中国科学: 信息科学 2015 年 第 45 卷 第 4 期: 459–479
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
中国 CPU 与 DSP 专刊 · 论文

龙芯指令系统融合技术

胡伟武[①②*], 靳国杰[②], 汪文祥[①], 张晓春[①], 王焕东[②]

① 中国科学院计算技术研究所, 北京 100190
② 龙芯中科技术有限公司, 北京 100095
* 通信作者. E-mail: hww@loongson.cn

收稿日期: 2014–10–16; 接受日期: 2015–01–08
国家科技重大专项“核高基”(批准号: 2009ZX01028-002-003, 2009ZX01029-001-003, 2010ZX01036-001-002, 2012ZX01029-001-002-002)、国家自然科学基金 (批准号: 61221062, 61100163, 61133004, 61173001, 61232009, 61222204) 和国家高技术研究发展计划 (863 计划) (批准号: 2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301) 资助

摘要 本文介绍了龙芯 CPU 的指令系统融合技术. 龙芯 CPU 在兼容 MIPS 指令系统的基础上进行扩充, 增加 MIPS 中不具备但 X86 和 ARM 具备的、对二进制翻译影响很大的核心功能, 形成龙芯指令系统 LoongISA. 在此基础上, 通过软硬结合的二进制翻译, 在龙芯 CPU 上运行 X86 和 ARM 的主流系统及应用, 以弥补 MIPS 架构软件生态的不足. 实验结果表明, 通过软硬件协同优化, 龙芯指令系统融合技术能够消除指令系统之间的壁垒, 使得 MIPS, X86, ARM 等不同指令集的软件能够融合到统一的 LoongISA 平台上, 不加区别地高效运行.

关键词 龙芯 CPU MIPS 指令系统 龙芯指令系统 二进制翻译 兼容 软件生态

1 引言

发展以 CPU 和操作系统为核心的自主软硬件已经成为中国的国家战略. 经过自 “十五” 以来的十余年努力, 以 “龙芯”、“飞腾”、“申威” 为代表的中国自主 CPU 研发取得了长足的进展, 部分设计技术达到了世界先进水平, 并在实际应用方面取得了若干突破, 出现了良好的发展势头. 随着自主 CPU 应用和产业化的不断深入, 相关的软硬件生态问题变得越来越突出. 虽然自主 CPU 都在不同程度上采取了与国外 CPU 兼容的技术路线, 如 “龙芯” 在取得 MIPS 架构授权的基础上采取与 MIPS 兼容的策略, 但比起主流的软硬件生态, 如 X86 在桌面/服务器领域的软件生态以及 ARM 在手持终端领域的软件生态, 自主 CPU 在软硬件生态方面还处在很初级的水平, 严重制约了自主 CPU 的产业化.

二进制翻译技术是实现跨指令系统兼容的重要手段. 二进制翻译技术在宿主机 (host) 上用软件模拟出一个目标机/客户机 (guest) 指令系统兼容的 CPU 来, 从而在宿主机上执行客户机的二进制代码, 达到兼容的目的. 如在 MIPS 计算机上模拟 X86 指令系统, 从而实现与 X86 兼容. 二进制翻译的最大问题是效率问题, 用软件模拟的 CPU 比硬件直接实现的 CPU 慢很多. 如在 MIPS 计算机上使用二进制翻译的方法运行 X86 二进制程序, 比起把该程序直接从源代码编译成 MIPS 指令并在 MIPS 计算机上执行, 运行速度一般有数量级的差异.

引用格式: 胡伟武, 靳国杰, 汪文祥, 等. 龙芯指令系统融合技术. 中国科学: 信息科学, 2015, 45: 459–479, doi: 10.1360/N112014-00300

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

<table>
<tr><td>Dalvik apps</td><td>ARM apps</td><td>X86 MS Windows</td><td>X86 apps on Linux</td><td rowspan="2">MIPS apps on Linux</td></tr>
<tr><td>Dalvik VM</td><td>ARM-MIPS apps BT</td><td>X86-MIPS system BT</td><td>X86-MIPS apps BT</td></tr>
<tr><td colspan="2">Android on MIPS</td><td colspan="3">Linux on MIPS</td></tr>
<tr><td colspan="5">LoongISA = MIPS + X86 BT support + ARM BT support</td></tr>
</table>

图 1 龙芯 CPU 运行 X86 和 ARM 应用的总体思路

Figure 1 Overview of X86/ARM applications running on Loongson CPU

通过硬件支持和软硬件协同可以有效提高二进制翻译的效率. 虽然不同的通用指令系统结构在功能上都是完备的, 可以通过指令组合实现各种复杂的功能, 但指令的具体定义不同, 翻译起来就会走弯路. 例如, X86 的每条运算指令都要产生 EFLAG 标志位, 如果用 MIPS 指令来模拟产生 EFLAG 标志, 至少需要几十条指令, 如果在 MIPS 中增加专门产生 EFLAG 的指令, 则翻译起来就可以做到一一对应. 又如, 访存指令都需要进行虚实地址转换, 从 X86 到 MIPS 的二进制翻译中, 要先把 X86 的虚地址转换成 X86 的物理地址, 再把 X86 的物理地址作为 MIPS 的虚地址转换成 MIPS 的物理地址, 上述转换过程如果由软件实现, 需要十多条指令, 如果在 MIPS 的 TLB (translation lookahead buffer) 中增加一些特定的支持, 就可以直接把 X86 的虚地址翻译成 MIPS 的物理地址, 彻底消除虚实地址转换的开销.

龙芯 CPU 在兼容 MIPS 指令系统的基础上通过 MIPS 指令系统的 UDI (user defined interface) 进行扩充, 增加 MIPS 中不具备但 X86 和 ARM 具备的核心功能, 形成龙芯指令系统 LoongISA, 使得从 X86 和 ARM 的二进制到龙芯 CPU 的二进制翻译更加方便. 通过软硬结合的二进制翻译, 在龙芯 CPU 上流畅运行 X86 和 ARM 的主流系统及应用, 以弥补 MIPS 架构软件生态的不足. 具体地说: (1) 在 Linux 操作系统上进行从 X86 到 LoongISA 的系统级二进制翻译, 以运行 MS Windows 系统及其应用. (2) 在 Linux 操作系统上进行从 X86 到 LoongISA 的应用级二进制翻译, 以运行基于 Linux 的 X86 应用. 虽然 Linux 上的多数应用都可以从开源社区获得并编译到 MIPS 环境下执行, 但有少数关键商业应用如 Oracle 数据库没有开源版本, 需要建立基于 Linux 的从 X86 到 LoongISA 的二进制翻译. (3) 在 Android 操作系统上进行从 ARM 到 LoongISA 的应用级二进制翻译, 以运行基于 Android 的 ARM 二进制应用. Android 系统支持 MIPS 指令集, 且其多数应用是基于 Dalvik 虚拟机的, 在 MIPS 系统上运行没有障碍. 但有少量对性能要求较高的应用中插入了 ARM 二进制码, 在 MIPS 指令系统上无法直接运行, 因此需要建立从 ARM 到 LoongISA 的应用级二进制翻译. 龙芯 CPU 与 X86 及 ARM 兼容策略如图 1 所示.

龙芯二进制翻译系统以 LoongISA 指令集兼容不同形态的 X86, ARM 指令集, 通过技术手段消除单一指令系统的壁垒, 使得 MIPS, X86, ARM 等不同指令集的系统及应用软件能够融合到统一的 LoongISA 平台上, 不加区别地运行. 作为中间层的二进制翻译器, 充分利用本地硬件支持最大化地提高模拟效率, 对上层应用软件提供目标指令集的虚拟运行环境. 不同二进制的应用软件与运行的硬件平台之间不再具有决定关系, 用于支持软件开发的编译器、汇编器, 以往被绑定在各自特定的 CPU 之上, 现在都能为 LoongISA 平台提供软件资源. 实验结果表明, 龙芯软硬结合的二进制翻译技术可以成数量级地提高从 X86/ARM 到 MIPS 的二进制翻译效率.

本文第 2 节介绍与二进制翻译有关的相关工作, 第 3 节介绍龙芯体系结构对 X86 和 ARM 二进制翻译的支持, 第 4 节介绍实现系统级和应用级 X86 兼容的龙芯二进制翻译系统, 第 5 节介绍 X86 二进制翻译的部分实验结果, 第 6 节是结论和未来工作.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

2 相关工作

跨指令系统的软件兼容是二进制翻译技术最重要的技术特征. 实现翻译代码的高效执行始终是该技术领域的核心问题. 学术界和工业界针对这一问题, 展开了大量的研究工作.

自 20 世纪 90 年代开始, 伴随着体系结构的多样化浪潮, 多种宿主机指令系统均针对 X86 目标机提出了各自的二进制翻译平台, 例如, Digital FX!32[1] 用于 Alpha 宿主机系统, IA-32 EL[2] 用于安腾 (Itanium) 宿主机, Transmeta CMS[3] 用于 Transmetas VLIW 指令系统. 此外, IBM 公司以 Power 体系结构为目标机, 提出了 DAISY[4] 二进制翻译系统, 以支持其 IBM VLIW 指令系统的推广. 上述二进制翻译系统能够实现较高的执行效率, 例如, FX!32 在 500 MHz Alpha 处理器上达到了 200 MHz X86 处理器的性能 [1], IA32 EL 在执行部分 SPEC CPU2000 定点测试时能够平均达到本地编译执行效率的 60%[2]. 然而, 上述二进制翻译平台针对特定目标机和宿主机, 缺乏针对不同目标机的适应性.

随着体系结构的不断发展, 多样化的指令系统对二进制翻译技术提出了新的挑战, 针对多种目标机的适应性成为关键需求. 该需求随着移动计算和云计算的迅速发展而受到广泛重视, 其根本原因在于移动计算和云计算环境广泛包含异构平台. 例如, Qualcomm 和 ARM 将异构计算应用于移动设备, 即 ARM big.LITTLE 体系结构 [5]. 又如, Google 的大型机群由配置多样的多种指令系统构成 [6]. 上述事例说明, 计算平台已经由传统的单一指令系统向多指令系统发展. 在此基础上, 移动计算和云计算要求二进制翻译能够在统一的框架下实现跨多种指令系统的软件调度和执行. 学术界对这一课题进行了开创性的探索. 文献 [7] 基于 QEMU 二进制翻译系统, 实现了异指令系统平台下的程序调度, 以利用多种异构核的独有特性, 如降低平均功耗等. 其中, QEMU[8] 是一种广泛应用的可移植二进制翻译系统, 能够适用于多种目标机 (如 X86, PowerPC, ARM, Sparc and MIPS) 和宿主机. 可移植性使得 QEMU 在诞生后的 10 年中得到了极为广泛的应用, 充分体现出可移植性 (即支持多种目标机和多种宿主机) 在当前二进制翻译技术发展中的重要意义.

然而, QEMU 的可移植性以大量牺牲模拟效率为代价. 评估表明, 未经优化的 QEMU 其模拟执行效率 (以龙芯平台上模拟 X86 为例) 仅达到本地编译执行效率的不到 10%[9]. 其根本原因是 QEMU 为实现对多目标机和多宿主机的兼容, 使用体系结构无关的中间代码, 从而无法利用宿主机体系结构特征, 难以对翻译代码进行深入优化. 对此, 龙芯平台已针对 X86 目标机提出了软硬件相结合的优化方案 [9], 即利用龙芯特有的虚拟化指令, 加速 X86 目标机程序的翻译执行效率, 基于 SPEC CPU2000 测试实现了 5∼10 倍的性能提升. 与此类似, 其他研究工作同样提出了利用宿主机特性, 优化二进制翻译性能 [10∼12]. 然而, 上述方法针对特定目标机, 并未提出能够适用于多种目标机的优化方法.

本文深入分析了二进制翻译系统的性能瓶颈, 提出了针对二进制翻译的通用软硬件协同优化技术, 在保持目标机的移植性的基础上实现广泛的性能优化. 与前人的工作相比, 本文的工作强调更加紧密的软硬件协同, 在少量的硬件支持下大幅度提高二进制翻译效率. 例如, 间接跳转始终是二进制翻译的重要性能瓶颈, 前人的相关优化如 FX!32 提出的 shadow stack[1] 方法, HDtrans 提出的 return cache[13], inline[14] 方法均针对特定的间接跳转, 优化函数调用和返回的过程. 龙芯提出了利用内容相联存储器 CAM 加速处理 X86 间接跳转的优化方法 [9], 本文进一步展示了对于多目标机, CAM 方法的适应性和挑战. 又如, 在系统级模拟中, 目标机虚地址到客户机实地址的转换过程是主要执行开销 [15]. 针对 Power 平台上的虚拟化环境, 提出了上述地址转换过程的加速优化方法. 然而, 由于二进制翻译中不同目标机的虚实地址转换机制的差异, 已有方法难以为二进制翻译中的地址转换提供通用的优化. 本文展示了龙芯平台上适用于不同目标机的通用软硬件协同优化方法. 再如, 翻译代码效率亦是二进制翻译的主要性能瓶颈. 已有研究基于 LLVM 优化引擎, 采用纯软件策略对翻译后代码进

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

行优化, 如 HQEMU[16] 和 LLBT[17]. 此类方法无法解决体系结构差异导致的性能瓶颈, 如模拟 X86 EFLAG、模拟 ARM 条件存取指令等. 本文基于对不同指令系统的深入研究, 提出了兼容多种目标机的二进制翻译指令系统.

3 体系结构对二进制翻译的支持

指令系统是计算机软硬件的界面. 不同指令系统在总体功能上都是自成体系的, 在多年实践中积累下来的成熟指令集如 X86, MIPS, ARM 等具有很大成分的共性, 都包括算术逻辑运算指令、转移指令、访存指令、系统管理指令等指令, 通用寄存器、浮点寄存器、TLB、内存空间等运行时环境, 各种例外入口, 包括用户态和核心态在内的多个系统状态等. 从高级语言程序编译成不同指令系统的二进制代码, 其执行效率不会有很大区别.

但由于不同指令系统的指令功能和运行时环境的具体定义不一样, 把一个指令系统的代码转换成另一个指令系统的代码执行效率会有很大差别. 例如, X86 指令系统通过运算指令产生 EFLAG 标志, 转移指令判断这些标志值进行转移, 而 MIPS 指令系统直接比较寄存器的值并根据比较结果转移, 各自效率都很高. 但如果要用 MIPS 的指令把每一条 X86 运算指令产生的 6 位 EFLAG 标志都模拟出来, 就需要至少几十条指令来模拟一条 X86 运算指令产生的 EFLAG 标志. 又如, MIPS 有 32 个通用寄存器, 而 X86 只有 8 个, 要在 X86 上模拟 MIPS 指令系统, 只能用 X86 的内存单元来存储 MIPS 中寄存器的值, 无疑会大大降低执行效率. 为了提高二进制翻译的效率, 要求宿主机指令系统不论从功能还是运行时环境方面都要足够丰富, 可以方便地、一一对应地模拟客户机的所有指令功能. 为此, 本文认真分析了 MIPS 和 X86 及 ARM 指令系统的差异, 在 MIPS 指令系统基础上增加了 MIPS 不具备、X86 和 ARM 具备而且对二进制翻译性能又是至关重要的指令功能, 形成了龙芯指令系统 LoongISA. LoongISA 以 MIPS 编码结构为基础, 在功能上是 MIPS, ARM, X86 的并集.

一个指令系统通常包括用户态和核心态两部分功能, 不同指令系统的用户态功能差异小于系统态功能差异, LoongISA 要实现 MIPS, ARM, 86 用户态功能的并集比较容易, 要实现 MIPS, ARM, X86 系统态功能的并集就难得多. 对于应用级的二进制翻译, 即在基于 MIPS 的操作系统中二进制翻译 X86 或 ARM 的应用程序, 应用程序所调用的系统调用直接由 MIPS 来实现, 二进制翻译的性能只取决于 LoongISA 对 X86 和 ARM 用户态指令的支持程度. 对于系统级二进制翻译, 即在基于 MIPS 的操作系统中二进制翻译 MS Windows 操作系统, 就要考虑对 X86 系统态功能的支持. 下面从指令功能、运行时环境、核心态功能等 3 个方面介绍 LoongISA 对 X86 及 ARM 到龙芯 CPU 二进制翻译的具体支持.

3.1 指令功能的扩展

(1) 增加生成运算标志的运算指令以及根据运算标志进行转移的转移指令. X86 和 ARM 的运算指令都能同时生成运算结果和标志, 转移指令则根据标志位转移. X86 有 6 位与运算结果有关的 EFLAGS 标志, 分别是进位标志 CF、奇偶标志 PF、辅助进位标志 AF、清零标志 ZF、符号位标志 SF、溢出标志 OF; ARM 有 4 位与运算结果有关的标志, 分别是符号位标志 N、清零标志 Z、进位标志 C、溢出标志 V. X86 和 ARM 的转移指令主要判断这些标志位进行转移; 而 MIPS 没有定点标志位, 其条件转移指令都是直接比较寄存器的内容作为转移条件, 如判断两个寄存器是否相等, 一个寄存器的值是否大于 0 等.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

```
0       SUB     ECX     EDX
1       JE      X86_target
                (a)

0.00            SUBU    Result  Recx    Redx
0.01            SRL     Rsf     Result  31      /*SF=Result[31]*/
0.02            BEQ     Result  R0      L1
0.03            ADD     Rzf     R0      R0      /*ZF=0*/
0.04            B       L2
0.05            NOP
0.06    L1:     ADDI    Rzf     R0      1       /*ZF=1*/
0.07    L2:     SRL     Rtmp1   Result  31
0.08            SRL     Rtmp2   Redx    31
0.09            SRL     Rtmp3   Recx    31
0.10            BEQ     Rtmp1   Rtmp3   L3
0.11            NOP
0.12            BEQ     Rtmp2   Rtmp3   L3
0.13            NOP
0.14            ADDI    Rof     R0      1       /*OF=1*/
0.15            B       L4
0.16            NOP
0.17    L3:     ADD     Rof     R0      R0      /*OF=0*/
0.18    L4:     SRL     Rhigh2  Recx    16
0.19            SRL     Rhigh1  Redx    16
0.20            SUBU    Rtmp    Rhigh2  Rhigh1
0.21            BEQ     Rtmp    R0      L5
0.22            NOP
0.23            BLTZ    Rtmp    L7
0.24            NOP
0.25            B       L6
0.26            NOP
0.27    L5:     SLL     Rlow2   Recx    16
0.28            SRL     Rlow2   Rlow2   16
0.29            SLL     Rlow1   Redx    16
0.30            SRL     Rlow1   Rlow1   16
0.31            SUBU    Rtmp    Rlow2   Rlow1
0.32            BLTZ    Rtmp    L7
0.33            NOP
0.34    L6:     ADD     Rcf     R0      R0      /*CF=0*/
0.35            B       L8
0.36            NOP
0.37    L7:     ADDI    Rcf     R0      1       /*CF=1*/
0.38    L8      ADD     Recx    Result  R0
1.00            BNE     Rzf     R0      MIPS_target
1.01            NOP
                (b)
0.0             SUBU    Result  Recx    Redx    /*Generates subtraction result*/
0.1             SETFLAG
0.2             SUBU    Reflag  Recx    Redx    /*Generates  EFLAG*/
1.0             X86JE   Ref lag MIPS_target     /*Conditional branch according to
                                                 EFLAG*/
                (c)
```

图 2 X86 EFLAG 翻译示例

Figure 2 Translation of X86 EFLAG. (a) Original X86 instructions; (b) translated MIPS instructions; (c) translated LoongISA instructions

如果用软件模拟生产 X86 和 ARM 的运算结果标志位, 则模拟一条运算指令的标志需要几十条指令. 图 2(b) 给出了模拟图 2(a) 中 X86 指令 “SUB ecx, edx” 运算结果及常用的四位标志 (SF, ZF, OF, CF) 的 MIPS 指令序列, 需要近 40 条指令. 为了降低模拟运算结果标志的二进制翻译开销, LoongISA 定义了一组专门用于产生 X86 和 ARM 的运算结果标志的指令. 对于每一条 X86 或 ARM 运算指令, 把它翻译成相应的 MIPS 运算指令生成运算结果之后, 增加一条专门用于产生运算标志的指令. 该指

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

```
 0          ADDGT    R1      R2      R3
                              (a)

0.00        BNE      Rz      R0      Lnext     /*Z==0 ?*/
0.01        NOP
0.02        BNE      Rn      Rv      Lnext     /*N==V ?*/
0.03        NOP
0.04        ADD      R1      R2      R3
0.05        ……                                /*Omitting N,Z,C,V bits
                                              generation */
0.06  Lnext ……
                              (b)

0.0         ADD      Rtmp    R2      R3
0.1         ARMADDGT Reflag  R2      R3        /*Generates EFLAG */
0.2         ARMMOVCT R1      Rtmp    Reflag    /*Conditional MOV
                                              according to EFLAG*/
                              (c)
```

图 3　ARM 条件 MOV 指令翻译示例

Figure 3　Translation of instruction MOV in ARM ISA. (a) Original X86 instruction; (b) translated MIPS instructions; (c) translated LoongISA instructions

令把运算标志保存在 MIPS 的通用寄存器中, 而不是增加硬件标志位, 节省了硬件开销. 为了节省指令槽, 只要在 MIPS 的运算指令之前加上一条 “SETFLAG” 前缀指令, 就把该运算指令的含义重新定义为产生标志的新指令. 同时, LoongISA 还增加了与 X86 及 ARM 的转移指令相对应的根据标志进行条件转移的指令, 所不同的是 LoongISA 判断的是存储在通用寄存器中的标志. 图 2(c) 给出了把图 2(a) 中的 X86 指令 “SUB ecx, edx” 翻译到 LoongISA 指令系统的结果, 比翻译到 MIPS 大大减少了指令数.

(2) 针对 ARM 的条件运算指令增加条件 MOV 指令. ARM 的普通运算指令也要判断四位标志作为是否把运算结果写回结果寄存器的条件. 如果直接翻译成 MIPS 指令, 则几乎每条 ARM 的运算指令都需要翻译成一个包括运算指令和转移指令在内的指令序列. 图 3(b) 给出了把图 3(a) 中 ARM 的条件运算指令 “ADDGT r1, r2, r3” 直接翻译成 MIPS 指令的指令序列, 仅为了模拟运算结果的条件写回就需要额外执行两次转移指令, 而转移指令会降低流水线执行效率. 如果在 MIPS 中也增加条件运算指令, 不论是指令编码槽还是硬件开销都很大. 为此 LoongISA 增加了与 ARM 的条件运算指令的 “条件” 相对应的条件 MOV 指令. 这样, 对于 ARM 的条件运算指令, 只要先用普通 MIPS 运算指令生成相应结果, 然后用条件 MOV 指令根据条件决定要不要把该运算结果 MOV 到另一个结果寄存器去即可. 图 3(c) 给出了把图 3(a) 中 ARM 的条件运算指令 “ADDGT r1, r2, r3” 翻译成 LoongISA 指令的新指令序列, 该指令序列避免了转移指令的使用.

(3) 增加对 X86 浮点特殊浮点格式的支持. X86 的浮点结构与 RISC 处理器不同, 包括栈结构寄存器号、80 位浮点数、浮点 Tag 寄存器等.

X86 结构的 X87 浮点部件有 8 个浮点寄存器, 这 8 个浮点寄存器的编号不是固定的, 而是相对于 X87 浮点状态字的 TOP 域编号. 指令码中的寄存器号加上浮点状态字中的 TOP 域的值才是要访问的寄存器编号. 如果在二进制翻译中动态计算上述浮点寄存器号, 会大幅度降低二进制翻译代码的执行效率. 为解决上述问题, LoongISA 定义了 3 位的 TOP 寄存器以及读写该寄存器的相关指令, 该 TOP 寄存器可以根据要求在译码阶段与指令的源寄存器号相加形成新的源寄存器号. 这样, 每条访问浮点寄存器的浮点指令翻译成 LoongISA 指令时可以节省 10 多条计算源寄存器号的指令.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

X86 结构中有一个 16 位的 Tag 寄存器用来指示 X87 的 8 个浮点寄存器内容. Tag 的每两位指出一个浮点寄存器的内容为有效浮点数、0、特殊浮点数和空 4 种可能的情况. X87 浮点部件根据每个浮点寄存器对应的 Tag 位来判断浮点寄存器栈上溢和下溢情况. 为此, LoongISA 定义了专门的指令来通过 MIPS 的通用寄存器模拟 X86 的 Tag 寄存器功能并定义了专门的例外入口处理浮点寄存器栈的溢出. 由于正常的 X86 程序不会产生浮点寄存器栈溢出, 因此二进制翻译器对一般的浮点运算不会生成 Tag 模拟指令.

X86 支持 80 位浮点格式而 RISC 只支持 64 位浮点格式. 通过软件指令在 80 位和 64 位浮点格式间互相转换需要 40 多条指令. LoongISA 定义了一条把存储在两个 64 位寄存器中的 80 位浮点数转换为 64 位浮点数的指令, 以及两条把存储在一个 64 位寄存器中的浮点数转换为 80 位浮点数并存储在两个 64 位寄存器中的指令, 可以大幅度减少 80 位和 64 位浮点数转换的指令数. 图 4 描述了 80 位的 X86 运算分别翻译成 MIPS 指令和 LoongISA 指令的结果代码.

(4) 对非对齐访存的支持. MIPS 要求访存地址是对齐的, 例如, 当访问单位为 4 字节的字时, 要求访问地址的低两位必须为 0; 而 X86 和 ARM 都支持非对齐访存, 对普通访存指令的访存地址没有限制. 由于访存地址在访存指令执行过程中才能确定, 二进制翻译过程中生成目标指令码时有时不知道相应 X86 或 ARM 访存指令是否地址对齐, 很难生成高效的 MIPS 二进制指令. 但 MIPS 提供了 LWL/LWR 以及 LDL/LDR 指令对, 可以通过两条指令的组合达到非对齐访存的目的. 例如, 指令 “LDL R1, 59(R0)” 把内存中第 59∼63 共 5 个字节的值取到 R1 寄存器的高 5 个字节, 而指令 “LDR R1, 66(R0)” 把内存中第 64∼66 共 3 个字节的值取到 R1 寄存器的低 3 个字节, 这样通过上述两条指令的组合把内存中第 59∼66 这 8 个字节的值取到 R1 寄存器, 间接实现了非对齐访存.

但 MIPS 中的 LWL/LWR 以及 LDL/LDR 指令对只能针对通用寄存器进行非对齐内存访问, LoongISA 把 LWL/LWR 以及 LDL/LDR 指令功能扩展到了浮点寄存器. 二进制翻译时, 软件对于能够判断出 X86 或 ARM 指令是对齐访问的, 直接翻译成 MIPS 的一条指令对齐访存指令, 对于判断不出是否对齐访问的才翻译成 LWL/LWR 或 LDL/LDR 指令对.

(5) 间接跳转指令的翻译. 提高间接跳转的翻译代码性能是二进制翻译的重要问题. 在把 X86 的间接跳转指令翻译成 MIPS 跳转指令过程中, 存在寄存器中的跳转目标是 X86 代码的目标地址, 而不是 MIPS 代码的目标地址. 在二进制翻译过程中需要通过哈希表的查找把 X86 代码的目标地址转换为 MIPS 代码的目标地址. 为了降低间接跳转指令的二进制翻译开销, LoongISA 定义了一个 64 项的 CAM(content associated memory) 来加速间接跳转指令目标地址的翻译. CAM 的每项内容包括一个进程号、一个地址域和一个数据域, 地址域可以用来存储源机器 (X86) 的间接跳转目标地址, 而数据域可以用来存储目标机器 (MIPS) 的间接跳转目标地址. 图 5 给出了一个通过 CAM 表把 X86 间接跳转指令到 LoongISA 指令的例子. 虽然实际翻译过程中还要考虑 CAM 不命中的情况, 但 CAM 失效的情况很少发生.

(6) 增加 PC 读取指令. 在 ARM 指令系统中 PC 是软件可见的寄存器, 大部分普通运算指令均可以将 PC 寄存器作为源或目的操作数直接访问, 而 MIPS 指令系统中 PC 并不作为软件可见寄存器. 对于 ARM 中将 PC 寄存器作为目的操作数的普通运算指令, 传统 MIPS 指令就可以高效翻译, 即先用普通 MIPS 运算指令计算出相同的运算结果, 再执行一条将该结果作为源操作数的寄存器间接跳转指令. 在翻译将 PC 寄存器作为源操作数的 ARM 普通运算指令时, 则需要首先将指令的 PC 值获取并写入到通用寄存器, 随后再执行相应的 MIPS 普通运算指令. 而 MIPS 指令系统中只能通过 link 类跳转指令将当前指令 PC+8 写入到通用寄存器, 不仅需要对寄存器中的值进行修正, 而且 link 类跳转指令在处理器中执行时存在引入误预测开销、污染返回地址栈 (RAS) 的不利因素. 为解决该问题,

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

```
0    FLD     *%R10
1    FMUL    *16(%R10)
2    FSTP    *%R10
```

(a)

```
0.00  LD      Rtmp1   12(R8)          /*Transforms the first 80-bit operand into 64 bits*/
0.01  LD      Rtmp2   4(R8)
0.02  ANDI    Rsign   Rtmp1
0.03  DSLL32  Rsign   Rsign   16
0.04  ANDI    Rexp    Rtmp1   0x3fff
0.05  DSRLL32 Rexp    Rtmp1   18
0.06  DSRL    Rexp    Rexp    2
0.07  DSLL    Rtmp2   Rtmp2   1
0.08  DSRL    Rsand   Rtmp2   12
0.09  OR      Rfp1    Rsign   Rexp
0.10  OR      Rfp1    Rfp1    Rsand
0.11  DMTC1   F7      Rfp1
0.12  LD      Rtmp1   28(R8)          /*Transforms the second 80-bit operand into 64 bits*/
0.13  LD      Rtmp2   20(R8)
0.14  ANDI    Rsign   Rtmp1   0xc000
0.15  DSLL32  Rsign   Rsign   16
0.16  ANDI    Rexp    Rtmp1   0x3fff
0.17  DSLL32  Rexp    Rexp    18
0.18  DSRL    Rexp    Rexp    2
0.19  DSLL    Rtmp2   Rtmp2   1
0.20  DSRL    Rsand   Rtmp2   12
0.21  OR      Rfp2    Rsign   Rexp
0.22  OR      Rfp2    Rfp2    Rsand
0.23  DMTC1   F8      Rfp2
1.00  MUL.d   F9      F7      F8      /*64-bit multiplication */
2.00  DMFC1   Rres    F9              /*The 64-bit product is transformed back into 80-bit **/
2.01  DSRL32  Rsign   Rres    31
2.02  DSLL    Rsign   Rsign   15
2.03  DSLL    Rexp    Rres    1
2.04  DSRA    Rexp    Rexp    4
2.05  DSLL32  Rexp    Rexp    17
2.06  OR      Rres1   Rsign   Rexp
2.07  ORI     Rint    Rzero   0x1
2.08  DSLL32  Rint    Rint    31
2.09  DSLL    Rsand   Rres    12
2.10  DSRL    Rsand   Rsand   1
2.11  OR      Rres2   Rint    Rsand
2.12  SD      Rres1   12(R8)          /*The result is written back*/
2.13  SD      Rres2   4(R8)
```

(b)

```
0.0  GSLQC1   F4   4(R8)          /*Loads128 bits to F4 and F5*/
0.1  CVT.d.ld F7   F4   F5       /*Transforms 80 bits into 64 bits*/
0.2  GSLQC1   F2   20(R8)        /*Loads128 bits to F2 and F3*/
0.3  CVT.d.ld F8   F2   F3       /*Transforms 80 bits into64 bits*/
1.0  MUL.d    F9   F7   F8       /*64-bit multiplication*/
2.0  CVT.ud.d F7   F9            /*Transforms 64 bits into high part of 80 bits*/
2.1  CVT.ld.d F8   F9            /*Transforms 64 bits into low part of 80 bits*/
2.2  GSSQC1   F7   4(R8)         /*Writes back 128-bit result*/
```

(c)

图 4　80 位浮点运算翻译示例

Figure 4　Translation of 80-bit float-point instructions. (a) Original X86 instructions; (b) translated MIPS instructions; (c) translated LoongISA instructions

LoongISA 增加了 PC 读取指令, 该指令直接将自身 PC 值写入到目标通用寄存器.

(7) SIMD 指令. X86 和 ARM 指令系统中都实现有向量指令, 并已经广泛应用. X86 指令操作的最大向量长度为 256 比特, ARM(A32) 指令操作的最大向量长度为 128 比特. 两套向量指令集均支持定点字节、半字、字、双字, 单精度、双精度浮点元素的操作. 在基本运算类指令上, X86 和 ARM 向

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

```
0     MOV     %RAX    %R11
1     JMPQ    %*R11
                (a)

0     MOVE    Rr11    Rrax
1.0   CAMPV   Rtmp    Rr11    /*Look up the first level indirect jump address */
1.1   CAMPV   Rtgt    Rtmp    /*Look up the final jump address */
1.2   JR      Rtgt
                (b)
```

图 5 间接跳转指令翻译示例

Figure 5 Translation of indirect branch instruction. (a) Original X86 instructions; (b) translated LoongISA instructions

量指令的定义与其他主流指令集中的向量指令大同小异, 细节的差异主要集中在对加、减、乘运算的中间结果和最终结果如何进行饱和、舍入、截断等操作. 而在混洗、排列类指令上, 两套向量指令集的定义则各具特色. 若直接参照 X86 和 ARM 的向量指令, 以 “一一对应” 的方式实现一套向量指令集, 则不仅庞杂难以实现, 还存在专利侵权等法务风险.

事实上, 向量指令的定义也存在着类似于标量指令集中 “RISC” 和 “CISC” 的两大流派. 再复杂的向量操作模式都可以分解为一系列基本而规整的向量操作模式, 而这些基本而规整的操作模式数量上并不多, 方便高效实现, 并且非常有利于编译软件进行调度优化. LoongISA 的 SIMD 指令定义采用了 “RISC” 的设计策略, 向量长度 256 比特, 以 MIPS 向量指令为基本集, 并扩展部分自定义指令. 扩展的自定义指令分为两大类: 一类是针对典型算法的专用加速指令, 如矩阵乘、FFT 运算等, 对于适配应用采用此类指令进行翻译, 甚至可获得更高的执行效率; 另一类是带有独立配置参数输入的通用饱和、舍入操作、通用混洗操作等指令, 此类指令的通用属性确保涵盖各类向量指令下的不同操作模式, 而采用独立的配置参数输入则可以避免将配置信息存于向量寄存器带来不利影响.

3.2 运行时环境的扩展

(1) 灵活的 TLB 格式支持客户机虚地址到宿主机实地址的转换由硬件直接完成. 系统级二进制翻译过程中所有客户机访存指令的执行都要两级虚实地址转换, 即先把客户机的虚地址翻译成客户机的物理地址, 再将客户机的物理地址作为宿主机的虚地址翻译成宿主机的物理地址. 在 QEMU 中, 对于 X86 的一条访存指令, 需要 10 多条本地 MIPS 指令来模拟一次 TLB 转换行为. 由于普通程序中访存指令的比例在 40%甚至更高, 所以软件虚实地址转换会极大地降低翻译效率. 因此需要在处理器的 TLB 中增加硬件支持, 由硬件完成从客户机虚地址到宿主机物理地址的翻译过程, 彻底消除虚实地址转换的开销.

为降低实现开销, LoongISA 的 TLB 可以同时存放 “宿主机虚地址 $\rightarrow$ 宿主机物理地址” 和 “客户机虚地址 $\rightarrow$ 宿主机物理地址” 两种类型的页表项, 前一类型的页表项由宿主机操作系统中的普通页表提供, 后一类型的页表项由二进制翻译虚拟机中维护的影子页表提供. 为了区分上述两类页表项, TLB 中增加机器地址空间标识号域 (machine ID, MID), 主机页表项和客户机页表项分配有不同的 MID. TLB 在进行虚实地址转换时, 不再使用 ⟨ASID, VADDR⟩ 而是使用 ⟨MID, ASID, VADDR⟩ 作为匹配查找内容.

基于 QEMU 进行系统级二进制翻译动态生成的代码中, 使用主机虚地址和客户机虚地址的访存指令紧密交织在一起, 如果将访存指令在进行虚实地址转换时所需的 MID 维护成一个全局的处理器状态, 则会导致大量处理器状态切换指令的插入. 为了克服上述问题, 动态生成的访存指令所使用的 MID 除了来自全局处理器状态外, 还可以由指令自身携带的信息生成. LoongISA 中定义了专门的访

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

Index	X86 virtual page		X86 physical page	
	Zero padding (12 bits)	Page base (20 bits)	Page base (20 bits)	Zero padding (12 bits)
0	000 h			000 h
1	000 h			000 h
…				
256	000 h			000 h

图 6　软件模拟的 TLB 结构

Figure 6　Architecture of software emulated TLB

```
 0        MOV     %EAX    4(%ESP)
                                          (a)

 0.1      ADDU    Rr11    Resp     4       /*Generates virtual address*/
 0.2      SRL     Rr12    Rr11     12      /*Base virtual address */
 0.3      ANDI    Rr13    Rr12     0xff    /*TLB index (Low 8bit of base virtual address)*/
 0.4      LUI     AT      %hi16(tlb)       /*start address of TLB table*/
 0.5      ORI     AT      AT       %low16(tlb)
 0.6      SLL     Rr14    Rr13     3       /*Evaluates offset in TLB table according to TLB
                                           index (each TLB entry takes 8 bytes)*/
 0.7      ADDU    Rr14    Rr14     AT      /*Address of TLB entry*/
 0.8      LW      Rr15    Rr14     0       /*Gets Base virtual address in TLB entry*/
 0.9      BNE     Rr15    Rr12     miss    /*TLB miss*/
0.10      NOP

      hit:                                /*TLB hit*/
0.11      LW      Rr15    Rr14     4       /*Base physical address in TLB entry*/
0.12      ANDI    AT      Rr11     0xfff   /*Offset within a page*/
0.13      ADD     AT      AT       Rr15    /*Physical address is evaluated*/
0.14      LW      Reax    AT       0       /*Loads data from physical address*/

      miss:
0.15      j       tlb_refill               /*Looks up page table and refills TLB table*/
                                          (b)

 0.1      SETMEM                           /*LoongISA-extended Instruction prefix*/
 0.2      LW      Reax    4(Resp)          /*Virtual address mapping is done by standalone X86
                                           TLB*/
                                          (c)
```

图 7　X86 访存指令翻译示例

Figure 7　Translation X86 memory access instruction. (a) Original X86 instruction; (b) translated MIPS instructions; (c) translated LoongISA instructions

存指令前缀包含了 MID 的信息.

图 6 描述了纯软件模拟 TLB 的结构, 图 7 描述了 X86 访存指令分别翻译成 MIPS 指令和 LoongISA 指令的结果代码. 纯软件模拟的 TLB 通常采用固定大小的数组, 翻译后代码要进行虚拟地址的比较、以及物理地址的计算, 至少要两次访存, 总的指令条数是 15. 在 LoongISA 中扩展了一条前缀指令 SETMEM, 后面跟随的访存指令都将使用独立 TLB 进行地址转换, 因此模拟代码只有两条指令.

(2) 硬件自动维护指令和数据 Cache 的一致性. 二进制翻译动态生成目标机器的代码, 这些生成的代码被作为翻译器的 “数据” 保存在数据 Cache, 需要从数据 Cache 写入下一级存储, 并从下一级存储中取到指令 Cache 作为指令执行. 通过软件把数据 Cache 的内容写回下一级存储非常耗时. 因此, 龙芯 CPU 通过硬件自动维护数据 Cache 和指令 Cache 的一致性, 当翻译器生成目标代码并写到数据

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved.　http://www.cnki.net

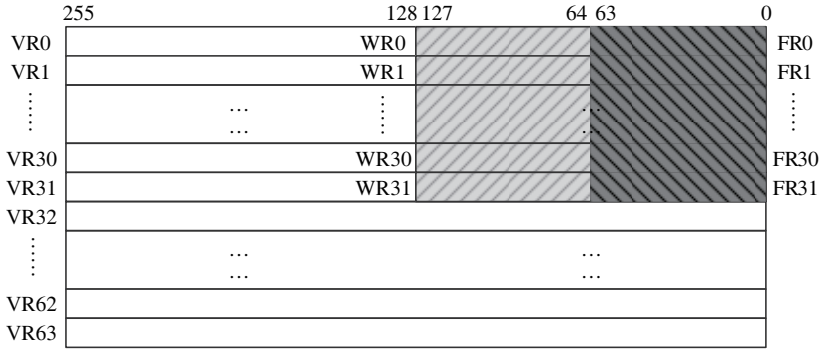


图 8 LoongISA 浮点/向量寄存器定义

Figure 8 Float-point and vector registers in LoongISA

Cache 后, 只要通过简单的同步, 就能维护与指令 Cache 的一致性.

(3) 增加宿主机寄存器数量, 避免用宿主机的内存模拟客户机的寄存器. 对客户机的寄存器的高效模拟是提高二进制翻译性能的关键. 如果宿主机没有足够的寄存器, 用内存单元来模拟目标机器的寄存器, 就会大幅度降低性能. X86 有 8 个通用寄存器, ARM 的 A32 结构有 16 个通用寄存器, 而 MIPS 有 32 个通用寄存器, 因此可以用 MIPS 的通用寄存器模拟 X86 和 ARM 的通用寄存器. X86 有 8 个浮点寄存器和 16 个 256 位的浮点/向量混用寄存器, ARM 的 A32 结构有 16 个 128 位的浮点/向量混用寄存器, MIPS 有 32 个 128 位的浮点/向量混用寄存器, LoongISA 对 MIPS 的浮点和向量寄存器进行了扩展, 扩展为 64 个 256 位的浮点/向量混用寄存器, 如图 8 所示, 因此足够用于模拟 X86 和 ARM 的浮点/向量寄存器.

此外, 每个体系结构都有一些专门用于系统控制的控制寄存器. 进行系统级二进制翻译时, 由于不同系统的控制寄存器千差万别, 难以用 MIPS 的控制寄存器模拟 X86 和 ARM 的控制寄存器, 一般的二进制翻译就用内存单元模拟宿主机控制寄存器的值. 为了提高系统级二进制翻译的性能, LoongISA 在通用定、浮点寄存器之外提供了足够的寄存器用于存放二进制翻译时模拟 X86 和 ARM 的控制寄存器及其他对性能影响很大的数据. LoongISA 实现了 MIPS 的 DSP 扩展中的 4 个 ac 寄存器组, 每个都由 64 位的 Hi 和 Lo 组成, 其中 Hi[0] 和 Lo[0] 是 MIPS 定点乘法和定点除法指令的目标寄存器. 在二进制翻译模式下不使用 MIPS DSP 指令, 可以把 3 个 Hi 寄存器和 3 个 Lo 寄存器共 6 个 64 位的寄存器给二进制翻译系统用于模拟 X86 和 ARM 的控制寄存器, 或者用于二进制翻译系统临时存储数据.

3.3 核心态功能的支持

(1) 增加系统状态提升虚拟机执行效率. 传统 MIPS 指令系统在进行系统级二进制翻译时, 客户机在核心态执行指令操作特权资源时将会触发例外, 陷入宿主机的核心态, 由操作系统进行模拟. 这种 “陷阱 — 模拟” 频繁发生, 将降低虚拟机的执行效率.

LoongISA 对传统 MIPS 的系统状态进行了扩展, 增加了一套具有有限特权的虚拟态, 称为客操作模式 (guest mode). 虚拟态下进一步划分出客 — 核心态、客 — 管理态和客 — 用户态 3 种状态, 其三者间的关系类似于现有 MIPS 架构中核心态、管理态和用户态. 处理器不处于虚拟态的状态称为根操作模式 (root mode), 也就是现有的处理模式. 现有模式下的核心态、管理态和用户态相应地称为根 — 核心态、根 — 管理态和根 — 用户态. 对根操作模式和客操作模式进行区分后, 客操作模式下的核心

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

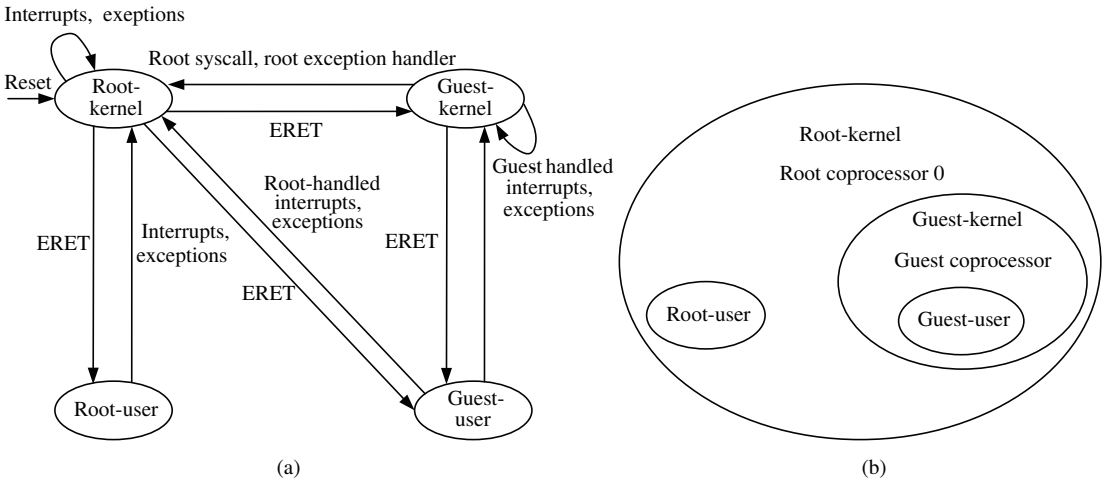


图 9 LoongISA 的系统状态

Figure 9 System status transition in LoongISA. (a) LoongISA VM operating modes; (b) operating state onion model

态可以允许虚拟机操作系统直接在虚拟机环境中对相关的特权资源进行操作, 无需再陷入至根 — 核心态进行模拟. 例如, 处理器处于客 — 核心态时客户机操作系统可以对其发生例外进行处理.

扩展后各系统状态之间可相互转换, 其关系如图 9(a) 所示. 结合根态和客态的划分, LoongISA 的虚拟化扩展采用 “洋葱分层模型”(图 9(b)) 来处理虚拟态下的地址映射和例外处理操作. 其中涉及虚拟机操作系统运行的系统状态有 3 个分别是: 具有最高特权的根 — 核心态、具有有限特权的虚拟 — 核心态以及无特权权限的虚拟 — 用户态. 根 — 用户态仍用于执行非虚拟化执行的用户程序. 在 “洋葱分层模型” 中, 每一层上的合法操作都可以在其外层所处的模式下执行.

(2) 增加专用的控制寄存器. 为实现上述操作系统态和例外类型的扩展, LoongISA 还增加了专用的控制寄存器. 这些新增的控制寄存器分为两大类: 一类是客态下的系统控制寄存器; 另一类是根态下新增的部分系统控制寄存器. 客态下的系统控制寄存器功能上是根态系统控制寄存器的一个子集, 物理上独立实现, 专门供虚拟机操作系统使用. 硬件同时支持两套系统控制寄存器可以实现客态和根态上下文的迅速切换, 避免耗时的通过内存保存/恢复上下文. 此外, 两套彼此独立的系统控制寄存器还允许客 — 核心态的特权指令代码能够尽可能避免虚拟机过多介入. 根态下新增的系统控制寄存器用于区分根态和客态, 对客态下特权资源的访问权限进行控制, 记录新增例外的例外类型, 配置新增例外入口的基址.

(3) 扩充例外类型. MIPS 与 X86 在系统态功能上存在较大差异, 为了在系统级二进制翻译过程中对 X86 核心态功能提供适当的支持, LoongISA 在 MIPS 指令系统基础之上扩展了一些新的例外类型和入口. 例如, 尽管 X86 客户机和 MIPS 主机的虚实地址映射合用同一套硬件 TLB, 但是 X86 客户机运行过程中发生的 TLB 重填例外使用独立的例外编码和例外入口, 否则, 若 X86 客户机和 MIPS 主机的 TLB 例外合用同一例外入口, 则软件必须对发生例外的指令和运行环境进行解析, 才能获知应该查找哪一套页表进行 TLB 重填. 由于 TLB 重填例外在系统运行时发生较为频繁, 因此此类例外处理程序的执行效率应尽可能高.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

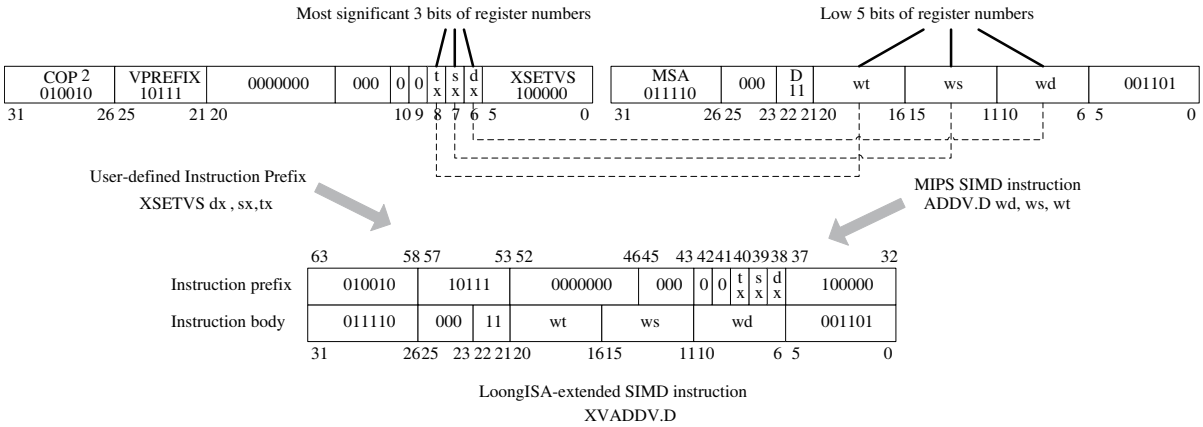


图 10 LoongISA 使用前缀指令的编码形式

Figure 10 Encoding of instructions with prefix in LoongISA

3.4 LoongISA 的指令编码

LoongISA 中为提升 X86 和 ARM 二进制翻译效率所增加的指令、控制寄存器和例外等都是在兼容 MIPS 指令系统的基础上通过 MIPS 指令系统的 UDI (user defined interface) 进行扩展的. 在 MIPS 指令系统的 UDI 中, 提供了少量的指令编码槽可用于新增指令的编码. 为了弥补这些可扩展指令编码槽数目有限的问题, LoongISA 除了利用部分扩展指令槽定义了一些最常用的新增指令外, 还通过定义专门的前缀指令来扩展指令编码空间. 前缀指令可将其后接指令的含义进行重新定义, 或是为其后接指令补充信息. 这种指令编码模式下, 后接指令或为已有 MIPS 指令, 或为采用 MIPS 指令编码风格的自定义指令. 图 10 给出了扩展向量指令前缀 “XSETVS” 与 MIPS 向量指令 “ADDV.D” 结合, 仍进行 “ADDV.D” 指令所表示的向量双字加操作, 只是通过前缀指令表明该指令操作将寻址 64 个 256 位的向量寄存器. 新指令操作数中 6 位寄存器号的最低 5 位记录在后接指令中, 最高位则统一位于前缀指令中, 在译码时拼接成完整的寄存器号.

4 龙芯二进制翻译系统

通过实现二进制翻译系统原型来验证本文所提出的指令系统融合技术. 根据指令集的代表性和使用的广泛程度, 选择了从 X86 到龙芯指令系统的二进制翻译系统, 其系统结构、实现方案、测试结果对于从 ARM 到龙芯指令系统的二进制翻译系统具有示范意义.

4.1 系统结构

龙芯 X86 二进制翻译系统 [9,18] 实现从 X86 到龙芯指令系统的二进制翻译. 该翻译系统基于跨平台的 QEMU 虚拟机进行, 分为在龙芯平台上运行 X86-Linux 应用程序的应用级翻译和在龙芯平台上运行 MS-Windows 操作系统及其应用的系统级翻译.

(1) 从 X86 到 LoongISA 系统级二进制翻译. 系统级二进制翻译是模拟一台目标机器的完整硬件, 对于 X86 的系统级二进制翻译, 模拟的硬件功能遵循标准 PC 机规范, 包括以下 3 方面内容: 一是 CPU 的模拟. 为了运行 Windows 操作系统, 需要模拟完整的 X86 指令集, 包括特权态、用户态指令, 以及 MMX/SSE 等扩展指令集. 除了指令集之外, 还需要模拟完整的 X86 运行时环境, 包括寄存器、

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

运行级别、分段存储、页表、保护机制等. 二是物理内存的模拟. 在 MIPS 本地内存中分配一段连续的区域, 来模拟 PC 机的物理内存. 三是总线控制器及基本设备集合的模拟. 实现北桥控制器如 PCI 规范的模拟, 以及实现 PC 机运行通用操作系统所需要的最小外设集合, 如串口、硬盘、显示器、时钟、网卡等. 对于每一种设备, 除了模拟该设备本身的硬件功能, 还需要模拟将设备与系统总线进行互联所需要的外设控制器. 系统级二进制翻译系统可以不经改造的运行原生 MS-Windows 系统和已有的全部 Win32 应用程序.

系统二进制翻译的瓶颈在于效率, 龙芯二进制翻译系统通过指令系统扩展和软硬件协同优化, 大幅度提高从 X86 到 LoongISA 的系统级二进制翻译效率. 例如, 采用加速寄存器访问模拟的全寄存器映射方法, 将 X86 通用寄存器组采用 LoongISA 的通用寄存器来模拟; 采用硬件 CAM 表来加速间接跳转指令目标地址的转换; 采用硬件 TLB 实现 X86 虚拟地址到 MIPS 物理地址的直接转换; 采用 MIPS 物理内存直接模拟 X86 的物理内存.

龙芯的 X86 系统级二进制翻译系统结构如图 11(a) 所示.

(2) 从 X86 到 LoongISA 应用级二进制翻译. 应用级二进制翻译器在功能上是系统级翻译器的子集. 应用级二进制翻译器只需要模拟 X86 CPU 中的应用态指令, 不需要模拟特权态指令, 在复杂度和实现代价上得以简化. 并且不需要模拟 X86 CPU 中全部的运行时环境, 例如, 寄存器集合只需要模拟应用程序可见部分, 只需要模拟应用态的运行级别, 不需要分段存储、页表、保护机制的模拟, 因此在性能上可以达到非常高的水平. 龙芯的应用级 X86 二进制翻译系统与系统级 X86 二进制翻译相比, 简化了指令集模拟子系统, 去除了存储管理模拟子系统和外部设备模拟子系统, 增加了模拟 X86 Linux 系统调用子系统.

龙芯的应用级 X86 二进制翻译系统结构如图 11(b) 所示.

4.2 二进制翻译系统优化

龙芯二进制翻译系统在性能上发挥软件、硬件结合设计的优势. 系统集成了二进制翻译领域内主要的成功优化方法, 包括动态优化 [19]、基本块 chaining[8]、自修改代码检查 [20] 等. 在此基础上, 除了充分利用 LoongISA 提供的硬件支持外, 还在提高代码质量和并行翻译方面进行了进一步优化. 主要包括以下几个方面:

(1) 基于静态数据流分析的基本块优化方法. 传统二进制翻译过程一般以基本块为翻译单位, 在翻译一个基本块时则以一条指令为单位. 一条客户机指令翻译为等价功能的数条本地指令. 为实现功能等价, 一般针对每一条指令进行寄存器映射, 并在不同指令之间通过寄存器间的数据传递指令实现对操作数的引用. 由于翻译过程的基本单位是单条指令, 因此翻译过程无法利用代码之间的数据相关性信息, 使得同样的数据被生成或传递多次, 造成冗余代码的出现. 此外, 二进制翻译平台一般强调跨平台通用性, 采用编译技术的中间代码机制. 中间代码往往具有平台无关性, 因此难以实现针对特定本地平台相关的后端优化. 龙芯二进制翻译系统提出了一种数据流优化方法. 在中间代码中增加对于宿主机体系结构的描述, 以数据流图描述宿主机指令的功能, 并通过数据流分析重建宿主机指令间的数据流图. 基于数据流图能够消除冗余的数据生成、数据传递、数据读写等操作指令, 以及基本块中的死代码, 最终提高翻译后代码效率.

(2) 带有基本块私有缓存的间接跳转的优化方法. 间接分支跳转指令是指跳转指令的目标地址位于一个寄存器或者内存单元中. X86 指令集大量利用了间接分支跳转指令, 实际的跳转目标地址只能在运行时刻确定, 对于虚拟机则需要额外的地址查找和地址转换代价. 现有解决这一问题的方法包括间接分支哈希表、目标地址内联、双地址返回栈等, 但是这些方法仍然未能充分利用查找目标的局部

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

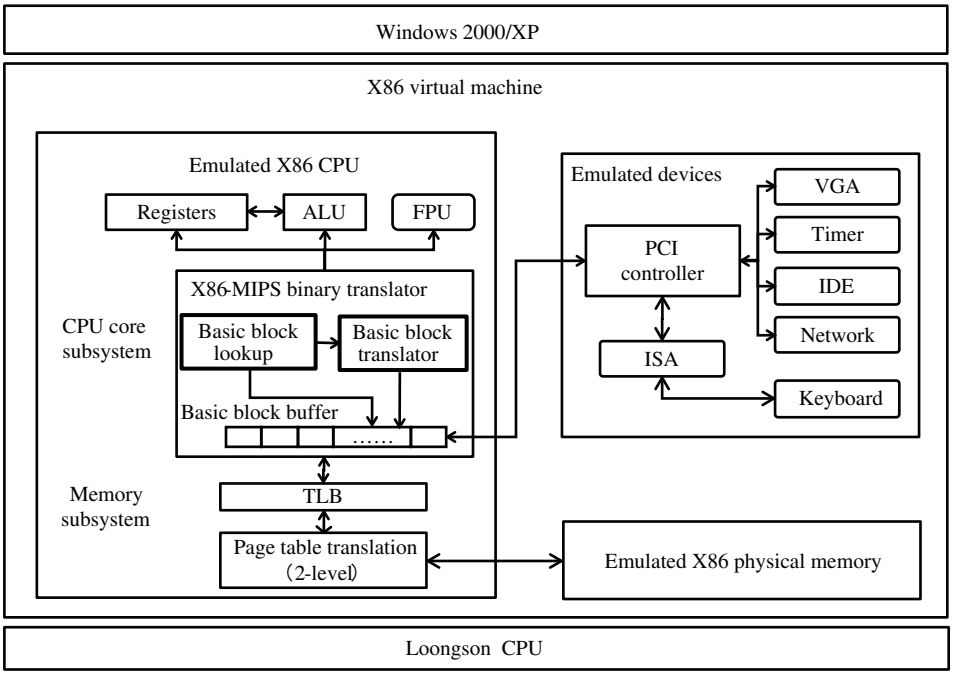


图 11 龙芯二进制翻译系统

Figure 11 Structure of Loongson binary translation system. (a) System-level binary translation from X86 to LoongISA; (b) application-level binary translation from X86 to LoongISA

性, 查找命中率仍然存在提升空间. 龙芯二进制翻译系统提出了一个为基本块分配私有缓存的优化方法, 为每一个间接分支基本块增加一个私有缓存, 缓存中保存该分支的目标地址, 提供给下一次预测查找. 分支目标良好的局部性带来了高的预测成功率, 减少了上下文切换的频度, 达到了与直接分支类似的基本块间链接效果.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

(3) 利用多核结构进行并行翻译优化. 为获得高质量翻译代码, 龙芯二进制翻译系统采用了重量级 (heavyweight) 代码优化技术, 包括死代码消除、常量传播、寄存器提升、窥孔优化、动态优化等. 为减少重量级翻译优化过程对整体执行效率的影响, 龙芯二进制翻译系统提出并实现了代码执行和翻译优化的多线程并行结构. 首先, 执行线程实现轻量级 (lightweight) 翻译过程, 并在翻译代码的执行过程中提供用于热点分析的采样数据; 其次, 翻译优化线程在不中止执行过程的条件下完成热点代码的分析和优化. 翻译优化线程根据采样数据确定执行热点, 进而将执行热点中的多个基本块组成超块, 最后利用重量级优化重新翻译超块. 在执行时间较短的程序中, 翻译优化过程在并行线程中进行, 因此不会影响程序的执行效率和响应速度. 在执行时间较长的程序中, 重量级优化将生成高效的翻译代码, 从而极大提高整体执行效率. 上述并行框架的主要开销在于多个线程对共享数据的访问, 包括采样数据、基本块缓冲区、翻译后宿主机代码等. 龙芯二进制翻译系统将在并行框架中针对大量并行读写过程设计简化的事务内存 (transactional memory) 模型, 以低开销的读写冲突检测和数据同步机制代替高代价的锁操作, 从而保证并行框架的执行效率.

5 实验结果分析

5.1 实验环境

本文的性能分析在龙芯 3A1500 四核处理器的 RTL 仿真验证平台上进行.

龙芯 3A1500 四核处理器实现了前述龙芯指令系统 LoongISA, 其结构如图 12(a) 所示. 龙芯 3A1500 通过片上交叉开关集成 4 个 64 位的四发射超标量 GS464E 高性能处理器核和 4 MB 共享三级 Cache. 该芯片还同时集成了 2 个 64 位 DDR2/3 控制器以及 2 个 HyperTransport 3.0 接口, 并支持多芯片通过 HyperTransport 接口互连和跨芯片的全局 Cache 一致性.

GS464E 处理器核结构如图 12(b) 所示. GS464E 处理器核的基本流水线包括 PC、取指、预译码、译码 I、译码 II、寄存器重命名、调度、发射、读寄存器、执行、提交 I 和提交 II 共 12 级.

取指与分支预测 GS464E 处理器核的取指部件每拍最多可以从指令 Cache 中取回 8 条指令在预译码阶段进行分支预测. 预译码阶段每拍最多可处理 4 条分支指令, 但只有最后一条才允许是预测跳转的分支指令. 经过预译码处理的指令后存入一个 64 项的指令队列中. 译码部件每拍从指令队列中取出 4 条指令, 将其翻译成处理器内部的指令格式送往后续流水级, 其中最多包含两条分支指令. 取指阶段的虚实地址转换通过指令 TLB 完成, 该 TLB 为 64 项全相联结构, 每一项存放一个页表项, 支持可变页大小.

在转移预测方面, GS464e 使用 128 项的分支目标缓存 (branch target buffer, BTB) 在取指流水级利用 PC 来预测可能存在的跳转指令的目标地址, 并及时更新取指 PC. 预译码流水级将根据其他预测器的预测结果对 BTB 的预测结果进行修正. 预译码流水级采用组合分支预测策略, 其中条件分支指令的跳转方向通过一个 8 K 项的全局转移历史表 (global branch history table, GBHT), 一个 8 K 项的局部转移历史表 (local branch history table, LBHT) 和一个 8 K 项的全局选择历史表 (global branch select table, GBSEL), 并采用 13 位的全局历史寄存器 (global histiry registor, GHR) 进行预测. 另有 1 K 项的转移目标地址缓存 (branch target address cache, BTAC) 用于预测非程序返回跳转类指令的目标地址, 以及一个 16 项的返回地址栈 (return address stack, RAS) 用于预测程序返回跳转指令的目标地址.

指令动态调度 GS464E 处理器核采用基于物理寄存器堆的重命名方式, 用于重命名定点寄存器

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

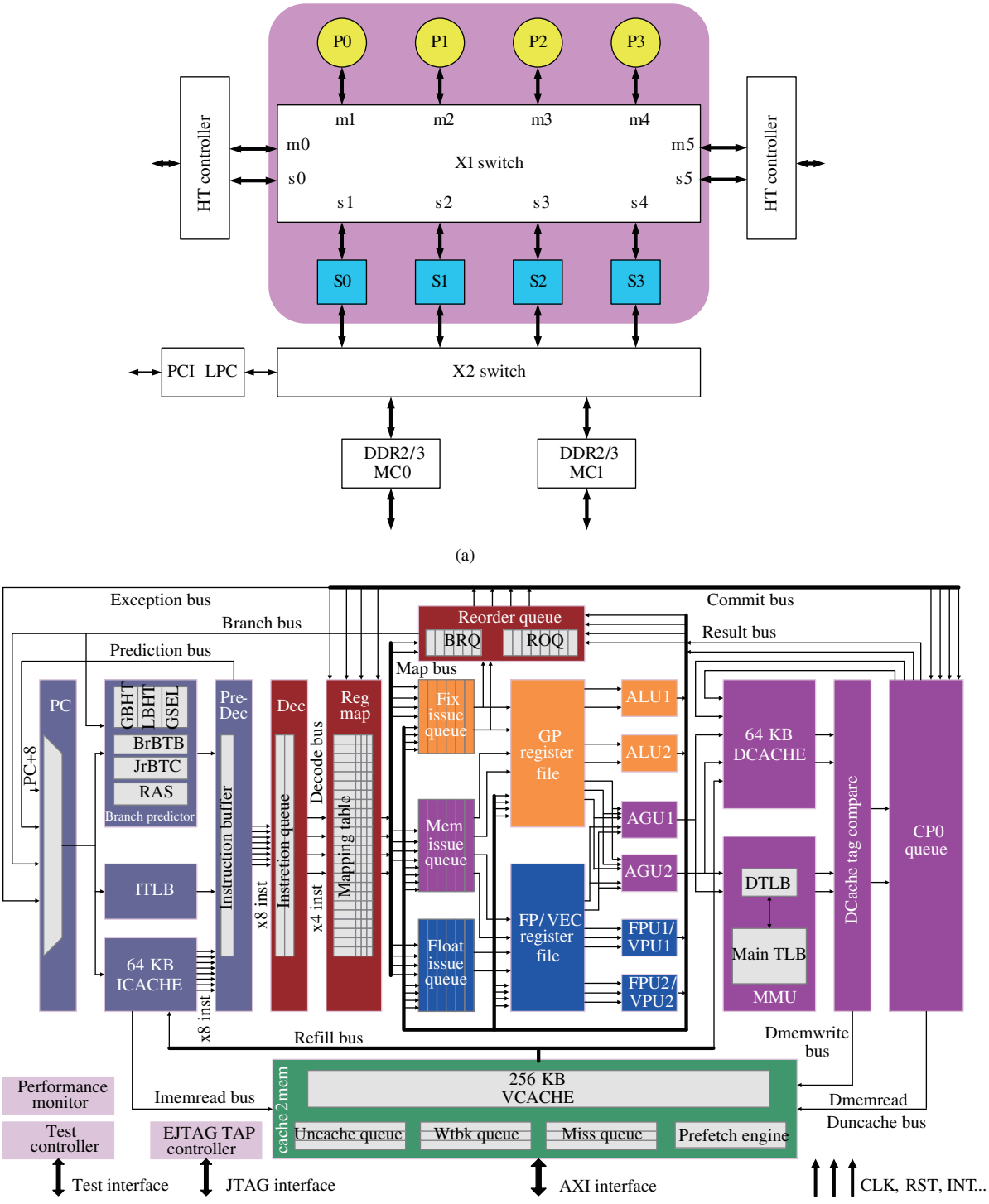


图 12 (网络版彩图) 龙芯 3A1500 及 GS464E 处理器核结构图

Figure 12 (Color online) Logical infrastructure of Loongson 3A1500 CPU and GS464E core. (a) Loongson 3A1500 CPU; (b) Loongson GS464E CPU Core

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

和重命名浮点寄存器的物理寄存器堆各 128 项, 另有 32 项重命名 HI/LO 寄存器、32 项重命名 DSPCtrl 寄存器和 32 项重命名浮点控制寄存器.

寄存器重命名后的指令送到保留站动态调度执行. GS464E 采用 3 个独立的分组保留站: 16 项的定点保留站、24 项的浮点保留站和 32 项的访存保留站. 完成寄存器重命名的指令在送入保留站同时也送入重定序队列, 用于确保按照程序顺序提交指令. 重定序队列共 128 项, 每拍最多可提交 4 条指令. 转移指令在重命名后进入重定序队列和保留站的同时进入转移队列. 转移队列共 24 项, 用于维护分支预测错误时回滚的顺序性.

运算功能部件 GS464e 包含 2 个定点部件 ALU1, ALU2 和 2 个浮点部件 FALU1, FALU2. 定点 ALU1 和 ALU2 功能相同, 能够执行定点加减、逻辑运算、移位操作、比较操作、分支、乘除操作以及所有 DSP 指令操作. 浮点 FALU1 和 FALU2 均能够执行浮点加减、取绝对值、取反、格式转换、比较和点乘、除、开方操作. 为了降低转移队列的复杂度, 浮点转移指令只在 FALU1 中执行.

访存部件 GS464E 处理器核的访存部件包含两个全功能的访存执行流水线, 每一个流水线都可以每拍流水地执行一条 load 或 store 操作. load 或 store 操作经访存保留站发射, 并在访存专用定点寄存器堆读取操作数后, 经过访存流水线执行. 访存执行包含地址计算、Dcache 读取、Tag 比较、数据写回四级流水.

GS464E 实现了非阻塞 Cache, 在一级数据 Cache 中未命中的访存操作将暂存在访存操作队列中等待数据返回, 后续 Cache 命中的访存操作可以提前执行并写回. GS464E 采用统一的访存操作队列, 可存放 load 和 store 操作, 共 64 项.

GS464E 中访存指令采用乱序发射机制, 即只要访存指令的操作数准备好即予以发射, 由访存操作队列维护乱序发射的访存指令之间的正确执行序. 访存操作在访问完毕一级数据 Cache 后即进入访存操作队列, 此时若发现存在因执行乱序而存在的数据相关违例情况, 则违例指令将回滚至访存保留站等待再次发射执行.

GS464E 的访存部件实现了两级 TLB 结构用于完成访存指令的虚实地址转换. 一级数据 TLB 的大小为 32 项, 每一项可以装载一个 MIPS 标准双页; 二级 TLB 由两个部分组成, 一个 64 项的全相联可变页大小的 CAM 部分, 以及一个 8 路组相联、每一路 128 项, 共计 1024 项的固定页大小的 RAM 部分.

片上 Cache 层次 GS464E 处理器核使用了 3 级片上 Cache 层次, 其中一级指令 Cache(Icache)、一级数据 Cache(Dcache) 以及数据和指令合用的二级牺牲 Cache(Vcache) 为每个处理器核内部私有, 第 3 级 Cache 为共享 Cache(Scache), 是所有处理器核共享的, 位于处理器核外. 一级指令 Cache 和一级数据 Cache 均为 4 路组相联, 各 64 KB 大小. 二级牺牲 Cache 为 16 路组相联, 共 256 KB. 三级共享 Cache 为 16 路组相联, 共 4 MB. 所有层次上的 Cache 使用相同 Cache 行大小, 为 64 字节. 一级指令 Cache 采用伪随机替换算法, 其余各级 Cache 均采用 LRU 替换算法.

GS464E 使用基于目录 (directory) 的 MESI 的 Cache 一致性 (Cache Coherency), 不支持升级 (Upgrade) 操作, 共享 (Shared) 块想要升级为独占 (Exclusive) 块, 需要先进行无效操作. Scache 会因为一个处理器核的请求而向另一个处理器核发出一致性请求, 包括无效请求、写回请求及无效并写回请求. 该一致性请求由被请求处理器核的 16 项的访存失效队列处理并给予响应.

龙芯 3A1500 处理器已于 2014 年 9 月流片. 本文的性能分析工作在龙芯 3A1500 的 RTL(register transfer level) 仿真加速平台进行. RTL 仿真平台采用 EVE 公司的 ZeBu Server 仿真加速器, 将龙芯 3A1500 流片的真实 RTL 代码综合到该仿真加速器中, 除了时钟频率比真实芯片慢 (每秒 1 M 时钟周期左右), 可以真实仿真龙芯 3A1500 每一拍的行为. 在性能分析时, 将核心时钟频率折算为 1 GHz,

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

表 1 性能测试结果

Table 1 Performance evaluation data

Test case	MIPS-Native	X86-MIPS original	X86-MIPS optimal
Linux booting	2.49×10^8	6.53×10^9	3.12×10^8

DDR3 时钟频率折算为 800 MHz.

5.2 初步实验结果

龙芯 3A1500 处理器还处于研发阶段, 已经完整实现从 X86 到龙芯指令系统的二进制翻译硬件支持, 对于从 ARM 到龙芯指令系统的二进制翻译硬件支持还处于由设计方案到工程实现的阶段, 因此以 X86 到龙芯指令系统的二进制翻译系统原型进行实验.

由于仿真环境只能提供有限的串口设备, 没有图形输出设备, 虚拟机无法模拟 Windows 所需要的 VGA 显卡和显示器, 因此实验的测试程序是以文本模式启动 Linux 操作系统. Linux 的启动过程需要计算机中所有组件协同支持, 反映了一个完整目标机器的综合模拟性能, 前文所述各项硬件支持和软件优化, 无论是针对 CPU 核心计算、访存、以及外设的优化都能够在 Linux 启动过程中发挥相应作用.

测试包括以下 3 种模式:

(1) 本地 MIPS 模式 (MIPS-Native). 把 Linux 通过本地 MIPS 的编译器进行编译, 生成 MIPS 机器指令, 直接在龙芯平台上运行.

(2) 未优化的二进制翻译模式 (X86-MIPS original). 把 Linux 通过 X86 的编译器生成 X86 机器指令, 使用原版 QEMU 二进制翻译器的 i386-softmmu 模式在龙芯平台上模拟运行.

(3) 优化的二进制翻译模式 (X86-MIPS optimal). 把 Linux 通过 X86 的编译器生成 X86 机器指令, 并且通过具有硬件支持以及软件优化的优化二进制翻译器进行模拟运行.

所有编译行为启用 GCC-O3 优化选项.

表 1 给出了优化前和优化后二进制翻译系统的性能测试结果. 通过表中数据可见, 优化后 Linux 操作系统启动是优化前的 21 倍, 达到本地直接编译运行的 79.8%.

需要说明的是, 上述实验是对龙芯 3A1500 在流片之前的初步验证, 实验平台是 EVE 仿真环境. 龙芯二进制翻译系统的优化还比较初步, 没有充分利用硬件支持和软件优化手段, 尚未反映最好的全面优化效果, 还需要在后续工作中还会继续深入磨合和优化, 尤其是在流片后的正式芯片中进一步提升其实用化的潜力.

6 结论及未来工作

本文的研究表明, 通过在 MIPS 指令系统的基础上扩展 MIPS 中不具备但 X86 和 ARM 具备的对二进制翻译影响很大的核心功能, 并通过软硬结合的二进制翻译, 可以大幅度提高从 X86(或 ARM) 到龙芯 CPU 的二进制翻译性能. 本文介绍的指令系统融合技术可以通过技术手段消除单一指令系统的壁垒, 使得 MIPS, X86, ARM 等不同指令集的系统及应用软件能够融合到统一的 LoongISA 平台上, 不加区别地运行. 实验结果表明, 龙芯软硬结合的二进制翻译技术可以大幅度提高二进制翻译的性能.

目前, 龙芯 3A1500 正在流片生产过程中, 龙芯二进制翻译系统也还比较初步, 有些硬件支持没有充分利用. 本文后续的工作包括两个方面: 一是在龙芯 3A1500 流片后在基于龙芯 3A1500 的计算机

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

上进一步优化二进制翻译系统, 并最终推出能够流畅运行 MS Windows 系统的计算机产品; 二是实现基于 LoongISA 的面向手持终端的 CPU 产品及基于 Android 的从 ARM 到 LoongISA 的二进制翻译系统, 并最终推出基于龙芯 CPU 的手持终端产品.

参考文献

1 Chernoff A, Herdeg M, Hookway R, et al. FX!32: a profile-directed binary translator. IEEE Micro, 1998, 18: 56–64

2 Baraz L, Devor T, Etzion O, et al. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, 2003. 191–201

3 Dehnert J C, Grant B K, Banning J P, et al. The Transmeta Code Morphing(TM) software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, Washington, 2003. 15–24

4 Ebcioglu K, Altman E, Gschwind M, et al. Dynamic binary translation and optimization. IEEE Trans Comput, 2001, 50: 529–548

5 Greenhalgh P. Big. LITTLE Processing with ARM Cortex-A15 & Cortex-A7. ARM White Paper, 2011

6 Mars J, Tang L. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, New York, 2013. 619–630

7 DeVuyst M, Venkat A, Tullsen D M. Execution migration in a heterogeneous-ISA chip multiprocessor. In: Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, 2012. 261–272

8 Bellard F. QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, Anaheim, 2005. 41

9 Hu W, Wang J, Gao X, et al. Godson-3: a scalable multicore risc processor with x86 emulation. IEEE Micro, 2009, 29: 17–29

10 Moore R W, Baiocchi J A, Childers B R, et al. Addressing the challenges of DBT for the ARM architecture. In: Proceedings of the 2009 ACM SIG-PLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, New York, 2009. 147–156

11 Mittal A, Bansal D, Bansal S, et al. Efficient virtualization on embedded power architecture platforms. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, 2013. 445–458

12 Ottoni G, Hartin T, Weaver C, et al. Harmonia: a transparent, efficient, and harmonious dynamic binary translator targeting the Intel architecture. In: Proceedings of the 8th ACM International Conference on Computing Frontiers, New York, 2011. 26: 1–10

13 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, New York, 2006. 175–185

14 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, Washington, 2007. 61–73

15 Chang X, Franke H, Ge Y, et al. Improving virtualization in the presence of software managed translation lookaside buffers. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, New York, 2013. 120–129

16 Hong D, Hsu C, Yew P, et al. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In: Proceedings of the 10th International Symposium on Code Generation and Optimization, New York, 2012. 104–113

17 Shen B, Hsu W, Yang W. A retargetable static binary translator for the ARM architecture. ACM Trans Architect Code Optimiz, 2014, 11: 1–25

18 Hu W, Liu Q, Wang J, et al. Efficient binary translation system with low hardware cost. In: Proceedings of IEEE International Conference on Computer Design, Lake Tahoe, 2009. 305–312

19 Ebcioglu K, Altman E. DAISY: dynamic compilation for 100% architectural compatibility. In: Proceedings of the 24th Annual International Symposium on Computer Architecture, New York, 1997. 26–37

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net

20 Gschwind M, Altman E, Sathaye S, et al. Dynamic and transparent binary translation. IEEE Comput, 2000, 33: 54–59

LoongISA for compatibility with mainstream instruction set architecture

HU WeiWu[1,2*], JIN GuoJie[2], WANG WenXiang[1], ZHANG XiaoChun[1] & WANG HuanDong[2]

1 *Institute of Computing Technology, Chinese Academy of Sciences, Beijing* 100190*, China*;
2 *Loongson Technology Co. Ltd., Beijing* 100095*, China*
*E-mail: hww@loongson.cn

Abstract This paper introduces the Loongson instruction set architecture (LoongISA), which extends the MIPS instruction set architecture for compatibility with X86 and ARM mainstream instruction set architectures. New instructions, runtime environments, and system states are added to MIPS through MIPS UDI (User Defined Interface) to accelerate the binary translation of X86 and ARM binary codes to LoongISA binary code. In addition, binary translation systems have been built based on LoongISA to run MS-Windows and its applications, X86 Linux applications, and ARM Android applications. LoongISA is implemented in the Loongson-3A1500 four-core CPU product of Loongson Technology Corporation Limited. Performance evaluations using the Loongson-3A1500 FPGA verification platform show that with hardware support, the binary translation system of Loongson 3A1500 can achieve very high efficiency.

Keywords Loongson CPU, MIPS instruction set architecture, Loongson instruction set architecture, binary translation, compatibility, software ecosystem



HU WeiWu was born in 1968. He received his Ph.D. degree from the Institute of Computing Technology of the Chinese Academy of Sciences, Beijing, in 1996. He is currently a professor and Ph.D. supervisor in the Institute of Computing Technology. His research interests include high-performance computer architecture, parallel processing, and VLSI design.



JIN GuoJie was born in 1979. He received his Ph.D. degree in computer science from Beihang University, Beijing, in 2010. Currently, he is a senior engineer at Loongson Technology Corporation Limited. His research interests include computer architecture, virtual machine, binary translation, and compilation optimization.



WANG WenXiang was born in 1983. He received his Ph.D. degree from the Institute of Computing Technology of the Chinese Academy of Sciences, Beijing, in 2012. His main research interests include high-performance computer architecture and microprocessor architecture.



ZHANG XiaoChun was born in 1984. He received his Ph.D. degree from the Institute of Computing Technology of the Chinese Academy of Sciences, Beijing, in 2014. He is currently an engineer in the Institute of Computing Technology. His research interests include microprocessor architecture and binary translation.

?1994-2015 China Academic Journal Electronic Publishing House. All rights reserved. http://www.cnki.net