# 数据竞争的研究现状

随着计算机技术的飞速发展，各式各样的先进的计算机技术不断被人提出，并实现。被广大编程人员所使用。其中多线程技术便是其中的一种很重要的技术。多线程技术，既允许一个程序同时执行多个任务，多线程是当今软件技术的又一重要成果，已成功应用于操作系统、应用开发等多个领域。在应用软件上，我们所使用的常用的操作系统，如Windows与Linux；常用的软件，如Google Chrome，Office等，均使用了多线程编程技术。在硬件上，Intel与AMD都推出了各式多核处理器，同时这些多核处理器也更好的支持了多线程技术，从而使原本在单核处理器上运行的多线程并行程序可以在多核上更加快速、流畅的运行。

然而，在多线程编程为程序运行速率带来好处的同时，编写多线程程序仍旧是一件比较令程序员头疼的事情。首先，多线程程序执行调度的不确定性使得多次执行一个相同的多线程程序有可能会产生不同的测试结果；其次，这种线程调度的结果组合的数目将会是非常的巨大，从而想要测试每一种调度组合是不太可能完成的任务。

在多线程程序设计中，存在一种很难被检测与调试的故障。这种故障是很难通过一两次的测试运行程序而发现的，而且就算是检测出有这种故障，对于这种故障的调试信息也不容易被获得。鉴于多线程程序的这种故障种类有别于传统程序的故障种类，而多线程技术又能给程序员带来许多的便捷之处，国内外的研究人员为此设计了一系列的检测算法，用于检测多线程程序中产生的这种故障—多线程的数据竞争检测。

数据竞争故障由于开发人员使用同步保护机制不当最容易发生，但是该故障很难通过一两次运行程序而发现，并且纵使发现有数据竞争故障，针对该故障的调试信息也非常难以获取。

其中同步保护机制是针对程序中临界资源而言的，因为多个线程能够并发的访问临界资源而由于线程访问顺序的不确定性，所以会导致程序最终运行结果不确定。为了解决这种程序运行结果不确定性，操作系统或编程语言无一例外采用了同步保护机制(锁机制)来对临界资源加以排他性的锁，通过加锁后的临界资源在任意时刻只能有一个线程中的同步方法访问，但这种对临界资源访问的序列化将会严重的影响程序的性能。由于线程之间的这种调度的不确定性，数据竞争有时会需要几周甚至几个月才能被检测出来[1]。因此，如何提高多线程程序的质量，有效的检测多线程程序中所存在的数据竞争故障是非常值得研究的问题。

多年来，许多专家学者在多线程程序数据竞争故障动态检测技术的研究领域，取得了不少研究成果。按照竞争检测的依据，数据竞争可分为基于线程发生序列（Happens-Before）的检测算法、锁集（Lock-Set）的检测算法以及混合型的检测算法。按照竞争检测的时机，数据竞争检测又可分为动态数据竞争检测和静态数据竞争检测以及将两者相结合的分析技术。

采用静态分析进行检测的工作主要是对程序执行状态进行抽象，计算程序点具有的属性，静态模拟并行线程间的交互，从而检测数据竞争等并行缺陷。一类方法是基于类型系统的，通过自动产生用于分析缺陷的类型信息来进行检测。该类代表性工作包括：Agarwal等人提出的系列方法PRFJ、EPRFJ、EPAJ、DEPAJ等；Abadi等人提出的Houdini/rcc方法；Boyapati等人提出的Safe Concurrent Java方法。Grossman等人的工作；Matsakis等人的工作。其余采用静态分析技术的工作主要基于数据流分析等技术。代表性工作包括：Flanagan等人提出的工具RccJava，该工具手动向源代码中插入相关标记，使其不运行程序即可检测程序中的数据竞争问题；Kalllon等人提出的快速精确的数据竞争检测方法；吴萍等人的方法，该方法基于精确的上下文敏感和流敏感的别名分析，分析交互原语形成的访问事件发生序，结合Escape分析缩小检测范围来提高分析效率；Naik等人提出的工具Chord；Voung等人提出的工具RELAY；Rerauchi等人设计的工具Locksmith；Chugh等人提出的工具RADAR；Terauchi等人的工作；KaIllon等人的工作。静态数据竞争检测实在程序编译时或者编译后对源程序或者目标代码进行检测，需要分析整个程序，不像动态数据竞争检测，其具有无需插桩代码，不需要占用程序的运行时间，不受程序规模限制等优点。但是由于静态分析的不可判定行，对于任何一个有一定规模的软件，希望获取有关它的全面描述通常是不现实的，往往采用不完备的近似算法，静态分析的结果相对比较保守。

动态数据竞争检测一般采用插桩技术，从而可以活的变量和别名的准确信息，准确的捕获内存访问的发生序列以及锁集的信息，得到运行中（on-the-fly）或者运行后（postmortem）的分析结果，并在此基础上检测数据竞争等并行缺陷。这类工作根据检测的精度和是否产生大量误报，可以大致分为早期基于vector clock的方法和后期基于lockset的方法。前者在检测京都上优于后者，但在性能上则远远不如后者。

本文主要采用基于动态的数据竞争检测方法，重点分析以下国内外动态数据竞争检测研究成果：

Hoare提出了监视器原则用以检测数据竞争[2]，该原则是第一个用于检测数据竞争故障的方法。其主导思想就是建立一个类似于监视器的模块，将每个变量在程序运行之前均加上锁，每次方法访问变量均需要获取和释放变量的锁，而监视器则监视程序中所有方法调用是否按照规则(访问变量时获取锁，离开方法体时释放锁)运行。

Dinning和Schouberg[3]、Mellor-Crummey、Netzcr[4]以及Perkovic和Keleher[5]基于Lamport的happens-before[6]关系对数据竞争故障检测进行了研究。Happens-before关系主要包含了两个要求，第一个要求是在同一个线程内部程序的执行是按照时序逻辑，即顺序执行的，第二个要求是在线程间程序的执行是依赖于同步的，如果违背了上述两个要求之一，便会产生数据竞争故障。Happens-before关系能够很好检测出现行多线程程序中包含的数据竞争故障，但是它也是有缺点的，它的缺点在于该关系高度的依赖于线程调度所制造的交错(Interleaving)顺序。因而想要完全的检测出待测程序中的数据竞争故障不仅需要大量的测试用例还需要依靠线程调度所制造的不确定的线程间交错顺序。

Stefan Savage提出了lockset(锁集)算法[1]，该算法定义了变量和线程的锁集，通过待测程序运行时对变量和线程的锁集进行相对应的操作，并依据变量的锁集是否为空作为判定待测程序是否包含数据竞争故障的依据，并根据该算法在Linux平台上实现了相关检测工具Eraser。Lockset算法较之happens-before关系在检测效率上有了显著的提高，首先它不依赖于大量的测试用例，其次它的检测效率在实验对比中要优于happens．before关系。但它也存在误报率较高、缺乏动态内存变量检测的能力和动态检测时获取现场数据不足导致检测出故障无法提供有效信息等缺陷。

Klaus Havelund在lockset的基础上做出了部分改进，即通过静态分析程序源代码来获取变量内存分配信息，使其能够对动态内存申请的变量进行检测，并在此基础上使用JAVA完成了检测算法的实现[7][8]。虽然Klaus做出了部分改进，但是其动态测试是在分析JAVA程序字节码的基础上模拟程序的运行来检测数据竞争故障，严格意义上不能算作动态检测，这相比Stefan的实际运行待测程序完成检测有所差距，因为仅通过模拟的方式无法准确的反映数据竞争这种运行时的并发故障。

Robert 0’Callahan和Jorig-Deok Choi[9]在happens-before关系与lockset基础上提出了内folk theorem通用框架，其主要思想是取二者检测的交集，通过happens-before关系来减少locksct的误报率，而]ockset解决了前者的检测效率问题。

Post-mortem[10]和on-the-fly[11][12][6][15]方法提高了数据竞争故障检测的精确度，拥有误报率低的优点，但是由于是离线分析程序运行结果导致其未能充分检测出待测程序中的数据竞争故障。

Liqiang Wang和Scott D.Stoller[16][17]将多线程数据竞争问题归结成为多线程的原子性问题，并提出了block-based算法加以检测。

董威[18]等提出一种对并发程序进行切片以缩减模型检验状态空间的方法，并针对并发语言SPL研究了用切片思想缩减并发程序模型检验状态空间的方法。

章隆兵[19]等利用存储一致性模型的框架模型，针对域一致性模型提出了增强发生序概念，并依此给出了一种基于锁集合的动态数据竞争检测算法并且在软件DSM系统JIAJIA上的实现获得了很好的效果。

王文文与武成岗[20]等人提出一种对分队陈数据竞争的检测算法-ARace，它使用共享变量保护和写缓冲区来动态容忍和检测非对称数据竞争。ARace不仅可以容忍临界区内和临界区外之间的非对称数据竞争，还可以对并发临界区之间的非对称数据竞争进行检测。ARace既不依赖程序源代码和编译器的支持，也不依赖额外硬件的支持．此外，还提出了一种通过动态二进制插桩技术实现ARace的方法。验结果表明，ARace在保证容忍和检测非对称数据竞争的同时，并未引入很大的性能开销和内存开销。

# 动态数据竞争检测

## Monitor

由Hoare引进的监视器原则[2]是早期避免数据竞争的方法，它是第一个用于检测数据竞争故障的方法。该方法检测数据竞争故障的步骤如下所示：

（1）获得源程序中所有静态变量的信息；

（2）将（1）中获取的所有变量的锁信息注册到监视器中；

（3）运行源程序并启动监视器监视源程序的运行；

（4）对源程序中的每个方法进行监视，如果存在某个方法在访问某个静态变量时却没拥有该变量的锁时，那么监视器就会发出一个数据竞争的警报信息。

监视器的运行流程如图 1所示：

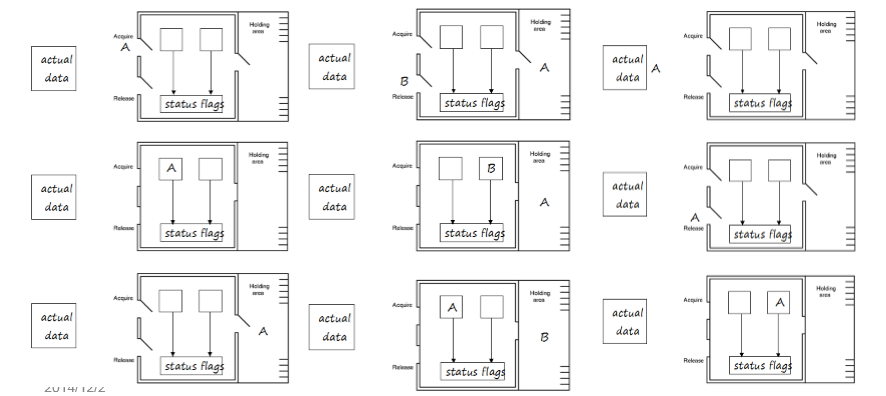


图 1 Monitor访问流程图

即当A来访问变量时，发现当前的共享变量不可用，正在被B访问，于是A就进入等待区等待B来释放该共享变量。当B使用完共享变量之后，会来监视器把共享变量的标志位置为可用状态，之后当A监视到变量可用时，再进入监视器，把变量置为不可用随后进行访问。这样一来，监视器就可以阻止两个线程同时访问到一个共享变量，从而防止了数据竞争的产生。

监视器原则能够很好的检测出数据竞争故障，但是它却有以下两点不足：一是由于该原则设计时处于面向过程程序设计的年代，因而所监视的变量均是静态变量，面对现行的程序设计语言，如C／C++或JAVA，均是在运行时期分配变量空间，从而该原则的无法在程序运行之前获取所有变量的锁，导致监视器原则无法完成检测工作；二是正因为该原则只能监控静态变量，从而监视器原则只能对多线程程序提供编译时期程序变量的数据竞争检测，但是无法完成运行时新增入堆内存变量的检测工作。

## Lamport timestamps

在动态分配内存的编程语言逐渐流行的时候，由于Hoare所提出的Monitor无法检测动态内存变量的问题，Lamport提出了一种Happens-before关系以及Vector clocks结构。在分布式计算环境下，对于动态数据竞争的检测方法大都是基于Lamport的理论。

### Happens-before

HB关系[6]（Happens-before relation，记作→）是一个偏序关系，它是表示在程序运行过程中，两个事件之间的关系，即这两个事件在现实中所执行的顺序。在同一线程内，事件的执行顺序是由它们发生的先后顺序决定的。即在线程1中，若事件a先于事件b执行，那么必有在线程1中事件a比事件b先发生。在不同的线程中，事件的执行顺序是由它们对线程中同步对象的访问权决定的。即如果线程1中的事件a先于线程2中的事件b执行，那么必有事件a先于事件b获得同步对象的访问权。

Happens-before关系的具体定义如下：

如果两个事件满足以下三个条件中的一个，那么它们存在Happens-before关系（定义->表示Happens-before）：

* 在同一个线程中，如果事件a比事件b先被执行，则 a ->b；
* a是一个消息的发送进程，b是同一消息的接收进程则a->b；
* 如果存在事件c，使a ->c，c->b，那么则有a ->b。

从Happens-before关系的定义可知它是一个偏序关系。



图 2 Happens-before关系图

由图 2可知，线程1执行的三条语句是具有Happens-before关系的，这是由于它们在同一线程中的连续性决定的。同理，在线程2中的三条语句也是具有Happens-before关系的。线程2的lock(mu)语句和线程1的unlock(mu)语句是具有Happens-before关系的，因为对于同一把锁，只有先前的拥有者释放之后才能被获取，这个类似于消息的发送和接收（Happens-before关系定义的第二条）。

具有Happens-before关系的事件是有先后关系的，反之，如果两个事件不具有Happens-before关系，那么它们是没有先后关系的，一般称之为并发事件。如果存在两个并发事件同时访问了同一个共享变量，且其中至少一个是写操作,则认为在该共享变量上存在数据竞争。基于Happens-before关系的数据竞争检测器就是基于上述原理进行数据竞争检测的。

基于Happens-before关系的数据竞争检测有效的克服监视器原则不能检测动态分配的共享变量的缺点，但它也存在缺陷。它的缺陷主要是该方法高度依赖线程调度者产生的交叉执行，需要大量的测试用例来验证每一种可能的线程调度。

### Vector clocks

Vector Clock[6]是逻辑时钟的一种实现方式，其目的是在不依赖于物理时钟确定在离散系统中各个事件的发生次序，并且基于这种两两事件之间的先后次序推导出多个线程上发生的各个事件的整体次序关系（Total Ordering）。

Vector Clock实际上是一个(node,counter)对列表(即(节点，计数器)列表)。它是与每个对象的每个版本相关联。通过审查其向量时钟，我们可以判断一个对象的两个版本是平行分枝或有因果顺序。如果第一个时钟对象上的计数器在第二个时钟对象上小于或等于其他所有节点的计数器，那么第一个是第二个的祖先，可以被人忽略。否则，这两个变化被认为是冲突，并要求协调。

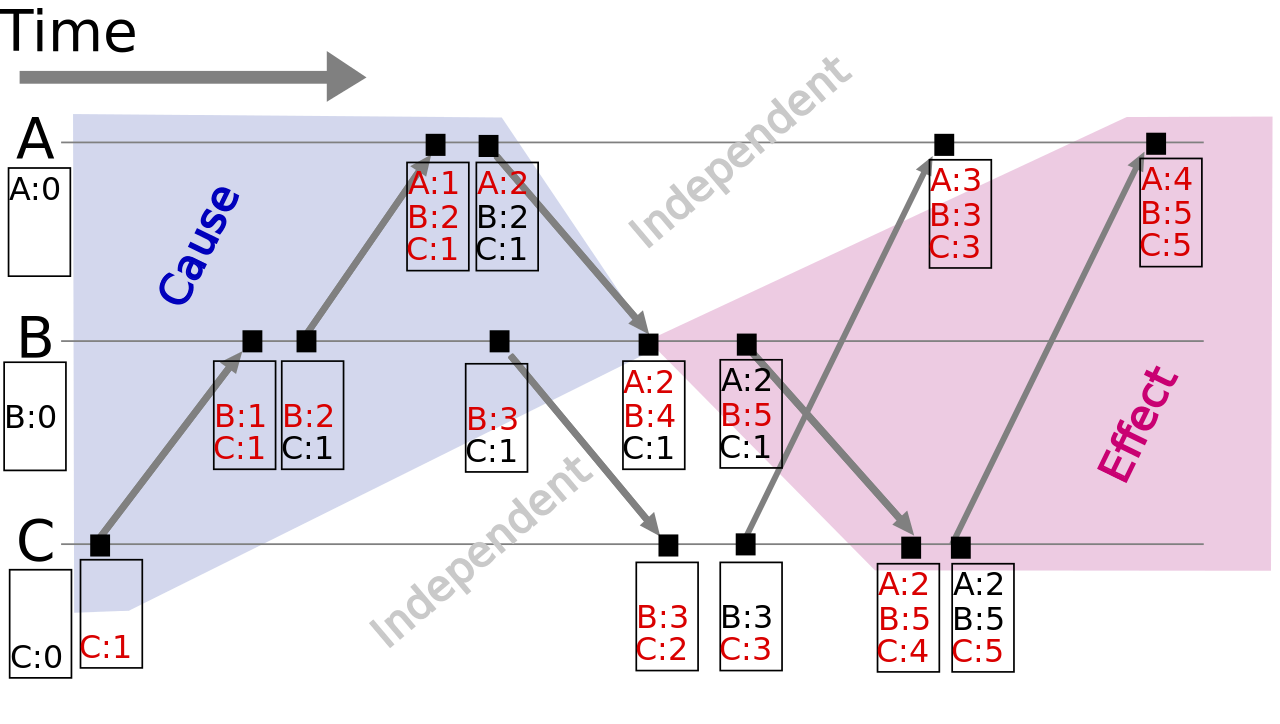


图 3 Vector Clock图

如图 3所示，我们假设A,B,C为三个并行执行的线程，每个线程都拥有自己的时钟向量，时钟向量定义为一个以为的向量(a，b，c)。我们一开始，把每个线程的时钟向量都初始化为0。每一个事件节点的并发顺序就是通过这些时钟向量来判断。每当线程发生一个内部事件时，会更新其代表本线程的时钟向量，如上图所示，单独看A线程上的事件节点中表示A线程事件发生事件的时钟在依次加一。而当线程之间发生同步事件时，即线程之间通信或者抓放同一把锁时，同样也会更新时钟向量。每次接收到同步消息时，首先会增加代表本线程的时钟向量，并且会更新以两个向量中最大的值更新其他的元素。如B线程的(0,2,1)事件与A线程的(1,2,1)事件的同步过程为，代表A线程本地的时钟向量从0加一变为1，这是，B线程的(0,2,1)事件与之发生同步关系，A线程事件的时钟向量为(1,0,0)而B线程事件节点的向量为(0,2,1)，所以更新了A线程事件节点的后两位。从图中我们可以看出，所有蓝色区域中的节点都是发生在B线程事件节点(2,4,1)之前的，而红色区域的事件是发生在B线程事件节点(2,4,1)之后的。

## Lockset

程序中的数据可以划分为变量和常量两大类，而数据竞争只可能发生在对某些共享变量的访问上，并不是对所有变量的读写都会发生数据竞争．在程序中，某些变量除了在定义时执行写操作外，其值一直保持不变，直到程序运行结束，因此线程对它们的访问只是读操作，是不会发生数据竞争的；而另一些变量的值会随着程序的向前推进而不断变化，如果它又为几个线程所共享的话，是可能发生数据竞争的。为了讨论方便，对于每一个变量v，可以根据它在程序运行过程中的表现将其所处的状态划分成以下几种：

初始态：当变量v刚刚被定义时所处的状态，它是一个瞬间状态．一俟创建完毕，即转入私

有态。

私有态：当变量v仅被某一线程读访问时所处的状态，此时变量v为该线程所私有。

共享读态：当变量v被两个以上的线程读访问后所处的状态。此时变量v为多个线程所共享。

共享写态：当变量v被读访问以外的线程写访问后所处的状态。此时变量v为多个线程所共享。

图 4是状态迁移图：



图 4 Lockset状态迁移图

根据上述对变量在程序运行期间状态迁移的分析，可以将程序运行期间的变量分为三类：

初始类：处于独享态的变量；

共享读类：处于共享读态的变量；

读写锁类：处于共享写态的变量。

显然，对于前面两类变量是不存在读写冲突的，因此对这两类变量在运行期间可以不用作任何特殊地处理，仅把它们当作一般的变量进行处理即可。但对于最后一类的共享变量却要好好地处理才行。

锁集算法有关的定义：

对于每个共享变量v，候选集合C(v)定义为程序运行到目前为止始终保护共享变量v的所有锁的集合。

对于每个线程t，locks\_held(t)表示在任意给定时刻线程t所获取的所有锁的集合。

锁集算法具体步骤如下所示：

（1）初始化：对于每个共享变量v，C(v)初始化为所有可能的锁的集合；

（2）locks\_held(t)初始化为空；

（3）线程t每次访问共享变量v时：

C(v) = C(v)∩locks\_held(t)；

如果C(v)为空集，则发出数据竞争的警告。

图 5显示了锁集算法检测数据竞争的过程。这个例子有两把锁，所以C(v)初始化为{mu1,mu2}，且locks\_held初始化为空。当抓住锁mu1且第一次访问共享变量v时，locks\_held变为{mu1}，C(v)被精简为{mu1}；当抓住锁mu2且第二次访问共享变量v时，locks\_held变为{mu2}，C(v)被精简为{}（即{mu1}∩{mu2}），表明没有锁一直保护变量v，故发出一个数据竞争的警报。

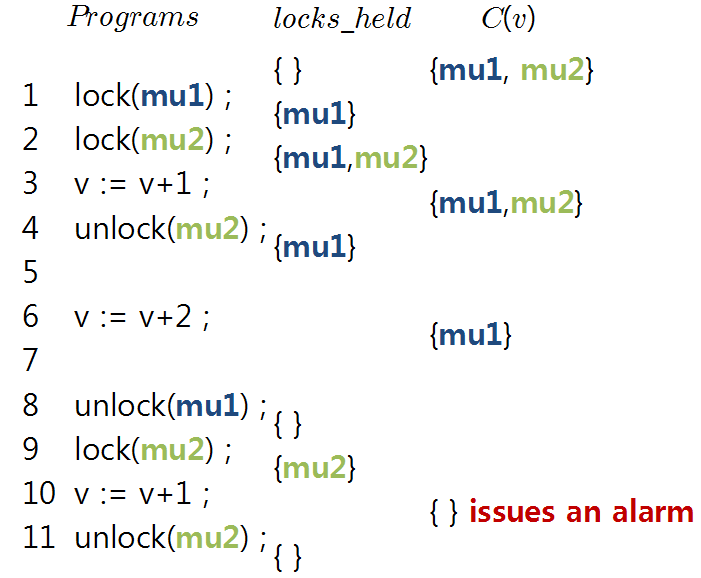


图 5 Lockset算法实例

# 插装技术研究

## 动态二进制插桩技术

随着软件复杂性的提高，插桩（一种向应用程序插入额外代码来观察程序行为的技术）变得越来越重要。插桩能够在以下四个不同阶段执行：源代码，编译期间，链接期间和运行期间。Pin是在运行期间进行二进制插桩的软件系统。

### Pin的特点

Pin[21]支持Windows和Linux操作系统。Pin有如下几个特点：

容易使用：Pin允许在可执行程序的任意位置插入任意代码（由C/C++编写的代码）。而且Pin还提供了非常丰富并且容易使用的API函数，用户可以利用这些API函数记录程序在运行期间的所有体系结构状态：比如寄存器内容、内存和控制流等。同时，Pin还为用户提供了许多与体系结构无关的Pintools（用C/C++编写的插桩工具）样例，包括分析工具、缓存仿真器、跟踪分析器以及内存漏洞检查等。

高效性：Pin使用了一种即时（JIT）编译器来插入和优化代码。除了代码缓存和跟踪轨迹链接外，Pin还实现了寄存器重分配、内嵌代码、现场分析以及指令调度来优化即时代码。就像一般的调试工具一样，Pin也能够嵌入一个程序，实现对该程序的插桩操作，然后收集一些与该程序相关的信息，最后与程序分离。应用程序仅仅只增加了Pin嵌入该程序期间的那部分插桩开销。对于那些大型、长时间运行的应用程序而言，这种能够随时嵌入与随时分离的能力是非常必要的。

健壮性：Pin的基于JIT的插桩使得代码插桩延迟到了运行期间，相比于使用静态插桩的系统，Pin会更加健壮。Pin能够完美地处理混合的代码和数据、变长指令、静态未知的间接跳转目标、动态加载库以及动态生成代码。

透明性：Pin提供的插桩透明性使得应用程序能够保持原有的程序行为。应用程序能观察到相同的地址（包括指令和数据）和相同的值（包括寄存器和内存），就好像该程序没有进行插桩一样。透明性使得插桩收集到的信息更有意义，它对于程序运行的正确性也是必需的。

### Pin的系统架构

图 6显示了Pin的系统架构。从图中可以看出Pin由三部分组成：虚拟机、代码缓存和Pintool调用的插桩API函数。下面本文将对这三部分做一个简单的介绍。

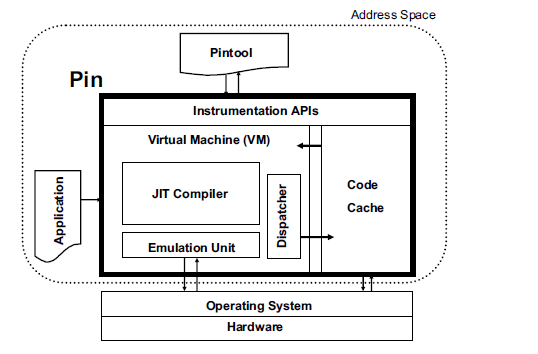


图 6 Pin系统架构图

虚拟机（VM）：虚拟机由JIT编译器、模拟器和分配器组成。JIT编译器主要是对源程序指令进行即时地重编译；仿真器是是用来解释一些不能被直接执行的指令；分配器则是用来启动JIT编译器和检测程序代码的。

代码缓存（Code Cache）：代码缓存是用来存放那些由JIT编译器生成的新的二进制代码。

插桩API函数：这是Pin提供给Pintool的通信接口，与Windows提供的API函数作用类似。

### Pin的插桩

Pin提供的丰富的API函数使得它能够观察到程序的所有体系结构状态，比如内存、寄存器内容和控制流等。它使用的模型和ATOM[15]类似，在该模型中，用户能向应用进程添加程序（和ATOM中的分析例程一样），并且通过编写插桩例程来确定调用分析例程的位置。分析例程的参数可以是体系结构状态或者常量。

Pin是由一个即时编译器编译执行的。该编译器不是输入字节码，而是一个本地的可执行文件。在图 2-2中我们列出了一份Pintool的源代码，该代码可以在\source\tools\ManualExamples目录下的pinatrace.cpp文件中找到，本文只是列出源代码中的一部分（本文将记录程序内存写指令的部分删除掉了）。该Pintool的主要功能是为可执行文件中的每一个内存读写访问打印出一条指令指针值（即PC值）和内存地址的踪迹。主函数首先对Pin进行了初试化操作，并注册了Instruction函数，然后去告诉Pin启动可执行文件。当向代码缓存中插入一条新指令时，JIT编译器会调用Instruction函数，并且传递一个句柄给该函数去对指令进行解码。如果该指令刚好是内存读指令和内存写指令的话，那么该Pintool会在该指令运行之前插入一个对RecordMemRead()函数和RecordMemWrite()函数的调用（调用程序的时间由INS\_InsertPredicatedCall()函数中的参数IPOINT\_BEFORE指定，还可以是IPOINT\_AFTER等），并且该函数还传递一个指令指针值（由IARG\_INST\_PTR指定）以及一个内存操作的有效地址（该地址由IARG\_MEMORYOP\_EA和memOp共同指定）。这里使用INS\_InsertPredicatedCall()是为了确保RecordMemRead()和RecordMemWrite()仅在内存指令被断定为真时才会被调用。

Pin的官网上提供了非常有帮助的用户手册，那里有所有Pin提供的API函数的详细介绍，本文在此就不再进行详述。

## 编译时插装技术

LLVM 项目致力于程序整个生命周期，包括编译过程，链接过程，执行过程等阶段的优化，并提供大量用于建立编译器的可重用模块，可帮助减少开发新编译器的工作量。目前编译前端已经支持C、C++等高级编程语言，后端则支持生成C语言代码以及X86、Sparc、PowerPC等处理器的汇编代码。

整个LLVM 编译系统的组成可分成三部分：LLVM 中间代码[22]，用于分析、优化、代码生成等工作的集成库，以及建立在以上集成库基础之上的工具，包括汇编器、链接器、调试器等等。

### LLVM中间代码

LLVM 中间代码是一种采用SSA 形式的IR（Immediate Representation，中间表达），使用的指令集为LLVM 虚拟指令集。该指令集是一个类似RISC（Reduced Instruction Set Computer，精简指令集计算机）的三地址指令集，含有简单的控制指令和使用带类型指针的访存指令，拥有独立于高级语言和目标处理器的语法，易于进行代码分析和优化。使用LLVM 虚拟指令集的LLVM 中间代码可以三种方式存在：存在于内存中的编译器IR、存在磁盘上的字节码（bytecode）、以及可供人阅读的汇编代码。这三种存在方式是同一中间代码的不同表达形式，可分别用于：在编译器执行编译遍时提供方便高效的中间代码转换或者中间代码分析、作为JIT编译器（Just-In-Time Compiler）在本地执行客户处理器代码的格式、以及方便开发人员进行代码调试的汇编代码。LLVM 中间代码由虚拟指令集、中间代码高层结构、以及中间代码类型系统三方面组成。

虚拟指令集可分为两大部分：单指令和内函数（Intrinsic Functions）。其中，单指令即具有一定功能的单一操作；而内函数在使用时以函数形式存在，编译时将被翻译成一条或若干条单指令。单指令包含以下几类：终结指令、算术运算指令、逻辑运算指令、向量运算指令、访存及地址操作指令、类型转换指令及其他指令。其中终结指令是指每个基本块的最后一条指令，该条指令决定了当前基本块结束后将进入的下一个基本块。内函数则包含可变参数处理内函数、精确垃圾回收内函数、代码生成内函数、标准C 库内函数、位操作内函数、以及调试内函数。

构成LLVM 中间代码的高层结构主要有以下这些：模块、链接类型、调用约定、全局变量、函数、模块级内嵌汇编。LLVM 模块是构成LLVM 程序的基本元素，每一个模块都是从输入的程序翻译得到，由函数、全局变量、符号表入口组成。全局变量用于定义在编译时就分配的内存空间，可对它做初始化，并可放置在指定的特定段显示指定对齐。LLVM 函数描述组成一个函数的各个方面，如返回类型、函数名、参数类表、一个包含一系列基本块的函数体，以及可选的链接类型、调用约定、对齐等。每个LLVM 函数定义时均需使用关键字‘declare’，两个函数可以同名，但是此时必须有不同的参数列表。对对上述的全局变量和函数均需指定链接类型，而对每个LLVM 函数以及call 和invoke 指令，均可以可选地指定一个调用约定。类似于GCC 的文件内嵌汇编，在LLVM中，模块可以内嵌模块级汇编。语句“module asm "xxx yyy zzz"”即为内嵌模块级汇编的语法格式，其中"xxx yyy zzz"为需内嵌的汇编代码。

普通的无类型的三地址中间表达必须在转换之后才能进行优化，而LLVM 提供了一个较完备的类型系统，这使得在进行任何转换之前就可以在LLVM 中间代码之上可以进行许多优化。LLVM 类型系统由两大部分组成：原子类型以及衍生类型。

原子类型是建立LLVM 类型系统的基础类型，目前已有的类型可见表格 1：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 描述 | 类型 | 描述 |
| void | 空值 | bool | 布尔类型 |
| ubyte | 无符号8位数 | sbyte | 有符号8位数 |
| ushort | 无符号16位数 | short | 有符号16位数 |
| uint | 无符号32位数 | int | 有符号32位数 |
| ulong | 无符号64位数 | long | 有符号64位数 |
| float | 32位浮点数 | double | 64位浮点数 |
| label | 跳转的目标 |  |  |

表格 1 基础类型表

衍生类型使用以上的原子类型扩展LLVM 类型系统可以描述的类型。这些衍生类型有数组类型、函数类型、结构类型、打包的结构类型、指针类型、打包类型等。

### 集成库

LLVM 提供了许多的集成库实现了编译器开发过程中需要的大部分基本功能，如描述程序基本块、函数、模块的类，程序编译过程中的编译遍添加管理，数据流分析，参数类型转换 (Arg Promotion)，调试，性能估计，以及其他各种优化。

这些集成库可以分为：核心库、分析库、转换库、代码生成库、目标处理器库、运行库。其中核心库主要用于中间代码的解析读写及相关处理、分析库提供各种数据分析及控制分析、转换库提供各种优化转换、代码生成库用于目标代码生成、目标处理器库提供已支持的后端描述、运行库提供解释执行机制及虚拟机器引擎。

### 工具集

LLVM 提供了大量现成的工具，方便以此为基础进行的后续开发工作。从总体上来说，这些工具可大致分为基本工具、编译工具、调试工具、后端工具四大类。

目前LLVM提供的工具见表格 2：

|  |  |
| --- | --- |
| llvm-as | 将人可读的.1l文本文件汇编成字节代码bitcode |
| llvm-dis | 将字节代码文件反编成人可读的.11文件 |
| opt | 用LLVM对字节码进行优化 |
| llc | 为一个字节代码文件生成本机器代码 |
| lli | 运行使用JIT生成的字节码的程序 |
| llvm-1ink | 将几个字节代码文件连接成一个 |
| llvm-ar | 字节代码文件 |
| llvm-ranlib | 为llvm.ar生成的文件建立索引 |
| llvm-nin | 在字节代码文件中打印名字和符号类型 |
| llvm-prof | 将“llvmprof.out”生成的数据格式化成人可阅读的格式 |
| llvm-1d | 带有可装载的运行时优化支持的通用目标连接器 |
| llvm-config | 打印LLVM编译选项、库等的设置信息 |
| llvmc | 一个通用的可定制的编译器驱动 |
| llvm-diff | 比较两个模块的相同和不同之处 |
| bugpoint | 自动案例测试减速器 |
| llvm—extract | 从LLVM字节码bitcode文件中解析出一个函数 |
| llvm-bcanalyzer | 字节码bitcode分析器 |
| FileCheck | 文件类型验证器 |
| tblgen | 目标描述阅读器和生成器 |
| lit | 用于运行测试LLVM的集成测试器 |

表格 2 LLVM基础工具表

# 参考文献

1. Savage S. Eraser: A Dynamic Data Race Detector for Multithreaded Programs[J]. ACM Transactions on Computer Systems, 1997, 1(15):391-411.
2. Hoare C. Monitors: an operating system structuring concept[J]. Communications of the ACM, 1974, 17(10):549-557.
3. Dinning A, Schonberg E. An empirical comparison of monitoring algorithms for access anomaly detection[C]. New York: PPOPP '90 Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming, 1990:1-10.
4. Netzer R H B, Miller B P. Improving the accuracy of data race detection[C]. New York: PPOPP '91 Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming, 1991:133-144.
5. Perkovic D, Keleher P J. Online data-race detection via coherency guarantees[C]. New York: OSDI '96 Proceedings of the second USENIX symposium on Operating systems design and implementation, 1996:47-57.
6. Lamport L. Time, clocks, and the ordering of events in a distributed system[J]. Communications of the ACM, 1978, 21(7):558-565.
7. Havelund K. Using Runtime Analysis to Guide Model Checking of Java Programs[C]. Stanford: 7th International SPIN Workshop, 2000:245-264.
8. Bodden E, Havelund K. Racer: effective race detection using aspectj[C]. New York: ISSTA '08 Proceedings of the 2008 international symposium on Software testing and analysis, 2008:155-166.
9. Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks[C]. New York: SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003:237-252.
10. Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks[C]. New York: SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003:237-252.
11. Richards B, Larus J R. Protocol-based data-race detection[C]. New York: SPDT '98 Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, 1998:40-47.
12. Robert O'Callahan, Choi J. Hybrid dynamic data race detection[C]. New York: PPoPP '03 Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2003:167-178.
13. Havelund K, Penix J, Visser W. Runtime Checking of Multithreaded Applications with Visual Threads[C]. Stanford: 7th International SPIN Workshop, 2000:331-342.
14. Bensalem S, Havelund K. Reducing False Positives in Runtime Analysis of Deadlocks[C]. CA United States: TACAS 2003 Conference, 2002.
15. Christiaens M, Bosschere K D. TRaDe, a topological approach to on-the-fly race detection in java programs[C]. JVM'01 Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, 2001:15-15.
16. Wang L, Stoller S D. Runtime Analysis of Atomicity for Multithreaded Programs[J]. IEEE Transactions on Software Engineering, 2006, 32(2):93-110.
17. Wang L. Run-Time Analysis for Atomicity[J]. Electronic Notes in Theoretical Computer Science, 2003, 89(2):191-209.
18. Howard K. AspectC#: An AOSD implementation for C#[C/OL]. [2002-9]. <http://hdl.handle.net/2262/764>.
19. 许育诚. 软件测试与质量管理[M]. 电子工业出版社, 2004-6.
20. 王文文, 武成岗. 动态容忍和检测非对称数据竞争[J]. 计算机研究与发展, 2014, 51(8).
21. Luk C, Cohn R. Pin: building customized program analysis tools with dynamic instrumentation[C]. New York: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005:190-200.
22. Adve V, Lattner C, Brukman M, etal. LLVA: a low-level virtual instruction set architecture[C]. Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on, 2003:205-216.