外文文献标题：Asynchronous Programs with Prioritized Task-Buffers

原文链接： <https://dl.acm.org/doi/pdf/10.1145/2393596.2393652>

具有优先任务缓冲区的异步编程

摘 要

通常我们认为异步软件系统的算法分析是构建可靠软件的一种手段。而设计这种分析的关键挑战之一是确定一个并发模型，该模型不会在实际系统中引入不可行的行为，不会在实际系统中排除实际行为，并将具有挑战性的特性隔离出来，以便分析集中关注。

在现实异步软件的指导下，我们提出了一种并发模型，该模型通过任务优先级和多任务缓冲区丰富了现有的串行任务-缓冲区异步模型[29]。

我们的模型允许非串行执行:具有高优先级的任务抢占低优先级的任务，并且从不同缓冲区提取的任务可以自由地相互交错。对这些特性进行建模允许分析算法检测由于缓冲区间交错和任务中断而导致的异步程序中未被捕获的编程错误，同时正确忽略由于不可行的超出优先级顺序执行而导致的假错误。

除了更精确地捕获现实世界的系统之外，受并发模型启发我们还得出了一种新的分析算法的设计。给定参数K1，K2∈N2,限制缓冲区间任务交错和缓冲区内任务重排序，我们给出了顺序程序的代码到代码转换，然后可以通过现成的程序分析工具进行分析。对于任何给定的参数值，得到的顺序程序编码可能行为的子集，在两个参数都接近无穷大的极限下，顺序程序编码所有行为。我们通过对原型实现进行实验来证明我们的技术的可行性。我们的原型可以与最先进的并发程序验证工具相媲美，并且能够在简化的Windows设备驱动程序代码中正确识别错误，同时忽略不可行的执行。

1. 引言

交互式计算机系统的用户希望应用程序对其操作的反应能力很少或没有延迟。例如，当与图形用户界面(GUI)交互时，，尽管应用程序可能正在执行长时间运行的计算，但每次单击鼠标和击键都希望得到即时反应。类似地，人们期望web服务器立即回复每个HTTP请求，尽管服务器可能正在处理数千个并发请求。为了确保这种低延迟行为，运行这些应用程序的现代操作系统提供了中断和并行化应用程序顺序控制流的机制。硬件事件启动中断当前执行进程的软件处理程序，例如，确保每个按键都立即传达给应用程序。在多处理系统中，逻辑上不相关的任务是并行执行的，例如，将不同的HTTP连接的处理划分到几个核心或处理器上。

传统上，这种响应式软件系统被设计为共享内存多线程程序:一组软件线程——每个线程本质上都表现为一个递归的顺序程序——并发执行，交错地对共享内存进行读写访问。虽然状态简单，但这种并发模型对于必须通过添加同步(如原子锁定指令)来预测和防止不希望的线程交互的程序员来说是复杂的。考虑到过度同步(例如，通过单个全局锁保护所有事务)会阻碍反应性并破坏并行化的机会，这项工作尤其困难。此外，由于底层运行时系统的性能通常很大程度上取决于系统范围内并发线程的数量，因此程序员还必须考虑外部因素，例如运行时平台和其他并发执行程序的存在。

针对多线程的缺点，响应式软件系统的程序员越来越多地采用基于串行任务缓冲区的“异步编程”范式。

从本质上讲，程序员将计算划分为短期运行到完成的任务，这可能会创建额外的任务来异步执行。每个任务在创建时都被放入任务缓冲区中，并且运行时系统确保来自同一缓冲区的任务是可串行化的。近年来，桌面应用程序、服务器和嵌入式系统都广泛采用了这种范式。

现代web浏览器的Javascript引擎[12]，MacOS和iOS中的Grand Central Dispatch [2]， Linux的工作队列[30]，. net中的异步[22]，以及Windows内核中的延迟过程调用[23]都是基于异步编程。即使在单处理设置(即，没有任何并行性)中，像Node.js[7]这样的异步框架也被广泛用于设计极具可伸缩性的(web)服务器。

尽管任务缓冲区提供了高度抽象，但构建可靠的无错误程序仍然很困难。因为程序员可能需要将长时间运行的任务t重组为一系列短时间运行的任务t1;:::;在tj和tj+1之间执行的其他任务可能会干扰t的预期原子计算。此外，跨多个任务缓冲区执行的任务(例如，在多核处理器上)不能保证串行执行，并且可能干扰共享内存访问。因此，算法形式推理是防止代价高昂的编程错误的可靠措施。

为了设计有效的程序分析，我们提出了一个正式的模型来捕捉现实世界中响应式系统的并发性。在我们的模型中，每个任务都有一个相关的优先级级别，任务从多个任务缓冲区执行和发送到多个任务缓冲区。来自每个缓冲区的任务依次执行，一个接一个，将新任务发布到从它们获得的同一个缓冲区，但是不同缓冲区的任务可能会交错。(最初，每个任务缓冲区至少包含一个初始任务。)当一个任务完成时，将从缓冲区中取出一个优先级最高的挂起任务并开始执行。在任何时刻，当任务t1提交了更高优先级的任务t2时，t1被挂起以执行t2。当没有更高优先级的任务剩余时，t1恢复执行。优先级级别和任务缓冲区的数量是有限的，并且是静态确定的。此外，从一个任务缓冲区取出的任务可能不会将任务发布到另一个任务缓冲区;唯一的缓冲区间通信是通过共享内存进行的。

我们的模型扩展了Sen和Viswanathan[29]的单任务缓冲区模型，以更准确地捕获现实世界中响应式异步程序的并发性[2,22,23,30]。在不考虑优先级的情况下，以前的模型允许在实际系统中永远不会出现的打破优先级顺序的执行。通过考虑单个任务缓冲区，先前的模型排除了实际系统中可能出现的缓冲区间任务交织。在形式验证的情况下，前者导致错误地检测到错误，而后者导致未捕获的错误。

虽然典型的程序分析问题对于没有[11,29]甚至有任务优先级[3]的单任务缓冲区模型来说是可确定的，但正交扩展到多个交错的任务缓冲区会使问题无法确定:递归多线程程序[28]很容易被捕获。尽管如此，我们认为多任务缓冲区异步模型非常重要，足以与多线程模型区分开来，原因有二。首先，将具有多个优先级任务缓冲区的异步程序编码为多线程程序需要添加额外的状态和同步，以确保(a)相同缓冲区的任务不会交错，(b)每个缓冲区中只有最高优先级的任务可以执行。

用通用同步机制对这些约束进行编码忽略了更具声明性的程序结构，并导致低效的程序探索(如我们的实验所示)。其次，通过利用实际程序中并发性的内涵结构，我们可以为优先级程序探索获得有用的启发式方法。例如，在上下文边界的前提下[24,26]，我们通过探索任务缓冲区之间相对较少的交替或相对较少的缓冲区内任务重排序(不直接限制执行的任务数量)的程序执行而受益。

本着开发异步程序结构的精神，我们将参数化程序分析简化为顺序程序分析。分析参数K1和K2限制了不同缓冲区任务间的交错次数(K1)和同一缓冲区任务间的重排序次数(K2);当我们增加K1 (resp。， K2)，我们的分析探索了越来越多的缓冲区间任务交织(例如:，缓冲区内任务重排序);当K1和K2都趋近于无穷时，我们的编码编码了所有有效的执行。对于任何给定的参数值，我们简洁地将一组有限的异步执行编码为具有多项式数目(在K1和K2中)的共享变量附加副本的非确定性顺序程序的执行。我们的编码是按照现有的顺序程序缩减的精神进行组合的[4,8,15,18]，因为每个任务缓冲区的执行都是在与其他任务缓冲区隔离的情况下进行的，并且任务缓冲区本身没有显式表示。这样的组合性避免了组合爆炸通过保持其他缓冲区任务的本地状态，甚至通过显式表示单个任务缓冲区的内容来产生。

还原分两步进行。首先，我们将具有多个任务缓冲区的异步程序减少为具有单个任务缓冲区的异步程序，同时保留任务优先级(§4)。我们通过引入共享变量的K1个拷贝的代码到代码转换来实现这一点。每个副本存储一个任务在被其他缓冲区的任务抢占后恢复的值;这些值最初是猜测的，然后进行验证。因此，通过移动到下一个共享变量副本，可以在本地模拟与其他任务缓冲区的交错。在第二步(§5)中，我们将具有任务优先级的单缓冲区异步程序减少为顺序程序。

我们再次通过代码到代码的转换来实现这一点，尽管在这种转换中，我们还为每个优先级级别引入了共享变量的副本。由于我们的翻译针对的是顺序程序，而没有显式表示任务缓冲区，因此每个异步任务发布都大致翻译为同步过程调用。由于低优先级任务的帖子不允许立即执行，因此我们使用额外的共享变量副本来汇总它们的执行，将它们的效果推迟到以后。

•受我们对现实世界的桌面、服务器和嵌入式响应式软件系统(§2)的研究的启发，我们提出了一个并发模型(§3)，该模型自然而准确地捕获了它们的并发行为。

•我们提出了一种增量程序分析技术，通过两步简化为顺序程序分析(§4和§5)。

•我们证明了我们的分析在实践中相对容易实施和有效。我们的原型实现能够发现异步C程序中的错误(§6)，而不会不精确地报告错误，这些错误将被基于现有异步模型的分析检测到。

通过将异步程序转换为顺序程序，我们允许将许多现有的顺序程序分析工具用于(欠近似的)异步程序分析。我们的翻译与程序中存在的数据类型无关，因此能够针对支持任意数据域的分析，例如，布尔程序、带有整数或列表的程序等。

1. 实践中使用的异步

为了构建实用的验证和调试工具，我们必须指定我们希望捕获的并发程序行为。为了更好地理解为什么现有的正式并发编程模型是不够的，我们检查了两个现实世界的应用程序:Windows操作系统中的硬件-软件交互，以及Apache web服务器中的异步多处理。

2.1 Windows内核中的硬件-软件交互

确保Windows内核中高性能硬件交互的主要机制是优先中断级别。在接下来的讨论中，我们将关注优先级递减顺序的三个级别——DEVICE\_LEVEL、DISPATCH\_LEVEL和PASSIVE\_LEVEL。

在DEVICE\_LEVEL运行软件“中断服务例程”(ISR)。连接设备到处理器核心的布尔值“中断线”触发固定ISR:当核心的中断线被触发，并且ISR当前没有运行时，当前运行的代码被中断以执行ISR。由于DEVICE\_LEVEL例程阻止执行任何其他代码，包括调度器，因此DEVICE\_LEVEL例程应该在很短的时间内执行，将剩余的计算委托给异步“延迟过程调用”(DPC)。Windows内核维护一个挂起的dpc队列和Windows调度器的周期性调用，并在DISPATCH\_LEVEL上逐个执行，直到完成，直到队列为空。正常的应用程序运行在PASSIVE\_LEVEL，因此，只在DPC队列为空时执行。与DEVICE\_LEVEL代码一样，dpc不应该休眠或阻塞等待I/O;相反，它们应该通过将另一个DPC排队来继续推迟未来的工作，或者将工作委托给PASSIVE\_LEVEL线程。尽管dpc保证不会与同一核心上的其他dpc或应用程序线程交织，但dpc可以与isr、dpc、Windows调度程序和其他核心的线程并发执行。

除了带来响应性之外，优先级方案还为设备的共享数据提供同步。由于PASSIVE\_LEVEL以上的代码是自动执行的，不会被同级或更低级别的代码抢占，因此从PASSIVE\_LEVEL提升到DISPATCH\_LEVEL将同步单核上的设备访问。

我们的模型通过为每个任务分配三个优先级中的一个，并将来自不同核心的代码划分到不同的任务缓冲区，可以精确地捕捉Windows的这些方面。为了捕获硬件和软件之间任意交错的交互，我们可以为单个旋转硬件模拟任务指定一个单独的任务缓冲区。请注意，忽略优先级可能会在使用基于级别的同步保护的设备数据上导致虚假的数据竞争错误，并且忽略多个缓冲区将会由于在不同的内核(或硬件)之间交错而错过真正的错误。

2.2 Apache中基于GCD的多处理

在最近发布的软件补丁中，曾经多线程的Apache web服务器被重新设计为使用libdispatch并发框架的异步程序[1]。在更新后的体系结构中，应用程序首先创建许多并发执行的连接侦听器对象，每个对象维护一个单独的传入连接请求队列。每个侦听器通过创建一个新的连接对象来处理连接请求，该对象除了存储与给定客户机连接相关的所有数据外，还维护与该连接相关的任务队列。低级网络套接字上的客户端活动触发附加的连接处理任务，这些任务将被放置在队列中。由周期性计时器和服务器响应触发的任务也被放在队列中。

重要的是，尽管负责初始化新连接数据的连接侦听任务分布在多个队列中，但操作任何给定连接数据的任务都放在同一个任务队列中。

底层并发管理器称为Grand Central Dispatch (GCD)，负责执行来自各个队列的任务。GCD确保队列中的每个任务只在前一个任务完成后执行对于Apache，这确保每个连接在任何时刻最多执行一个任务，并允许来自不同连接和连接侦听器的多个任务并发执行。

对于任何有限数量的连接和侦听器，我们的编程模型通过将任务缓冲区关联到每个连接和连接侦听器来准确捕获可能的执行。现有的正式模型是不够的。单缓冲区异步程序无法捕获连接之间和连接侦听器之间可能不安全的交错;在不添加额外同步的情况下，多线程允许同一连接的任务错误地交错访问它们的共享内存。

2.3 任务缓冲顺序的抽象

这些系统很好地符合我们的编程模型，但有一点需要注意:虽然它们的任务缓冲区可能是FIFO顺序的，但我们通过假设任务可以以任何顺序从缓冲区中取出来来抽象该顺序。我们认为这有两个理由。首先，关于进程访问无界有序队列的算法形式推理仍然是一个难题其次，我们对这些特定系统的理解使我们相信，虽然FIFO语义对于确保公平性和反应性很重要，但关键的安全属性可能不依赖于顺序。当然，这不是一个技术限制——如果需要，可以使用共享内存同步对FIFO顺序进行编码(但代价是引入显著的复杂性)。

1. 异步编程

我们考虑一个编程模型，其中计算被划分为多个任务。每个任务都有固定的优先级和与之相关联的任务缓冲区，其行为本质上就像一个递归顺序程序。除了访问由所有其他任务共享的全局内存外，每个任务还可以将其他任务发布到其任务缓冲区以供以后执行。来自每个缓冲区的同级任务一个接一个地串行执行，但会被更高级别的任务中断，并与来自不同缓冲区的任务并行执行。我们将没有当前正在执行任务的任务称为空闲任务，将缓冲区中尚未执行的任务称为挂起任务，已完成执行的任务称为完成任务，并在另一个任务完成分派时选择执行任务。最初，每个任务缓冲区都是空闲的，并且至少包含一个待挂任务。当缓冲区空闲或任务完成时，将调度优先级最高的待处理任务。在任何时刻，当任务t1向更高优先级的任务t2提交任务时，t1被挂起以执行任务t2——我们说t1被t2中断;一旦同一缓冲区的所有高优先级任务完成，t1就恢复执行。

该模型通过增加优先级[3]和多个任务缓冲区，推广了异步程序的经典模型[29]。在这里，只有最高优先级的任务可以被分派，并且来自不同任务缓冲区的任务可以并行执行。尽管我们倾向于保持任务缓冲区与物理实体(如进程、线程、处理器、内核等)断开连接的概念，但第2节描述了在实际异步系统中从这些实体到任务缓冲区的特定映射。

我们对我们的模型采用交错语义:在任何时间点，只有来自单个任务缓冲区的任务可以执行，但是(缓冲区)抢占可以将控制权转移到来自另一个缓冲区的任务。

我们用一个叫做zield的特殊程序语句显式地标记了这种控制权转移的机会。此外，为了进一步扩展我们模型的适用性，我们引入了一个类似的控制转移，称为(任务)抢占，它暂停当前任务的执行，并将控制转移到同一缓冲区的另一个同级任务。我们用一个叫做yield的特殊程序语句来标记这些转移的机会。为这种控制转移提供明确的说明，可以使模型具有很大的灵活性。我们编写原子方法(例如，同步操作)只需在原子部分期间省略yield和yield。

我们通过省略yield(但不省略yield)来模拟DISPATCH\_LEVEL在Windows中的延迟过程调用(参见第2节)。此外，通过在访问共享内存的每个语句之前插入yield，可以简单地对共享内存多线程程序进行建模。

我们的模型假设内存访问顺序一致[20]。虽然大多数现代硬件都暴露了较弱的内存一致性语义，但我们的模型仍然有用:首先，在顺序一致的内存上的bug，依然属于弱内存模型上的bug;其次，我们仍然考虑良好同步程序的所有行为，即没有数据竞争的程序。由于弱内存语义引入了它自己的一组挑战，我们考虑处理这种语义与本文提出的工作正交。

3.1 编程句法

设Procs是一组过程名，Vals是一组值，表达式是一组表达式，N;M 2 N任务缓冲区和优先级级别的数目，如。我们表示集合fn 2n: N < Ng × N。

图1的语法描述了我们的异步程序语言(具有优先级的任务缓冲区)。我们不指定表达式的语法，尽管我们坚持val包含true和false，并且express包含val和(虚值)选择操作符。

一个多缓冲区。(单缓冲区)程序是带有(resp.)的程序。(无)字段语句。一种程序，其中每个发布过程的第一个语句都是while ?Do yield被称为reorderable，在这个程序中，yield语句不会重复执行。(不)发生，否则(包括传递调用的过程)称为抢占式(resp.)。没有yield或zield语句的程序称为调度器相关程序。直观地说，依赖调度器的程序在顺序语句(例如，使用?算子)，并对任务调度点可能做出的不确定性选择取模;在可调程序中，任何已调度的任务都可以立即再次挂起。顺序程序是没有post、yield和zield语句的程序。

每个程序P声明一个共享的type-T全局变量g，以及一个名为p1::: pi 2 Procs \*的过程序列，每个P都有一个type-T形参l和一个顶级语句sp。程序语句集s表示为Stmts。此外，每个程序P声明N个无参数初始过程，命名为main(0)， main(1)，…， main(N−1)，该函数既不发布也不调用;我们将假设初始框架main(n)最初为每个任务缓冲区n2n挂起，直观地说，post (n)语句是对过程p的异步调用，参数e将在优先级m上执行。yield语句将控制转移到相同任务缓冲区的相同优先级的挂起任务，而zield语句将控制转移到另一个任务缓冲区的任务。假设e语句只在e求值为真时才继续;我们将使用该语句来阻止后续缩减顺序程序时不希望的执行[18]。

我们所考虑的编程语言很简单，但是非常具有表现力，因为表达式的语法是自由的，并且我们不会因为只考虑单个全局变量和局部变量而失去一般性。本文的补充扩展版本[9]列出了几个语法扩展，这些扩展很容易简化为我们的语法语法，我们将在后续章节的源到源翻译中使用它们。此外，可以使用全局共享内存对硬件中断和基于锁的同步进行编码，参见本文的补充扩展版[9]和3.3节。

3.2 编程语法

由于我们考虑的是协作多任务模型，其中缓冲区之间和同一缓冲区的任务之间的控制传输是显式的，因此我们的操作模型在任何时刻只需要考虑单个缓冲区中当前正在执行的单个任务。当同一缓冲区的高级任务中断低级任务时，我们的模型只是将高级任务的帧添加到当前过程堆栈的顶部。因此，我们用当前正在执行的活动任务a(n) 2个任务来表示每个任务缓冲区n2n，以及一个由待处理任务组成的多集b(n) 2m [Tasks]，我们定义了配置c = hg;n;一个;全局变量g的值，加上当前活动的任务缓冲区n 2n，一个活动的任务a: n !每个缓冲区的任务，以及多组待挂任务b: N !M[任务]每个缓冲区。图2总结了各种语义域和元变量命名约定。

在本文的补充扩展版本[9]中，将异步程序P的迁移关系!P表示为组态之间的操作步骤c1 !P c2的集合。顺序语句的转换或多或少是标准的。post post语句的转换创建了一个新的帧f来执行级别m的给定过程p，并将f放在当前活动任务缓冲区的待挂任务中。当(单帧)任务f完成其执行时，f从过程堆栈中被丢弃，以继续执行下面的任务。构象hg的产率转变;n;一个;Bi允许缓冲区n中的活动任务t1(即a(n) = t1t3)通过从b(n)中删除t2，将a(n)更新为t2t3并将t1放入b(n)中来放弃对另一个相同优先级的挂起任务t2 2b (n)的控制。从一种构型hg到zield的转变;n1;一个;bi只是将当前活动的任务缓冲区从n1更新到任意的n2 2 n。最后，当当前执行的任务具有较低级别时，任务调度规则调度最高级别的任务;只有当当前执行的任务是最高级别的任务(每个缓冲区)时，才启用所有其他转换。

3.3 同步建模

由于跨多个任务缓冲区执行任务可能会交错，通常情况下，程序需要一种确保数据访问原子性的方法。在我们的模型中实现同步指令并不困难，因为我们假设任务只在指定的yield点被其他缓冲区的任务抢占，或者在指定的yield点被相同缓冲区的任务抢占。例如，锁可以通过添加一个额外的全局变量var lock来实现:获取锁的方法如下:

释放锁就像设置lock:= false一样简单。

注意，while循环的退出和赋值锁:= true保证是自动发生的，因为没有干扰的yield或zield语句。一旦锁被一个任务持有，任何其他获取锁的任务都必须等待，直到锁被释放。

1. 单缓冲区规约

为了将异步程序的状态可达性问题简化为顺序程序的状态可达性问题，我们首先将多缓冲区程序转换为单缓冲区程序，同时保留任务优先级。由于具有任务优先级的单缓冲区有限数据异步程序的状态可达性问题对于非抢占程序来说是可确定的[3]，而我们的多缓冲区变化则不是，因此我们的翻译必然代表了原始问题的近似。虽然我们的翻译编码了各种缓冲区间交错，但它不能编码每个缓冲区间交错。为了控制和细化所考虑的交错的数量，从而达到近似的程度，我们的转换采用一个边界参数K。按照原始多线程序列化的方法[18]，对于给定的边界参数K，我们只按缓冲区索引顺序探索K轮循环执行;也就是说，在每一轮中，来自缓冲区0的任务执行到一条(可能不是第一条)zield语句，此时来自缓冲区1的任务执行到一些zield语句，以此类推。在zield语句中，来自缓冲区N−1的任务放弃控制，第二轮开始，恢复缓冲区0的挂起任务。注意，边界允许在缓冲区内任意长时间执行。

例1(限制缓冲区间交错)。下面这个具有2个任务缓冲区和一个优先级的异步程序演示了k轮探索如何限制程序行为。