译码制导的动态二进制翻译优化

董卫宇 王瑞敏 戚旭衍 曾 韵
(数学工程与先进计算国家重点实验室 郑州 450000)

摘 要 提出了一种译码制导的轻量级动态二进制翻译优化技术,该技术在译码阶段提取源指令的高层语义信息,结合上下文对其进行标注,并在翻译阶段利用标注信息直接生成优化的目标指令。该技术可识别动态二进制翻译系统中主要的基本块级优化机会,去除load/store冗余、精确异常导致的冗余和标志位处理冗余。测试表明,相比QEMU,该优化技术的跨平台x86系统虚拟机ARCH-BRIDGE的翻译开销降低了53%,翻译块尺寸降低了78%,load和store操作数量分别了降低了50%和21%。

关键词 动态二进制翻译,系统虚拟机,软件透明移植,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.2015.6.041

Decoding-directed Dynamic Binary Translation Optimization

DONG Wei-yu WANG Rui-min QI Xu-yan ZENG Yun
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China)

Abstract The paper introduced a decoding-directed lightweight optimization technique for dynamic binary translation. In decoding phase, it extracts high-level semantics from source instructions, attaches appropriate annotations to them according to the context, and in translation phase, it emits optimized local instruction directly using the annotation information. The technique may identify most block-level optimization opportunities of dynamic binary translation system, and remove redundancies generated by load/store, precise exception supporting and flags handling. Evaluation demonstrates that taking QEMU for reference, the translation overhead of cross-platform x86 system virtual machine ARCH-BRIDGE using above technique gets a decrease of 53%, while the translation block size decreases by 78%, and the load/store operation number deceases by 50% and 21% respectively.

Keywords Dynamic binary translation, System VM, Software transparent porting, SW processor

动态二进制翻译(Dynamic Binary Translation, DBT)是一种灵活的、在运行时控制和修改程序的机制,是跨平台二进制代码迁移、系统仿真、动态优化、逆向工程、安全检测等领域的使能技术,有着广泛的应用和重要的研究价值。

DBT系统按需发现后续将要执行的源机器代码片段,将其翻译为本地机器代码,并付诸执行。相比代码的本地运行,利用DBT技术运行程序具有一定的性能开销,主要表现在两个方面:1)启动开销,即必须将源机器指令转换为等价的本地指令才能执行,从而导致程序启动上的延迟,影响程序的响应性;2)稳定运行开销,即由于翻译代码质量不高或本地硬件资源缺失等原因,影响程序的运行速度。

典型的DBT系统多采用多阶段模拟(Staged Emulation)和基于中间表示(Intermediate Representation, IR)的翻译优化技术来改善程序启动速度和稳定运行速度。例如,DAISY[1]、DBT86[2]、Dynamo[3]等系统采用解释技术负责代码的初始执行,直到确认代码为热点后才转入翻译执行阶段;DynamoRIO[4]、CrossBit[5,6]、QEMU[7]等系统首先将源机器代码翻译为IR,之后对IR序列进行优化,最后将IR翻译为本地机器指令。多阶段模拟的缺点是需要同时支持指令的解释和翻译,增加了工程量和系统的复杂度;基于IR的翻译优化的缺点是增加了翻译环节,反而制约了程序的启动速度。FastBT[8]、HDTrans[9]等系统采用表驱动(Table-Driven)的翻译方法,以表的形式组织指令的译码信息和翻译程序,以指令操作码作为表索引,具有较快的启动速度,但两者均是同构平台上的DBT系统,且仅对流程转移指令进行了优化。

本文探讨了在采用表驱动翻译方法的情况下对翻译块进行优化的效果,提出了一种译码制导的轻量级动态二进制翻译优化技术,并在跨平台系统级虚拟机ARCH-BRIDGE上进行了验证。测试表明,相比QEMU,采用该优化技术的跨平台x86系统虚拟机ARCH-BRIDGE的翻译开销降低了53%,翻译块尺寸降低了78%,load和store操作数量分别降低了50%和21%。

本文第1节介绍了跨平台系统虚拟机ARCH-BRIDGE及其DBT机制;第2节识别了动态二进制翻译系统中的主要

到稿日期:2014-07-10 返修日期:2014-12-22

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:xinbaoer.dong@gmail.com;王瑞敏(1982-),女,硕士,讲师,主要研究方向为体系结构;戚旭衍(1984-),女,硕士,讲师,主要研究方向为系统虚拟化;曾 韵(1976-),女,硕士,讲师,主要研究方向为操作系统。

优化机会；第 3 节给出了译码制导的动态二进制翻译优化方法；第 4 节对该方法进行了测试和结果分析；最后对本文工作进行了总结。

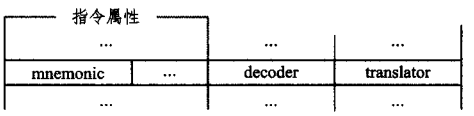
1 ARCH-BRIDGE 及其 DBT 机制

ARCH-BRIDGE 是基于申威处理器的 x86 系统虚拟机监控器，其目的是研究申威与 x86 间的指令集架构差异，识别基于申威平台的 x86 系统虚拟机的性能瓶颈，提出面向 x86 架构兼容的申威处理器优化扩展方案，并在后续硬件支持的基础上最终演化为一款软硬件协同设计虚拟机监控器[10,11]（Co-designed Virtual Machine Monitor），实现 x86 操作系统和应用程序在申威平台上的透明高效运行。相比进程级虚拟机[12]，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。

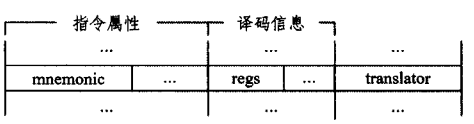
目前，ARCH-BRIDGE 可运行于申威、ALPHA、OPENRISC 等处理器平台，支持 Intel P6 处理器定义的全部定点指令、FPU 指令和 MMX 指令，支持 PCI 总线、南北桥、IDE、VGA、APIC 等典型 x86 接口，能够运行基于 x86 Linux 2.6 内核的操作系统，能够运行 BusyBox 工具集中的全部应用程序，通过了 SPEC CPU 2006 测试集中全部定点程序和大部分浮点程序的测试。

为提升程序的启动效率，ARCH-BRIDGE 采用表驱动翻译方法，如图 1(a)所示，所有 x86 指令的信息在逻辑上被组织成以操作码为索引的表，称为指令信息表。表的每一行提供 3 类信息，分别是指令的属性、译码方法入口和翻译方法入口，其中属性部分记录了仅凭操作码即可确定的指令信息，包括助记符、是否含有 mod/rm 字段、是否含有立即操作数、可否作为基本块结尾、是否影响标志位等，这些信息可作为译码和翻译方法的参数，指导指令译码和翻译块生成。

译码制导的 DBT 优化方法分为译码和翻译两个阶段。在译码阶段，DBT 引擎逐条对源机器指令流中的指令进行译码，直到遇到满足基本块结束条件的指令（如 JMP/Jcc/CALL 等）为止。根据指令的操作码确定指令信息表的索引[1)]，以此确定该指令的属性和译码方法 decoder，并以属性为参数调用 decoder 进行译码。在译码过程中进一步提取指令的信息，包括指令引用和修改的寄存器、是否可能产生异常等，这些信息连同指令信息表中的相关信息一起被记录在如图 1(b)所示的指令译码序列中，以指导后续的优化和翻译工作。具体机制将在第 2 节中介绍。



(a)指令信息表组织



(b)指令译码序列

图 1

DBT 引擎定义了一套接口原语（以 gen_prim_为前缀命名），各个指令的翻译方法 translator 仅调用若干接口原语 gen_prim_xxx 来翻译 x86 指令，而 gen_prim_xxx 的具体实现依赖于宿主平台。采用接口原语使得 ARCH-BRIDGE 在不引入中间表示的情况下支持多种宿主平台，并且具有更快的翻译速度。对于某些具有复杂操作的 x86 指令（如 INT 等），DBT 引擎利用辅助的 C 代码进行仿真，并在翻译时生成对 C 代码的调用指令。

在每条影响标志位的 x86 指令末尾精确计算标志位信息非常耗时，ARCH-BRIDGE 采用标志位延迟计算方法，在翻译时插入 store 指令将 x86 指令的操作、源操作数和执行结果（目的操作数可由上面 3 个信息恢复出来）保存到全局变量 cc_op/cc_src/cc_res 中，后续指令仅在必要时才利用 cc_op/cc_src/cc_res 恢复出标志位的值。

翻译基本块后得到的本地指令序列称为翻译块（Translated Block，TB）。为提高重用率，翻译得到的 TB 将被保存在代码缓存（Code Cache）中。为减少 DBT 引擎与翻译块间的切换次数，采用了翻译块链（Block Chaining）机制，若刚翻译的块与上次执行的块间存在直接跳转关系，则在两者之间建立链接。

在完成译码、翻译、代码缓存管理、翻译块链接等工作后，DBT 引擎将执行翻译块。由于 DBT 引擎与翻译块之间共用一套宿主机结构寄存器，在执行翻译代码之前，还需要首先执行一段前导代码（prologue），按照宿主平台寄存器使用约定将 DBT 引擎正在使用的部分寄存器保存到堆栈中，并在翻译块执行完毕后执行一段收尾代码（epilogue），从堆栈中将此前保存的寄存器状态恢复回来。若在翻译块执行中检测到 x86 异常，将采用 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-BRIDGE 的 DBT 引擎、翻译块，以及辅助 C 代码间的交互关系如图 2 所示，图中虚线表示检测到异常后的控制转移。

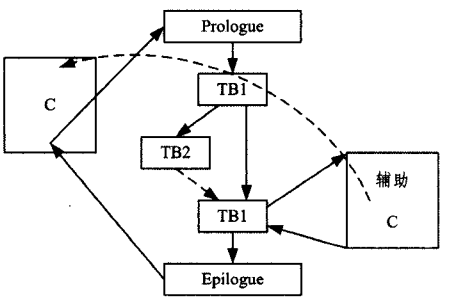


图 2 DBT 引擎、TB、辅助 C 代码间的交互

2 DBT 优化机会识别

以申威宿主平台为例，在未采用优化机制的情况下，图 3 左侧的 x86 基本块将被翻译为图 3 中部的申威指令，图 3 右侧给出了各个申威指令的简单注释。ARCH-BRIDGE 利用一块名为 cpu_state 的内存区域模拟 x86 寄存器以及 DBT 工作所需的关键数据（如标志位相关信息 cc_op/cc_src/cc_res、辅助 C 函数入口地址等），该内存区域的访问频次很高，故使用 gcc 的全局寄存器变量特性，令宿主寄存器 r15 指向 cpu_state 的基址，再辅以偏移量来访问其中的数据。在图 3 中，此类位移量以 o_为前缀。例如 o_eax 表示 eax 寄存器在 cpu_state 区域的偏移。简单起见，图 3 中略掉了翻译 movl 指令

1) 对于具有多字节操作码或扩展操作码的 x86 指令，需要对操作码进行一些转换后得到指令信息表的索引。

时生成的段界限检查代码。

```
1   orl %eax, %edx          ldw r10, o_eax(r15)      ;加载eax
2                           ldw r9, o_edx(r15)       ;加载edx
3                           bis r10, r9, r9
4                           stw r9, o_cc_res(r15)    ;保存计算结果（冗余）
5                           bis r31, 22, r1          ;
6                           stb r1, o_cc_op(r15)     ;保存影响标志位的操作（冗余）
7                           stw r9, o_edx(r15)       ;保存edx（冗余）
-------------------------------------------------------------------
8   addl %ecx, %edx         ldw r10, o_ecx(r15)      ;加载ecx
9                           ldw r9, o_edx(r15)       ;加载edx（冗余）
10                          addl r10, r9, r9
11                          stw r10, o_cc_src(r15)   ;保存源操作数（可能冗余）
12                          stw r9, o_cc_res(r15)    ;保存计算结果（可能冗余）
13                          bis r31, 10, r1          ;
14                          stb r1, o_cc_op(r15)     ;保存影响标志位的操作（可能冗余）
15                          stw r9, o_edx(r15)       ;保存edx（可能冗余）
-------------------------------------------------------------------
16  movl (%edx), %eax       ldw r9, o_edx            ;加载edx（冗余）
17                          ldw r1, o_ds_base(r15)   ;加载DS段基址
18                          addl r9, r1, r9          ;计算有效地址EA
19                          bis r31, r9, r16         ;将EA放入参数寄存器r16
20                          ldi r17, 4(r31)          ;将操作宽度（4）放入参数寄存器r17
21                          ldl r27, o_mmu_rd(r15)   ;得到C函数mmu_rd的入口
22                          call r26, r27            ;调用mmu_rd读取内存
23                          stw r0, o_eax(r15)       ;读到的内存内容存入eax
-------------------------------------------------------------------
24  subl $32, %edx          addw r31, 32, r10        ;加载立即数
25                          ldw r9, o_edx(r15)       ;加载edx（冗余）
26                          subl r9, r10, r9
27                          stw r10, o_cc_src(r15)   ;保存源操作数
28                          stw r9, o_cc_res(r15)    ;保存计算结果
29                          bis r31, 16, r1
30                          stb r1, o_cc_op(r15)     ;保存影响标志位的操作
31                          stw r9, o_edx(r15)       ;保存edx
-------------------------------------------------------------------
32  jc    some_label        ...
```

图 3　未优化的 x86 到申威指令翻译

以图 3 所示代码为例，若在不考虑指令上下文的情况下进行翻译，将至少产生如下几类冗余指令：① load 冗余。ARCH-BRIDGE 利用内存模拟 x86 寄存器，在不考虑指令相关性的情况下，即使后续 x86 指令使用了相同的寄存器，也要重新装入。例如，orl、andl、movl、subl 指令均使用 edx，第 9、16、25 行的 load 操作是冗余的。② store 冗余。采用简单翻译策略时，若 x86 指令修改寄存器，在翻译时将生成 store 操作保存寄存器的内容，但由于 orl 和 addl 都修改了 edx，因此第 7 行的 store 操作是冗余的。③ 由精确异常导致的冗余。x86 处理器支持精确异常，保守的翻译策略在每条指令结尾处提交对虚拟机的状态修改，但如果第 16 行的访存指令 movl 未引发异常，则第 11－15 行都是冗余的。④ 标志位处理冗余，例如，由于 addl 指令影响全部标志位，4－6 行 orl 指令的标志位处理是冗余的。

源指令流一般经过编译优化，基本块大小也有限，因此存在的优化机会不多。但受指令集架构差异的影响，源指令在经过二进制翻译后往往会引入冗余。虽然可以利用传统的编译优化方法对翻译块进行优化（例如，标志位冗余可看做定值引用问题，load/store 冗余可看作复写传播问题，都可以利用数据流分析方法优化），但大多数系统在中间表示一级进行优化，增加了翻译的阶段，且数据流分析算法开销很大。

本文提出了一种译码制导的 DBT 优化技术，主要思路是利用译码阶段提供的更高层语义信息指导翻译，在译码阶段提取 x86 指令的相关信息，并结合上下文对 x86 指令进行标注，在翻译阶段利用标注信息直接生成优化的本地指令。该技术只需两个阶段即可生成优化的目标代码，去除由 DBT 引入的典型冗余，相比其他 DBT 系统先产生中间表示 IR、再删除冗余、再生成本地指令的三阶段做法，具有更高的效率。

3　译码制导的 DBT 优化方法

本节重点说明如何利用译码制导的 DBT 优化方法消除第 2 节介绍的前 3 种冗余（load/store 冗余、精确异常冗余），该方法同样适用于消除标志位处理冗余。

3.1　译码及标注算法

为消除 load/store 冗余和精确异常冗余，ARCH-BRIDGE 支持如下与优化翻译相关的指令标注信息：①regs_read/regs_write，分别表示指令所读取和写入的寄存器集合；②gen_excpt，指令是否有可能触发异常。上述信息可以在对单条指令译码的过程中得到。此外，还在基本块范围内维护各个 x86 寄存器的活跃区间信息（Live Range）。一个 x86 寄存器的活跃区间定义为：基本块内首次使用（读或写）该寄存器的指令序号到最后一次使用该寄存器的指令序号所构成的闭区间，以 $LR=[start,end]$ 表示，该信息可在对基本块译码的过程中迭代获得。

译码及标注算法 decode_annotate 的类 C 语言描述如图 4 所示。设译码标注后的 x86 指令信息序列为 insns，序列中元素类型为 i_info_t。设 x86 寄存器的活跃区间信息为数组 lr[8]，各个元素类型为 lr_t。

```
void decode_annotate () {
  i_info_t insn;
  int i=0;
  do {
    // 译码一条 x86 指令，解析出信息并保存
    insns[i] insn decode_insn();
    for (集合 insn. regs_read ∪ insn. regs_write 中的每个 x86 寄存器 r) {
      // 更新 x86 寄存器活跃区间
      if (lr[r]. start==-1)
        lr[r]. start=i;
      lr[r]. end=i;
      }
    ++i;
  } while (insn 不是基本块的结尾);
}
```

图 4　decode_annotate 算法描述

decode_annotate 算法中，decode_insn 负责从指令流中取出下一条 x86 指令，解析出其相关信息并保存之后，更新各个 x86 寄存器的活跃区间。

3.2　翻译算法

翻译算法 translate 对基本块中的每条指令实施优化翻译，它利用译码阶段对指令的标注结果，尽量延迟机器状态的提交，从而使冗余的操作得到合并。翻译过程中在基本块范围内维护全局信息 x86_regs_uncmted，表示已修改但尚未提交到 cpu_state 区域的 x86 寄存器的集合，初值为空。translate 的类 C 语言描述如图 5 所示。

```
void translate() {
  int pos=0;
  for (译码信息序列 insns[]中的下条指令 insn) {
    for (集合 insn. regs_read ∪ insn. regs_write 中的每个 x86 寄存器 r) {
      if (lr[r]. start==pos) {
        为 r 分配宿主寄存器 hr;
        if (r ∈ insn. regs_read || 指令 insn 的操作宽度非 32 位)
          生成 load 指令将 r 的内容装入 hr;
      }
    }
    if (insn. gen_excpt) {
      for (集合 x86_regs_uncmted 中的每个 x86 寄存器 r)
```

```
        生成 store 指令保存 r 的内容到 cpu_state 区域;
    x86_regs_uncmted Φ;
      }
    调用 insn. translator()产生本地指令模拟 x86 指令的各种操作;
    x86_regs_uncmted x86_regs_uncmted ∪ insn. regs_write;
    for (每个 x86 寄存器 r) {
      if (lr[r]. end==pos) {
        if (r ∈ x86_regs_uncmted) {
          生成 store 指令提交 r 的状态;
            x86_regs_uncmted x86_regs_uncmted − {r};
          }
            释放 r 对应的本地寄存器;
      }
        }
        if (insn 是基本块的结尾)
            break;
    pos++;
  }
}
```

图 5　translate 算法描述

对基本块中的每条指令，翻译算法的主要工作是：①对首次出现的 x86 寄存器，为其分配宿主寄存器，并在必要时加载其内容；(注意，对于某些操作宽度为 8 位或 16 位的只写操作(如 mov)，为防止寄存器的高位被破坏，也要首先加载其内容。)②如果指令可能引发异常，提交已修改的 x86 寄存器；③调用指令自身的翻译函数 translator()产生本地指令，模拟 x86 指令的各种操作，此时指令所使用的 x86 寄存器已有对应的宿主寄存器，只需根据指令语义分配翻译过程中所需的临时寄存器即可，临时寄存器的生命周期不会跨越指令边界；④依据指令所修改的寄存器，更新 x86_regs_uncmted；⑤对于活跃区间已结束的 x86 寄存器，释放其对应的宿主寄存器，若寄存器已修改，则提交其内容。

在结构寄存器丰富的 RISC 处理器上实现 x86 系统虚拟机，预留 8 个本地寄存器专用于映射 x86 寄存器是可以接受的。但受到处理器 ABI(Application Binary Interface)的限制，Callee-saved 寄存器(由被调函数负责保存寄存器)数量可能不足 8 个(如申威处理器仅 $9～$14 属于 Callee-saved 寄存器)，当非 Callee-saved 寄存器被分配时，若在 TB 中调用辅助 C 函数(如大量存在的访存操作)，需首先将这类寄存器溢出(Spill)到堆栈中，并在之后从堆栈中恢复内容，这种开销是很大的，这也是在 translate 算法中进行本地寄存器动态分配和释放的原因。实际实现中，预留了 4 个 Callee-saved 寄存器和 4 个非 Callee-saved 寄存器供 x86 寄存器使用(其余 2 个 Callee-saved 寄存器用于指令翻译过程中所需的临时寄存器)，并总是优先为 x86 寄存器分配 Callee-saved 寄存器。实际工作中，基本块中同时引用的 x86 寄存器很少超过 4 个，因此少有寄存器溢出情况。

在可能引发异常的指令执行前，无条件地提交已修改的 x86 寄存器是不必要的，这类操作可以延迟到异常发生时再进行。在实现中，本文针对大量存在的访存操作进行了优化翻译。为此，ARCH-BRIDGE 以快速和慢速两种方式实现访存操作，在快速方式下，由指令片段 *mmu_rd_fast* 和 *mmu_wt_fast* 负责 TLB 查询以及 TLB 命中后的宿主存储器访问，其数量为 20 左右，慢速方式下的访存操作以辅助 C 函数 *mmu_rd* 和 *mmu_wt* 实现。在翻译访存指令时，首先生成指令调用快速访存指令片段，之后再生成指令调用慢速访存函数，仅当 TLB 未命中时，慢速访存方式才会得到执行。由于 *mmu_rd_fast* 和 *mmu_wt_fast* 的任务简单(不访问页表)，固定使用参数传递寄存器 r16～r21 即可完成工作，无需考虑寄存器切换，效率较高。实际测试中 TLB 命中率可达 99.4%以上，因此绝大部分访存操作可以快速方式完成，且不会引发异常。

在上述机制下，改进已修改寄存器的提交方式，将图 5 中黑体部分的操作延迟到 *mmu_rd_fast* 和 *mmu_wt_fast* 之后进行，从而可以去除大量不必要的精确异常冗余。以 8 位读内存操作原语 gen_prim_ld_m8u 为例，改进后的访存类指令翻译策略如图 6 所示。*mmu_rd_fast* 以 r16 为返回值，r16 为 1 说明访存成功执行；否则才提交 x86 寄存器状态，并调用 *mmu_rd*。

```
void gen_prim_ld_m8u(…)
{
  生成指令调用 mmu_rd_fast;
  生成分支指令 blbs,若 r16 为 1 转结束;
    for (x86_regs_uncmted 中每个寄存器 r)
      生成 store 指令提交 r 的内容;
  生成指令调用 mmu_rd;
}
```

图 6　访存指令翻译策略

4　测试及分析

本文的测试环境如下：宿主机为运行中标麒麟操作系统的 SW-410 平台，虚拟机监控器为 ARCH-BRIDGE，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)。为叙述方便，分别将优化前后的 ARCH-BRIDGE 系统简称为 AB 和 AB-OPT。另外，本文以 QEMU 作为参考系统，利用 QEMU 0.10 提供的 TCG 机制，将其移植到 SW-410 平台，简称为 QEMU-SW。

本文应用译码制导的动态二进制翻译优化技术，对 ARCH-BRIDGE 的翻译机制进行改造。由于调试一款系统级虚拟机并使其稳定运行需要很长的工作周期，本文采用了踪迹驱动(trace-driven)的测试方法，即通过可以稳定运行的 AB 来采集虚拟机的执行踪迹(trace)，并将该 trace 信息送入 AB-OPT，以提取翻译过程中的主要指标信息。有关 QEMU-SW 的性能数据可以利用 QEMU 提供的 monitor 机制得到。

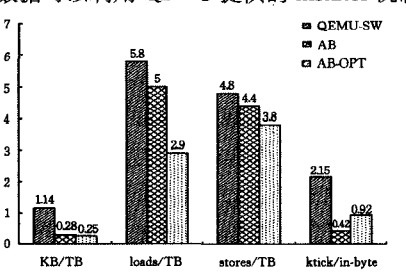


图 7　QEMU-SW、AB、AB-OPT 指标数据对比

本文测试的有关动态二进制翻译的一些指标数据包括：引导过程中的各个基本块对应的翻译块的大小(tb-size)、平均每个 x86 指令字节所占用的翻译时间周期数(ticks/in-byte)、各个翻译块中由于读取或保存 x86 寄存器所产生的 load/store 数量(loads/stores)等，测试结果如图 7 所示。

(下转第 203 页)

Indiana, USA: ACM, 2010: 663-674

[14] Cormen T H, Leiserson C E, Rivest R L, et al. Introduction to Algorithms[M]. Cambridge MA, USA: the MIT Press, 2009: 65

[15] Gunopulos D, Kollios G, Tsotras V, et al. Approximating multi-dimensional aggregate range queries over real attributes[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Dallas, Texas, USA: ACM, 2000: 463-474

[16] Aboulnaga A, Chaudhuri S. Self-tuning histograms: building histograms without looking at data[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Philadelphia, Pennsylvania, USA: ACM, 1999: 181-192

[17] Robinson J T. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes[C]// Proceedings of the ACM SIGMOD International Conference on Management of Data. Ann Arbor, Michigan, USA: ACM, 1981: 10-18

[18] Nievergelt J, Hinterberger H, Sevcik K C. The grid file: An adaptable, symmetric multikey file structure[J]. ACM Transactions on Database Systems, 1984, 9(1): 38-71

[19] Finkel R A, Bentley J L. Quad trees a data structure for retrieval on composite keys[J]. Acta Informatica, 1974, 4(1): 1-9

(上接第 192 页)

QEMU-SW 将每个 x86 访存操作对应的翻译代码内联到翻译块中，且在调用辅助 C 函数时有很多的 Spill 操作，因此 TB 的尺寸较大，平均每个 TB 达到 1144 字节(代码膨胀率为 67.4)。AB 和 AB-OPT 以辅助 C 函数的方式实现访存操作，并且特别规划了 Callee-saved 寄存器的使用，因此翻译块的平均大小分别为 279 字节和 255 字节，代码膨胀率降低到了 17.3 和 15.8。相比 AB 而言，虽然 AB-OPT 的代码膨胀率降低不多，但由于其中存在一些不会执行的补偿代码(主要是 store 操作)，因此实际代码膨胀率还要低一些。

本文在测试中仅考虑由于读取或保存 x86 寄存器所产生的 load/store 操作，访问虚拟机内存所引起的 load/store 操作不会被优化，因此不予考虑。QEMU-SW 孤立地将每条 x86 指令翻译为中间表示，即使后续指令使用了相同的寄存器，也要重新生成中间表示来将其由内存装入虚拟寄存器。另外，为支持精确异常，还需在每条源指令的翻译块的末尾生成 store 指令来将结果保存到源寄存器中。因此，QEMU-SW 翻译代码中的 load/store 数量较大，经过其自身的活跃性分析优化后，平均每个 TB 中的 load/store 数量分别为 5.8 和4.8。AB 的做法与 QEMU 类似，平均每个翻译块中的 load/store 数量为 5 和 4.4。由于 AB 对段级存储管理进行了优化，在计算访存地址时，若段基址为 0，则忽略其影响(测试表明，97.3%的访存指令符合此条件)，免去了从 cpu_state 区域装入段基址的操作，因此 AB 的 load 数量少于 QEMU。AB-OPT 中，平均每个翻译块中的 load/store 数量为 2.9 和 4.4，其中 store 的数量与 AB 基本持平，但其中一部分为补偿代码，并不真正执行，以 TLB 命中率为 99.4%进行计算，每个 TB 中的实际执行的 store 数量为 3.8，相比 QEMU、load/store 的数量分别降低了约 50%和 21%。

由于 QEMU-SW 采用三阶段翻译机制，因此翻译开销较大，每个 x86 指令字节的翻译时间(cycles/in-byte)为 2153 TICKS；AB 的翻译时间最短，其 cycles/in-byte 为 424 TICKS，但翻译代码质量不高；由于译码标注及优化翻译会引入一定的开销，因此 AB-OPT 的翻译时间有所增加，其 cycles/in-byte 为 920 TICKS，在翻译代码质量优于 QEMU-SW 的情况下，翻译开销降低了约 57%。

结束语 本文提出了一种译码制导的轻量级动态二进制翻译优化技术，在译码阶段提取源指令的高层语义信息，结合上下文对其进行标注，并在翻译阶段利用标注信息直接生成优化的目标指令。该技术可在不影响系统移植性的情况下，利用译码和翻译两个阶段即可识别动态二进制翻译系统中主要的基本块级优化机会，生成优化的本地代码，在降低动态二进制翻译开销的同时，提升翻译代码的质量。测试表明，相比 QEMU，采用该优化技术的 ARCH-BRIDGE 跨平台 x86 系统虚拟机的翻译开销降低了 53%，翻译块尺寸降低了 78%，load 和 store 操作数量分别了降低了 50%和 21%。

参 考 文 献

[1] Ebcioğlu K, Altman E R. DAISY: Dynamic compilation for 100% architectural compatibility[J]. ACM SIGARCH Computer Architecture News, ACM, 1997, 25(2): 26-37

[2] Hertzberg B, Olukotun K. DBT86: A Dynamic Binary Translation Research Framework for the CMP Era[C]// PESPMA 2009. 2009: 41-46

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

[4] Bruening D, Qin Zhao, Amarasinghe S. Transparent Dynamic Instrumentation[J]. Sigplan Notices-SIGPLAN, 2012: 133-144

[5] Guan H B, Ma R H, Yang H B. MTCrossBit: A dynamic binary translation system based on multithreaded optimization[J]. Science ChinaInformation Sciences, 2011, 54(10): 2064-2078

[6] 包云程，梁阿磊，管海兵. 动态二进制翻译基础平台 CrossBit 的设计与实现[J]. 计算机工程，2007，33(23)：100-101
Bao Yun-cheng, Liang A-lei, Guan Hai-bing. Design and Implementation of CrossBit: Dyanmic Binary Translation Infrastructure[J]. Computer Engineering, 2007, 33(23): 100-101

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

[8] Payer M, Gross T R. Generating low-overhead dynamic binary translators[C]// Proceedings of the 3rd Annual Haifa Experimental Systems Conference. ACM, 2010: 22-36

[9] Sridhar S, Shapiro J S, Bungale P P. HDTrans: a low-overhead dynamic translator[J]. ACM SIGARCH Computer Architecture News, 2007, 35(1): 135-140

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

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

[12] 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
Huang Gong-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