基于汇编代码的指令调度器的设计与实现

田祖伟[1,2] 李勇帆[1]
(湖南第一师范学院信息技术系 长沙 410205)[1] (国防科学技术大学计算机学院 长沙 410073)[2]

摘 要 随着嵌入式处理器在各个领域的广泛应用,嵌入式软件的复杂度越来越高。充分发掘嵌入式处理器的性能,需要高级编译优化技术的支持。指令调度是编译器发掘程序指令级并行性的关键技术之一。设计并实现了一个基于汇编代码的指令调度器。实验结果表明,在TECC嵌入式编译器中集成指令调度器后可显著提高程序的性能。

关键词 指令调度,编译优化,汇编代码,表调度
中图法分类号 TP31 **文献标识码** A

Design and Implementation of Instruction Scheduler Based on Assembly Code

TIAN Zu-wei[1,2] LI Yong-fan[1]
(Information Technology Department, Hunan First Normal College, Changsha 410205, China) [1]
(School of Computer, National University of Defence Technology, Changsha 410073, China) [2]

Abstract With the wide use of the embedded processors in various fields, the complexity of embedded software is becoming higher and higher. In order to exploit effectively embedded processor's performance, advanced compilation optimizations are needed. Instruction scheduling is one of the key technologies for compiler to extract the instruction level parallelism. An instruction scheduler based on assembly code was designed and implemented, the experimental results show that integrating the instruction scheduler into TECC embedded compiler can improve effectively the program's performance.

Keywords Instruction scheduling, Compilation optimization, Assembly code, List scheduling

1 引言

随着嵌入处理器应用领域的不断扩展,嵌入式软件的复杂度越来越高,人们对嵌入式系统的性能要求也越来越高,采用传统的低级语言编程不但开发周期长而且代码可重用性差。因此,现在的嵌入式应用程序的编写大部分采用C语言等高级语言。目前,嵌入式计算机的数量已经超过了通用计算机的数量,嵌入式编译技术已成为人们研究的一个热点,但与通用的编译器相比,还有很大的差距。嵌入式编译系统生成代码的质量不高已成为制约嵌入式处理器性能发挥的一个主要瓶颈。为了充分发挥嵌入式处理器的性能,编译器在满足正确性的前提下,要尽可能地对代码进行优化,以便不断提高指令级并行度,充分利用有限的机器资源,缩短程序的执行时间。

指令调度是一种重要的编译优化技术[1,2],其目的是对目标指令序列重新排序,使得新的指令序列可以最大限度地利用目标处理器的并行性,以提高目标指令序列的执行速度。在目标处理器的体系结构特点如单条指令的执行周期、流水级数、可使用功能单元数目等固定的前提下,对目标指令序列进行调度,最大限度地发掘目标处理器并行性的任务高度依赖于编译器的指令调度能力。

嵌入式处理器都使用流水线方式使指令的执行可以重叠进行以提高效率。流水线的CPI(Cycles Per Instruction,即执行每条指令所用的时钟周期数)等于基本CPI加上各种停顿所花费的周期数的总和:

流水线CPI = 理想流水线CPI + 资源停顿 + 数据冲突停顿 + 控制停顿

其中,理想流水线CPI是指流水线的最大流量。减少等式右边的任何一项停顿,可以有效减少流水线CPI,从而提高IPC(Instruction Per Cycle,即为1/CPI)。从编译器角度来看,在硬件特性固定的情况下,编译器是可以减少等式右边的数据冲突停顿和控制停顿的,采取的方法就是指令调度,即通过改变指令在程序中的位置,将相关指令之间的距离加大到不小于指令执行延迟,将相关指令转化为无关指令,尽量使不冲突的指令相邻,从而减少停顿的时间。从嵌入式处理器的特性角度来看,把指令调度交由编译器完成有利于降低功耗、处理器成本和芯片复杂度。

2 指令调度算法

指令调度的任务是在满足依赖关系、资源约束及硬件施

到稿日期:2008-08-29 本文受湖南省教育厅优秀青年基金项目(08B014),湖南省科技厅科技计划项目(2008GK3134),湖南第一师范学院校级课题(XYS05N01)资助。
田祖伟(1973-),男,副教授,博士研究生,主要研究方向为编译优化,E-mail:tianzuwei@126.com;李勇帆(1959-),男,教授,主要研究方向为计算机软件与理论。

加的其它约束条件的前提下，重新排定指令执行的顺序，提高资源利用率，使多个操作能够并行执行，同时减少因相关而引起的处理器停顿，以缩短程序的执行时间。很显然，正确的指令调度是一个语义等价的程序变换。

2.1 指令相关

指令间的相关是造成流水线停顿的主要原因，也是面向ILP开发的各种技术必须解决的关键问题[2]，指令相关主要包括以下几类：

数据依赖(Data Dependence)：对于指令 i 和 j，如果指令 j 使用指令 i 产生的结果，或者指令 j 与指令 k 数据依赖且指令 k 与指令 i 数据依赖，则指令 j 与指令 i 数据依赖。数据依赖可以分为以下三类：

流依赖：一个变量在一个语句中赋值，在后续执行的语句中使用。

反依赖：一个变量在一个语句中使用，在后续执行的语句中赋值。

输出依赖：一个变量在一个语句中赋值，在后续执行的语句中重新赋值。

控制依赖(Control Dependence)：由分支指令引起的依赖。控制依赖决定了与分支指令有关的指令的执行顺序，即非分支指令只能在该执行的时候才能执行。程序中除了第一个基本块中的指令外，控制均依赖于某条或某些分支指令。控制依赖于一个分支的指令不能被移到分支之前执行。没有控制依赖于一个分支的指令也不能移至该分支指令之后从而受这个分支控制。通常情况下，控制依赖关系是必须保留的。

指令调度的算法很多，其原理是消除或减少指令间的停顿，而这种停顿是由于指令间某种依赖关系产生的。判断指令间是否有相关关系，对于判断程序中有多少并行性以及如何利用并行性是做指令调度的前提。如果两条指令是可以并行的，那么它们可以在流水线上同时执行而不会造成停顿。

2.2 表调度算法

表调度的基本思想是在一条指令依赖的所有操作都执行完时立即调度该指令，是一种针对基本块进行指令调度的方法[3,4,6]，其目的是构造基本块的DAG的拓扑序，使得：(1)生成正确的结果，(2)基本块的执行时间尽可能少。基本块调度问题属于NP完全问题，通常采取启发式方法而不是精确的方法来解决这个问题[5]。

表调度方法开始时先从DAG的叶结点向根结点进行遍历，在遍历过程中，用从那个结点到基本块末尾的最大可能延迟来标识每一个结点，即Delaytime。接着从根结点向叶结点方向遍历DAG，在遍历过程中选择要调度的结点，并记录当前时间CurTime和每一个结点为避免一个停顿而应该被调度的最早时间ETime。Sched是已经调度过的结点序列；Cands是每一步骤的候选结点集合，候选结点是那些还没有被调度的，但其前驱已经被调度了的结点。Cands有两个子集：MCands，即到基本块末尾具有最大延迟时间的候选者集合；ECands，即MCands中最早开始时间小于或者等于当前时间的结点集合。

3 基于汇编代码的指令调度器的设计与实现

为进一步提高TECC编译器的性能，我们引入了基于汇编代码的指令调度和优化功能，由于公开发表的针对汇编代码优化的资料比较少，因此在借鉴高级语言编译优化技术的基础上，我们实现了一个基于汇编代码的指令调度器，在实现该功能的过程中最大程度地利用了C++语言提供的STL(Standard Template Library)，STL的引入使得在调度和优化器编写过程中不用再花费大量精力来处理链表和数组等一些基本数据结构，程序员可以更加侧重于功能的设计和实现。

3.1 指令调度器总体设计方案

指令调度器以TECC编译器产生的后缀名为asm的汇编文件为输入，完成调度后产生新的汇编文件。调度器总体流程如图1所示。

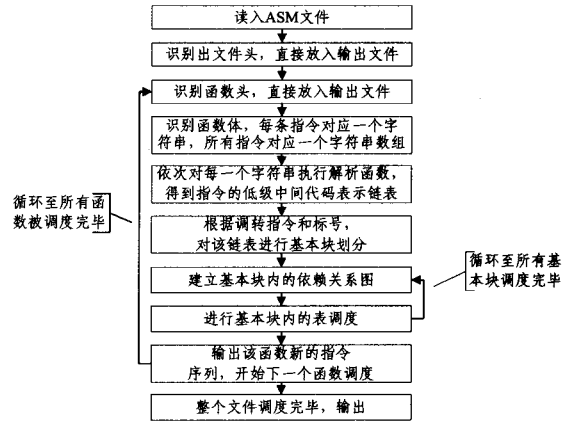

图1 指令调度器总体结构

3.2 指令调度器的算法实现

main()函数是整个流程的开始。tec_instschd()函数以asm文件名为参数，负责该asm文件的总体控制，它调用asm_to_licode()函数逐条把asm指令转换为licode中间代码表示，当tec_instschd()知道完成一个asm函数的转换后就调用bblock_from_licode()进行基本块的划分，随后调用list_sched_bblock()进行基本块的表调度，最后调用finish_sched()释放开辟的空间，这样就完成了一个asm函数的调度。hed_instschd()根据继续读入的数据决定是开始新的调度流程还是结束整个文件，流程如图2所示。

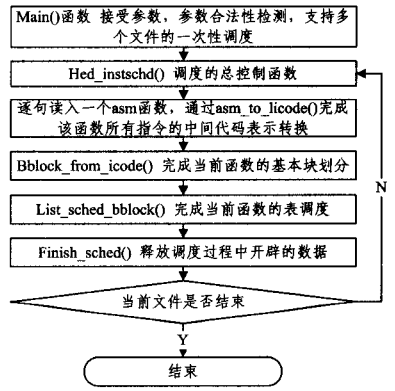

图2 指令调度器的工作流程

主要算法如下：

(1)tec_instschd()函数根据asm文件中的特殊单词来进行单个文件内的调度流程控制，这些特殊单词是在总结asm文件结构特点的基础上发现的。

(2) asm_to_icode()函数负责把一条以字符串形式存在的汇编指令转换成以asminst和operand结构体表示的形式，这个

转换过程使用了根据特殊符号分解字符串和逐步消去的方法。

(3)bblock_from_licode()函数负责基本块的划分。

(4) list_sched_bblock()函数是表调度算法的总调度函数,以当前函数的所有基本块为输入,对每个基本块构建基本块对应的DAG图,然后进行调度。

(5) DAG. init()函数是DAG类的初始化函数,一个DAG类对应一个基本块,该函数实现两个功能:把该基本块中的指令依次放入vertics数组中,统计有多少条指令,也就是DAG图中顶点的个数。

(6)DAG. genEdges()函数是DAG类中负责生成边的函数,如果这两个顶点代表的指令间有依赖关系,那么两个顶点间存在边。两个顶点间是否存在依赖关系是由函数Dependence()来负责分析的。如果两个顶点间存在边,那么后面的结点是前面的结点的DAG后继,前面的结点是后面的结点的DAG前驱。

(7) DAG. cmpt_delaytime()函数负责计算DAG图中各个结点的delaytime,这个delaytime是该结点到基本块末尾的最大可能延迟。其计算公式如下:

若 n 是叶结点 $Delay(n)=ExecTime(n)$

其他 $Delay(n)=\max_{m\in DagSucc(n,DAG)} Late\_Delay(n,m)$

其中,$Late\_Delay(n,m)=Latency(n,2,m,1)+Delay(m)$,根据这个计算公式的特点,我们采用了递归算法,即cmpt_delaytime()本身就是一个递归函数,其中的DagSucc就是前面提到的结点的DAG后继,这个过程是一个后序遍历图的过程,叶结点就是那些没有后继的结点,其Delaytime是其指令执行时间,一个具有多个后继的结点的Delaytime是所有后继的L_D值中最大的。

(8)DAG. listsched()函数是针对当前基本块的DAG图进行表调度的具体实施函数,其接受的输入是当前基本块的DAG图,其中每个顶点(对应基本块的每条指令)都有DAG后继和Delaytime,调度后的输出结果存放在Sched容器中,也就是新的指令序列。调度过程如下:

①首先把根结点放入Cands集合中,也就是当前步骤下可选顶点集合,实际就是那些没有前驱的顶点;

②从Cands中挑选一个顶点放入到Sched集合中,并把该顶点从Cands中删除,接着把该顶点的符合一定条件的后继顶点加入到Cands中;优先选择Cands中Delaytime值最大的顶点,因为延迟最大的顶点可以和更多的顶点并行,如果有多个顶点具有最大延迟时间,那么优先选择ETime小于等于CurTime的顶点,因为ETime较小代表为避免冲突该指令应该被较早地调度执行,如果符合上述条件的顶点仍然具有多个,那么可以根据目标处理器的特征进行启发性选择;更新CurTime;从Cands中删除被选中的顶点,把该顶点的DAG后继顶点中那些符合条件的顶点放入Cands中;更新该顶点的ETime;

③重复步骤②直到Cands为空,则调度完成,Sched中存放的就是新的指令序列。

(9)Dependence()函数计算两个指令间是否存在依赖关系,这里需要注意的是某些特殊操作会潜在地影响到其他的寄存器,如push和pop会影响R7寄存器的值等。两条指令存在依赖则返回1,否则返回0。

(10)licode_to_asm()函数把调度后的指令输出到asm文件。

经过"输入-调度-输出"三个步骤就完成了基于asm文件的指令调度,代码实现借助C++的STL,使用Visual C++ 6.0完成代码编写和编译。

4 实验结果分析

我们在TECC编译器中实现了对本文中提出的基于汇编代码的指令调度器。TECC编译器是基于16位CPU核CIU96自主开发的一款高性能嵌入式编译器,CIU96具有独立的数据空间和代码空间同时支持8位和16位运算方式,部分指令支持32位运算。为了充分发挥该芯片的性能,我们在TECC编译器集成了基于汇编代码的指令调度器。从基准测试程序中选取encode. c,decode. c,jpeg. c等程序进行实验,首先将测试程序分别通过集成/未集成文中提出的指令调度模块的编译器进行编译得到目标代码,然后将它们在CIU96芯片的模拟器上运行,得出每个程序相应的运行周期,最后,求出加入集成指令调度模块后相对于未集成此模块前的加速比,如图3所示。

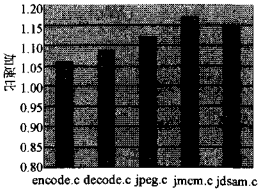

图3 实验数据

从实验结果可以看出,加入指令调度模块后,5个基准测试程序代码执行周期的平均加速比约为1.12,优化后的代码大大缩短了代码执行时间,性能获得了较大的提高。

结束语 随着嵌入式处理器应用领域的不断扩展,嵌入式软件的复杂度越来越高,人们对嵌入式系统的性能要求也越来越高。充分发掘嵌入式处理器的性能,需要高级编译优化技术的支持。本文设计并实现了一个基于汇编代码的指令调度器。实验结果表明,在TECC嵌入编译器中集成指令调度器后可显著提高程序的性能,加入指令调度模块后,5个基准测试程序代码执行周期的平均加速比约为1.12,性能获得了较大的提高。目前,基于嵌入式领域编译优化技术还有大量的研究[6,7],我们将进一步研究嵌入式处理器和CIU96汇编语言的特点,以实现更多的优化,进一步增强TECC编译器的后端优化能力。

参考文献

[1] Aho A, Ullman J. Principles of Compiler Design[M]. Boston: Addison-Wesley, 1986: 25-81

[2] Muchnik S S. Advanced Compiler Design and Implementation [M]. San Francisco: Morgan Kaufmann, 1998: 531-573

[3] Paolo F, Fisher J A, Cliff Y. Instruction Scheduling for Instruction Level Parallel Processors[J]. Proceedings of the IEEE, 2001, 89(11): 1638-1658

[4] Lavery D M, Chang P P, Mahlke S A, et al. The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors[J]. IEEE Transactions on Computers, 1995, 44(3): 353-370

(下转第89页)

```
}
if(数据包在中间节点上)
    { if(N_{m-1}…N_i…N_0 ⊕ D_{m-1}…D_i…D_0 ! =0)
        if(bit_x==1) out_dir=数据包向左发;
        else out_dir=数据包向右发;
      else if(N_{m+n-1}…N_k…N_m ⊕ D_{m+n-1}…D_k…D_m ! =0)
        if(bit_y==1) out_dir=数据包向上发;
        else out_dir=数据包向下发;
      else out_dir=已经到达终点;
    }
```

X-Y 路由算法采用(x,y)标识节点，数据包所经过的每个节点上路由的实现都必须进行算术运算。在上面的路由算法中，中间的节点路由只进行简单的 3 个或 5 个逻辑运算(bit_x,bit_y 是否为 0 通过一个逻辑运算就能证实)，在源节点上多 4 个异或运算和 2 个比较运算。逻辑运算实现相对算术运算要简单得多。同样地可以描述基于 X-Y 的交错路由。

4 NoC 节点的设计

NoC 采用的 Torus 结构，节点的设计结构如图 3 所示。

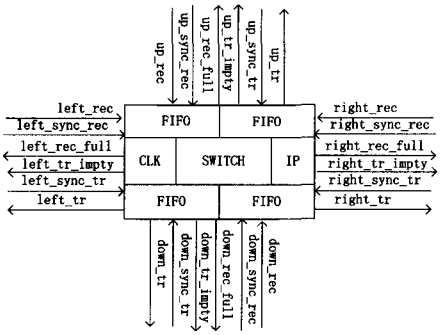

图 3　节点的结构

节点主要由 SWITCH、4 个方向的异步数据缓冲队列 FIFO，IP 以及时钟产生 CLK 等模块组成。其中节点发送 tr 和接收 rec 的数据为并行多位的数据帧格式，帧含有二维平面约翰逊码编址。发送和接收同步信号为 sync_tr 和 sync_rec。tr_full 和 rec_impty 信号表示数据缓冲队列接收已满或发送已空信号，overflow 表示发送有溢出。FIFO 写时钟用相邻节点的时钟进行写，读时钟用本节点的时钟进行读。每个节点内部有一个 IP 模块，IP 与交换单元 SWITCH 共用一个系统时钟，IP 核产生数据帧通过局部链路发给 SWITCH 单元进行转发。

每个节点与周围 4 个节点相连，相连节点之间存在双向传输链路，节点内部各模块之间共享同一个时钟，而节点之间的系统时钟可以不相同，即全局异步局部同步(GALS)。路由采用 3 节中给出的改进的 X-Y 路由方式。节点内部的控制属于集中式控制，由一个控制单元来决定节点的内部交换单元的消息的接收和转发。消息的接收和转发采用的是公平策略，通过轮询节点的 4 个方向的数据缓冲队列 FIFO 中是否存在数据，如果存在，则取数据，然后分析其目的地址和本节点地址决定是转发还是接收；如果不存在，则询问下一个队列，做同样的处理。当需要转发数据时，如果下一个节点的输入 FIFO 未满，则输入至该节点的 FIFO 中；如果已满，下一个节点输出一个高电平的 rec_full 信号至该节点。如果该节点再发送消息至下一个节点，则输出溢出，over_flow 信号输出为高电平，如图 3 所示。

结束语　本文分析了 NoC 的拓扑结构和路由算法，并从超大规模集成电路(VLSI)实现的角度探讨如何选择合适的拓扑结构和相应的节点编码方法。研究了 NoC 的节点编码，提出了基于约翰逊码的二维平面编码。该编码隐含了 Torus 网络拓扑结构以及网络节点之间的连接关系，能够简化片上网络的路由算法的设计和实现。基于该编码和利用 X-Y 路由的路由确定性特点，提出改进 X-Y 路由，降低路由的计算复杂性，并且设计相应的 NoC 节点结构。这种编码方式与 Torus 结构的结合，在 NoC 中具有广泛的应用前景。

参 考 文 献

[1] Bjerregaard T, Mahadevan S. A Survey of Research and Practice of Network-on-Chip. ACM, 2006: 3-15

[2] Nickray M, Dehyadgari M, Afzali-Kusha A. Power and delay optimization for network on chip. Circuit Theory and Design, 2005 (3): 273-276

[3] Rantala V. Network on Chip Routing Algorithms. TUCS Technical Report. No 779. 2006(8): 10-12

[4] Dally W, Towles B. Route packets, not wires: on-chip interconnection networks // Proc. the Design Automation Conference. 2001: 684-689

[5] Rao M V, Chalamaiah Dr N. Hypercube and Its Variant Networks: A Topological Evaluation // Proceedings of the Eighth International Conference on High-performance Computing in Asia-pacific Region. 2005

[6] Lee T-Y, Hsiung P-A, Chen S-J. TCN: Scalable Hierarchical Hypercubes // Proceedings of the Ninth In ernational Conference on Parallel and Dis ributed Systems, 2002

[7] Hu Jingcao. DyAD-Smart Routing for Networks-on-chip // Proc. 41'st ACM/IEEE Design Automation Conf. San Diego, CA, 2004

[8] Kim J, Park D, Theocharides T, et al. A low latency router supporting adaptivity for on-chip interconnects // Proc. of the Design Automation Conf. 2005: 559-564

[9] Bononi L, Concer N. Simulation and analysis of network on chip architectures: Ring, spidergon and 2D mesh // Proc. of the Design, Automation and Test in Europe. 2006

[10] 朱可静. xmesh：一个 xmesh-like 片上网络拓扑结构. 软件学报, 2007, 18(9): 2194-2204

(上接第 47 页)

[5] Berson D, Gupta R, Soffa M L. Integrated Instruction Scheduling and Register Allocation Techniques[C] // Proc. of the 11th International Workshop on Languages and Compilers for Parallel Computing. London: Springer-Verlag, 1998: 247-262

[6] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures, a Dependence-based Approach[M]. San Francisco: Morgan Kaufmann, 2001: 511-547

[7] Leupers R. Compiler Design Issues for Embedded Processors [J]. IEEE Design and Test of Computers, 2002, 19(4): 51-58