摘 要

与传统的DSPs相比，现代DSPs采用了更多的ILP技术以提高性能。同时，为了适应媒体处理的特殊需要，现代的DSPs在硬件的设计方面的一个普遍的特征就是引入了SIMD指令。这就使得低精度的媒体数据的快速处理成为可能。一个真正的高性能、低功耗的系统，不仅要有良好的硬件支持而且更重要的是要有一个能够充分利用这些硬件特征的优化的软件系统。编译器作为这个软件系统中最重要的一环，其性能的优劣将直接影响系统的整体性能。但目前的编译技术却不能很好的提供对SIMD指令的支持，所以本文就深入的研究了支持SIMD指令的编译优化技术，取得了如下的一些研究成果。

1. 提出了一套完整的支持SIMD指令的代码选择技术的实现框架。该框架使得代码选择面向基本块，而不再是语句，扩大了指令注释的搜索空间，产生了SIMD 指令。同时采用机器描述来刻画目标机器指令集，使得代码选择中与机器相关的部分局限于机器描述中，提高了代码选择的灵活性和可移植性。

2. 设计并实现了一种基本块的数据流图（DFG）表示，这种表示既能充分表达Lcode语义又比较适合模式匹配算法。传统的代码选择技术往往基于的是数据流树（DFTs）的中间表示，这也正是其不能很好的支持SIMD指令的一个重要原因。同时为了借鉴模式匹配的思想，在这里还实现了从DFG到DFTs的转化。

3. 采用一种树文法描述了目标机器的指令集，并对其进行预处理，使其转化为相应的指令模板。树文法为目标机器指令集的描述提供了一种易读、易写、易修改的方式，同时通过预处理隐藏了机器描述的细节，为编译的树匹配过程提供了统一的接口。

4. 改进了传统的树匹配和动态规划算法。使得其在匹配每棵输入树时，不再是产生唯一的最优覆盖，而是产生多个可选的最优覆盖。为整数线形规划从全局角度最终确定最优覆盖做准备。

5. 采用了整数线形规划的方法解决了最终的最优覆盖的选择问题。在充分考虑最终覆盖的正确性和有效性的情况下，提取出其所应满足的一系列约束及相应的目标方程，该方程的最大解即表示最终将产生的SIMD指令的数目。

关键词：代码产生　树匹配和动态规划　整数线性规划　数据流图　数据流树　SIMD指令

ABSTRACT

Compared with traditional DSPs, modern DSPs use more ILP technologies to improve their performance. Meanwhile, in order to meet the special need of media processing, modern DSPs have a common characteristic in the hardware designing and implementation – SIMD. So it is possible that the lower precision media data is processed quickly. a true system which is higher performance and lower power consumption needs not only good hardware supporting but also optimized software that may make full use of the hardware. Beside others , compiler is the most important element. Its performance will influent the total performance of the entire system directly. But current compiler doesn't have the ability to support these SIMD instructions smoothly. So in this thesis, we do a deep research on compiler optimizing techenology which is support of SIMD, making main contributions as follows.

1.A whole implementation framework of code selection which is support of SIMD has been advanced. The framework makes code selection not face single statement but a basic block., expanded the searching space of code selection ,generated the instructions of SIMD. Meanwhile,the instruction set of target machine is characterized by machine description. It makes the part that is dependence of machine be constrained to machine description and makes the code selector more flexibility and retargetability.

2.A DFG representation of basic block is designed and implemented, the representation not only expresses the semantic of Lcode sufficiently but also adapts to the need of tree matching algorithm. The traditional code selection thechnology is always based on the middle representation of DFT. This is one of main reasons that it can not suppot of SIMD.

3.A tree grammar is used to describe the instruction set of target machine,the description is preprocessed to generate the corresponding instruction template. The tree grammer provides a way that is easy to read, modify and write for the description of target instruction set.meanwhile,hiding the detail of the description through preprocess module,so providing the general interface for process of tree matching.

4.Improved on the traditional tree matching with dynamic programming algorithm. For each subject tree,it generates not a single optimizing cover but many alternative covers. This step makes a good foundation for integer linear programming which is

used to decide the final cover.

5.Using integer linear programming method resolves the choice of final cover. Considering the correctness and efficiency fully, we have abstracted a series of constraints and a corresponding target function,the maximum solution means the count of SIMD that will be generated.

Keywords: Code Generation, tree matching with dynamic programming, integer linear programming, data flow graph, data flow tree , SIMD instruction

图目录

表目录

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：支持SIMD的DSP编译优化技术的研究和实现

学位论文作者签名：赵常智　　日期：2007 年 11 月 14 日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：支持SIMD的DSP编译优化技术的研究和实现

学位论文作者签名：赵常智　　日期：2007 年 11 月 14 日

作者指导教师签名：　　日期：2007年 11月14日

第一章 引言

§1.1 课题背景与意义

在当今这个信息技术高速发展的时代，DSP 技术日益在嵌入式系统中得到了广泛的应用，因此便对 DSP 系统的开发提出了新的更急迫的需求[11]。而在 DSP 系统开发中，性能、芯片大小及功耗三个方面的问题是我们首先必须关注的。为此，DSP 在硬件设计方面为实现这些目标作出了巨大的努力，比如广泛采用 EPIC 和 VLIW 的体系结构，为 ILP(instruction level parallel)提供更好的硬件支持；由于 DSP 的一个重要应用是媒体处理，而媒体处理最大的特点就是数据具有高度的并行性，同时在媒体处理中通常也只要求 8 位或 1 6 位的数据精度，然而，许多媒体处理器具有 32 位字长，这就潜在的存在着计算资源浪费的可能。为了解决这一问题，许多媒体处理器采用一类特殊的指令—SIMD(single instruction multiple data) 指令，它允许把一个物理寄存器分成几个虚拟的子寄存器，然后在这几个虚拟的子寄存器上并行执行相同操作。在此尤其应该引起我们注意的是当今许多高性能 DSP 芯片都提供了对 SIMD 指令的支持。我们自行研制的 YHFT-DSP/700 便是这样的一款。有了这样的硬件支持以后，如果想真正的得到一个高性能、低功耗的系统，就必须构造能充分利用硬件特征的优化的软件系统。固然直接利用汇编语言来编写应用程序可以在很大程度上利用机器的硬件特性，但在当前直接用汇编语言来编写应用程序的方法正越来越不被人们采用。这主要是因为用汇编语言来编写应用程序是费时的、低效的而且还会导致应用程序的不可移植的问题。这就要求我们构造的编译器必须具有能够把用高级语言编写的应用程序转化为能充分利用机器硬件特征的汇编代码的能力。因此本论文将主要介绍充分支持处理器硬件特征—SIMD 指令的编译优化技术。支持 SIMD 指令的编译技术是本文的重要研究内容，同时本文也涉及 SIMD 指令对低功耗编译技术的影响。有研究表明 SIMD 指令在在提高 DSP 性能和降低功耗方面的作用巨大。例如德国的多特蒙德大学的 Rainer Leupers 等人已经为 Texas Instrument C62xx 和 Philips Trimedia TM1000 这两种 DSP 芯片实现了支持 SIMD 的指令选择[3][2]。他们得到的具体的实验结果如下：

表 1-1 在两款芯片上的试验结果

source	type	unroll	no SIMD	SIMD	CPU sec
TI C62XX					
vector add	short	1	8	4	0.7
IIR filter	short	0	21	17	2.9
Convolution	short	1	8	6	0.6
FIR filter	short	1	15	11	0.9
N complex update	short	1	20	16	3.0
Image compositing	short	1	14	11	3.1
Trimedia TM1000					
vector add	short	1	8	4	0.7
IIR filter	short	0	22	22	5.1
Convolution	short	1	8	8	0.9
FIR filter	short	1	15	9	0.9
N complex update	short	1	20	20	4.7
Image compositing	short	1	14	7	3.2
vector add	char	3	16	4	5.0
FIR filter	char	3	36	18	26.5

从上面的实验结果[17]可以看出利用 SIMD 指令可以使所需指令的数目在很大程度上得到减少，其中对于向量加可以减少 50%。同时多特蒙德大学的 Markus Lorenz 和 Peter Marwedel 等也首次对 SIMD 指令对功耗的影响进行了研究（基于 M3-DSP 这样一款 DSP 芯片），其研究结果表明：虽然一条 SIMD 指令消耗的能量很可能是一条 SISD 指令消耗能量的一倍或几倍，但若充分利用 SIMD 指令可以达到能量消耗平均减少 72%而执行时间平均减少 76%的效果[7][34][23]。由此不难看出，构造能充分的挖掘和利用 SIMD 指令的优化的编译器便显得至关重要了。而事实上对支持 SIMD 指令的编译技术的研究也是编译技术研究的一个重要的组成部分[35]。

§1.2 本文的贡献

本文的贡献主要有以下的几个方面：

第一：提出了在现有的编译框架下支持 SIMD 指令的完整的算法。面向 DSP 的编译器设计与面向通用处理器的编译器设计有很大的不同，其中很重要的一个方面就是由于 DSP 处理器在体系结构上的特殊性使得经典的编译技术不断受到挑战。SIMD 指令的出现就属于这种情况，而且伴随着 DSP 处理器的广泛使用，支持 SIMD 指令的编译技术的研究已经成为亟待解决的理论和技术问题。

第二：设计和实现了一个可移植的代码选择算法。现代的编译器日益重视其可移植性和可重构性。因此，在这里我们用机器描述语言描述[25][26][27][33]了从中间

语言到更低级语言的映射，由预处理器[15][34]将它装换成相应的模式集，代码生成器将中间语言与这个模式集作匹配，在匹配过程中生成更低级代码。匹配算法是与机器无关的，代码生成中与机器相关的成分都集中在模板描述中，这就使得代码生成算法和机器特征隔离开来。这使得它有良好的可移植性。

第三：设计了一种与代码生成算法相适应的数据流图中间表示。在 IMPACT C 的编译器中，它采用了一种类似三地址码的中间表示—Lcode。但在我们的代码生成算法中，却要求基于一种 DFG 图中间表示。在这里把 Lcode 转化为什么样的 DFG 图表示，既要考虑它们的语义等价性和目标指令集的特征更重要的是要满足树匹配和动态规划算法的要求。

第四：设计并实现了由基本块构造 DFG 表示的算法，同时为了借鉴树匹配和动态规划算法的思想，通过消除公共子表达式把 DFG 分解为一系列 DFTs。

第五：用整数线形规划的方法解决了最终覆盖的选择问题，提取出了生成 SIMD 指令所需满足的约束以及表达最大限度生成 SIMD 指令的目标方程。

§1.3 论文结构

第二章叙述了目标机器 VLIW DSP 体系结构特征以及所采用的 IMPACT 编译框架，重点讨论了这款芯片所引入的 SIMD 指令集合以及 IMPACT 编译器的中间表示 Lcode。这二者构成了本文讨论的出发点。第三章分析了传统的代码产生算法的思想，指出它们之所以不能直接支持 SIMD 指令产生的原因。同时针对这些原因提出了相应的解决对策，给出了支持 SIMD 指令的编译框架。第四章介绍了这个框架中最前端的两个并列的过程，关于描述目标机器指令集的机器描述问题以及关于如何设计 DFG 图[3]表示和如何如何把 Lcode 表示转化为 DFG 图表示的问题。第五章叙述了树匹配和动态规划算法的思想以及为支持 SIMD 指令和便于算法实现所作的相应改进。第六章叙述了产生 SIMD 指令所必需满足的约束以及如何根据这些约束生成相应的目标方程和约束不等式，并用 lp_solver 来解这个方程，最后根据计算结果作相应的语义动作。第七章为工作小结以及关于未来工作的一些设想。

第二章 SIMD 指令及其编译器

本章主要叙述了两方面的内容，其一：介绍了 SIMD 指令的基本特征以及 VLIW DSP 所支持的几种 SIMD 指令的语法和语义。其二：介绍了 IMPACT 编译器的编译框架，其中主要介绍了其中间表示 Lcode[9][30]。

§2.1 SIMD 指令概述

2.1.1 基本概念

如果一条指令是对存储在子寄存器而不是整个寄存器中的数据进行操作，我们就说这条指令是 SIMD 指令。 为了利用 SIMD 指令，通常把一个 32 位的数据寄存器看成是两个 16 位子寄存器或是四个 8 位子寄存器。从而，对 C 语言来说，一个 32 位寄存器可以存放四个 char 型的数据或两个 short 型的数据或一个 int 型的数据。图 2-1 给出一个 YHFT-DSP/700 的 SIMD 指令的例子—ADD2。

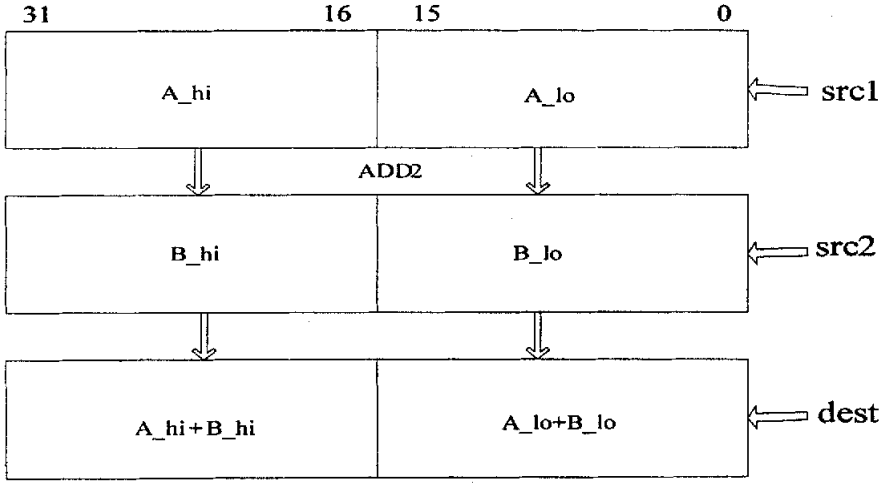

图 2-1 ADD2 指令的功能示意图

为了充分的利用 SIMD 指令，把被操作的八位或十六位的数据高效的从存储器中读出和写入是至关重要的，在某些条件下，可以利用 32 位的操作来取操作数和向存储器写 SIMD 指令的计算结果。正如下面的例子所示：

```
void f (short* A,short* B,short* C)
{ int i;
for (i = 0; i < N; i += 2)
```

```
{
A[i] = B[i] + C[i];
A[i+1] = B[i+1] + C[i+1];
}
}
```

这是一个 short 类型数据的向量加法，为了开发潜在的并行性，循环体已经被循环展开了一次。在这里应用上面的 ADD2 指令可以使循环体中的两个加法并行执行。然而这要求把 B[i]，C[i]和 B[i+1]，C[i+1]分别放置在两个参数寄存器的低半部分和高半部分，因此 B[i]和 B[i+1]必须用一个 32 位的 load 指令来取而不应该用两个 16 位的 load 指令，同样的对 C[i]和 C[i+1]也应如此。这也正是因为相邻的数组元素被存储在相邻的内存单元内的缘故。在 ADD2 指令执行以后，它的运行结果也可以用一个 32 位的 store 操作把 A[i]和 A[i+1]放置到目的寄存器的低半部分和高半部分。很容易看出利用了 SIMD 指令，执行向量和运算所需的指令数减少了 50%.执行过程如图 2-2 所示：

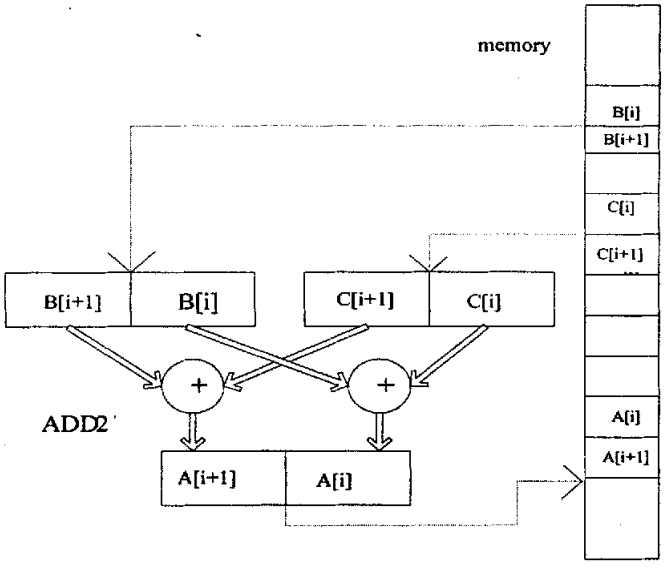

图 2-2 利用 SIMD 指令进行并行的向量加

2.1.2 YHFT-DSP/700 支持的 SIMD 指令

YHFT-DSP/700 提供了如下的 SIMD 指令，下面分别介绍其语法及语义：

ADD4 四对八位数相加，结果为四个八位数。

其语法格式为：ADD4 (.unit) src1, src2, dst

表 2-1 ADD4 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1	i4	.L1,.L2	1100101
src2	xi4		
dst	i4		

在上表中，i4 表示 4 个八位的整数，在操作数的类型前加上一个 x 表示该操作数从一个和目的寄存器不同的寄存器文件中读出，在下文的语义相同。该指令的执行过程如下所示：

```
if (cond)  {
           byte0 (src1) + byte0 (src2)  —> byte0 (dst)
           byte1 (src1) + byte1 (src2)  —> byte1 (dst)
           byte2 (src1) + byte2 (src2)  —> byte2 (dst)
           byte3 (src1) + byte3 (src2)  —> byte3 (dst)
           }
else       nop
```

该指令不执行饱和操作，其中任意一个 8 位加法的进位不影响其余 8 位加法的执行。该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄存器写入结果，故其延时槽为 0。

SUB4 不带饱和，捆绑的 8 位有符号数的减法。

其语法格式为：SUB4 (.unit) src1, src2, dst

表 2-2 SUB4 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1	i4	.L1,.L2	1100110
src2	xi4		
dst	i4		

该指令的执行过程如下：

```
if (cond)  {
           (byte0 (src1) - byte0 (src2))  —> byte0 (dst);
           (byte1 (src1) - byte1 (src2))  —> byte1 (dst);
           (byte2 (src1) - byte2 (src2))  —> byte2 (dst);
           (byte3 (src1) - byte3 (src2))  —> byte3 (dst);
           }
else       nop
```

该指令不执行饱和操作，其中任意一个 8 位减法的借位不影响其余 8 位减法的执行。该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄存器写入结果，故其延时槽为 0。

ADD2 2 对 16 位数相加，结果为 2 个 16 位数。

其语法格式为：ADD2 (.unit) src1, src2, dst

表 2-3 ADD2 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	i2 xi2 i2	.L1,.L2	0000101
src1 src2 dst	i2 xi2 i2	.D1,.D2	0100

在上表中，i2 表示两个 16 位整数，x 的含义与上文相同。该指令的执行过程如下所示：

```
if (cond)   {
            msb16（src1）+ msb16（src2）  —> msb16（dst）
            lsb16（src1）+ lsb16（src2）  —> lsb16（dst）
            }
else        nop
```

该指令不执行饱和操作，其中任意一个 16 位加法的进位不影响其余 16 位加法的执行。该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄存器写入结果，故其延时槽为 0。

SUB2 不带饱和，捆绑的 16 位有符号数的减法。

其语法格式为：SUB2 (.unit) src1, src2 ,dst

表 2-4 SUB2 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	i2 xi2 i2	.L1,.L2	0000100
src1 src2 dst	i2 xi2 i2	.D1,.D2	0101

该指令的执行过程如下：

```
if (cond)   {
            （lsb16（src1）— lsb16（src2））  —> lsb16（dst）;
            （msb16（src1）— msb16（src2））  —> msb16（dst）;
            }
else        nop
```

该指令不执行饱和操作，低 16 位减法的借位不影响高 16 位减法的执行。该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄存器写入结果，故其延时槽为 0。

SADD2 带饱和操作的捆绑 16 位有符号数的加法。

其语法格式为：SADD2 (.unit) src1, src2, dst

表 2-5 SADD2 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	s2 xs2 s2	.S 1 ,.S2	0000

在上表中，s2 表示 2 个有符号数，该指令的执行过程如下：

```
if (cond)   {
            sat (msb16 (src1) + msb16 (src2))  —> msb16 (dst);
            sat (lsb16 (src1) + lsb16 (src2))  —> lsb16 (dst);
            }
else        nop
```

该指令执行饱和操作，低 16 位的进位不影响高 16 位的执行。对每一个 16 位的结果均执行饱和操作是指它满足以下规则：

如果加法的和在－215 至 215－1 之间，无需执行饱和操作，结果保持不变；

如果加法的和大于 215－1，则结果置为 215－1；

如果加法的和小于－215，则结果置为－215。

该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄存器写入结果，故其延时槽为 0 。

SADDU4 带饱和操作的捆绑 8 位无符号数的加法。

其语法格式为：SADDU4 (.unit) src1,src2, dst

表 2-6 SADDU4 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	u4 xu4 u4	.S 1 ,.S2	0011

在上表中，u4 表示四个 8 位无符号整数，该指令的执行过程如下：

```
if (cond)   {
            sat (ubyte0 (src1) + ubyte0 (src2))  —> ubyte0 (dst);
            sat (ubyte1 (src1) + ubyte1 (src2))  —> ubyte1 (dst);
            sat (ubyte2 (src1) + ubyte2 (src2))  —> ubyte2 (dst);
            sat (ubyte3 (src1) + ubyte3 (src2))  —> ubyte3 (dst);
            }
else        nop
```

该指令执行饱和操作，其中任意一个 8 位加法的进位不影响其余 8 位加法的执行。对每一个 8 位的结果均执行饱和操作是指它满足以下的规则：

如果加法的和在 0 至 28－1 之间，无需执行饱和操作，结果保持不变；

如果加法的和大于 28－1，则结果置为 28－1。

该指令在流水线的 E1 阶段读取源操作数 src1 和 src2 并在这一阶段向目的寄

存器写入结果，故其延时槽为0。

SADDUS2 带饱和操作的捆绑16位无符号数与有符号数的加法。

其指令格式为：SADDUS2 (.unit) src1, src2, dst

表 2-7 SADDUS2 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	u2 xs2 u2	.S1,.S2	0001

在上表中，u2表示两个捆绑的16位无符号整数，s2表示两个捆绑的16有符号整数，指令的执行结果为两个捆绑的16位无符号数。其执行过程如下：

```
if (cond)   {
            sat (umsb16 (src1) + smsb16 (src2))  —> umsb16 (dst);
            sat (ulsb16 (src1) + slsb16 (src2))  —> ulsb16 (dst);
            }
else        nop
```

对每一个16 位的加法均执行饱和操作，即每一个16加法满足如下规则：

如果加法的和在0至216−1之间，无需执行饱和操作，结果保持不变；

如果加法的和大于216−1，则结果置为216−1；

如果加法的和小于0，则结果置为0；

该指令在流水线的E1阶段读取源操作数src1和src2并在这一阶段向目的寄存器写入结果，故其延时槽为0。

SADDSU2 带饱和操作的捆绑16位有符号与无符号加法（伪指令）。

其指令格式为：SADDSU2 (.unit) src2, src1, dst

该指令为一条伪指令，汇编器往往用SADDUS2 (.unit) src1,src2,dst来代替上述指令。

MPY2 不带饱和操作的捆绑16位有符号数乘法。

其指令格式为：MPY2 (.unit) src1,src2,dst

表 2-8 MPY2 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	s2 xs2 ullong	.M1,.M2	00000

在上表中，ullong表示两个long型的有符号数，其执行过程如下：

```
if (cond)   {
              lsb16(src1) × lsb16 (src2) —> dst_e
              msb16 (src1) × msb16 (src2) —> dst_o
            }
else  nop
```

该指令在流水线的E1阶段读取源操作数，并在E2阶段向目的寄存器写入结

果，故它的延时槽为 1。

MPYSU4 不带饱和操作的捆绑 8 位有符号数与无符号数乘法。

其指令格式为：MPYSU4 (.unit) src1,src2 ,dst

表 2-9 MPYSU4 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	s4 xu4 dws4	.M1 ,.M2	00101

在上表中，dws4 表示放置 4 个计算结果的双字，其中每个计算结果都为 16 位有符号整数。该指令的执行过程如下：

```
if (cond)  {
                sbyte0(src1) × ubyte0(src2) —> lsb16(dst_e)
                sbyte1(src1) × ubyte1(src2) —> msb16(dst_e)
                sbyte2(src1) × ubyte2(src2) —> lsb16(dst_o)
                sbyte3(src1) × ubyte3(src2) —> msb16(dst_o)
           }
else nop
```

该指令在流水线的 E1 阶段读取源操作数，并在 E2 阶段向目的寄存器写入结果，故它的延时槽为 1。

MPYU4 不带饱和操作的捆绑 8 位无符号数乘法。

其指令格式为：MPYU4 (.unit) src1, src2, dst

表 2-10 MPYU4 指令描述

使用的操作码映射域	操作数类型	功能单元	操作码域
src1 src2 dst	u4 xu4 dwu4	.M1 ,.M2	00100

其中每个计算结果都为 16 位的无符号整数。该指令的执行过程如下：

```
    if (cond)  {
                    ubyte0(src1) × ubyte0(src2) —> lsb16(dst_e)
                    ubyte1(src1) × ubyte1(src2) —> msb16(dst_e)
                    ubyte2(src1) × ubyte2(src2) —> lsb16(dst_o)
                    ubyte3(src1) × ubyte3(src2) —> msb16(dst_o)
               }
```

该指令在流水线的 E1 阶段读取源操作数，并在 E2 阶段向目的寄存器写入结果，故它的延时槽为 1。

§2.2 IMPACT 编译器简介

IMPACT 编译器是可重定目标的优化 C 编译器[12]，由 Illinois 大学 IMPACT

研究小组开发。为了有助于评估各种编译技术的好坏，IMPACT 的设计非常模块化[33][32][31]。编译器实际上被分成了多个阶段，每个阶段有不同的输入和输出文件。这些中间文件由一系列的 shell 脚本与 IMPACT 的各模块相连接。因此本节的第一部分着重介绍编译器的整体框架。Lcode 是 impact 编译器前端产生的与机器无关的中间代码，为三地址形式，类似于 RISC 指令。代码产生模块是基于 Lcode 中间表示进行的，SIMD 指令的产生也不例外。因此本节的第二部分着重来介绍 Lcode 中间表示。

2.2.1 编译器整体结构

针对目标机器 YHFT-DSP/700 的特性，面向 YHFT-DSP/700 目标机器的编译程序的编译前端采用 IMPACT 的前端[22]，通过 IMPACT 前端进行词法、语法等分析和机器无关的优化，最终生成机器无关的中间代码文件 Lcode[18]。图 2-3 是我们编译程序的总体结构图。

如图 2.3 所示，我们的编译程序主要分成两个大的部分：编译前端和编译后段。其中前端主要由与目标机无关的模块组成，编译后端则是由一系列与目标机器 YHFT-DSP/700 相关的模块组成，主要分成三个阶段(phase)：代码选择(代码注释)，优化、指令调度、寄存器分配和汇编代码生成。第一个阶段处理前端生成的机器无关中间代码 Lcode，生成机器相关的中间代码 Mcode；第二阶段为每条指令安排执行功能单元和执行时间；第三阶段把 Mcode 最终转化为 YHFT-DSP/700 的汇编代码。其中前两个阶段通过和机器描述(MDES)、机器规格说明(Mspec)进行交互获得目标机器 YHFT-DSP/700 的相关信息进行相应的工作。在现有的代码选择方法中，它采用了逐条扫描 Lcode 并把它转换为相应得 Mcode 的方法，它是一种硬编码的方法。采用这种方法的编译器不仅可移植性较差而且根本不能支持 SIMD 指令。因此本论文的工作主要从代码选择技术入手，寻求一种能提高代码选择器的可移植性同时更为重要的是能够充分挖掘程序中潜在的并行性，能够充分支持 SIMD 指令的代码选择技术。而后还需要对优化、指令调度、寄存器分配和汇编代码生成部分进行相应的改进，使得生成的汇编代码真正包含 SIMD 指令。

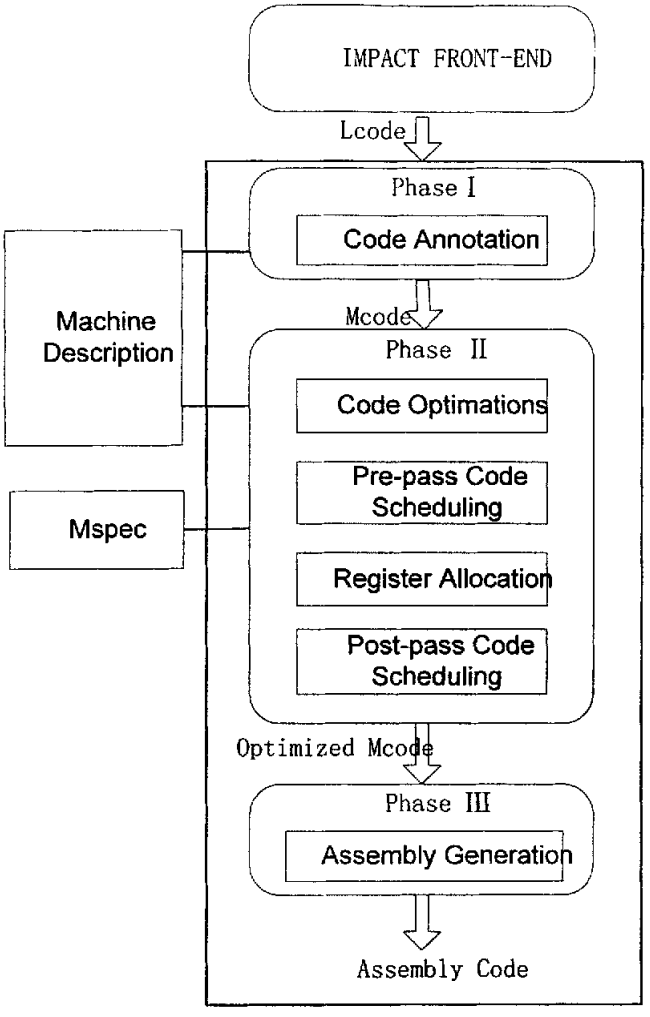

图 2-3 编译程序整体结构图

2.2.2 Lcode 中间表示

它分为两种表示：内部表示和外部表示。下面分别对这两种表示进行说明。

2.2.2.1 内部表示

Lcode 分数据和代码两部分，这里只介绍代码部分，其内部数据结构自顶向下依次有函数（L_Func）、控制块（L_Cb）、流（L_Flow）、指令（L_Oper）、属性(L_Attr)和操作(L_Operand).下面分别介绍各个数据结构如下：

2.2.2.1.1 函数（L_Func）

函数由控制块组成，它的数据结构见图 2-4。name 字段为函数名，first_cb 与 last_cb 分别指向函数的第一个和最后一个控制块，函数的所有控制块构成一双向

链表。Attr 为属性函数。

```
typedef struct L_Func {
    char     *name;       /* name of function */
    L_Cb     *first_cb;   /* first sequential cb */
    L_Cb     *last_cb;    /* last sequential cb */
    L_Attr   *attr;       /* additional attributes */
} L_Func;
```

图 2-4 Lcode 函数的数据结构

2.2.2.1.2 控制块(L_Cb)

控制块由操作组成，它的数据结构如图 2-5 所示。id 是控制块的唯一标识，first_op 和 last_op 分别指向控制块的第一个和最后一个操作，控制块的所有操作构成一双向链表。attr 为块属性，属性能以简要的形式保存控制块的辅助信息。src_flow 和 dest_flow 则是保存控制块之间的控制流信息，dest_flow 描述的是在程序中当前控制块的后继控制块，即当程序执行完本控制块之后应跳到那个控制块继续执行；而 src_flow 则描述了当前控制块的前继控制块，即当那个控制块执行完了以后可能跳到当前的控制块执行。prev_cb 和 next_cb 用于构成控制块双向链表。

```
typedef struct L_Func {
    char     *name;       /* name of function */
    L_Cb     *first_cb;   /* first sequential cb */
    L_Cb     *last_cb;    /* last sequential cb */
    L_Attr   *attr;       /* additional attributes */
} L_Func;
```

图 2-5 Lcode 控制块的数据结构

2.2.2.1.3 流(L_Flow)

控制流从属于控制块，它由 L_Flow 数据结构描述，如图 2-6 所示，字段 cc 给出分支条件(跳转还是不跳转)。src_cb 和 dest_cb 分别用于标记相应于该 flow 的源和目的控制块。prev_flow 和 next_flow 用于构成 flow 的双向链表。

```
typedef struct L_Func {
    char     *name;       /* name of function */
    L_Cb     *first_cb;   /* first sequential cb */
    L_Cb     *last_cb;    /* last sequential cb */
    L_Attr   *attr;       /* additional attributes */
} L_Func;
```

图 2-6 Lcode 控制流的数据结构

2.2.2.1.4 指令(L_Oper)

Lcode 指令的内部数据结构由 L_Oper 给出，如图 2-7 所示。其中 id 表示本指令在控制块中的编号。proc_opc 是该 Lcode 指令相对应的目标机器指令的操作码。Opcode 则表示 proc_opc 对应的操作码助记符。prev_op 和 next_op 分别保存的是该指令在当前控制块内的前一条和后一条指令，从而构成控制块内指令的一个双向链表。

```
typedef struct L_Oper {
    int      id;          /* unique id of oper */
    int      opc;         /* integer representation of opcode,
                             only to be used for internally
                             defined opcodes */
    char     *opcode;     /* name of the operation */
    int      proc_opc;    /* processor specific opcode */
    L_Operand    **dest;  /* ptr to array of ptrs to dest operands */
    L_Operand    **src;   /* ptr to array of ptrs to src operands */
    L_Attr       *attr;   /* additional attributes */
    struct L_Oper    *prev_op;    /* previous oper in sequential mode */
    struct L_Oper    *next_op;    /* next oper in sequential mode */
} L_Oper;
```

图 2-7 Lcode 指令的数据结构

2.2.2.1.5 操作数(L_Operand)

Lcode 操作数的内部数据结构由 L_Operand 给出，如图 2-8 所示。L_Operand 用来描述指令的操作数的类型和它的内容。type 字段表明操作数是否为寄存器、macro、标号、字符串指针、立即数，等等。如果 type 字段为寄存器或是 macro，则 ctype 表示它的内容的 C 语言类型。字段 value 是联合类型，引用它的哪个条目要看 type 字段为何值。

```
typedef struct L_Operand {
    unsigned char   type;   /* operand type */
    unsigned char   ctype;  /* data type */
    union {                                                    /
        struct L_Cb *cb;    /* control block (branch target) */
        int     i;      /* integer constant */
        float   f;      /* float constant */
        double  f2;     /* double constant */
        int     mac;    /* macro (special register) */
        char    *s;     /* string constant */
        int     r;      /* register */
        char    *l;     /* label constant */
    } value;
} L_Operand;
```

图 2-8 Lcode 操作数的数据结构

2.2.2.2 外部表示

每个 Lcode 文件包含下面两个大的部分：

1)数据部分：该部分由(ms data)表示开头，在数据部分中定义了静态和动态的变量、数据对齐方式、分配空间的大小、保留的空间的大小、数据访问的范围、数据类型等信息。

2)代码部分：该部分则是以(ms text)开头，在其中包含了当前 Lcode 文件中的代码信息、函数信息、控制块信息、操作信息等。

下面主要对代码部分中的控制块信息和操作信息加以说明。

2.2.2.2.1 控制块(cb)

控制块由一系列操作组成；控制块只有一个入口但可以有一个或多个出口，如果只有一个入口和一个出口，则这个控制块就是编译中的基本块(basic block)；一个具有多个出口的控制块是一个超级块(super block).图 2-9 是一个典型的控制块图。

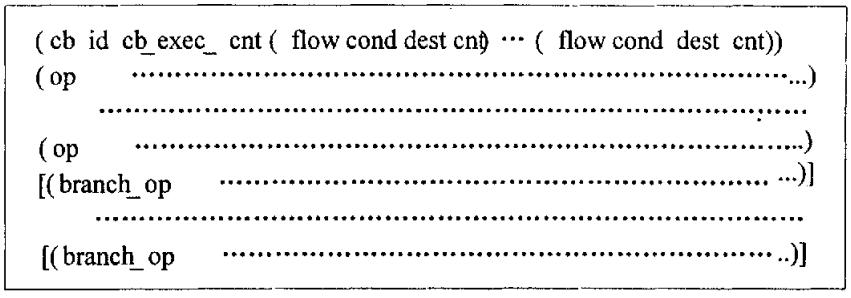

图 2-9 控制块结构视图

控制块由前缀 cb 开头，紧接在其后的是一个在本地函数中唯一的一个控制块序号(id)和该控制块的执行次数 cb_exec_cnt(该信息可以通过 profile 或静态分析得到)。紧跟其后的是流(flow)信息,流信息由四个基本元素组成：flow 前缀、该流执行的条件 cond、目的控制块号 dest 以及该分支流执行的次数 cnt。在函数中，通常一个流（flow）信息对应于一条分支操作，流信息的 cond 为 1 时，目的控制块号表示该分支操作执行成功时跳转到的控制块；cond 为 0 时，目的控制块为该分支失败时将执行的控制块。控制块除了上面说的流信息外，主要由一系列的操作(op)信息组成。下面来阐述操作信息。

2.2.2.2.2 操作（operation）

操作以 op 前缀开头，余下主要由四个部分组成：操作编号、操作码信息、操作数信息和属性信息。操作编号是该操作在所在函数中的唯一标识。图 2-10 是一个典型的 Lcode 指令的操作格式。

op id opcode flags pred dest sources attributes

图 2-10 典型的 Lcode 指令格式

在Lcode中，每个操作最多可以使用2个目的操作数和3个源操作数。属性信息主要描述了该操作的一些具体信息，如该操作相对应的目的机器指令的操作码、调度时间等。

第三章 支持SIMD指令的代码选择技术

本章叙述了传统的经典的代码产生技术的核心思想，指出其不能直接用于产生SIMD指令的原因，在此基础上提出一套完整的支持SIMD指令的编译实现方案[10]。3.1节主要介绍了三种经典的代码选择算法，并比较了它们的优劣。3.2节叙述了传统编译器对SIMD指令的简化处理[9]，并分析了传统代码产生技术不能直接支持SIMD指令的原因[2]。3.3节在前两节的基础上提出了一套完整的支持的SIMD指令的实现方案。

§3.1 经典的指令选择算法

代码生成的任务是把给定的输入语法结构转换成更低级的表示形式或汇编语言表示，即把中间语言所表示的操作码和操作数映射到更低级表示或汇编语言上去。由于目标机是多种多样的，为每一种目标机都从零开始构造一个代码生成系统，需要做大量重复而琐碎的工作。针对这个问题，从八十年代开始了自动代码生成的研究，考虑如何减少移植代码生成器中的工作量，同时保证能够生成具有较高效率的代码。在这个探索过程中，产生了一批优秀的成果。

3.1.1 解释型代码生成(Interpretative Code Generation)

这是出现的最早的一种方法，它的基本思想是定义一个虚拟的目标机，由编译器的前端把源语言映射到虚拟的目标机语言上去，然后再对这个虚拟的目标机生成代码。引进了中间语言后，增加了编译器的遍数，但它带来的好处也是巨大的。它区分了机器相关和机器无关这两大部分，使得优化可以在与机器无关的层面上进行，增大了前端代码的可移植性。也有利于编译工程的管理。但实际上这种解释型的代码生成的思想只是提出了一种科学的解决代码生成问题的编译框架，并不是一种具体的实现算法。而其它的代码生成技术都是在这种思想的指导下进一步深化的结果。这种方法的结构视图如图3-1所示：

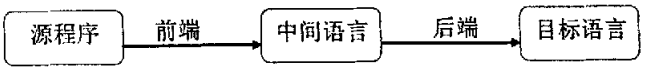

图3-1 解释型代码生产的结构视图

3.1.2 模式匹配代码生成

模式匹配代码生成(Pattern Matching Code Generation)在解释型代码生成的基础上有了较大的提高。它用机器描述语言描述了中间语言到汇编语言的映射，由预处理器将它转换成包含相应模式集的代码生成器，代码生成器将中间语言与这个模式集作匹配，在匹配过程中生成更低级的代码或汇编代码。在这里匹配算法是与机器无关的，代码生成中与机器相关的成分都集中在模板描述中。这种方法的结构视图如图 3-2 所示：

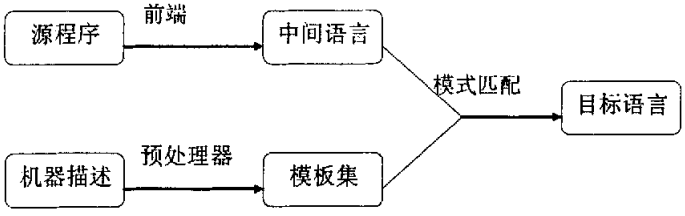

图 3-2 模式匹配方法的结构视图

3.1.3 表驱动代码生成

表驱动代码生成(Table Driven Code Generation)借鉴了前端扫描输入串完成语法分析的方法。首先用上下文无关文法给出机器指令集的描述，由一个预处理器对它识别，生产一个状态-输入的语义动作表，然后它把输入的表达式树看成是一个输入串，在扫描输入树时，按语义动作完成诸如移进，归约等状态转换的语义动作，生产相应的更低级的代码或汇编代码。这种方法的结构视图如图 3-3 所示：

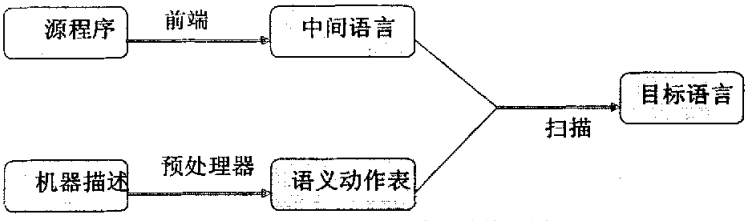

图 3-3 表驱动的代码生成方法的结构视图

3.1.4 两种代码生成方法优缺点的比较

在对后两种代码生成方法进行了深入的研究和探讨之后，发现它们的主要差别表现在以下几个方面：

1)模式匹配方法中的匹配过程事实上是一个先自底向上，而后再自顶向下的一个推导过程，而表驱动方法可以是自顶向下的推导，也可以是自底向上的归约。

2)模式匹配方法依次匹配的是一个模式，匹配一种模式失败后就匹配下一个模式，后来的匹配过程不继承前面匹配过程的结果。而表驱动方法一次匹配事实上尝试了所有可能与之匹配的语法规则，它优于一次尝试一种匹配。

3)表驱动方法要求把机器指令描述成一个 LR(1)的文法，但是描述机器指令的文法特别是由于 DSP 指令的不规则性使得它不可能像编译器的前端那样有完整和标准的文法规则。

4)表驱动的方法产生的语义动作表往往是非常庞大的，需要采用专门的表压缩技术处理。而且它产生的代码也可能不是最优的。

基于以上分析，不难发现，虽然表驱动的代码生成方法也有许多优点和长处，但要把这种方法用于 DSP 处理器的代码生成却不是明智的选择。所以在这篇论文中将主要借鉴模式匹配的代码产生技术的设计思想。

§3.2 传统的编译器对 SIMD 指令的支持

由于SIMD指令在媒体处理器中的特殊地位和作用（不但可以提高媒体处理器的性能而且还可以降低其功耗），使得对SIMD指令的编译支持成为很多编译器设计人员无法回避、必须认真加以处理的一个棘手的问题。但由于当时在这一领域尚没有比较成熟和科学的处理方法供采用，所以很多编译器采用了简化的处理办法。虽然这些方法在一定程度上提供了对SIMD指令的支持，但它们的缺点是明显的和致命的，下面就将两种常用的简化处理方法[13][15]介绍如下:

3.2.1 建立内联函数（intrinsics）的方法

内联函数是直接映射为内联的处理器指令的特殊函数，即先要建立内联函数的数据库，使得编译器“知道”这些函数。然后使用时就像调用其它函数一样来调用它们，当程序员调用这些函数时，编译器就对它们进行特殊处理，在代码生成时直接将它们替换为相应的汇编指令。

3.2.2 优化汇编库的方法

这种方法是使用手工优化的汇编库。明显的这两种方法都存在着严重的弊端，它们不但增加了程序员的负担，而且使得开发的应用程序中存在很多与机器相关的代码，使得移植一个应用程序到其它的目标处理器上时需要做大量的修改工作，不满足程序的可移植的要求。

3.2.3 IMPACT 代码注释的方法

IMPACT编译器中的代码注释实际上是要用目标机器的操作复写Lcode代码的语义，注释得到的代码仍用Lcode表示，但因为每个操作都是目标机器支持的，因此也称它为Mcode。

在IMPACT编译器中，它的注释过程是逐操作进行的。最简单的情形是一个Lcode操作目标机器本身就支持，这时只需重写这个操作即可。稍复杂些的情形，Lcode操作目标机器是支持的，但它的操作数格式不合要求，因此需用其它操作来调整操作数的格式。最复杂的情况是Lcode操作在目标机器中没有相应的操作，需要用其它操作组合来实现。但对于需要同时考虑几条Lcode操作，并希望把这几条操作合并为一条复杂的目标机器指令的这种情形，这种方法明显的无能为力了。但实际上，在为DSP处理器进行代码选择的过程中，这种情况确实是经常遇到的。比如图3-4所示：

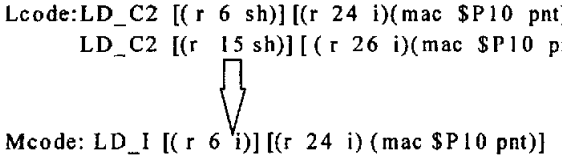

图 3-4 多对一的注释

从上面的介绍不难看出，这种处理方法的缺点是明显的，首先它实际上是一种硬编码的方式，注释过程与目标机器相关，根本不具备可移植性，使得当移植这个编译框架到其它目标机器上时需要重写这个过程。而更为重要的是对于这种注释方法，它每次只针对单条的语句进行分析，而不能结合同一基本块中的其它相关语句一起分析，这就使得这种方法并不能提供对目标机器指令集中的复杂指令和SIMD指令的支持。总的来说，由于原有注释方法搜索空间的限制，使得原有的逐条注释的方法并不能实现指令的多对一等价注释，不能充分挖掘机器指令集的硬件特性。

3.2.4 传统的模式匹配算法不支持 SIMD 指令的原因

通过以上分析，不难发现，传统的代码选择技术并不能很好的支持 SIMD 指令的产生，它们往往把产生 SIMD 指令的部分从通用的代码产生算法中独立出来加以特殊和简化处理，而产生 SIMD 指令的部分又往往是与机器相关的。那么传统的模式匹配算法为什么不能支持 SIMD 指令呢？究其原因，主要有以下的两个方面：

第一，对于一般的通用的机器指令，在进行机器描述时，往往只用一条模板来描述就可以了。然而对于 SIMD 指令却不是这样，它往往要求采用两条或多条

模板才能描述清楚。比如对于 ADD2,在我们的机器描述中用图 3-5 所示的两条模板来描述：

```
reg_lo: Lop_ADD(reg_lo,reg_lo)
reg_hi: Lop_ADD(reg_hi,reg_hi)
```

图 3-5 ADD2 的机器描述

其中reg_lo和reg_hi分别表示一个32位寄存器的低16位和高16位。同时传统的模式匹配的代码产生技术基于的是一种数据流树(DFTs)的中间表示，把模式匹配的问题转化为用可用的指令模板匹配数据流树(DFTs)的问题,而这时很可能描述SIMD指令的多个模板分别匹配了不同的DFT中的节点(这是不正确的，因为这样将产生无效代码),即使他们匹配了同一棵树的多个节点，传统的模式匹配技术也没有能力把描述同一个SIMD指令的多个模板整合起来而生成SIMD指令。SIMD指令的产生要求基于基本块的数据流图(DFG)中间表示而不是DFT，以便从全局的角度考虑模板的整合问题[19]。

第二，传统的模式匹配的代码产生技术往往对输入的每个DFT产生唯一的最优覆盖，而SIMD指令的产生要求在覆盖每个DFT时产生多个可选的最优覆盖，然后再从全局的角度选择一个最优的覆盖[20][21]，使得这个覆盖要包含尽可能多的SIMD指令。

§3.3 支持 SIMD 指令的代码选择技术实现框架

基于传统代码选择技术的不足，我们提出要在基本块的完全数据流图DFG（full data flow graph）的基础上进行代码选择[2]，而不再是基于数据流树DFT(data flow tree)的中间表示。同时，由于希望借鉴模式匹配的代码生成思想，所以在这里采用把DFG分解为多个DFT的方法，以便使用模式匹配的方法完成对每个DFT的标注。注意在这里我们对传统的模式匹配与动态规划算法进行了改进，同时对每个DFT产生多个可选的最优覆盖。最后再使用整数线性规划的方法，从全局的角度在这多个可选的最优覆盖选择唯一的一个最优覆盖，使得这个覆盖中尽可能多的包含SIMD指令。其基本实现框图如图3-6所示：

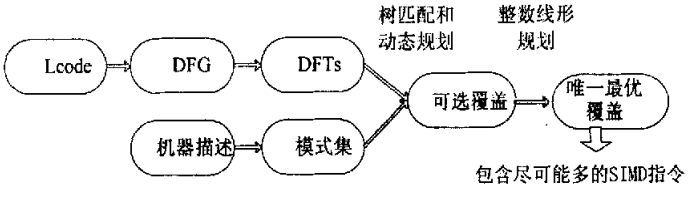

图 3-6 支持 SIMD 的代码选择的实现框图

在这里我们借鉴了传统的树匹配和动态规划算法的思想，同时又对它进行了改进，主要表现在如下的两个方面：其一,传统的树匹配和动态规划算法基于的是

数据流树的中间表示，但考虑到SIMD的产生需要从整个基本块的角度来寻求最优覆盖，所以在这里我们采用基本块的数据流图作为中间表示。其二，在覆盖每个数据流树（DFT）时（为了采用模式匹配的思想，先要把DFG再分解为一系列的DFTs，如上图示），不再是生成唯一的最有覆盖，而是在每个节点处标注多个可选的最优覆盖。当这一系列的DFTs匹配完毕以后，就意味着整个DFG已经匹配完毕。在此基础上对代码选择所需满足的约束条件进行建模，生成目标方程，使用整数线性规划的方法解出目标方程的最大解，这个最大解就意味着能最大限度的产生SIMD指令的数目。最后根据这个计算结果作相应的语义动作，最终生成包含尽可能多的SIMD指令的Mcode代码。

§3.4 小结

本章分析了传统代码选择技术在支持SIMD指令时所面临的困难和不足，在此基础上提出了图3-6所示的支持SIMD指令的代码选择实现框图。为了便于在后面的几章中具体的介绍该过程，在此我们把这个代码选择的过程分成如下的三个子过程。第一个子过程：DFT和指令模板生成过程。它包括的具体内容有：设计基本块的DFG表示，把Lcode中间表示转化为DFG表示，把DFG中间表示分解为一系列DFTs；描述目标机器的指令，对指令描述进行预处理，生成相应的内部表示。不难发现该子过程实际包含了两个互不相关的部分：一部分与编译器相关而与目标机器无关。相反地，另一部分则与目标机器相关而与编译器无关。这正是编译器在软件开发中的角色--个缩影，即把与机器无关的高级语言映射为目标机器的汇编语言。第二个过程：模板覆盖过程，有了第一个过程的准备，在这个过程中，用指令模板去覆盖每个DFT，在这里采用了改进的树匹配和动态规划算法，为了产生SIMD指令，该过程将为每个DFT标注多个可选的最优覆盖。第三个过程，覆盖选择过程，这个过程将从第二个过程所标注的多个可选最优覆盖中，选择一个能尽可能多的产生SIMD指令的覆盖，我们把这个覆盖称为最终的最优覆盖。在这里，“最优”包含两方面的含义：一方面和第二个过程中的最优意义相同，指通过动态规划所计算出来的总开销为最小，另一方面是指该覆盖能最大限度的产生SIMD指令。

第四章 指令模板和 DFT 生成

本章主要叙述了两个方面的内容，一是：关于描述目标机器指令集的问题[4][5]，二是：关于如何设计 DFG 中间表示以及如何把 Lcode 表示转化为 DFG 表示的问题。这是两个并行的问题，一个是与机器无关的中间代码表示的问题，一个是如何描述目标机器的问题。它们相当于树匹配和动态规划算法的两个输入，通过树匹配和动态规划算法[1]把机器相关的部分和机器无关的部分结合起来。其中 4.1 节着重介绍前一个问题，包括描述目标机器指令集所采用的树文法和文法规范[8]，以及描述 YHFT-DSP/700 指令集的情况。4.2 节则着重介绍后一个问题，包括如何设计 DFG 中间表示以及把 Lcode 转化为 DFG 表示。

§4.1 目标机器指令集描述

在本论文所介绍的代码选择技术中，我们用树文法来描述目标机器指令集，下面将给出这种文法的简单概括和具体的文法规范[8]。

4.1.1 树文法

树文法形式化的表示为：$G=(\sum_N,\sum_T,S,R,c)$，其中 $\sum_T$ 是终结符号的集合，$\sum_N$ 是非终结符号的集合，S 是文法的开始符号， R 是规则的集合，c 是规则对应的开销。$\sum_T$ 中的符号用来表示 DFG 中的每个节点（变量，常量，操作符）。$\sum_N$ 主要用来表示存储数据的硬件资源（寄存器，存储器），另外，也用来表示指令模板中的公共部分。而指令模板本身用 R 中的规则来建模，每一个规则描述了一条指令的行为。例如，规则“reg：PLUS(reg,reg)”描述 PLUS 指令行为：把两个寄存器的内容相加，结果写到另一个寄存器中。其中，PLUS 是一个终结符，表示加法操作符。reg 是一个非终结符，表示一个通用寄存器。又比如，对于 SIMD 指令“ADD2”，由于它同时执行两个 16 位加法，所以可用两条分开的规则来模拟“ADD2”的动作：

reg_lo: Lop_ADD(reg_lo,reg_lo)
reg_hi: Lop_ADD(reg_hi,reg_hi)

在这里 reg_hi 和 reg_lo 分别表示一个完整寄存器的低和高 16 位子寄存器。这两个规则都被赋予了和 32 位的加法同样的开销值。应该注意的是，所有表示在子寄存器上进行操作的规则都不能被看成是独立的模板，否则可能会产生无效的

代码。而这些规则只能用来覆盖 DFG 节点对。DFG 节点对的概念将在后面进行介绍。

4.1.2 文法规范

如图 4-1 所示，为一个典型的扩展的 BNF 文法规范：

```
grammer—>   { dcl } %% { rule}
   dcl   -   > % start nonterm
               | % term( identifier= integer}
  rule   -   >  nonterm: tree = integer[cost];
  cost   -   > ( integer)
  tree   -   > term( tree , tree)
               | term( tree)
               | term
               | nonterm
```

图 4-1 扩展的 BNF 文法规范

文法规范由声明部分，%%分隔符以及规则部分组成，其中声明部分声明了终结符，即数据流树(DFT)中操作符，每一个操作符和一个唯一的正的外部符号编号相联系。非终结符通过它在一条规则的左边的出现来声明。在图 4-1 中，term 和 nonterm 分别表示终结符和非终结符。规则部分定义了树模板，这些模板以带括号的前缀形式表示。每个非终结符标识一棵树，每个操作符都有固定的元数，这可以从描述它的规则中推断出来。如果一条规则的模板是另外一个非终结符，那么称这条规则为链规则。如果没有显式的声明开始符号，那么第一条规则定义的非终结符将被看作是开始符号。每一条规则有一个唯一的，正的外部规则编号，它跟在模板后面和模板之间以等号连接。外部规则编号被用来向用户提供的语义动作例程传递匹配规则的信息。在规则的最后是一个可选的非负的整数开销，它被用来作为动态规划的依据。

4.1.3 YHFT-DSP/700 指令集的描述

4.1.3.1 描述策略

这个描述由两种产生式组成，一种是没有汇编语句和它对应的产生式，一种是有汇编语句和它对应的产生式。第一种产生式的目的是为了简化模式描述写法，从而使后面的产生式的个数尽量少，看起来更简单，也更容易检查正确性[24]。比如对于操作数的各种类型，可以用非终结符 regmac 统称通用寄存器和宏寄存器，而用非终结符 reg 来统称必须先放到寄存器中操作数类型，例如标号类型、字符串类型、立即数类型等。如图 4-2 所示：

```
regmac: YHFT_REG=11;
regmac: YHFT_MAC=12;
```

图 4-2 regmac 的产生式

同时注意到，对于这样的模板，其开销值常被置为 0，因为它们不产生汇编指令。第二类的产生式有汇编指令与其对应，例如图 4-4 所示的模板。

4.1.3.2 操作数类型的准确描述

为了准确的描述 YHFT-DSP/700 的指令集，我们就必须充分的了解其指令格式的多样性和灵活性。比如 SUB 指令的在.L 功能单元上的指令格式如图 4-3 所示：

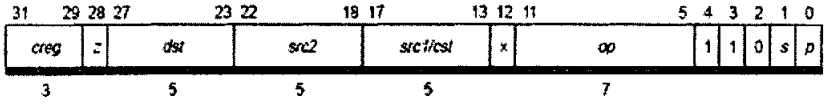

图 4-3 SUB 指令在.L 单元上的指令格式

其中对操作数 src1,它的类型可能是寄存器，宏寄存器，立即数等。但从其指令格式可以看出，如果 Src1 的操作数类型是立即数时，应该分两种情况来描述：如果 Src1 为五位有符号或无符号立即数时，我们就可以直接把它编码到指令格式中去，但如果这个立即数是 16 位或 32 位时，则必须先把它们放到一个寄存器中，然后再把这个寄存器编码到指令格式中。区分这两种情况是至关重要的，因为这不但能有效的减少指令的条数，而且还可以减少寄存器的压力。因此对于 SUB 指令，就用如下图 4-4 所示的几条模板来描述：

```
reg:  Lop_SUB(reg,regmac)=302          (1);
reg:  Lop_SUB(regmac,reg)=303          (1);
reg:  Lop_SUB_U(reg,regmac)=304        (1);
reg:  Lop_SUB_U(regmac,reg)=305        (1);
reg:  Lop_SUB(regmac,regmac)=306       (1);
reg:  Lop_SUB_U(regmac,regmac)=307     (1);
reg:  Lop_SUB(YHFT_IMMED5S,regmac)=308     (1);
reg:  Lop_SUB(regmac,YHFT_IMMED5S)=309     (1);
reg:  Lop_SUB(regmac,YHFT_IMMED5U)=310     (1);
reg:  Lop_SUB_U(YHFT_IMMED5S,regmac)=311   (1);
reg:  Lop_SUB_U(regmac,YHFT_IMMED5S)=312   (1);
reg:  Lop_SUB_U(regmac,YHFT_IMMED5U)=313   (1);
reg:  Lop_SUB(YHFT_IMMED5S,reg)=510        (1);
reg:  Lop_SUB_U(YHFT_IMMED5S,reg)=511      (1);
reg:  Lop_SUB(reg,YHFT_IMMED5S)=512        (1);
reg:  Lop_SUB_U(reg,YHFT_IMMED5S)=514      (1);
reg:  Lop_SUB(reg,YHFT_IMMED5U)=515        (1);
reg:  Lop_SUB_U(reg,YHFT_IMMED5U)=516      (1);
reg:  Lop_SUB(reg,reg)=517             (1);
reg:  Lop_SUB_U(reg,reg)=518           (1);
```

图 4-4 描述 SUB 指令的模板

在这里，终结符 YHFT_IMMED5S 和 YHFT_IMMED5U 分别表示五位有符号

立即数和五位无符号立即数。

§4.2 中间代码转换

正如在第三章介绍的那样，传统的基于数据流树(DFTs)的模式匹配的代码产生技术由于搜索空间的限制，不能有效的产生 SIMD 指令。针对这个原因，在我们改进的匹配技术中提出了要在完全数据流图（DFG）的基础上进行代码选择，那么在这一部分我们就要着重讨论如何设计一种既能充分表达 Lcode 的语义，又适合于模式匹配与动态规划算法的 DFG 表示，如何把基本块构造为一个 DFG，以及如何把 DFG 再分解为一系列 DFTs 等问题。

4.2.1 基本块的 DFG 表示

在这里设计的 DFG 表示中，每个节点表示一个操作(算术、逻辑、load、store)，一个常量或者一个变量，每一条边(n ,m)表示在节点 n 和节点 m 之间的一个数据依赖关系。在 IMPACT 编译器中，他们主要是 L_code.h 中定义的关于 Lcode 的各种操作以及为了表达 Lcode 操作数的多样性，我们自行定义了另外的几个常量，如图 4-5 所示：

```
YHFT_VOID=500 YHFT_CB=501  YHFT_CONSTI=502 YHFT_STR=503
YHFT_LAB=504  YHFT_MAC=505 YHFT_REG=506   YHFT_IMMED4U=603
YHFT_IMMED5U=600 YHFT_IMMED5S=601 YHFT_IMMED16S=602
```

图 4-5 表示 Lcode 数据类型的常量

在这里每一个常量后面的数字，就是该常量的内部符号编号，在前一节中论述了这个问题，同时可以看到，在设计基本块的 DFG 表示时，我们也充分的考虑了 DSP 指令集的特征，把立即数区分为了 32 位常数、16 位常数、5 位有符号立即数、5 位无符号立即数以及四位无符号立即数等不同情况，以便能够充分的利用指令的编码格式，并且减轻寄存器的压力。同时在这种 DFG 表示中，每一条边(n ,m)表示在节点 n 和节点 m 之间的一个数据依赖关系。注意，一般来说，一个 DFG 图可以包含公共子表达式(CSEs)并且可以有几个互不相连的子图构成。DFG 图具有如下的特征：

1.图的叶节点，即无子节点的节点，以图 4-5 所示的各个常量作为标识，表示各个互不相同的操作数类型。

2.图的内部节点，即有子节点的节点，以一 Lcode 的运算符作为标识，表示这个节点代表应用该运算符对其后继节点所代表的值进行运算的结果。

3.图中各个节点上可能附加一个或多个虚拟寄存器名，表示这些虚拟寄存器都保存了该节点所代表的值。在我们的工程实现中，用了如下的数据结构来描述

DFG 节点：

```
typedef struct Tree{
int op;                 /*操作类型*/
int node_id;            /*按节点生成顺序给节点编号*/
struct Tree *kids[2]; /*左右子节点*/
L_Oper *oper1;     /*用于记录生成该节点的操作*/
L_Operand *val;         /*only useful to leaf node */
int count;              /*记录节点的访问次数*/
struct {STATE_TYPE state;} x;  /*主要用于表示匹配算法*/
struct tree* parent_left[SIZE];
struct tree* parent_right[SIZE];
Dag_Label*  label;
}  tree;
```

在这里 op 字段为关键字段，指明了操作的类型或操作数类型，node_id 按照节点的生成顺序给每个节点一编号，*kids[2]字段分别指向该节点的左右子节点；*oper1 字段记录了生成该节点的 Lcode 操作；*val 字段只用于叶节点，记录立即数的具体数值，以便具体区分该节点为何种立即数(比如是五位有符号数还是 16 位有符号数)；count 字段记录了该节点被访问的次数，用于区分该节点为公共子节点还是根节点；x 域则记录了该节点被标注后的结果，这是为树匹配和动态规划算法而设计的；parent_left 和 parent_right 字段都记录了该节点的父节点，但对于 parent_left 数组中的每个节点，该节点为它们的左孩子，而对于 parent_right 数组中的每个节点，该节点为它们的右孩子，这个域被用于 DFG 图的分解；最后的一个域为 label 域，它记录了在一个节点上所标注的多个标号。

4.2.2 特殊指令的 DFG 表示

4.2.2.1 MOV 指令

按照前面给出的设计原则，对于如下的一条 MOV 指令: MOV [(r 1 i)][(i 0)]，本应该生成如图 4-6 a)所示的 DFG 图，但实际上我们却采用了图 4-6 b)所示的表示形式：

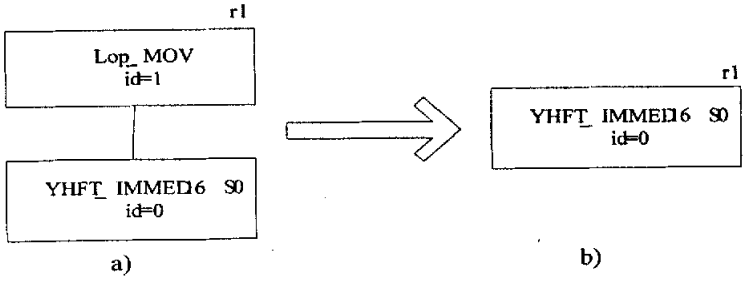

图 4-6 mov 指令的 DFG 图表示

这样做的目的有以下的两个方面。第一，明显的 b）的表示形式要比 a）简单，减少了构造 DFG 的工作量。第二，在机器描述中可以采用这样的简单模板：reg : YHFT_IMMED16S ;来表达需要把一个 16 位的立即数放到一个寄存器这样的语义，所以当这个模板经树匹配和动态规划算法标注在 b)所示的节点上以后，我们在相应的语义动作中就可以直接生成一条 mcode 的 mov 指令，把立即数 0 送到一个寄存器中。从而不难看出这样的简化和特殊处理是合理和高效的。

4.2.2.2 STORE 指令

在 Lcode 中，以 ST_C 为例，其语法形式如下：

```
Lop_ST_C src1, src2, src3
```

明显的它有三个源操作数，而在我们的 DFG 图中，每个节点最多只有两个子节点，因此必须对这样的一类操作进行特殊的处理，在这里我们采用了这样的处理办法：在构造 DFG 图时，引入一个新的操作符 Lop_ADDR,它表示 store 指令的前两个操作数计算后所得到的地址。比如对于这样的一条 store 指令，对其构造的 DFG 图如图 4-7 所示：

```
Lcode:  st_c2  [ ][(r 24 i)(mac $P8 pnt)(r 11 i)]
```

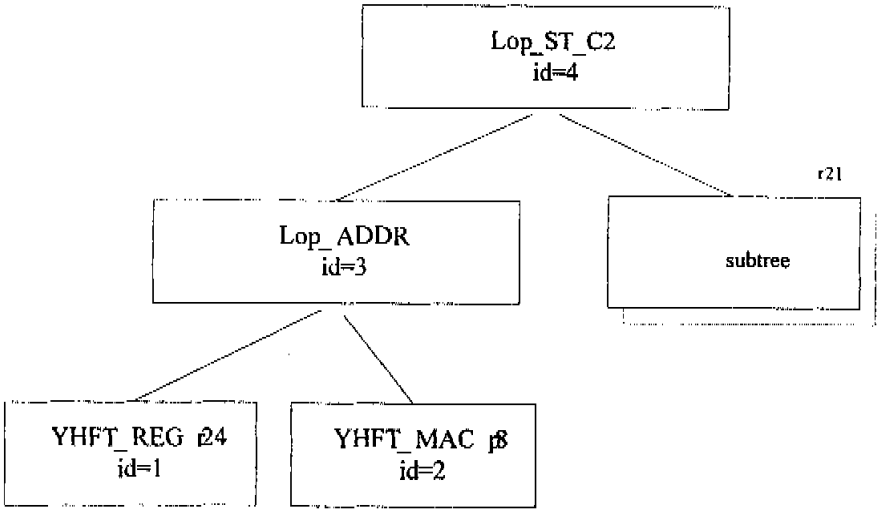

图 4-7 store 指令的 DFG 图表示

相应地，在对 store 指令的 DFG 表示进行特殊处理以后，在对 store 指令进行模板描述时也采用了如图 4-8 所示的形式：

```
addr:  ADDRLP(reg,regmac)=520;
addr:  ADDRLP(regmac,regmac)=521;
addr:  ADDRLP(regmac,reg)=522;
addr:  ADDRLP(reg,reg)=525;

stm:   Lop_ST_C(addr,regmac)=314          (1);
stm:   Lop_ST_C(addr,reg)=315         (1);
```

图 4-8 store 指令的模板描述

4.2.3 由基本块构造 DFG 的过程

4.2.3.1构造算法

/*这个过程遍历输入基本块中的每一条语句，根据其操作类型转入相应的处理程序 L_dag_xxx(* L_Oper)*/

```
void creat_Dag(L_Cb *p)
{
        initial opcode HASH table;
        initial operand HASH table;
        for (all the oper in the Cb)
        L_dag_xxx(oper);
 }
```

/*这个过程输入一条 Lcode 语句，根据这条语句构造相应的 DFG 图*/

```
void L_dag_opcode(L_oper *oper)
{
    Operand_Tbl *t1,*t2,*t3;
    Tree p1,p2,p3;
    Op_Tbl  *t4;
   /*处理源操作数*/
   if (src[i] exist)
    {
      判断src[i]的类型;
      if((ti=find_operand(oper->src[i]))==NULL) {
          pi=create_operand_node(op,oper->src[i],oper);
          add_operand(oper->src[i],pi);
          insert_dag_list(pi);
      }
      else
           pi=ti->data;
    }
   /*处理操作符*/
   if ((t4=find_op(Lop_opcode,p1,p2))==NULL){
     p3=creat_op_node(opcode,p1,p2,oper);
```

```
        add_op(op, p1, p2, NULL, p3);
        insert_dag_list(p3);
      }
     else
        p3=t4->data;
     /*处理目的操作数*/
     if((t3=find_operand(oper->dest[0]))==NULL)
      {
        add_operand(oper->dest[0], p3);
        add_label(p3 , oper, oper->dest[0]);
      }
    else
      {
        del_label(t3->data, oper->dest[0]);
        t3->data=p3;
        add_label(p3, oper, oper->dest[0]);
      }
}
```

这个过程对于每条输入的Lcode语句，构造相应的DFG图。处理过程为：1)对于源操作数src[0]和src[1](如果存在的话),首先在操作数HASH表中查找，看是否已经为该操作数构造了相应的节点，如果已经构造了，就记录下这个节点，如果没有就为该操作数构造相应的节点，并把这个节点加入到操作数HASH表中。2)对于操作符，先处理它的操作数，如1)所示。然后结合它的操作数节点，去查找操作HASH表，这个过程由函数find_op(Lop_code,p1,p2)实现，如果这个操作符为一元操作符，那么p2为NULL。如果存在这样一个节点，它的op域为Lop_code且它的左右子树分别为p1和p2，那么就记录下这个节点，如果不存在，就为该操作符构造相应的节点并把它加入到操作HASH表中。3)对于目的操作数，同样的要首先在操作数HASH表中查找，看操作数HASH表中是否已经存在该操作数的HASH表项，如果存在，就说明该操作数为某个节点的标号，那么就从那个节点上删除这个标号同时修改该HASH表项，并把这个操作数作为标号加到操作符节点上。在这个过程中，有两个重要的数据结构分别为操作数HASH表和操作符HASH表：

```
typedef struct Operand_Tbl
 {
   L_Operand   *p;                          /*关键字*/
   Tree    data;                    /*引用一个 DAG node 指针指向数据节点*/
   struct Operand_Tbl *next_hash;   /*HASH 表项 next 指针*/
   struct Operand_Tbl *prev_hash;
 } Operand_Tbl;
Operand_Tbl    *Hash_1[N];          /*定义一个固定大小的 HASH 表指针数*/
```

```
Typedef struct Op_Tbl
{
    int     op_val;     /*操作符的内部编码*/
    Tree    leftPtr;    /*左子树节点*/
    Tree    rightPtr;   /*右子树节点*/
    Tree    data;       /*本节点  */
    struct  Op_Tbl  *next_hash; /*前一个 hash 表项*/
    struct  Op_Tbl  *prev_hash; /*后一个 hash 表项*/
    struct  Op_Tbl   *tail;
} Op_Tbl;
```

从上面的分析不难发现，在这个算法实现中，有一个非常重要的过程—查找（不但包括操作数节点的查找还包括操作符节点的查找），因此认真设计和实现这个过程是至关重要的，下一部分就对这个过程进行较为详细的介绍。

4.2.3.2 查找过程的设计与实现

查找过程为构造 DFG 图的一个非常重要且使用频率很高的一个子过程，它被用来确定公共子表达式以及每个公共子表达式被引用的次数。为了使得查找过程尽可能的高效，我们采用了 HASH 表的实现方法，同时把对操作数的查找和对操作符的查找区分开来。这两个 HASH 表的数据结构如上一部分所示。图 4-9 描述了具体的查找过程：

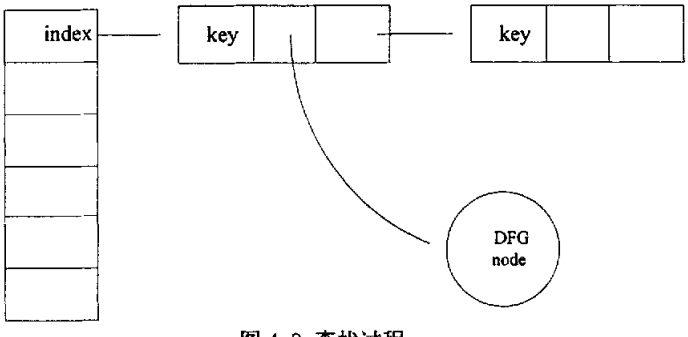

图 4-9 查找过程

在上图中，index 为索引值，不同类型的操作数，它的索引值的计算方法不同；而操作数和操作符索引值的计算方法更不相同。比如，对于寄存器或宏寄存器操作数，我们以它们的内部编号作为索引，而对于立即数，则以它的值作为索引。特别的对于操作符，其计算方法比较复杂，在这里我们用操作码的内部编号与表示它的操作数的节点的地址的和作为索引。Key 为关键字字段，它唯一的确定了一个 hash 表项，对于操作数表项，我们就用 L_Operand *p 作为它的关键字段，用函数 L_same_operand(L_Operand *,L_Operand *)进行比较；而对于操作符表项，则采用 int op_val、 Tree left_ptr、 Tree right_ptr 三个域作为关键字段，因为只有这三项才能唯一的确定一个操作符表项。

4.2.4 把 DFG 分解为 DFT 的过程

在构造基本块的 DFG 表示过程中，不仅把各个节点按 DFG 图的形式组织而且还把它们链成一个链表。而后，我们采用两遍遍历该链表的方法来分解该 DFG 图。第一次遍历主要是识别根节点，即那些不为任何节点的子节点的节点，并把它们链成一个根节点链表。第二遍遍历主要是识别公共子表达式，即以那些为多个节点的子节点的节点为根节点的子树，在识别了一个公共子表达式以后，把该子树的根节点加入到上面的根节点链表中，同时生成相应的节点来替代原来的子树。经过这两遍扫描以后，就得到了一个 DFTs 的链表。

第五章 模板匹配

本章主要叙述树匹配和动态规划算法的核心思想，以及为支持 SIMD 指令和便于实现所进行的改进。5.1 节继第四章的机器指令集描述和 DFG 图构造之后，提出了后续处理的主要任务。5.2 节叙述了树匹配和动态规划算法[1]的核心思想。5.3 节给出了为支持 SIMD 指令和便于实现而进行的算法[2][3][4]改进。5.4 节则为本章小结。

§5.1 问题描述

以如下的一段简单的 C 语言程序为例：

```
void main(short* A, short* B, short* C)
{
   int i;
   for(i=0;i<10;i+=2)
   {
     A[i]=B[i]+C[i];
     A[i+1]=B[i+1]+C[i+1];
   }
}
```

这段 C 语言程序经过 IMPACT 编译器前端处理以后，转换成其中间表示 Lcode，这段 Lcode 代码的核心部分如下所示：

```
(cb 3 0.000000 [(flow 1 3 0.000000)(flow 0 5 0.000000)] <(trace (i 2))>)
(op 13 ld_c2 [(r 6 sh)] [(r 24 i)(mac $P10 pnt)] <(lab_P_B_2_27___1)(param (i 1))>)
(op 15 ld_c2 [(r 9 sh)] [(r 24 i)(mac $P12 pnt)] <(lab_P_C_2_36___1)(param (i 2))>)
(op 16 add [(r 11 i)] [(r 6 sh)(r 9 sh)])
(op 18 st_c2 [] [(r 24 i)(mac $P8 pnt)(r 11 i)] <(lab_P_A_2_18___1)(param (i 0))>)
(op 21 ld_c2 [(r 15 sh)] [(r 26 i)(mac $P10 pnt)] <(lab_P_B_2_27___1)(param (i 1))>)
(op 24 ld_c2 [(r 19 sh)] [(r 26 i)(mac $P12 pnt)] <(lab_P_C_2_36___1)(param (i 2))>)
(op 25 add [(r 21 i)] [(r 15 sh)(r 19 sh)])
(op 28 st_c2 [] [(r 26 i)(mac $P8 pnt)(r 21 i)] <(lab_P_A_2_18___1)(param (i 0))>)
(op 29 add [(r 1 i)] [(r 1 i)(i 2)])
(op 38 add [(r 26 i)] [(r 26 i)(i 4)])
(op 34 add [(r 24 i)] [(r 24 i)(i 4)])
(op 30 blt [] [(r 1 i)(i 10)(cb 3)] <(loop_nest (i 1))(LB_inner)(LE_inner)>)
```

正如在第三章的图3-6所示的那样，我们要把这段Lcode代码转化为一种DFG图表示，并把它分解为一系列 DFTs。关于构造什么样的 DFG 图、怎样构造 DFG 图以及如何把一个 DFG 图分解为一系列 DFTs 的问题已经在第四章中介绍了。上面的这段代码经过这些处理后生成了 5 棵数据流树，在这里主要考虑如图 5-1 和图 5-2 所示的两棵树：

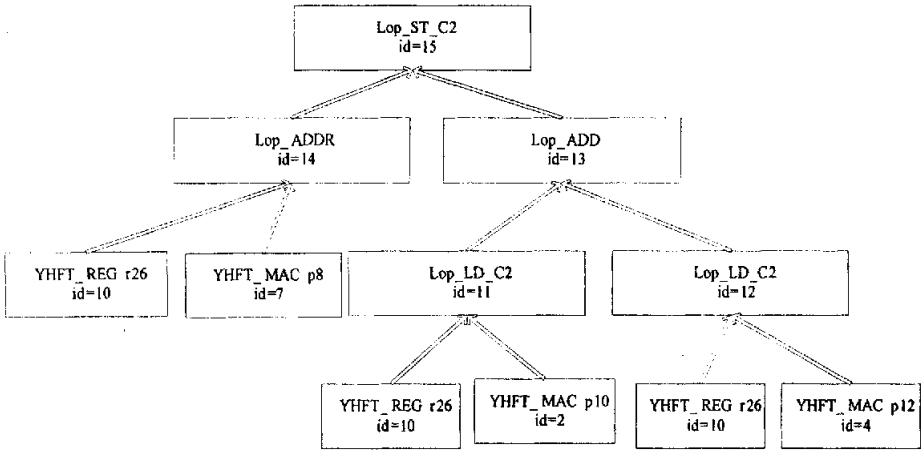

图 5-1 数据流树(一)

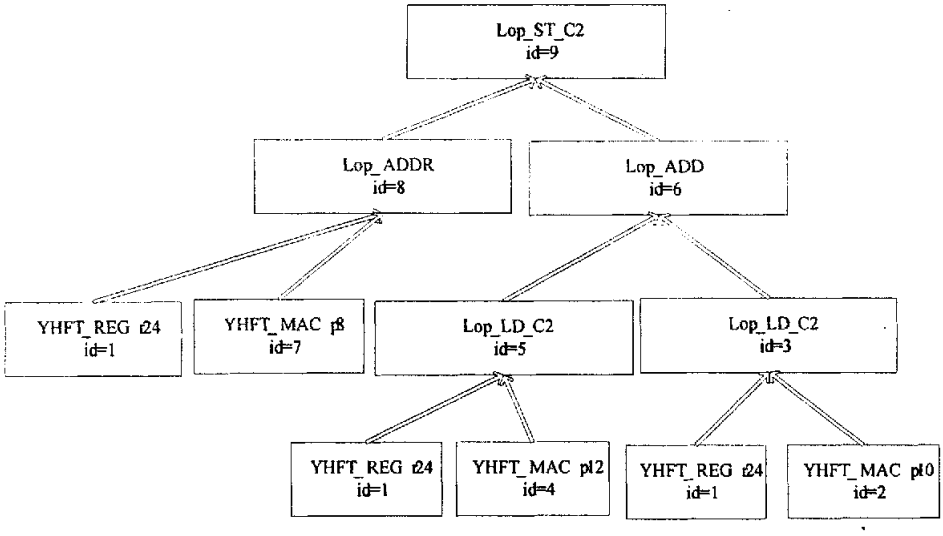

图 5-2 数据流树（二）

在 YHFT-DSP/700 的机器指令集描述中。采用了如下的指令模板::

模板号	标号	模板	开销
11	regmac	: YHFT_REG	0
12	regmac	: YHFT_MAC	0
192	reg	: Lop_LD_C2(regmac,regmac)	1
193	reg_hi	: Lop_LD_C2(regmac,regmac)	1

194	reg_lo	: Lop_LD_C2(regmac,regmac)	1
320	stm	: Lop_ST_C2(addr , reg)	1
321	stm	: Lop_ST_C2(addr , reg_hi)	1
322	stm	: Lop_ST_C2(addr , reg_lo)	1
361	reg	: Lop_ADD(reg,reg)	1
362	reg	: Lop_ADD(reg_lo,reg_lo)	1
363	reg	: Lop_ADD(reg_hi,reg_hi)	1
521	addr	: ADDRLP(regmac,regmac)	1

将这些模板应用到数据流树(DFTs)上，生成如图 5-3 和 5-4 所示的覆盖：

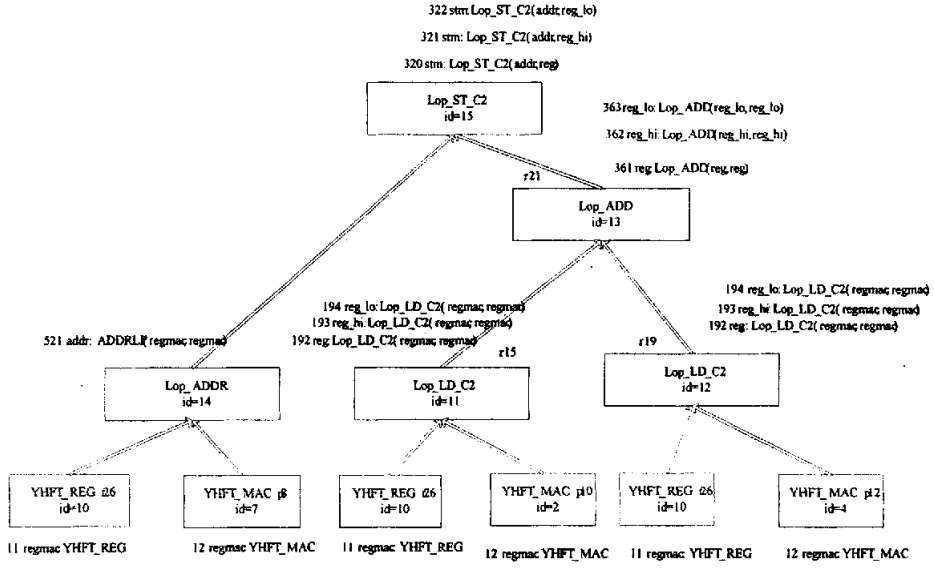

图 5-3 模板匹配后的数据流树(一)

从上面的例子可以容易的看出代码选择的任务主要有以下的几个方面：

第一：构造 Lcode 的 DFG 表示，并把它分解为一系列 DFTs，已完毕。

第二：给出目标机器指令一个描述，使得机器相关的部分都集中在这个描述中，已完成。

第三：给出机器描述的一个预处理程序，使得这个机器描述转化为相应的内部表示。

第四：要有一种模式匹配的过程来匹配中间语言和指令模式，从而生成整个 DFG 图的多个可选的最优覆盖。

第五：要有一种机制，使得最终生成的覆盖从整个 DFG 图的角度看来是最优的，即要由一种动态规划的机制来计算每个覆盖的开销大小。

第六：要有一种机制，能够从 DFG 图的多个可选覆盖中确定唯一的最优覆盖，这个覆盖以最大限度的生成 SIMD 指令为目标。

第七：根据整个 DFG 图的最优覆盖执行相应的语义动作，生成包含尽可能多的 SIMD 指令的 Mcode 代码。

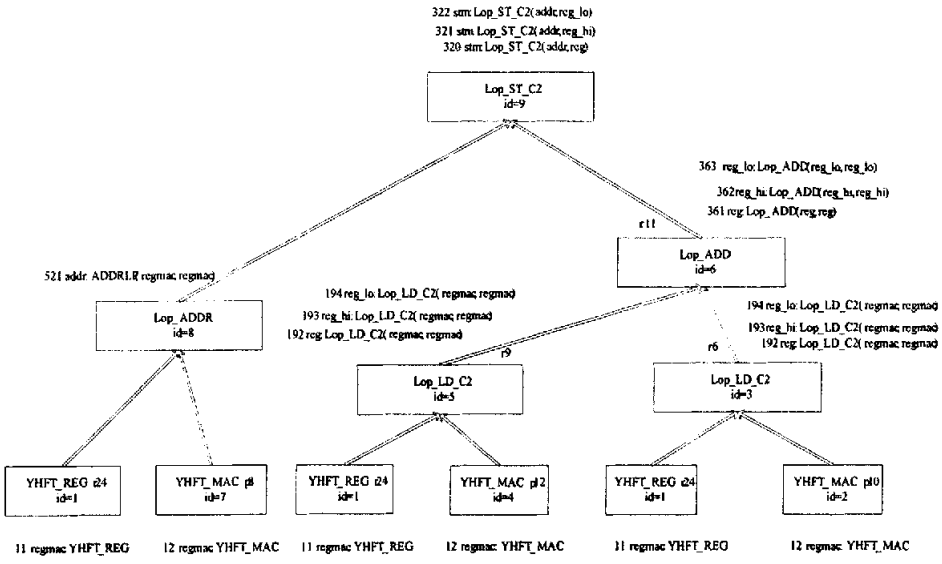

图 5-4 模板匹配后的数据流树（二）

在本章中，将一种快速的树模式匹配算法和一种高效的动态规划算法结合起来来产生最优的覆盖。同时对这种算法进行改进，使得每一个数据流树（DFT）在经过匹配以后，不是生成唯一的最优覆盖，而是像图 5-3、图 5-4 所示的那样，生成多个可选的覆盖。而后，再通过整数线性规划的方法，从这多个可选覆盖中选出一个唯一的最优覆盖，使得这个覆盖包含尽可能多的 SIMD 指令。其中树模式匹配算法属于上面所说的任务 4，而动态规划和整数线性规划则属于上面所说的任务 5 和任务 6。

§5.2 树匹配和动态规划算法

在不同的文献中已经看到了多种不同的树模式匹配算法。但对于像代码产生这样的应用，它们都有一定的缺陷。为此，我们提出一种新的方法，即把 Aho-Corasick 的多关键字匹配算法扩展成为一种自顶向下的树模式匹配算法[1][7]。

5.2.1 多关键字匹配算法

问题描述：给定一个关键字集合 S，在一个输入字符串中查找所有的包含在集合 S 中子串。

算法思想：这个算法的本质是要从给定的关键字集合构造一棵树，把这棵树

转换为一个模式匹配自动机，然后利用这个自动机并行的在输入字符串中搜索关键字。

假设 K 为关键字集合。树的构造过程是这样的：首先要构造一个根节点，然后对于K中的任意一个关键字都产生一条从根节点出发到另外一个节点的通路使得从根节点到这个节点之间的边上的标号组成这个关键字。这样树中的每一个节点都唯一个的被一个符号串所标示，这个符号串由从根节点到这个节点之间的边上的标号组成。

模式匹配自动机从这棵树出发来构造。自动机的状态就是树的节点：开始状态是根节点，接受状态就是那些对应于完整的关键字的节点（即树的叶节点）。如果节点 S 和节点 T 之间有一条边并且这条边上的标号为 c，那么在模式匹配自动机中，状态 S 接受到输入字符 c 后就会转换到状态 T。另外，当向自动机输入的字符不是一个关键字的首字符时，那么它会从开始状态转换到它本身。

模式匹配自动机中每一个不是开始状态的状态，它都有一个“失效指向”，对于一个由字符串 u 刻画的状态，它的失效指向是一个指针，它指向一个字符串 v 刻画的状态，在这里 v 是关键字集合 K 中的最长前缀，同时也是字符串 u 的真后缀。

在这个算法中，构造树和模式匹配自动机的时间复杂度都和集合 K 中的所有关键字的长度成正比。而一旦构造了模式匹配自动机，那么对于任意一个输入串 x，它的运行时间和 x 的长度成正比，而与集合 K 的大小无关。从而，从输入串 x 中寻找所有包含在关键字集合K中的子串的这个问题的完整解决所需的时间复杂度为 O(| K| +| x |)。

5.2.2 自顶向下的模式匹配算法

上面介绍的算法可以直接推广为树匹配算法，这是因为一棵树可以用从它的根节点到它的叶节点的通路的集合来刻画，在这里对于从每一个节点出发的每一条边按照从左到右的顺序依次被编号为 1，2，3，等。例如，可以考虑图 5-5 所示的三个树型指令模板：

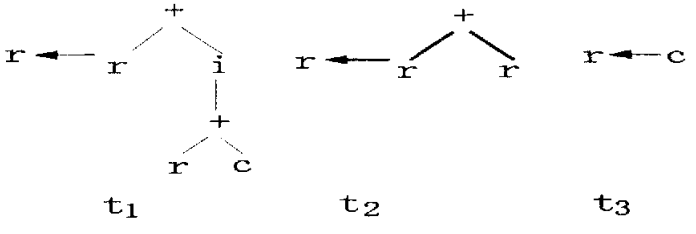

图 5-5 树形指令模板

它们分别被称之为 T1,T2,T3。这三个树型模版可以出如下的通路集合来刻画：

+1r

+2i1+1c

+2i1+2r

+2r

C

可以看到，第一个串+1r 表示了树型指令模板 T1 和 T2 的最左通路，而+2i+1+1c 和+2i1+2r 分别表示了树 T1 的右子树的两条通路。同时注意，我们用长度为 2j+1 的串来刻画树中深度为 j 的通路。从这个通路字符串出发，我们可以构造一个模式匹配的自动机来并行的匹配通路字符串。从上面的三棵树出发，我们可以构造如图 5-6 所示的模式匹配自动机。

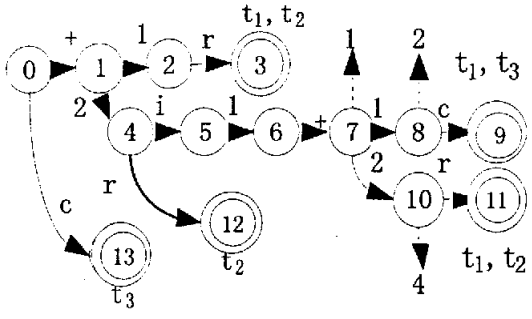

图 5-6 模式匹配自动机

在这个图中，状态 0 为开始状态，带双圈的状态为接受状态。在这个自动机中，状态 7 的“失效指向”指向状态 1，状态 8 指向状态 2，状态 10 指向状态 4，而对于其他状态都指向状态 0。所有不指向状态 0 的“失效指向”都用虚线表示。在每一个接受状态，还标注了是哪一个树型模板的哪一个通路串被识别了。例如在状态 9，被识别的串分别为模板 T1 右子树中一个通路和模板 T2 中一个通路。

设 T 为一个形如 li：ti 的指令模板集合。在这里称 li 为标号，称 ti 为模板规则。我们像上面那样构造一个类似的自动机来并行的识别输入的主题树中的模板规则。设 succ（δ，a）表示自动机中的状态 δ 接受字符 a 以后所到达的状态。这个自动机在输入树的每一个节点 n 处创造一个信息记录：n.parent 表示节点 n 的父节点，n.symbol 表示节点 n 上所标注的字符，n.state 表示自动机接受了 n.symbol 以后所到达的状态。下面我们来描述这个算法中的三个关键的子过程。

首先对于以节点 n 为根节点的输入树，我们要按深度优先的顺序来遍历这棵树同时给每个节点赋予一个状态。在这里用 label(n)来表示这个过程：

```
label(n)
{
  if n is root node
```

```
        n.state = succ(start state,n.symbol);
    else
        //n is the kth child of n.parent
        n.state = succ(succ(n.parent.state,k),n.symbol);
    for every child c of n do
        label(c);
    post_process(n)
}
```

由上面的过程遍历了输入树以后，如果有节点 n，其 n.state 为接受状态，那么就说明在节点 n 识别了一个通路字符串。想要寻找这个匹配是从输入树的那个节点开始的也很简单，我们可以从这个节点出发沿着输入树向根节点的方向回溯就可以了。如果输入树中有一个节点，从它开始匹配了一个指令模板中的所有通路字符串，那么就说这个节点匹配了一个指令模版。

在这里我们采用这种技术来确定开始匹配的节点。在输入树的每个节点为每一个指令模板 ti 维持一个位串来记录部分匹配的情况。在输入树的每个节点 n，为每一个模板 ti 设置一个位串 n.bi,位串的长度为 ti 中最长的通路字符串的长度加 1；这个位串的最低位在最右边，从右到左依次被编号为 0，1，2，等。如果模板 ti 中的一个长度为 2j+1 的通路字符串在输入树的节点 n 处被识别，那么就把 n.bi 的第 j 位置 1。

在一个节点的所有孩子节点被遍历了以后，紧接着就有一个过程 post_process 来识别被匹配的指令树模板。在每一个节点 n，新的位串 n.bi 通过把节点 n 的所有子节点的位串右移一位做 and 运算以后再和旧的位串 n.bi 做 or 运算后得到的。直观的理解是：对于输入树中的一个节点 n，如果 n 的标号匹配了模板 ti 中一个节点的标号并且 n 的所有孩子节点匹配 ti 中深度为 j+1 的节点，那么就说 n 匹配了 ti 中深度为 j 的一个节点。这种方法提供了一个确定树匹配的一个充分必要条件。下面给出这个确定树匹配过程的例程的详细描述：

```
post_process( n )
{
    n.bi = 0;
    if n.state is accepting state
        set_partial( n , n.state );
    for every ti do
    // C(n) is the set of all children of node n;
    n.bi = n.bi or ∏c∈C(n)c.bi/2 ;
    do_reduce(n);
}
```

```
set_partial(n , 6 )
{
for each path string of ti of length 2j+1 is recognized at 6  do
   n.bi = n.bi or 2j;
}
```

在树模板被识别出来以后，下面一个过程根据匹配的结果对输入树进行化简，我们用 do_reduce 来表示这个过程。上面的过程 set_partial(n)根据在这个节点被识别的长度为 2j+1 的 ti 的通路字符串，把其相应的 n.bi 的第 j 位置一。在经过 post_process（n）这个过程处理以后，模板 ti 匹配输入树中以节点 r 为根节点的一棵子树当且仅当 r.bi 为奇数，也就是说，它的最右一位被置为 1。

为了发现了一个覆盖，必须考虑进行化简。也就是说，一旦从输入树中发现了一个像模板 ti 这样的部分，就必须考虑把输入树中这个类似 ti 的部分化简为 ti 的标号，以便发现包含这个匹配的覆盖。由于在这个算法中。我们同时考虑多个匹配，所以化简的过程不能改变输入树的形状；在这里只是改变节点中的一些域的值来反映这个化简的过程。例程 do_reduce(n)来完成这些更新域值和动态规划的任务。

过程 cost(ti ,n)来确定模板 ti 匹配一棵以节点 n 为根节点的子树的开销，一般来讲，这个开销也依赖于模板 ti 中那些为标号的叶节点的匹配开销。例如对于图 5-5 所示的三个树型模板 ti，在输入树的任意一个节点 n,模板 ti 匹配以 n 为根节点的子树的开销可以分别如下来计算：

```
cost( t1,n) = 3+cost of matches at leaves labeled r
cost(t2,n) = 1+cost of matches at leaves labeled r
cost(t3,n) = 1;
```

动态规划的开销被保存在数组 n.cost 中。每一个数组元素 n.cost[l]保存的是所有匹配以这个节点为根节点的子树的形如 l : tree 的模板中的最小开销。这个最小开销所对应的模板编号则被保存在 n.match[l]中；也就是说如果 n.match[l]=j,那么 cost(tj,n) = n.cost[l]并且 l = lj。在初始化时，对于所有节点 n，令 n.cost[l]=∞且 n.match[l]=0。

```
do_reduce(n)
{
  for every ti such that the zeroth bit of n.bi is 1 do
  if cost(ti ,n)<n.cost[li]
    {
     n.cost[li] = cost(ti ,n);
```

```
        n.match[li] = i;
        if n is the root node
            б  = succ(start state,li);
        else
            //n is the kth child of n.parent;
            б  =succ(succ(n.parent.state ,k),li);
        if  б  is   an accepting state
            set_partial(n ,  б )
        }
}
```

以图 5-7 所示的一个输入树为例：

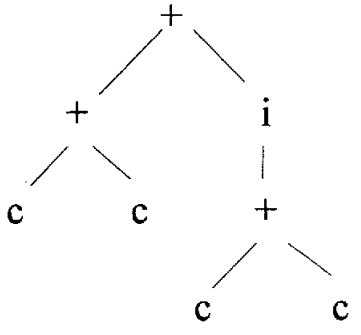

图 5-7 输入树

经过上面介绍的自顶向下的模式匹配算法的处理以后，最终得到的图 5-7 的覆盖如图 5-8 所示：

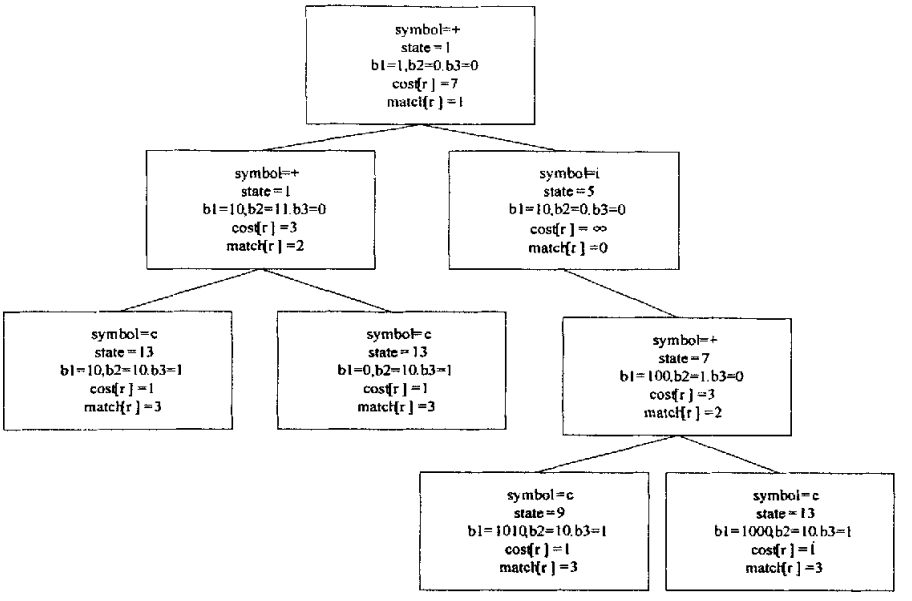

图 5-8 输入树的匹配

有了上面的匹配结果，然后再自顶向下遍历这棵树来确定最终的唯一覆盖。可以看到，这个覆盖以模板 t1 匹配以根节点为根节点的一棵子树开始。在 t1 匹配了这棵子树以后，寻找下一个要匹配的子树的根节点，明显的它们分别为根节点的左子节点和最右下角的符号为 c 的叶节点。在这里我们可以看到，模板 t2 虽然匹配了右子树中以符号＋节点为根节点的一棵子树，但最终的覆盖并没有包含这个匹配。这正是在这个自顶向下的模式匹配算法中结合了动态规划算法的结果。

5.2.3 动态规划算法

从 5.2.2 节的分析不难发现，自顶向下的模式匹配算法之所以能够最终确定输入树的一个唯一的最优覆盖，主要是因为它结合了动态规划算法的思想。

Aho 和 Johnson 曾提出一种基于动态规划的原则为表达式产生寄存器机器机器代码的算法。在这里采用了这个算法的简化形式，首先我们把寄存器分配和指令选择隔离开来分别加以处理，同时对于输入树的每一个子树用一个标量而不是向量来刻画其开销。标量开销通过模板中的开销表达式来计算。这样当多个匹配同时发生时，就可以根据其开销值的不同，选择开销值最小的覆盖。

§5.3 算法的改进和实现

5.3.1 算法思想

不难看出 5.2 介绍的方法能够很好的解决最优代码选择的问题，而且可以很容易的扩展上述算法，使得它不再仅产生唯一的最优覆盖，而是在每个节点处都产生多个可选的最优覆盖。同时考虑到实现的复杂度[14][16]，在我们的实现中采用了下面介绍的方法。

采用如下的两个过程来处理输入树的覆盖问题，使得经过处理以后，在输入树的每个节点处都标注多个可选的最优匹配。第一个过程为 label(p)，这个过程自底向上，从左到右遍历输入树 p，如果存在这个输入树的一个覆盖，那么这个过程将寻找以最小的开销覆盖输入树的指令模板。否则它返回不存在输入树 p 的覆盖。在每个节点处标注了(M , C)就表示与外部编号 M 对应的模板匹配了以这个节点为根节点的一棵子树，其开销为 C。假设模板 M 形如 nonterm: tree;C 为它的开销，nonterm 为某一个特定的非终结符标号。只有当 C 小于所有匹配节点 n 且形如 nonterm: treen 的模板的开销时，这个节点才能被标注为(M,C)。下面用例子来说明这个问题：

假设存在如下的指令模板的集合：

```
%term   ADDI=1    ADDRLP=2   ASGNI=3
%term   CNSTI=21   CVCI=22   IOI=23   INDIRC=67
%%
stmt : ASGNI(disp,reg) = 4   (1);
stmt : reg = 5;
reg : ADDI(reg ,rc) = 6 (1);
reg : CVCI(INDIRC(disp)) =7   (1);
reg : IOI = 8;
reg : disp = 9;   (1);
disp :   ADDI(reg , con) = 10;
disp :   ADDRLP = 11;
rc   :    con = 12;
rc   :    reg = 13;
con :    CNSTI = 14;
con :    IOI = 15;
```

正如在 4.1.2 节所描述的那样，前两句声明终结符，他们分别表示数据流树中的操作符和操作数类型（即叶节点的类型），例如 ADDI 和 ASGNI 为二元操作符，

分别表示整数加和整数赋值操作，CVCI 和 INDIRC 为一元操作，分别表示把一个字符型数据扩展为一个整数数据和从内存中取一个字符型数据；而 ADDRLP,CNSTI 和 IOI 则都是叶节点，其中 ADDPLR 表示一个局部变量的地址，CNSTI 表示整形常量，IOI 则专指 0。%%下面的部分则是具体的指令模板，每个模板都和一个唯一的正整数相联系，这个正整数被称之为外部规则标号，这个外部规则编号被用来向用户提供的语义动作例程传递哪个指令模板匹配了某个节点的信息。每个规则后面都跟着一个非负整数的开销值，默认的为 0。

同时假设如图 5-9 所示的数据流树：

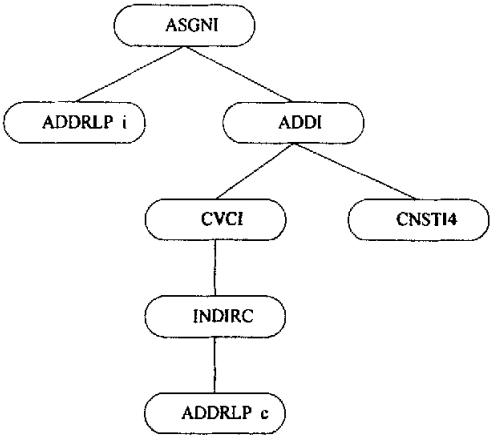

图 5-9 表达式 i=c+4 的中间表示树

这个数据流树经过过程 label(p)的标注以后，所得的结果如图 5-10 所示。

在图 5-10 的每个节点处，标注的第一个数字为指令模板的外部编号，紧接着为模板本身，最后一个数字为用该模板匹配一棵子树时所计算出来的开销值。容易看出，指令模板中第 10 行左边为非终结符 disp 的模板匹配了以节点 ADDI 为根节点的一棵子树，这也就意味着这棵子树也匹配所有仅有 disp 在模板右边的模板，即外部编号为 9 的模板，根据传递性，它也匹配外部编号为 5 和 13 的模板，但是由后面三个指令模板所计算出来的开销值也都为 2，不小于在 ADDI 节点上已经标注的左边为相同的非终结符的相应的模板的开销。所以模板为 6 的模板并没有被记录下来。这正是采用了动态规划思想的结果。明显的，在把中间语言转换为通用的简单指令的过程，这种做法是完全正确和高效的。但是当我们考虑通过代码选择来产生 SIMD 指令时，这种做法已经不能满足我们的要求了，正如在前面所介绍的那样，对于像 SIMD 指令这样的复杂指令，不可能用一条指令模板来描述，而是要用两条或多条指令模板才能够描述清楚，这就要求我们在寻找输入树的覆盖时，不能仅像上面那样只保留唯一的最优覆盖，而是把开销相同的几个可选最优覆盖都保留下来。另外在 IMPACT 编译器中，由于中间表示 Lcode 的

简单性，使得除了隶属于 SIMD 指令的模板可能和另外的一条简单指令的模板冲突外，两条简单指令的模板不可能发生冲突。这就为我们解决这个问题提供了方便，即只需把开销相同的模板都保留下来即可。

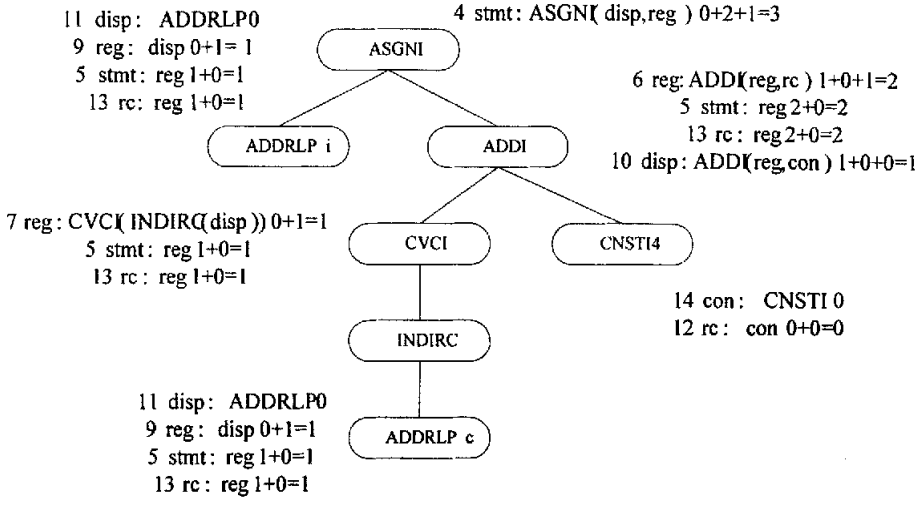

图 5-10 标注后的中间表示树

一旦标注完成以后，就可以通过深度优先遍历输入树来化简它，同时执行相应的语义动作，比如产生最终的目标代码。这个过程用 reduce(p)表示。它主要有三个过程组成：一个过程返回匹配的树模板的外部编号 M，一个过程确定要递归遍历的下一棵子树，最后还有一个过程就是相应的语义动作。

5.3.2 设计与实现

为了记录在每个节点处的标注结果，正如在 4.2.1 节所示的那样，在数据流树节点的数据结构中增加了 struct { STATE_TYPE state;} x 这样的一个域，所以当一个输入树被标注了以后，这个状态域就被赋予一个“状态号”，它编码了所有关于匹配和开销的信息。这个“状态号”实际上是保存这些数据的记录的指针。这个记录类型的数据结构如图 5-11 所示：

```
struct burm_state {
    int op;
    struct burm_state *left, *right;
    short cost[6];
    struct {
        unsigned burm_stm:6;
        unsigned burm_reg:9;
        unsigned burm_regmac:2;
```

```
        unsigned burm_regpred:8;
        unsigned burm_addr:3;
    } rule;
};
```

图 5-11 burm_state 的数据结构

在这里 op 域为被匹配节点的操作符，struct burm_state *left,*right 则分别表示该节点的左右子节点的状态。cost[6]则记录了匹配该节点的模板的最小开销，这个数组用模板集中的非终结符来索引，为此把这些非终结符进行如下编号：

```
#define burm_stm_NT 1
#define burm_reg_NT 2
#define burm_regmac_NT 3
#define burm_regpred_NT 4
#define burm_addr_NT 5
```

例如，cost[burm_stm_NT]就记录了所有形如 stm : Treen 且匹配这个节点的所有模板中的最小开销。rule 域中的每一个位串则分别和 cost 数组中的相应元素相对应，比如 rule.burm_stm_NT 则记录了具有最小开销的那个左边标号为 stm 的模板的外部编号。在这里出于节省存储空间的考虑，没有采用数组 rule[6]来记录匹配规则的信息，而是用了位串。因为对于一个非终结符，以它为标号的模板的数目是有限的。比如对于非终结符 regmac,在我们的机器描述中，以它为左边标号的规则只有 2 个，所以采用两位位串来表示这个信息就足够了，而不必用一个 short 形的域。由于采用了这样的优化处理，我们还需要用一个表来恢复匹配的规则的编号。这个表(只是表的一部分)如下所示：

```
static short burm_decode_regmac[] = {
    0,
    11,
    12,
};
static short burm_decode_addr[] = {
    0,
    520,
    521,
    522,
    525,
};
```

另外，我们采用了如下的宏作为接口来访问 burm_state 数据结构中的各个域.OP_LABEL,LEFT_CHILD,RIGHT_CHILD 分别返回节点外部符号编号，以及它的左右孩子节点。STATE_LABEL 则返回该节点的状态记录的指针。

正如在 5-3 节介绍的那样，该算法的实现分两个过程进行。如图示：

```
void gen(NODEPTR_TYPE p){
        if (burm_label(p) == 0)
                fprintf(stderr, "no cover\n");
        else
                dumpCover(p,1,0);
}
```

第一个过程深度优先遍历输入树，标注每个节点的状态信息。其实现方法如下：

```
static void burm_label1(NODEPTR_TYPE p) {
    burm_assert(p, PANIC("NULL tree in burm_label\n"));
    switch (burm_arity[OP_LABEL(p)]) {
    case 0:
        STATE_LABEL(p) = burm_state(OP_LABEL(p), 0, 0);
        break;
    case 1:
        burm_label1(LEFT_CHILD(p));
        STATE_LABEL(p) = burm_state(OP_LABEL(p),
            STATE_LABEL(LEFT_CHILD(p)), 0);
        break;
    case 2:
        burm_label1(LEFT_CHILD(p));
        burm_label1(RIGHT_CHILD(p));
        STATE_LABEL(p) = burm_state(OP_LABEL(p),
            STATE_LABEL(LEFT_CHILD(p)),
            STATE_LABEL(RIGHT_CHILD(p)));
        break;
    }
}
```

可以看到，burm_label()函数调用了函数 burm_state()，而函数 burm_state()参

数的个数则根据操作符的元数来确定，对于一元操作符和叶子节点分别忽略最后一个参数或后两个参数（忽略的参数用 0 来表示）。至于 burm_state()函数的实现则采用了 5.2 节的树匹配和动态规划的算法。由于在那里已经进行了较为详细的介绍，这里就不再重复。

第二个过程自顶向下再次遍历标注以后的输入树，确定匹配以该节点为根节点的子树的模板的外部编号 M，然后根据该模板确定下一个将要匹配的子树的根节点同时递归的处理这些子树。当所有子树处理完了以后，紧跟作与模板 M 相应的语义动作。具体过程如下所示：

```
static void dumpCover(Tree p, int goalnt,int indent){
  int eruleno = burm_rule(p->x.state,goalnt);
  short * nts = burm_nts[eruleno];
  Tree kids[10];
  int i;
  id = p->node_id;
  if(id==0)
      L_punt("In the dumpCover:node id in the tree cannot be 0");
  calcu_rule(id); /*把每个节点处标注的所有规则放到数组 r[id][]中*/
  burm_kids(p,eruleno,kids);
  for (i = 0; nts[i]; i++)
        dumpCover(kids[i],nts[i],indent + 1);
  #ifdef _DEBUG_
  for(i = 0;i < indent; i++)
  fprintf(stderr, " ");
  fprintf(stderr , "%d %s\n",eruleno,burm_string[eruleno]);
  #endif
  }
```

注意：在这里函数 calcu_rule()把每个节点处的多个可选最优覆盖记录在数组中，整数线性规划根据这些信息来最终确定唯一最优覆盖。burm_rule()函数确定模板的外部编号，burm_kids()函数确定下一个即将匹配的子树的根节点。整数线性规划为第六章的主要内容。而后 semantic()函数则接受节点 p 以及匹配该节点的模板的外部编号，从而执行相应的语义动作。

§5.4 小结

经过第四章的准备以后，我们有了对目标机器指令集的描述以及描述中间代码 Lcode 的数据流图(DFG)。在此基础上，第五章把一种快速的树模式匹配算法和一种高效的动态规划算法结合，找到一种解决代码选择问题的有效途径，而且还改进了这一算法，使得输入树在经过这一算法的处理以后，不是生成唯一的最优覆盖而是生成多个可选的最优覆盖，为进一步从整个 DFG 的角度确定最终的覆盖做好了准备。在下面的一章将重点介绍通过整数线性规划的方法，在多个可选的最优覆盖中确定一个最终的覆盖，使得这个覆盖能产生尽可能多的 SIMD 指令。

第六章 覆盖选择

经过改进的树匹配和动态规划算法的处理以后，使得 DFG 图中的每个节点处都标注了多个可选的最优匹配。本章就叙述如何通过整数线性规划的方法来确定最终的唯一覆盖。在这一阶段，我们的目标就是最终选定的对整个 DFG 图的覆盖能够尽可能多的产生 SIMD 指令。在这里把这个问题转化为一个整数线性规划的问题[10]。

§6.1 生成最优覆盖的约束

一个合理并且满足最大限度的生成 SIMD 指令的覆盖所应满足的约束主要包括以下的几个方面：

约束一：对于 DFG 图中的每个节点 ni，在 DFG 覆盖阶段都将返回一个可选模板的集合 R(ni)，这个集合中每一个模板都以最小的开销匹配节点 ni。在这里引入布尔变量 xij 来表达是否选择模板 rj∈R(ni)来匹配（或不匹配）节点 ni。即如果 xij =1,就表示最终选择模板 rj∈R(ni)来匹配节点 ni，否则就表示选择了集合 R(ni)中的其它模板来匹配这个节点。一个有效的代码选择过程都严格要求每个节点被唯一的一条模板所覆盖，因此在每个节点处，我们强加如下的约束：

$$\sum_{r_j \in R(n_i)} x_{ij} = 1 \qquad \text{（约束一）}$$

约束二：如果为某个节点 n_i 选择了某个模板 r_j,那么就会对这个节点的孩子节点的覆盖问题暗含一些约束。例如，如果节点 n_i 被如下的规则覆盖：reg_lo : Lop_ADD(reg_lo,reg_lo),那么就必须确保 n_i 的第一个和第二个孩子节点能够从非终结符 reg_lo 推导出来，也就是说 Lop_ADD 操作的两个操作数必须寄存器的低 16 位子寄存器中。推广到一般情况来讲，对于节点 n_i，设 x_{ij}=1,即选择了模板 r_j ∈R(n_i)来匹配节点 n_i。同时假设节点 n_k 为节点 n_i 的第 k 个孩子且为节点 n_k 选择了模板 r_l∈R(n_k)。由于节点 n_k 为节点 n_i 的第 k 个孩子，所以模板 r_l 左手边的非终结符必须等于 r_j 右手边的第 K 个非终结符，把这个非终结符记为 n_{tk}。设 RK(n_k) ⊂R(n_k)表示使得 LHS(r_l)= n_{tk} 且匹配节点 n_i 的模板的集合，那么节点 n_i 和节点 n_k 之间的这种相关性可用如下的不等式来表达：

$$x_{ij} \le \sum_{r_l \in R_m(n_k)} x_{kl} \qquad \text{（约束二）}$$

约束三：选择 SIMD 指令的约束：这类约束将确保将指令有效的打包为 SIMD 指令，为此，我们引入了 SIMD 对的概念。如果 DFG 中的一对节点(n_i , n_j)满足如下的三个条件，那么就称节点对(n_i , n_j)为 SIMD 对[5]。（1）n_i 和 n_j 之间没有调度优先关系。（2）按照树文法规则，n_i 可能存储在一个寄存器的高 16 位而 n_j 也可能存储在一个寄存器的低 16 位。（3）如果 n_i 和 n_j 表示 16 位的 load 或 store 操作且 a_i 和 a_j 为相应的存储器地址，那么 a_i-a_j 的差应为 16 位数据的存储长度。

假设 S 为 DFG 中所有的 SIMD 对的集合，那么包含在任何 SIMD 对中的节点都可能被映射为一条 SIMD 指令，然而，我们必须确保任何被选择的 SIMD 指令只覆盖 DFG 节点对中的一对，而且任何一个节点也至多被一条 SIMD 指令所覆盖。

为了表达上面的这个约束，我们引入了一个新的布尔变量 y_{ij}，当 y_{ij}=1 时，它表示把节点 n_i 和 n_j 捆绑成一条 SIMD 指令。对于任何节点 n_i，假设 $R_{hi}(n_i) \subset R(n_i)$ 和 $R_{lo}(n_i) \subset R(n_i)$分别为操作在高和低子寄存器上的规则的集合。那么如果 n_i 被规则 $r_k \in R_{hi}(n_i)$所覆盖，那么必定存在节点 n_j 使得$(n_i,n_j) \in P$ 且 n_j 被规则 $r_l \in R_{lo}(n_i)$所覆盖。反之亦然。这一约束可用下面两式来表达：

$$\sum_{j:(n_i,n_j)\in P} y_{ij} = \sum_{r_k \in R_{hi}(n_i)} x_{ik}$$

$$\sum_{j:(n_j,n_i)\in P} y_{ji} = \sum_{r_k \in R_{lo}(n_i)} x_{ik} \qquad \text{（约束三）}$$

约束四：保持可调度性的约束：这类约束用来确保代码选择的决定不会引起调度死锁问题。对于 DFG 中的任何节点 n_i，让 pred(n_i)表示必须在 n_i 之前被调度的节点的集合，succ(n_i)表示必须在节点 n_i 之后被调度的节点的集合。如果 DFG 中的 SIMD 对(n_i,n_j)被一条 SIMD 指令 I1 所覆盖且存在另外一个 SIMD 对(n_k,n_l)，其中 $n_k \in$ pred(n_j)且 $n_l \in$ succ(n_j)或反之，那么就必须确保 SIMD 对(n_k,n_l)不能在被另外的 SIMD 指令 I2 所覆盖。否则将导致调度死锁，如图 6-1 所示：

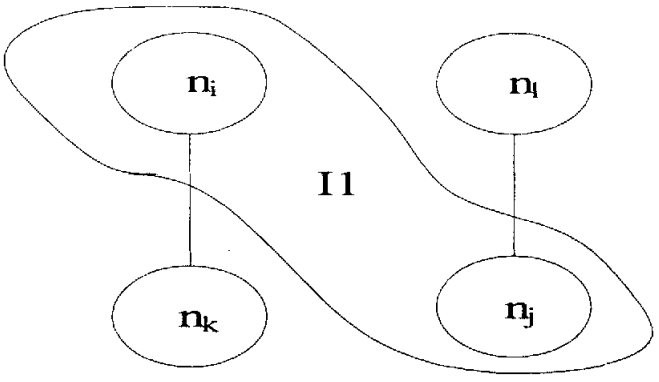

图 6-1 死锁示意图

对于任何 SIMD 对(n_i,n_j)，令 $S_{ij}=\{(n_k,n_l)|n_k \in pred(n_i)$且 $n_l \in succ(n_j)$或反之$\}$，那么对于任意的 SIMD 对(n_i,n_j)和(n_k,n_l)必须强加如下的约束来避免调度死锁：

$$y_{ij} + y_{kl} \leq 1 \qquad \text{（约束四）}$$

目标函数：为了得到在上面的正确性约束下的最优的代码选择，我们必须保证在整个 DFG G 中，最大限度的利用 SIMD 指令。对于任意节点 n_i，令 $T(n_i)=R_{hi}(n_i) \cup R_{lo}(n_i) \subset R(n_i)$表示所有作用在子寄存器上的规则的集合，那么就必须让下面的目标函数取最大值：

$$f = \sum_{n_i \in G} \sum_{r_j \in T(n_i)} x_{ij} \qquad \text{（目标方程）}$$

§6.2 实例分析

仍以第五章的 C 语言程序为例，在前面几章，已经完成了如下几方面的工作：由 C 语言生成其 Lcode 表示，由 Lcode 构造基本块的 DFG 表示，把 DFG 分解为多个 DFTs，给每个 DFT 标注多个可选的最优覆盖，而本章的实验主要是从这多个最优覆盖中最终确定一个能尽可能多的产生 SIMD 指令的覆盖。

根据 6.1 节介绍的整数线形规划的方法，生成了如下约束条件和目标方程：

y113+y125+y136+y159+y613+y915;

row1: x12192+x12193+x12194=1;

row2: x11192+x11193+x11194=1;

row3: x13361+x13362+x13363=1;

row4: x15320+x15321+x15322=1;

row5: x3192+x3193+x3194=1;

row6: x5192+x5193+x5194=1;

row7: x6361+x6362+x6363=1;

row8: x9320+x9321+x9322=1;

row9: x9320<=x6361;

row10: x9321<=x6362;

row11: x9322<=x6363;

row12: x6363<=x5194;

row13: x6363<=x3194;

row14: x6362<=x5193;

row15: x6362<=x3193;

row16: x6361<=x5192;

row17: x6361<=x3192;

row18: x15320<=x13361;

row19: x15321<=x13362;

row20: x15322<=x13363;

row21: x13361<=x11192;

row22: x13361<=x12192;

row23: x13362<=x11193;

row24: x13362<=x12193;

row25: x13363<=x11194;

row26: x13363<=x12194;

row27: y113=x11193;

row28: y125=x12193;

row29: y136=x13362;

row30: y159=x15321;

row31: y613=x6362;

row32: y915=x9321;

row33: y113=x3194;

row34: y125=x5194;

row35: y136=x6363;

row36: y159=x9322;

row37: y613=x13363;

row38: y915=x15322;

```
int x13362,x13363,x15321,x15322,x6362,x6363,x9321,x9322;
int x12192,x12193,x12194;
int x11192,x11193,x11194;
int x13361,x13362,x13363;
int x15320,x15321,x15322;
int x3192,x3193,x3194;
int x5192,x5193,x5194;
int x6361,x6362,x6363;
int x9320,x9321,x9322;
int y113,y125,y136,y159,y613,y915;
```

其中第一行为目标方程，后面为约束条件。用 lp_solver 来求解，所得的结果为：

```
y915=0;
y113=1;
y125=1;
y136=1;
y159=1;
y613=0;
```

从这里可以看出，目标方程的最大解为 4，即从该 C 语言程序中最多能挖掘出 4 条 SIMD 指令，这和手工实现的汇编代码是吻合。即在 DFG 图中，节点 11 和节点 3、节点 12 和节点 5、节点 13 和节点节点 6、节点 15 和节点 9 都能够捆绑生成 SIMD 指令。而后根据上述计算结果，作相应的语义动作，生成最终的 Lcode 代码即可。

§6.3 小结

从上述各章的介绍可以看出，用本文第三章提出的实现框架解决支持 SIMD 的代码选择问题是完全可行的。它扩展了代码选择的视角，使得从整个基本块的角度来考虑最优代码的生成问题而不再局限于逐条语句分析，基本遵循了综合-分解-综合这样的分析问题、解决问题的方法和思路。该实现框架还具有良好的可扩展性，增加其它的 SIMD 指令，只需修改相应的指令模板描述即可。但具体的实现过程还在不断改进中，同时对于最终生成相应的汇编代码，还需要对后端的指令调度[29]和寄存器分配模块[28]进行相应的修改。这都是需进一步完善的工作。

第七章 结束语

本文讨论了支持 SIMD 指令的代码选择和优化技术。采用这种技术的代码选择器具有易修改的特点，它将机器指令集的描述从匹配算法中剥离出去，使得匹配算法本身与机器无关。因此，当修改或增加其他指令时，只需修改相应的机器描述即可。同时该技术改进了树匹配和动态规划算法并且结合了整数线性规划的思想。作为尝试，在具体的实现中，我们只是考虑了对部分 SIMD 指令的支持，至于对其它的 SIMD 指令的支持还有待于进一步扩充和完善。对于这一技术还可以从以下的几个方面去进一步的研究和探讨。

第一：扩展这一技术，提供对其他的复杂指令的支持[6]。下面的例子说明了这个问题。

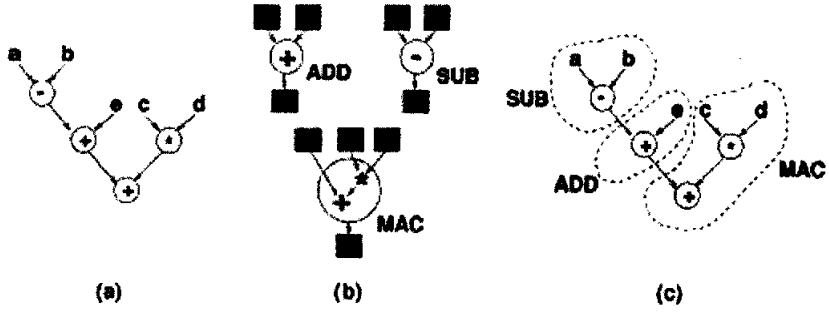

图 7-1 代码选择的形象化表示

在上图中(a)显示的一个数据流树，(b)显示的是三个指令模板，容易看出这其中包含了一个复杂指令 MAC（先加后乘指令）。(c)则给出了用(b)中的指令模板的实例对(a)中的数据流树的一个可能的覆盖，每一个指令模板的实例覆盖了数据流树的一部分。对于这样的一个问题，我们发现它和 SIMD 指令的生成极为相似。因为它们都在于开发指令间的并行性。

第二：进一步增强 SIMD 指令在低功耗编译中的作用。已经有实验表明，SIMD 指令可以使程序的执行时间在减少 76%的同时，其功耗也减少大约 72%左右。因此，可以说凡是提供了对 SIMD 指令支持的编译器，也同时提供了对低功耗编译的支持。但并不是说仅提供对这些指令的支持就足够了。实际上还有很多针对 SIMD 指令的编译优化技术值得我们去进一步研究。比如 loop fission 等。

第三：对指令级功耗模型的研究。为编译器的开发提供科学、合理的衡量程序的功耗大小的标准。

致谢

在本文即将完成之际，作者谨向所有给予我指导、关心、支持和帮助的老师、领导、同学和亲人致以衷心的感谢！

首先衷心感谢我的导师刘春林教授！

衷心感谢刘老师两年多来对我在生活、学习和工作上的关心和帮助，感谢刘老师为我进行课题研究所提供的良好条件和自由的选题空间。正是因为刘老师所提供的这些条件，我才得以在自己感兴趣的领域进行深入的研究，也正是刘老师的严格要求和认真把关，才使本文得以顺利完成。刘老师高深的学术造诣、敏锐的思维、认真负责的工作态度、严谨而不失谦和的长者之风令我十分景仰，是我终生学习的楷模。

衷心感谢607陈书明教授为我进行课题研究所提供的良好机房条件，感谢陈老师在课题研究的过程中给予我的信任和鼓励。陈老师勤奋严谨的工作作风给我留下了深刻的印象，这种耳濡目染的影响必将使我终生受益！

衷心感谢VLIW DSP编译小组的所有成员，包括已经离开的和新近加入的。尤其要感谢胡定磊博士和陈惠彬师兄，他们为我深入了解IMPACT编译器给予了多方面的帮助，并且他们所做的VLIW DSP编译器的前期研究为我的课题创造了条件。感谢由我指导进行毕业设计的本科生卢建平，他完成了一些关于中间代码转换的实现问题。感谢王凤琴，经常和她讨论加深了我对一些问题的理解。

最后，感谢学院队领导和同学对我的关心和帮助。感谢我的母亲和家人对我的关心和体贴。

攻读硕士期间发表的论文

[1] 赵常智，刘春林，胡定磊，陈书明. 一种支持 SIMD 指令的表驱动的代码注释算法. 计算机应用研究.2006.5-6 拟刊出.

参考文献

[1] Alfred V.Aho,Mahadevan Ganapathi,Steven W.K.Jiang. Code Generation Using Tree Matching and Dynamic Programming[C]. ACM Transactions on Programming Languages and System , vol 11 , NO.4 , October 1989 , Pages 491-516.

[2] Rainer Leupers, Code Selection for Media Processors with SIMD Instructions[C]. Design , Automation, and Test in Europe 2000, pp. 4-8.

[3] Rainer Leupers ,Steven Bashford. Graph-based Code Selection Techniques for Embedded Processors[C].ACM Transaction on Design Automation of Electronic Systems, Vol.5,No.4,October 2000 , pp.794-814.

[4] Rainer Leupers , Code Generation for Embedded Processors[C]. 2000 IEEE 1080-1082/00.

[5] Steven Bashford,Rainer Leupers,Constraint Driven Code Selection for Fixed-Point DSPs[C].36th Design Automation Conference (DAC),1999.

[6] Rainer Leupers,Peter Marwedel,Instruction Selection for embedded DSPs with Complex Instructions[C].In European Design Automation Conference(EURO-DAC).

[7] M.Lorenz,L.Wehmeyer,T.Drager.Energy aware Compilation for DSP swith SIMD Instructions[C]. In Proc. Of LCTES/SCOPES , 2002.

[8] Christopher W.Fraser,Engineering a simple,Efficient Code-Generator Generator [C], ACM Letters on Programming Language and Systems,vol.1,NO.3,September 1992,pp213-226.

[9] 胡定磊，陈书明，刘春林.VLIW DSP 编译器的构造[J].高技术通讯，2002 年增刊.

[10] 赵常智，刘春林，胡定磊，陈书明.一种支持 SIMD 的表驱动的代码选择技术。计算机应用研究，2006 年 5 月拟刊出.

[11] J. Glossner, J. Moreno, M. Moudgill, J. Derby, E. Hokenek, D. Meltzer, U. Shvadron, and M. Ware, "Trends in compilable DSP architecture," in proceedings of 2000 Workshop on Signal Processing Systems, pp. 181-199, October 2000.

[12] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," The Journal of Supercomputing, vol. 7, pp. 9-50, January 1993.

[13] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," in Proceedings of the 18th International Symposium on Computer Architecture, pp. 266-275, May 1991.

[14] R. M. Graham, Principles of Systems Programming, John Wiley & Sons, 1975.

[15] D. Salomon, Assemblers and Loaders, Ellis Horwood, 1993.

[16] D. M. Dhamdhere, Systems Programming and Operating Systems, Second Revised Edition, 北京: 清华大学出版社, 2001.

[17] J. R. Levine, T. Mason, and D. Brow. 杨作梅, 张旭东 等译. lex 与 yacc. 第二版, 北京:机械工业出版社, 2003.

[18] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[19] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.

[20] S. A. Mahlke, "Design and Implementation of a Portable Global Code Optimizer," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[21] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," The Journal of Supercomputing, vol. 7, pp. 229-248, January 1993.

[22] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1991.

[23] B. T. Sander, "Performance Optimization and Evaluation for the IMPACT X86 Compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[24] J. Gyllenhaal, "A Machine Description Language for Compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[25] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "Hmdes version 2.0 specification," Tech. Rep. IMPACT-96-3, The IMPACT Research Group, University of Illinois, Urbana, IL, 1996.

[26] S. Aditya, V. Kathail, and B. R. Rau, "Elcor's machine description system:versin 3.0." Tech. Rep. HPL-98-128, Hewlett-Packard Laboratories, 1998.

[27] B. R. Rau, V. Kathail, and S. Aditya, "Machine-description driven compilers for epic processors," Tech. Rep. HPL-98-40, Hewlett-Packard Laboratories, 1998

[28] R. E. Hank, "Machine Independent Register Allocation for the IMPACT-I C Compiler", MS thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, 1993.

[29] P. Faraboschi, G. Desoli, J. A. Fisher, "Clustered Instruction-Level Parallel Processors," Tech. Rep. HPL-98-204, Hewlett-Packard Laboratories, 1998.

[30] Rainer Leupers, Instruction Scheduling for Clustered VLIW DSPs, IEEE PACT, 2000, pp. 291-300.

[31] Scott A. Mahlke, William Y. Chen, Pohua P. Chang, and Wen-mei W. Hwu. Scalar Program Performance on Muliple-Instruction-Issue Processors with a Limited Number of Registers. Proceedings of the 25th Annual Hawaii Int'l Conference on System Sciences, Jan. 6-9, 1992, pp. 34-44.

[32] 袁正才, 刘春林, 胡定磊, 陈书明. 分簇 VLIW DSP 调度技术. 计算机应用研究, 2004, 21 (8): 80-82.

[33] 袁正才，刘春林，胡定磊. 一种基于机器描述的 VLIW DSP 编译技术. 计算机工程, 2004, 30 (22): 79-81.

[34] 陈惠斌，刘春林，胡定磊，陈书明. YHFT-D4 汇编器的设计与实现. 电脑与信息技术. 2005.1.

[35] M.Lorenz,T.drager,Rainer leupers, compiler based Energy Savings by SIMD Operations.LCTES'02-SCOPES'02,June 19-21,2002,Berlin,Germany.