处理 Java 程序不确定性问题的技术研究和综述

季晓枫 宋昶衡 李 弋
(复旦大学软件学院 上海 201203)
(上海市数据科学重点实验室(复旦大学) 上海 201203)

摘 要 随着多核处理器的发展,大规模并行程序已经成为了主流。然而与单线程程序不同,并行程序并不能保证程序每次执行的路径都相同。路径不确定性却给程序的性能评估和错误调试带来了许多问题和挑战。而 Java 虚拟机本身的即时编译,垃圾收集等机制,更是加剧了程序的不确定性。如何解决不确定性的影响,一直是性能评估和错误调试两个领域的重要课题。从这两个方面,分别对处理 Java 并行程序不确定性的方法进行分析和总结。在此基础上,进一步比较了各项技术的优劣,也对性能评估和确定性重放两项技术的可能发展方向进行了展望。

关键词 Java 不确定性 并行程序

中图分类号 TP3 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2018.08.002

RESEARCH AND REVIEW ON DEALING WITH VARIABILITY IN JAVA PROGRAMS

Ji Xiaofeng Song Changheng Li Yi
(*School of Software, Fudan University, Shanghai 201203, China*)
(*Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China*)

Abstract With the development of multi-core processor, large scale parallel programs have become prevalent. However, unlike serial programs, parallel programs cannot guarantee that the execution paths are the same during different runs. The uncertainty of the execution paths has brought many problems and challenges to the evaluation and debugging. Meanwhile, mechanisms inside the Java virtual machine such as just-in-time compiler and garbage collection also amplify the variability of the program. Therefore, how to deal with the variability during execution has become an important issue in the fields of both performance evaluation and debugging. From these two aspects, this paper analyzed and summarized the methods to deal with the variability of Java parallel programs. Further, this paper presented the prospects of the possible development direction of Java performance evaluation and deterministic replay.

Keywords Java Variability Parallel program

0 引 言

随着硬件的发展,小到个人计算机大到数百上千核的超级计算机,多核处理器已经十分普遍。为了能够充分利用多核处理器性能,并行程序变得越来越流行,并开始占据了主流的地位。

然而,与单线程程序不同,并行程序并不能保证每一次的执行与之前的执行从结果和性能上都完全相同。在多核处理器上执行并行程序的过程中,抢占式的线程切换使得程序在不同的运行过程中,线程的调度过程都是不确定的行为。而在多核处理器上,对共享变量的访问更是加剧了程序的不确定性。在不同的执行循环过程中,共享变量访问的读写依赖关系也是无法确定的。由于每次执行时共享变量的放回值都可能变化,这导致程序的运行路径在多次的执行之间也可能是不同的。

而在 Java 虚拟机上,这一不确定性变得更为显

收稿日期:2018-03-04。国家重点研发计划项目(2016YFB0200501);上海市科委科研计划项目(17511102200)。季晓枫,硕士生,主研领域:计算机系统软件。宋昶衡,硕士生。李弋,讲师。

著。Java 语言在过去的十数年中一直是最受欢迎的语言之一。Java 受欢迎的一个非常重要的原因就是 Java 可运行在虚拟机上,保证了 Java 程序的可移植性,独立性以及安全性。然而,在 Java 虚拟机中的使用的许多机制都会给 Java 程序的执行带来不确定性,这其中包括了即时编译器、垃圾收集等。即时编译器通过对程序的采样分析决定对哪些方法进行优化,垃圾收集机制的触发也是在运行时决定的,这些机制在 Java 虚拟机中都由额外的线程来完成。这使得即使是单线程的 Java 程序,其不确定性也会比 C 或者 C ++ 程序来的更为显著。图 1 展示了 Java 的多线程程序测试集 DaCapo 的性能结果不确定性。图中黑色方块代表了变异系数的区间。其中,变异系数最大的 sunflow 达到了 12.12%,平均变异系数达到了 6.87%。由此可见,不确定性在多线程 Java 程序中表现得非常明显。

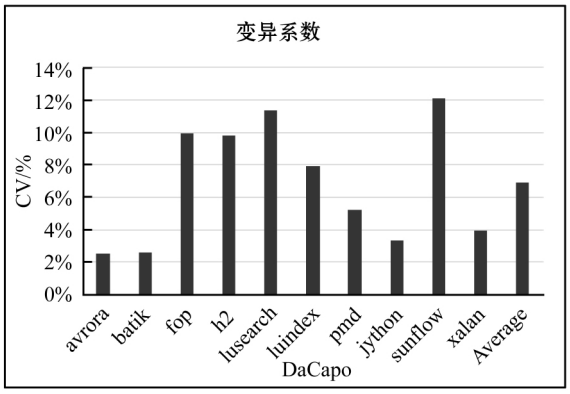

图 1 DaCapo 测试集的不确定性

程序的不确定性会给许多原本看似简单工作带来新的问题。并行程序的不确定性主要是由于其本身的路径不确定性所产生的,而路径的不确定性会给许多相关工作带来问题。首先,这会很大程度上影响性能评估的工作,路径的不确定性会导致性能的不稳定。当研究人员要评估性能结果的优劣时,受到程序不确定性的影响,简单地取均值的方法就不再适用,简单的方法难以总结和描述程序的性能表现。其次,路径的不确定性会给错误调试带来很大的挑战。在进行循环调试的时候,如果不能每次都重现错误发生的过程,就会使得调试变得非常困难。

本文首先分析了 Java 虚拟机内部会给程序带来不确定性的原因,并对于不确定性在性能评估和错误调试方面造成的问题进行了分析。在此基础上,本文从处理带有不确定性的性能数据以及控制程序不确定性的技术两个方面对处理 Java 虚拟机性能不确定性的方法进行了总结。此外,在不确定性程序的错误调试方面,本文对解决这一问题的常见确定性重放技术进行了分析和总结。最后,本文对适用于这两个问题的常见的框架的技术思路进行了分析和比较,对于处理 Java 虚拟机不确定性的技术发展的未来可能的发展方向进行了展望。

1 Java 程序不确定性造成的问题

Java 程序的不确定性问题所带来的影响主要包含两个方面:(1) 性能结果的不确定性对于性能分析的影响;(2) 路径不确定性对于对错误调试的影响。本节首先将对产生不确定性的因素进行总结,然后将分别对这两个方面影响进行分析。

1.1 多线程程序的不确定性

Java 多线程程序本身的多线程执行就会给程序带来不确定性。首先,线程的调度不能保证在每一次的执行中,每个线程的调度顺序都完全一致。其次,在有些多线程程序中,工作并不会均匀地分配到每一个线程中去,比如,在多个线程竞争访问任务队列的时候,每次程序的执行,各个线程所完成的工作都是随机的。最后,线程迁移和访问共享资源的竞争都会增加多线程程序在执行过程中的不确定性。

1.2 Java 程序的不确定性

除了多线程执行本身的不确定性之外,在 Java 虚拟机中,也有许多部分会对于程序性能产生影响,大致包括以下几个部分:即时编译器、线程调度、垃圾收集机制以及其他的一些系统因素。

(1) 即时编译器 Java 程序在运行前被保存为字节码的形式,Java 虚拟机可以解释执行字节码状态的程序,但是这样做的效率非常低。于是 Java 引入了即时编译器。即时编译器通常会对程序进行采样分析,寻找程序中的热点,当程序中的某个方法成为了热点之后,即时编译器会对该方法进行优化编译。由于并行程序本身的不确定性,在多次执行中,触发即时编译器的方法的顺序并不会是固定的,这就会导致程序在不同执行循环中的性能变化。此外,即时编译器本身作为 Java 虚拟机中的一个或多个线程进行工作,同样会增加程序的不确定性。

(2) 线程调度 和其他并行程序一样,Java 并行程序的线程调度也是程序不确定性的来源之一。Java 虚拟机会直接对线程进行调度,或者对线程进行封装之后交由系统来进行调度。在不同的执行循环中,程序可能经过了完全不同的调度,导致线程之间不同的交互,最终对性能产生影响。

(3) 垃圾收集 垃圾收集机制在 Java 虚拟机中同样由额外的线程实现,大部分垃圾收集器在执行过

程中还会将 Java 程序暂停。此外,垃圾收集的过程也会对程序的不确定性造成影响,这取决于使用的垃圾收集机制是确定性的或者是非确定性的。

此外,其他的一些系统因素,比如系统中断,外部 IO 也会增加程序的不确定性。

1.3　不确定性对于性能评估的影响

传统的性能评估方法大致包含了取平均、取中位数、取最好值、次好值以及最差值等几种[1]。当研究需要获得性能测试的平均结果时,取平均数和中位数是比较好的选择。取最佳值和最差值是两种比较特殊的方法。取最佳值法通常用于寻找类加载和即时编译器等虚拟机系统行为对 Java 程序影响最大的情况下的性能,而与之相对应的,取最差值法是用于寻找类加载和即时编译器对程序运行影响最小的情况下的性能。

以上这些性能评估方法给出的结果都是单一的数值表示。但是,由于 Java 程序的不确定性,多次运行的 Java 程序的性能结果会有不同幅度的波动。仅仅用一个数字来代表程序的性能没有办法完整的描述 Java 程序的性能。比如 Java 程序性能的波动范围以及在相应范围上的分布都是一个数字所无法描述的。这时,就可以使用置信区间来进行描述。置信区间可以表示出 Java 程序性能的波动范围,给出程序性能落在某个区间上的概率。假如程序结果的分布是正态分布的话,置信区间配合变异系数可以很好地描述 Java 程序的不确定性。然而如果 Java 程序不是正态分布的话就需要配合其他的方法来对程序结果进行描述。

对于程序的性能评估来说,如何在更少的测试时间内得到更精确的结果是这个问题的主要研究方向。

此外,对于 Java 程序的性能评估来说,不同的阶段会有不同的不确定性影响因素。如果在一次启动 Java 虚拟机的过程中多次循环运行一个程序,那么前几次循环的性能会受到动态类装载和即时编译器非常大的影响。而在之后的循环中,程序性能的波动幅度,也就是变异系数会逐渐稳定,在评估 Java 程序的性能是要将这两种状态下的性能区分开来评估,分为启动状态和稳定状态[2-3]。

1.4　路径不确定性对错误调试的影响

由于并行程序在执行过程中的存在路径不确定性,这对错误调试带来很大的麻烦。路径不确定性对于错误的影响主要在于线程切换和同步,以及对共享变量访问的依赖关系。在不同的执行循环中,抢占式的线程切换以及线程同步都会导致不同的线程调度结果。这两个因素,同共享变量的不同访问顺序一起,造成了程序执行路径的不同。而在并行错误调试的过程中,对于内存访问顺序的重现在解决死锁、数据竞争等问题时又是至关重要的。如果对一个共享变量的访问顺序发生了变化的话,很可能程序就不会再进入死锁状态。

为了解决这些问题,并帮助开发者能够顺利的进行循环调试,现今最为主要的技术就是程序的确定性重放。

确定性重放技术分为两个阶段:(1) 记录阶段,在记录阶段,确定性重放的工具会记录程序一次运行中的信息,并存放到文件中;(2) 重放阶段,工具会读取记录到文件中的信息,并按照记录的运行轨迹重现之前程序的运行路径。

一般而言,记录阶段需要记录程序中所有的不确定因素,这样才能确保在重放阶段时的执行路径和记录的那次运行一模一样。然而,随着多线程程序的规模变得越来越庞大,其中用到的共享变量访问越来越多,完全记录所有的不确定性信息变得越来越不现实。所以,现今对于确定性重放的研究主要目标就是降低记录阶段的开销,减小记录阶段用到的日志文件大小。

此外,对于 Java 程序而言,错误也可能发生在虚拟机内部,比如即时编译器或者垃圾收集阶段,所以也有研究针对 Java 程序的机制做了确定性重放的工具,比如 Ogata[21] 等的工作。

2　处理性能不确定性

在对 Java 程序进行评估的过程中,Java 程序的不确定性会带来很多问题。

首先,在 JVM 的一次生命周期中运行多次 Java 程序循环时,即时编译器、类装载等因素会使得初始的多次运行的行为表现与之后的运行截然不同。如何对这两种状态进行划分并且分别进行性能评估是进行 Java 程序评估时非常重要的一点。

其次,由于 Java 多线程程序的不确定性比一般的多线程程序更为复杂,如何使用更少的运行遍数进行性能评估也是亟待解决的课题。

2.1　利用统计学处理性能不确定性

由于即时编译器以及类加载机制的存在,在利用统计学处理 Java 程序的不确定性时首先需要划分出启动状态下和稳定状态下的 Java 程序结果。其后,可以对这两种状态下的结果使用统计学的方法进行相应的处理。

Java 程序的不确定性会使得程序的结果分布在一个区间上。当这些结果在区间上的分布呈现正态的时

候,可以简单地使用置信区间来描述程序的性能,并使用参数检验来比较两组数据的性能。但是,当程序的性能结果分布呈现非正态的时候就需要使用非参数检验来判断两组数据的性能优劣。

(1) 对正态分布数据的处理 正态分布,又称高斯分布,以平均数为中心左右对称,向两边逐渐均匀下降。正态分布是最常见的分布之一,而且使用很简单的参数就能够对该分布进行描述。此外,最重要的是,根据中心极限定理,即使是非正态分布,大量独立同分布的随机变量的平均数是正态分布的。这给了研究者使用正态分布来对多线程程序性能进行描述的机会。

Georges 等[1]综合了一系列的统计学方法,提出了一种严格的统计学测试框架,用来评估 Java 程序性能。这种框架针对启动状态和稳定状态分别做了处理。

首先是启动状态,启动状态的性能会受到类加载,即时编译器等因素的影响。针对启动状态,Georges 等提出的方法如下:测量启动状态的 Java 程序性能需要多次启动虚拟机,每次只运行程序一遍,记录其运行时间。然后根据指定的置信度计算其置信区间。对于启动状态性能的比较,可以使用 z 检验或者 t 检验来实现。

其次,是稳定状态的性能评估。稳定状态时由于程序已经运行过一段时间,受到即时编译器的影响比较少,也几乎不会收到类加载的影响。为了测量稳定状态的性能,首先要确定程序的运行在什么时候进入了稳定状态。Georges 等[1]使用在一次虚拟机的生命周期中多次运行一个比较短的 Java 程序的方法来模拟长期运行的 Java 程序。他们提出的方法如下:假设整个测试同样要运行 p 次虚拟机周期。在每个虚拟机的生命周期中,循环运行 Java 程序 q 次。在程序循环运行了 k 次之后,计算最后 k 次运行的变异系数,当变异系数小于预设的阈值时,则认为该 k 次运行都达到了稳定状态,记录其结果。计算每次虚拟机生命周期中的 k 次稳定状态运行结果的平均值,那么这 q 个平均值就构造了一个新的性能结果分布。计算这个新的分布在指定置信水平上的置信区间,用该置信区间来评价程序在稳定状态中的性能。

在评估稳定状态性能的过程中,之所以需要记录多次虚拟机运行中的数据,而不是在虚拟机的一次运行中就多次循环把所有数据都跑完,是为了保证最终结果的正态分布性。在虚拟机每一次的运行中,多次循环后的 k 个结果之间并不是独立的,但是多次虚拟机的运行之间是互相独立的,所以基于中心极限定理,取每次虚拟机运行的 k 个结果的平均值就能够得到一个独立的结果分布。

随后 Kalibera 等[4]为了提高性能评估的效率,针对性能评估的各个阶段提出了一系列的建议,用以减少性能评估中的重复次数,并在合理的时间内完成性能评估。他们提出了一种新的识别稳定状态的方法。这种方法通过识别程序的运行是否进入“独立”状态来判断循环是否已经稳定。“独立”状态意味着程序这次的运行与之前的循环无关。该方法利用了 Lag 图、ACF 图和运行序列图,帮助研究人员根据这三种图来进行人工判断。通过这种方法,可以快速地判断一个程序是否进入了“独立”的状态。Kalibera 等[4]认为,对于没有在较短的时间内进入“独立”状态的程序,建议选取多次虚拟机生命周期中初始化状态之后的第一次循环作为性能评估的对象。而对于每次虚拟机运行中的初始化状态,同样用人工识别的方法,对运行序列图进行观察并选取,这个工作也能在数秒内完成。

除此之外,Kalibera 等[4]还评估了 Java 程序在虚拟机运行内部循环时和多次虚拟机运行间的性能波动幅度,并且发现有一些程序在虚拟机运行间的波动性远大于内部循环时的波动性。由此提出,对于类似的程序可以减少内部循环的重复次数,增加虚拟机的运行次数,而对于特性相反的程序就可以减少虚拟机运行的次数,增加内部循环数,减少由于虚拟机重新启动所需的热身时间。

与文献[1]之前的方法进行比较,之前的方法对于达不到“独立”状态的程序并不适用,而文献[4]通过对于性能评估中的多个重复周期加以区分,在不确定性较小的层次中减少重复次数,并且引入了人工的阶段识别,提升的性能评估的整体效率。

(2) 对非正态分布数据的处理 多线程程序的运行时间分布大多不符合正态分布。为了能够使用置信区间来表征程序的性能,前文提到的方法利用了中心极限定理使得收集到的数据能够呈现正态分布,但是中心极限定理要求上百个运行结果,才能足够构成正态的平均值分布[5]。

Chen 等[5]提出了一种利用非参数统计方法的框架,用于利用更少的程序运行遍数来获取准确的性能比较数据。这种方法利用了 Wilcoxon 秩和检验,使用 Wilcoxon 秩和检验可以在给定的置信水平上判断两组运行数据的性能差距是否显著。在判定加速比的时候,将一边的数据乘以预计的加速比,再使用非参数检验的方法就能判断这个加速比是否成立,最后通过缓慢调整加速比得到最好的结论。这种方法既可以用于评估单个程序在两种架构上的性能,也能应用于用整

个测试程序集评估架构的性能。

这种方法的好处是在实验数据有限的情况下也能够得到一个可能比较保守但是正确的结论,与简单的计算平均值相比置信水平和加速比的准确性都有大幅度的提高。

2.2　控制不确定性变量

为了控制 Java 程序的不确定性在性能评估时的影响,除了利用统计学方法对性能测试结果进行处理之外,还有另一种思路,即对程序的运行过程中的变量进行控制。

Charlie 等提出了 Stabilizer[6] 。程序在运行时,堆栈和代码的分布等因素都会受到架构的影响,这些不确定因素使得程序的性能结果未必会呈现正态分布。

Stabiilizer 对于多个层面的不确定因素都进行了包装,并且主动进行与结构层面的无关的随机化。在堆中,Stabilizer 会根据随机函数,给新的动态内存请求分配指定的位置。代码块同样会被随机地存储在堆上。在每个函数内部 Stabilizer 都会插入一个 trap 指令,在运行到这个指令的时候就会根据随机函数将代码块在堆上进行分配。对于栈的随机化,Stabilizer 在每个栈帧之间都会插入一个随机大小的空白块,最大不超过 4 096 比特,用来保证每个函数的栈都是随机的。

此外,程序每运行过一段时间,都会对所有层面重新进行一次随机化。这样,根据中心极限定理就能确保这些因素对于程序的影响是符合正态分布的。在确保了程序结果的正态性之后,就可以用方差分析等方法对程序的性能结果进行分析。

Chen 等[7] 利用符号执行建立了一个 Java 程序性能分布的概率模型,并将这个模型上的性能结果分布进行了可视化。

在符号执行的过程中,根据输入集中的权值以及相应的输入,计算程序执行某一分支路径的概率。之后通过对这些输入进行测试执行,得到的加权平均值就是该路径的性能。在完成符号执行的搜索之后,就能获得该程序的性能分布模型。

这一方法的能够获得比较完整的程序性能分布,但是所受的限制也比较多。首先,暂时还不支持不确定性多线程程序,Luckow 等[8] 正在尝试实现多线程的版本。其次,该方法的性能模型仍然不够完备,没能把太多的体系结构因素考虑进去。

3　处理路径不确定性

正如前文所提到的,Java 程序的不确定性多除了对性能的评估造成了影响之外,对于错误调试也会造成很大的干扰。

程序的错误调试经常需要多次循环执行程序并重现错误来解决问题。但是多线程程序的不确定性使得程序即使多次循环也未必能够重现错误执行的路径并将错误重现。在 JVM 虚拟机上执行时,不确定性的问题也显得更为严重。本节将针对 Java 程序的确定性重放技术进行总结和综述。

早期 Java 程序的重现方法着重于确保相同的线程切换顺序以及相同的程序状态。比如 DejaVu[9] 、JaRec[10] 和 ReVirt[11] 等,这些框架适合用于单核机器上的确定性重现。

DejaVu 会记录并重现每次线程切换,包括同步操作和抢占式的切换。DejaVu 确保在重放时能够切换到正确的线程,并且在切换后程序的状态与记录阶段的状态相同。此外,为了处理插桩对于程序性能的影响,DejaVu 还会记录框架本身所有对于 Java 程序有影响的操作,并在重放过程中复现这些操作,确保重放与记录的一致性。

JaRec 同样只记录 Java 程序中的同步操作,它利用了 Lamport 逻辑时钟来记录每一次线程同步的逻辑顺序,在重放的时候只需要读取记录文件中的逻辑时钟就能实现同样的线程切换和同步的顺序。JaRec 使用 JVMPI 实现插桩。重放的程序所需时间是与原程序的 1.5 倍到 39 倍不等,这一比率取决于同步操作的数目。与 DejaVu 相比,JaRec 可以在多核平台上使用。

此外,Ronnse[12] 等和 Choi 等[13] 也实现了利用逻辑时钟的重放框架。随着多线程程序变得越来越庞大,仅仅记录线程的切换行为并不足以完成并行程序的重放,性能也越来越差。如何将对共享内存的访问记录下来并重现成为了确定性重放技术的新难题。为了解决这一问题,研究人员提出了许多技术来对记录阶段进行优化,工作主要集中于如何减少记录的事件数,这样既能降低记录阶段的开销,也能减少记录文件的大小。

3.1　对记录阶段的优化

传统的确定性重放技术会记录程序所有不确定性操作,并记录下他们的全局顺序以便之后的重放。但这会导致在记录阶段需要许多同步操作来确定事件的顺序。

Leap[14] 对这一现象进行了优化。Leap 记录每个共享变量的线程访问顺序,而不是记录共享内存的全局访问顺序。通过这个方法,降低在记录阶段的开销。Leap 首先通过 Java 程序的静态分析框架 Soot 找出在

运行中会产生多个线程访问的共享变量，然后在运行时记录每一个共享变量的线程访问序列。在重放时，重放引擎会强制程序按照记录的线程访问队列执行。Leap 通过 Soot 实现，经过测试，其记录阶段的性能比传统的逻辑序列，全局序列要快 5 ~ 10 倍。

其后的 Order[15]、CARE[16]、Ditto[17] 都对记录阶段的局部性做了优化，提升记录阶段的性能。

Order 框架同样记录了共享变量的线程访问序列。Order 框架的记录以 Java 中的对象为单位，将访问记录放在对象的头部，这样能减少垃圾收集时产生的依赖关系对记录的影响。此外，将对共享内存的访问信息存放在对象的头部，能够提高记录阶段的局部性。最后，Order 还会记录一些 Java 独有的导致不确定性的因素，其中包括了垃圾收集，动态编译以及类的初始化等操作的时间点，从而能够重现 Java 虚拟机内部产生的错误。相对于正常运行的程序，基于 Apache Harmony 实现的 Order 记录阶段的平均额外开销为 8%，比 Leap 快 1.4 ~ 3.2 倍。

CARE 提出了一种减小记录阶段所需的日志文件大小的技术。CARE 为每个线程都添加了一个的共享变量的软件缓存，只有当读操作返回的值与缓存中的值不同时，CARE 才会记录下这次读操作的读写依赖关系，通过这种方法能减少需要记录的条目。CARE 基于 JVMTI 进行插桩。相比于 LEAP、CARE 的记录开销是 LEAP 的 38.47%，日志文件是 LEAP 的 20.41%。

此外，Ditto 也是一种记录对共享变量的线程访问顺序的框架，Ditto 利用逻辑时钟来保存不同线程上内存访问的顺序，同时利用偏序关系的传递性减少需要记录时的日志文件大小。Ditto 也同样利用了 Soot 静态分析框架来区分局部变量和全局变量，对记录过程进行优化。

除了记录足够的读写顺序之外，还有一些方法减少了记录的信息，并在重放阶段对程序的执行路径进行推断，如 ODR[18]、CLAP[19] 和 PRES[20] 等。这些方法通常只能保证重放阶段的部分精度。

Stride[21] 将两种方法进行了融合，这一框架在记录所有读写关系顺序的方法和对程序路径进行搜索的方法之间进行了权衡，使用 Java Soot 实现。Stride 不会记录所有确切的读写关系。对于每次读操作，它会记录对于这个变量最近版本的写操作，这个写操作在之后确定具体读写联系的时候就是搜索时的上界。相比传统的记录顺序的方法，Stride 在记录时能够减少许多同步操作。与 Leap 相比，Stride 在记录阶段快 2.5 倍，记录文件小，是 LEAP 的 25.77%。

3.2 在特殊虚拟机环境下进行确定性重放

为了一些特殊的目的，比如安全性的要求，或者为了降低确定性重放的难度，一些确定性重放技术通过定制的虚拟机环境进行实现，比如 ReVirt[11]、SMP-ReVirt[22] 和 TDR[23]。

ReVirt 框架在一个自己实现的虚拟机上进行记录和重放，这样能够移除程序对于外部操作系统的依赖。ReVirt 的重放机制与前文所述的框架类似，只适用于单核上的多线程程序。

SMP-ReVirt 基于 Xen 实现了多核模拟器上的确定性重放。SMP-ReVirt 利用了并发读、单独写(CREW)的协议来控制虚拟机对于共享变量的访问，然后通过硬件页保护来确定读写的顺序。

TDR 为了能够在进行确定性重放之外，还能确保程序的执行时间与原来的运行相同，在虚拟机上实现了确定性重放功能。为了使得记录和重放的环境不受干扰，TDR 重新实现了一个 Java 虚拟机，在这个虚拟机上尽可能地排除了系统因素对程序的影响，达到干净状态。除了虚拟机之外，TDR 还需要另一个核心来帮助虚拟机执行 IO 和中断等操作。由于这个虚拟机暂时只支持单核执行，TDR 所使用的重放方法是传统的记录线程切换的方法。在同样配置的两台机器上进行记录和重放，TDR 能够让运行时间的误差保持在 1.85% 之内。TDR 的虚拟机并没有垃圾收集和动态编译的功能。

3.3 针对 Java 具体机制的确定性重放

Java 虚拟机中，即时编译器在每次运行时的行为都是不确定的。为此，Ogata 等[24] 实现了动态编译的重放。这一框架由两个即时编译器组成，记录阶段用的编译器会在每次进行动态编译的时候将编译器收到的所有输入，包括系统配置、虚拟机状态和分析数据存储到日志文件中。这些记录信息会在系统转储时写入其中，重放编译器工作时会将系统转储文件载入地址空间，然后读取日志文件作为编译器的输入，进行动态编译的重放。这一框架在记录阶段的开销只有 1%，日志文件非常小。

3.4 根据路径特征信息进行不确定性分析

VarCatcher[25] 通过记录程序的路径特征，对齐并进行处理之后得到并行特征向量(PCV)。通过对多次运行的并行特征向量进行聚类处理，可以得到路径类似的运行结果。得到运行路径相似的运行结果之后可以实现和确定性重放类似的效果。此外，通过对于并行特征向量的处理，可以对程序在不同路径下的性能

结果进行更深入的分析。利用 Intel Processor Trace 机制,该框架在记录阶段只有 3% 的额外开销。

4　分析与展望

本节我们将对上文提到的各类不确定性处理机制进行比较和分析,并对未来可能的发展方向进行了展望。

4.1　分析总结

(1) 对处理性能不确定性研究的分析　针对性能不确定性的研究主要分为两大类,一类从性能测试和统计学的角度出发,对测试方法和性能分析的方法进行优化;另一类,从程序本身出发,控制运行过程中的不确定性变量,使得对性能不确定的分析更加简便。

文献 [1] 提出的严格的统计学处理方法,也就是表 1 中的 Rigorous,为 Java 并行程序的性能测试提出了一种完备的框架。文献 [4] 的工作,也就是表中的 Reasonable,对前者的工作进行了优化,将测试的过程分层,在不必要的层上减少重复次数。随后文献 [5] 的工作,也就是表中的 HPT,使用了一种非参数检验的方法,能够提升从已知的数据从归纳出更精确的结果。

表 1　处理性能不确定性研究的比较

框架	测试开销	方法
Rigorous	数百遍	数据处理
Reasonable	大部分小于一百遍	数据处理
HPT	数十遍	数据处理
Stabilizer	每次运行约 7%	人工随机化
Distribution	符号执行的时间	符号执行

Stabilizer 和 Distribution,也就是文献 [7] 的工作,则将减少性能不确定性的努力放在了程序本身上。Stabilizer 能够使程序运行的结果呈正态分布;而 Distribution 则希望通过符号执行建立程序的性能分布模型。值得注意的是,这项技术还并不完全成熟,并不足以模拟多线程不确定性和系统对程序的影响。

(2) 对 Java 程序的确定性重放技术的分析　确定性重放技术的研究方向大致分为两种:(1) 致力于对记录阶段进行优化,降低重放技术的额外开销;(2) 通过特定的虚拟机环境,实现额外功能的重放技术。对于前者,表 2 给出了本文总结的几种确定性重放技术的性能比较。由于这几篇文献的性能分析都是以 Leap 为比较对象,表格中就以 Leap 为基准进行比较,其中 Ditto 给出了比较数据,但没有具体的总体性能比较。可以看到,后面几篇文献进行各自的优化之后,都取得了比较好的结果。Order 针对 Java 虚拟机中的编译器,垃圾收集等做了额外的信息记录。Ditto 在记录阶段利用偏序的传递性,减少不必要的信息记录。CARE 通过自己实现的软件缓存减少需要记录的读写操作。Stride 结合了路径推断的技术,减少记录阶段的同步操作。

表 2　各确定性重放相对于 Leap 框架的性能比较

确定性重放技术	相对性能	日志大小优化
Leap	1	1
Order	1.4x – 3.2x	小于 100 m/h
Ditto	超过 Leap	超过 Leap
CARE	2.6	4.9
Stride	2.5	3.88

4.2　展　望

4.2.1　对处理系统不确定性研究的展望

在对数据处理的研究上,前文所述的研究已经做出了十分显著的成果,但依然有一些问题得不到解决。比如在对于 Java 虚拟机内部机制和并行程序不确定性之间的关系,这方面的研究仍然比较缺乏。如果能够知道即时编译器和垃圾收集等因素会在多大的程度上,会如何影响 Java 程序执行时的不确定性,对分析 Java 程序的性能结果会很有帮助。并行程序结果的分布形态与程序本身的关系也是可以进一步研究的方向,这也有助于将程序的结果分布正态化。

此外,通过符号执行对程序性能分布建立模型的工作也需要进一步发展,当这项工作能够将所有的因素都考虑到模型中时,该技术才能得到更广泛的应用。

4.2.2　对 Java 程序的确定性重放技术的展望

尽管前文提到的工作对确定性重放技术有了非常大的提升,但是确定性重放技术的代价仍然是比较大的,这项技术还有进一步提升的空间。

目前的大多数工作致力于减少对共享变量访问的记录。如果要在这一方面有所突破,可以进一步对多线程程序访问共享变量的模式进行优化,减少不确定性的共享变量访问。另外,更多的使用事务内存,也可以简化对于共享变量的记录,只在检查点上对这些访问记录进行处理。

另一方面,结合前文提到的 TDR 技术也是一个非常有潜力的方向。随着大数据处理和云计算变得越来越普及,在虚拟机上运行程序将是未来不可避免的趋势。如果 TDR 的虚拟机能够实现更高效的性能,那么将有潜力解决未来在云端的程序确定性重放问题。

此外,TDR 的时间确定性重放对于解决并行程序

的性能不确定性也是非常有帮助的。如果在两台配置相同的机器上能实现一模一样的性能,那在两台配置不同的机器上运行,两者之间的速度差异就可能是确切的性能差异。若能实现这个效果,将会极大地性能评估的效率,并解决性能不确定性的问题。

5 结 语

随着多核处理器的发展,大规模并行程序已经成为了主流。然而并行程序的不确定性却给程序的性能评估和错误调试带来了许多问题。而 Java 虚拟机由于自身系统存在的不确定性,加剧了程序的不确定性,这使得 Java 并行程序的性能评估以及确定性重放的错误调试方法变得更加重要。本文针对解决 Java 并行程序的性能评估方法以及确定性重放技术分别进行了总结和分析。Java 并行程序的性能评估的难点主要在于提升性能测试的效率以及对程序的性能进行总结和描述。基于 Java 程序的确定性重放技术的主要难点在于如何提升记录阶段的性能。最后,本文总结了这两个方向如今还存在的问题,并对解决不确定性问题的前景进行了展望。

参 考 文 献

[1] Georges A, Buytaert D, Eeckhout L. Statistically rigorous java performance evaluation [C] //ACM SIGPLAN Conference on Object-oriented Programming Systems & Applications. ACM, 2007:57-76.

[2] Arnold M. Adaptive Optimizations in the Jalapeno JVM [C] //Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications. 2000: 65-83.

[3] Arnold M, Hind M, Ryder B G. Online feedback-directed optimization of Java [J]. ACM SIGPLAN Notices, 2002, 37 (11):111-129.

[4] Kalibera T, Jones R. Rigorous benchmarking in reasonable time [C] //International Symposium on Memory Management. ACM, 2013:63-74.

[5] Chen T, Guo Q, Temam O, et al. Statistical Performance Comparisons of Computers [J]. IEEE Transactions on Computers, 2012, 64(5):1442-1455.

[6] Curtsinger C, Berger E D. Stabilizer: Statistically sound performance evaluation [C] //Eighteenth International Conference on Architectural Support for Programming Languages & Operating Systems. ACM, 2013:219-228.

[7] Chen B, Liu Y, Le W. Generating performance distributions via probabilistic symbolic execution [C] //IEEE/ACM, International Conference on Software Engineering. IEEE, 2017:49-60.

[8] Luckow K, Dwyer M B, Filieri A, et al. Exact and approximate probabilistic symbolic execution for nondeterministic programs [C] //ACM/IEEE International Conference on Automated Software Engineering. ACM, 2014:575-586.

[9] Alpern B, Choi J D, Ngo T, et al. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications [C] //International Parallel & Distributed Processing Symposium. IEEE Computer Society, 2001:23.

[10] Georges A, Christiaens M, Ronsse M, et al. JaRec: a portable record/replay environment for multi-threaded Java applications [J]. Software Practice & Experience, 2004, 34(6): 523-547.

[11] Dunlap G W, King S T, Cinar S, et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay [J]. ACM SIGOPS Operating Systems Review, 2002, 36 (S1):211-224.

[12] Ronsse M, Bosschere K D. RecPlay: a fully integrated practical record/replay system [J]. ACM Transactions on Computer Systems, 1999, 17 (2):133-152.

[13] Choi J D, Srinivasan H. Deterministic Replay of Java Multithreaded Applications [C] //Proceedings of the SIGMETRICS Symposium on Parallel & Distributed Tools. 1998: 48-59.

[14] Huang J, Liu P, Zhang C. LEAP: lightweight deterministic multi-processor replay of concurrent java programs [C] // Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2010:385-386.

[15] Yang Z, Yang M, Xu L, et al. ORDER: object centric deterministic replay for Java [C] //USENIX Conference on USENIX Technical Conference. 2012:30-30.

[16] Jiang Y, Gu T, Xu C, et al. CARE: cache guided deterministic replay for concurrent Java programs [C] //Proceedings of the 36th International Conference on Software Engineering. ACM, 2014:457-467.

[17] Silva J M, Simão J, Veiga L. Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors [C] //ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, Berlin, Heidelberg, 2013:405-424.

[18] Altekar G, Stoica I. ODR: output-deterministic replay for multicore debugging [C] //ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, Usa, October. DBLP, 2009:193-206.

[19] Huang J, Zhang C, Dolby J. CLAP: Recording local executions to reproduce concurrency failures [C] //Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation, PLDI, 2013: 141-152.

(下转第 30 页)

实验表明,当处理任务中键值分离的开销很高时,APS 相比产生键值分离的 APK 有巨大的性能提升,故倾向于调整为 HASH。当处理任务中键值分离的开销很低时,APS 相比负载偏移较多的 HASH 有巨大的性能提升,故倾向于调整为负载更加均衡的 APK。

5 结 语

本文提出了一种叫做自适应数据分发策略(APS)的分发方法,为基于 mini-batch 的分布式流处理任务提供更好的性能表现。同时,本文还为数据分发方法的表现性能提供了一种叫作整体分发评估的估计方法。

通过真实数据集上的实验分析,本文验证了 APS 相比现有被广泛使用的静态分发方法的优越性和整体分发评估的准确性。通过模拟数据集上的实验分析,本文进一步分析了 APS 在不同实验设定下的表现能力。

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters [J]. Communications of the ACM-50th anniversary issue: 1958-2008, 2008, 51(1):107-113.

[2] Ben-Haim Y, Tom-Tov E. A Streaming Parallel Decision Tree Algorithm [J]. Journal of Machine Learning Research, 2010, 11(11):849-872.

[3] Kreps J, Narkhede N, Rao J. Kafka: A distributed messaging system for log processing [C]//Proceedings of the NetDB. 2011: 1-7.

[4] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing [C]//Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012: 2-2.

[5] Zaharia M, Das T, Li H, et al. Discretized streams: Fault-tolerant streaming computation at scale [C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 423-438.

[6] Beame P, Koutris P, Suciu D. Communication cost in parallel query processing [DB]. eprint arXiv:1602.06236, 2016.

[7] Koutris P, Suciu D. A Guide to Formal Analysis of Join Processing in Massively Parallel Systems [J]. ACM SIGMOD Record, 2017, 45(4):18-27.

[8] Nasir M A U, Morales G D F, Garcia-Soriano D, et al. The power of both choices: Practical load balancing for distributed stream processing engines [C]//Data Engineering (ICDE), 2015 IEEE 31st International Conference on. IEEE, 2015:137-148.

[9] Nasir M A U, Morales G D F, Kourtellis N, et al. When two choices are not enough: Balancing at scale in distributed stream processing [C]//Data Engineering (ICDE), 2016 IEEE 32nd International Conference on. IEEE, 2016: 589-600.

[10] Katsipoulakis N R, Labrinidis A, Chrysanthis P K. A holistic view of stream partitioning costs [J]. Proceedings of the VLDB Endowment, 2017, 10(11):1286-1297.

[11] Chambers C, Raniwala A, Perry F, et al. FlumeJava: easy, efficient data-parallel pipelines [C]//Acm Sigplan Conference on Programming Language Design & Implementation. ACM, 2010:363-375.

[12] Mitzenmacher M. The power of two choices in randomized load balancing [J]. IEEE Transactions on Parallel and Distributed Systems, 2001, 12(10):1094-1104.

[13] Papadimitriou S, Sun J, Faloutsos C. Streaming pattern discovery in multiple time-series [C]//Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 2005:697-708.

[14] Berinde R, Indyk P, Cormode G, et al. Space-optimal heavy hitters with strong error bounds [J]. ACM Transactions on Database Systems (TODS), 2010, 35(4):26.

(上接第 16 页)

[20] Park S, Zhou Y, Xiong W, et al. PRES: probabilistic replay with execution sketching on multiprocessors [C]//ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, Usa, October. DBLP, 2009:177-192.

[21] Zhou J, Xiao X, Zhang C. Stride: search-based deterministic replay in polynomial time via bounded linkage [C]//International Conference on Software Engineering. IEEE Press, 2012:892-902.

[22] Dunlap G W, Lucchetti D G, Fetterman M A, et al. Execution replay of multiprocessor virtual machines [C]//International Conference on Virtual Execution Environments, VEE 2008, Seattle, Wa, Usa, March. DBLP, 2008:121-130.

[23] Chen A, Moore W B, Xiao H, et al. Detecting covert timing channels with time-deterministic replay [C]//Usenix Conference on Operating Systems Design and Implementation. USENIX Association, 2014:541-554.

[24] Ogata K, Onodera T, Kawachiya K, et al. Replay compilation: improving debuggability of a just-in-time compiler [J]. Acm SIGPLAN Notices, 2006, 41(10):241-252.

[25] Zhang W, Ji X, Song B, et al. VarCatcher: A Framework for Tackling Performance Variability of Parallel Workloads on Multi-Core [J]. IEEE Transactions on Parallel & Distributed Systems, 2017, 28(4):1215-1228.