

指令集仿真器的关键技术

付琳^{1*}, 胡锦¹, 梁利平²

(1. 湖南大学 物理与微电子科学学院, 长沙 410082; 2. 中国科学院 微电子研究所, 北京 100029)

(* 通信作者电子邮箱 mingtianwendy@sina.com)

摘要:为适应嵌入式系统开发中对指令集仿真器仿真速度的要求,提出一种改进的指令集仿真技术。该技术在现有的静态多核仿真器基础上引入指令预处理、动态译码缓存、多线程 C 函数生成和动态调度运行等技术,实现对仿真器性能的优化。该技术已成功应用于中国科学院微电子所自主研发的 IME-Diamond DSP 处理器的多核指令集仿真器 OPT-ISS 中。实际应用程序测试结果表明,该技术在仿真速度提升方面有明显效果。

关键词:多核;指令集仿真器;多线程;译码缓存;C 函数生成

中图分类号:TP368.1 **文献标志码:**A

Key techniques for fast instruction set simulator

FU Lin^{1*}, HU Jin¹, LIANG Liping²

(1. College of Physics and Microelectronics, Hunan University, Changsha Hunan 410082, China;

2. Institute of Microelectronics, Chinese Academy of Sciences, Beijing 100029, China)

Abstract: In order to adapt to the the requirement of the Instruction Set Simulator (ISS) simulation speed in embedded system development, an improved ISS technology was put forward. The technology introduced instruction preprocessing, dynamic decode cache structure, multi-thread C function generation and dynamic scheduling technique based on the existing static multi-core simulator to achieve the optimization of the simulator performance. This technique has been applied successfully in forming OPT-ISS, which is based on IME-Diamond multi-core DSP processor. The experimental results show that this technique improves the simulation speed indeed.

Key words: multi-core; Instruction Set Simulator (ISS); multi-thread; decoding cache; C function generation

0 引言

指令集仿真器 (Instruction Set Simulator, ISS) 是针对特定的处理器架构采用高级语言开发的软件模拟环境,具有以下特点:ISS 对嵌入式系统的基本结构部件进行了描述,如算术逻辑单元、高速缓存及地址转换缓存等,实际处理器上运行的程序在 ISS 上都能够模拟运行。故其不仅用于程序调试,而且还有助于实际硬件的开发和调试;ISS 一般采用高级编程语言开发,相较于相同复杂度的硬件开发周期短,修改维护方便^[1]。以上特点使得 ISS 在嵌入式系统设计前期体系结构规划、系统软件开发和硬件电路验证等工作中,扮演着相当重要的角色。

ISS 的基本仿真策略分为解释型和编译型两种^[2]。在此基础上又衍生出了新的 ISS 结构和功能,文献[3]在静态编译的基础上,将指令转换成虚拟机机器码执行,充分发掘本地机运行速度的优势。文献[4-5]均在译码时采用预解码方式降低译码开销,仿真初期将整个程序文件读入主存中,并将指令解码成执行单元可识别的中间代码,仿真运行时,执行单元只需要读取中间代码以实现仿真。文献[4]在执行时采用优化线索解释技术来提升 ISS 的速度,译码得到的中间代码中包含分派代码,指向指令模拟函数执行。文献[5-6]设计的

ISS 均具有调试功能,文献[6]的 ISS 内置可配置缓存和多种分支预测器模型,作为处理器结构规划的参考。文献[7]提出一种结合了编译型与解释型策略特点的通用即时指令集仿真器 (Just-In-Time Instruction Set Simulator, JIT-ISS) 技术,待仿真指令按照编译-执行的方式实现仿真,编译后的信息储存在缓存中,避免重复编译。文献[8]为提升 ISS 的仿真速度,将耗时较多的译码过程放在编译阶段处理,仿真阶段按照解释执行策略运行。文献[9]的全系统仿真器将用户级仿真器中常用的时序与功能分离的组织形式应用到设计中,并提出“值预测校验算法”解决存储一致性问题,支持包括硬盘、网卡、终端控制台等在内的计算机系统各个部件的精确模拟。文献[10-11]中均用到了底层虚拟机 (Low Level Virtual Machine, LLVM) 编译器后端实现对本代码的优化执行,借助 LLVM 的良好代码优化性,提升 ISS 的速度,其中文献[11]的 ISS 使用 ArchC 描述语言自动生成。

IME-Diamond DSP 指令集是中国科学院微电子所自主研发的 DSP 指令集,同时兼容 CPU 指令集,其中包括供通信和多媒体应用的 DSP 指令,涵盖装载/存储指令、算术/逻辑/移位指令、控制/转移指令、单指令多数据流 (Single Instruction Multiple Data, SIMD) 向量操作指令等,有效支持嵌入式系统开发。IME-Diamond DSP 处理器采用 IME-Diamond DSP 指令

收稿日期:2014-12-17;修回日期:2015-02-24。 基金项目:湖南省科技计划项目(2014GK3002, 2014GK3148)。

作者简介:付琳(1990-),女,湖南邵阳人,硕士研究生,主要研究方向:多核 DSP 验证技术; 胡锦(1963-),男,湖南长沙人,教授,主要研究方向:电子系统与专用集成电路; 梁利平(1969-),男,湖北汉川人,研究员,博士,主要研究方向:高性能数字信号处理器/微处理器、高速混合信号集成电路设计。

集系统,硬件实现为 4 核处理器,分为 1 个主核和 3 个从核,支持单核/多核工作,从核初始为休眠状态,需要时由主核通过指令激活。STA-ISS 是现有多核 ISS^[6],其工作方式是先对程序预解码产生中间代码,再对中间代码模拟 DSP 硬件的多核工作方式执行,实现对指令的仿真。

在借鉴了上述仿真策略及各种优化方法的基础上,提出一种引入指令预处理、动态译码缓存、多线程 C 函数生成和动态调度运行的多核 ISS 构建技术,该技术已成功应用于 IME-Diamond 四核 ISS——OPT-ISS 的搭建,OPT-ISS 采用 C/C++ 语言设计,只引用标准库函数。它是针对 IME-Diamond DSP 处理器体系结构设计的多核 ISS。实验结果表明,使用该技术构建的 OPT-ISS 性能较现有 STA-ISS 有明显提升。

1 总体设计思想

OPT-ISS 工作流程如图 1 所示。OPT-ISS 被划分为两个线程,图 1 上半部分是主线程,下半部分是子线程。ISS 一旦开始工作,主线程将指令段数据加载到 ISS 的模拟主存中,供主线程的译码单元和子线程的剖析模块读取,子线程被唤醒。主线程中的多核调度单元控制 4 个核的运行和状态更新(在未被主核唤醒前,从核处于休眠状态,唤醒后处于工作状态,工作完成切换回休眠状态)。子线程负责对程序进行预处理和 C 函数转换。子线程生成的动态链接库函数(Dynamic Link Library, DLL)供仿真器直接调用执行,相较于单条指令仿真方式具有更快的运行速度,是本文的优化重点。这种多线程并行处理方式不会在原有的动态仿真模式之上增加时间开销,只要子线程的转换足够快,主线程能够尽量多次地调用动态库函数,减少直接对指令的执行,从而提高整体的仿真速度。子线程的转换速度与程序的静态指令数成正比,时间开销小。

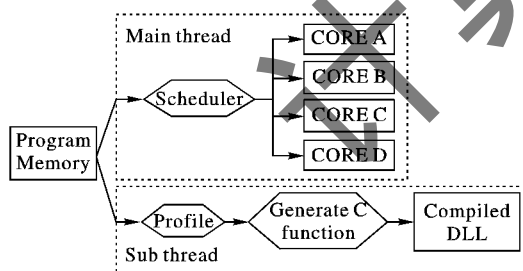


图 1 优化仿真器工作流程

2 ISS 优化关键技术及实现

根据程序的局部性原理,程序运行过程中 90% 的时间里执行其中 10% 的指令,即程序反复执行的就是一小部分重复的指令^[13]。此外,ISS 是在宿主机上实现仿真的,目标指令集通常不同于本地指令集,仿真时一条目标指令的实现需要多条本地指令参与方可完成,因此,减少程序中热点指令的重复译码和执行对 ISS 的性能提升将起到重要的作用。针对译码和执行,本文分别引入了动态译码缓存和多线程 C 函数调用两种优化。

2.1 动态译码缓存

动态仿真模式借鉴了文献[7]的方法,在解释型仿真中耗时较多的译码阶段引入译码缓存,译码缓存用 PC 值作为索引,缓存中保存 PC 值对应的指令操作码和操作数,称为中

间代码。ISS 取指后,先在译码缓存中寻找 PC 值,假如当前 PC 值出现在译码缓存中,ISS 将不再对该指令译码,直接将缓存中该 PC 值指向的中间代码传递给执行单元。否则说明当前 PC 值指向的指令没有出现或者缓存发生过溢出,将调用译码单元对当前指令进行译码,并将译码结果保存在译码缓存中。下次 ISS 再遇到该 PC 值指向的指令时,直接从缓存中提取中间代码给执行单元。这种一次译码、多次使用的方式,减少了译码过程中的时间开销。

2.2 多线程 C 函数调用

传统 ISS 的执行过程是执行单元获得对当前 PC 值指向的指令译码后的中间代码,调用执行函数进行处理,执行完成更新寄存器堆、内存和堆栈。每条指令都需要调用执行函数执行,对于大型程序,执行阶段的时间开销非常可观。OPT-ISS 中的多线程 C 函数调用技术参考文献[10]的指令块转换思想,但与其 LLVM 的中间转换结果不同的是,本文将指令块转换为 C 函数。参考文献[12]中的多线程任务分配方式,将转换与调用在不同线程进行,实现多个线程并行工作。ISS 工作初期,主线程动态地译码、执行,子线程剖析程序中的热点,按照指令转换规则对满足条件的热点指令块产生与其对应的 C 函数,编译 C 函数得到动态库函数。在下次遇到该热点时,主线程将动态得调用与该热点对应的 DLL,结合译码缓存,热点指令块中的指令无需再经过单条指令的译码和执行过程。

从程序运行复杂度的角度分析,如表 1 所示,假设程序文件中指令数为 n ,运行过程中执行的指令数为 N ,执行热点的指令数为 n' ,则有 $n < N$, $n' < N$ 。对解释型 ISS,译码的复杂度为 $O(N)$,执行的复杂度为 $O(N)$;对加入译码缓存的动态 ISS,译码的复杂度约为 $O(n)$ (假设译码缓存足够大),执行的复杂度为 $O(N)$;假设程序的热点指令块执行的指令数为 n' ,通过采用译码缓存和多线程 C 函数转换执行的优化方式,在译码缓存足够大、C 动态函数生成足够快的条件下,译码的复杂度为 $O(n)$,执行的复杂度为 $O(N - n')$,根据前文提到的程序局部性原理, $n'/N \approx 90\%$,假如能够找到程序中的热点加以改善,将大幅降低执行过程的复杂度,提升 ISS 的性能。

表 1 不同 ISS 的复杂度对比

ISS	译码复杂度	执行复杂度
解释型 ISS	$O(N)$	$O(N)$
加入译码缓存的解释型 ISS	$O(n)$	$O(N)$
OPT-ISS	$O(n)$	$O(N - n')$

综上所述,准确快速地确定程序中的热点并加以转换,是优化工作的重点及难点。由于实际仿真环境为多核 Linux 服务器,支持线程级并行,故引入子线程负责热点指令块的选择和转换,同时主线程负责整体的调度执行,从而实现 ISS 性能的优化。

2.2.1 子线程工作说明

将子线程按照功能划分成指令剖析、热点选取和函数生成三个阶段,如图 2 所示。将只能被调用而不包含对其他函数块调用的叶子函数视为热点指令块。子线程单独对程序中的机器码译码,判断得到的操作码和操作数是否符合叶子函数的条件,假如剖析直至指令块末尾都没有遇到违背条件的指令,则认为该指令块为热点指令块,可以进行转换,否则指

令块被舍弃,调整 PC 值指向内存中的下一个指令块。对于已剖析过确认为热点的指令块,指令的译码结果存放在名为 INS_BLOCK 的二维顺序容器中,交付给指令转换函数生成以指令块入口地址为函数名的 C 函数,并调用 system 函数实现对刚才生成的 C 函数的动态编译,得到 DLL。剖析、判断和转换以指令块为单位,子线程不断重复上述工作直至内存中指令段的末尾。

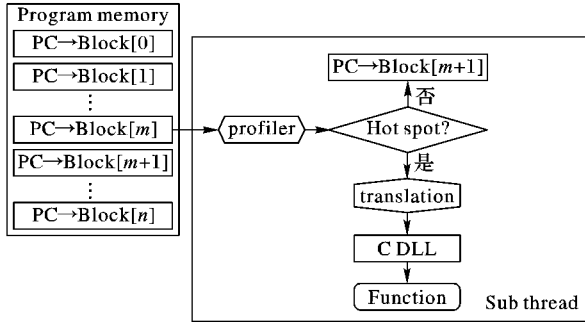


图2 子线程工作框图

1) 指令剖析阶段。

指令剖析阶段 (Instructions Profile Phase, IPP) 负责对内存中的指令块顺序进行剖析,将指令块中的指令顺序译码,译码结果交付给热点选取阶段,译码结果保存在容器 INS_BLOCK 中。

2) 热点选取阶段。

热点选取阶段 (Hot spot Selection Phase, HSP) 负责对 IPP 提供的结果进行分析判断,提取出热点指令块。与文献[12]不同的是,本文对热点指令块的选取遵循以下原则:

指令条数高于阈值(主线程动态调用时也会耗时,优化效果不明显),这个阈值可通过宏定义的方式调整;

指令块中不允许出现 JAL 和 JALR 指令(这两条指令都为非叶子函数的子程序调用);

不允许出现位于块中的 JR 指令,但指令块应以 JR 指令结尾(作为非叶子函数的调用返回);

对其他的条件或非条件跳转指令,其目的地址不能超出当前指令块长度范围。

具体操作时,HSP 顺序遍历每个指令块,一遇到违背上述条件的指令,就停止对当前指令块的判断,转至下个指令块,直至指令段末尾。

3) 函数生成阶段。

函数生成阶段 (Function Generation Phase, FGP) 负责将 HSP 判断出的热点指令块转换成 C 函数实现。FGP 中设计了一个查找表,查找索引为 IDP 生成的指令操作码,表项为每个指令单独的 C 实现,具体到代码中以宏定义形式实现,示例如下:

```
#define _LH_(rt, offset, base) { \
    unsigned int addr0, addr1, k, offset0; \
    ... \
    addr0 = (base + offset) >> 2; \
    addr1 = addr0 << 2; \
    offset0 = base + offset - addr1; \
    ... \
    else; }
```

转换时将 INS_BLOCK 中存储的热点指令块中指令的操作码作为查找表的输入,索引找到每条待转换指令对应的 C

实现,并结合该指令的当前实际操作数、PC 值等信息按一定规则写入到待生成的 C 函数文件中。最终生成的 C 函数形式如下:

```
... \
#include "ins_define.h" \
int _8017271c_(struct func_var var) \
{ \
    int t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, a0, a1, a2, a3, v0, v1; \
    bool branch = false; \
    a0 = * var. param0; \
    ... \
    pc8017271c: _ADDU_(t1, a2, zero) \
    pc80172720: _SUBU_(a0, zero, t2) \
    pc80172724: _SLTU_(a2, a1, a2) \
    ... \
}
```

程序中的“ins_define.h”头文件是保存指令宏定义实现的文件,该头文件涵盖了 TME-Diamond DSP 指令集系统中的全部指令。生成的转换函数具有以下特征:

a) 函数名为指令块的入口地址(十六进制)加左右下划线。

b) 输入变量为主线程提供的结构体 func_var,该结构体包括分别指向寄存器和指向数据内存的指针变量。

c) 函数中的局部变量 a0~a3, v0~v1 分别代表指令集定义的函数输入变量寄存器和返回值寄存器, t0~t9 为指令集中定义的临时寄存器。布尔变量 branch 用于条件跳转的判断,条件为真时 branch 为真,否则为假。

d) 为实现跳转指令的转移,每条指令前都有一个以“pc”开头,后接 8 位十六进制 PC 值的标号。

e) 标号后接带实际指令操作数的宏调用。

值得说明的是指令集中有不同类型的跳转指令,在其实现上需要分别对待。按寻址方式不同将其分为绝对寻址指令和相对寻址指令:绝对寻址指令细分为直接跳转指令和间接跳转指令;相对寻址指令细分为普通分支指令和可能分支指令,可能分支指令的延时槽指令在条件不成立时将不被执行,其目的是提高处理器的运行速度,而普通分支指令的延时槽指令不论条件成立与否都会被执行。此外,跳转指令中还有一类特殊的跳转并链接指令,它们以“AL”结尾,在跳转到目的地址的同时,还要将函数的返回地址放在指定的寄存器(如 RA)中,本文将跳转指令划分为 5 种类型分别处理:条件分支(并链接)指令、可能分支(并链接)指令和绝对跳转指令。

条件分支(并链接)指令实现形式:

```
Branch_label: \
(ra = pc + 8) \
if(v0 != t7) branch = true; \
else branch = false; \
Delay_label: \
delay_slot \
if(branch) goto pcxxxxxxx;
```

其中“ra = pc + 8”这条命令只会出现在跳转并链接指令中。这类跳转指令的特点是先对条件进行判断,再执行延时槽内的指令,最后决定跳转与否。

可能分支(并链接)指令实现形式:

```
Branch_label: \
(ra = pc + 8)
```

```

if( v0 != t7)
{
    Delay_label:
    delay_slot
    goto pcxxxxxx;
}
else;

```

与条件分支指令不同,可能分支指令只在条件满足时执行延时槽指令,所以将延时槽标号连带目的地址的 goto 语句都放在条件判断为真的范围内,条件为假时不执行任何操作。

绝对跳转指令实现形式:

```

B_label:
{
    Delay_label:
    delay_slot
}
J

```

因为无需进行条件判断,绝对跳转指令的形式比较简单,将延时槽标号和跳转都放在跳转标号范围内即可。

按照这种分类方式,实现了跳转指令的转换,最后产生的 C 函数清晰易懂,便于调试修改。经过以上 3 个阶段,热点 C 函数生成并被编译成动态库函数。

2.2.2 主线程工作说明

主线程中多核调度模块实时跟踪记录各个核的状态,对于处于工作状态的某核,其工作方式如图 3 所示。

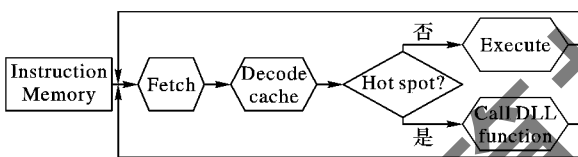


图3 单核工作示意图

任意单核内部均包括取指、译码缓存、模式调度、执行、动态调用等模块。每个核均具有两种工作模式——动态仿真模式和 C 函数执行模式,由各核私有的模式调度模块负责工作模式的选择。动态仿真模式在译码阶段引入译码缓存,避免重复译码。仿真初期,各个核按照动态仿真的译码-执行模式工作,当模式调度模块检测到存在与当前指令对应的 DLL 时,将停止动态仿真模式,转而进入 C 函数执行模式,调用 DLL 以指令块为单位动态地仿真,指令块中的指令不再经历译码和执行的过程,从而实现对 ISS 的优化。

3 测试结果分析

本文采用 H.264 数字视频编解码标准程序作为测试程序。H.264 是国际上通用的一种视频压缩格式。H.264 作为大型程序,在 ISS 环境下运行时耗时较长,将它作为 ISS 优化效果的评判标准,具有较强的针对性和可靠性。

图 4 和图 5 给出的是 ISS 在 Intel Xeon E5520 主频 2.27 GHz、内存 8 GB、Linux 系统环境下仿真 H.264 程序得到的测试结果。图 6 是 H.264 程序解码不同帧数图像时的仿真指令数。STA-ISS 是现有的 ISS,其特点是在解释型 ISS 基础上单独优化了译码方式,对程序一次性译码再执行,以减少译码的开销。在解码图像帧数较少时,STA-ISS 与 OPT-ISS 速度相近;随着程序的不断运行,OPT-ISS 的优势开始显现。解码 20 帧时,执行指令数达到 3.86×10^8 条,OPT-ISS 相比于

STA-ISS 速度提升了 20%,此后 OPT-ISS 一直保持着这样的加速比工作,平均仿真速度约为 7 MIPS (Million Instructions Per Second)。

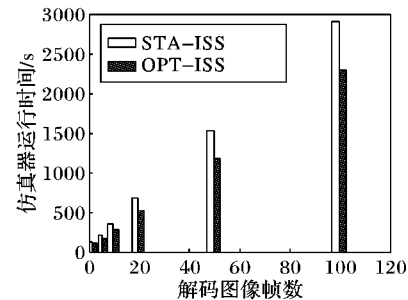


图4 ISS性能比较

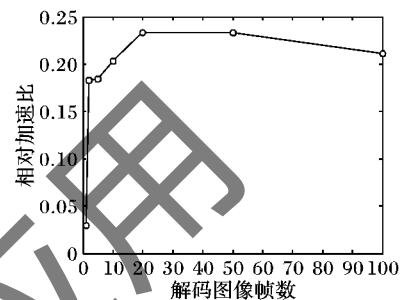


图5 ISS加速比

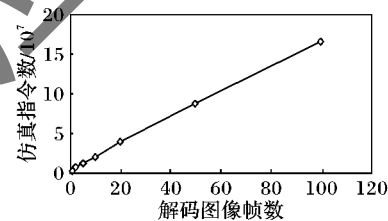


图6 仿真指令数

结果分析如下:

1) 解码图像帧数较少时, OPT-ISS 的优化效果不明显,这与 OPT-ISS 的设计机理相吻合。因为 OPT-ISS 的性能改善源于程序中的热点指令块执行方式改变,在程序运行的初期,还没有重复执行的指令块,即使存在热点指令块,其对应的 C 转换函数也可能尚未生成,此时 OPT-ISS 只能按照译码-执行的方式运行,STA-ISS 相比于动态 ISS 开销少,速度快。

2) 但是 OPT-ISS 的单一工作模式的过程很短,子线程的剖析是对静态指令顺序得处理,没有复杂的计算过程。测试结果表明,解码 50 帧时,程序中共 8.71×10^8 条指令,子线程约 22 秒完成对程序的剖析,此后主线程遇到热点指令块时都按照调用 DLL 的方式运行。

采用 Beyond Compare3 软件将 STA-ISS 和 OPT-ISS 二者运行后的解码结果进行对比,数据比较结果完全一致,采用视频播放软件播放解码后的视频时效果正常,表明 OPT-ISS 的仿真结果正确,上述测试数据真实可靠。

4 结语

本文提出了一种复杂度较低的指令集优化仿真技术。该技术充分地发掘了线程级并行和动态仿真的优势,并已经成功地在 IME-Diamond DSP 体系结构 ISS 上得到了实现。实际测试程序也证明该优化技术的有效性。在接下来的工作中,还将探索如何对更大更多的指令块进行转换,尝试进一步提

升ISS的性能。

参考文献:

- [1] YU Z, JIN H, ZOU N. Computer architecture software-based simulation [J]. *Journal of Software*, 2008, 19(4): 1051 – 1068. (喻之斌, 金海, 邹南海. 计算机体系结构软件模拟技术[J]. 软件学报, 2008, 19(4): 1051 – 1068.)
 - [2] WANG Y, WANG S. Optimization design and realization of IA instruction set simulator[J]. *Radio Engineering*, 38(11): 49 – 51. (王颖, 王赛宇. IA 指令集仿真器的优化设计与实现[J]. 无线电工程, 38(11): 49 – 51.)
 - [3] ZHU J, GAJSKI D D. An ultra-fast instruction set simulator[J]. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2002, 10(3): 363 – 373.
 - [4] GUAN X, CHEN Z, DANG J. Design of fast SPARC V7 instruction set simulator[J]. *Computer Engineering and Design*, 2011, 32(2): 531 – 534. (关小川, 陈朝晖, 党纪红. 快速 SPARCV7 指令集模拟器的设计方法[J]. 计算机工程与设计, 2011, 32(2): 531 – 534.)
 - [5] LUO H, LIANG L, YE T. Technique for multi-core instruction-set simulator[J]. *Application Research of Computers*, 2013, 30(10): 3035 – 3037. (罗汉青, 梁利平, 叶甜春. 一种多核指令集仿真器构建技术[J]. 计算机应用研究, 2013, 30(10): 3035 – 3037.)
 - [6] XUE B, ZHOU Y. Design of CPU simulator compatible with MIPS32 instruction set[J]. *Computer Engineering*, 2009, 35(1): 263 – 265. (薛勃, 周玉洁. MIPS32 指令集兼容的 CPU 模拟器设计[J]. 计算机工程, 2009, 35(1): 263 – 265.)
 - [7] BRAUN G, NOHL A, HOFFMANN A, *et al.* A universal technique for fast and flexible instruction-set architecture simulation[J]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004, 23(12): 1625 – 1639.
 - [8] RESHADI M, MISHRA P, DUTT N. Instruction set compiled simulation: a technique for fast and flexible instruction set simulation [C]// *Proceedings of the 40th Conference on Design Automation*. Piscataway: IEEE, 2003: 758 – 763.
 - [9] GAO X, ZHANG F, TANG Y, *et al.* SimOS-Goodson: a goodson-processor based multi-core full-system simulator[J]. *Journal of Software*, 2007, 18(4): 1047 – 1055. (高翔, 张福新, 汤彦, 等. 基于龙芯 CPU 的多核全系统模拟器 SimOS-Goodson[J]. 软件学报, 2007, 18(4): 1047 – 1055.)
 - [10] HELMSTETTER C, JOLOBOFF V, ZHOU X, *et al.* Fast instruction set simulation using LLVM-based dynamic translation[C]// *Proceedings of the 2011 International MultiConference of Engineers and Computer Scientists*. Berlin: Springer, 2011: 212 – 216.
 - [11] WAGSTAFF H, GOULD M, FRANKE B, *et al.* Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description[C]// *Proceedings of the 50th Annual Design Automation Conference*. New York: ACM, 2013: 21.
 - [12] BÖHM I, EDLER von KOCH T J K, KYLE S C, *et al.* Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator[C]// *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York: ACM, 2011: 74 – 85.
 - [13] HENNESSY J L, PATTERSON D A. Computer architecture: a quantitative approach[M]. 3rd ed. ZHENG W, TANG Z, WANG D, translated. Beijing: Publishing House of Electronics Industry, 2004: 31. (HENNESSY J L, PATTERSON D A. 计算机体系结构: 量化研究方法[M]. 3 版. 郑伟民, 汤志忠, 汪东升, 译. 北京: 电子工业出版社, 2004: 31.)
-
- (上接第 1420 页)
- [2] KIROVESKI D, MALVAR H. Robust spread-spectrum audio watermarking [C]// *Proceedings of the 2001 International Conference on Acoustics, Speech and Signal Processing*. Piscataway: IEEE, 2001: 1345 – 1348.
 - [3] WU S, HUANG J, HUANG D, *et al.* Efficiently self-synchronized audio watermarking for assured audio data transmission[J]. *IEEE Transactions on Broadcasting*, 2005, 51(1): 69 – 76.
 - [4] BHAT K V, SENGUPTA I, DAS A. An adaptive audio watermarking based on the singular value decomposition in the wavelet domain[J]. *Digital Signal Processing*, 2010, 20(6): 1547 – 1558.
 - [5] HUANG N E, SHEN Z, LONG S R, *et al.* The empirical mode decomposition and Hilbert spectrum for nonlinear and non-stationary time series analysis[J]. *Proceedings of the Royal Society of London: Series A*, 1998, 454(1971): 903 – 995.
 - [6] KHALDI K, BOUDRAA A O, TURKI M, *et al.* Audio encoding based on the empirical mode decomposition[C/OL]. [2014-06-20]. <http://www.eurasip.org/Proceedings/Eusipco/Eusipco2009/contents/papers/1569192283.pdf>.
 - [7] KHALDI K, BOUDRAA A O. On signals compression by EMD[J]. *Electronics Letters*, 2012, 48(21): 1329 – 1331.
 - [8] WANG L, EMMANUEL S, KANKANHALLI M S. EMD and psychoacoustic model based watermarking for audio [C]// *Proceedings of the 2010 International Conference on Multimedia and Expo*. Piscataway: IEEE, 2010: 1427 – 1432.
 - [9] ZAMAN A N K, KHALILULLAH K M I, ISLAM MD W, *et al.* A robust digital audio watermarking algorithm using empirical mode decomposition [C]// *Proceedings of the 2010 IEEE Canadian Conference on Electrical and Computer Engineering*. Piscataway: IEEE, 2010: 1 – 4.
 - [10] CHEN B, WORNELL G W. Quantization index modulation methods for digital watermarking and information embedding of multimedia[J]. *Journal of VLSI Signal Processing Systems*, 2001, 27(1/2): 7 – 33.
 - [11] LIE W, CHANG L. Robust and high quality time-domain audio watermarking based on low-frequency amplitude modification[J]. *IEEE Transactions on Multimedia*, 2006, 8(1): 46 – 59.
 - [12] HUANG X, JIANG W, WANG H, *et al.* Ratio-based digital audio blind watermarking algorithm in wavelet domain[J]. *Journal of the China Railway Society*, 2011, 33(5): 66 – 71. (黄雄华, 蒋伟贞, 王宏霞, 等. 基于比值的小波域数字音频盲水印算法[J]. 铁道学报, 2011, 33(5): 66 – 71.)
 - [13] MANSOUR M F, TEWFIK A H. Data embedding in audio using time-scale modification[J]. *IEEE Transactions on Speech Audio Processing*, 2005, 13(3): 432 – 440.
 - [14] LI W, XUE X, LU P. Localized audio watermarking technique robust against time-scale modification[J]. *IEEE Transactions on Multimedia*, 2006, 8(1): 60 – 69.