高技术通讯 2014 年 第24 卷 第5 期:443 ~451

doi:10.3772/j.issn.1002-0470.2014.05.001

动态二进制翻译中间接跳转的热点跟踪及其一致性维护优化①

张晓春②[*,**,***,****] 高 翔[*,****] 郭 崎[*****] 刘宏伟[*,**,***,****] 靳国杰[*,**] 孟小甫[*,**,***,****]

([*]计算机体系结构国家重点实验室 北京 100190)

([**]中国科学院计算技术研究所 北京 100190)

([***]中国科学院大学 北京 100049)

([****]龙芯中科技术有限公司 北京 100190)

([*****]IBM 中国研究院 北京 100094)

摘 要 针对动态二进制翻译(DBT)系统对地址转换过程进行一致性维护的基于锁操作的传统方法会在单线程和多线程执行中都造成严重的执行开销的问题,提出了优化一致性维护的机制,通过跟踪热点跳转,在命中率较高的热点跳转的地址转换过程中,避免使用锁操作,仅在检测到并发读写冲突时进行冗余的地址转换。为实现上述检测过程,提出了指令执行时序和地址转换数据的优化设计方法。在基于 Godson-3 处理器的 X86 模拟平台上,实验结果显示,优化机制极大地提高了二进制翻译的执行效率,在 SPEC CPU2000/2006 单线程测试中能够降低平均27.7%(1.8%到58.5%)的执行开销,在 NPB 多线程测试中能够降低平均18.4%(3.3%到64.6%)的执行开销。

关键词 动态二进制翻译(DBT),间接跳转,多线程,一致性维护,热点跟踪

0 引 言

动态二进制翻译系统中,为在多核处理器上翻译执行多线程程序,处理间接跳转时需要对地址转换过程进行一致性维护。动态二进制翻译(dynamic binary translation,DBT)是一种将源二进制代码实时翻译为本地二进制代码的技术。大量的 DBT 系统,例如 Pin[1]、Qemu[2]、Valgrind[3]等,已被广泛用于指令集翻译[4]、程序检测[5]、动态优化[6,7]、安全性[8]以及体系结构模拟[9]等领域。DBT 系统的主要问题在于其执行开销较高。例如,在 Pin 平台下,SPEC2006 INT 测试集的翻译执行开销达到本地执行(native execution)开销的 3 倍[10];在引文[11]中,一个基于 MIPS 处理器的 X86 模拟平台,其执行开销达到 X86 本地执行开销的 4 倍。上述执行开销的重要来源之一是间接跳转的翻译执行[12-15]。为减少翻译过程的开销,DBT 系统以跳转指令为标志,将源二进制代码分割为源二进制代码块(source binary block,SBB),然后将其翻译并保存为翻译后二进制代码块(translated binary block,TBB)。对于直接跳转,跳转目标在执行前已经确定,代码连接方法(code chaining)[2]能够在 TBB 之间实现直接跳转。但是对于间接跳转,由于跳转目标只有在执行过程中才能确定,DBT 系统不可避免地需要进行 SBB 和 TBB 之间的地址转换以完成跳转,并由此导致了大量的执行开销。

在间接跳转的翻译执行开销中,一致性维护是主要来源之一。随着多核处理器和多线程程序的普及,DBT 系统需要支持多线程的并行翻译执行。当发生并发的间接跳转时,DBT 系统需要避免同时对地址转换数据或代码进行写入、读取、执行等操作,以保证地址转换结果的正确。传统上,DBT 系统基于锁机制实现上述一致性维护过程。由于在程序被翻译执行之前 DBT 系统无法判断在执行中是否存在多线程并行,因此,即使被模拟的程序为单线程程序,DBT 系统依然执行锁操作。统计结果表明,在单线程测试(SPEC CPU2000/2006)和多线程测试

① 863 计划(2012AA011002,2012AA012202,2013AA014301),国家自然科学基金(61100163,61133004,61173001,61232009)和国家"核高基"科技重大专项课题(2009ZX01028-002-003,2009ZX01029-001-003,2010ZX01036-001-002)资助项目。

② 男,1984 年生,博士;研究方向:基于龙芯处理器的二进制翻译系统优化;联系人,E-mail: zhangxiaochun@ict.ac.cn

(收稿日期:2013-08-16)

(NPB[16])中,锁操作都造成了严重的执行开销。针对上述问题,本文提出了优化一致性维护的机制。优化方法通过跟踪热点跳转,在命中率较高的热点地址转换过程中,避免使用锁操作,仅在检测到并发读写冲突时进行冗余的地址转换。为实现上述检测过程,针对基于数据的地址转换方法和基于指令的地址转换方法,本文提出了地址转换数据优化机制和指令执行时序优化机制,分别称为 A-Hash 和 A-Sieve。实验表明,相比 A-Hash,A-Sieve 避免了上下文切换和同步指令的开销,优化效果更好,在 SPEC CPU2000/2006 单线程测试中能够降低平均 27.7%(1.8%到 58.5%)的执行开销,在 NPB 多线程测试中能够降低平均 18.4%(3.3%到 64.6%)的执行开销。

1 间接跳转翻译执行中的一致性维护

1.1 地址转换与一致性维护

在动态二进制翻译(DBT)系统中,源二进制代码块(SBB)被翻译并保存为翻译后二进制代码块(TBB)。当翻译过的 SBB 被再次执行时,DBT 系统根据 SBB 地址,通过地址转换过程找到并执行对应的 TBB。DBT 系统的地址转换机制包括两种类型,即基于数据的地址转换(如 Hash 查找[2]、IBTC[17])和基于指令的地址转换(如 Sieve[18])。对于基于数据的地址转换,DBT 系统维护地址转换数据表,并在数据表项中储存 SBB 和 TBB 地址。对于基于指令的地址转换,DBT 系统维护一系列地址转换代码块,并在代码块中通过立即数指令加载 SBB 和 TBB 地址。当发生间接跳转时,如果跳转目标地址与载入的 SBB 地址匹配,则跳转到相应的 TBB 地址继续执行;如果地址转换过程无法找到匹配的 SBB 地址,则需要翻译目标 SBB,并更新地址转换数据表或代码块序列。

在翻译执行多线程程序时,可能发生并发的间接跳转。如果被更新的地址转换数据或代码块,正在被另一线程读取或执行,则可能导致错误的匹配判定,进而造成地址转换错误。因此,DBT 系统需要针对地址转换过程进行一致性维护。一般地,需要更新地址转换数据或代码块的情况有如下 3 种:

(1) 翻译新的 SBB 后,添加新的地址转换数据或代码块。

(2) 当地址转换数据或代码块失效时,例如发生代码自修改时,删除相应表项或代码块并释放其内存空间。

(3) 一些 DBT 系统根据执行过程中的热点跳转更新地址转换数据或代码块,以便在此后发生间接跳转时,能够更快地找到匹配的 SBB 和 TBB 地址。

对于地址转换数据表或代码块,传统的一致性维护机制在地址转换过程中,通过加锁和解锁保证添加、删除、读取、执行、更新等操作之间的互斥。虽然在翻译执行单线程程序时不会发生并发的间接跳转过程,但是在程序被翻译执行之前,DBT 系统无法判断在执行中是否存在多线程并行的情况。如果要在多核处理器平台上支持多线程并行程序,则在任何情况下,都需要在间接跳转的翻译执行过程中进行一致性维护。因此,即使被模拟程序为单线程程序,一致性维护过程(加锁和解锁)依然被执行并导致了严重的开销。

1.2 锁操作的执行开销

基于 Qemu[2],本文分析了间接跳转处理过程中基于锁操作的一致性维护的开销。Qemu 采用基于数据的地址转换方法,即 Hash 查找方法。为支持多线程程序的翻译执行(本文对 Qemu 中的系统调用过程进行了部分修正,以使其能够健壮地支持多线程程序的翻译执行),Qemu 在进行地址转换前执行加锁操作,并在完成地址转换后解锁,以保证对 Hash 表读写操作的互斥。其执行流程如图 1 所示。在四核 Godson-3 处理器上模拟 X86 体系结构的 Qemu 平台中,本文通过 Godson-3 的性能计数器(performance counter register, PCR),以时钟周期(CPU Cycle,CC)为单位,统计了锁操作的执行时间以及地址转换过程中的其他开销。

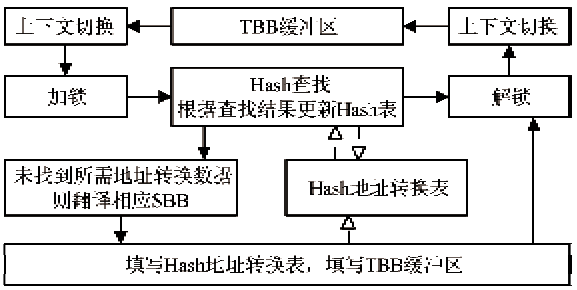

图 1 Qemu 中间接跳转的处理流程

1.2.1 单线程测试中锁操作的开销

图 2 统计了在 SPEC CPU2000/2006 (www.spec.org)测试程序的翻译执行过程中,锁操作的开销以及地址转换过程中 Hash 查找和上下文切换的

开销。数据显示,对于间接跳转密集的测试,如 gzip、parser 等,间接跳转的处理过程在总执行开销中占 60% 以上;对于间接跳转不密集的测试,如 art、cuctus 等,间接跳转造成的开销占不足 5%。从总体上看,间接跳转造成严重的执行开销平均达到 27.7%。其中锁操作的开销远高于 Hash 查找和上下文切换的开销,平均达到总执行开销的24.1%,是翻译执行开销的主要来源之一。

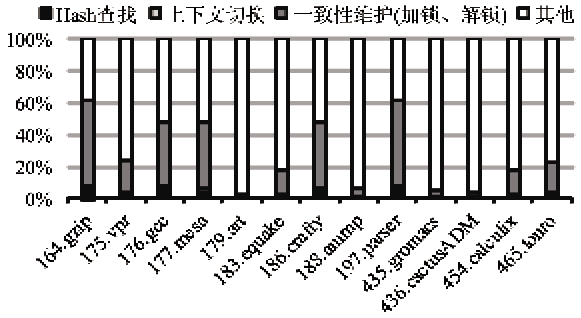

图 2　Hash 查找方法中各部分执行开销的比例

1.2.2　多线程测试中锁操作的开销

在多线程测试中,NPB 测试程序(NAS 并行基准测试程序)被设置为 4 线程,在 4 个处理器核上并行执行。各部分开销的统计结果如图 3 所示。其中,总执行开销为 4 个线程在 4 个处理器核上的执行时间之和。对于并行度较高的测试程序,如 ep,总执行开销约为并行执行时间的 4 倍。

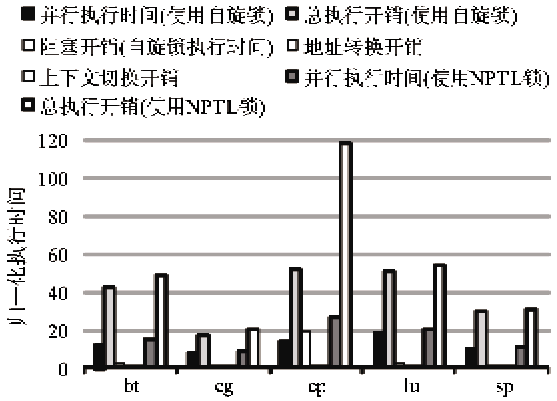

图 3　多线程测试中 Hash 查找方法的执行开销

相比单线程测试,在多线程测试中还存在阻塞开销。当发生并发的间接跳转时,多个线程都需要读取 Hash 地址转换表。锁操作保证每次只有一个线程执行 Hash 查找,致使其他地址转换操作处于等待状态,导致阻塞开销。我们使用自旋锁对阻塞开销进行了评估。自旋锁在发生阻塞时持续占用处理器,并在阻塞结束后退出锁操作,因此自旋锁执行时间的总和即为阻塞开销。根据图 3 的实验数据,对于并行度较高的测试程序 ep,阻塞开销达到总执行开销的 36.1%,是多线程翻译执行的主要开销来源之一。

然而,自旋锁持续占用处理器的特性造成了处理器计算资源的浪费。Qemu 中的锁操作通过调用 NPTL 库实现,其好处是在发生阻塞时,操作系统能够接管被阻塞的线程,并将处理器调度给其他进程,从而合理地利用处理器。但同时这也进一步增加了锁操作的开销。图 3 显示,对于间接跳转不密集的测试,如 sp,阻塞现象不频繁,自旋锁和 NPTL 锁操作造成的开销差异较小;但对于间接跳转密集的测试,如 ep,阻塞现象很频繁,在导致较高的阻塞开销的同时,大大增加了由于线程调度造成的开销,此时使用 NPTL 锁的总执行开销为使用自旋锁的总执行开销的 231%,前者的并行执行时间为后者的 194%。

综上所述,基于锁操作的一致性维护在单线程和多线程翻译执行过程中,都会造成严重的执行开销,是优化的重点。对于基于指令的地址转换方法,传统的一致性维护过程同样需要在开始地址转换前加锁,并在完成地址转换后解锁,因此同样存在严重的锁操作执行开销。为优化这部分开销,本文根据间接跳转的执行特性,提出了能够减少锁操作的一致性维护机制。

2　优化一致性维护机制

间接跳转的执行特点是,大量跳转的跳转目标在局部执行过程中保持不变,从而形成局部热点。在翻译执行过程中,对于热点地址转换数据或代码块的读取或执行操作较多,添加、更新和删除操作较少。在发生并发读取或执行时,取消锁操作不会对执行结果造成影响。因此,优化机制在处理热点间接跳转时不再执行锁操作。

在发生代码自修改时,部分地址转换数据或代码块可能失效。传统机制将删除失效数据或代码块,并释放相应内存空间。此时,在没有锁机制的情况下,并发的读取或执行将导致执行错误。因此,优化机制不删除失效数据或代码块,仅将相应表项设置为失效。为避免内存泄露,表项被置于固定大小的地址转换数据空间或代码空间(统称地址转换空间),并根据 SBB 地址的 Hash 键值进行索引。固定

大小的地址转换空间无法实现对所有跳转地址的转换,因此仅用于热点跳转的地址转换。对于非热点跳转,由传统的地址转换机制完成间接跳转的翻译执行。

为提高热点跳转地址转换过程的命中率,当其不命中时,需要根据传统地址转换的结果更新热点地址转换空间。由于更新操作仅在传统地址转换过程中进行,因此,传统的锁机制能够保证写入操作的互斥。但热点地址转换过程取消了锁操作,因此可能发生读取或执行操作与写入操作的冲突。优化机制通过读取标志数据或执行标志指令检测上述冲突,并在冲突发生时由传统地址转换过程重新进行地址转换。

优化机制正确性的关键是,在未检测到冲突时能够确保地址转换结果的正确。其正确性依赖于特殊设计的指令执行时序。具体说,对于基于数据的地址转换和基于指令的地址转换有不同的实现策略,本文将其分别称为 A-Hash(adaptable Hash-look-up)和 A-Sieve(adaptable sieve)方法。

2.1 A-Hash 方法

A-Hash 方法通过 A-Hash 地址转换数据表实现热点间接跳转的地址转换,其执行流程如图 4 所示。当 A-Hash 不命中时,跳转到传统地址转换机制。本文中,传统地址转换机制采用 Hash 查找方法,称为慢速 Hash 查找。慢速 Hash 查找中的锁机制保证对 A-Hash 表的更新操作彼此互斥。每个 A-Hash 表项中,包含一个变量指示该表项是否已经被修改,称为失效标志。当需要更新 A-Hash 表时,首先标记失效标志,完成标记后再修改 A-Hash 表中的地址转换数据。在标记失效标志之前进行如下判定:如果 A-Hash 表项中的地址转换数据与慢速 Hash 查找的地址转换结果一致,说明 A-Hash 表已经完成更新,此时无论失效标志是否被标记,都消除该标记,且不更新 A-Hash 表。初始化的 A-Hash 表项均标记为失效。

A-Hash 方法在进行地址转换时,首先完成地址转换数据的读取,之后再读取失效标志。如果失效标志未被标记,则根据已读取的数据进行地址转换;否则,认为地址转换数据已失效,退出 A-Hash 地址转换过程并进入慢速 Hash 查找。由于 A-Hash 表能根据程序执行过程动态更新地址转换数据,因此能够自适应间接跳转的局部热点,获得较高的命中率。这样,在 DBT 系统执行过程中,大量 A-Hash 无需锁操作,仅在慢速 Hash 查找中执行锁操作,从而削减了大部分锁操作造成的执行开销。

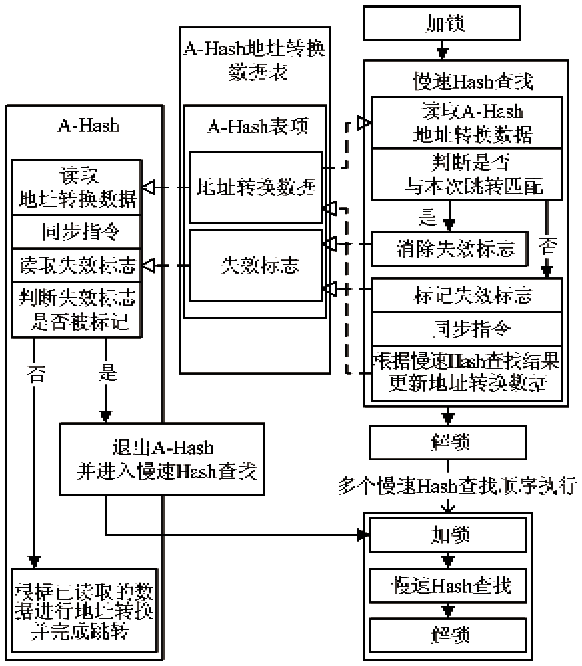

图 4 A-Hash 方法的执行流程

为保证在 A-Hash 中完成读取地址转换数据后再读取失效标志,以及在慢速 Hash 查找中完成写入失效标志后再更新 A-Hash 表,需要在读写地址转换数据和读写失效标志的操作之间加入同步指令。同步指令能够保证该指令之前的其他指令完成提交后,再提交其后的指令,从而在乱序执行的处理器体系结构中使无相关性的指令按规定的时序执行。基于该机制,在 A-Hash 中,如果读取的失效标志未被标记,则可以确定在读取失效标志之前已经读取的地址转换数据一定是在本次 A-Hash 过程中未被修改的,从而可以使用已经读取的数据进行地址转换。实验表明,一条同步指令的执行开销平均为 7.5CC(时钟周期),远小于一次加锁操作(221CC)和一次解锁操作(96CC),因此能够极大削减一致性维护的开销。

2.2 A-Sieve 方法

A-Sieve 方法在固定大小的代码空间中保存 A-Sieve 地址转换代码块,用于实现热点间接跳转的地址转换。发生间接跳转时,DBT 系统根据目标 SBB 地址的 Hash 键值,从 TBB 跳转到 A-Sieve 块进行地址转换。当 A-Sieve 块中加载的跳转地址与本次跳转不匹配时,则退出 A-Sieve 并根据慢速 Hash 查找结果更新相应 A-Sieve 块,从而使 A-Sieve 能够自适应程序执行过程中的局部热点跳转,提高 A-Sieve

的命中率。初始化的 A-Sieve 块被设定为退出 A-Sieve。应用 A-Sieve 方法的 DBT 系统结构如图 5 所示。

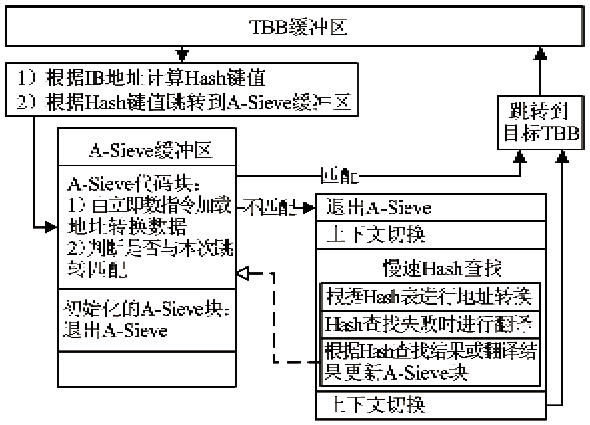

图 5　应用 A-Sieve 方法的 DBT 系统执行流程

A-Sieve 中,通过条件分支来判断加载的地址数据是否与本次跳转相匹配。通过修改该条件分支指令,使 A-Sieve 能够处理在多线程模拟中并发的执行和写入操作。其执行流程如下所述:

(1) A-Sieve 代码块中,首先通过立即数指令加载 SBB 和 TBB 的地址,再通过条件分支指令判断是否与本次跳转匹配。如果匹配,则条件分支不跳转,继而根据 TBB 地址进行跳转并继续执行;如果不匹配,则条件分支指令执行跳转,退出 A-Sieve,进入慢速 Hash 查找。

(2) 在慢速 Hash 查找中更新 A-Sieve 代码时,首先将条件分支指令修改为无条件分支指令,使执行线程退出 A-Sieve。完成上述修改后再更新立即数指令,从而改变 A-Sieve 的地址转换数据。

(3) 在更新 A-Sieve 代码前,进行如下判定:通过读取 A-Sieve 中的立即数指令可以获得其地址转换数据,如果 A-Sieve 块与本次跳转匹配,则将 A-Sieve 中的分支指令恢复为原有的条件分支指令;否则按步骤(2)执行。

图 6 展示了 A-Sieve 方法的执行流程及其一致性维护机制。其中,在 A-Sieve 代码块中,无需执行同步指令。因为 A-Sieve 中的地址数据通过立即数指令加载,一旦指令被处理器获取,其执行结果不受执行时序的影响,因此只需考虑处理器获取指令的时序。而处理器根据指令地址,顺序获取指令。因此,A-Sieve 避免了同步指令造成的执行开销。

A-Sieve 中的立即数指令一定在条件分支指令(或修改后的分支指令)之前被处理器获取,而且慢

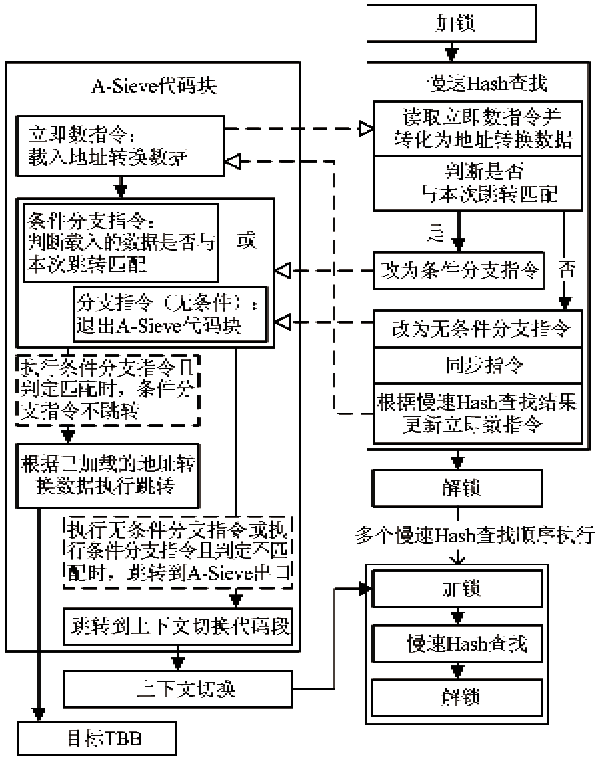

图 6　A-Sieve 的执行流程

速 Hash 查找中的同步指令保证对立即数指令的修改一定在完成修改分支指令之后进行。因此,如果处理器执行了未被修改的条件分支指令,则一定执行未被修改的立即数指令;而一旦处理器获取了被修改的立即数指令,则一定会执行修改后的分支指令。这样,A-Sieve 在未发生执行和写入冲突时正常执行,在发生冲突时一定退出 A-Sieve,从而保证了地址转换结果的正确性。

不同的体系结构对于指令 cache 一致性有不同的处理策略。在某些平台上,如 Godson-3,需要执行指令 cache 同步指令使更新结果在其他处理器核上生效;在另外一些平台上,如 X86,不同处理器核的 cache 由硬件自动完成同步,无需执行特殊 cache 指令。无论何种策略,指令的更新都会导致指令 cache 不命中,造成一定的开销。但在 A-Sieve 命中率较高的情况下,由此造成的执行开销很小。

2.3　热点地址转换空间的配置

上述优化的一致性维护机制能够高效执行的基础是 A-Hash 和 A-Sieve 方法能够保持较高的命中率。一旦其命中率降低,伴随慢速 Hash 查找的锁操作将导致严重的执行开销。一般地,增大热点地址转换空间(即 A-Hash 数据空间和 A-Sieve 代码空间)能够提高命中率,但会增加数据或指令 cache 不命中的概率,增大执行开销。为权衡上述两种开销,

需要通过实验确定合适的热点地址转换空间的大小。

一次 A-Hash 过程的执行开销包括 Hash 查找开销,以及上下文切换和同步指令的开销。Hash 查找时,读取地址转换数据可能发生数据 cache 不命中。一次 A-Sieve 过程的执行开销包括 A-Sieve 代码块的执行时间,以及从 TBB 到 A-Sieve 块跳转过程的执行时间。跳转过程中,可能发生指令 cache 不命中。基于 SPEC CPU2000/2006 测试程序,图 7 统计了配备不同大小的热点地址转换空间时,A-Hash 和 A-Sieve 地址转换过程的平均执行开销($T_{A\text{-}hash}$、$T_{A\text{-}sieve}$,统称 T_A)和命中率($R_{A\text{-}hash}$、$R_{A\text{-}sieve}$,统称 R_A)的变化。

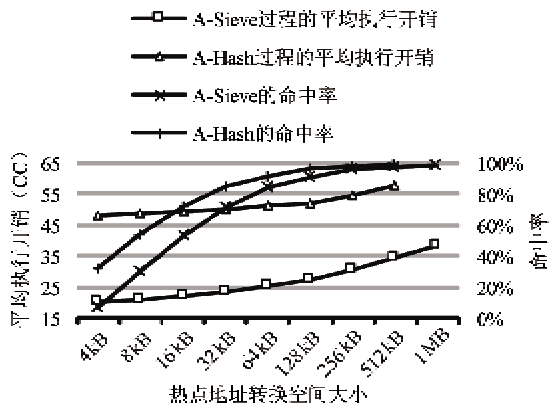

图 7 配备不同大小的热点地址转换空间,A-Hash 和 A-Sieve 地址转换过程的平均执行开销及其命中率

测试表明,慢速 Hash 查找(包含加锁和解锁)的平均执行开销($T_{slow\text{-}hash}$)为371.3CC。DBT 系统执行过程中,A-Hash 或 A-Sieve 与慢速 Hash 查找共同导致的间接跳转翻译执行的平均开销($T_{average}$)可以近似通过公式(1)计算。优化目标是使 $T_{average}$ 达到最小。根据图 7 的实验数据,$T_{average}$ 随热点地址转换空间大小的变化如图 8 所示。权衡执行开销与存储开销,可以确定 A-Hash 数据空间的大小为 128K 字节,A-Sieve 代码空间的大小为 256K 字节。

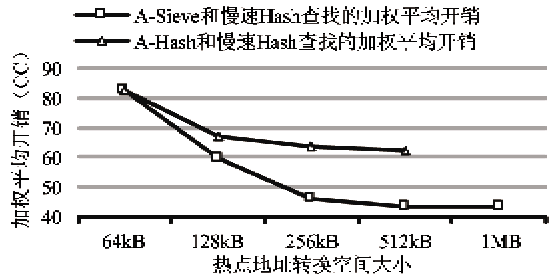

图 8 A-Hash/A-Sieve 和慢速 Hash 查找的加权平均执行开销与热点地址转换空间大小的变化关系

3 实验评估

本章基于 Godson-3 处理器,对以 X86 体系结构为目标的 Qemu 二进制翻译平台及其优化方法进行了评估。Godson-3 是主频 1GHz 的 4 核处理器,采用 64 位 MIPS 体系结构,每核配备 64K 一级指令 cache 和 64K 一级数据 cache,多核共享 4M 二级 cache。实验数据统计了 Qemu 的执行开销,以及应用 A-Hash 和 A-Sieve 方法后的 Qemu 的执行开销。在对比各方法的执行效率时,本文基于程序的本地执行时间,对 Qemu 的执行开销以及使用 A-Hash 和 A-Sieve 方法的翻译执行开销进行了归一化。

基于 SPEC CPU2000/2006 测试程序,单线程测试的实验结果如图 9 所示。数据显示,对于间接跳转密集的测试程序,A-Hash 和 A-Sieve 方法都获得了明显的优化效果。对于优化效果最显著的测试 parser,两种方法分别使执行开销减少了54.1% 和 58.5%;对于间接跳转不密集的测试程序 art,执行开销依然削减了1.7% 和1.8%。平均地,A-Hash 和 A-Sieve 方法分别使执行开销减少了 25.6% 和 27.7%。

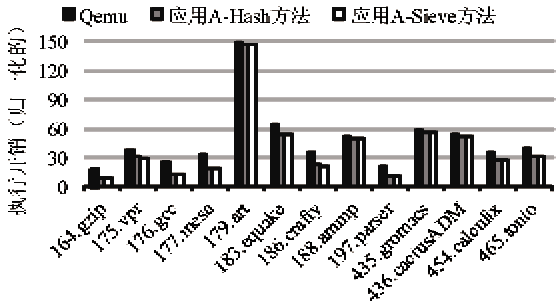

图 9 SPEC 测试中,使用 A-Hash 和 A-Sieve 方法的翻译执行开销以及 Qemu 的执行开销

基于 NPB 测试程序,我们评估了多线程翻译执行中 A-Hash 和 A-Sieve 的执行效率。由于基于锁操作的一致性维护机制在翻译执行多线程程序时还存在阻塞开销,为评估对阻塞开销的优化效果,我们分别在单线程和多线程模式下翻译执行 NPB 测试程序。在单线程模式下,实验结果如图 10 所示。其中,对于间接跳转密集的 ep 测试,A-Hash 和 A-Sieve 方法分别削减了 30.6% 和 32.5% 的执行开销。

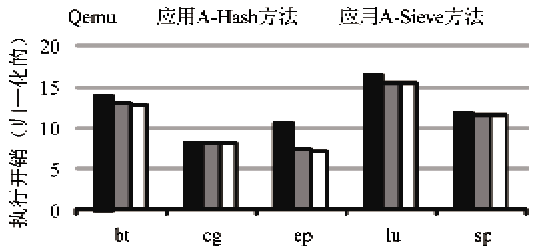

图 10　单线程模式的 NPB 测试中,使用 A-Hash 和 A-Sieve 方法的翻译执行开销以及 Qemu 的执行开销

在多线程模式下,NPB 测试被设置为 4 线程并行执行。由于优化的一致性维护机制只在热点地址转换不命中时才执行锁操作,因此能够减少阻塞现象,并获得比单线程模式下更好的优化效果。实验结果如图 11 所示。数据表明,对于并行度较高的 ep 测试,A-Hash 和 A-Sieve 方法分别削减了 63.8% 和 64.6% 的执行开销,远高于单线程模式下的优化效果,说明优化方法有效避免了阻塞造成的开销。平均地,两种方法在多线程模式下削减了 17.6% 和 18.4% 的执行开销。

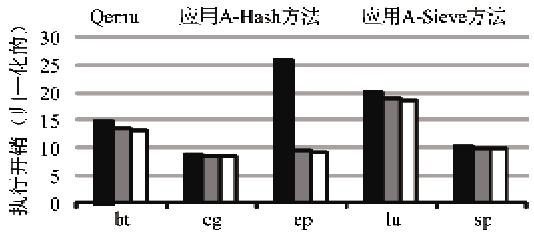

图 11　多线程模式的 NPB 测试中,使用 A-Hash 和 A-Sieve 方法的翻译执行开销以及 Qemu 的执行开销

基于单线程和多线程测试结果,A-Sieve 的优化效果好于 A-Hash,这是由于 A-Sieve 避免了 A-Hash 中的上下文切换和同步指令的开销。而且,相比其他基于指令的地址转换方法(如 Sieve),A-Sieve 中不同地址段之间的跳转较少,降低了指令 cache 不命中的概率,执行开销相对较低。总体上,A-Hash 和 A-Sieve 方法都极大地降低了锁操作造成的执行开销,提高了 DBT 系统的执行效率。

4　相关工作

动态二进制翻译是一个热门研究领域,研究工作的主要方向之一是性能优化[19]。其中,提高间接跳转的处理效率是重要环节。一些方法,如 Sieve、IBTC 等,用以实现间接跳转的翻译执行。评估工作[12]对这些方法的效率进行了对比和分析,总结了各种方法的适用条件。然而,已有的评估主要针对单线程执行环境。对于多线程程序的翻译执行,本文进行了深入的测试分析,剖析了一致性维护造成的性能瓶颈,并提出了针对性的优化方法。

此外,一些方法针对间接跳转的特殊特性进行优化。例如,Inline[20]方法针对跳转目标相对固定的间接跳转,RATS[21]针对返回类型的间接跳转。一方面,这些方法能够被用于本文提出的 A-Hash 和 A-Sieve 处理机制中;另一方面,针对本文所述的 CISC 到 RISC 体系结构的模拟平台,并非所有的方法都能获得有效的性能提升[13]。在应用这些方法时,需要基于特定的应用场景进行取舍。

最新的研究成果在软件和硬件方面提出了新的优化方法,但均有一定的局限性。例如软件优化方法 Spire[15],利用与源二进制代码相对应的影子空间(shadow space)简化了地址转换过程,并通过分析热点跳转降低存储开销。但其局限性体现在,首先难以在 CISC 到 RISC 的二进制翻译中实现,其次增加了不同地址段之间的跳转,可能导致指令 cache 性能的降低。

CAM[21]是一种高效的硬件优化方法。由于地址转换由硬件数据表完成,同时由硬件保证数据一致性,因此执行开销很低。但对硬件功能的特定要求及其硬件开销限制了 CAM 方法的应用范围。此外,由于数据表项数量的限制,CAM 在一些测试中命中率较低[11]。此时,传统方法中的 s 锁操作依然会造成较高的执行开销,本文提出的优化方法能够进一步提高间接跳转的翻译执行效率。

5　结　论

在动态二进制翻译中,间接跳转的翻译执行开销除了包含地址转换之外,还包括从执行流程进入地址转换过程中引入的执行开销,如上下文切换、一致性维护、指令 cache 不命中等。在翻译执行多线程程序时,还存在由于锁操作导致的阻塞开销。本文的评估显示,一致性维护和阻塞开销是多线程测试中的主要开销来源。而且在单线程测试中,一致性维护操作依然被执行并导致严重的开销。

本文针对性地提出了 A-Hash 和 A-Sieve 优化方法,能够削减上述开销。首先,优化的一致性维护机制避免使用锁操作,削减了大量的一致性维护开销和阻塞开销,极大提高了翻译执行效率。此外,基

于指令的 A-Sieve 方法在保证较高命中率的同时，避免了上下文切换和同步指令的开销，对于大部分测试，其效率高于 A-Hash 方法。

参考文献

[1] Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005. 190-200

[2] Bellard F. Qemu, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, 2005. 41-41

[3] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007. 89-100

[4] Bansal S, Aiken A. Binary translation using peephole superoptimizers. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, 2008. 177-192

[5] Bruening D, Zhao Q, Amarasinghe S. Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, 2012. 133-144

[6] Pavlou D, Gibert E, Latorre F, et al. Ddgacc: boosting dynamic ddg-based binary optimizations through specialized hardware support. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, 2012. 159-168

[7] 唐遇星，邓鹏，周兴铭. 基于 trace-cache 的多级动态优化框架设计. 电子学报，2005，33(11)：1946-1951

[8] Hu W, Hiser J, Williams D, et al. Secure and practical defense against code-injection attacks using software dynamic translation. In: Proceedings of the 2nd International Conference on Virtual Execution Environments, 2006. 2-12

[9] Patel A, Afram F, Chen S, et al. MARSSx86: A Full System Simulator for x86 CPUs. In: Design Automation Conference, 2011

[10] Dhanasekaran B, Hazelwood K. Improving indirect branch translation in dynamic binary translators, In: Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments, 2011. 11-18

[11] Hu W, Liu Q, Wang J, et al. Efficient binary translation system with low hardware cost. In: Proceedings of the 2009 IEEE International Conference on Computer Design, 2009. 305-312

[12] Bruening D, Garnett T, Amarasinghe S. An infrastructure for adaptive dynamic optimization. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, 2003. 265-275

[13] Hiser J D, Williams D, Hu W, et al. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In: Proceedings of the International Symposium on Code Generation and Optimization, 2007. 61-73

[14] Borin E, Wu Y, Characterization of DBT overhead. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, 2009. 178-187

[15] Jia N, Yang C, Wang J, et al. Spire: improving dynamic binary translation through spc-indexed indirect branch redirecting. In: Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2013. 1-12

[16] Bailey D H, Barszcz E, Barton J T, et al. The nas parallel benchmarks-summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. 1991. 158-165

[17] Scott K, Davidson J. Strata: A software dynamic translation infrastructure. In: IEEE Workshop on Binary Translation, 2001

[18] Sridhar S, Shapiro J S, Northup E, et al. Hdtrans: an open source, low-level dynamic instrumentation system. In: Proceedings of the 2nd International Conference on Virtual Execution Environments, 2006. 175-185

[19] 李剑慧，马湘宁，朱传琪. 动态二进制翻译与优化技术研究. 计算机研究与发展，2007，44(1)：161-168

[20] Ebcioglu K, Altman E R. Daisy: Dynamic compilation for 100% architectural compatibility. In: Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997. 26-37

[21] Hazelwood K, Klauser A. A dynamic binary instrumentation engine for the arm architecture. In: Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006. 261-270

Hotspot tracing and consistency maintenance optimization for indirect branches in dynamic binary translation

Zhang Xiaochun[*,**,***,****], Gao Xiang[*,****], Guo Qi[*****], Liu Hongwei[*,**,***,****],
Jin Guojie[*,**], Meng Xiaofu[*,**,***,****]

([*]Key Laboratory of Computer System and Architecture, Chinese Academy of Sciences, Beijing 100190)

([**]Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([***]Graduate University of Chinese Academy of Sciences, Beijing 100049)

([****]Loongson Corporation, Beijing 100049)

([*****]IBM Research - China, Beijing 100094)

Abstract

The consistency maintenance for address mapping during indirect branch handling in a dynamic binary translation (DBT) system was studied, and a novel approach to optimization of the consistency maintenance was proposed based on the analysis of the traditional lock mechanism based consistency maintenance scheme's major shortcoming of causing great overhead both in singlethreaded and multithreaded execution. The new method avoids lock operations during the hot branch handling through tracing the hotspot of the indirect branches, and operates redundant address mapping when read-write conflicts are detected. For the detection, a dedicated mechanism was designed to organize the timing sequence of instructions and the address mapping data. The final results of the experiments on the Godson-3 platform emulating the X86 architecture, show that the proposed approach can reduce the execution overhead by 27.7% on average (1.8% to 58.5%) for singlethreaded benchmarks, and by 18.4% on average (3.3% to 64.6%) for multithreaded benchmarks.

Key words: dynamic binary translation (DBT), indirect branch, multithreaded, consistency maintenance, hotspot tracing