基于热例程的动态二进制翻译优化

董卫宇 刘金鑫 戚旭衍 何红旗 蒋烈辉
(数学工程与先进计算国家重点实验室 郑州 450000)

摘 要 依据对系统级程序行为特性的观察,提出了一种基于热例程的动态二进制翻译优化方法。该方法以频繁执行的例程作为优化单位,通过块内和块间优化算法消除动态二进制翻译引入的冗余。相比基于踪迹的优化方法,该方法具有优化单位发现开销更小、代码区域更大、无重复翻译等优点,更适用于系统虚拟机中操作系统代码的优化。在跨平台系统虚拟机监控器ARCH-BRIDGE上的测试表明,通过对内核代码实施该优化方法,SPEC CPUINT 2006程序的效率提升了3.5%～14.4%,相比基于踪迹的优化,性能最大提升了5.1%。

关键词 跨平台系统虚拟机,动态二进制翻译,动态二进制优化,申威处理器
中图法分类号 TP332 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2016.5.005

Hot-routine Based Optimization of Dynamic Binary Translation

DONG Wei-yu LIU Jin-xin QI Xu-yan HE Hong-qi JIANG Lie-hui
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China)

Abstract According to observation of the behavior of system level program, the paper provided a hot-routine based optimization method of dynamic binary translation, which takes frequently executed routines as optimization unit, uses intra-block and inter-block optimization algorithm to remove redundancies introduced by dynamic binary translation. Compared with the trace based optimization, this method has the advantages of less optimization unit discovery overhead, bigger code region, no duplicated translation, and is more suitable for the optimization of OS code in the virtual machine. Evaluation on the cross-platform virtual machine monitor ARCH-BRIDGE demonstrates that, by applying the optimization method to kernel code, performance of SPEC CPUINT 2006 programs gets a speedup of 3.5%～14.4%, and is 5.1% faster than the trace based optimization at most.

Keywords Cross-platform system VM, Dynamic binary translation, Dynamic binary optimization, SW processor

1 引言

跨平台系统虚拟机(Cross-Platform System Virtual Machine)技术可使针对某处理器平台(源平台)编译的操作系统和应用程序运行于其它处理器平台(目标平台),实现软件跨平台透明移植,对体系结构创新、新型处理器推广、软件逆向分析等具有重要意义[1-3]。动态二进制翻译(Dynamic Binary Translation,DBT)[4]是跨平台系统虚拟机实现源平台处理器虚拟化的主要机制,但受源和目标平台指令集架构(Instruction Set Architecture,ISA)差异影响,以上下文无关方式进行源到目标指令的翻译将产生许多冗余,严重制约跨平台系统虚拟机的效率。为提高DBT机制所产生的翻译代码的质量,可采用动态二进制优化(Dynamic Binary Optimization,DBO)[5]方法,在运行期间发现翻译代码的优化机会并进行优化。DBO的主要做法是利用各种剖析(Profiling)手段发现频繁执行的热点代码区域,对热点代码区域进行优化,并将生成的优化代码放入代码缓存,后续执行将优先从代码缓存中取指,以提升程序运行的效率。

优化单位的选取是DBO需要考虑的首要问题,目前大多数系统以基本块或踪迹(Trace)作为优化单位。例如,QEMU[6,7]以基本块为单位进行优化,主要利用活跃性分析消除基本块内的翻译冗余;TransStar[8]和Transmeta CMS[9]以Trace作为优化单位,DAISY[10]以树组(Tree Group,可看作是Trace的变种)作为优化单位;另外,以Dynamo[11]为代表的同构平台上的DBO系统也以Trace作为优化单位。以基本块作为优化单位,其由于大小受限,因此优化机会不多。以Trace作为优化单位扩大了代码区域,增加了优化机会,且Trace内的代码可以比较容易地转换为静态单赋值(Static Single Assignment,SSA)形式,优化方便,是比较好的优化单位。

但以Trace作为优化单位也存在一些问题:1)为扩大优化单位尺寸,Trace一般采用尾部复制(Tail Duplication)技术,不同的Trace间可能具有重复代码,从而导致更大的代码膨胀,对于工作集较大的程序(如系统虚拟机中的操作系统)

到稿日期:2015-03-09 返修日期:2015-07-06
董卫宇(1976－),男,硕士,副教授,主要研究方向为系统虚拟化、体系结构,E-mail:dongxinbaoer@163.com;刘金鑫(1993－),男,硕士生,主要研究方向为动态二进制翻译;戚旭衍(1984－),女,硕士,讲师,主要研究方向为系统虚拟化;何红旗(1972－),男,硕士,副教授,主要研究方向为体系结构;蒋烈辉(1967－),男,博士,教授,主要研究方向为体系结构。

而言将带来内存压力；2）Trace 可能只适用于程序执行的某个阶段，当程序行为变化时，Trace 就会过早地退出（Early-Exit），退化为不良 Trace（Delinquent Trace），造成过多的上下文切换，从而造成性能的降低[12]；3）在系统级虚拟机中由于异常或中断等机制导致的执行流程变更，会为 Trace 构建带来困难。

依据对系统级程序行为特性的观察，本文提出了一种基于热例程（Hot Routine，HR）的系统级动态二进制翻译优化方法，该方法以频繁执行的例程作为优化单位，通过块内和块间优化算法消除动态二进制翻译引入的冗余，更适用于系统虚拟机中公共服务程序（如客户操作系统代码）的优化。在跨平台系统虚拟机 ARCH-BRIDGE 平台上的测试表明，通过对内核代码实施该优化方法，SPEC CPUINT 2006 程序的效率提升了 3.5%～14.4%，相比基于踪迹的优化，性能最大提升了5.1%。

本文第 2 节简要介绍了 ARCH-BRIDGE 系统虚拟机及其 DBT 机制；第 3 节分析了在运行系统级程序时 Trace 机制的一些局限性；第 4 节给出了基于热例程的优化单位选择方法；第 5 节给出了热例程翻译代码的优化方法；第 6 节对本文提出的方法进行了测试；最后总结全文。

2 ARCH-BRIDGE 及其 DBT 机制

ARCH-BRIDGE 是一款跨平台系统虚拟机监控器，其目的是实现 x86 操作系统和应用程序在新型申威处理器（以下简称 SW）平台上的透明高效运行，为后者提供丰富的软件支持。目前，ARCH-BRIDGE 支持 Intel P6 处理器定义的全部定点指令、FPU 指令和 MMX 指令，支持 PCI 总线、南北桥、IDE、VGA、APIC 等典型 x86 接口，能够运行基于 x86 Linux 2.6 内核的操作系统和 BusyBox 工具集中的全部应用程序，通过了 SPEC CPU 2006 测试集中全部定点程序的测试。相比进程级虚拟机而言[13]，ARCH-BRIDGE 需要对整个 x86 指令集架构进行完备的虚拟化，设计和实现难度很大。

ARCH-BRIDGE 的设计方案如图 1 所示，这里重点介绍 ARCH-BRIDGE 的 DBT 机制，略去有关内存和 I/O 虚拟化的内容。DBT 是在异构平台上进行 x86 处理器虚拟化的关键技术，ARCH-BRIDGE 的 DBT 引擎设计要点如下：

1）利用一块类型为 x86_cpu_state_t 的内存区域模拟 x86 寄存器以及 DBT 工作所需的数据（如标志位处理相关信息 cc_op、cc_src、cc_res 等），由于该区域被频繁访问，ARCH-BRIDGE 利用 gcc 的全局寄存器变量特性，令某个宿主寄存器（如 SW 的 r15 寄存器）指向该内存区域的基址，再辅以偏移量来访问其中的数据。

2）为提高翻译效率，没有采用其他虚拟机监控器惯用的中间表示，而是采用 x86 指令到 SW 指令直接翻译。某些 x86 指令（如 INT 指令）的操作十分复杂，ARCH-BRIDGE 利用辅助的 C 代码仿真这些指令。

3）为提高翻译代码的重用率，采用了两级代码缓存（Code Cache，TC）机制来保存翻译过的 x86 代码。其中，二级代码缓存用于保存翻译过的 x86 基本块，当二级代码缓存满时，采用简单的全清空策略清除所有翻译块；一级代码缓存用于保存更大的优化翻译单位（如热例程的翻译代码），当一级代码缓存空间不足时，利用专用的线程按 LRU 策略淘汰不常用的翻译代码。

4）为降低 DBT 引擎与翻译块间的上下文切换开销，采用翻译块链（Block Chaining）机制链接翻译过的代码块；为避免构成循环的翻译块阻碍对中断的响应，需在周期性定时器超时时将当前正在执行的翻译块从块链中移除，以便返回执行引擎检查中断。

5）在取指令或翻译块执行过程中检测到 x86 异常时，采用 longjmp()将流程转移到 DBT 引擎入口位置，进行异常检查和派发工作。ARCH-BRIGE 支持精确异常，可确保在异常派发前将机器状态（包括通用寄存器、标志寄存器、EIP、内存内容）恢复到异常指令执行前的状态。

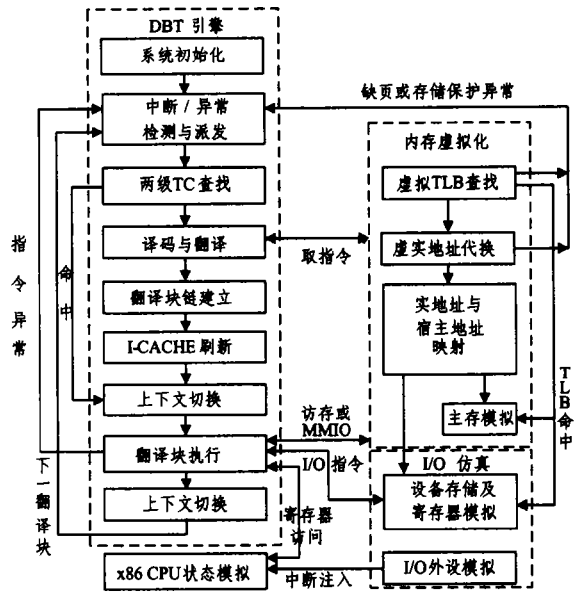

图 1 ARCH-BRIDGE 设计方案

3 Trace 的局限性

动态二进制翻译系统大多以 Trace 作为优化单位，Trace 的构造一般采用 NET（Next Execution Tail）方法[14]。NET 方法通过监控程序中的后向分支（Backward Branch）来发现 Trace，如果后向跳转的目标地址的执行次数超过了预设的门限，则认为该地址为 Trace 的头部（Head），并将紧接着执行的一个路径作为 Trace 的尾部（Tail），直到再次发现反向跳转、遇到间接转移指令或路径超出预设的长度。为支持跨例程的 Trace，NET 方法将 CALL 和 RET 指令也加入到了监控范围，即如果 CALL 或 RET 指令的目标地址相比自身为低地址，则该目标地址也可能被作为 Trace 头部。

将 NET 方法引入到 ARCH-BRIDGE 系统中（后向跳转目标地址的执行次数门限设置为 100），在虚拟机中运行 Linux 操作系统及若干 SPEC 测试程序（401.gzip、429.mcf 和 456.hmmer）一段时间并采集 Trace。实验结果表明，相比应用程序代码，按照 NET 方法为系统级代码构造 Trace 表现出一些不同的特性。

首先，NET 方法可能遗漏一些由于例程频繁调用而产生的 Trace。如图 2 所示，在实验结果中，Linux 内核代码中至少存在两处对入口地址为 0xC1019E80 的函数的反向调用，调用指令所在的块地址分别为 0xC101AC17 和 0xC101C9EA，调用次数分别为 184 和 730，且使块 0xC1019E80 执行次数超过门限的调用来自块 0xC101AC17，

因此图 2 中的 Trace-1 被构造出来。由于 NET 方法以头部地址标识 Trace，即使后续出现多次来自块 0xC101C9EA 的向块 0xC1019E80 的后向跳转，也不会将 Trace-2 识别出来，并且由于 Trace-1 不是 0xC101C9EA 调用 0xC1019E80 后的热路径，因此将导致大量的 Trace 早期退出，影响性能。

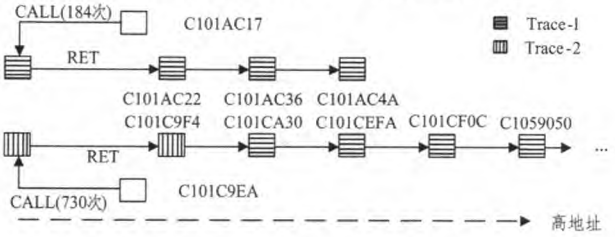

图 2　内核代码中的 Trace 示例

造成上述问题的主要原因在于操作系统内核代码具有比较明显的层次结构，且受地址布局影响，底层函数一般位于低地址，上层函数一般位于高地址，如果某个底层函数 F 被多个上层函数频繁使用，则 F 可能向多个上层主调函数返回，导致存在多条以 F 的入口为头部、以 F 的函数体和主调函数的部分代码为尾部构成的 Trace（当 F 中不存在其它后向转移时），但由于前述的 NET 方法的局限性，将只能识别这些 Trace 中的一条。用户态代码同样存在调用公共代码的情形（例如，应用程序大多会调用 Glibc 中的库函数），但由于应用程序的执行映像一般被加载到低地址（如以 0x08000000 开始的区域），而库函数被加载到较高的地址（如 0xb0000000 以上的区域），Trace 头部位于主调函数中，各个 Trace 的头部地址差异较大，因此较少出现 Trace 被遗漏的情况。

上述实验中，在内核态代码中共观测到 58 组头部相同但源地址不同的 Trace，其中 34 组具有不同 Trace 尾部，占比 58.6%；在用户态代码中存在 16 组头部相同但源地址不同的 Trace，其中 5 组具有不同 Trace 尾部，占比 31.2%。这说明，对于内核态代码，采用 NET 方法构造 Trace 更易造成遗漏，并且更容易出现 Trace 的早期退出。

其次，对于系统级代码而言，Trace 具有较多的重复块。在实验中关闭了 Linux 内核的地址随机化支持（将 randomize_va_space文件置为 0），以便将公用动态链接库加载到固定的虚拟地址空间，实验结果如图 3 所示。在所有的 Trace 中，位于库代码区域和内核代码区域的 Trace 之间具有较多的重复块，且重复率较高，而应用程序映像区域的 Trace 的重复块较少，且重复率较低。例如，内核地址为 0xC1007DC4 区域的块重复了 11 次，即在 11 条 Trace 中均有出现。造成该现象的主要原因在于内核中存在较多的、为多个进程共用的例程，这些例程可能被来自应用程序的多个执行路径所引用，重复翻译并优化这些公共代码将带来优化环节的开销，并增加代码缓存的压力。

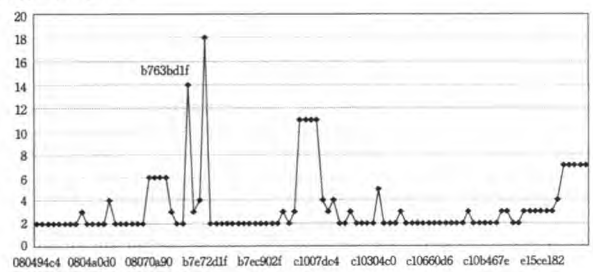

图 3　用户态与内核态代码中的 Trace 中的重复块情况

本文认为，在系统级虚拟机中存在较多的被多个应用程序频繁使用的热例程，选择这些例程进行优化可以提升系统的效率，并且相比 Trace 机制而言，以热例程为优化单位具有如下优点。

1）发现热例程的开销比 Trace 更小。发现 Trace 要求对所有的流程转移指令（包括 JMP/Jcc/CALL/RET）插桩，并且不能启用翻译块链机制，否则将无法识别 Trace 尾部的各个块。如第 3 节所述，发现热例程仅要求对 CALL 及间接跳转指令插桩，并且允许在二进制翻译的整个过程中启动块链。

2）热例程的代码区域比 Trace 更大。Trace 仅具有单个执行路径，早期退出的几率更大，而热例程内部允许存在多条执行路径，退出优化单位的几率更小。

3）热例程不存在重复翻译优化问题。多条 Trace 间可能具有重复的块，将导致重复翻译优化，并增加代码膨胀率，而以热例程为单位不会导致重复翻译优化问题。

4　热例程的选择

本文以热例程作为优化单位，一旦选中热例程，则将其中的代码作为一个整体进行优化。热例程的选择包括两方面的工作：1）从系统工作集中识别出频繁执行的热例程集合；2）发现热例程中的代码区域。

4.1　热例程的识别

为识别热例程，ARCH-BRIDGE 不在 CALL 指令与其调用目标之间建立块链（其它流程转移指令与其转移目标间仍可建立块链），这样每个以 CALL 指令结尾的基本块执行完毕后将返回 DBT 引擎，从而使 DBT 引擎有机会统计例程的被调用次数。

x86 平台上多使用间接跳转形式的 JMP 指令实现模块间的动态链接，即主调程序利用 CALL 指令调用以间接跳转形式的 JMP 指令来实现的桩代码，由动态链接程序根据动态链接库的加载地址来修改 JMP 指令的目标地址，再由桩代码转移到动态链接库中的被调例程。在仅对 CALL 指令插桩的情况下，将会导致识别出一些仅含桩代码的热例程，而忽略了动态链接库中的例程主体。为此，ARCH-BRIDGE 将间接跳转形式的 JMP 指令也纳入插桩范围，以识别被频繁调用的动态链接库中的程序。另外，考虑到系统调用、异常或中断派发是进入内核代码的主要方式，ARCH-BRIDGE 将 x86 中断派发表（Interrupt Dispatch Table，IDT）中的入口地址也纳入监控范围。

由于系统中例程的数量很大，很难预先决定采用多大的存储空间保存例程的入口与调用次数，为此 ARCH-BRIDGE 将例程的调用计数保存在例程入口块对应的翻译块描述符中，DBT 引擎每次在基本块代码缓存中查找例程入口对应的翻译块时将有机会更新例程的调用计数。

当例程的被调用次数达到了预先设定的门限时，将例程看作热例程。相比 Trace，上述热例程发现方法避免了对全部流程转移指令的插桩，简化了实现，降低了动态剖析的开销。

4.2　热例程代码区域的发现

ARCH-BRIDGE 按照翻译块链的提示来识别热例程中

所包含的代码。当确定热例程后，从其入口开始，按照广度优先的方法，沿着翻译块链来还原例程的控制流。为避免引入重复代码，热例程的代码区域不包含子例程中的代码。由于CALL指令与其调用目标之间不存在块链，因此当遇到子例程调用时，将无法沿着翻译块链继续发现当前例程的代码，ARCH-BRIDGE的做法是从调用子例程的CALL指令的返回地址(也就是包含CALL指令的基本块的下一个块)开始，继续热例程的代码发现过程。

设热例程的入口块为hr_entry，热例程所包含的块集合为hr_blk_set，热例程代码发现算法如图4所示。由于ARCH-BRIDGE不会在以RET指令结尾的块及其目标块间建立链接，因此当遇到热例程出口块时，该算法会终止以出口块为起点的搜索。

```
void hr_code_discover() {
  hr_blk_set =∅;
  working_set = {hr_entry};
  while(working_set !=∅) {
    从 working_set 中取出一个块 cur_blk;
    hr_blk_set = hr_blk_set ∪ {cur_blk};
    if(块 cur_blk 以 call 指令结尾) {
      next_blk = 以 call 指令返址为入口的块;
      if(next_blk∉hr_blk_set)
          hr_blk_set ∪ {next_blk};
    } else {
      for(cur_blk 经块链连接的每个块 next_blk)
        if(next_blk∉hr_blk_set)
          hr_blk_set ∪{next_blk};
          记录 cur_blk 和 next_blk 间的前趋后继关系;
    }
  }
}
```

图4　热例程代码发现算法描述

需注意的是，上述算法所能识别的代码依赖热例程的实际执行路径，因此一般不会发现热例程的全部代码，后续的优化是对热例程部分代码的优化。另外，由于动态二进制翻译的特性，所产生的基本块为动态基本块，少数基本块间会有重叠，但不会造成大量的代码膨胀。

5　热例程的优化

虽然例程中所包含的源指令一般会经过编译优化，但经过二进制翻译后仍会引入大量冗余，去除此类冗余可以有效地提升翻译代码的质量。在未采取优化措施的情况下，load/store冗余和标志位处理冗余是两类主要的冗余，并且在块内翻译和块间转移时均有体现。

以翻译图5中B1块为例来说明块内冗余。在翻译基本块B1时，由于DBT引擎利用内存模拟x86寄存器，在无优化的情况下，即使orl已装入edx，在翻译movl和subl时仍将重复装入edx，导致load冗余。同理，即使orl对edx的修改可能被后续的subl指令所覆盖，在无优化的情况下，仍需要在orl指令的翻译指令序列的尾部保存edx，导致store冗余。x86处理器支持精确异常，即使采用懒惰思想计算标志位，也需要在每条影响标志位的指令结尾保存标志位处理信息(包括影响标志位的指令的源操作数、目的操作数和操作语义)，但若movl未引发异常，则orl指令对标志位的影响将被subl指令覆盖，导致标志位处理冗余。

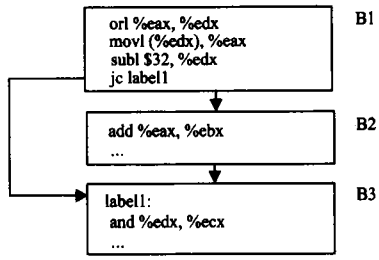

图5　热例程翻译冗余举例

块间转移导致的冗余对性能的影响也很大。简单的翻译策略一般要求在基本块的结尾提交机器的状态，在翻译后续块时需要重新从内存中加载所需的寄存器的值。以图5中B1块向B2块和B3块的转移为例，在无优化的情况下，在B1块的尾部需要将已修改的寄存器(eax、edx)的内容提交到内存，B2块在引用这些寄存器时，需要重新从内存中装入，导致load冗余。同理，即使在B2和B3中覆盖了B1对标志位的影响，也需要在B1尾部提交标志位处理信息，导致标志位处理冗余。

需要说明的是，在系统虚拟机环境下只能进行一些比较保守的优化，系统代码中可能存在某些对程序结果无影响的代码，如用于延时的代码、在高级语言中利用volatile关键字修饰的不可优化代码等。在二进制翻译情况下，由于高层语义的缺失，采用激进的优化方法(如死代码删除等)可能造成程序的行为错误，因此本文仅探讨对二进制翻译引入的load/store冗余和标志位处理冗余的优化。

5.1　热例程的块内优化

为消除例程块内翻译冗余，ARCH-BRIDGE引入了虚拟寄存器(Virtual Register，VR)的概念。其中，x86的8个通用寄存器以及3个标志位处理相关的变量(cc_op、cc_src、cc_res)被看作全局虚拟寄存器(Global VR，GVR)，其它翻译过程中所需要的用于保存中间操作数或地址的寄存器被看作临时虚拟寄存器(Temp VR，TVR)。GVR的生命周期可跨越基本块，而TVR的生命周期仅为当前指令。每个VR都对应固定的编号(可看作VR的名字)，例如x86寄存器eax被看作GVR，其编号为VRI_EAX(0)，用于保存有效地址的TVR的编号为VRI_LA0(11)。

在翻译每条指令之前，首先根据指令的译码信息以及指令的操作进行VR的分配，并且在分配VR的同时为其指派一个宿主机上的物理寄存器(Physical Register，PR)。为减少寄存器间的拷贝操作，多个VR可能被指派相同的PR，可利用引用计数表示一个PR当前被多少个VR所引用，仅当引用计数为0时，才真正释放PR。另外，当PR正在被多个VR所引用，且指令的操作会修改其中的某个VR时，需利用写时拷贝机制为该VR重新指派PR，并在必要时(如只修改VR的部分字节时)生成寄存器拷贝指令复制原值。在结束当前指令的翻译前，TVR将被释放，但GVR继续存活，以供块内的后续指令及后续的基本块使用。表1给出了ARCH-

BRIDGE定义的各种与VR有关的操作。

表1 VR相关操作

操作	语义
vr_alloc_global (vri,load)	分配全局虚拟寄存器vri,并为其指派物理寄存器,Load指示是否需要生成指令从内存装入vri的初值。若vri已分配,该原语无实际操作
vr_alloc_tmp(vri)	分配临时虚拟寄存器vri,并为其指派物理寄存器
vr_set(vri_src,vri_dst,type)	利用vri_src的值修改vri_dst,type给出操作宽度
vr_assign(to,from)	将虚拟寄存器from指派给to
vr_assign_const(vri,val)	将常量指派给vri,无需分配物理寄存器,主要用于设置cc_op
vr_mark_dirty(vri)	标志全局虚拟寄存器vri为脏
vr_commit()	将所有标记为脏的全局虚拟寄存器的值提交到x86_cpu_state_t(内存)中
vr_free(vri)	释放vri,若为vri所指派的物理寄存器的引用计数为0,则同时释放物理寄存器
vr_get_pr(vri)	给定vri,返回其对应的物理寄存器的编号

利用上述虚拟寄存器机制及其操作,可有效去除块内的翻译冗余。图6给出了B1块中的orl指令的优化翻译流程。由于在orl的翻译中利用vr_alloc_global分配了全局虚拟寄存器eax及edx,并生成Load指令从内存中装入它们的值,因此在翻译后续的movl和subl指令时将无需再次装入这些寄存器的值。另外,由于cc_dst和cc_op的值分别来自于虚拟寄存器edx和常量,因此可使用vr_assign为它们指派值。由于edx、cc_dst和cc_op的值被改变,因此需要利用vr_mark_dirty标记它们为脏。注意,vr_assign和vr_mark_dirty均不生成任何指令。

```
vr_alloc_global(VRI_EAX, 1);
vr_alloc_global(VRI_EDX, 1);
gen_prim_orl(vr_get_pr(VRI_EAX), vr_get_pr(VRI_EDX), vr_get_
pr(VRI_EDX));
vr_mark_dirty(VRI_EDX);
vr_free(VRI_CCSRC);//对于orl指令,只需用cc_dst即可恢复标志位
vr_assign(VRI_CCDST, VRI_EDX);
vr_mark_dirty(VRI_CCDST);
vr_assign_const(VRI_CCOP, OP_ORL);
vr_mark_dirty(VRI_CCOP);
```

图6 orl指令的优化翻译流程示例

跨平台系统级虚拟机必须考虑支持源处理器的精确异常机制。保守翻译需要在每条可能产生异常的x86指令前提交机器状态,由于异常的实际发生几率很低,而后续指令可能再次修改机器状态,因此这种翻译方法可能导致很多的冗余。ARCH-BRIDGE的思路是尽量将机器状态的提交延迟到异常真正发生前。

考虑到访存指令是最主要的异常来源(页面故障),ARCH-BRIDGE在已有的以C实现的虚拟机内存读写操作mmu_rd和mmu_wt的基础上,引入指令片段mmu_rd_fast和mmu_wt_fast来负责异常检测以及无异常情况下的内存读写。在翻译访存操作时,首先生成指令来调用mmu_rd/wt_fast,之后再生成对mmu_rd/wt的调用指令,仅当可能引发异常时(如TLB未命中、访存操作跨页、页面有写保护等),mmu_rd和mmu_wt才会得到执行。

上述做法的好处主要有两点:1)提升了访存速度,mmu_rd/wt_fast直接利用机器指令实现,大小约为20条指令,并且采用SW的参数传递专用寄存器r16—r21进行操作,避免了从翻译块转移到C函数时的上下文切换工作;2)可以延迟机器状态的提交,实测中约99.4%的访存操作可以在mmu_rd/wt_fast中以无异常的方式执行,在翻译时,可以将机器状态的提交指令放到mmu_rd/wt_fast与mmu_rd/wt之间,即仅当异常发生几率很大时,才提交机器状态。

图7给出了图5的B1块中的movl指令的优化翻译流程示意。首先,利用vr_alloc_global为edx分配寄存器,由于edx已经在之前的orl指令中装入,因此该操作不会生成任何指令。之后,生成指令调用mmu_rd_fast,结果放在临时寄存器VRI_TMP0中,若访存过程中无异常发生(返回值r16为1),则直接跳转到LABEL_OK,并将结果赋予eax。仅当访存发生异常时,才利用vr_commit生成指令来提交由orl指令引起的机器状态变化,并调用慢速的mmu_rd。图7中带下划线的代码可看作是很少执行的补偿代码。

```
    vr_alloc_global(VRI_EDX,1);
    vr_alloc_tmp(VRI_TMP0);
    vr_assign(VRI_LA0,VRI_EDX);
    //生成指令调用mmu_rd_fast,结果放入VRI_TMP0中
    //返回值r16为1表示无异常
    …
    gen_insn_br(INSN_BLBS,r16,LABEL_OK);
    vr_commit();
    //生成指令调用mmu_rd,结果放入VRI_TMP0中
    …
LABEL_OK:
    vr_assign(VRI_EAX,VRI_TMP0);
    vr_mark_dirty(VRI_EAX);
    vr_free(VRI_TMP0);
    vr_free(VRI_LA0);
```

图7 movl指令的优化翻译流程示例

5.2 热例程的块间翻译优化

ARCH-BRIDGE尽量将基本块尾部的活跃的GVR传递给其后继基本块,以此避免在基本块尾部提交机器状态,并减少在后继基本块中的GVR的重复分配和装入工作,从而消除例程块间的翻译冗余。

设某个基本块x的后继基本块为y,且y是控制流图中的汇点(具有多个前趋基本块),由于y的不同前趋尾部可能具有不同的GVR活跃信息,此时将x块尾部的活跃寄存器列表传递给y块是不安全的。ARCH-BRIDGE采用扩展基本块(Extended Basic Block,EBB)来限定活跃寄存器列表的传递边界,并在EBB给定的控制转移的引导下翻译基本块。

扩展基本块是以某个基本块为入口的热例程控制流图的最大子图,在该子图中,除了入口基本块外的其它基本块均不是汇点。显然,EBB是由若干基本块构成的树(或有向无环图),并且可以将热例程的控制流图划分为若干扩展基本块的集合。图8给出了构造EBB集合的算法MakeEbbSet,其中G为热例程对应的控制流图,EBBS为G中包含的扩展基本块集合,TMP_EBB为临时EBB,ROOTS为所有EBBS的根构成的集合。给定基本块x,以$succ(x)$表示其后继集合,以$pred(x)$表示其前趋集合。给定图g,以$NODES(g)$表示其节点集合,以$EDGES(g)$表示其边集合。BLOCK表示基本块对应的数据类型。

由EBB的定义可知,EBB中的任意非入口块y具有唯一的前趋块x,且从热例程入口到y的任何路径必经过x,因此

将块 x 出口处的 GVR 活跃信息传递给 y 是安全的。

```
void MakeEbb(BLOCK p,BLOCK s) {
  BLOCK x;
  NODES(TMP_EBB) ∪= s;
  if(p !=Nil)
    EDGES(TMP_EBB)∪=⟨p,s⟩;
  for(每个 xsucc(s))
    if(集合 pred(x)的元素个数为 1) {
      if(x∉NODES(TMP_EBB)) MakeEbb(s,x);
    } else { ROOTS ∪= s; }
}
void MakeEbbSet() {
  BLOCK r;
  EBBS=φ;
  ROOTS={G 的入口};
  while(ROOTS 非空) {
      从 ROOTS 中移除元素 r;
      if(r 不是当前 EBBS 中任意扩展基本块的根) {
          TMP_EBB = φ;
          MakeEbb(Nil,r);
          EBBS∪=TMP_EBB;
      }
  }
}
```

图 8　扩展基本块构造算法描述

为方便后续的介绍，将 EBB 中的基本块的出口分为 3 类，图 9 给出了热例程中各类出口的示意，其中虚框标出了识别出的各个 EBB，若块入口是子例程的返回地址，则强制标识该块为汇点。注意，EBB 中的每个块最多有两个出口，当块以 Jcc 指令结尾时出口数量为 2，以 JMP/CALL/RET/INT/IRET 指令结尾时出口数量为 1。

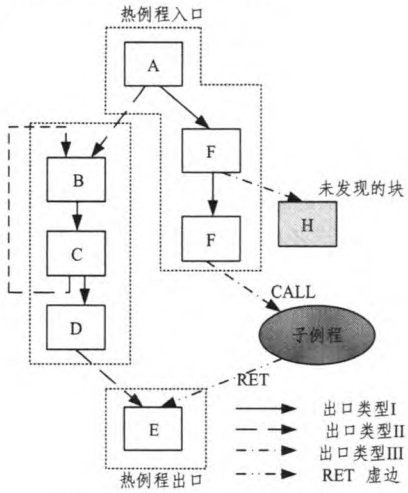

图 9　EBB 中块的出口类型示意

1)类型 I，基本块的出口在 EBB 中存在后继。

2)类型 II，基本块的出口在 EBB 中不存在后继，但在热例程的控制流图中存在后继。

3)类型 III，非类型 I 和 II 的出口。例如，考虑到动态执行的特性，在热例程代码区域发现阶段得到的可能是热例程的部分控制流图，若尚未发现某个出口的后继块，则将该出口归类为类型 III。再如，当块以 CALL/RET/INT/IRET 或间接转移类型的 JMP 指令结尾时，将其出口归类为类型 III。

在构造完热例程的扩展基本块集合后，ARCH-BRIDGE 将按图 10 给出的 hr_trans_opt 算法分两个阶段对热例程进行优化翻译，其中 EBBS 为前一阶段构造的扩展基本块集合，TMP_BLK_SEQ 为临时基本块序列。第一阶段按照广度优先的顺序翻译 EBB 给出的控制流图中的每个块，这样可保证翻译每个块时其前趋出口处的 GVR 活跃信息已存在。在每个块的入口处，采用从其前趋块传递下来的 GVR 活跃信息来指导块内指令的翻译。对扩展基本块的入口块而言，可认为初始 GVR 活跃信息为空。在每个块的出口处(块最多有两个出口，即块以 Jcc 指令结尾的情况)，若该出口为类型 I，则无需提交 GVR 信息，并保持各 GVR 的活跃性；若出口为类型 II，则生成指令，提交并释放各个 GVR；若出口为类型 III，则除生成指令提交并释放各个 GVR 外，同时生成返回 DBT 引擎的指令，退出热例程的执行。第二阶段在热例程中的基本块之间建立块链，对于热例程中的每个基本块，若其存在类型 I 或类型 II 的出口，则建立该出口到其后继基本块之间的块链。对于循环而言，该阶段可建立一条循环尾部到循环入口的块链。

```
  void hr_trans_opt() {
  // phase 1
  for(集合 EBBS 中的每个 ebb) {
    将 ebb 的根加入 TMP_BLK_SEQ 的尾部;
    while(TMP_BLK_SEQ 非空) {
      从 TMP_BLK_SEQ 头部移除块 b;
      将 b 的后继块加入 TMP_BLK_SEQ 的尾部;
      利用 b 的前趋提供的 GVR 活跃信息翻译 b;
      for(b 的每个出口 e)
        if(e 为类型 I 的出口)
          记录 GVR 的活跃信息;
        else if(e 为类型 II 的出口)
          提交并释放活跃 GVR;
        else {
          提交并释放活跃 GVR;
          生成指令返回 DBT 引擎;
        }
      }
    }
  }
  // phase 2
  for(热例程控制流图中的每个块 b)
    for(b 的每个出口 e)
      if(e 为类型 I 或 II 的出口)
        建立该出口到其后继间的块链;
}
```

图 10　热例程翻译优化算法描述

5.3　向 ARCH-BRIDGE 中的集成

利用 ARCH-BRIDGE 提供的两级代码缓存机制，当完成热例程的优化翻译后，将其对应的翻译代码加入一级代码缓存，并在一级代码缓存的查找表中生成一个以热例程入口物理地址为关键字的表项。另外，如果热例程中包含对其它子例程的调用，则还要将返回地址作为关键字加入一级代码缓存。在后续的执行过程中发生函数调用或返回引发的流程转移时，将首先查找一级代码缓存，若匹配成功则执行优化的翻译代码。

为了防止热例程陷入无限循环而影响对外部中断的响应，ARCH-BRIDGE 记录热例程中的全部类型 II 出口(循环

一定通过类型 II 出口建立块链),当周期性定时器超时时,如果当前正在某个热例程中执行,则在定时器超时处理程序中将该热例程的所有由类型 II 出口建立的块链断开,以便进入 I/O 子系统检测中断,并在中断检测结束后恢复被断开的块链。

为支持热例程的淘汰,ARCH-BRIDGE 为每个热例程设置了一个老化计数器 AGE。DBT 引擎每次在一级代码缓存中命中一个热例程的翻译代码时,将 AGE 清 0。另设一个专门的线程定期扫描热例程列表并将 AGE 增 1,当 AGE 超过预设的阈值时可认为热例程已很少使用(如相应进程退出),将热例程翻译代码淘汰。

6 测试

本文对基于热例程的系统级二进制翻译优化的效果进行了测试,并与以 Trace 为单位的优化效果进行了比较。主要测试环境如下:虚拟机监控器为 ARCH-BRIDGE,宿主机为运行中标麒麟操作系统的 SW-410 平台,虚拟机操作系统为经过服务裁剪的 tty-linux(内核版本为 2.6.38),测试程序集为 SPEC CPUINT 2006。由于完整运行 SPEC 程序需要数天的时间,在虚拟机环境下进行测试的时间还要更长,因此选取了 SPEC 测试集中耗时较短的 TEST 类型用例进行测试。

表 2 给出了分别采用 Trace 和热例程为优化单位时运行各个 SPEC 程序所采集到的优化单位数量和尺寸,以及利用插桩方法统计得到的运行时优化单位内部的块占总执行块数的比例。显然,热例程中的每个块都属于某个 EBB,除了基本块个数为 1 的 EBB 外,其它 EBB 中的基本块都有机会得到优化,同理,长度为 1 个基本块的 Trace 的优化价值也不大,因此在统计优化单位的尺寸时,排除了基本块个数为 1 的 EBB 以及长度为 1 个基本块的 Trace。

表 2 以 Trace 和热例程为优化单位时各 SPEC 程序的优化单位数量、尺寸及运行时优化单位内部的块占总执行块数的比例

测试程序	优化单位数量		平均优化单位尺寸		块处于优化单位内部的几率	
	Trace	HR	Trace	HR	Trace(kernel/user)	HR(kernel/user)
400. perlbench	2116	1622	3.82	13.49	63.3%/47.3%	84.0%/75.1%
401. bzip2	484	219	2.85	7.42	62.7%/76.0%	75.8%/11.4%
403. gcc	5110	3291	4.30	14.66	62.4%/53.1%	74.8%/78.9%
429. mcf	401	428	2.98	8.73	63.3%/65.1%	75.0%/69.7%
456. hmmer	321	248	2.54	8.16	62.9%/56.4%	75.6%/31.4%
458. sjeng	608	403	3.03	12.77	64.7%/48.0%	75.0%/85.3%
462. libquantum	122	194	2.75	7.37	59.6%/78.2%	74.9%/30.2%
464. h264. ref	1103	503	2.80	9.44	65.3%/58.7%	75.8%/22.9%

在优化单位数量方面,大多数情况下运行 SPEC 程序过程中产生的热例程数量比 Trace 数量要少,在平均优化单位尺寸方面,热例程代码区域中的块的数量普遍比 Trace 要多,说明热例程方法具有更好的优化效率。在块处于优化单位内部的几率方面,内核代码处于热例程区域的几率普遍比处于 Trace 区域的几率要大,某些情况下,用户代码处于热例程区域的几率比处于 Trace 区域的几率小。导致该结果的原因在于,内核代码主要以例程调用方式被激活,且在例程内部很少有大量的循环,即使例程没有被识别为热例程,所损失的优化机会也不大;而用户代码可能存在一些很少被调用的但其中包含大循环的例程,从而在热例程发现阶段被遗漏,导致优化机会的损失。以 401. bzip2 为例,在其 main 函数中存在一个入口地址为 0x080492D0 的大小为 6126 次的循环,但由于 main 仅被调用一次,因此其无法被热例程方式所识别。从这个角度讲,热例程方法更适合于内核代码的优化。

表 3 给出了 ARCH-BRIDGE 在未采用优化机制时(AB)以及在对内核代码分别采用 Trace(AB-Trace)和热例程(AB-HR)进行优化时运行各个 SPEC 程序的时间;图 11 给出了相比 AB 而言 AB-Trace 和 AB-HR 运行各个 SPEC 程序的加速比。结果显示,采用 Trace 机制优化(AB-Trace)后的 SPEC 程序的性能提升了 2.0%～8.7%,采用热例程机制优化(AB-HR)后的 SPEC 程序的性能提升了 3.5%～14.4%。相比 AB-Trace,AB-HR 的最大性能提升达到了 5.1%。

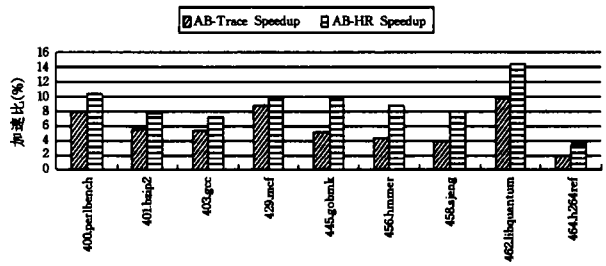

图 11 AB-Trace 和 AB-HR 运行各 SPEC 程序的加速比

表 3 AB、AB-Trace 和 AB-HR 的 SPEC 程序的运行时间(s)

程序	测试命令	AB	AB-Trace	AB-HR
400. perlbench	perlbench_base. i386-m32-gcc42-nn test. pl	327	301	293
401. bzip2	bzip2_base. i386-m32-gcc42-nn dryer. jpg 2	1940	1833	1790
403. gcc	gcc_base. i386-m32-gcc42-nn cccp. i -o cccp. s	732	693	679
429. mcf	mcf_base. i386-m32-gcc42-nn inp. in	665	607	599
445. gobmk	gobmk_base. i386-m32-gcc42-nn --mode gtp --gtp-input connect. tst	484	459	437
456. hmmer	hmmer_base. i386-m32-gcc42-nn --fixed 0 --mean 325 --num 45000 --sd 200 --seed 0 bombesin. hmm	1556	1488	1421
458. sjeng	sjeng_base. i386-m32-gcc42-nn test. txt	2025	1942	1864
462. libquantum	libquantum_base. i386-m32-gcc42-nn 66 5	256	231	219
464. h264ref	h264ref_ base. i386-m32-gcc42-nn -d foreman_test_encoder_baseline. cfg	15471	15150	14929

(下转第 41 页)

[9] Intel Corporation. Intel® Xeon® Processor E5-1600/E5-2600/E5-46 00 Product Families Datasheet Volume One [EB/OL]. [2015-03-05]. http://www.intel.com/products/processor%5Fnumber/

[10] Intel Corporation. An Introduction to the Intel QuickPath Interconnect[EB/OL]. [2009-01-30]. http://www.intel.com

[11] 王恩东,等. MIC 高性能计算编程指南[M]. 北京:中国水利水电出版社,2012

[12] Jeffers J, Reinders J. Intel Xeon Phi coprocessor high performance programming[M]. Newnes, 2013

[13] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume2(2A, 2B & 2C): Instruction Set Reference, A-Z [EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[14] Intel Corporation. Intel® C++ Compiler User and Reference Guides [EB/OL]. [2015-03-05]. http://www.intel.com

[15] Free Software Foundation. Inc. GCC 4.9 Release Series[EB/OL]. [2014-07-16]. http://gcc.gnu.org/gcc-4.9/

[16] Manchanda N, Anand K. Non-Uniform Memory Access(NUMA)[OL]. http://cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf

[17] Intel Corporation. Intel® 6 4 and IA-32 Architectures Software Developer's Manual Volume 3(3A, 3B & 3C): System Programming Guide[EB/OL]. [2015-03-05]. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[18] Feng Q Y. Research on Data Prefetching Techniques for Loop-Level Array References[D]. Changsha: National University of Defense Technology, 2008(in Chinese)
冯权友. 面向循环级数组访问的数据预取技术研究[D]. 长沙:国防科学技术大学,2008

[19] Igor Ostrovsky Blogging. Gallery of Processor Cache Effects [EB/OL]. http://igoro.com/archive/gallery-of-processor-cache-effects

(上接第 33 页)

结束语 本文提出了一种基于热例程的系统级动态二进制翻译优化方法,该方法以频繁执行的例程作为优化单位,分别通过虚拟寄存器机制和扩展基本块划分来消除动态二进制翻译引入的块内和块间冗余。相比基于踪迹的优化方法而言,该方法具有优化单位发现开销更小、代码区域更大、无重复优化翻译等优点,更适用于系统虚拟机中操作系统代码的优化。在跨平台系统虚拟机 ARCH-BRIDGE 平台上的测试表明,通过对内核代码实施该优化方法,SPEC CPUINT 2006 程序的效率提升了 3.5%~14.4%,相比基于踪迹的优化,性能最大提升了 5.1%。

参 考 文 献

[1] Chen Wei. Research on Dynamic Binary Translation based Co-Designed Virtual Machine[D]. National University of Defense Technology, 2010

[2] Hu W, Wang J, Gao X. Godson-3: A scalable multicore RISC processor with X86 emulation[J]. Micro, IEEE, 2009, 29(2): 17-29

[3] Heng Yin, Song D. TEMU-Binary Code Analysis via Whole-System Layered Annotative Execution[R]. Berkeley: UC Berkeley, 2010

[4] Wang Rong-hua. Research on Dyanmic Binary Translation Optimization[D]. Hangzhou: Zhejiang University, 2013(in Chinese)
王荣华. 动态二进制翻译优化研究[D]. 杭州:浙江大学,2013

[5] Slechta B, Crowe D. Dynamic optimization of micro-operations [C]//Proceedings. The Ninth International Symposium on High Performance Computer Architecture, 2003 (HPCA-9 2003). IEEE, 2003: 165-176

[6] Bellard F. QEMU, a fast and portable dynamic translator[C]// USENIX annual technical conference, FREENIX Track. 2005: 41-46

[7] Hong D Y, Hsu C C, Yew P C. HQEMU, a multi-threaded and retargetable dynamic binary translator on multicore[C]//Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, 2012: 104-113

[8] Cao Hong-jia, Tang Yu-xing, Zhou Xing-ming. Parallel Dynamic Binary Translation and its Cache Maintanance[C]//Proceedings of National Conference on Information Storage Technology. Xi'an, 2004(in Chinese)
曹宏嘉,唐遇星,周兴铭. 并行动态二进制翻译及其缓存维护[C]//全国信息存储技术学术会议论文集. 西安,2004

[9] Dehnert J C, Grant B K, Banning J P, et al. The Transmeta Code Morphing? Software: using speculation, recovery, and adaptive retranslation to address real-life challenges[C]//Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and Runtime Optimization. IEEE Computer Society, 2003: 15-24

[10] Ebcioglu K, Altman E, Gschwind M, et al. Dynamic binary translation and optimization[J]. IEEE Transactions on Computers, 2001, 50(6): 529-548

[11] Bala V, Duesterwald E, Banerjia S. Dynamo: a transparent dynamic optimization system[J]. ACM SIGPLAN Notices, ACM, 2000, 35(5): 1-12

[12] Hsu C C, Liu P, Wu J J, et al. Improving dynamic binary optimization through early-exit guided code region formation[C]// Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, ACM, 2013: 23-32

[13] Huang Cong-hui, Chen Jing, Gong Shui-qing, et al. Research of Method for Virtualizing 64-bit Windows Application Binary Interface[J]. Computer Science, 2014, 41(1): 39-42(in Chinese)
黄聪会,陈靖,龚水清,等. 64 位 Windows ABI 虚拟化方法研究[J]. 计算机科学,2014,41(1):39-42

[14] Duesterwald E, Bala V. Software profiling for hot path prediction: Less is more[J]. ACM SIGOPS Operating Systems Review, ACM, 2000, 34(5): 202-211