第36卷 第19期 计 算 机 工 程 2010年10月
Vol.36 No.19 Computer Engineering October 2010
·软件技术与数据库· 文章编号：1000—3428(2010)19—0104—03 文献标识码：A 中图分类号：TP302

面向 ASAP 自定义指令生成算法研究

王 军[1]，周学海[2,3]
(1. 安徽新华学院信息工程学院，合肥 230088；2. 中国科学技术大学计算机系，合肥 230027；
3. 中国科学技术大学苏州研究院嵌入式系统实验室，江苏 苏州 215123)

摘 要：从处理器的指令集进行扩展的优势主要是降低系统设计时间和代价以及可减小代码大小、寄存器压力，从而降低取指频率和功耗。基于此，结合 ASAP 框架给出自定义指令生成的算法，通过数据流分析、指令簇标记、子图枚举、子图合并的方法，找出符合自定义扩展指令的多个约束要求的候选指令集合。实验结果表明，该算法能够高效地找出目标应用的所有非平凡自定义指令集合。
关键词：自适应处理器；指令集扩展；指令生成

Study on ASAP-oriented Self-defined Instruction Generation Algorithm

WANG Jun[1], ZHOU Xue-hai[2,3]
(1. School of Information Engineering, Anhui Xinhua University, Hefei 230088, China;
2. Department of Computer, University of Science and Technology of China, Hefei 230027, China;
3. Embedded System Laboratory, Suzhou Institute for Advanced Study of USTC, Suzhou 215123, China)

【Abstract】The advantage of the extension from the existing command set of processor is primarily to minimize the time and cost of system design, reduce the code size, limit the command fetching frequency, release the pressure on registers, thus the overall system power consumption is lower. On the basis of this, this paper presents a self-defined instruction generation algorithm combined with the frame of ASAP. The algorithm finds candidate instruction set complying with multiple requirements by self-defined instruction expansion, through data flow analysis, instruction clustering, sub-graph enumerating and sub-graph merging methods. Experimental results show that the algorithm can enumerate all the non-trivial candidates efficiently.
【Key words】 self-adaptive processor; instruction set extension; instruction generation

1 概述

专用指令集处理器(Application Specific Instruction-set Processor, ASIP)是为特定的应用而设计的专用处理器。可重构计算系统作为一种新的体系结构，采用可编程的硬件模块来实现计算，以及面向可重构系统的操作系统来管理硬件资源、划分和调度硬件任务，并向开发人员提供高层次的编程模型[1]。专用自适应处理器(Application Specific Adaptive Processor, ASAP)将 ASIP 技术与可重构技术相结合，让处理器能够动态地扩展自定义指令以适应变化的应用需求；同时，保证底层硬件的重构对上层软件透明，使自定义指令能够动态地映射而不改变对应用程序的接口，以重用原有工具链。指令集扩展算法一般被划分为 2 个子问题：(1)自定义指令的生成；(2)自定义指令的选择。本文的研究工作主要针对前者进行。

2 指令生成问题及算法描述

常见的指令生成算法大部分是基于数据流图的分析，剖析数据流图的热点块，将运行多次的指令序列作为自定义指令，由于指令序列存在各种数据流依赖关系，指令集扩展时不能简单地随意选择其中的子序列的指令作为自定义指令，否则将因为数据依赖关系而破坏扩展指令的原子性。如图 1 所示，如果将其中的 0、1、3 指令选择为一条可扩展指令，则由于数据流依赖关系，该指令必须先执行一部分，将数据传递给指令 2 之后，等待 2 的结果，然后才能继续执行。这样的方式是违背指令执行原则的。因此，自定义指令生成首先要分析数据流依赖关系，保证不会违反指令的原子性。

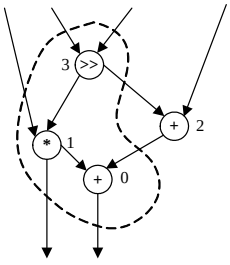

图 1 指令扩展与数据流依赖关系

仿存指令以及 I/O 指令等由于存储层次的复杂性及外部设备响应的不可预期性，无法对其执行时间进行预测。如果将这类指令加入自定义指令，无法保证自定义指令的顺利执行，因此，必须将这类指令去除。

基金项目：国家“863”计划基金资助项目(2008AA01Z101)；安徽省自然科学基金资助项目(070412030).
作者简介：王 军(1972－)，男，讲师、硕士，主研方向：嵌入式系统，计算机体系结构；周学海，教授、博士生导师
收稿日期：2010-04-10 **E-mail：**wjnj99@mail.ustc.edu.cn

而且，产生扩展指令本身还必须考虑目标体系结构的一些限制：

(1)操作数个数。由于目标体系结构的寄存器文件的端口一般是有限制的，因此扩展指令的输入输出寄存器数会受到限制。

(2)扩展指令条数。原指令集的编码等限制了能添加的新指令条数。

(3)面积。扩展指令实现到硬件中之后总会占用一定的额外面积，而嵌入式系统中对面积的限制相对比较严格，芯片内的面积也相当宝贵。因此，一般处理器会对可扩展指令的总面积进行必要的限制[2]。

综合上述要求，将指令集扩展问题定义如下：

(1)问题定义

令数据流图 $G(V,E)$表示指令间的数据依赖关系，节点 V 代表指令，边 E 代表数据依赖关系。$G(V,E)$是有向无环图，$G(V,E)$中可以被包含在扩展指令中的节点为有效节点 v，不能被包含的则为无效节点 φ。令 P 是当前处理的节点集合。在 P 中所有从 P 外输入操作数的节点为输入节点集 IN(P)，所有从 P 中向 P 外输出操作数的节点为输出节点集 OUT(P)。微体系结构上对可扩展指令的输入输出限制分别为 M_{in} 和 M_{out}。$PATH(u,v)$ 表示图 $G(V,E)$中 u 到 v 的路径。则自定义指令生成问题即寻找合适的节点集合 P，使得：

$$\begin{cases} \left|\text{IN}(P)\right| \le M_{\text{in}} \\ \left|\text{OUT}(P)\right| \le M_{\text{out}} \\ \forall \varphi\ ,\ \varphi\ \notin P \\ \forall v_i, v_j \in P,\ \neg\exists v_k \notin P,\ \text{s.t. } \exists PATH(v_i,v_k) \wedge \exists PATH(v_j,v_k) \end{cases}$$

另外，生成出来的自定义指令还需要满足总指令条数和面积约束，这些约束在自定义指令选择问题中处理。

(2)指令簇划分

由于指令执行时，并不是所有的指令都是互相依赖的，可能存在完全不互相依赖的 2 个指令序列，如果在这些指令序列中去除无效的指令，则形成的连通子图将会满足自定义指令除输入输出约束外的要求。因此，可以先将指令序列划分为指令簇，然后，在上面进行约束检查等处理。一个简单指令簇的示意如图 2 所示，图中的序号标示指令执行的顺序，其中，虚线的节点为无效节点，图中的节点可以分为 2 个指令簇[3]。

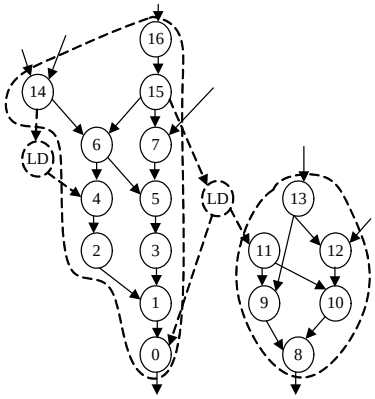

图 2　指令簇示意图

令数据流图为 $G(V,E)$，则指令簇是满足不包含无效节点，且簇内任意 2 点至少存在一条路径；簇内的任何节点不存在与簇外任何有效节点连通路径的最大子图 $R(V^*,E^*)$。

考虑到数据流有一定的传递关系，对数据流图进行标记，然后根据标记进行指令簇划分。簇标记算法如下：

```
for i=0 to num_nodes do
n=g.vGraph[i];
if is ValidNode(n)!=OK then
continue;
end
regmark=(mark[i]==-1)?regionnum+1:mark[i];
regmark=getminregmark();//从它和它的后继中获取最小的簇号
if regmark==regionum+1 then//后继中有被标记的，且它也没有
//被标记，新建一个簇
regmark= regionum++;
end
else//后继中有被标记的，需要更新标记
collectremarknodes(remarklist);
mark[i]=remark;
for j=0 to succ_index do//标记后续节点
mark[successors[j]]=regmark;
end
remark(remarklist,regmark);//更新需要重新标记的节点的标记
end
end
```

(3)子图枚举

在划分了指令簇之后，如果整个簇恰好符合扩展指令的其他约束的要求，则该指令簇就可以作为一条候选的自定义指令。但是，由于指令流中的簇往往会有多个输入输出，很容易违反约束；在违反了约束之后，就需要通过某种方法再将簇细分为更小的子簇才能形成候选指令。另一方面，将整个指令簇作为一条候选自定义指令，其最终获得的性能不一定会总比使用它的一个子簇的性能高(这一点，在后面的自定义指令选择问题中就能清楚地看到)。因此，应该尽可能多地提供候选的自定义子簇，然后由后面的自定义指令选择算法来选择出对总体性能更优的候选。为此，本文设计实现了一个子簇枚举算法，用来枚举所有满足自定义指令约束的非平凡子簇(指令数>1)。

算法的基本思想如下：首先，找出指令簇的数据流图上的各个节点对应的连通子图，然后，对子图逐个进行扩展和拼接，得到新子图。找连通子图时，分别从 2 个不同的方向进行，具体的枚举连通子图的算法如下：

步骤 1　按逆拓扑序标记指令簇中的所有节点。

步骤 2　对指令簇中的所有节点按枚举方向的逆序进行步骤 3~步骤 6 的处理，计算该节点符合约束的连通子图集合。

步骤 3　设当前处理的指令节点 v，则将只含单个节点的子图{v}加入到目标集合里。

步骤 4　对所有与 v 相邻的前驱，计算这些前驱的可能组合。

步骤 5　对组合中的所有元素，合并各元素的连通子图。

步骤 6　根据连通约束和输入输出约束，删除不符合约束的子图，将符合约束的子图加入到目标集合。

(4)子图合并

执行完连通子图枚举算法之后，所有的指令节点将包含 2 个集合，这 2 个集合分别包含了该方向上与之连通的所有连通子图。为了得到指令节点的连通子图，需要合并 2 个方向上的连通子图，具体算法如下：

步骤 1 对指令簇中的所有节点按逆拓扑序进行步骤 2~步骤 5 的处理。

步骤 2 找出当前节点的最大连通子图上可以扩展的节点集，加入 *v* 的待扩展节点集的集合。

步骤 3 如果 *v* 的待扩展节点集的集合为空，则输出该连通子图，转步骤 7。

步骤 4 对于 *v* 待扩展节点集的集合中的每个集合，对其中的节点可能的组合 *V'*，进行步骤 5、步骤 6 的处理。

步骤 5 如果该组合可以消除，则转步骤 4；否则合并原有子图与待扩展节点集合 *V'*在另一个方向的连通子图。

步骤 6 如果合并后子图存在可扩展节点，则将这些节点的集合加入 *V'*的待扩展节点集合，转步骤 4。

步骤 7 删除节点 *v* 在指令簇的集合中出现的所有实例。

查找可扩展集合时，根据与顶点连通的节点进行判断，如果存在所在连通子图以外的节点，则该节点可以扩展；可扩展节点集则是这些可扩展节点进行不同的组合而得到的集合。合并联通子图时则采用与算法 2 类似的方法，以组合枚举的方式逐个合并与其连通的节点在另一个方向上的子图。扩展是一个递归的过程，以深度优先的关系，不断地扩展子图，直到不可扩展为止。为避免重复扩展，算法中还加入了重复检测的机制，即如果可扩展的节点之间在扩展方向上有前驱后继的关系，则不进行后继的扩展。

一个简单的示例如图 3 所示。连通子图枚举时，对于节点 1，向上的枚举将会枚举到的集合为{{1},{1,2},{1,5},{1,5,6},{1,2,5},{1,2,5,6}}；而对节点 4，向下的枚举集合是{{4},{4,3}}；向上的枚举集合是{{4},{4,5},{4,5,6}}。

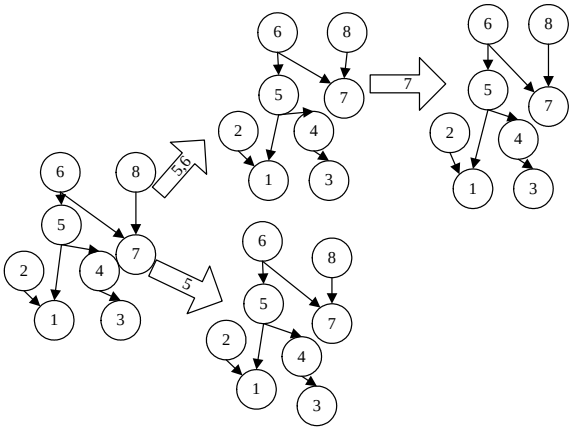

图 3 子图合并示意图

在合并子图时，首先找到的是其最大连通子图{1,2,5,6}，该子图的节点 5、节点 6 有向下的连接，为可扩展节点，可扩展节点集为{5,6}。对于可扩展节点集，有 3 种组合：{5},{6},{5,6}，但是因为在扩展方向上，节点 6 是节点 5 的后继，所以不会进行以节点 6 为扩展节点的扩展，只剩{5},{5,6}2 种扩展可能。扩展{5}时，合并连通子图{3,4}后已无可扩展节点，返回。扩展{5,6}时，先合并了{3,4},{7}子图后，节点 7 可以继续扩展；因此，对节点 7 进行另外一个方向上的扩展，合并了子图{8}；合并后无扩展节点，返回。

(5)算法讨论

算法中较多的操作是子图的合并操作和子图的集合操作。为了支持高效的子图合并，最好使用位向量来表示子图，位向量的长度为整个指令簇的大小，位向量中的每一位代表一个节点，为 1 表示在子图中，为 0 则表示不在子图中。这样的枚举、合并过程，不可避免地会得到重复的子图，检测重复的子图时，需要能高效地查找和插入，可以使用树的结果来保存最终的子图，在查询和插入操作时的时间复杂度可以降低到 $O(\mathrm{lb}(n))$。

3 实验及结果分析

为了检验指令集扩展算法的正确性，设计实现了如图 4 所示的实验框架。使用 VCG 格式保存 Data Flow Graph(DFG)图，并作为指令集扩展算法的输入。采用 VCG 格式便于查看和核对算法执行的结果。实验时采用的指令集是 NIOSII。

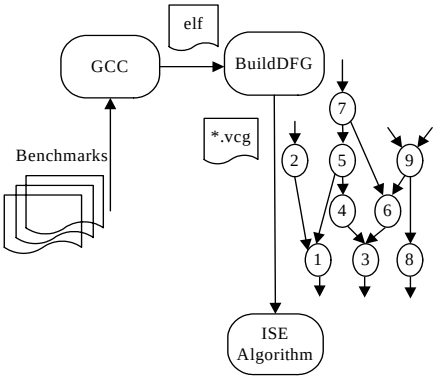

图 4 指令集扩展的实验框架

由于 ASAP 中的指令集扩展算法是应用于基本块上的，有必要先分析应用的特点，以确定算法运行的时空复杂度，因此对 NetBench 和 MiBench 的部分程序进行了剖析，分析了其中的基本块大小。本文统计了基本块大小的分布，大部分应用程序的基本块以小块(0~10 条指令)的居多。最大的基本块大小范围一般为 50~100。基本块大小超过 500 的程序基本都是加密类型的应用。这些数据表明，自定义指令生产算法执行时，其输入的基本块大小一般都比较小，算法的时空复杂度相对比较低，可以在嵌入的优化控制器内实现。选择测试程序中较大的基本块进行指令集扩展实验，大部分应用程序得到的指令簇的大小均在 50 以下，所以，对后面的子图枚举和合并算法的复杂度要求不高。

4 结束语

本文提出的划分合并算法将全部搜索空间分解，在分解的小空间上找出符合条件的集合，将这些子空间解集合归并得到全部搜索空间上的解集合，这种分解合并的方式不会丢失符合条件的组合。该算法可以较好地解决图中节点较多的情况。

参考文献

[1] 王晟中, 陈伟男, 彭澄廉. 可重构计算硬件平台的改进设计[J]. 计算机工程, 2010, 36(5): 250-252.

[2] Yu Pan, Mitra T. Characterizing Embedded Applications for Instruction-set Extensible Processors[C]//Proc. of the 41st Annual Design Automation Conference. San Diego, CA, USA: [s. n.], 2004: 723-728.

[3] Yu Pan, Tulika M. Efficient Custom Instruction Identification with Exact Enumeration[R]. Singapore: National University of Singapore, Technical Report: TRB5/07, 2007.

编辑 索书志