看做一个增加准确性的一个方法。Frost的设计使用了这个过滤技术。跟之前的假定数据竞赛已知来生成线程调度的工作不同，Frost使用补充调度来检测未知的数据竞赛。

Frost也有极其高的信息量，因为他能够决定性地回放一个小片直到数据竞赛出现（有时还能更远）。这允许了Frost来生成任何的开发者需要的诊断信息，比如栈轨迹。比如，在4.2节我们就使用了这个能力去搭建了一个完整地动态竞赛检测器。其他的工具，如Intel的Thread Checker[38]提供了两个竞赛线程的栈轨迹，并且一些工具，如RackTrack[52]，能够保证给一个线程生成栈轨迹。

Pike[16]也使用了多个副本来通过通过序列化请求的执行来比较交叠的请求来测试并行故障（被认为是正确的）。Pike需要应用程序提供对其与线程交叠无关的状态一个规定好的表示，这需要耗费很多时间。Pike同样有很高的功耗（需要一个月来测试一个程序），但除了竞赛故障还能找到更多类型的并行故障。TightLip[53]比较了一个访问敏感数据的副本和一个没有访问的副本来检测信息泄露。

6 总结

Frost引入了两个主要的思想来减轻数据竞赛的问题：补充调度和基于输出竞赛检测。使用补充调度运行多个副本保证了对于大部分类型的数据竞赛故障，至少一个副本能躲过使竞赛指令出错的顺序。这个性质使我们做成了一个新的，快速的动态数据竞赛检测器，他比较不同副本的输出，而不是分析触发的事件。在Frost检测到一个数据竞赛之后，它分析了结果的组合并且选择了可能生还竞赛的策略。

7 鸣谢

我们感谢没有署名的评论和我们的监护人，Tim Harris，他的评论改进了这篇文章。Jie Hou和Jessica Ouyang帮助收集了这篇文章的数据。这个工作被国家科学基金支持，CNS-0905149和CCF-0916770。这篇文章的观点和结论是作者个人的，而不应该被认为是官方的，即NSF，密西根大学，美国政府，或者工业上的赞助。

8 引用

[1] A. Avizienis. The N-version approach to fault-tolerant software. IEEE Transactions on Software Engineering, SE-11(12):1491 – 1501, December 1985.

[2] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), Washington, D-C, October 2003.

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for C/C++. In Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications, pages 81 – 96, Orlando, FL, October 2009.

[4] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada, June 2006.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In Proceedings of the International Conference on Object Oriented Programming Systems, Languages, and Applications, pages 97 – 116, Orlando, FL, October 2009.

[6] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation, Toronto, Canada, June 2010.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In Proceedings of the 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications, pages 211 – 230, Seattle, WA, November 2002.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. Addison-Wesley, 1993. in Distributed Systems, edited by Sape Mullender.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A technique for cheap recovery. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pages 31 – 44, San Francisco, CA, December 2004.

[10] J.-D. Choi, K. Lee, A. Loginov, R. O' Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[11] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A

secretless framework for security through diversity. In USENIX Security, August 2006.

[12] H. Cui, J. Wu, C.-C. Tsai, and J. Yang. Stable deterministic multithreading through schedule memoization. In Proceedings of the 9th Symposium on Operating Systems Design and Implementation, Vancouver, BC, October 2010.

[13] D. Engler and K. Ashcraft. RacerX: Efficient static detection of race conditions and deadlocks. In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pages 237 - 252, Bolton Landing, NY, 2003.

[14] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In Proceedings of the 9th Symposium on Operating Systems Design and Implementation, Vancouver, BC, October 2010.

[15] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, pages 121 - 133, Dublin, Ireland, June 2009.

[16] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In Proceedings of the European Conference on Computer Systems, Salzburg, Austria, April 2011.

[17] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In Proceedings of the 6th Workshop on Hot Topics in Operating Systems, pages 67 - 72, Cape Cod, MA, May 1997.

[18] P. Godefroid. Model checking for programming languages using VeriSoft. In Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 174 - 186, Paris, France, January 1997.

[19] R. Huang, D. Y. Den, and G. E. Suh. Orthrus: Efficient software integrity protection on multi-cores. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 371 - 383, Pittsburgh, PA, March 2010.

[20] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In Proceedings of the 25th International Symposium of Fault-Tolerant Computing, pages 381 - 390, Pasadena, CA, June 1995.

[21] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In Proceedings of the 8th Symposium on Operating Systems Design and Implementation, pages 294 - 308, San Diego, CA, December 2008.

[22] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), Washington, DC, October 2003.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558 - 565, 1978.

[24] D. Lee, B. Wester, K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Respec: Efficient online multiprocessor replay via speculation and external determinism. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 77 - 89, Pittsburgh, PA, March 2010.

[25] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. IEEE Computer, 26(7):18 - 41, 1993.

[26] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, October 2000.

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes —a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 329 - 339, 2008.

[28] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development, 6(2):200 - 209, 1962.

[29] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: efficient sampling for lightweight data-race detection. In Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, Dublin, Ireland, June 2009.

[30] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Proceedings of the 8th Symposium on Operating Systems Design and Implementation, pages 267 - 280, San Diego, CA, December 2008.

[31] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, CA, June 2007.

[32] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execu-

tion in a distributed file system. In Proceedings of the 20th ACM Symposium on Operating Systems Principles, pages 191 – 205, Brighton, United Kingdom, October 2005.

[33] K. Poulsen. Software bug contributed to blackout. SecurityFocus, 2004.

[34] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 179 – 190, San Diego, CA, June 2003.

[35] M. Prvulovic and J. Torrellas. ReEnact: using thread-level speculation mechanisms to debug data races in multithreaded codes. In Proceedings of the 30th Annual International Symposium on Computer architecture, pages 110 – 121, San Diego, California, 2003.

[36] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In ACM Symposium on Operating Systems Principles, pages 235 – 248, Brighton, United Kingdom, October 2005.

[37] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. ACM Transactions on Computer Systems, 17(2):133 – 152, May 1999.

[38] P. Sack, B. E. Bliss, Z. Ma, P. Petersen, and J. Torrellas. Accurate and efficient filtering for the Intel thread checker race detector. In Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, pages 34 – 41, San Jose, CA, October 2002.

[39] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In Proceedings of the European Conference on Computer Systems, April 2009.

[40] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems, 15(4):391 – 411, November 1997.

[41] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys, 22(4):299 – 319, December 1990.

[42] E. Schonberg. On-the-fly detection of access anomalies. In Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, Portland, OR, June 1989.

[43] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, December 2009.

[44] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Long Beach, CA, March 2011.

[45] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. Automata Studies, pages 43 – 98, 1956.

[46] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In Proceedings of the 8th Symposium on Operating Systems Design and Implementation, pages 281 – 294, San Diego, CA, December 2008.

[47] Y.-M. Wang, Y. Huang, and W. K. Fuchs. Progressive retry for software error recovery in distributed systems. In Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 1993.

[48] J. We, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In Proceedings of the 9th Symposium on Operating Systems Design and Implementation, Vancouver, BC, October 2010.

[49] M. Xu, R. Bodik, and M. D. Hill. A serializability violation detector for shared-memory server programs. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, June 2005.

[50] J. Yu. Collection of concurrency bugs. http://www.eecs.umich.edu/jieyu/bugs.html.

[51] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In Proceedings of the 36th Annual International Symposium on Computer Architecture, pages 325 – 336, June 2009.

[52] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In Proceedings of the 20th ACM Symposium on Operating Systems Principles, pages 221 – 234, Brighton, United Kingdom, October 2005.

[53] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping applications from spilling the beans. In Proceedings of the 4th Symposium on Networked Systems Design and Implementation, pages 159 – 172, Cambridge, MA, April 2007.