软件流水中的一种数据分配算法*

罗 军 汤志忠 张赤红 于 涛

(清华大学计算机系 北京 100084)

E-mail:tzz_dcs@tsinghua.edu.cn

摘要 数据元素的存储器分配是指令级并行优化编译过程中不可回避的一个关键性问题.该问题解决得好坏直接关系到编译优化的效率.本文第 1 节主要介绍 ILSP(interlaced inner and outer loop software pipelining)算法的基本思想.第 2 节以距阵乘法为例阐述了在 ILSP 算法下多重循环中数据元素的存取特点.第 3 节则从理论上对该特点进行了深入的分析研究,同时就一般多重循环给出了一个行之有效的 ILSP 算法下数据元素内存分配算法.第 4 节给出一个实验比较结果.最后是结论.

关键词 指令级并行,ILSP 算法,存储器分配,数据存取.

中图法分类号 TP311

指令级并行 ILP(instruction-level parallelism)[1~6]是当前国内外学者十分关注的研究课题,它主要包括 ILP 体系结构、ILP 算法与 ILP 优化编译技术等.在 ILP 体系结构中,存储器的分配与寻址方式又是一个重要的问题,它直接关系到 ILP 体系结构的运行效率.但是,通常存储器的分配与寻址方式对 ILP 算法及其具体应用有较强的依赖性.

在科学计算(信号处理、图象处理等)的应用程序中,大部分数据元素的寻址方式都是有规律的面向数组的地址计算.对于这类寻址方式,通常采用地址计数器代替常规地址寄存器的方法,以使数组元素的地址计算不产生目标代码,因而不占用 CPU 时间.这样不仅能够有效地简化编译程序、节约硬件资源,而且可大量地增加程序操作的并行度,显著地加快程序的运行速度.

ILSP(interlaced inner and outer loop software pipelining)内外层交替执行的多重循环流水算法是一个细粒度的软件流水算法.[7~12]它较成功地解决了对多重循环的软件流水问题.但算法对数据元素的存取方式提出了特殊的要求.因此,如何根据 ILSP 算法思想,对这种数据元素的存取方式进行分析并对数据元素进行内存分配,简化地址计算的复杂度,并将地址计算简单地交给硬件实现,则是本文所要解决的.

1 ILSP 算法的基本思想

在 ILP 优化编译技术中,对循环程序主要通过软件流水进行优化.软件流水是开发循环程序指令级并行性的一种有效方法.在不改变程序语义的前题下,使不同次循环的操作并行执行,以加快运行速度.一般处理多重循环的软件流水算法有内层流水、外层串行;内层展开法;分层优化法等.ILSP 是新近提出的一种算法.[13]其核心思想是:内层循环与外层循环交替执行,通过同时展开多个外层循环体,对每个外层循环的内层循环串行执行,并从最内层就开始软件流水,使内层与外层的执行不再是分离的两个部分.图 1(a)是一个典型的两重循环的例子,其中 $OP_i(i=1,2,\ldots,5)$分别代表 5 个不同的操作.图 1(b)是图 1(a)的数据相关图.在不存在体间相关,并且资源不受限制条件下,表 1 给出了该循环在 ILSP 算法下的执行情况.

分析 ILSP 算法下多重循环执行过程,可以将 ILSP 的基本特征描述如下:ILSP 算法在对各层循环进行软件流水时,将该层循环内嵌套的各层循环视为只执行一次.每当一个由某内层循环体构成的新模式出现时,当前外层循环暂停执行,而转去对该内层循环进行软件流水,这时外层循环放弃对全部或部分资源的支配权,供该内层使用.内层循环

---

* 本文研究得到国家自然科学基金资助.作者罗军,1971 年生,硕士,主要研究领域为计算机并行算法,并行体系结构,优化编译技术.汤志忠,1946 年生,教授,主要研究领域为计算机并行算法,并行体系结构,优化编译技术.张赤红,1963 年生,副教授,主要研究领域为计算机并行算法,并行体系结构,优化编译技术.于涛,1971 年生,硕士,主要研究领域为计算机并行算法,并行体系结构,优化编译技术.

本文通讯联系人:汤志忠,北京 100084,清华大学计算机系

本文 1997-01-02 收到原稿,1997-02-20 收到修改稿

在完成其流水阶段时,将执行排空操作,在返回父循环后,恢复其父循环对资源的支配权.

```
for(i=1; i<n; i++)                    op1
  {                                    ↓
  op1;                                op2
  for(j=1; j<m; j++)                   ↓
    {                                 op3
     op2;                             ↓
     op3;                            op4
     op4;                             ↓
    }                                op5
  op5;
  }
```

(a)　　(b)

图 1　一个典型的两层循环的例子

表 1　ILSP 算法的基本原理

周期	操作	说明
0	$op1(1,-)$	外层装入
1	$op1(2,-)$　$op2(1,1)$	内、外层同时装入
2	$op1(3,-)$　$op2(2,1)$　$op3(1,1)$	
3	$op1(4,-)$　$op2(3,1)$　$op3(2,1)$　$op4(1,1)$	内层充满,切换
4	$op2(1,2)$　$op3(3,1)$　$op4(2,1)$	内层流水,外层暂停
5	$op2(2,2)$　$op3(1,2)$　$op4(3,1)$	
...	...	
x	$op2(3,m)$　$op3(2,m)$　$op4(1,m)$	
$x+1$	$op1(5,-)$　$op2(4,1)$　$op3(3,m)$　$op4(2,m)$　$op5(1,-)$	第 1 次内层排空,同时外层装入、流水,同时进行第 2 次内层装入
$x+2$	$op1(6,-)$　$op2(5,1)$　$op3(4,1)$　$op4(3,m)$　$op5(2,-)$	
$x+3$	$op1(7,-)$　$op2(6,1)$　$op3(5,1)$　$op4(4,1)$　$op5(3,-)$	
$x+4$	$op2(4,2)$　$op3(6,1)$　$op4(5,1)$	再次内层流水,外层暂停
...	...	
y	$op2(n,m)$　$op3(n-1,m)$　$op4(n-2,m)$	
$y+1$	$op3(n,m)$　$op4(n-1,m)$　$op5(n-2,-)$	最后,整个循环排空
$y+2$	$op4(n,m)$　$op5(n-1,-)$	
$y+3$	$op5(n,-)$	

注:$opk(i,j)$ 表示操作 k 处于第 i 个外层循环体与第 j 个内层循环体之中.

2　ILSP 算法下数据元素的存取特点

下面首先从应用程序中较常见的矩阵乘法来分析多重循环中数据元素的存取特点.实现矩阵乘法的多重循环程序如图 2 所示.其中设 $C[i][j]$ 的初值为零.

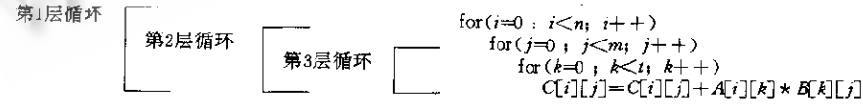

图2 矩阵乘法应用实例

在对图 2 应用实例的编译过程中,该实例的第 1 层循环被同时展开了 9 个循环体.当 $m=n=t=18$ 时,矩阵 $A_{18,18}$,$B_{18,18}$,$C_{18,18}$ 的目标代码执行时数据元素读取顺序分别如图 3(a)~(c)所示.这里我们假定 n,m 和 t 初值均为 18.图中的弧线表示数据元素的读取顺序.如果一个元素同时有两条射出弧,则标有数字的弧线(比如 $a_{8,17}\xrightarrow{18}a_{0,0}$)优先执行.只有当该弧被连续执行并达到标注数字所指定的次数后,另一条弧才开始执行.从图 3 可以看出,矩阵 A,B,C 在 ILSP 算法下元素的读取顺序与在串行执行时是完全不同的.以矩阵 $A_{18,18}$ 为例,在图 3(a)中,机器在读取元素 $a_{0,0}$ 后并未立即读取 $a_{0,1}$,而是紧接着读取 $a_{1,0}$,并且沿着第 1 列一直读到 $a_{8,0}$,然后才转而读 $a_{0,1}$.实际上,对于任意 $a_{i,j}$,

$a_{k,h}\in A_{18,18}$，当 $0<j<18, 0\leqslant k<\left\lceil\frac{i}{9}\right\rceil*9, 0\leqslant h<j$ 时，表达式 $first(a_{k,h})\ll first(a_{i,j})$ 恒成立. 其中，$first(a_{p,q})$ 表示元素 $a_{p,q}\in A_{18,18}$ 第1次被机器读取的时刻，符号"≪"表示时刻上的先. 在图3中，矩阵 A 总是有9行元素被读取，即有9行元素同时参加运算. 象这种数据空间被有规律地划分成若干个存储体，且各存储体中的数据元素交叉地被读取并同时参加运算，正是ILSP算法下多重循环中数据元素的主要存取特点.

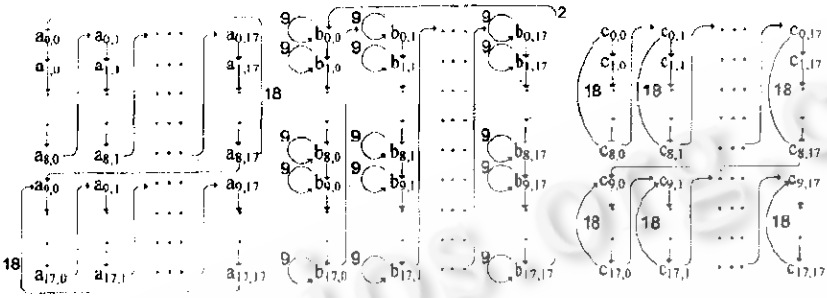

图3 ILSP算法下数据元素存取特点

3 数据元素的存储器分配

对于一个 n 维数组，我们用符号 I_h 表示该数组的第 $h(0<h<n)$ 维，i_h 表示该数组在第 I_h 维上的下标变量. 在下面的讨论中，我们另外用符号 $n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})})$ 表示一个 $n(n\geqslant 1)$ 维数组. 其中 $e(i_h), h=0,\ldots,n-1$，表示数组 X 在第 I_h 维上的下标变量表达式. 该表达式可以表示为"$u*i_h+v$"，$i_h(i_f\neq i_t, f,t=0,1,\ldots,n-1$，并且 $f\neq t)$ 是数组在第 I_h 维上的下标变量，u,v 均为常量. 符号 $SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))$ 表示数组 X 在各维上的下标变量的集合. 即 $SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))=\{i_0\}\cup\ldots\cup\{i_{n-1}\}$. 类似地，我们用符号 $k\text{-}Loop(A)$ 表示一个 k 重($k\geqslant 1$)循环 A，并将 $k\text{-}Loop(A)$ 的层号规定为：从最外层循环到最内层循环，各层循环的层号依次为 $1,2,\ldots,k$. $k\text{-}Loop(A)$ 的第 $i(i=1,2,\ldots,k)$ 层循环记作 $k\text{-}Loop^i(A)$. 符号 $var(k\text{-}Loop^i(A)), len(k\text{-}Loop^i(A)), rec(k\text{-}Loop^i(A)), init(k\text{-}Loop^i(A))$ 分别表示 $k\text{-}Loop^i(A)$ 的循环控制变量、循环次数、循环控制变量的修正值、循环变量的初值. $k\text{-}Loop(A)$ 的各层循环控制变量的集合用 $VAR(k\text{-}Loop(A))$ 表示，即 $VAR(k\text{-}Loop(A))=\{var(k\text{-}Loop^1(A))\}\cup\ldots\cup\{var(k\text{-}Loop^k(A))\}$. $op(k\text{-}Loop(A), n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}))$ 表示 n 维数组 X 在 K 重循环 A 中对存储器的操作.

定义1. 若在 $k\text{-}Loop^j(A)(1\leqslant j\leqslant k)$ 中有 $op(k\text{-}Loop(A), n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}))$，则称 $k\text{-}Loop(A)$ 在第 j 层上对 $n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})})$ 进行包含，记作：$con(k\text{-}Loop(A), j, n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}))$.

一般地，对于 $n(n\geqslant 1)$ 维数组我们有如下定理：

定理1. 设有 $n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}), k\text{-}Loop(A)$，且有 $con(k\text{-}Loop(A), f, n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}))$，其中 $1\leqslant n\leqslant k, 1\leqslant f\leqslant k$，则 $k\text{-}Loop(A)$ 必定唯一地能够化简为一个 n 重循环 $n\text{-}Loop(A*)$，且同时满足以下条件：

(1) 表达式 $con(n\text{-}Loop(A*), n, n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))exists$；

(2) 等式 $OP(n\text{-}Loop^n(A*))=op(k\text{-}loop^k(A))$ 成立，且 $OP(n\text{-}Loop^g(A*))=\varnothing, g=1,\ldots,n-1, OP(n\text{-}Loop^g(A*))$ 是 $n\text{-}Loop^g(A*)$ 中的所有操作的集合；

(3) 等式 $VAR(n\text{-}Loop(A*))=SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))$ 成立；

(4) 令 p 等于 $layer(k\text{-}Loop(A), i_f)$，$q$ 等于 $layer(n\text{-}Loop(A*), i_f)$，其中 $i_f\in SUB(n-Array(X_{e(i_0),\ldots,e(i_{n-1})}))$，则下列4个条件等式成立：

$$var(k\text{-}Loop^p(A))=var(n\ Loop^q(A*)) \tag{1}$$

$$len(k\text{-}Loop^p(A))=len(n\text{-}Loop^q(A*)) \tag{2}$$

$$rec(k\text{-}Loop^p(A))=rec(n\text{-}Loop^q(A*)) \tag{3}$$

$$init(k\text{-}Loop^p(A))=init(n\text{-}Loop^q(A*)) \tag{4}$$

(5) 如果 $layer(k\text{-}Loop(A), i_d)<layer(k\text{-}Loop(A), i_g)$，则 $layer(n\text{-}Loop(A*), i_d)<layer(n\text{-}Loop(A*), i_g)$，其中 $i_d, i_g\in SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))$，$d,g=0,\ldots,n-1, d\neq g$.

证明:(1)存在性:将 k-$Loop(A)$中所有关于 n-$Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})})$的相关循环层找出,保留它们的循环控制变量、循环次数、循环控制变量的修正值、循环变量的初值,去掉循环中的操作,并按照它们在 k-$Loop(A)$中的相对内外层关系构成一个新的多重循环,记为 n-$Loop(A*)$.将 $op(k\text{-}Loop(A),n\text{-}Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})}))$置于 n-$Loop(A*)$中,最后得到的 n-$Loop(A*)$满足定理1中的5个条件,所以就是所要求的多重循环.存在性得证.

(2)唯一性:假设可从 k-$Loop(A)$化简得到两个 n 重循环,分别记作 n-$Loop(B)$,n-$Loop(C)$,且同时满足定理1中的5个条件.由 n-$Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})})$的唯一性和 n-$Loop(B)$与 n-$Loop(C)$都满足条件(2),可得知集合 $VAR(n\text{-}Loop(B))$与集合 $VAR(n\text{-}Loop(C))$相等,因此 n-$Loop(B)$与 n-$Loop(C)$是由 k-$Loop(A)$的相同的 n 个循环所构成.由于 n 重循环的相对内外层的关系在 k-$Loop(A)$中是唯一确定的,而 n-$Loop(B)$与 n-$Loop(C)$同时满足条件(3),因而这 n 个循环在 n-$Loop(B)$与 n-$Loop(C)$的相对关系也必然是唯一的.而条件(1)和(4)、(5)本身的满足条件就是唯一的.综上所述,n-$Loop(B)$与 n-$Loop(C)$是相同的.唯一性得证.

在从 k-$Loop(A)$化简得到 n-$Loop(A*)$后,直接分析 n-$Array(X_{e(i_0),e(i_1),\ldots,e(i_{n-1})})$的存储器分配还有可能较为困难.这是因为出现在实际程序中的数组下标往往是关于数组下标变量的表达式,还需要进一步的化简.本文考虑实际程序中最常见的数组下标为关于下标变量的线性表达式的情况.由以上的分析,算法1给出了一个对 k-$Loop(A)$中的数组下标表达式进行化简并最终得到数组元素的存储器分配方案.

算法1.

$SUP\leftarrow SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}))$,$i\leftarrow k$,$OP_{temp}\leftarrow op(k\text{-}Loop^k(A))$,$\S\leftarrow\varnothing$

while $i\neq 0$ do

begin

　if $(var(k\text{-}loop^i(A))\in SUP)$

　then $\S\leftarrow\S\cup\{i\}$;

　endif

　$i\leftarrow i-1$;

end; /* while end */

$i\leftarrow 1$;

while $\S\neq\varnothing$ do

begin

　temp$\leftarrow$min($\S$);

　$i\text{-}Loop^i(A^*)\leftarrow k\text{-}Loop^{temp}(A^*)$;

　$\S\leftarrow\S-\{temp\}$;

　$i\leftarrow i+1$;

end; /* while end */

$op(n\text{-}Loop^n(A^*))\leftarrow OP_{temp}$,$h\leftarrow 0$,$n\text{-}Loop(A^{**})\leftarrow n\text{-}Loop(A^*)$;

while SUP$\neq\varnothing$ do

begin

　$rec(n\text{-}Loop^i(A^{**}))\leftarrow\nabla u(e(i_h))*rec(n\text{-}Loop^i(A^*))$;

　$init(n\text{-}Loop^i(A^{**}))\leftarrow\nabla u(e(i_h))*init(n\text{-}Loop^i(A^*))+\nabla v$;

　$e(i_h)\leftarrow i_h$;

　SUP$\leftarrow$SUP$-\{i_h\}$;

end; /* while end */

$arrange(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})}),l_0,\ldots,l_{n-1})$;

if $(var(k\text{-}Loop^1(A)\in SUB(n\text{-}Array(X_{e(i_0),\ldots,e(i_{n-1})})))$

then

$$addr(X_{e(i_0),\ldots,e(i_n)})\leftarrow loc(X)+\xi*\left\lfloor\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))*\nabla k}\right\rfloor*\nabla k*len(n\text{-}Loop^2(A^{**}))*len(n\text{-}Loop^3(A^{**}))$$

$$*\ldots*len(n\text{-}Loop^{\ })))*\ldots*len(n\text{-}Loop^n(A^{**}))*\frac{l_2-init(n\text{-}Loop^2(A^{**}))}{rec(n\text{-}Loop^2(A^{**}))}+\ldots+\xi*\nabla k*len(n\text{-}Loop^n(A^{**}))*$$

$$\frac{l_{n-1}-init(n\text{-}Loop^{n-1}(A^{**}))}{rec(n\text{-}Loop^{n-1}(A^{**}))}+\xi*\nabla k*+\xi*\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{ren(n\text{-}Loop^1(A^{**}))}-\xi*\left\lfloor\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))*\nabla k}\right\rfloor=loc(X)$$

$$+\xi*\left\lfloor\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))*\nabla k}\right\rfloor*\nabla k*\prod_{i=2}^{n}len(n\text{-}Loop^i(A^{**})+\xi*\nabla k*\sum_{f=1}^{n}\frac{l_f-init(n\text{-}Loop^f(A^{**}))}{rec(n\text{-}Loop^f(A^{**}))}\prod_{h=f}^{n}len(n\text{-}Loop^h(A^{**}))$$

$$+\xi*\frac{l_1-init(n-Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))}-\xi*\left\lfloor\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))*\nabla k}\right\rfloor$$

else

$$addr(x_{e(i_0),\ldots,e(i_n)})\leftarrow loc(X)+\xi*len(n\text{-}Loop^2(A^{**}))*len(n\text{-}Loop^3(A^{**}))*\ldots*len(n\text{-}Loop^n(A^{**}))*$$

$$\frac{l_1-init(n\text{-}Loop^1(A^{**}))}{rec(n\text{-}Loop^1(A^{**}))}+\xi*len(n\text{-}Loop^2(A^{**}))*\cdots*len(n\text{-}Loop^n(A^{**}))*\frac{l_2-init(n\text{-}Loop^2(A^{**}))}{rec(n\text{-}Loop^2(A^{**}))}+\cdots+\xi*len(n\text{-}Loop^n(A^{**}))*\frac{l_{n-1}-init(n\text{-}Loop^{n-1}(A^{**}))}{rec(n\text{-}Loop^{n-1}(A^{**}))}+\xi*\frac{l_n-init(n\text{-}Loop^n(A^{**}))}{rec(n\text{-}Loop^n(A^{**}))}=loc(X)+\xi*\sum_{f=1}^{n}\frac{l_f-init(n\text{-}Loop^f(A^{**}))}{rec(n-Loop^f(A^{**}))}\prod_{h=f+1}^{n}len(n\text{-}Loop^h(A^{**}))$$

endif

其中 $loc(X)$ 表示数组 $n\text{-}Array(X_{e(i_0),\cdots,e(i_{n-1})})$ 的第1个元素的地址；$addr(X_{e(i_0),\cdots,e(i_{n-1})})$ 表示元素 $X_{e(i_0),\cdots,e(i_{n-1})}$ 的地址；$arrange(n\text{-}Array(X_{e(i_0),\cdots,e(i_{n-1})}),l_1,\cdots,l_n)$ 表示按照 $layer(n\text{-}Loop(A^{**}),l_j)$ 的值的大小从小到大依次重新排列数组 $n\text{-}Array(X_{e(i_0),\cdots,e(i_{n-1})})$ 的下标变量的顺序，$j=0,\cdots,n-1$. 重新排列后所得到的顺序表示为 $l_1,\cdots,l_n$；$\min(\S)$ 实现寻找集合 $\S$ 的最小值的功能；∇u 表示 u 在表示式 $u*i_h+v$ 中的值，∇k 表示 v 在表示式 $u*i_h+v$ 中的值，$i_h\in SUB(n\text{-}Array(X_{e(i_0),\cdots,e(i_{n-1})}))$，$h=0,\cdots,n-1$；$\xi$ 表示一个数据元素所需要占用的内存单元数.

4 实验结果

按照算法1，我们以矩阵 $A_{18,18}$ 为例，其存储器分配如图3所示. 在实际的编译过程中，我们对矩阵 $A_{18,18}$ 展开了9个体.

$$a_{i,k}=loc(A)+\left\lfloor\frac{i}{9}\right\rfloor*18*9+k*9-i-\left\lfloor\frac{i}{9}\right\rfloor;$$

$loc(A)$ 表示数据元素 $a_{0,0}$ 内存中的地址.

本文随机选取了5个程序，分别在采用了和未采用算法1的情况下将ILSP算法的运行结果作了一个比较，其结果如表2所示.

表2 实验比较结果

程序名称	循环层数	采用算法1的实验结果			未采用算法1的实验结果			加速比 (T_2/T_1)
		最内层循环环操作个数	内层循环启动间距	执行时间 T_1(拍)	最内层循环环操作个数	内层循环启动间距	执行时间 T_2(拍)	
矩阵加法	2	4	1	150	10	1	150	1
矩阵乘法	3	6	1	5 841	12	2	11 682	2
卷积乘	3	6	1	333	15	2	666	2
求最大最小值	2	7	1	90	13	2	180	2
快速排序	3	5	1	301	13	2	602	2

从表中可以看到采用算法1可以获得的平均加速比为1.8. 通常，一个多重循环的执行时间主要取决于最内层循环的执行时间. 由于最内层循环的执行时间受硬件资源的限制，因此如果能够减少最内层循环对硬件的需求，就有可能缩小循环的启动间距，并因此缩短最内层循环的执行时间. 由于算法1的目标是将地址计算交给硬件完成，我们可以省去大量的与地址运算有关的操作. 在硬件资源受限的情况下，运算量的减少不仅有助于缩小软件流水的起动间距，而且可以将节约出来的功能部件执行其它操作，从而增加程序执行的并行度和加快程序的执行时间.

5 结束语

本文的核心是针对ILSP算法的特点提出一个有效解决其数据元素内存分配问题的算法，以达到对数据元素的地址计算进行简化并交给硬件完成，增强系统的运行效率的目的. 实验结果表明通过采用算法1显著减少程序的目标代码的长度，缩短程序运行时间. 目前，一个指令级并行及体系结构的设计已经完成，优化编译器及体系结构的模拟已经实现. 系统的运行结果证明了算法1的处理实际问题的正确性和实用性.

参考文献

1 Rau B R, Fisher J A. Instruction-level parallel processing: history, overview and perspective. The Journal of Supercomputing, 1993,1:9~50

2 Colwell R P, Nix P R, Donnell J J *et al*. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers, Aug. 1988,C-37:967~979

3 Fisher J A. Very long instruction word architecture and the ELI-512. In: Proceedings of the 10th Annual International Symposium on Computer Architecture. Stockholm, Jun. 1983. 140~150

4 Rau B R, Glaeser C D. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In: MICRO-14, Oct. 1981. 183~198
5 Jouppi N P, Wall D. Available instruction level parallelism for superscalar and superpipelined machine. In: Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems. Boston, Apr. 1989. 272~282
6 Acosta R D, Kjelstrup J, Torng H C. An instruction issuing approach to enhancing performance in multiple function unit processors. IEEE Transactions on Computers, Sept. 1986, C-35: 815~828
7 Wolf M, Lam M. A loop transformation theory and algorithm to maximize parallelism. In: IEEE Transactions on Parallel and Distributed Systems. Oct. 1991, 2(4): 452~471
8 Rau B R. Iterative modulo scheduling: an algorithm for software pipelining loops. In: MICRO-27. Nov. 30~Dec. 2, 1994. 63~74
9 Lam M. Software pipelining: an effective scheduling technique for VLIW machines. In: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. Jun. 1988. 318~328
10 Su B, Wang J, Tang Z *et al*. A software pipelining based VLIW architecture and optimization compiler. In: MICRO-23. 1990. 17~27
11 Charlesworth A E. An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family. Computer, 1981, 14(9): 18~27
12 Allan V H, Jones R B, RANDALL M *et al*. Software Pipelining ACM Computing Surveys, Sept. 1995, 27(3): 371~473
13 Wang Lei, Tang Zhizhong, Zhang Chihong. Ruminate method-a novel fremework for software pipelinig on nested loops. In: Proceedings of the MPCS'96 Conference on Massively Parallel Computing Systems, May 1996. 475~482

An Algorithm to Data Allocation in Software Pipeling

LUO Jun TANG Zhi-zhong ZHANG Chi-hong YU Tao

(*Department of Computer Science Tsinghua University Beijing* 100084)

Abstract In the optimizing compilation process of instruction-level parallelism, the memory allocation of data elements is an unavoidable key problem, which has a direct effect on the efficiency of the final results of optimizing compilation. In this paper, the first part briefly presents the background of the paper. Part 1 is the principles of a brand-new software pipelining algorithm—ILSP(interlaced inner and outer loop software pipelining). In part 2, taking the example of matrix multiplication, the authors expound the characteristic of data accessing under ILSP algorithm, and make a deep theoretical analysis on that characteristic and conclude a general algorithm on memory allocation of data elements in the nested loops. The experiment is presented in part 4. And part 5, the conclusion.

Key words Instruction-level parallelism, ILSP algorithm, memory allocation, data access.

Class number TP311