微处理器体系结构级测试程序自动生成技术*

朱 丹+, 李 暾, 郭 阳, 李思昆

(国防科学技术大学 计算机学院,湖南 长沙 410073)

Microprocessor Architectural Automatic Test Program Generation

ZHU Dan+, LI Tun, GUO Yang, LI Si-Kun

(School of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: danzhu@nudt.edu.cn, http://www.nudt.edu.cn

Received 2004-08-11; Accepted 2005-07-11

Zhu D, Li T, Guo Y, Li SK. Microprocessor architectural automatic test program generation. *Journal of Software*, 2005,16(12):2172−2180. DOI: 10.1360/jos162172

Abstract: In this paper, a novel specification driven and constraints solving based method to automatically generate test programs from simple to complex ones for advanced microprocessors is presented, and its prototype sytem——MA^{2}TG (microprocessor architectural automatic test program generator) is introduced. It can generate not only random test programs but also a sequence of instructions target to specific constraints. The proposed methodology makes three important contributions. First, it simplifies the microprocessor architecture modeling and eases adoption of architecture modification via Architecture Description Language (ADL) specification. Second, it generates test programs for specific constraints utilizing the power of state-of-art constraints solving techniques. Finally, the size of the test program for microprocessor verification and the verification time are dramatically reduced. MA^{2}TG has been applied on DLX processor and the embedded microprocessor EStar to illustrate the usefulness of the approach.

Key words: architecture description language; CSP; ITL; test program generation

摘 要: 提出了一种由体系结构描述驱动的基于约束求解的微处理器体系结构级测试程序自动生成的新方法,并基于此开发了原型系统——MA^{2}TG(microprocessor architectural automatic test program generator).该系统不仅可以随机生成测试程序,最主要的是可以产生针对特定要求的测试程序.其优点在于:首先,通过体系结构语言描述简化了体系结构建模,方便了对目标处理器体系结构的探索;第二,利用比较成熟的约束求解技术来生成满足需求的测试程序;第三,极大地缩减了测试程序的大小以及微处理器的验证时间.MA^{2}TG 已应用于 DLX 处理器和自主开发的 EStar 嵌入式微处理器的验证.实验结果表明了此方法的有效性.

关键词: 体系结构描述语言;约束满足问题;指令模板库;测试程序生成

* Supported by the National Natural Science Foundation of China under Grant Nos.90207019, 60403048, 60573173 (国家自然科学基金)

作者简介: 朱丹(1980−),女,重庆人,博士生,主要研究领域为微处理器验证;李暾(1974−),男,博士,讲师,主要研究领域为并行模拟、微处理器验证,电子 CAD;郭阳(1971−),男,博士,副教授,CCF 高级会员,主要研究领域为微处理器验证,电子 CAD;李思昆(1941−),教授,博士生导师,CCF 高级会员,主要研究领域为电子 CAD,VLSI 设计方法学,虚拟现实.

中图法分类号: TP302 文献标识码: A

随着微处理器复杂度的不断增长,验证已成为微处理器设计的瓶颈[1].目前,体系结构级验证主要有形式化验证和模拟验证两种方法.形式化验证存在状态空间爆炸问题,能处理的设计规模较小,无法处理整个微处理器的设计.模拟验证仍然是主要手段[2],它通过在验证对象上模拟大量测试程序来验证尽可能多的体系结构功能.在模拟验证中,测试程序的生成和质量是该方法的瓶颈.完全依靠手工编写所有的测试程序非常耗时,而且缺乏质量保证,因此有必要研究自动测试生成方法来提高微处理器模拟验证的效率和质量.

目前,国际上的测试程序自动生成器主要分为 3 类:多处理器测试生成器(MPTG)[3]、基于模型的测试程序生成器(MBTG)[4]以及体系结构验证程序生成器(AVPGEN)[5]. MPTG 利用 cache 一致性协议作为抽象机模型,使测试说明能够直接控制促使 cache 事件发生的特定测试序列的生成.MPTG 对于 cache 的验证非常有效,但其验证范围仅局限于 cache,在实际使用中,必须与其他随机测试生成工具(例如 MBTG)结合使用.MBTG 采用专家系统技术开发测试生成器,采用 C++语言为指令建立语义建模,其建模工作复杂,需要的时间长,例如,为 DSP 建模就需要 3~4 个人月[6],而且由于 MBTG 针对独立指令进行验证[7],很难生成验证某些功能块的特定指令序列.AVPGEN 采用符号执行、约束求解等技术来生成测试程序,但它为用户提供的控制测试向量生成的方法过分详细,难以维护.此外,Aharon 等人开发的随机测试向量生成器(RTPG)[8]采用 bias 技术来创建能够为设计正确性提供高可信度的测试子集.Lieh-Ming Wu[9]等人提出了一种基于 BNF 的测试程序生成方法.该方法利用用户菜单控制测试程序生成过程,可以生成随机的和指定的测试程序,但是它不能保证一定生成满足用户约束(例如数据冲突约束)的测试程序,本质上仍是随机生成方法.Prabhat Mishra 等人[10]提出一种 ADL 描述驱动的生成方法,该方法仅限于流水线的验证.

本文针对微处理器的体系结构级验证提出了一种由描述驱动的基于约束求解[11]的测试程序自动生成方法.该方法将微处理器的测试程序生成建模为约束满足问题(CSP),不仅能够随机生成测试程序,还能够生成满足用户特定需求的测试程序.其中,目标微处理器的体系结构采用我们自己设计的一种专门面向验证的体系结构描述语言(ADL)——VADL(verification ADL)描述,用户的验证需求文件采用我们自主定义的约束描述语言书写,测试程序的约束求解选用一般的约束求解库——EFC 库[12]完成.目前,原型系统——MA^{2}TG (microprocessor architectural automatic test program generator)系统已经实现.灵活的体系结构建模技术、独立于体系结构的约束描述语言以及强有力的约束求解技术使得本文的方法能够为具有 RISC,DSP,ASIP 以及 VLIW 等多种体系结构的处理器生成测试程序.目前,MA^{2}TG 系统已经初步应用于 DLX 体系结构的处理器和 LEON2 处理器模型的验证,运行结果表明了本文方法的有效性.

本文的贡献在于:第 1,为微处理器的体系结构级验证提出了一种有效的测试程序自动生成方法;第 2,专门面向验证的 ADL——VADL 及其编译器的采用能够高效支持体系结构建模,而且,由于基于 ADL 描述的设计方法易于设计空间探索(DSE),随着体系结构的发展和修改,该方法能够灵活地完成微处理器的体系结构建模.第 3,将测试程序生成问题建模为 CSP 使得我们的工具不仅能够随机生成测试程序,更重要的是能够生成满足特定验证需求的测试程序,从而提高了测试程序质量,缩短了验证时间.

1 MA^{2}TG 系统概述

图 1 给出了 MA^{2}TG 系统的框架结构及工作流程.MA^{2}TG 系统的设计原则是使得普遍微处理器的体系结构验证知识与特定体系结构验证知识相分离,约束编译器负责完成普遍微处理器体系结构验证问题测试生成的约束建模工作,指令模板库(ITL)提供特定体系结构测试生成的约束模型.测试程序生成问题的完整约束模型由约束描述文件的内容决定,约束描述文件是验证工程师用于描述验证计划的文件.

MA^{2}TG 系统由 ADL 编译器与约束编译器两部分组成,约束编译器是核心.为了生成满足特定需求的测试程序,系统首先根据目标处理器的 ADL 描述生成指令模板库(ITL)和体系结构特征配置文件(ACF),ITL 包含了根据特定体系结构验证知识为测试程序建立的约束模型,ACF 包含了所有从体系结构描述文件中提取的用于

配置系统核心部件——约束编译器的体系结构特征信息.然后,约束编译器根据普遍微处理器的体系结构级验证知识为测试程序生成建立满足用户验证需求的约束模型.最后,来自约束编译器的约束模型与来自 ITL 的面向特定体系结构验证问题建立的约束模型共同构成了测试程序生成问题的总约束模型,该模型与约束求解库链接得到一个可执行程序,执行该程序即得到满足用户需求的适用于目标处理器的测试程序.

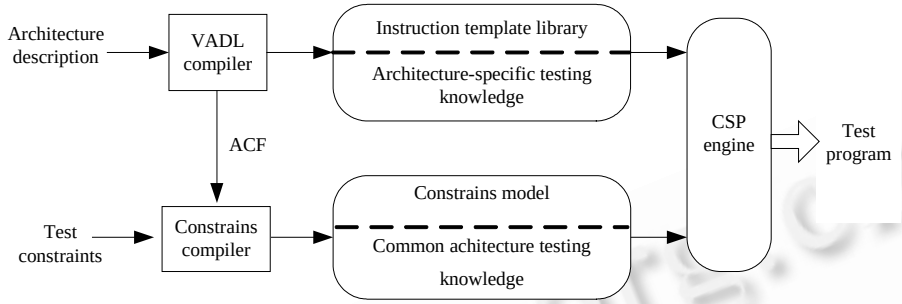

Fig.1 The framework and flow of MA^{2}TG

图 1 MA^{2}TG 系统框架及流程

框架采用的是自主设计的专门面向验证的 ADL——VADL.该语言以类 EXPRESSION[13]的语法来简化体系结构描述,增强可读性,不仅具有 EXPRESSION 建模的简单灵活性,而且还能够为验证提供足够多的体系结构信息.框架中的约束需求文件采用我们根据验证需求描述特点设计的约束描述语言描述,该语言自然、描述能力强,使得用户可以灵活地控制测试程序生成.框架采用的约束求解库是 EFC,库中集成了各种约束求解技术,并能够直接求解多元约束问题.

2 MA^{2}TG 系统

MA^{2}TG 系统采用 ADL 描述来捕获与验证相关的体系结构信息,并支持处理器体系结构模型的自动生成.这种基于 ADL 的建模方法支持体系结构模型和测试生成环境的快速开发与维护,并且使体系结构模型与测试生成引擎分离,让用户能够灵活地维护多个不同的设计.此外,还能为处理器的 DSE 过程提供验证支持.

2.1 VADL描述

现有的 ADL,如 MIMOLA[14],nML[15],LISA[16],EXPRESSION 等都是针对 SOC 设计提出的,对快速 DSE 非常有用,并且支持编译器和模拟器工具集的自动生成,但是都不适合面向验证的体系结构描述.

VADL 是针对 EXPRESSION 等 ADL 对体系结构级功能验证的支持不足而提出来的.通过对处理器-存储器系统的行为和结构进行混合描述,VADL 能够自动、高效地为目标微处理器建立与验证相关的体系结构模型,为测试程序生成捕获足够的体系结构信息.

VADL 的行为部分包括指令描述、初始化操作数的指令组合描述和基本指令集中典型指令与目标指令集中相应指令映射关系的描述.结构部分包括对存储子系统描述和流水线中可能出现的各种数据冲突的描述.VADL 通过自动生成的指令模板库和体系结构特征配置文件为约束编译器提供建立约束模型所需的体系结构固有约束.

在行为部分中,指令描述包括操作数类型定义、立即数类型说明和指令集描述 3 个子部分.指令集描述主要说明指令类型、指令语义、指令操作数、格式等与验证相关的属性.初始化操作数的指令组合描述部分专门为构造验证特定问题的测试程序提供初始化操作数的指令组合.约束编译器利用这些指令组合将目标指令的指定源操作数初始化为求解出的数值,从而使得生成的测试程序能够触发需要验证的体系结构级事务.指令映射部分可以帮助约束编译器将基本指令集中的主要指令映像到目标指令集中,以便于编译器根据指令语义挑选合适的指令构造测试程序.

在结构部分中,存储子系统描述支持对存储器子系统结构的显式描述,并且能够为各种具有新型结构的存储器系统、片上 DRAM、帧缓冲、分离的存储器地址空间等提供支持.数据冲突描述部分为约束编译器产生能

够触发数据冲突的测试程序段提供了必要支持.

VADL 只要求描述与验证相关的体系结构信息,且为体系结构描述提供了一种自然、统一的描述机制,从而简化了微处理器的体系结构建模工作,减少了模型描述出错的概率,有利于提高处理器的验证效率.由于 VADL 支持与各种存储器系统组织和层次相耦合的 RISC,DSP,ASIP 以及 VLIW 等多种体系结构的微处理器的描述,极大地拓展了 MA^{2}TG 的应用范围.

2.2 指令模板库(ITL)

ITL 由 ADL 编译器自动产生.编译器根据体系结构信息在 ITL 库中建立 3 种类型的约束模型:指令模型、指令组合模型以及验证问题模型.这些模型都以 C++类的形式出现.目标指令集的每条指令在 ITL 中都对应一个指令类.指令类中包含了指令的语义信息、指令的体系结构约束、如何引发与该指令相关的异常的 C++方法以及指令的打印方法.指令的语义信息包括指令名、操作数类型.这里给出了 ITL 中 DLX 的加法指令引发上溢的约束方法的例子:

```
bool Add::overflow(int rs1_in, int rs2_in) const{
return rs1_in+rs2_in > 0xFFFFFFFF;
}
```

在约束求解的过程中,如果用户要求验证加法指令的上溢,那么该约束将作为求解测试程序的约束条件之一.ITL 除了为单条指令建立相应的类以外,还会为复杂的验证问题建立单独的类,例如:读后写(WAR)等数据相关、存储器地址对准、cache 的验证等.这些类中包含了引发相应验证问题的约束网络.这里给出了存储器类中产生地址未对准异常的约束方法:

```
bool Mem::misalign(int memaddr,int aligntype) const{
return memaddr MOD (memaddr,aligntype)!=0;
}
```

其中,aligntype 表示访存指令的对准类型的字节数(例如:字对准指令的 aligntype 取 4,半字对准指令取 2).

ITL 可以保证测试程序的有效性,库中所包含的约束网络相当于验证的专家知识,可以提高验证者书写验证需求的思考层次,达到降低验证复杂度、提高验证效率的目的.此外,由于 ITL 是 ADL 编译器自动产生的,还能将验证人员从繁杂的指令语义建模工作中解脱.用户在验证时,只需用约束语言进行简单描述.

2.3 体系结构特性配置文件(ACF)

MA^{2}TG 系统的设计目标是能够为各种体系结构生成测试程序,这就要求它的核心部分(约束编译器)必须可配置.约束编译器的主要任务是根据约束需求为目标微处理器建立测试程序的约束求解模型.因此,约束编译器可配置是指编译器可以根据目标体系结构特征生成能够在具有该体系结构的微处理器上运行的测试程序.ACF 的作用正是向约束编译器提供体系结构特征.与 ITL 一样,ACF 是 MA^{2}TG 系统中连接体系结构模型与约束编译器的纽带.

ACF 对指令属性的描述按照指令类型(例如 ALU 类型、访存类型等)来组织,以便用户在验证某个指令组合的属性时可以随机选择其中的指令.每条指令的属性包括它所影响的标志位、操作个数和类型、指令的语义信息.对于跳转类指令还会给出用于决策跳转与否的标志位信息.ACF 还对子程序调用与定义类指令、跳转类指令和标号、访问堆栈类指令进行了特殊标识,以便于约束编译器为保证测试程序合法性进行特殊处理.此外,ACF 中还可以找到操作数类型的定义,存储器地址范围、立即数类型及范围.这些都是为了保证求解产生的测试程序的合法性.在产生测试程序的过程中,当要构造复杂验证问题(例如:异常)时,需要指令或者指令组合中的寄存器操作数满足约束条件,因此 ACF 还提供了一组可以将寄存器初始化为任意合法值的指令组合.例如:c 以下给出的验证 DLX 中 ADDUI 指令的测试程序中,前面的 3 条指令是从 ACF 得到的用于初始化寄存器 R28 的指令组合.

```
SUB     R28   R1     R1
LHI     R28   0xFFFF
```

```
ADDI    R28   R28    0xFFFF
ADDUI   R2    R28    45
```

2.4 约束编译

2.4.1 约束建模特点

用于体系结构级验证的测试程序应该满足两个需求:1) 有效,即程序行为和指令格式必须符合目标微处理器的体系结构定义;2) 高质量,即生成的测试程序应该尽可能多地覆盖约束需求和功能点,能够找出潜在错误.因此,一个理想的测试程序应该在满足约束的前提下尽可能随机产生.我们很自然地将这种有效的、满足验证需求的测试程序生成问题建模为 CSP.这就需要将体系结构级资源建模为 CSP 变量.这些资源包括寄存器单元、存储器单元等,它们的取值范围都相当大.例如,64 位寄存器的取值范围包括 2^{64} 个数值.因此,代表这些资源的 CSP 变量的定义域都非常庞大.大定义域的简单组合、线性约束以及非线性非单调的约束使得对这些定义域的存储、表示和操作都相当困难.因此,CSP 的建模和求解面临着严峻挑战.

此外,微处理器设计中存在的错误数目和触发这些错误所需测试场景的数目都是非常惊人的,依靠约束描述文件毫无冗余地精确定义恰好能够实现完全覆盖的测试程序是不可能的.因此,我们希望测试生成器在满足验证需求的前提下,尽可能覆盖更多的功能点.这就需要解空间随机、均匀地分布.这一点与传统的约束求解相悖,传统的 CSP 通常要求找到任意一个解、所有的解或者最好的解.

2.4.2 约束建模策略

典型的测试程序由一系列测试场景构成.从处理器运行的角度来看,每个场景都可以触发特定的体系结构级事务,即指令的执行和系统级交互(例如:cache 失效).在测试程序的 CSP 模型中,许多变量都可以看作这些事务的属性变量.大多数情况下,同一事务的内部变量间都存在大量约束,而事务间的约束并不太多.因此,测试生成问题的完整约束网络都由大量网间约束关系松散、内部约束关系相对紧密的小约束网络组成,每个小约束网络代表一个事务.

按照体系结构级事务,将测试生成问题分解成若干个子问题,每个子问题对应一个事务.从约束网络的角度来看,就是将测试生成问题的完整约束网络划分为体系结构级事务对应的小约束网络,然后为每个小约束网络建立约束模型.

下面以 cache 失效的约束建模为例,说明如何将局部约束转化为指令内部约束.假设主存大小为 M,n 路组相联 cache 的容量为 C,块大小为 B,不难求出 cache 的组数为 C/nB.因此,要引起 cache 失效,就要求测试程序中至少出现 $n+1$ 条访存指令,所有地址都必须映射到同一个 cache 组内,且任意两个地址不相等,即满足以下两个约束关系:

$$\begin{cases}(x\%B)\%(C/(n\times B))=(y\%B)\%(C/(n\times B))\\ x\neq y\end{cases},$$

其中,“%”为取模运算符,“/”为整除运算符.公式适用于各种相联策略的 cache,区别仅在 n 的取值.全相联 cache 的相联度为 cache 的块数(n=M/B),直接映射 cache 的相联度为 1(n=1),组相联映射 cache 的相联度等于路数.

根据上述分析,cache 失效事务的约束网络中至少包含 $n+1$ 个地址变量,约束关系较为复杂.为了简化约束求解,我们并不将这 $n+1$ 个变量的约束关系都建立到一个单独的网络中.根据 cahce 映射原理,上面的这种多条指令间的约束网络可以转化成单条指令内部的约束网络.假设要产生 u 条访存指令($u>n+1$),那么可以将存储器划分为 u 个相邻的均匀子空间,其中,前面的 $u-1$ 个空间都由连续的 M/u 个存储单元组成,最后一个子空间由 $M-M(u-1)/u$ 个存储单元组成.为了保证任意两个访存地址不相等,可以让 u 条指令的访存地址分别落在 u 个子空间内,并且让所有地址都映射到 cache 的第 g 组,而 g 的值可以由约束编译器随机产生,且 $0<g<C/nB$(C/nB 表示 cache 的组数).即让任意的第 k 条访存指令的地址 x_k 满足下面的条件:

$$\begin{cases}(x_k\%B)\%(C/(n\times B))=g\\ (k-1)M/u\le x_k\le kM/u-1\end{cases}.$$

为 cache 失效事务建立约束模型是分解局部约束网络,仅为单条指令建立约束模型的典型例子.在为体系

结构级事务建立约束模型的过程中,根据每个问题的特点,都可以找到分解指令间局部约束网络的方法.这种策略极大地克服了约束求解效率不高的问题.

2.4.3 特殊指令的处理

为了保证测试程序的合法性,约束编译器还会对特殊指令作分类预处理:

堆栈访问指令:PUSH/POP 指令的出现必须保证堆栈不会出现上溢和下溢.在 MA^{2}TG 中,编译器会约束 PUSH/POP 指令的出现次数匹配,以保证堆栈状态正常.

CALL 类型指令:当生成 CALL 类型指令时,必须要定义相关的子程序.编译器必须施加约束,保证子程序体中包含 RETURN 指令.此外,还要考虑子程序对测试程序大小的影响.假设在约束文件中说明程序的指令总数为 S,且 $x<S<y$,并且编译器已经知道子程序调用指令前面将产生 n 条指令,后面还有至少 m 条指令未处理,则子程序指令条数 K 必须满足 $x<n+K+m<y$,即 $x-(m+n)<K<y-(m+n)$.

转移类指令:图 2 中的箭头表明了转移方向的定义.无论转移方向如何,编译器都会首先在目标语句前添加标号,然后采用资源锁定的方法来避免测试程序中出现死循环,具体步骤如下:

1) 约束编译器从 VADL 的初始化寄存器的指令组合描述部分提供的指令组合中随机挑选一组指令来初始化分支条件,并将该组合插入到与标号语句相邻的前一位置.

2) 锁定分支条件,不允许循环体中任何指令修改它.

3) 从 VADL 的指令映射部分挑选一条指令插入循环体,该指令必须能够影响分支条件所依赖的数值,并使得该数值经过若干次循环修改之后能够导致分支失败,跳出循环.

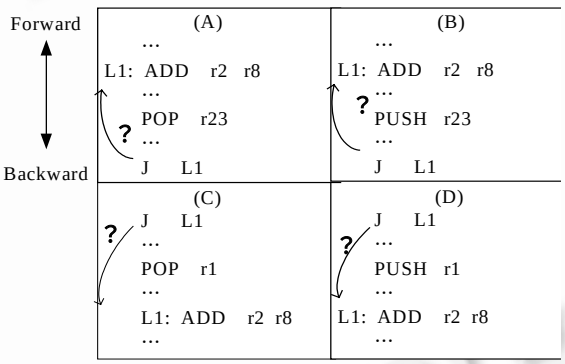

Fig.2 Several cases for jump destination

图 2 跳转目标的几种情况

在处理转移类指令时,需要考虑如图 2 所示的 4 种出现堆栈类指令的特殊情况.图中指令 J 代表转移类指令.在(A)中,指令 J 将使得程序的执行跨过它前面的某条弹栈指令,这可能会造成堆栈下溢.同理,(B)的情况可能导致堆栈上溢.下面所给出的说明该算法能够找到合适的标号位置,以避免程序不合法.(C)和(D)的情况可以采用前面提到的保证堆栈指令合法性的策略进行处理.

```
int procedure label_location(){
  do{
     randomly select a location before j, suppose is K;
     if(|numberof(PUSH)-numberof(POP)|<=STACK_size){
        found the legal location for label instruction as jump destination;
     }
  }until (find legal location);
  return(K);
}
```

2.5 实例分析

以 MA²TG 系统验证 DLX 处理器为例,目的是验证 DLX 指令集中每条指令、每种异常、数据冲突、控制相关、cache 失效处理的正确性.我们在约束描述文件中要求 MA²TG 系统尽可能选择 ADD/SUB 指令来构造用于验证 DLX 体系结构级功能的测试程序.表 1 列出了 MA²TG 系统为 DLX 的每项功能验证所生成的测试程序段指令数,表项名称对应体系结构级功能名.

Table 1 Instruction number for verifying each architecture feature of DLX processor

表 **1** 验证 DLX 的每个体系结构特征的指令数

Instruction set	Control hazard	Data hazard			ALU exception	Memory hierarchy	
		WAW	WAR	RAW		Memory misalignment	WAW
54	+0	0	0	2	37	2	3

表中第 1 项表示 MA²TG 系统为验证指令集中 54 条非浮点指令功能而生成的测试程序段指令数.由于验证指令集功能的测试程序段包含了转移类指令,能同时验证控制相关,因此,表中第 2 项取值“+0”表示不需要专门验证控制相关的测试程序段.由于 DLX 流水线中不存在 WAW 相关和 WAR 相关,所以不需要为这两种数据冲突的验证生成测试程序.

图 3 给出了部分由 MA²TG 自动生成的测试程序.

```
1   ADD     R1,  R23,  R3        //verify RAW
2   SUB     R4,  R5,   R1

3   SUB     R14,  R1, R1
4   LHI     R14,  0xFFFF
5   ADDUI   R14,  R14,  0x0078
6   SUB     R15,  R6, R6
7   LHI     R15,  0xFFFF
8   ADDUI   R15,  R15, 0xFFFF
9   ADDU    R2,   R14, R15       //verify the overflow of ADDU

10  SUB     R28,  R1,  R1
11  LHI     R28,  0xFFFF
12  ADDUI   R28,  R28,  0xFFFF
13  ADDUI   R2,   R28,  45       //verify the overflow of ADDUI

14  SUB     R13,  R7,  R7;
15  LHI     R13,  0x7FFF
16 ADDUI    R13,  R13,  0xFFFE;
17  SUB     R1,   R4, R4;
18  LHI     R1,   0x0000;
19  ADDUI   R1,   R1, 0x0005;
20  ADD     R3,   R13,R1;        //verify the overflow of ADD

21  SUB     R14,  R10, R10;
22  LHI     R14, 0x7FFF
23  ADDUI   R14, R14, 0xFFFE;
24  ADDI    R4,   R14, 3;        //verify the overflow of ADDI

25  SUB     R4,   R13, R13
26  ADDUI   R15, R4, 89
27  SUBU    R2,   R4, R15        //verify the underflow of  SUBU

28  SUB     R14, R25,  R25
29  SUBUI   R2,   R14,  11       //verify the underflow of  SUBUI

30  SUB     R16,  R7,  R7;
31  ADDUI   R15,  R16, 67;
32  LHI     R16,  0xFFFF;
33  ADDUI   R16,  R16, 0xFFFF;
34  SUB     R3,   R16,  R15      //verify the underflow of  SUB

35  SUB     R4,  R23,  R23;
36  LHI     R4,  0xFFFF;
37  ADDUI   R4,  R4,   0xFFFF;
38  SUBI    R3,  R4,   13；      //verify the underflow of  SUBI

39  SUB     R15,  R1,  R1
40  SW      25(R15),  R6         //verify the memory misalignment

41  SUB     R2,   R1,  R1；
42  LW      R16,  820 (R2)
43  LW      R16,  1140 (R2)      //verify the Dcache
```

Fig.3 Part of test program from our tool

图 3 部分由工具产生的测试程序

第 1、2 行的 ADD 和 SUB 指令对是工具为验证 RAW 相关构造的测试程序段,其中,ADD 指令的目的操作数与 SUB 指令的源操作数 2 选用了相同的寄存器 R1.所以表 1 的“RAW”子项取值为 2.第 3~38 行列出了工具为验证非浮点算术逻辑运算异常(例如上溢和除数为零异常)的测试程序.验证不同种算术异常的程序段由空行隔开,每段的最后一条指令是引发异常的核心指令,前面的指令作用在于初始化核心指令的寄存器操作数.为了

验证存储器地址未对准异常,工具生成了第 39、40 行的两条指令.前一条 SUB 清零 R15,使得后一条指令 SW 的访存地址为 25,这与 SW 的字对准原则相悖,所以必然引起存储器地址未对准异常.因此表 1 的“存储器地址未对准异常”项取值为 2.

3 实验及结果

国际上大多数已有的测试程序生成器本质上都是随机测试生成系统,为了将 MA^{2}TG 与已有系统进行比较,我们自己开发了一个随机测试生成器 RTG.在为 DLX 处理器生成测试程序的实验中,作了两类比较:MA^{2}TG 与 RTG 生成测试程序的覆盖率比较,带覆盖率反馈的测试生成与不带反馈的测试生成进行比较.

图 4 给出了本文方法与随机生成方法的覆盖率随测试程序数目增长的曲线图.在实验中,随机生成方法总共生成 100 个平均大小为 400 行的测试程序,而 MA^{2}TG 系统总共生成 60 个平均大小为 100 行的测试程序.如图所示,MA^{2}TG 系统的覆盖率随着测试程序数目的增长而迅速增长,50 个程序已经使覆盖率达 0.95.然而,随机生成方法的覆盖率曲线显示其覆盖率随着测试程序数目的增长而变得缓慢,100 个测试程序的覆盖率仅达 0.39.这是因为随机生成方法缺乏指导,它可能反复为已经覆盖过的指令、指令组合以及体系结构级事务生成测试程序.而在本文方法中,用户可以根据覆盖率报告提供约束描述文件,有针对性地指导 MA^{2}TG 系统为未覆盖的指令、指令组合和体系结构级事务生成测试程序,使覆盖率迅速增长,避免了冗余测试程序的产生.

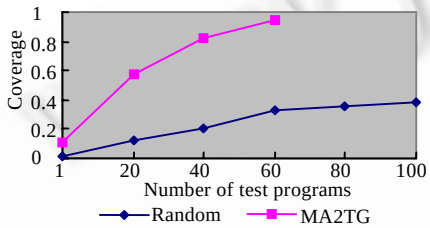

Fig.4 Coverage for random and MA^{2}TG

图 4 RTG 与 MA^{2}TG 的覆盖率曲线

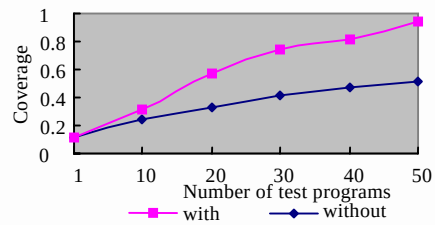

Fig.5 Coverage with and without feedback

图 5 无覆盖率反馈方法与有反馈方法的覆盖率曲线

图 5 给出了本文方法分别在有、无覆盖率反馈两种情况下生成的测试程序覆盖率随测试程序数目增长的曲线.如图所示,测试程序数目超过 30 以后,无覆盖率反馈的 MA^{2}TG 系统生成的测试程序覆盖率曲线逐渐趋于平缓,50 个测试程序的覆盖率仅 0.51.而在有反馈的情况下,50 个测试程序已经使覆盖率达 0.95.因此,覆盖率反馈能够有效提高 MA^{2}TG 系统生成的测试程序对目标微处理器功能的覆盖率.

实验表明,在本文方法的原型系统 MA^{2}TG 中,约束描述文件能够根据覆盖率反馈有效指导测试程序的生成,大大提高了微处理器验证效率,使验证人员从手工书写复杂测试程序的繁琐工作中解脱出来.

4 结 论

本文提出了一种由 ADL 描述驱动的基于 CSP 的体系结构级测试程序自动生成方法,初步实现了该方法的原型系统——MA^{2}TG 系统,并成功应用于 DLX 微处理器和自主设计的嵌入式微处理器 EStar 的验证.今后的研究将进一步扩展 ADL 的面向验证的描述能力;研究约束求解算法,为 EFC 库增加针对测试程序自动生成的约束求解方法,提高求解效率;进一步完善 MA^{2}TG 系统,将本文的方法更广泛地应用于实际的微处理器验证中.

References:

[1] Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Boston: Kluwer Academic Publishers, 2000.

[2] Dill DL. What’s between simulation and formal verification? In: Proc. of the 35th Design Automation Conf. San Francisco: ACM Press, 1999. 328–329.

[3] O’Krafka B, Mandyam S, Kreulen J, Raghavan R, Saha A, Malik N. MPTG: A portable test generator for cache-coherent multiprocessors. In: Meitz RO. Proc. of the 14th Annual Int’l Phoenix Conf. on Computers and Communications. Scottsdale: IEEE Service Center, 1995. 38–44.

[4] Aharon A, Goodman D, Levinger M, Lichtenstein Y, Malka Y, Metzger C, Molcho M, Shurek G. Test program generation for functional verification of powerPC processors in IBM. In: Proc. of the 32nd Design Automation Conf. San Francisco: ACM Press, 1995. 279−285.

[5] Chandra A, Iyengar V, Jameson D, Jawalekar R, Nair I, Rosen B, Mullen M, Yoon J, Armoni R, Geist D, Wolfsthal Y. AVPGEN—A test generator for architecture verification. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 1995, 3(2):188−200.

[6] Rubin S, Levinger M, Pratt RR, Moore WP. Fast construction of test-program generators for digital signal processors. In: Proc. of the IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing, Vol 4, Phoenix: Omni Press, 1999. 1989−1992.

[7] Malik N, Roberts S, Pita A, Dobson R. Automaton: An autonomous coverage-based multiprocessor system verification environment. In: Proc. of the 8th IEEE Int'l Workshop on Rapid System Prototyping. IEEE Computer Society, 1997. 168−172.

[8] Aharon A, Bar-David A, Dorfman B, Gofman E, Leibowitz M, Schwartzburd V. Veryfication of the IBM RISC System/6000 by a dynamic biased pseudo-random test program generator. IBM Systems Journal, 1991,30(4):527−538.

[9] Wu LM, Wang KC, Chiu CY. A BNF-Based automatic test program generator for compatible microprocessor VADL. ACM Trans. on Design Automation of Electronic Systems, 2004,9(1):105−132.

[10] Mishra P, Dutt N. Graph-Based functional test program generation for pipelined processors. In: Proc. of the 2004 Design Automation and Test in Europe Conf. and Exposition (DATE 2004). Paris: IEEE Computer Society, 2004. 182−187.

[11] Kumar V. Algorithms for constraint-satisfaction problems: A survey. AI Magazine, 1992,13(1):32−44.

[12] Halambi A, *et al*. 2004. http://www.cs.toronto.edu/~gkatsi/efc/README

[13] Grun P, Halambi A, Khare A, Ganesh V, Dutt N, Nicolau A. Expression: An ADL for system level design exploration. Technical Report, 98-29. Irvine: University of California, 1998.

[14] Bashford S, Bieker U, Harking B, Leupers R, Marwedel P, Neumann A, Voggenauer D. The MIMOLA Language Version 4.1. University of Dortmund, 1994.

[15] Freericks M. The nML Machine description formalism. Technical Report, Technische Universitat Berlin, Fachbereich Informatik, 1991.

[16] Pees S, Hoffmann A, Zivojnovic V, Meyr H. LISA: Machine description language for cycle-accurate models of programmable DSP architecture. In: Proc. of the 36th ACM/IEEE Conf. on Design Automation. New York: ACM Press, 1999. 933−938.