第36卷 第16期 计 算 机 工 程 2010年8月
Vol.36 No.16 Computer Engineering August 2010
·开发研究与设计技术· 文章编号：1000—3428(2010)16—0246—03 文献标识码：A 中图分类号：TP311

基于二进制翻译的仿真器关键技术研究

张 激，李宁波
(华东计算技术研究所，上海 200233)

摘 要：介绍基于构件化的仿真器设计方法，比较动态二进制翻译和静态二进制翻译的区别，分析动态二进制翻译技术在仿真器开发中的应用，并对基于二进制动态翻译的仿真器的关键技术——指令集仿真、T-cache 缓存管理等问题进行分析，针对二进制动态翻译中的自修改代码翻译问题，给出具体的解决方案。

关键词：仿真器；二进制翻译；指令集仿真

Key Technology Research of Simulator Based on Binary Translation

ZHANG Ji, LI Ning-bo
(East-China Institute of Computer Technology, Shanghai 200233)

【Abstract】 This paper introduces the component-based simulator architecture design, and compares the difference between the dynamic binary translation and the static binary translation, analyses the dynamic binary translation technology used in the development of simulator and the key technology of the dynamic binary translation based simulator, such as instruction set simulation, T-cache management. Aiming at the difficult problem, self-modifying code, it promotes the specific solution.

【Key words】 simulator; binary translation; instruction set simulation

1 概述

随着集成电路技术以及通信技术的飞速发展，嵌入式系统的研究与开发已经成为当今计算机科学的一个重要分支。由于应用领域的特点，嵌入式系统研发往往依赖于特定的硬件环境。对硬件环境的过度绑定导致传统的系统研发模式具有明显的缺陷，即软硬件开发无法并行开展，造成系统开发周期过长，使得设计工作缺乏灵活性。

为解决上述问题，基于软件的仿真器已经成为嵌入式系统研发的重要手段之一。软件仿真器就是用计算机来模拟特定硬件系统的全部或部分外部特性和内部功能，实现对目标硬件的高度模拟，使得运行于仿真器上的应用如同运行于真实的硬件平台之上，以验证嵌入式系统行为模型的合理性和软硬件接口的有效性，提高了软硬件的并行开发能力，缩短了开发周期。在仿真器开发过程中，主要有解释器技术、二进制翻译技术等实现方式[1]。由于二进制动态翻译在指令集仿真上的灵活性和多平台支持特性，因此被广泛应用于仿真开发。

二进制翻译是指使用软件将一种体系结构的二进制代码翻译成另一种体系结构的二进制代码。原则上来说，在一台计算机上的所有运算都能通过二进制翻译技术在另一台计算机上模拟运行，因此，二进制翻译技术被广泛应用于仿真技术方面。本文结合二进制翻译技术对基于二进制翻译的仿真器关键技术：指令集仿真，自修改代码处理，微码缓存管理等展开讨论，并给出相应的实现方法。

2 二进制翻译

二进制翻译分为静态翻译和动态翻译。静态翻译是指先将目标机器上的二进制代码完全翻译成本地机器上的可执行程序，然后在本地运行。要静态发现一个程序的所有指令代码难度较大，且在程序运行过程中可能会涉及到动态处理的部分，如共享库、自修改代码等，导致了静态翻译开销巨大，而且还需要终端用户的支持，缺乏一定的透明性。

动态翻译是采用一边翻译一边执行的方法，源代码在被执行时才被翻译。动态翻译能够收集程序动态运行时的信息，有目的地翻译和优化目标机器代码，打破了一些静态二进制翻译的局限性，具有较好的透明性。

二进制翻译把一种指令集的二进制可执行代码翻译为可以在另一种指令集体系结构上的可执行二进制代码，实现了跨平台的指令之间的转换，屏蔽了因为指令集而造成的底层平台的异构性，实现了体系结构的虚拟化，因此，二进制翻译技术在仿真器研发中被广泛采用，为实现多平台仿真提供了技术解决方案。

3 仿真器的结构

传统仿真器通常只支持具体的处理器型号或同一系列处理器，而随着电子技术的发展，目前处理器的种类繁多，体系结构和指令集各不相同，越来越复杂的应用对外部设备的仿真也提出了新的需求，因此，在保证仿真器性能的前提下，如何快速适应各种体系结构和指令集的变化，避免重新开发新的仿真器，开发支持多平台和多外设的多目标仿真器成了仿真器研究的重点。

基于构件化的仿真器架构[2]能够很好地解决上述问题。所谓构件化，是采用面向对象方法，将所仿真的目标处理器的硬件结构以及指令集以模块的形式通过通用的接口集成至仿真系统。在实际应用中，由用户选择所模拟的处理器对应

基金项目：国家“核高基”重大专项基金资助项目(2009ZX01041-001-002)
作者简介：张 激(1966－)，男，高级工程师，主研方向：嵌入式系统软件；李宁波，助理工程师
收稿日期：2009-11-12 **E-mail：**ecizhang@sh163.net

的模块，后端模拟器负责将相关模块连接起来，基于插件构建的仿真器实际上是将各种处理器模拟所需的核心代码融合至一个集成环境中，是一个多处理器仿真器的合成版。基于构件化设计的仿真器结构如图 1 所示。

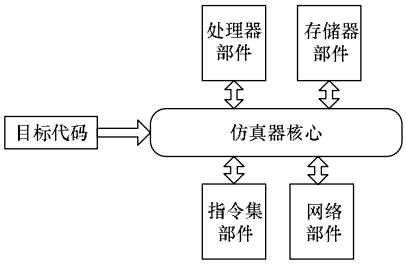

图 1　基于构件化设计的仿真器结构

应用构件化开发的仿真器对支持的处理器类型，模拟速度较快，准确度较高，另外通过提供标准接口的方式，开放仿真器部分接口，使得用户可以自行编写需要支持仿真器部件来拓展仿真器，降低了仿真器开发的难度，在很大程度上提高了仿真器开发的效率。

4　仿真器关键技术

4.1　指令集仿真

指令集仿真[3]是仿真器的核心部分，通过指令集仿真能够在本地机器上运行不同体系结构的应用程序。仿真器由控制核心、解释器、翻译器和翻译缓存等几部分组成。指令集仿真示例如图 2 所示。

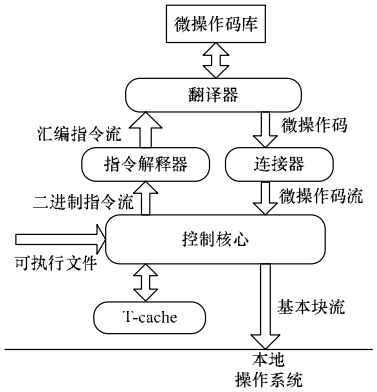

图 2　指令集仿真示例

在代码翻译过程中，首先由控制核心对进入仿真器的 ELF 文件进行加载，控制核心对 ELF 文件中各个段进行分析，并找到符号表和程序的入口地址，根据符号表和代码段数据，可以得到一个目标机器的二进制指令输入流。指令解释器将指令输入流中的每条指令分解为更小、更为简单的微操作[4]，每个微操作在仿真器微操作码库中都有一个唯一的微操作目标码与之相对应，微操作可由 C 代码描述，在仿真器编译过程中，通过编译器，所有微操作源码都被编译形成微操作目标码库，这样通过链接器可以将一系列的微操作码连接起来，将目标机器二进制指令流转换为支持本地机器指令的二进制指令流，指令翻译过程如图 3 所示。

通过微操作来翻译目标机器指令的方式虽然能够实现跨平台运行，但是微操作的引入会导致翻译后代码体积增大，尤其是对中间变量的操作会严重影响系统的仿真性能，因此，为了尽量减少系统性能的损失，在宿主机寄存器数量允许的情况下，可以将中间变量以寄存器变量的方式处理，以提高翻译代码的执行效率。

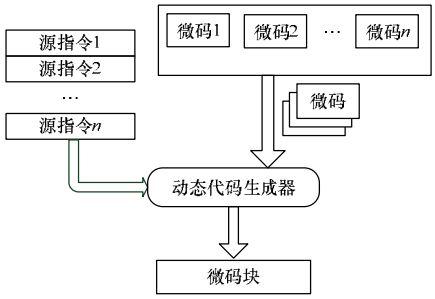

图 3　指令翻译过程

4.2　T-cache 管理

在动态翻译过程中，源指令到微操作的转换是一个匹配的过程，而在函数调用过程中，可能某块指令会被反复执行到，如果每条指令都是翻译出来后被立即执行，则代码重用度较低，势必影响仿真器的性能，因此，在源指令到微操作翻译的过程中可以以块为基本翻译单元，基本翻译单元是只有一个入口和出口的程序段，选择基本块做翻译单元可以省去很多有关函数调用的操作，比如堆栈的维护，同时可以充分挖掘基本块内的指令并行性。采用基本块作为翻译单元，动态二进制翻译器可以将翻译和优化的结果保存至 T-cache，下次遇到相同的代码段时，可以直接执行预存的翻译后代码。利用 T-cache 加速翻译效率的同时，需要对翻译出来的代码块进行有效的管理，使得既节省了本地内存空间，又不至于引起 T-cache 由于空间不足导致频繁的替换操作以致性能下降，对 T-cache 的管理可以采用以下策略：

(1)不替换原则：所有代码块都由始至终保存在 T-cache 中，每个代码块翻译只有在第一次会发生未命中，代码块在被翻译后，被翻译的微码块会被保存至 T-cache 供下次调用，该策略通过牺牲大量空间的方式来换取高速的优势，对于资源受限的系统来说，该策略不可取。

(2)LRU(Least-Recently-Used)策略：LRU 策略在发生 T-cache 满的情况下，选择将使用次数最少的微码块替换出去，该策略的优点在于考虑了程序的执行特性和程序的时间局限性，缺点在于实现复杂。

(3)FIFO(First-In-First-Out)策略：在 T-cache 中从地址底位到高位依次存放翻译好的微码块，当 T-cache 空间不足时，将最早进入 T-cache 的微码块换出，该策略实施简单，因而效率较高，同时也考虑了程序的阶段性特性，因为最早进入的微码块可能后面不会再被执行到。

(4)全清空策略：在全清空策略下，当 T-cache 空间不足时，T-cache 中所有的微码块都被清除，该策略最大的优点就是算法简单，实现起来容易，管理起来也较为方便，所以被很多二进制翻译系统所采用，但是该策略没有考虑到程序执行特性，对时间空间的局部性考虑不够，可能会导致很多较热的微码块被替换出去，许多块被重新导入导出，增加了不必要的开销。

以上介绍了几种 T-cache 微码块替换策略，在实际执行过程中，可以根据系统状况和执行目标采用合适的策略。

4.3　地址转换

在二进制动态翻译过程中，只有一个入口和出口的程序段被划分为一个翻译单元，翻译单元被翻译为一个微码块后，作为执行单元被送至宿主机上执行，在执行至微码块的结束时，一条跳转指令会将控制转向其他微码块，由于 T-cache 的空间局限性，有可能跳转指向的地址的微码块还没有被翻

译或更为复杂的情况是地址空间是不可达的，如映射至内存的 IO 空间，在这种情况下，为了保证执行过程的连续性，需要对跳转地址进行统一管理和控制。

为提高地址转换的效率，在微码块的执行过程中，可以创建一个地址映射表，用以保存源地址到翻译后地址的映射关系，源地址和翻译后地址以成对的形式保存在地址映射表中，为加快地址翻译的速度，可以按图 4 方式组织地址映射表的数据。在地址翻译时，通过源地址 tag 和 index 查找地址映射表，可以快速找到并转换为翻译后地址，然后跳转到相应的微码块继续执行。如果在地址映射表中没有找到相应的表项，表明翻译程序还没有翻译对应的微码块或该微码块已经被替换出 T-cache 队列，需要重新翻译该微码块，并继续后续的执行过程。

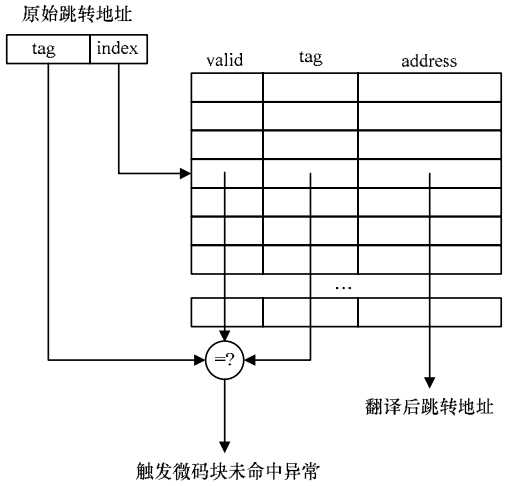

图 4　源地址和翻译地址的映射关系

在二进制翻译过程中，以上查询地址映射表的操作被添加到每个微码块最后的跳转指令之前，也就是在跳到其他微码块之前，需要通过地址翻译确定执行下一步所在的微码块的地址，并执行相关操作。为了进一步提高仿真速度，在支持 TLB 的 CPU 类型仿真过程中，可以将地址映射表以 TLB 表项的形式存放于 TLB cache 中，借助于硬件完成地址映射过程。

由于系统资源的有限性，地址映射表的大小受到了限制，为了保证地址转换的命中率，需要采取一定替换策略对地址映射表的表项进行管理，具体的策略和 T-cache 替换策略相同，两者需保持一致，在 T-cache 中微码块发生替换时，需要同步更新地址映射表中的表项。

4.4　自修改代码

自修改代码是在程序运行过程中向代码段写入数据，并且写入的数据作为指令执行，是程序运行期间修改和产生代码的一种机制。在嵌入式系统开发过程中，很多应用场景都是在资源非常受限的情况下进行，尤其是内存资源受限，当一些指令被执行结束后，可能不会再被执行，但其仍占据着部分内存空间，为了充分利用内存空间，可以采用代码自修改技术，将执行过的代码修改成新的指令，并执行完成相应程序功能。另外，代码自修改在增加代码的理解难度阻止逆向工程和软件保护方面也有着广泛的使用，如可以把自修改特性和 RSA 加密算法结合以用于软件版权保护。

虽然代码自修改情况不是很常见，但是在基于二进制翻译的仿真器设计中需要做特别处理，因为自修改代码在修改原有代码后，使得原有的程序代码发生改变，可能导致翻译好的微码块失效，如果 T-cache 和地址翻译 cache 不及时更新的话，将导致程序运行异常。要解决代码自修改问题，需要捕捉到发生代码自修改的时刻，并及时更新相关缓存数据，对这个问题，利用现在被广泛应用于虚拟地址映射和地址空间保护的 MMU 技术可以很好地解决，具体描述如下：

(1)修改被写保护的源代码将引发异常，翻译器得到通知；

(2)解除源二进制码的写保护；

(3)清空所有 T-cache 和地址翻译 cache 中的数据；

(4)修改源二进制代码；

(5)置修改后的源二进制码为写保护状态；

(6)翻译器继续翻译源代码块。

以上解决办法通过 MMU 的写保护机制，在向被写保护的页面执行写操作时，通过异常通知了翻译器，使得及时更新了 T-cache 和地址翻译 cache 的缓存数据，确保了在整个翻译过程的数据的一致性，保证了仿真过程的正常运行，不足的是在处理缓存数据时采用了全部清空的策略，影响了仿真性能。

5　结束语

本文研究了基于二进制动态翻译技术的仿真器，并对其关键技术和难点问题进行了分析。目前已有多种仿真器采用该技术实现，如仿真器软件 QEMU 和国内清华大学开发的开源仿真器软件 SkyEye 等，其中，QEMU 由于其开源性和支持多平台和多外设的仿真特性，应用最广。

参考文献

[1] 邓春梅. 嵌入式系统软件仿真技术的研究与实现[D]. 成都: 电子科技大学, 2004.

[2] 古幼鹏, 熊光泽, 桑　楠. 基于构建的嵌入式软件仿真开发环境模型研究[J]. 系统工程与电子技术, 2004, 26(10): 1495-1499.

[3] Bellard F. QEMU, A Fast and Portable Dynamic Translator[C]//Proc. of USENIX Annual Technical Conference. [S. l.]: ACM Press, 2005.

[4] 姜玲燕, 梁阿磊, 管海兵. 动态二进制翻译的中间表示[J]. 计算机工程, 2009, 35(9): 283-284.

编辑　顾姣健

(上接第 245 页)

[4] Radulescu A, Dielissen J, Pestana S G, et al. An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-memory Abstraction, and Flexible Network Configuration[J]. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2005, 24(1): 4-17.

[5] Bhojwani P, Mahapatra R. Interfacing Cores with On-Chip Packet-switched Networks[C]//Proc. of the 16th International Conference on VLSI Design. [S. l.]: IEEE Computer Society, 2003: 382-387.

编辑　金胡考