DynamoRIO 对多线程支持的研究

2009 年 10 月 10 日

左保京
zuobaojing@gmail.com

上海交通大学 CROSSBIT 项目组

目录

摘要

运行大型应用程序远远不是跑通静态链接的单线程 SPEC 那么简单，Crossbit 要支持大型应用程序必须解决的问题之一是对多线程的支持。DynamoRIO 在这方面做得比较好，而且最近宣布开源，这给我们提供了学习的机会。

本文主要研究总结了 DynamoRIO 认为支持多线程需要解决的问题。

关键字： DynamoRIO, 多线程，透明性

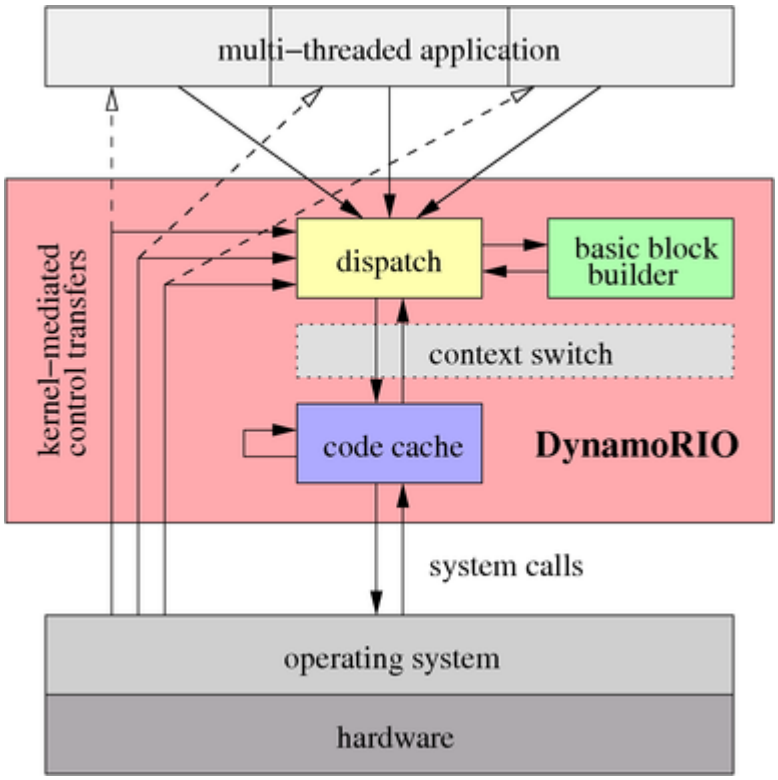
1. 引言

目前 DBT 小组走到了一个十字路口，一种建议是不要那么看重多源多目标的初步设想，而转向在同平台上支持大型应用程序。这与之前几年的研究方向差别比较大，之前的 CrossBit 更加侧重于支持异构平台,由于异构平台的复杂性，Crossbit 只能够运行静态编译的小型标准测试程序(statically compiled SPEC 2000)。

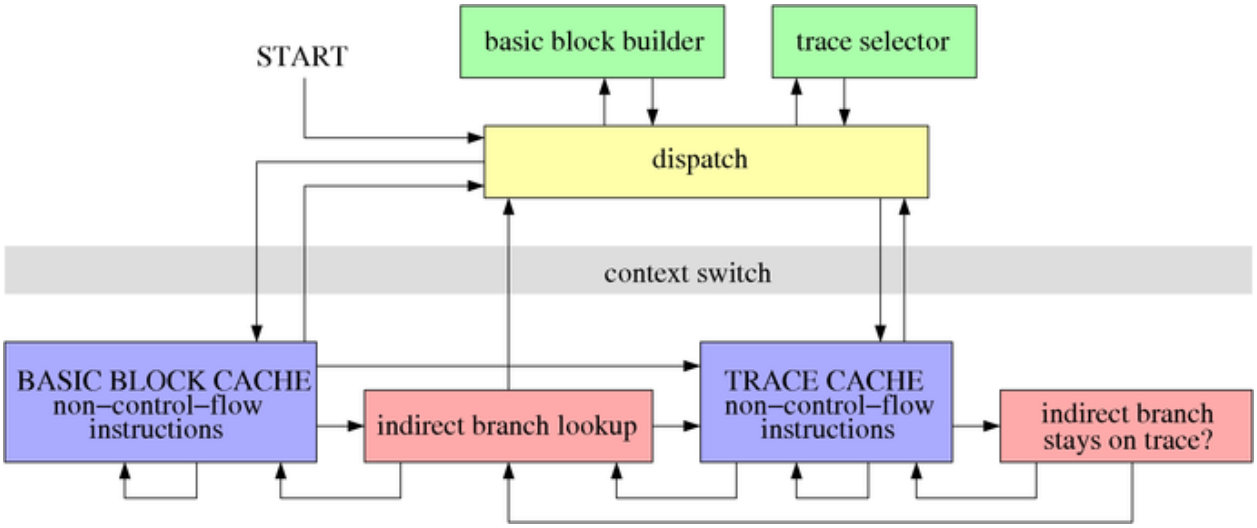
为了实现这个转变，我们只能从头做起，前不久刚刚宣布开源的动态二进制操纵系统 DynamoRIO 正好为我们学习提供了方便。

2. DynamoRIO简介

DynamoRIO 是一个运行于 IA-32/(Windows, Linux)平台上提供对运行时程序监视、分析和插桩的基础系统。DynamoRIO 对程序的操纵环境是本地而不是虚拟化程序环境，它只是作为一个动态库加载到程序的进程空间中。它把程序的执行流从其原来的结构转移到代码缓存(code cache)中，在这个缓存中的代码可以被任意修改。DynamoRIO 与运行的程序占有相同的进程空间，它完全控制了程序的执行，当控制离开缓存或者 OS 控制执行流程结束时 DynamoRIO 依然可以取得控制权：



DynamoRIO 每次复制一个基本块到它的代码缓存中运行，在此基础上将基本块进行链接（link）和优化热路径（trace）来提高程序运行的效率。DynamoRIO 的各个部件和代码缓存之间的执行流程图如下所示：



DynamoRIO 模块结构图

值得一提的是，由于 DynamoRIO 运行在同构平台上，它不需要对程序进行翻译，不需要考虑异构平台上的动态二进制翻译器需要解决的很多问题。

DynamoRIO 的资料（包括代码）可以从其 Google Code 页面（http://code.google.com/p/dynamorio）获得。

本文主要研究 DynamoRIO 对 IA-32/Linux 2.4 平台上的多线程应用程序的支持。

3. 透明性

DynamoRIO 认为，要运行大型应用程序，DynamoRIO 必须得保持绝对的透明性(transparency)。也就是说，不能试图去干扰程序的特性，不能对程序的寄存器使用、调用约定、栈的布局、内存/堆的用法、I/O 和其他系统调用的使用做任何假设。这是 DynamoRIO 自始至终坚持的一点，绝大多数的工作也围绕这些透明性进行展开的。

DynamoRIO 的透明性规则是：避免资源利用冲突；尽可能不要改变原来的程序；迫不得已发生改变时要通过调整让它意识不到发生了变化。

3.1. 资源冲突

DynamoRIO 执行流程需要的资源应该与程序运行所需要的资源完全隔离开，但由于两者运行在同一个进程空间中，这一点很难做到。DynamoRIO 也只是尽可能的去避免了一些冲突，下面详述。

3.1.1. 库的透明性

由于有些库函数具有不可重入性，而且 DynamoRIO 可以在程序执行的任意时刻转移执行流，如果两个切好同时使用同一个库函数就可能会引起问题。

DynamoRIO 的解决办法是，只用系统调用而不用用户库调用外部资源。

在 Linux 平台上，DynamoRIO 面临着 C 库和 LinuxThreads 库的一些问题，如果用__libc__库会引起更多的问题，GNU C 库的不同版本之间差别比较大（比如 2.2 版本和 2.3 版本），不能完全兼容。还有就是在系统调用层和 C 库层对信号数据结构的处理都会碰到问题，有些数据结构在内核层和 glibc 层是不同的。所以，要想完全独立于 C 库就必须自己在不同的内核版本上进行两者之间的转换。一句话，这个问题的根本就是，Linux 实现了外部对用户开放的接口，但内部具体实现是经常变化的。目前 DynamoRIO 也没找到更好的办法完全避免这个问题。所以可以预见如果 Linux 哪个新版本做了一些改动，可能 DynamoRIO 就不能正确地支持上面运行的程序了。

3.1.2. 堆的透明性

为 DynamoRIO 分配的内存必须和为程序分配的隔离开来。因为共享堆申请的函数违反了库的透明性，大部分堆申请函数不具有有可重入性（线程安全，但不可重入）；而且 DynamoRIO 不应该干扰程序的数据空间布局，也不应该干扰程序的内存 bug（错误的透明性）。

DynamoRIO 的解决办法是，实现自己的堆分配器，并且是从系统调用获得内存。

3.1.3. 输入输出的透明性

DynamoRIO 实现了自己的 I/O 函数，以便与程序的 I/O 区分开，免得 DynamoRIO 干扰程序 I/O 的缓存。

3.1.4. 同步的透明性（Synchronization Transparency）

共享锁会引起很多问题，主要是并发不好控制。

最明显的一个问题是一个线程挂起另外一个，当目标线程正在执行 DynamoRIO 的临界函数时不能将其挂起，因为这个临界函数可能不具有可重入性，而且目标线程可能拥有临界

锁。DynamoRIO 对这个问题的处理办法是，截获每个挂起进程的系统调用，然后检查目标程序是否执行到一个安全点上。

还有比如 DynmoRIO 代码中的竞争条件，不能对此做任何假设。这个问题主要体现在指令解码和取消内存映射时。对于取消内存映射，DynamoRIO 的缓存一致性算法中考虑了所有的需要同步的情况。

3.2. 尽可能不要改变原来的程序

程序应该尽可能的保持不发生改变，但这一点很难做到，比如转移代码到代码缓存中。但原来的二进制代码、程序的数据和线程数可以保持不变。

3.2.1. 线程的透明性

为了避免干涉到程序原来的行为（比如程序可能会监控进程中所有的线程），DynamoRIO 不应该创建自己的线程。DynamoRIO 的代码是在程序的线程中运行的，在两者切换的时候要先保存当前线程的上下文然后转到另外的地方去执行。

DynamoRIO 的处理办法是，程序的每个线程都拥有自己的 DynamoRIO 上下文。每个线程有自己专用的 DynamoRIO 线程会使得透明性（线程本地状态和 LinuxThreads 的透明性）比较容易实现，但也会引起性能的下降；用一个统一的 DynamoRIO 线程会使得开销太大，因为程序的每个进程在进入 DynamoRIO 代码中之前得先要等待。

3.2.2. 执行的透明性

程序为了在 DynamoRIO 上运行不应该进行额外的准备，程序的二进制代码不应该发生改变，也不管用了什么编译器或者是不是用了编译器。能够本地运行的程序应该都能运行到 DynamoRIO 上面。

3.2.3. 数据的透明性

DynamoRIO 应该让程序的数据保持不变，这样可以避免很多潜在的透明性问题。保持数据布局不变需要让堆保持透明性（3.1.2）。

3.2.4. 栈的透明性

程序的栈必须与当程序在本地直接执行时是一样的，这就不能用程序的栈来保存临时数据。之所以这样是因为有些应用程序可能不完全严格遵循栈的用法约定，比如有的程序会直接访问栈顶之下某地址的数据。

DynamoRIO 的处理办法是，对每个线程使用 DynamoRIO 自己私有的栈（而不是分享应用程序的线程的栈。如前所述，每个线程中有两个执行流：DynamoRIO 和其上的程序）。还有就是当应用程序在 Dynamo 中执行发生栈溢出但在本地执行不发生栈溢出时，DynamoRIO 应该合理处理。

3.3. 迫不得已发生改变时要通过调整让它意识不到发生了变化

对于那些必须进行的改变，比如中断、信号和异常，DynamoRIO 应该对它们进行合理的处理，使得它们像是在本地执行的一样。

3.3.1. Cache 的一致性

DynamoRIO 必须保持代码缓存中的程序与实际在内存中的程序的一致性。如果程序卸载了某个共享库并加载了一个新的或者修改了自己的代码，DynamoRIO 要在代码缓存中做相应的修改。

3.3.2. 地址空间的透明性

DynamoRIO 需要让程序以为它没有干扰到程序的地址空间。

如果在本地执行的程序有 bug 往非法的内存中写数据并产生异常，在 DynamoRIO 中也应该这样。DynamoRIO 的做法是，保护所有的 DynamoRIO 的内存空间以免被程序无意修改。保护的办法是采用页保护机制。

DynamoRIO 应该对程序隐藏自己在内存中做的改变。比如，有的程序会通过遍历内存的所有区域来判断该区域中是否加载了某个共享库（Mozilla 浏览器会这么做）。DynamoRIO 要能够检测到这样的行为并修改返回给程序的值，以便让程序以为某片区域没有加载共享库。

3.3.3. 程序地址的透明性

虽然程序是在代码缓存中执行的，程序中的地址必须还是原来内存中的值，而不是代码缓存中的地址，在有间接跳转或者操作系统处理信号及异常的时候 DynamoRIO 需要翻译这两种地址。

这在没有链接的时候不难处理，因为在有链接的时候跳转的目标地址是代码缓存中的地址而不是原来内存中实际的地址。这时候就需要 DynamoRIO 能够捕获每此访问返回地址并将其翻译会程序的实际地址。比如，地址无关的代码通过调用（call）下一条指令的地址然

后将返回地址出栈的方式来获取当前的 PC 值（SPEC 2000 中 perlbmk 就有很多这样的情况）。DynamoRIO 通过模板匹配的方式来处理这种情况和其它类似的情况（通过读取程序的返回地址来进行后面的操作）。这个情况概括来说，就是分清楚什么时候该翻译真实地址为缓存中的地址，什么时候不该翻译。

3.3.4. 上下文的翻译

DynamoRIO 必须翻译操作系统到程序的上下文，让程序以为上下文原来是保存在程序代码而不是代码缓存中。对上下文翻译首先是把 PC 值从缓存中的值翻译为其相应的程序空间中的值；然后再确保寄存器中保存了相应的值。

3.3.5. 错误的透明性

程序运行在 DyanmoRIO 上发生的错误必须得与在本地执行时发生的是一样的。比如非法的指令或跳转到非法的地址不应该引起 DynamoRIO 的指令解码器崩溃，这些错误应该传递给程序。当错误被传递给程序时需要让程序以为错误是在本地执行发生的。

在有代码改动的情况下支持精确同步中断是比较困难的，DynamoRIO 目前没有完全解决这个问题。

错误的透明性与堆的透明性、栈的透明性、地址空间的透明性以及上下文的翻译在不少地方是重叠的。

之所以错误需要透明，是因为在发生异常的情况下，有的应用程序会先处理掉这个异常，然后正常往下执行。如果没有错误透明性，这样的程序就没法正确执行了。

3.3.6. 时序的透明性

DynamoRIO 想要尽量让程序不能检测到它是否运行在 DynamoRIO 中，但在某些方面这一点是很难达到的，比如某些执行的确切时序。

改变多线程的执行时序可以发现在本地执行时不会发现的问题，比如有的应用程序在某些时序情况下，某个线程会卸载一个共享库，在此之后可能另外的某个线程会需要前面卸载的共享库。但这也说明程序本身是有问题的，当程序在本地执行时如果处理器发生了某些改变也会引发 DyanamoRIO 中类似的问题。所以这也不能完全说是透明性的问题。

3.3.7. 调试的透明性

调试器应该能够像附在本地执行的程序那样能够附在 DynamoRIO 中的程序上。有的调试器会向被调试的进程中注入一个线程，以便方便访问被调试进程的地址空间，DynamoRIO

应该能够识别这种情况，并且让这个注入的线程在本地执行，但目前的版本中尚未实现此功能。

4. 结论

通过这次对 DyanmoRIO 的研究工作，初步了解了要支持实际的多线程应用程序时需要解决的问题。但是由于 DyanmoRIO 的目的是为了对同平台上运行的程序进行检测和分析，它只是作为一个共享库被加载到进程空间中，当程序加载之后控制程序的执行，并没有牵涉到二进制翻译的问题。

DynamoRIO 的一条最重要的原则就是保持程序在其中运行的绝对透明性，理论上，实现了绝对透明性也就能够支持所有的应用程序了。事实上，DynamoRIO 也没有实现完全的透明性。考虑到异构平台（不同 ISA 和 OS）的二进制翻译，由于平台的差异性，实现透明性是一件非常有挑战性的事情。这个透明性的原则是否过于严格，也就是说在不透明的情况下是否能保证程序的正确执行尚有待继续研究。

5. 参考资料

[1] Derek Bruening. Efficient, Transparent, and Comprehensive Runtime Code Manipulation.Ph.D. Thesis, MIT, September 2004.
[2] http://code.google.com/p/dynamorio
[3] http://dynamorio.org