第 37 卷 第 2 期
Vol.37 No.2

计 算 机 工 程
Computer Engineering

2011 年 1 月
January 2011

• 开发研究与设计技术 • 文章编号：1000—3428(2011)02—0284—02 文献标识码：A 中图分类号：TP391

面向可重构编译技术的 RAM 访问优化算法

杨 敏，吴艳霞，顾国昌，孙延腾

(哈尔滨工程大学计算机科学与技术学院，哈尔滨 150001)

摘 要：在基于低层虚拟机的四层 C-to-VHDL 可重构编译架构上，针对 RAM 访问和设计执行性能之间的矛盾，提出一种 RAM 读取优化算法。通过对 IR 访存指令及数据相关性的分析，创建专用数据通路，优化 RAM 的访存过程。实验结果表明，该优化算法能够有效减少 RAM 访问次数。

关键词：C-to-VHDL 可重构编译；FPGA 设计；低层虚拟机；RAM 访问优化

RAM Access Optimization Algorithm Oriented to Reconfigurable Compiling Technique

YANG Min, WU Yan-xia, GU Guo-chang, SUN Yan-teng

(College of Computer Science and Technology, Harbin Engineering University, Harbin 150001, China)

【Abstract】 Aiming at the conflict between the RAM access and the design performance, in the four layer C-to-VHDL reconfigurable compiling framework which is based on Low Level Virtual Machine(LLVM) framework, this paper proposes an optimization algorithm for RAM access. By analyzing the load and store instruction in LLVM's IR and the data dependence, it creats a dedicated data path to optimize the process of reading and writing to RAM. Experimental results show that the optimized algorithm can effectively reduce the access number to RAM.

【Key words】 C-to-VHDL reconfigurable compiling; FPGA design; Low Level Virtual Machine(LLVM); RAM access optimization

DOI: 10.3969/j.issn.1000-3428.2011.02.100

1 概述

近年来，C-to-VHDL 可重构编译技术成为高性能计算领域的研究热点。如何提高编译后 FPGA 设计的速度和性能，成为 C-to-VHDL 设计者普遍关注的问题。FPGA 对 RAM 的访问消耗较多的 CPU 时间，所以，应尽量减少 RAM 的访问次数，节省相应的时钟周期和功耗[1]。

目前，将 ANSI_C 或 C++程序编译成 VHDL 的编译器主要有 SPARK、DWARV、ROCCC 等。SPARK[2]对源程序进行较多的预处理优化，但无 RAM 访存优化。DWARV[3]不进行任何优化，直接将 C 程序转换成 VHDL 程序。ROCCC[4]利用 smart buffer 实现数据重用，优化 RAM 的访问次数，但其只适用于滑动窗口且耗费较多的存储资源。本文在映射前，基于软硬件映射关系，对 IR(Intermediate Representation)进行修改，在源头上减少 RAM 访问次数，耗费资源少且普遍适用。

2 基于 LLVM 的可重构编译框架

2.1 LLVM 简介

低层虚拟机(Low Level Virtual Machine, LLVM)[5]是美国伊利诺斯大学开展的一个开放源代码项目。它采用 GCC 的高级语言前端来解析代码，并将程序转换为静态单一赋值(Static Single Assignment, SSA)形式的 IR，易于进行代码分析和优化工作，因此，LLVM 是一个有效的前端开发工具。

2.2 基于 LLVM 的 C-to-VHDL 可重构编译架构

鉴于 LLVM 良好的编译性能，本文以其为框架支持，进行二次开发，提出的 4 层编译技术架构如图 1 所示。

第 1 层：前端编译。LLVM 利用 GCC 对 C 代码进行语法分析，产生底层独立目标代码，并利用中间代码优化器对其进行软件优化，如死代码消除、公共子表达式消除等。

第 2 层：IR 硬件映射优化。由于 IR 和 VHDL 这 2 种语言的表述能力不同，软件优化后的 IR，没有发挥硬件特性，因此对其进行面向硬件的优化。该层优化主要包括重命名不合法名称、RAM 访存优化、运算并行等。

第 3 层：面向语义级优化。对 IR 进行指令及时序分析，以实现并行及流水设计，提高编译后 FPGA 设计的性能。

第 4 层：VHDL 代码生成。进行 IR-to-VHDL 的转换，输出适合 FPGA 综合的 VHDL 代码。

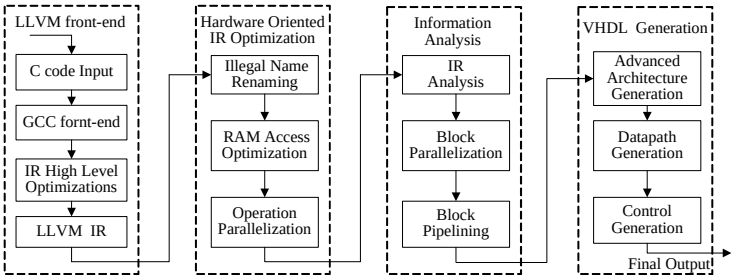

图 1 C-to-VHDL 编译技术架构

3 RAM 读取优化原理

数组在 IR 中以指针形式存在，映射为 RAM 的 0 地址单元指针，数组的下标序列映射为 RAM 的地址序列。IR 中的 load, store 指令映射为 RAM 访存操作，C 程序到 FPGA 设计

基金项目：国家自然科学基金资助项目(61003036)；中国博士后基金资助项目(20100471022)

作者简介：杨 敏(1985－)，女，硕士，主研方向：嵌入式系统，低层虚拟机；吴艳霞，讲师、博士；顾国昌，教授；孙延腾，硕士

收稿日期：2010-05-17 **E-mail：**yangmin.hope@gmail.com

映射方式如图 2 所示。

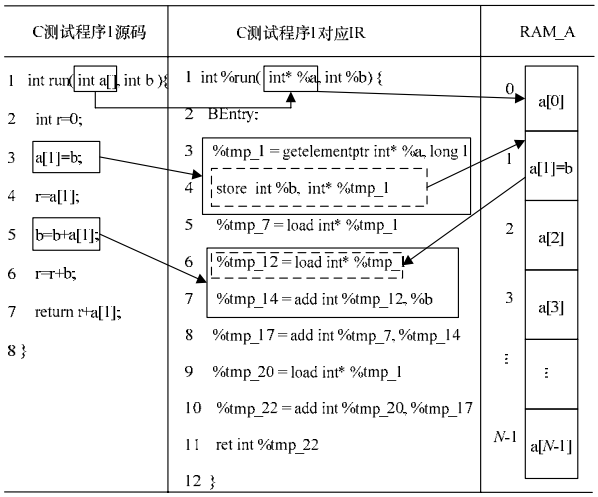

图 2 C 程序到 FPGA 设计映射方式

本文针对读后读及写后读访存类型提出优化算法。

(1)读后读优化原理

当多次对同一存储单元进行读取时，需多次访问 RAM。此时如何减少 RAM 的访问次数，本文提出如下解决方式：将第 1 次读取出的数据存储在寄存器中，将数据通过专用数据通路送入运算部件。这样不必每次使用时都对 RAM 进行访问，从而省去对 RAM 的重复读取过程，将多次读取优化为一次读取。

(2)写后读优化原理

W、R 操作访问 RAM 同一地址单元时，会导致当前数据引用之前处理过的数据，产生访存操作数相关问题，造成写回数据的再读取过程。在访问 RAM 不同地址单元时，访存操作数不具备数据依赖性。不具备数据依赖的读写操作，即使在空间位置上是连续的，也不会造成写回数据的再读取过程。所以，优化时要打破空间位置的限定，在数据流中，基于数据依赖性进行优化。以不同模块串行访问为例进行说明，如图 3 所示，虚框代表一个基本模块。

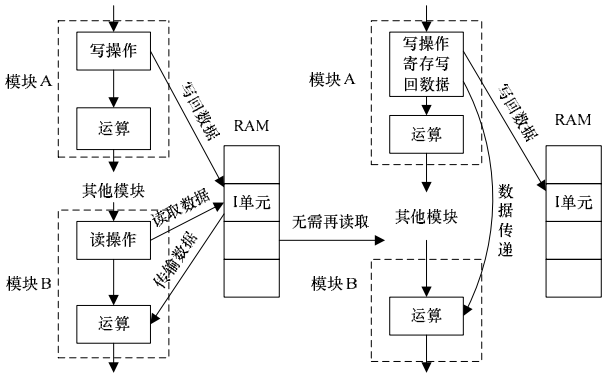

图 3 W-R 串行访问优化

将写回数据直接送入运算部件，不必写回 RAM 后再读取，从而消除相关性。该方法既可去除访问 RAM 的开销，又可减少 RAM 的访问次数。W、R 处于同一模块并行访问时，同理。

4 RAM 读取优化设计实现

根据上述分析，设计相应的算法实现其优化功能。

4.1 读后读优化算法

基于读后读优化原理，设计如下算法实现该优化功能：

(1)对 IR 的 CFG 图进行的深度优先遍历，获得一条根节点到叶子节点的完整路径，将遍历结果存至 map 容器 BB_Map()。

(2)遍历容器 BB_Map，将该路径上的所有指令存入 vector 向量容器 InstrPool()中。

(3)在 InstrPool 中查找对同一单元读取的指令 L1、L2。

(4)判断 L1、L2 间有无对该单元的写回指令，若无，则转(5)；否则，转(6)。

(5)查找和 L2 有依赖关系的指令，将 L1 指令读取的变量传递给 L2 指令的所有使用者。

(6)创建专用数据通路，删除 L2 指向的读取指令。

(7)L2++，继续查找和 L1 对应的读取指令，进行优化。直到 L2 到达向量容器的末端。

(8)L1++，转(2)，直到 L1 达到向量容器末端。

(9)断点返回。判断所有路径是否遍历完成，否，转(1)，是，算法结束。

IR 中只有同一路径上的数据流会有数据依赖，所以，深度优先遍历一定要根据数据流获取模块的遍历序列，以测试程序 2 为例：

```
1 int run ( int a[], int b, i nt c ) {
2       int i, r = a[2] + b;
3    a[2] = b + c;
4    if ( a[2] > 10 ) {
5       for ( i = 0; i < b ; i++ )
6             r = r + a[2];
7       b = a[2] +10;
8         c = a[2] + 20 ;
9         a[2] = b + c;
10        r = a[2] + r;
11            return r;
12   }else{
13        r= r+a[2];
14            return r;
15    }
16 }
```

对应 IR 模块结构见图 4，为方便描述，以数字标识各模块。模块 0 对应第 2 行~第 4 行代码，模块 1 对应第 5 行代码，模块 2 对应第 6 行代码即循环体部分。模块 3 对应第 7 行~第 11 行代码。模块 4 对应第 12 行~第 15 行代码。断点即模块的分支处。

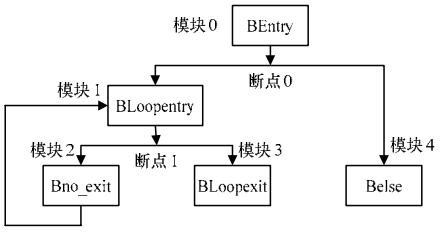

图 4 IR 模块结构 CFG 图

首先获得一条根结点到叶子节点的完整路径(0-1-2-3)，将该路径上所有模块的指令存入向量容器，进行读取优化。然后进行断点返回，遍历右子树。断点返回时(如从模块 3 返回到断点 0 时)，要删除断点跨越的所有模块的指令(删除 1、2、3 模块的指令)，以获得正确的数据流指令序列。

4.2 写后读优化算法

该算法的思想与 4.1 算法类似，限于篇幅，不再赘述。

(下转第 289 页)