

基于静态寄存器分配的系统仿真协同优化方法

蒋烈辉, 陈慧超*, 董卫宇, 张彦文

(信息工程大学 数学工程与先进计算国家重点实验室, 郑州 450001)

(* 通信作者电子邮箱 huichaochen@outlook.com)

摘要:针对 X86 系统仿真中基于静态寄存器分配的代码翻译机制导致的目标代码膨胀率高、翻译引擎和执行引擎间切换开销大两方面问题,提出了以寄存器映射、自定义指令和影子寄存器为基础的软硬协同优化方法。寄存器映射优化将对内存中模拟的源机器寄存器的操作转化为对本地机器寄存器操作,降低了翻译后目标代码膨胀率;自定义指令和影子寄存器优化将引擎切换时上下文的备份和恢复操作简化为 2 条自定义指令,提升了引擎切换效率。相比协同优化前, X86 仿真系统 Linux-0.2 的翻译后目标代码膨胀率降低了 21.9%, 开关机时间获得了 1.35 的加速比。测试结果表明了该协同优化方法对于提升系统仿真效率具有可行性和有效性。

关键词:系统仿真; 协同优化; 寄存器映射; 自定义指令; 影子寄存器

中图分类号: TP391.9 **文献标志码:** A

Co-design optimization approach based on static register allocation in system emulation

JIANG Liehui, CHEN Huichao*, DONG Weiyu, ZHANG Yanwen

(State Key Laboratory of Mathematical Engineering and Advanced Computing, Information Engineering University, Zhengzhou Henan 450001, China)

Abstract: To reduce the expand rate of translated code and the cost of switching between translation engine and execution engine based on static register allocation strategy in X86 system emulation, a co-design optimization approach based on register mapping, custom instruction and shadow register was proposed. The operation to the emulated source registers in memory was transformed into the operation to local registers by optimization of static register allocation and the expand rate of translated object code was reduced; the backup and restore operation between engine switching were simplified to two custom instructions by co-design optimization of custom instructions and shadow registers. After co-design optimization of X86 emulation system Linux-0.2, the average expand rate of translated code comes down by 21.9%, the period of the emulation system's boot and shutdown obtains speed-up of 1.35. Evaluation demonstrates the feasibility and effectiveness of the proposed co-design optimization approach.

Key words: system emulation; co-design optimization; register mapping; custom instruction; shadow register

0 引言

系统仿真(System Emulation)能够解决跨平台操作系统和软件的透明移植^[1-3],使新型精简指令集处理器(Reduced Instruction Set Computing, RISC)能够得到快速的推广应用,因而近年来受到越来越多的关注。

QEMU(Quick Emulator)是一个基于动态二进制翻译的多源到多目标仿真器,能够仿真目标机器的用户态和系统态^[4],其将目标寄存器固定地保存在本地内存中,代码翻译中大量使用临时变量,当翻译代码需对目标寄存器操作时,必须从内存中载入目标寄存器值或将寄存器值写回内存。显然这种方法导致了大量内存读写操作,降低了系统仿真效率。基于 X86 和 ARM 平台,文献[5]试图对 QEMU 中使用的寄存器策略进行改进,并对多种映射策略进行了对比实验,但翻译程序的性能提高微乎其微,根源在于其所使用的映射策略并没有有效降低对内存的访问频度。文献[6]以 PowerPC 到 Alpha 的系统仿真平台为基础,提出了分段映射和特殊寄存

器功能剪裁相结合的方法,使翻译程序的性能得到了一定的提高,但对通用寄存器的映射关注较少。文献[7]以 X86 到龙芯 MIPS(Microprocessor without Interlocked Piped Stages)平台的系统仿真为基础,映射使用频繁的 EAX、ESP、EBP 到龙芯 s4、s5、s6 寄存器,使基于 X86 的操作系统启动和关闭时间明显减少,由于其并没有深度结合代码翻译和执行机制,实际应用中容易产生意想不到的错误。文献[8]使用全寄存器映射方法,将 X86 架构下 8 个通用寄存器全部映射到 MIPS 处理器 s1~s8,系统仿真性能虽得到了一定的提高,但在引擎切换的上下文中引入了全局寄存器变量和内存中由数组结构维护的寄存器数组间的同步操作,增加了引擎切换开销。此外,现有对引擎切换时上下文备份和恢复的优化技术,基本都集中在以减少引擎切换上下文频度为目标的基于热路径的超级快优化^[9]和基于跳转指令的块链优化^[10],还未见有以降低引擎切换操作复杂度为优化目标的相关工作。

本文依托 X86 系统仿真软件平台体系结构桥(Architecture Bridge, ARCH-BRIDGE),在深入分析基于静态

收稿日期:2013-11-27;修回日期:2013-12-18。 基金项目:国家 863 计划项目(2009AA012200)。

作者简介:蒋烈辉(1967-),男,浙江东阳人,教授,博士生导师,博士,主要研究方向:体系结构、系统虚拟化; 陈慧超(1988-),男,河南开封人,硕士研究生,主要研究方向:系统虚拟化; 董卫宇(1976-),男,辽宁沈阳人,副教授,博士研究生,主要研究方向:操作系统、系统虚拟化; 张彦文(1987-),男,湖南娄底人,硕士研究生,主要研究方向:逆向工程、二进制翻译。

寄存器分配的代码翻译机制的基础上,提出了以解决翻译后目标代码膨胀率高的寄存器映射优化,和以降低引擎切换操作复杂度为目标的自定义指令和影子寄存器相结合的协同优化技术。寄存器映射优化技术深度结合静态寄存器分配策略,将对内存中的模拟源机器寄存器的操作转化为对本地机器寄存器操作,简化了代码翻译规则,降低了翻译后目标代码膨胀率。引擎切换操作优化利用自定义指令和影子寄存器,将引擎切换时上下文的备份和恢复操作简化为2条自定义指令,使引擎切换的效率得到了极大的提升。可见,寄存器映射优化和引擎操作优化都能够有效提升系统仿真效率。

1 基于静态寄存器分配的代码翻译机制

寄存器的静态分配策略是在 X86 系统仿真的代码翻译阶段实施的,所以理论分析从代码翻译机制切入,继而分析静态寄存器分配策略及其引入的系统仿真开销。

1.1 代码翻译机制

ARCH-BRIDGE 目前的翻译机制比较简单,对每条指令翻译的基本步骤如下:

- 1) 将指令操作数从虚拟机中加载到宿主机器寄存器;
- 2) 生成执行的指令;
- 3) 将执行结果由宿主机器寄存器保存到虚拟机中。

为辅助对 X86 指令的翻译,ARCH-BRIDGE 定义了若干支持函数类,包括函数类 `gen_insn_xxx()`、`gen_prim_xxx()`、`gen_code_xxx()`、`xxx_translator()`。为协调宿主机器寄存器的分配,将这些支持函数分为4个级别(级别3最高,级别0最低),高级别函数可直接调用任何一层低级别函数,但低级别函数不应调用高级别或同级别的函数,这样可确保所使用的宿主机器寄存器不会被二次分配,各级别函数只使用特定的寄存器子集,如表1。

表1 代码翻译支持函数特性说明

函数命名	级别	函数职责	寄存器分配
<code>gen_insn_xxx</code>	0	构造单条 RISC 指令	不分配寄存器
<code>gen_prim_xxx</code>	1	构造基本 X86 微操作	GPRIM_TMP_X
<code>gen_code_xxx</code>	2	构造 X86 微操作组合	GCODE_TMP_X
<code>xxx_translator</code>	3	翻译1条 X86 指令	TRANS_TMP_ALL

定义代码翻译支持函数的好处是方便移植,只需修改支持函数就可以将 ARCH-BRIDGE 移植到其他 RISC 平台。

1.2 静态寄存器分配

寄存器分配策略就是要明确翻译后目标代码可以使用哪些寄存器以及如何使用这些寄存器,确保系统仿真中翻译引擎、执行引擎、支持函数 C Helpers 三者协作执行时不发生寄存器冲突。ARCH-BRIDGE 目前采用静态寄存器分配方案,即约定目标代码只使用特定的宿主机器寄存器的子集,并在生成目标指令的同时确定使用哪个宿主寄存器。

假定宿主机器采用 OR1200 处理器,则各个宿主寄存器的使用约定可以分为以下4类:

类别1 R13/R15/R17/R19/R21/R23/R25/R27/R29/R31。从被调函数(Callee)的角度讲,这些寄存器称为无需保存寄存器(Scratch Register),表示被调函数可以直接使用这些寄存器而不必调用前保存(入栈)和调用后恢复(出栈),这些寄存器的保存和恢复由主调函数(Caller)负责。

类别2 R20/R22/R24/R26/R28/R30。从被调函数的角度讲,这些寄存器称为需保存寄存器(Saved Register),被调函数在使用这些寄存器前必须进行保存并在调用后恢复,主调函数可以不必考虑这些寄存器的保存和恢复问题。

类别3 R3/R4/R5/R6/R7/R8。从被调函数的角度讲,这些寄存器称为参数寄存器(Argument Register, ARG_REG),在函数调用时主调函数要通过这些寄存器向被调函数传递参数。

类别4 R0/R1/R2/R9/R11/R12。按照 OR1200 的应用程序二进制接口(Application Binary Interface, ABI)约定,这6个寄存器是 OR1200 处理器内的特殊目的寄存器,其中:R0 恒为0, R1 为栈指针寄存器, R2 为栈基址指针寄存器, R9 为返回地址寄存器, R11 为返回值寄存器, R12 为高32位返回值寄存器。

ARCH-BRIDGE 中,目标代码使用的寄存器来自四个集合 $GPRIM_TMP_X = \{R13, R15, R17\}$, $GCODE_TMP_X = \{R19, R21, R23\}$, $TRANS_TMP_ALL = \{R25, R27, R28, R30, R20, R22, R24, R26\}$, $ARG_REGS = \{R3, R4, R5, R6, R7, R8\}$ 。其中:R20/R22/R24/R26/R28/R30 寄存器属于 Saved 寄存器。

1.3 引擎切换

翻译引擎切换到执行引擎时,翻译引擎为 Caller,执行引擎为 Callee,按照约定执行引擎须保存 Saved 寄存器;执行引擎切换到翻译引擎时,执行引擎为 Caller,翻译引擎为 Callee,按照约定翻译引擎须恢复 Saved 寄存器。引擎切换时上下文的保存和恢复工作实际上是由函数 `g_tb_wrapper()` 实现的。不考虑特殊情况,引擎切换的序列循环为:翻译引擎→`g_tb_wrapper()`→执行引擎→`g_tb_wrapper()`→翻译引擎,函数 `g_tb_wrapper()` 中包含有 prologue 和 epilogue 两小段汇编代码,分别用于保存和恢复寄存器 R20/R22/R24/R26/R28/R30、返回地址寄存器 R9。可见,每次保存与恢复工作至少14条加载存储指令来完成。

X86 系统仿真中,C Helpers 是为了翻译比较复杂的 X86 指令所添加的支持函数集,在执行引擎中的目标代码调用 C Helpers 时,执行引擎为 Caller, C Helpers 为 Callee。按照 OR1200 的 ABI 约定,C Helpers 可能破坏 Scratch 寄存器,因此执行引擎应该保存这些寄存器,但由于每条 X86 指令执行后都将结果保存回虚拟机,因此在 X86 指令边界处执行引擎不占用任何宿主机器寄存器,则执行引擎调用 C Helpers 时,不需要保存 Scratch 寄存器。此外,执行引擎调用 C Helpers 时,需要通过寄存器 R3~R8 向 C Helpers 传递参数,按照约定也应该保存这些寄存器,但由于 ARCH-BRIDGE 的静态寄存器分配策略,除了调用 C Helpers 之外执行引擎并不使用寄存器 R3~R8,因此也无需保存。

1.4 开销分析

通过对代码翻译机制的分析可知,每条 X86 指令执行后都将结果保存回虚拟机,这种机制的优点是实现简单,可以保证 X86 的精确异常特性,并且由于在 X86 指令边界处为该指令分配的宿主机器寄存器的内容都已作废,减少了调用 C Helpers 时寄存器现场的保存工作。该代码翻译机制的缺点也比较明显,即在翻译后目标代码中引入了大量的虚拟机状态与宿主机器寄存器间的数据传输(Load/Store),生成的翻译后

目标代码的质量不高。此外,RISC 处理器中,内存操作指令一般比寄存器操作指令占用更长的时钟周期,进一步降低了 X86 系统仿真效率。

由以上分析可知,基于静态寄存器分配的代码翻译机制引入两方面的仿真开销:一是由于翻译策略引入的大量的虚拟机状态与宿主机寄存器间的数据传输;二是在执行翻译代码之前,需要进行翻译引擎到执行引擎的上下文切换,将翻译引擎正在使用的部分寄存器保存到堆栈中,翻译后目标代码块经执行引擎正常执行完毕后,再从执行引擎环境切换到翻译引擎环境,将堆栈中此前保存的动态寄存器状态恢复回来。因此,系统仿真中应考虑对这两方面开销进行优化。

2 协同优化设计与实现

2.1 依托平台

32 位 RISC 处理器 OR1200 是 OpenRISC 项目的一个分支,由硬件开源组织 OpenCores 开发和维护,具有完整的 Verilog HDL 源代码和交叉编译工具链。其采用哈佛(Harvard)结构,具有五级流水线,支持内存管理单元(Memory Management Unit, MMU)、数据/指令缓存(Cache)及基本的数字信号处理(Digital Signal Processing, DSP)功能,实现了基本指令集,包括数据处理指令、特殊寄存器访问指令、转移类指令、异常处理类指令、乘法除法类指令和加载存储类指令。Linux Kernel 主线从 V3.1 版本开始正式加入了对 OpenRISC 架构的支持^[11],QEMU 从 1.2 版本^[12]已支持 OpenRISC 架构指令集仿真。由于 OpenRISC 完全开放性和免费性,近年来国内外许多个人和组织使用 OpenRISC 处理器核做了很多应用方面的研究。

2.2 寄存器映射

基于寄存器静态分配的代码翻译机制,引入了占用较多时钟周期的虚拟机状态与宿主机寄存器间大量的数据传输指令,提高了翻译后目标代码膨胀率,降低了系统仿真效率。OR1200 处理器可用通用寄存器数目为 32 个,则尽可能地将目标机器程序中最常用的寄存器,固定映射到本地机器的寄存器,就可以避免过多的访存操作,这是一种为提高仿真性能而牺牲系统仿真灵活性的方法。

X86 共有 8 个通用寄存器,分别为 EAX、EBX、ECX、EDX、ESP、EBP、EDI 和 ESI。对 ARCH-BRIDGE 系统仿真器上 Linux-0.2 操作系统启动过程的 profile,输入代码量为 8000 个基本块,利用 ARCH-BRIDGE 中文本日志功能,在基本块翻译完成后产生一个翻译日志,记录源程序中寄存器的使用情况。通过字符串匹配指令(grep),编写匹配查询程序(shell)对翻译日志中 X86 体系结构中的 8 个通用寄存器进行统计,得到 X86 架构下操作系统启动过程对这 8 个通用寄存器的访问次数及其所占百分比,如表 2 所示。

表 2 Linux-0.2 启动过程对寄存器的访问情况

X86 通用寄存器	访问次数	访问次数百分比/%	X86 通用寄存器	访问次数	访问次数百分比/%
EAX	25 547	29.6	ESP	13 839	16.0
EBX	10 876	12.6	EBP	4 180	4.8
ECX	6 946	8.1	EDI	5 587	6.5
EDX	11 437	13.3	ESI	7 890	9.1

通过对 ARCH-BRIDGE 上 Linux-0.2 操作系统启动过程

的分析,可以得到 X86 架构运行时最常用的寄存器是 EAX、EBX、EDX 和 ESP。结合第 1 章介绍的代码翻译机制和静态寄存器分配方案,将这 4 个最常用的 X86 寄存器分别对应映射于 OR1200 处理器的通用寄存器 R10、R14、R16 和 R18,如图 1。

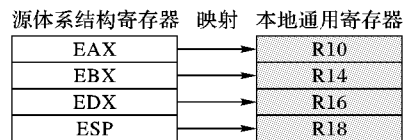


图 1 寄存器映射关系

利用编译技术,声明全局寄存器变量 R10、R14、R16 和 R18^[13],为使这 4 个寄存器能够被安全映射,通过修改编译器,使翻译引擎运行时保证不使用这些已经被执行引擎中的目标代码占用的寄存器;修改翻译策略,当翻译 X86 指令时如遇到 EAX、EBX、EDX 和 ESP 寄存器时,直接使用这 4 个已固定映射到本地的通用寄存器代替,无需再从内存中模拟的虚拟机寄存器中加载。

此外,根据上述分析,翻译引擎并不使用寄存器 R10、R14、R16 和 R18,执行引擎运行时把上述寄存器当成 X86 下对应放的通用寄存器 EAX、EBX、ESP 和 EBP 来使用,这就相当于系统仿真时执行引擎独占这 4 个寄存器,因此,引擎切换时,无须对这四个寄存器进行保存和恢复。当然,如果宿主机的其他程序使用这些寄存器,需要首先把这些寄存器中的值保存到内存中,程序退出后再将这些寄存器的值从内存中恢复回来。

可见,寄存器映射优化,可以简化代码翻译规则,降低翻译后目标代码膨胀率,从而提升 X86 系统仿真效率。

2.3 影子寄存器

经 1.3 节分析可知,引擎切换时,函数 g_tb_wrapper() 中 prologue 和 epilogue 两小段汇编代码分别用于保存和恢复寄存器 R9/R20/R22/R24/R26/R28/R30 值。每一次的保存与恢复工作都需要额外增加 14 条加载存储指令来完成。这些重复性极强的工作极大地影响了 X86 系统仿真效率,因此可以考虑在硬件上设置对应的影子寄存器来简化相关寄存器的保存和恢复工作,如图 2。

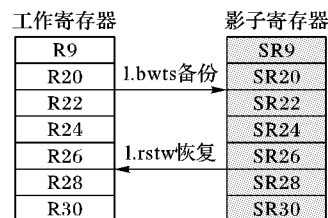


图 2 影子寄存器与自定义指令

在 OR1200 处理器源代码文件 or1200_cf.v 中设置对应于通用寄存器 R9/R20/R22/R24/R26/R28/R30 的影子寄存器 SR9/SR20/SR22/SR24/SR26/SR28/SR30 值,配合 2 条自定义指令 l.bwts (Backup from Working Registers to Shadow Registers)、l.rstw (Restore from Shadow Registers to Working Registers) 用于控制通用寄存器与影子寄存器之间的数据传输。则函数 g_tb_wrapper() 中的 14 条加载存储指令才能够完成的寄存器保存和恢复工作,就被简化后的 2 条自定义指令所取代,引擎切换的效率能够得到极大提升,如图 3。

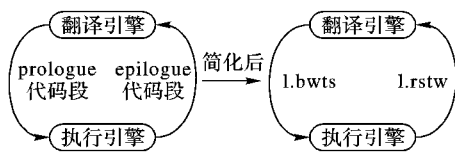


图3 简化后的引擎切换操作示意图

2.4 自定义指令

OR1200 处理器指令集设计之初预留了 8 个自定义指令槽,共有两种格式,如图 4,每个自定义指令预分配了一个操作码,范围为 0x1c~0x1f,0x3c~0x3f。指令可用与否取决于具体的实现。由于影子寄存器的备份与恢复设计无显式的输入与输出,所以选择图 4 中的格式 A。

格式 A	6位(操作码)		26位			
	31~26		25~0			
	0x1c~0x1f, 0x3d~0x3f		reserved			

格式 B	6位(操作码)	5位	5位	5位	6位	5位
	31~26	25~21	20~16	15~11	10~5	4~0
	0x3c	rD	rA	rB	L	K

图4 自定义指令的两种格式

修改 OR1200 处理器核心相关源代码文件,添加并指定指令 l.bwts 操作码 0x1c、机器码 0x70000000;l.rstw 操作码 0x1d、机器码 0x74000000,取代 g_tb_wrapper() 中的 prologue 和 epilogue,分别负责引擎切换时寄存器的备份与恢复。

OR1200 处理器的交叉编译工具链不支持这两条自定义指令,解决此问题的方法简便方法是:更改汇编器,手工编写机器码,内嵌机器码,指定自定义指令到二进制编码的映射,软件直接利用内嵌汇编来使用自定义指令。这样就可以直接跳过对编译器代码的修改,在汇编执行前插入指令。

3 测试与分析

3.1 测试环境

本文测试基于 OR1200 硬件平台的协同设计 X86 系统仿真框架原型 OpenRISC 架构协同设计体系结构桥 (Co-design ARCH-BRIDGE for OpenRISC, CoAB For OR),如图 5。

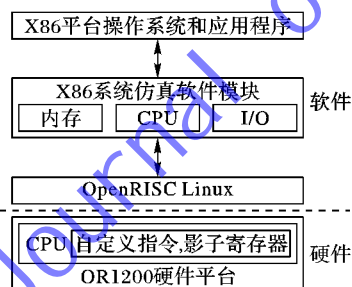


图5 CoAB For OR 原型整体结构

OpenRISC Linux 运行于 OR1200 硬件平台,内核版本号为 3.4.0。CoAB For OR 软件部分由 X86 系统仿真软件模块及运行在其上的 X86 平台操作系统和应用程序组成,其中 X86 系统仿真软件由 X86 处理器仿真、内存仿真、I/O 仿真三个子系统;硬件部分的改动主要集中在处理器内部的自定义指令和影子寄存器模块。

3.2 代码膨胀率

选取 Linux-0.2 操作系统的启动过程为测试集,输入代码量为 8000 个基本块,测试方法为利用 Co-AB For OR 文本日志(LOG)模块接口,设置在每一个基本块翻译完成后产生

翻译日志,记录源程序对应的目标指令信息到日志文件,最后通过对比计算得到代码膨胀率。如图 6,优化前后代码膨胀率分别为 10.52,8.22,膨胀率相对降低了 21.9%。

寄存器映射优化,使涉及到访问寄存器 EAX、EBX、EDX 和 ESP 的指令不需经过内存操作,而是直接使用固定映射到本地机器的寄存器 R10、R14、R16 和 R18,这样大量涉及到访问相关寄存器的指令翻译就可以得到简化,从而降低了翻译后目标代码膨胀率。

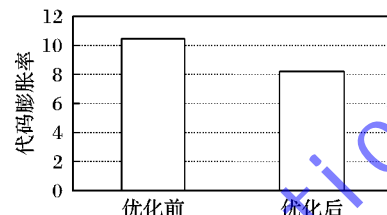


图6 代码膨胀率测试

3.3 性能分析

为测试优化后的系统仿真性能,选择在 CoAB For OR 上 Linux-0.2 操作系统启动和关机时间来对比观察优化后系统仿真性能的提升。设置 Linux-0.2 系统启动后马上自动关机,计算系统从开始启动到关机完毕的时间,测试结果如表 3 所示。寄存器映射优化使得翻译规则得到简化,翻译后目标代码规模下降;以影子寄存器和自定义指令为基础的引擎切换操作优化后,寄存器保存和恢复工作由原来的 14 条加载存储指令转变为 2 条自定义指令,引擎切换的效率能够得到极大提升。因此,减少了系统的启动关闭时间,系统仿真获得了较高的加速比。

表3 优化后的性能提升

测试项目	优化前/s	优化后/s	性能加速比
Linux-0.2 开关机时间	116.872	86.636	1.35

4 结语

本文深度结合基于静态寄存器分配的代码翻译机制,提出并实现了寄存器映射优化以降低翻译后代码膨胀率,自定义指令和影子寄存器相结合的协同优化以降低引擎切换操作复杂度。经测试,相比优化前,X86 仿真系统翻译后目标代码膨胀率降低了 21.9%,仿真系统获得了 1.35 的加速比。由于软硬协同优化设计方法与硬件耦合度较高,虽然取得了一定的性能提升,但降低了仿真系统的可移植性。

需要特别指出的是,本文的方法并不限于特定硬件平台,在国产自主处理器平台或其他开源平台上亦有很强的实践意义。

参考文献:

- [1] DONG W, WANG L, JIANG L, et al. Accelerate X86 system emulation with protection cache [J]. Computer Engineering and Design, 2013, 34(2): 606-610. (董卫宇, 王立新, 蒋烈辉, 等. 基于保护缓存的 X86 系统仿真优化[J]. 计算机工程与设计, 2013, 34(2): 606-610.)
- [2] HU W, WANG J, GAO X, et al. GODSON-3: a scalable multicore RISC processor with X86 emulation [J]. IEEE Micro, 2009, 29(2): 17-29.

(下转第 1422 页)

集地插入检查指令,本方法能更好地保证操作系统的实时性,而且错误覆盖率相比软加固前也有了明显提高。同时该方法可以通过改变所选加固函数的数目,达到更好地在性能与错误覆盖率之间做出权衡的目的。

操作系统的复杂性决定了需要多种软加固方法配合使用来提高错误覆盖率。在下一步工作中,拟研究一种异步周期检查的控制流错误检测方法,以进一步提高星载操作系统的可靠性。

参考文献:

- [1] XU J, QIAO Q, XIONG M, *et al.* Software fault-tolerance techniques for transient faults [J]. *Computer Engineering and Science*, 2011, 33(11): 132-139. (徐建军, 谭庆平, 熊荫乔, 等. 面向瞬态故障的软件容错技术[J]. *计算机工程与科学*, 2011, 33(11): 132-139.)
- [2] OH N, SHIRVANI P P, McCLUSKEY E J. Control-flow checking by software signatures [J]. *IEEE Transactions on Reliability*, 2002, 51(1): 111-122.
- [3] OH N, SHIRVANI P P, McCLUSKEY E J. Error detection by duplicated instructions in super-scalar processors [J]. *IEEE Transactions on Reliability*, 2002, 51(1): 63-75.
- [4] NICOLESCU B, IGNAT N, SAVARIA Y, *et al.* Analysis of real-time systems sensitivity to transient faults using microC kernel [J]. *IEEE Transactions on Nuclear Science*, 2006, 53(4): 1902-1909.
- [5] IGNAT N, NICOLESCU B, SAVARIA Y, *et al.* Soft-error classification and impact analysis on real-time operating systems [C]// DATE 2006: Proceedings of the 2006 Conference on Design, Automation and Test in Europe. Piscataway: IEEE, 2006: 182-187.
- [6] NEISHABURI M H, DANESHTALAB M, KAKOEE M R, *et al.* Improving robustness of Real-Time Operating Systems (RTOS) services related to soft-errors [C]// AICCSA 2007: Proceedings of the 2007 IEEE/ACS International Conference on Computer Systems and Applications. Piscataway: IEEE, 2007: 528-534.
- [7] YANG M, WANG H, ZHANG Y, *et al.* Design of radiation-hardened real-time operating system for pico-satellites [J]. *Journal of Zhejiang University: Engineering Science Edition*, 2011, 45(6): 1021-1026. (杨牧, 王昊, 张钰, 等. 抗辐射加固的皮卫星用实时操作系统设计[J]. *浙江大学学报: 工学版*, 2011, 45(6): 1021-1026.)
- [8] Texas Instruments. TI DSP/BIOS real-time operating system v6. x user's guide [DB/OL]. [2013-06-02]. http://downloads.ti.com/dsp/dsp_public_sw/sdo_sb/targetcontent/sysbios/6_21_01_16/exports/docs/docs/Bios_User_Guide.pdf.
- [9] HARI S K S, ADVE S V, NAEIMI H. Low-cost program-level detectors for reducing silent data corruptions [C]// DSN 2012: Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks. Piscataway: IEEE, 2012: 1-12.
- [10] XU J, TAN Q, LI J, *et al.* An extendable control flow checking method based on formatted signatures [J]. *Journal of Computer Research and Development*, 2011, 54(4): 638-646. (徐建军, 谭庆平, 李建立, 等. 一种基于格式化标签的可扩展控制流检测方法[J]. *计算机研究与发展*, 2011, 54(4): 638-646.)
- [11] LI J, TAN Q, XU J. A software-implemented configurable control flow checking method [C]// PAAP 2010: Proceedings of the 2010 Third International Symposium on Parallel Architectures, Algorithms and Programming. Piscataway: IEEE, 2010: 199-205.
- [12] LI J, TAN Q, XU J. Reconstructing control flow graph for control flow checking [C]// PIC 2010: Proceedings of the 2010 IEEE International Conference on Progress in Informatics and Computing. Piscataway: IEEE, 2010: 527-531.

(上接第1407页)

- [3] CAI W, TAI Y, LIU Q, *et al.* Memory virtualization on MIPS architecture [J]. *Journal of Computer Research and Development*, 2013, 50(10): 2247-2252. (蔡万伟, 台运方, 刘奇, 等. 基于MIPS架构的内存虚拟化研究[J]. *计算机研究与发展*, 2013, 50(10): 2247-2252.)
- [4] BECKER M, BALDIN D, KUZNIK C, *et al.* XEMU: an efficient QEMU based binary mutation testing framework for embedded software [C]// EMSOFT '12: Proceedings of the Tenth ACM International Conference on Embedded Software. New York: ACM, 2012: 33-42.
- [5] LIANG A, GUAN H, LI Z. A research on register mapping strategies of QEMU [C]// Proceedings of the 2nd International Symposium on Intelligence Computation and Applications. Berlin: Springer, 2007: 168-172.
- [6] WEN Y, TANG D, QI F. Register mapping and register function cutting out implementation in binary translation [J]. *Journal of Software*, 2009, 20(S): 1-7. (文延华, 唐大国, 漆锋滨. 二进制翻译中的寄存器映射与剪裁的实现[J]. *软件学报*, 2009, 20(S): 1-7.)
- [7] CAI S, LIU Q, WANG J, *et al.* Optimization of binary translator based on GODSON CPU [J]. *Computer Engineering*, 2009, 35(7): 280-282. (蔡嵩松, 刘奇, 王剑, 等. 基于龙芯处理器的二进制编译器优化[J]. *计算机工程*, 2009, 35(7): 280-282.)
- [8] LIAO Y. Dynamic binary translation modeling and parallelization research [D]. Hefei: University of Science and Technology of China, 2013. (廖银. 动态二进制翻译建模及其并行化研究[D]. 合肥: 中国科学技术大学, 2013.)
- [9] DENG H. Reorganization and optimization of the backend code in dynamic binary translation [D]. Shanghai: Shanghai Jiao Tong University, 2011. (邓海鹏. 动态二进制翻译后端代码热路径的重组优化[D]. 上海: 上海交通大学, 2011.)
- [10] SUN T. Branch analysis and optimization in dynamic binary translation [D]. Shanghai: Shanghai Jiao Tong University, 2010. (孙廷韬. 动态二进制翻译中跳转分析与优化[D]. 上海: 上海交通大学, 2010.)
- [11] CALLEJA D. Linux 3.1 [EB/OL]. [2011-10-24]. http://kernelnewbies.org/Linux_3.1.
- [12] QEMU. Change Log/1.2 [EB/OL]. [2012-09-05]. <http://wiki.qemu.org/ChangeLog/1.2>.
- [13] RICHARD M S. Using the GNU compiler collection [EB/OL]. [2013-05-06]. <http://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc.pdf>.