

DSP 编译器中一种基于子图的分簇算法

郑启龙, 汪 胜, 夏 霏

(中国科学技术大学 计算机学院, 安徽 合肥 230036)

摘 要: 为了提高发射宽度, 高端 DSP 普遍采用分簇结构设计. 分簇结构的处理器依赖编译器在代码生成的时候指定每条指令所在的簇. 针对传统分簇算法中存在参考信息太过局部的问题, 提出一种依据资源压力将指令依赖图划分成多个子图, 然后对子图进行分簇的算法. 最后, 验证了该算法能够提高分簇效率.

关键词: 数字信号处理器; 分簇; 编译器; 依赖图

中图分类号: TP31

文献标识码: A

文章编号: 1000-7180(2010)08-0049-04

Cluster Assign Algorithm Based on Subgraph for DSP Compiler

ZHENG Qi-long, WANG Sheng, XIA Fei

(Department of Computer Science, USTC, Hefei 230026, China)

Abstract: For more issue width, the clustered VLIW processor is a common design in advanced DSP. A clustered processor need compiler assign cluster to each instruction in code-generation. For the problem that traditional cluster assign algorithm using local reference information, the paper proposes an algorithm which divide dependence graph to multiple sub-graph first, then assign cluster to each sub-graph later. It is verified that the algorithm is effective.

Key words: DSP; cluster; compiler; dependence graph

1 引言

随着 VLIW 处理器发射宽度的增加, 单个周期内, 访问寄存器文件读写端口、运算器等功能单元的请求增加, 处理器内部资源争用严重. 这直接导致了处理器的主频难以提高且功耗增加. 采用分簇结构可以有效的避免这些情况^[1]. VLIW 处理器中的分簇结构是指, 单个的集中的寄存器文件划分成多个独立的寄存器文件, 每个独立的寄存器文件与其私有的功能单元相连组成一个簇. 簇与簇之间采用簇间总线传递数据. 指令被分配到多个簇上并行执行. 在宽发射的 DSP 中, 由于分簇降低了单个寄存器文件读写端口的每个周期内访问数量, 处理器能够获得较高的主频. 高端 DSP, 如 TigerShark201, TI 的 TMS320C6x 普遍采用了分簇结构.

2 分簇处理器模型与分簇算法简介

为描述方便, 本文定义一个典型的 4 簇的 VLIW 处理器模型, 其结构如图 1 所示: 共有 X, Y, Z, T 四个簇. 簇与簇之间通过簇间传输总线传送数据. 每个簇的执行资源如表 1 所示. 所有指令的执行延时为 1.

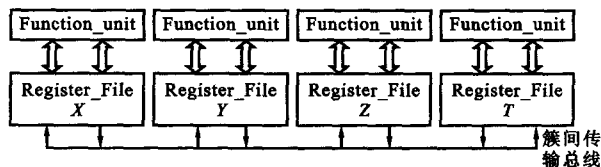


图 1 分簇的体系结构

表 1 每个簇的可用资源列表

资源 ID	mul	alu	imm	Id
资源数量	1	2	2	2

收稿日期: 2010-06-10; 修回日期: 2010-07-10

基金项目: 核高基项目(2009ZX01034-001-001-002); 安徽省自然科学基金(090412068)

分簇结构下, 每条指令编码中包含有簇编号, 簇编号指明指令执行时使用的资源. 指令 $\{X\} R_1 = R_2 + R_3$, 表示使用 X 簇上的一个 alu 资源求 X 簇上的 R_2, R_3 寄存器之和, 结果保存在 X 簇的 R_1 寄存器中. 指令 $\{X\} R_1 = \{Z\} R_2$ 表示传递 Z 簇上的 R_2 寄存器的值到 X 簇上的 R_1 寄存器中.

分簇结构的处理器需要编译器在代码生成的时候指定每条指令在哪个簇上执行. 指令分簇后, 指令依赖图的关键路径长度反映了编译器分簇结果的好坏. 在同样的资源限制下, 最终指令依赖图关键路径越长代表最终指令序列执行的时间越长. 分簇的目的是在满足处理器资源限制的情况下, 通过充分使用多个簇上的资源, 缩短最终指令依赖图的长度. 论文在研究已有的分簇算法的基础上, 提出在寄存器分配前通过划分依赖图为多个子图进行分簇的算法.

2.1 BUG 算法

最早的指令分簇算法是 bulldog 编译器^[2]提出的 BUG(bottom up greedy)算法. 该算法首先建立指令序列的依赖图, 依赖图是一个有向无环图, 每个节点代表指令, 节点之间有向边代表指令之间的依赖. 每个节点都有一个最早开始时间(earliest_time)和最晚开始时间(latest_time). BUG 算法从依赖图的叶子节点出发向上递归回溯访问每一个节点, 在访问过程中如果一个节点有多个未被访问的前驱, 则优先访问最早开始时间(earliest_time)最大的前驱. 在递归回退的过程中根据当前每个簇上的资源负载情况和该节点最优的调度时间, 为当前节点分配簇.

BUG 算法从依赖图的叶子节点出发按照最早开始时间最晚优先的原则遍历整个依赖图, 所以依赖图的关键路径上的节点会首先被访问、被分配单个簇上资源, 降低了关键路径上的节点被分配到不同簇上的概率, 保证了分簇完成后关键路径的长度不会由于插入的簇间传输指令而明显增大^[3,4].

2.2 传统的 BUG 算法存在的问题

但是, 我们观察到基本的 BUG 算法存在以下问题:

(1) 在分簇的过程中由于簇间的传递延时, 可能使得依赖图的关键路径发生变化. BUG 算法能保证关键路径的节点被优先处理分簇, 但是不能保证在分簇完成以后这些节点还在关键路径上.

(2) 在决定当前一个节点的分簇过程中, 该节点可能有多个簇可供选择, 决定分簇的参考信息主要是当前每个簇上的资源压力和已分配节点的调度时间, 而当前的分簇决定对未处理的子节点的影响

却忽略了.

3 基于 component 的分簇算法

基于 component 的分簇算法将分簇决定推迟到依赖图被划分成多个子图(component)以后, 以 component 作为分簇的基本单元, 即不再是以指令作为分簇的基本单元. 由此减少了 BUG 算法由于分簇时机过早带来的盲目性. 需要声明的是算法中使用的“资源压力”这个概念: 对任意节点 n 使用的任意资源 RID, 在该节点 n 可调度的 $[n::\text{earliest_time}, n::\text{latest_time}]$ 时间范围内 RID 对被分配的簇都有 $1/n::\text{slack}$ 的资源压力, 其中 $n::\text{slack} = n::\text{latest_time} - n::\text{earliest_time} + 1$. 这种概念借鉴自经典的模调度算法^[5].

算法大概过程为从依赖图的叶子节点出发, 向上递归遍历访问前驱节点, 如果存在多个未访问的前驱节点, 则优先访问最早开始时间最大的前驱节点. 在递归访问一个节点时候根据当前 component 的资源压力, 决定将此节点加入到当前 component 中或者创建一个新的包含此节点的 component. 遍历完成后, 我们得到一个按照创建顺序排列的 component 队列, 每个 component 中的指令产生的资源压力不会超过单个簇能提供的资源总量. 随后我们以 component 为分配对象, 按照 component 创建的顺序为每个 component 分配簇. 单个 component 的资源压力不超过单个簇能提供的资源保证了不会由于资源竞争导致单个 component 调度时间拉长.

在遍历的过程中需要检测 component 当前的资源压力, 为此每个 component 内部会维护一个二维表 reservation_table[schedule_time][resource_ID], 该二维表记录了每个调度周期资源压力情况. reservation_table[t][id]: 表示 component 中在 t 时刻, 编号为 id 的资源的资源压力. 为描述单个簇上的资源, 定义向量 Resource_Table. Resource_Table[resource_ID] 代表编号为 resource_ID 的资源在单个簇上的数量.

在算法开始前我们还定义全局变量 global_reservation_table 记录分簇时各个簇上的资源压力情况. global_reservation_table 是一个三维数组, global_reservation_table[cluster_ID][time][resource_ID] 表示在 cluster_ID 簇上的 time 时刻 resource_ID 资源的资源压力. 全局队列 component_queue 按创建顺序保存在遍历过程中依次创建的 component. 算法中“COMPONENT”用来代表依赖

图子图类型.

基于 component 的分簇算法如下:

(1) 建立依赖图,计算每个节点在依赖 DAG 图中的最早开始时间(earlist_time)和最晚开始时间(latest_time).每个节点 n 在依赖图中都有 latest_time 和 earlist_time,代表该节点可以被调度执行的最晚时间和最早时间,即在资源无限的情况下节点 n 在[earlist_time, latest_time]区间内调度,可以保证总的调度时间不走过关键路径的长度.slack = latest_time-earlist_time 代表节点 n 可以被调度的宽松程度.收集依赖图的叶节点,按照最早开始时间(earlist_time)越大优先级越高的原则建立处理 leaf_nodes 队列;

(2) 按顺序处理 leaf_nodes 队列中每个节点 n ,从节点 n 向上递归建立一个 component:

输入:叶子节点 Node n ;调用者创建的初始 COMPONENT comp;

```
void create_component(Node n, COMPONENT comp)
{
    FOR EACH pre IN n's precedence nodes In Order Of
    biggest earlist_time
    { //按最早开始时间最大优先的顺序递归遍历所有
    未访问的前驱节点
        IF (is_visited(pre)) continue;
        set_visited(pre);
        IF(check_resource(pre, component))
            create_component(pre, comp); //递归访问
        ELSE
        {
            Component new_comp = New COMPONENT; //
            创建新的 component
            push_back(component_queue, new_comp);
            create_component(pre, new_comp); //递归访问
        }
        register_resource(n, component);
    }
}
```

算法在一开始从叶子节点出发向上递归回溯的遍历各个节点,在遍历的过程中如果一个节点存在多个没被遍历过的前驱,则选择最早开始时间(earlist_time)最大的节点优先遍历.在遍历之前先检测当前 component 的资源压力;check_resource(pre, component)是用来检测如果前驱节点 pre 被安排到 component 中会不会导致 component 中 pre 可调度的时间范围内资源超过限制.register_resource(node, component)是用来注册当前的节点到

component 中.register_resource 和 check_resource 两个过程相互配合,精确的维护 component 中的资源压力不超过单个簇上的资源限制.

register_resource(n , component):注册当前节点 n 到 component 中.为了反应当前节点带来的资源压力,同时更新 component 内部资源记录表 reservation_table.更新过程是,对 n 使用的每种资源 resource_id,在[n ::earlist_time, n ::latest_time]内的每个 time 都有:reservation_table[time][resource_id] += 1/ n ::slack.

check_resource(pre, component):检测如果当前前驱节点 pre 被安排到当前 component 内,会不会导致 component 内的资源压力过大.同样也是使用 reservation_table 表,检测当前在节点 pre 可调度的范围[pre::earlist_time, pre::latest_time]范围内的每一个调度周期 time,如果存在 reservation_table[time][resource_ID] + 1/pre::slack > Resource_Table(resource_ID)即代表当前 component 内的资源压力不允许将此节点加入 component 内.

(3) 按照顺序从 component_queue 取出当前要处理的 component,依次尝试将当前的 component 分配到 X, Y, Z, T . 下面的算法尝试将 component 分配到 c 簇上.其中 component::earlist_time 代表 component 内所有节点的 earlist_time 最小值,component::latest_time 代表 component 内所有节点的 latest_time 的最大值.

```
BOOL assign_cluster (COMPONENT component,
CLUSTER_ID c)
{
    FOR time FROM component::earlist_time TO component::latest_time
    { //检查每个调度时刻
        FOR resource_ID in ALL_RESOURCE_ENUM
        { //检查每种资源类型
            IF(global_reservation_table[c][time][resource_ID] + component::reservation_table[c][time][resource_ID] >
                Reservation_Table(resource_ID)
                return FALSE; //分簇不成功
        }
    }
    //更新全局资源压力表
    global_reservation_table[c] = component::reservation_table;
    return TRUE;
}
```

assign_cluster 返回 FALSE 代表 component 不能被分配到 c 簇上. 对于这种情况我们设置定义 restore_comps 队列保存这些未能成功分簇的 component.

(4) 在 component_queue 被处理后, 对 restore_components 中的任意一个子图 m 统计与 m 连接边最多的已分配子图 n , 将 m 与 n 分配同样的簇. restore_components 中的 component 代表着当前处理器上的资源不够用而导致的不能被分簇处理的节点. 而这种情况在高端 DSP 中很少存在.

基于 component 的分簇算法由于采用 BUG 算法一样的遍历访问顺序, 所以关键路径上的节点得到了优先处理, 由此建立的 component_queue 中的 component 也会按照关键路径优先的顺序排列. 同时由于单个子图被整体分簇避免了 BUG 的分簇时参考信息的局部性.

基于 component 的分簇算法在创建初始 COMPONENT: comp 以后, 首先从节点 end 出发向上遍历, 节点 {1, 3, 2, 5} 依次被加入到 comp 中. 随后递归回溯, 访问 end 的第二个前驱节点 6, check_resource(node_6, component) 检测到, 在 6 号节点加入到当前 component 中后, 将导致 component 中的资源压力超过单个簇, 则返回 FALSE. 此时分簇算法建立新的 COMPONENT: new_comp, 然后调用 create_component(node_6, new_comp) 将节点 {4, 6} 依次加入到 new_comp 中. 最后回溯到 end 节点, end 节点加入到 comp 中. 最后进行分簇的时候 {1, 2, 3, 5, end} 作为一个 COMPONENT 被分配到了 X 簇上, 而 {4, 6, end} 被分配到了 Y 簇上. 最终结果如图 2 所示. 基于 component 的分簇算法成功的避免了 BUG 算法分簇时的盲目的把节点 4 分配到 X 簇上.

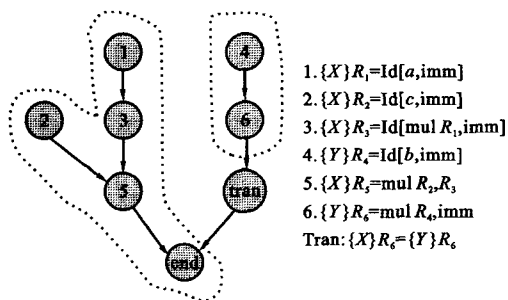


图 2 基于 component 分簇算法的结果

4 实验评测

与配置相同资源的集中结构的处理器相比, 分

簇结构带来的簇间传输指令会导致最终的总的周期数增加. 假设相同资源配置下集中结构的处理器的周期数为 UT, 分簇结构处理器的周期数为 CT, 评价分簇算法的好坏是 UT/CT 的值越接近 1 越好. trimaran^[6] 中实现了 BUG 算法和分簇体系结构模拟器^[7], 我们在 trimaran 的基础上采用 DSPstone 作为测试标准, 比较了 component 的分簇算法和 BUG 算法的性能. 在 trimaran 模拟器中定义分簇处理器 CP 和集中结构处理器 UP. CP 有四个簇, 每个簇有 32 个寄存器, 每个簇的运算资源配置如表 1 所示. UP 拥有 CP 四簇上所有的运算资源和寄存器数量. UP 和 CP 的指令发射宽度都是 16 发射. 图 3 比较了 BUG 算法和 component 分簇算法产生的 UT/CT 比值. 可以看到 component 算法在大部分 DSP 应用中都比 BUG 算法有效的减少了簇间传输带来的延迟. 但是在某些程序中 (如 $n_real_updates$), 单个簇基本上就能满足程序中所有的可并行指令执行的时候, component 算法不如 BUG 算法. 这是因为基于 component 分簇算法带来的簇间传递会导致关键路径发生改变, 在少数情况下, 这种改变大于单个簇上的资源竞争带来的关键路径改变.

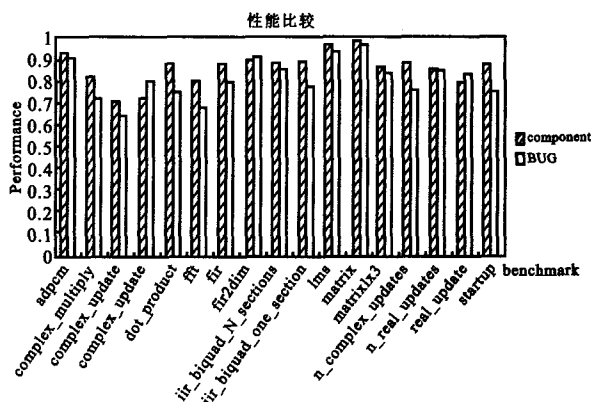


图 3 性能比较

5 结束语

分析了 BUG 算法存在分簇时参考信息太过狭窄的问题, 提出通过资源压力检测, 将依赖图划分成多个子图, 每个子图的资源压力不超过单个簇上的资源总量, 最后对子图进行分簇的算法, 该算法与 BUG 算法相比显著改善分簇的效率. 由此编译器通过优化分析为分簇结构提供了支持, 保证了高发射宽度处理器的主频不会由于发射宽度增加而下降.

(下转第 56 页)

量效率. 理论和仿真结果均说明该算法性能优于 LEACH. 该算法具有如下优点: (1) 能量效率高, 由于花费函数的引入, 既在一定程度上平衡了簇头的能量消耗, 又提高了簇成员节点的能量效率; (2) 显著延长了网络的生命周期, 生命周期比 LEACH 延长达到 40%.

参考文献:

- [1] Heinzelman W, Chandrakasan A, Balakrishnan H. Energy - efficient communication protocol for wireless microsensor networks[C]// Proceedings of the 33rd International Conference on System Sciences (HICSS '00). Hawaii, 2000: 3005 - 3014.
- [2] Heinzelman W, Chandrakasan A, Balakrishnan H. An application - specific protocol architecture for wireless microsensor networks [J]. IEEE Transaction on Wireless

Networking, 2002, 1(4): 660 - 670.

- [3] Handy M J, Haase M, Timmermann D. Low energy adaptive clustering hierarchy with deterministic cluster - head selection[C]// Proc. of the 4th IEEE Conf. on Mobile and Wireless Communications Networks. Stockholm: IEEE Communications Society, 2002: 368 - 372.
- [4] Younis O, Fahmy S. Heed: a hybrid, energy - efficient, distributed clustering approach for ad - hoc sensor networks [J]. IEEE Trans. on Mobile Computing, 2004, 3(4): 660 - 669.
- [5] 曹涌涛, 何晨, 蒋铃鸽. 无线传感器网络中基于自适应定时器策略的分簇算法[J]. 电子学报, 2007, 35(9): 1719 - 1723.

作者简介:

尚凤军 男, (1972 -), 博士, 副教授. 研究方向为无线传感器网络、网络流量测量、WiMAX 等.

(上接第 52 页)

参考文献:

- [1] 胡定磊, 陈书明, 刘春林. 分簇结构超长指令字 DSP 编译器的设计与实现[J]. 小型微型计算机, 2006, 27(2): 45 - 50.
- [2] Ellis J. Bulldog: a compiler for VLIW architectures[D]. Massachusetts: CS department of MIT, 1982.
- [3] Lowney P G, Freudenberger S M. The MultiFlow trace scheduling compiler[J]. The Journal of supercomputing, 1995(7): 51 - 147.
- [4] Giuseppe Desoli. Instruction assignment for clustered VLIW DSP compilers: a new approach[J]. HPLab tech - report, 1998(5): 32 - 87.
- [5] Monica Lam. Software pipelining: an effective scheduling technique for VLIW [C]// proceedings of PLDI. New

York, USA, ACM, 1988: 318 - 328.

- [6] Trimaran research group. Trimaran manul[EB/OL]. [2010 - 01 - 10]. <http://trimaran.org>.
- [7] Michael L Chu. Cooperative data and computation partitioning for decentralized architectures PhD - thesis [D]. Michigan, CS department of Michigan university, 2007.

作者简介:

郑启龙 男, (1969 -), 副教授. 研究方向为并行编程环境与技术.

汪 胜 男, (1983 -), 硕士研究生. 研究方向为编译器后端优化.

夏 霏 男, 博士研究生. 研究方向为嵌入式 CPU 体系结构与编译器.