支持 SIMD 与簇间双字传输体系下的 VLIW DSP 分簇算法①

陈思灵, 郑启龙, 冯玉谦, 付和萍

(中国科学技术大学 计算机科学与技术学院, 合肥 230026)

摘 要: VLIW DSP 通过软件流水获得时间并行性, 通过指令分簇获得空间并行性. 指令的分簇本质上是资源分配问题. 传统的指令分簇假设一条指令分到某一簇执行, 而某些体系结构提供 SIMD 指令, 传统的分簇算法对这类体系结构并不完全适用. 提出的基于评估模型的分簇算法能对 SIMD 指令和普通指令进行合理的分簇. 分簇之后, 通过调度簇间传输指令, 合成适当的簇间双字传输指令. 由于 SIMD 和簇间双字传输的引入, 以及较好的分簇决策, 程序整体的调度延迟变短. 对许多数字信号处理程序相对于没分簇的情况下的性能有 2~3 倍的性能提升, 相对寄存器压力分簇算法有约 7~10%性能的提升.

关键词: 单指令多数据流; 指令分簇; 簇间双字传输指令; 调度延迟; 数据流图

VLIW DSP Clustering Algorithm for Architecture Supporting SIMD and Inter-Cluster Double Word Transfer

CHEN Si-Ling, ZHENG Qi-Long, FENG Yu-Qian, FU He-Ping

(School of Computer Science and Technology, University of Science And Technology of China, Hefei 230039, China)

Abstract: VLIW DSP obtain time parallelism through software pipelining, and obtain space parallelism through instruction clustering. The essence of clustering is resource allocation. Traditional clustering assumes that one instruction assigns to certain cluster, but that does not applicable to some architecture offering SIMD instructions. This article proposes an algorithm based on evaluation model can do well with the problem of clustering for ordinary instructions and SIMD instructions. By scheduling inter-cluster transfer instruction, we synthesize inter-cluster double word transfer instruction. With the help of SIMD instruction, inter-cluster double word transfer instruction and good clustering policy decision, we make the schedule latency shorter. For many DSP programs, comparing with no clustering, we obtain 2 ~ 3 times increase in performance, comparing with clustering algorithm based on register allocation, we obtain 7~10% increase in performance.

Key words: SIMD; instruction clustering; inter-cluster double word transfer instruction; scheduling delay; DFG

最早的分簇算法是 BUG 算法. BUG 算法降低了 DAG 关键路径上的节点被分到不同簇上的概率, 避免插入簇间传输指令使得关键路径变长. 文献[1]提出基于 component 的算法, 首先将 DAG 根据指令数目分成不同的 component, 然后将各 component 分到各簇上执行. 它还根据 DAG 节点间连接的程度分情况使用不同的算法将 component 分到个簇上. 文献[2]提出基于子图的分簇算法是对基于 component 的算法的改进. 尽量使得关键路径上的节点分到同一个 component 中. 文献[3]提出基于寄存器压力的分簇算法. 考虑到了由于寄存器溢出导致引入 load, store 对关键路径的影响. 在一些提供 SIMD 指令的体系结构下, 这些算法并不适用. 因为这些分簇算法都假定一条指令在一个簇上执行, 而 SIMD 指令要求在各个簇上同时执行. 本文

① 基金项目:核高基重大专项(2009ZX01034-001-001-002)
收稿时间:2012-02-18;收到修改稿时间:2012-04-03

基于文献[4]中的分簇算法, 针对支持 SIMD 与双字传输的体系结构, 提出一种分簇算法.

1 目标机器与OpenImpact编译器扩展概述

BWDSP100 是一款高性能, 16 发射的 VLIW 结构数字信号处理器. 采用 7 级流水线: 取指, 缓冲对齐, 预译码, 译码, 取操作数, 执行, 写回. BWDSP100 有 4 个计算核, 分别是 X 核, Y 核, Z 核, T 核, 每个计算核上有 8 个 ALU, 4 个特殊功能单元, 4 个乘法器, 每个计算核上有 64 个数据寄存器. BWDSP100 有 U、V、W 三个地址产生单元 AGU, 用作访存地址的计算, 每个 AGU 有 16 个地址寄存器. BWDSP100 的结构图如图 1 所示.

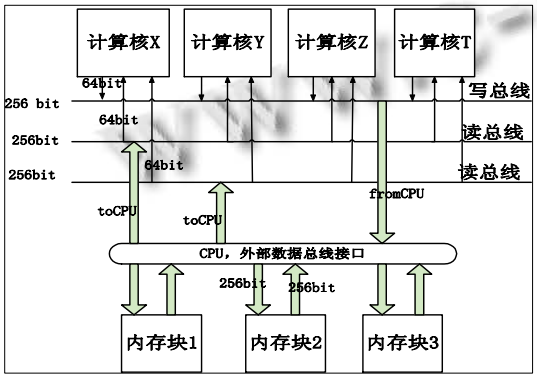

图 1 BWDSP100 体系结构图

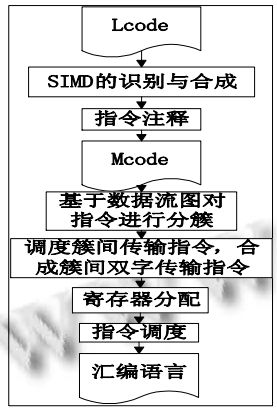

图 2 openImpact 编译器后端流程图

Openimpact 编译器前端的中间语言称为 Pcode, 它能有效的表达源语言级信息, Pcode 的内部数据结构是抽象语法树形式. 后端的中间语言称为 Lcode, 它是一种与机器无关的指令集. 本文所做的工作主要在 Lcode 之后. 先识别生成 SIMD 指令, 指令注释后, 基于数据流图对指令进行分簇. 当指令打标上簇信息以后, 通过调度簇间传输指令生成簇间双字传输指令. 再进行寄存器分配, 指令调度, 最后生成汇编语言. 流程如图 2 所示.

2 各模块的算法设计与实现

2.1 SIMD 指令的识别与合成

通过编译制导将制导信息填入 Lcode 的控制块的结构中, 在有此标记的控制块中进行 SIMD 指令的识别与合成.

SIMD 的识别与合成的步骤是: 循环检测, 循环展开, 常规优化, 变量重命名, 循环不变量和累加变量扩展, SIMD 的合成.

循环检测主要是对循环进行分析, 观察其是否有足够的数据并行产生 SIMD 指令: 循环展开可以将不同迭代的循环体展开在一个基本块内, 从不同迭代的指令之间识别 SIMD 指令就可以在基本块内进行; 常规优化是在 SIMD 优化之前用来精简代码的操作; 变量重命名使得不同迭代中定义和使用不同的变量; 循环不变量和累加变量扩展是为了消除不同迭代之间的流依赖, 经过上述处理, 就可以进行 SIMD 指令的合成.

2.2 分簇算法的设计与实现

本文的分簇算法是根据对资源使用的评估进行分簇决策的. 以控制块为单位建立数据流图, 基于控制块的数据流图进行分簇决策. 之所以不以函数为单位而以控制块为单位建立数据流图进行分簇决策是因为在函数中并不能预测程序的执行路径, 不在执行路径中的指令会使得基于资源的使用评估的算法不准确.

首先要建立数据流图 DFG. DFG 中的每个节点代表一条指令, 节点之间的边代表指令之间的数据依赖关系, 边上的权值表示时间延迟. 在 OpenImpact 编译器中利用到达定值即可获得指令间的数据依赖关系, 通过读入机器描述文件即可获得指令的延迟信息, 也即 DFG 边上的权值. 然后根据 DFG 计算每个节点的最早开始和最晚开始时间. 接下来对 DFG 上的 SIMD 指令进行分簇. 所有 SIMD 指令都打标上簇信息后再对普通指令进行分簇.

2.2.1 指令分簇的优先级规则

优先级规则 1: 先对 SIMD 指令进行分簇, 再对普通指令进行分簇.

优先级规则 2: 最晚开始时间早的指令先分簇.

优先级规则3: 最晚开始时间相同的, 自由度小的指令先分簇. 自由度公式如下.

$$\sigma_{\text{自由度}} = \mathrm{T}_{\text{最晚开始时间}} - \mathrm{T}_{\text{最早开始时间}} \quad (1)$$

优先级规则四: 自由度相同的, 后继结点数较多的指令先分簇.

2.2.2 代价模型

$$\mathrm{cost}(v,c) = \partial \mathrm{cost}_{function} + \beta \mathrm{cost}_{transfer} + \gamma \mathrm{cost}_{register} \quad (2)$$

$\mathrm{cost}(v,c)$ 是数据流图中的 v 节点分到 c 簇上的总代价. $\mathrm{cost}_{function}$ 表示 v 节点分到 c 簇上, 消耗计算功能单元产生的代价. $\mathrm{cost}_{transfer}$ 表示 v 节点分到 c 簇上, 产生簇间传输指令的代价. $\mathrm{cost}_{register}$ 表示寄存器资源溢出引起插入 load, store 产生的代价. α, β, γ 代表各自代价的权重参数, 由大量的实验测试来确定. (一般情况下 α 略小于 β, γ 大于 α, β)

指令分簇的目标是使得程序的调度延迟最小. 在有相同最小调度延迟不同分簇决策中, 选择有较少簇间传输的分簇决策, 也即选择有较低功耗的分簇决策. 当一个节点分到某一簇, 而此簇此时计算功能单元饱和引起等待资源有可能引起整个数据流图调度延迟变长. 在做一个节点的分簇决策时, 如果引起簇间传输指令可能使整个数据流图的调度延迟变长. 在做一个节点的分簇决策时, 如果由于寄存器溢出而插入 load, store 指令, 可能使整个数据流图的调度延迟变长. 这三种因素导致调度延迟变长的程度决定了 $\mathrm{cost}_{function}$, $\mathrm{cost}_{transfer}$, $\mathrm{cost}_{register}$ 的权重值.

$\mathrm{cost}_{function}$ 的计算

算法用Load[c][t][funtype]二维数组表示在c簇时刻t计算资源 funtype 使用的情况, 用 resource[c][funtype] 表示在 c 簇拥有计算资源 funtype 的个数. 当节点 v 分到 c 簇时, 对 $T_{\text{最早开始时间}} \le t \le \mathrm{T}_{\text{最晚开始时间}}$,

$$\text{Load[c][t][funtype(v)] +=} \frac{1}{\sigma_{\text{自由度}} + 1} \quad (3)$$

利用公式 4 计算出 v 节点分到 c 簇上 t 时刻的计算功能单元的代价:

$$\cos t(v,c,t) = \begin{cases} 0, \\ resource[c][t][funtype(v)] \ge load[c][t][funtype(v)] \\ load[c][t][funtype(v)] - resource[c][t][funtype(v)], \\ resource[c][t][funtype(v)] < load[c][t][funtype(v)] \end{cases} \quad (4)$$

利用公式 5 计算出 v 节点分到 c 簇上的计算功能单元的代价:

$$\mathrm{cost}_{function} = \sum_{t=T_{\text{最早开始时间}}}^{\mathrm{T}_{\text{最晚开始时间}}} \mathrm{cost}\ (\mathrm{v},c,t) \quad (5)$$

$\mathrm{cost}_{register}$ 的计算

使用寄存器资源的代价是由虚拟寄存器在生命周期内与已完成分簇的指令的虚拟寄存器的干涉数决定的. thresh 为阈值参数.

如果虚拟寄存器 r 在 c 簇上与已经分配到该簇的虚拟寄存器的干涉数超过 thresh 阈值, 则 cost(r, c)为 1.

$$\mathrm{cost}(r,c) = \begin{cases} 0, & \mathrm{interfere(r,c)} < \mathrm{thresh} \\ 1, & \mathrm{interfere(r,c)} >= \mathrm{thresh} \end{cases} \quad (6)$$

寄存器资源产生的代价是这条指令的所有虚拟寄存器与本簇已分配的其他虚拟寄存器的超阈值干涉的数目.

$$\mathrm{cost}_{register} = \sum \mathrm{cost}(r,c) \quad (7)$$

$\mathrm{cost}_{transfer}$ 的计算

DFG 节点分成三类: SIMD 指令节点, 与 SIMD 指令有数据依赖的普通节点(即 SIMD 指令节点的前驱与后继普通节点), 一般普通节点. 如图 3 所示.

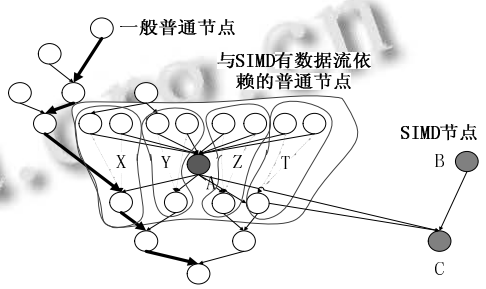

图 3　数据流图示意图

通过自顶向下遍历 DFG, 找出所有的 SIMD 指令. 然后对 SIMD 指令进行分簇, SIMD 的指令在 BWDSP100 体系结构下有 8 个源操作, 4 个目标操作数. 其中 2 个源操作数与 1 个目标操作数是成组配对的. 我们通过在 DFG 中加入虚边表示虚边前驱节点的目标操作数(也即 SIMD 的源操作数)与虚边后继节点的源操作数(也即 SIMD 指令的源操作数)是成组配对的. SIMD 指令的分簇也就是将这些成组的虚拟寄存器分到不同的簇的问题. 由于我们分簇是按照分层进行

的, 所以当对某一 SIMD 进行分簇时, 层次关系在前的 SIMD 指令已经分完簇了. 如下图, 灰色的节点代表 SIMD 指令. 对于没有相互数据依赖的 SIMD 指令节点, 比如节点 A 和节点 B, 成组的虚拟寄存器分到哪一簇是随机分配的, $\cos t_{transfer}$ 为 0. 但是对于已经分配的 SIMD 指令节点有数据流依赖的 SIMD 指令节点, 比如 C 节点对于前面已经分完簇的 SIMD 指令 B 有数据流依赖. 则 C 节点需将虚拟寄存器组分到各簇, 使得引起的簇间传输最少. C 节点只需将与前驱 SIMD 指令节点相关联的虚拟寄存器组进行穷举, 计算需要簇间传输的次数, 选择最少簇间传输的分配方式, 其余的虚拟寄存器组随机分配. 由于 BWDSP100 只有四个簇, 所以穷举最多有 4！=24 种.

通过顶向下遍历 DFG, 找出所有的普通指令, 然后对普通指令进行分簇. 普通指令节点分成与 SIMD 指令有数据依赖的普通节点和一般普通节点. 对普通指令而言:

$$\cos t_{transfer} = \cos t_{pre} + \cos t_{post} \quad (8)$$

对一般的普通节点而言, $\cos t_{pre}$ 为当前指令节点所在的簇与所有一般普通前驱节点所在簇不同的个数. $\cos t_{post}$ 为当前指令节点所在的簇与此节点的一般后继节点已分配的一般前驱节点所在簇不同的个数. 对于与 SIMD 指令有数据依赖且位于上层的普通节点, $\cos t_{pre}$ 与一般普通节点算法一样, $\cos t_{post}$ 需在原来的基础上加附加值. $\cos t_{post\_plus}$ 为节点所在簇(即这一节点的目标操作数所在簇)与 SIMD 相应的源操作数不在同一簇的个数:

$$\cos t_{post} = \cos t_{\text{一般普通节点post}} + \cos t_{post\_plus} \quad (9)$$

对于与 SIMD 指令有数据依赖且位于下层的普通节点, $\cos t_{post}$ 与一般普通节点算法一样, $\cos t_{pre}$ 需在原来的基础上加附加值. $\cos t_{pre\_plus}$ 为节点所在簇(即这一节点的源操作数所在簇)与 SIMD 相应的目标操作数不在同一簇的个数:

$$\cos t_{pre} = \cos t_{\text{一般普通节点pre}} + \cos t_{pre\_plus} \quad (10)$$

2.3 簇间双字传输指令的合成

在分簇模块之后, 添加了簇间传输指令, 这些指令只有一个源操作数和一个目标操作数, 源操作数和目标操作数在不同的簇. 我们通过调度这些簇间传输指令, 将源操作数所在簇相同且目标操作数所在簇相同的指令两两分组. (当然调度这些指令要遵循依赖准则)这时将一条双字传输单播指令替换在一个组中的两条簇间传输指令. 我们通过第二次调度这些双字传输指令, 将具有相同源操作数(两个源操作数都要相同)目标操作数在不同簇上的组成组, 将一条双字传输组播指令替换在同一组中的若干双字传输指令.

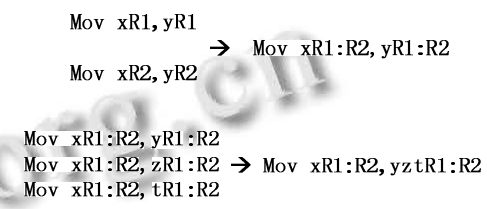

图 4 簇间双字传输指令生成示意图

如图 4, x 簇有 2 条簇间传输指令, 均传输到 y 簇, 这两条簇间传输指令在保证数据依赖的前提下调度到一起, 便可合成 Mov xR1:R2 ,yR1:R2 这样一条双字传输单播指令. x 簇有三条簇间双字单播指令分别传输至 y,z,t 簇, 可将其合成 Mov xR1:R2,yztR1:R2 这样一条簇间双字组播指令.

2.4 寄存器分配

IMPACK 编译框架使用启发式的图着色算法进行全局寄存器分配. 具体实现是为函数中的所有寄存器建立冲突图之后, 将所有的虚拟寄存器按使用频率的大小排序, 然后按照从大到小的顺序依次进行着色.

我们在此实现的基础上改进着色算法, 以满足分簇结构上 SIMD 指令对寄存器的要求. 算法从排好序的虚拟寄存器列表中依次取出虚拟寄存器, 先判断其是否为 SIMD 指令的某个操作数, 如果是, 则同时对其所在的 SIMD 指令操作数的所有寄存器进行分配, 分配不同簇上相同编号且各自不冲突的物理寄存器, 分配失败则对这些不同簇上的寄存器进行溢出处理; 如果不是, 则从所在簇中分配一个不冲突的物理寄存器, 若分配失败则对该寄存器进行溢出处理.

由于双字传输指令要求使用相邻寄存器, 算法从排好序的虚拟寄存器列表中依次取出虚拟寄存器, 先判断其是否为双字传输指令的某个操作数, 如果是, 则同时对其所在的双字传输指令操作数的所有寄存器进行分配, 分配指定簇上编号一致的相邻的各自不冲突的物理寄存器, 若分配失败则对该寄存器进行溢出

处理.

经过寄存器分配后, SIMD 指令, 簇间传输指令和一般指令, 就交由调度模块进行调度.

3 性能测试

表 1 FFT_radix2 测试数据

FFT_radix2	64	512	1024	4096	16384
TS201 优化(-O)	8906	84435	180266	1117993	5110479
BWDSP100 优化(-O)	8144	60902	126090	551314	2422298

表 2 FFT_radix4 测试数据

FFT_radix4	64	256	1024	4096	16384
TS201 优化(-O)	6189	28019	127187	762408	3471913
BWDSP100 优化(-O)	4082	14771	59738	254231	1088252

表 3 IIR 测试数据

IIR	64	256	1024	4096	16384
TS201 优化(-O)	3951	15632	62500	250253	1002247
BWDSP100 优化(-O)	5394	21122	84098	336002	1343618

实验通过对比 TS201 和 BWDSP100 的时钟周期数进行性能测试, 测试集选择常用的数字信号处理程序. 其中, BWDSP100 实现了本文的分簇算法, 时钟周期数是在 BWDSP100 的集成开发环境 ECS 上测量的.

表格第一行代表数字信号处理程序输入的点数. 第二行是 TS201 这款数字信号处理器上编译器产生目标代码执行的指令周期数. TS201 只有一个簇. 第三行是 BWDSP100 这款数字信号处理器上编译器产生目标代码执行的指令周期数. BWDSP100 有四个计算簇, 使用的是本文所描述的分簇算法. 我们可以看到, 对 FFT_radix2 和 FFT_radix4 而言, 我们通过较好的分簇算法充分利用四个计算簇的资源, 能使程序的性能相对没有用分簇算法的性能提高 2-3 倍. 当然对于一些难以利用并行性的程序, 分簇和不分簇并没有大的区别, 如 IIR 程序.

4 结语

生成 SIMD 指令, 对 SIMD 指令和普通指令进行合理分簇, 生成簇间双字传输指令, 能使一部分数字信号处理程序的整体调度延迟变短, 从而提高程序的性能. 而有些程序本身不可并行化或难以并行化, 对于这类程序分簇并没有得到好处. 本文提供的分簇算法在 BWDSP100 的体系下能使一部分典型的数字信号处理程序的性能提升 2-3 倍.

参考文献

1 Desoli G. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach.HPL-98-13, Hewlett-Packard Laboratories,1998:1–17.

2 郑启龙,汪胜,夏霏.DSP 编译器中一种基于子图的分簇算法.微电子学与计算机,2010,27(8):49–52.

3 雷一鸣,洪一,徐云,姜海涛.一种基于寄存器压力的 VLIW DSP 分簇算法.计算机应用,2010,30(1):274–276.

4 Lapinskii VS, Jacome MF, De Veciana G. Cluster assignment for high-performance embedded VLIW processors. ACM Trans. on Design Automation of Electronic Systems, 2002,7(3):430–454.

（上接第 70 页）

除算法及其应用.中国机械工程,2001,12(2):173–175.

8 Shreiner D, Woo M, Neider J, Davis T. 徐波译.OpenGL 编程指南.6th ed.北京:机械工业出版社,2008.1–420.

9 陈志杨,喻谷鸣,张引.基于样条链(环)的 CAD 模型过渡特征识别与抑制.中国机械工程,2011,22(22):2707–2711.

10 丁永祥,夏巨谌,王英,肖景容.任意多边形的 Delaunay 三角剖分.计算机学报,1994,17(4):270–275.

11 Wu XB. A New Study of Delaunay Triangulation Creation. Acta Geodaectica et Carto graphica Sinica, 1999,28(1):28–35.

12 徐永安,杨钦,吴壮志,陈其明,谭建荣.三维约束 Delaunay 三角化的实现.软件学报,2001,12(1):103–110.