HAL
open science

Decomposed software pipelining

J. Wang, Christine Eisenbeis

To cite this version:

J. Wang, Christine Eisenbeis. Decomposed software pipelining. [Research Report] RR-1838, INRIA. 1993. inria-00074834

HAL Id: inria-00074834
https://inria.hal.science/inria-00074834

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N°1838

Programme 2

Calcul symbolique, Programmation et Génie logiciel

DECOMPOSED SOFTWARE PIPELINING

Jian Wang
Christine Eisenbeis

Janvier 1993

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

DECOMPOSED SOFTWARE PIPELINING *

PIPELINE LOGICIEL PAR DECOMPOSITION

Jian WANG† Christine EISENBEIS‡

INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex
FRANCE

Abstract

This report presents a new view on software pipelining, in which we consider software pipelining as an instruction level transformation from a vector of one-dimension to a matrix of two-dimensions. Thus, the software pipelining problem can be naturally decomposed into two subproblems, one to determine the row-numbers of operations in the matrix and another to determine the column-numbers. Using this view-point as a basis, we develop a new loop scheduling approach, called decomposed software pipelining, to exploit instruction-level parallelism for any loop with/without conditional jumps.

Résumé

Ce rapport propose une nouvelle appréhension du problème de pipeline logiciel des boucles, qui consiste à considérer le pipeline logiciel comme la transformation d'un vecteur à une dimension en une matrice à deux dimensions. Le problème de pipeline logiciel est ainsi décomposé naturellement en deux sous-problèmes : détermination des indices de lignes et détermination des indices de colonnes. Sur cette base, nous développons une nouvelle approche pour l'ordonnancement des boucles, qui exploite un parallélisme de bas niveau (entre instructions) pour des boucles pouvant contenir des branchements conditionnels.

*This work was partially supported by ESPRIT Project COMPARE
†Email: Jian.Wang@inria.fr; Tel: 33-1-39635361; Fax: 33-1-39635330.
‡Email: Christine.Eisenbeis@inria.fr.

1 Introduction

Since loop execution dominates total execution time of almost all practical programs, the exploitation of instruction level parallelism for loops is a major challenge in the design of optimizing compilers for high-performance computers such as VLIW, superscalar and pipelined processors [Alm89]. Software pipelining is an effective instruction level loop optimization technique behind which the idea is that the body of a loop can be reformed under the constraints of resources, dependences (including control dependences, loop independent and loop-carried dependences) and cyclicity so that one iteration of the loop can start before the previous iterations finish their execution.

We generally call the software pipelining of branch-free loops the local software pipelining and, correspondingly, the software pipelining of loops with conditional jumps the global software pipelining. In the past ten years a number of local/global software pipelining approaches have been presented [Cha81,Tou84,Aik88,Eis88,Lam88,Bod89,Mun91,Mun92,Su86, Su87,Su91a,Su91b,Ebc87,Nak90,Gao91]. Simultaneously considering the above constraints, these approaches develop various heuristics to construct software pipelined loops. However, the resource constraint and the loop-carried dependences make the software pipelining problem very complicated and difficult so the existing software pipelining approaches can not get a satisfactory time and space efficiency with low computation complexity.

Recently, two new local software pipelining approaches have been presented [Gas92, Eis92, Mun91]. In [Gas92], the given loop is first preprocessed under unlimited resources; then some information from the preprocessing step is available for deleting some edges from the loop's Data Dependent Graph (DDG) such that the DDG is acyclic; finally a list scheduler is used to construct the new loop body under the constraint of resources. Another new approach is presented in [Eis92, Mun91] which can only pipeline the loops whose DDGs are acyclic. It consists of two steps: first, under resource constraint the new loop body is built by "forgetting" the dependences among operations, then the iteration number, called the iteration index, of each operation is determined so that all dependences are satisfied. Inspired by these two approaches, we present a new software pipelining approach, called DEcomposed Software Pipelining. Our basic idea is that we first consider software pipelining as an instruction level transformation from a vector of one-dimension to a matrix of two-dimensions, and then functionally decompose the software pipelining problem into two subproblems, one to determine the row-numbers of operations in the matrix and another to determine the column-numbers. In fact, the former subproblem is the loop-free code scheduling problem which can be effectively solved by list scheduling technique; and the latter is independent of resource constraints and can be easily solved by the classical algorithms of graph theory. We also extend this idea to global software pipelining and present Global DEcomposed Software Pipelining approach to exploit instruction-level parallelism for any loop with conditional jumps.

The rest of this report is organized as follows. In the next section, we present the ideas of decomposed software pipelining. In Section 3, using decomposed software pipelining as a basis, we present two new local software pipelining algorithms. In Section 4 and Section 5, we extend decomposed software pipelining to global software pipelining and present global

decomposed software pipelining approach. In Section 6 we give the discussion and the comparisons with the existing local/global software pipelining approaches and Section 7 is a conclusion.

2 Decomposed Software Pipelining

2.1 A New View on Local Software Pipelining

Local software pipelining is actually an instruction-level loop transformation under the constraints of resources, loop-independent data dependences, loop-carried dependences and cyclicity. We model a loop as a doubly weighted data dependence graph, $G = (O, E, \lambda, \delta)$, called a Loop DDG (LDDG), where O is the set of operations in the loop, E is the set of dependence edges, λ and δ are two non-negative integers associated to each edge, for instance, $e = (op_i, op_j) \in E$, $(\lambda(e), \delta(e))$ denotes that op_j can only be executed $\delta(e)$ cycles after op_i of the $\lambda(e)$th previous iteration has started executing. Not more formally, the local software pipelining problem can be described as follows:

Construct a loop schedule σ, a mapping function from $O \times N$ to N (non-negative integer set), $\sigma(op, i)$ denotes the execution cycle where the instance of operation op of ith iteration is issued. If the following constraints are satisfied:
1. Resource constraints: In each cycle, the same resource (or resource stage for pipelined resource) can not be used more than one time.
2. Dependence constraint: $\forall e = (op_i, op_j) \in E,\ \forall k \in N,\ \sigma(op_i, k) + \delta(e) \leq \sigma(op_j, k + \lambda(e))$.
3. Cyclicity constraint: σ must be expressible in the form of a loop, that is,
$\exists \alpha, \beta \in N,\ \forall op \in O,\ \forall i \in N\ and\ i > 0,\ \sigma(op, i) = \sigma(op, (i\ mod\ \beta)) + \alpha * (\lceil i/\beta \rceil - 1)$.
then we say that σ is a valid loop schedule for the given loop. α/β is called the average initiation interval of σ. The goal of local software pipelining is to find a valid loop schedule with the minimum average initiation interval.

If β is greater than 1, we say that σ is a software pipelining with loop unrolling. Sometimes, loop unrolling can improve the time efficiency of local software pipelining but degrades its space efficiency. We can consider a local software pipelining technique with loop unrolling in such a way that we first unroll the loop and then software pipeline the unrolled loop without any unrolling [Su91a, Eis92]. Thus, in this paper, we focus on the case without loop unrolling.

Now let us see an example of local software pipelining technique shown in Fig.2.1. Assume the machine has one adder and one multiplier which can operate in parallel. (a) is a loop body and its LDDG is shown in (b). (c) gives the local software pipelining result in which $\sigma(op_i, j) = i + 2 * j$. Note that the new loop body is gotten by an instruction-level loop transformation from the old loop body (in which we suppose that an instruction is an operation), and in the new loop body, there may be more than one operation of the old loop body in each cycle. Thus, we can consider local software pipelining in such a way that we in fact transform a vector of one-dimension (the old loop body), whose each element is an

operation, into a matrix of two-dimensions (the new loop body) whose each element is an operation or a group of operations. In this matrix, rows denotes the cycles and columns denotes the iterations. For the above example, the transformation is done from the vector $(1, 2, 3, 4)$ to the matrix as follows:

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

We will define the row-number, rn, and the column-number, cn, for each element in the matrix such that $rn(op_1) = 1$, $rn(op_2) = 2$, $rn(op_3) = 1$, $rn(op_4) = 2$, $cn(op_1) = 1$, $cn(op_2) = 1$, $cn(op_3) = 2$, $cn(op_4) = 2$. The row-number and the column-number are two very important notations. If two operations have the same row-number, then the two operations will be executed in the same cycle so they can not use the same resource (or resource stage). If two operations have the same column-number, then the instances of the two operations come from the same iteration. In this case, we must carefully consider whether they have loop-independent dependence. If they have, we must satisfy it.

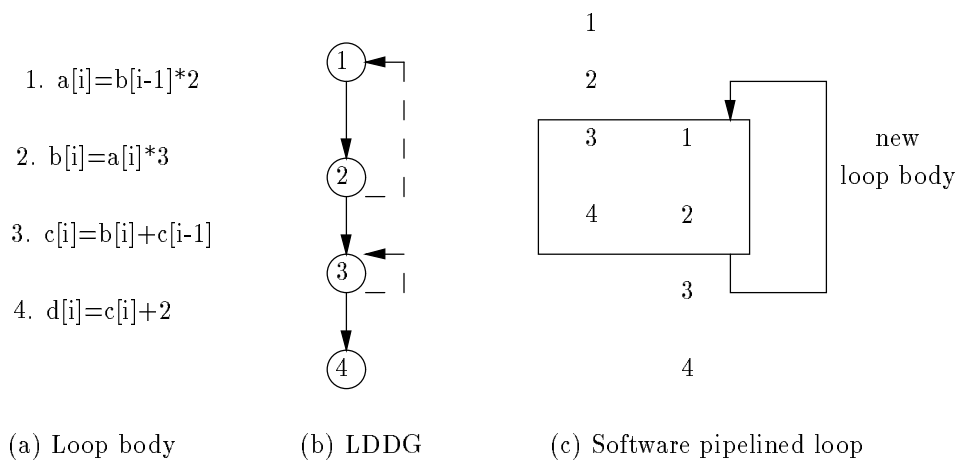


Fig.2.1 An example of software pipelining

Definition 2.1 (row-number and column-number) Let σ be a valid loop schedule and h be the length of the software pipelined loop, in the software pipelined loop body (the matrix), for each operation op, we define its row-number, rn, and its column-number, cn, such that $\sigma(op, 1) = rn(op) + h * (cn(op) - 1)$ *and* $\sigma(op, i) = \sigma(op, 1) + h * (i - 1)$.

The two equations in Definition 2.1 give the relations among cycle, row-number, iteration and column-number.

From the above discussion, the resource constraint, dependence constraint and cyclicity constraint can be expressed in the terms of row-number and column-number as follows.

1. Resource constraint: If two operations have the same row-number, they can not use the same resource (or resource stage).

2. Dependence constraint:
$\forall e = (op_i, op_j) \in E, \quad rn(op_j) - rn(op_i) + h * (\lambda(e) + cn(op_j) - cn(op_i)) \geq \delta(e)$.
3. Cyclicity constraint: The transformation itself satisfies cyclicity constraint as the matrix is actually the new loop body.

So a valid loop schedule σ can be denoted as (rn, cn, h).

2.2 Analysis of the Constraints upon Row-number and Column-number

In this subsection, we further analyze the influence of dependences on the row-number and the column-number of each operation and present two sufficient and necessary conditions for satisfying all dependences among operations.

First we analyze the influence of the determination of row-number on the determination of column-number. Let $e = (op_i, op_j)$ be a dependence edge of LDDG and h be the length of the new loop body. In order to satisfy this dependence, the row-number and the column-number of op_i and op_j must satisfy

$$rn(op_j) - rn(op_i) + h * (\lambda(e) + cn(op_j) - cn(op_i)) \geq \delta(e).$$

It is easy to see that $rn(op_j) - rn(op_i) < h$, so we can decompose the above equation into two cases:
1. if $rn(op_j) - rn(op_i) \geq \delta(e)$, then $cn(op_j) - cn(op_i) \geq -\lambda(e)$.
2. if $rn(op_j) - rn(op_i) < \delta(e)$, then
$cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil(\delta(e) + rn(op_i) - rn(op_j))/h\rceil$.

The above two cases imply that the determination of the row-number may introduce some new constraints on the determination of the column-number. More precisely, for two operations, op_i and op_j, if there is a dependence from op_i to op_j, then the relation between $cn(op_j)$ and $cn(op_i)$ is dependent on the relation between $rn(op_j)$ and $rn(op_i)$. Thus, we introduce a new notation, called iteration-distance (denoted as τ), on each dependence edge of LDDG.

Definition 2.2 Let $LDDG = (O, E, \lambda, \delta)$ be a loop DDG of the given loop. We define $LDDG^\tau$ to be (O, E, τ) in which the weight of each edge is τ, instead of (λ, δ). After determining the row-number of each operation, we can exactly define the value of τ for each edge $e = (op_i, op_j)$ as follows:
(1) if $rn(op_j) - rn(op_i) \geq \delta(e)$, then $\tau(e) = -\lambda(e)$.
(2) if $rn(op_j) - rn(op_i) < \delta(e)$, then
$\tau(e) = -\lambda(e) + \lceil(\delta(e) + rn(op_i) - rn(op_j))/h\rceil$.

Definition 2.3 A valid schedule σ^τ is a schedule[1] of $LDDG^\tau$ such that for each edge $e = (op_i, op_j) \in LDDG^\tau$, $\sigma^\tau(op_j) - \sigma^\tau(op_i) \geq \tau(e)$.

[1] A schedule is different from a loop schedule. The former is defined as a mapping function from O to N, and the latter is defined as a mapping function from $O \times N$ to N.

Theorem 2.1 For a given $LDDG^\tau$, there exists at least one valid schedule if and only if for each cycle C of the $LDDG^\tau$,

$$\sum_{\forall e \in C} \tau(e) \leq 0$$

Proof: if: We first construct a schedule of σ^τ and then prove that it is a valid schedule of $LDDG^\tau$.

We introduce a new node, called source (denoted as s), into $LDDG^\tau$. From the source to each operation of $LDDG^\tau$, we connect a new edge with the iteration-distance of 0. Let $\sigma^\tau(s) = 1$.

We use a longest path algorithm to compute the longest distances from s to each operation op_i, denoted as $ld(s, op_i)$. Let $\sigma^\tau(op_i) = ld(s, op_i)$.

Now we prove σ^τ is a valid schedule of $LDDG^\tau$. For any edge of $LDDG^\tau$, $e = (op_i, op_j)$, we only need to show $\sigma^\tau(op_j) - \sigma^\tau(op_i) \geq \tau(e)$.

case 1: There is not any path from op_j to op_i, so $ld(s, op_j) \geq ld(s, op_i) + \tau(e)$. That is $\sigma^\tau(op_j) - \sigma^\tau(op_i) \geq \tau(e)$.

case 2: There is at least one path from op_j to op_i.
case 2.1: If any longest path from s to op_i does not pass op_j, then $ld(s, op_j) \geq ld(s, op_i) + \tau(e)$. That is, $\sigma^\tau(op_j) - \sigma^\tau(op_i) \geq \tau(e)$.
case 2.2: If there is a longest path from s to op_i which pass op_j, say P. Assume $P = P_{s \to op_j} P_{op_j \to op_i}$ where $P_{s \to op_j}$ denotes the path from s to op_j and $P_{op_j \to op_i}$ the path from op_j to op_i. Let

$$\tau(P_{op_j \to op_i}) = \sum_{\forall e \in P_{op_j \to op_i}} \tau(e)$$

Note that $P_{s \to op_j}$ is a longest path from s to op_j and $P_{op_j \to op_i} e$ is a cycle, we have

$$\tau(P_{op_j \to op_i}) + \tau(e) \leq 0$$

Replacing $\tau(P_{op_j \to op_i})$ with $ld(s, op_i) - ld(s, op_j)$

$$ld(s, op_i) - ld(s, op_j) + \tau(e) \leq 0$$

That is, $\sigma^\tau(op_j) - \sigma^\tau(op_i) \geq \tau(e)$.

Therefore, we prove that σ^τ is a valid schedule of $LDDG^\tau$.

only if: Let C be a cycle and $C = op_1 e_{12} op_2 e_{23} op_3 \cdots op_k e_{k1}$. Assume there is a valid schedule σ of the $LDDG^\tau$, then we have

$$\begin{aligned}
\sigma(op_2) - \sigma(op_1) &\geq \tau(e_{12}) \\
\sigma(op_3) - \sigma(op_2) &\geq \tau(e_{23}) \\
&\vdots \\
\sigma(op_1) - \sigma(op_k) &\geq \tau(e_{k1})
\end{aligned}$$

Thus, $0 \geq \tau(e_{12}) + \cdots + \tau(e_{k1})$
That is,

$$0 \geq \sum_{\forall e \in C} \tau(e)$$

□

Corollary 2.1 For a $LDDG^{\tau}$ without cycle, we can always find a valid schedule.

By the similar analysis, we have the following theorem to describe the influence of the determination of column-number on the determination of row-number due to dependences.

Theorem 2.2 Let (O, E, λ, δ) be a LDDG of the given loop and suppose that we have determined the column-number of each operation. We can find a row-number for each operation such that

$$\forall e = (op_i, op_j) \in E,\ rn(op_j) - rn(op_i) \geq \delta(e) - h * (\lambda(e) + cn(op_j) - cn(op_i))$$

if and only if $cn(op_j) - cn(op_i) \geq -\lambda(e)$, where h is the length of the new loop body.

Proof: if: Let $C = op_1 e_{12} op_2 e_{23} op_3 \ \cdots \ op_k e_{k1}$ be a cycle of $LDDG$, then

$$\begin{aligned} cn(op_2) - cn(op_1) &\geq -\lambda(e_{12}) \\ cn(op_3) - cn(op_2) &\geq -\lambda(e_{23}) \\ &\vdots \\ cn(op_1) - cn(op_k) &\geq -\lambda(e_{k1}) \end{aligned}$$

Note that $\lambda(e_{i,(i+1)mod k}) \geq 0$, for $i = 1, 2, 3, \ldots, k$ and $\lambda(e_{12}) + \lambda(e_{23}) + \cdots + \lambda(e_{k1}) > 0$. Thus, there must be at least one edge, say e_{ij}, such that

$$cn(op_j) - cn(op_i) > -\lambda(e_{ij})$$

So we have actually proved that for any cycle of $LDDG$ there must be at least one edge $e = (op_i, op_j)$ such that $cn(op_j) - cn(op_i) + \lambda(e) > 0$.

Now we construct a graph, $LDDG^{cr} = (O, E_s, \delta)$ in which $E_s = E - E'$, where $E' = \{e | e = (op_i, op_j) \in E \ and \ cn(op_j) - cn(op_i) + \lambda(e) > 0\}$. The weight of each edge, e, of E_s is $\delta(e)$. In $LDDG^{cr}$, there is not any cycle, thus we can use a list scheduler to find a row-number for each operation.

If we choose h such that

$$h = max(\max_{\forall op_i \in O} rn(op_i), \max_{\forall e = (op_i, op_j) \in E'} \lceil \frac{rn(op_j) - rn(op_i) - \delta(e)}{\lambda(e) + cn(op_j) - cn(op_i)} \rceil$$

Then, for any $e = (op_i, op_j) \in E$, we have

$$rn(op_j) - rn(op_i) \geq \delta(e) - h * (\lambda(e) + cn(op_j) - cn(op_i))$$

only if: If $rn(op_j) - rn(op_i) \geq \delta(e) - h * (\lambda(e) + cn(op_j) - cn(op_i))$, then

$$h - 1 \geq \delta(e) - h * (\lambda(e) + cn(op_j) - cn(op_i))$$

Note that $\delta(e) \geq 0$, so we have

$$\lambda(e) + cn(op_j) - cn(op_i)) \geq 0$$

That is, $cn(op_j) - cn(op_i)) \geq -\lambda(e)$.

□

2.3 Our New Approach

The basic idea of DEcomposed Software Pipelining (denoted as DESP) approach is very simple. We pipeline a loop by two steps: one to determine the row-numbers and another to determine the column-numbers. For an operation, if its row-number and column-number have been determined, its position in the new loop body can be fixed. In other words, if the row-numbers and the column-numbers of all operations have been determined, the new loop body can be found. We can first determine the row-numbers of all operations and then determine the column-numbers of all operations. We can also first determine the column-numbers of all operations and then determine the row-numbers of all operations.

- **First Row-number Last Column-number(FRLC):**

In this case, we first determine the row-numbers of all operations. The problem can be described as follows:

Find a row-number for each operation, such that h (the length of the new loop body) is minimum and the following constraints can be satisfied:
(1) For any cycle, C, of $LDDG^{\tau}$,

$$\sum_{\forall e \in C} \tau(e) \leq 0.$$

(2) If two operations have the same row-numbers, then they cannot use the same resource (or resource stage).

From Corollary 2.1, if the given LDDG is acyclic, we can directly schedule the operations only under constraint (2). However, if the LDDG is cyclic, we must satisfy constraint (1) as well as constraint (2). Since all cycles are included in the strongly connected components (denoted as SCCs) of the LDDG, in order to reduce the computation complexity of the algorithm, we first treat with these SCCs before we determine the row-numbers.

Our basic ideas are given as follows:
1) Find all SCCs from the given LDDG;
2) Remove some edges from these SCCs such that: (a) The modified SCCs become acyclic and (b) in the absence of resource constraint, we can use list scheduling to get the row-numbers for the operations of these SCCs with the minimum h which satisfy constraint (1);

3) Remove all edges which are not included in these SCCs;
4) Use list scheduling to determine the row-numbers under the constraints of modified LDDG and resources.

The detailed algorithms will be presented in the next section.

• **First Column-number Last Row-number(FCLR):**

In this case, we first determine the column-numbers of all operations. We can easily find the lower bound of the length of the new loop body, h_{lb}: $h_{lb} = max(h_0, h_{cs})$. where h_{cs} is the number of usages of the critical resource and h_0 is defined as

$$h_0 = \max_{\forall C \in LDDG} \lceil \delta(C)/\lambda(C) \rceil, \quad \delta(C) = \sum_{\forall e \in C} \delta(e) \ and \ \lambda(C) = \sum_{\forall e \in C} \lambda(e)$$

C denotes a cycle.

After determining the column-numbers, we hope that the given LDDG can be transformed into an acyclic one (denoted as $LDDG^{cr}$). The transformation is actually to remove those edges each of which satisfies $cn(op_j) - cn(op_i) + \lambda(e) > 0$. It is easy to see that if the LDDG is acyclic we can get the column-numbers such that all edges satisfy $cn(op_j) - cn(op_i) + \lambda(e) > 0$ so the $LDDG^{cr}$ includes no dependence edges. If the LDDG is cyclic, we can not get the column-numbers such that all edges satisfy $cn(op_j) - cn(op_i) + \lambda(e) > 0$. There must be some edges which $cn(op_j) - cn(op_i) + \lambda(e) = 0$. So the $LDDG^{cr}$ includes some dependence edges.

Thus, we can describe the problem as follows:

Find a column-number for each operation such that
(1) $LDDG^{cr}$ is acyclic;
(2) For each edge $e = (op_i, op_j)$, $cn(op_j) - cn(op_i) \geq -\lambda(e)$.
(3) The height of $LDDG^{cr}$ is minimum.

To solve the above problem, we present the following heuristics:
1) Find SCCs from the given LDDG;
2) Generate a local software pipelining result $\sigma_0 = (rn_0, cn_0, h_0)$ for the SCCs with initiation interval h_0. For any edge of SCCs, $e = (op_i, op_j)$, if $rn_0(op_j) - rn_0(op_i) \geq \delta(e)$, then $cn(op_j) - cn(op_i) + \lambda(e) = 0$; if $rn_0(op_j) - rn_0(op_i) < \delta(e)$, then $cn(op_j) - cn(op_i) + \lambda(e) = \lceil (\delta(e) + rn_0(op_i) - rn_0(op_j))/h_0 \rceil$.
3) For any edge $e = (op_i, op_j)$ which is not included in SCCs,
$cn(op_j) - cn(op_i) + \lambda(e) = 1$.
4) Construct $LDDG^{cn} = (O, E, cn)$ in which the weight of each edge, $e = (op_i, op_j)$ is $cn(e)$, $cn(e) = cn(op_j) - cn(op_i)$.
5) For any cycle C of $LDDG^{cn}$, $\sum_{\forall e \in C} cn(e) \leq 0$, so we can determine the column-numbers.

The detailed algorithms will be presented in the next section.

3 Local DESP Algorithms

In this section, we give the descriptions of FRLC algorithm and FCLR algorithm.

3.1 FRLC Algorithm

Definition 3.1 For a given LDDG, we define $LDDG_{SCC}$ as a subgraph of LDDG which consists of all strongly connected components of LDDG.

A simple algorithm to find $LDDG_{SCC}$ from LDDG is given as follows:

Algorithm 3.1
1. Let $LDDG'_{SCC} = LDDG$;
2. Compute the in-degrees and the out-degrees for all operations of LDDG;
3. if (there is an operation whose in-degree or out-degree is zero) then remove it and all edges connected to it from $LDDG'_{SCC}$; goto 2; else goto 4;
4. Find all edges and nodes not included in any strongly connected component of $LDDG'_{SCC}$; the resulting graph is denoted as $LDDG_{SCC}$;
$\diamond$

The following algorithm can generate a software pipelining result with initiation interval h_0 for $LDDG_{SCC}$.

Algorithm 3.2
1. Compute h_0;
2. Add a weight d on each edge of $LDDG_{SCC}$, $d(e) = \delta(e) - \lambda(e) * h_0$;
3. Add a source, s, to $LDDG_{SCC}$ and connect a new edge with weight 0 from s to each operation of $LDDG_{SCC}$, the resulted $LDDG_{SCC}$ is denoted as $LDDG'_{SCC}$;
4. Using the finding longest path algorithm, we can get a valid loop schedule of $LDDG_{SCC}$, denoted as $\sigma^d = (rn^d, cn^d, h_0)$.
$\diamond$

Now we can use algorithm 3.1 and 3.2 to describe an algorithm to modify $LDDG_{SCC}$ for FRLC algorithm.

Algorithm 3.3 (This algorithm is derived from [Gas92])
1. Use algorithm 3.1 to find $LDDG_{SCC}$; If $LDDG_{SCC} = emptyset$, then return;
2. Use algorithm 3.2 to generate σ^d;
3. For each edge $e = (op_i, op_j)$ of $LDDG_{SCC}$, if $rn^d(op_i) + \delta(e) > rn^d(op_j)$, then remove e from $LDDG_{SCC}$. We denote the modified $LDDG_{SCC}$ as $LDDG_{SCC-m}$.
$\diamond$

We describe the FRLC algorithm as follows:

FRLC Algorithm:
1. Find $LDDG_{SCC}$ from LDDG by Algorithm 3.1;
2. Modify $LDDG_{SCC}$ by Algorithm 3.2 and 3.3 and get $LDDG_{SCC-m}$;
3. Remove all edges which are not included in $LDDG_{SCC}$. We denote the set of these edges as E_r, the edge-set of $LDDG_{SCC}$ as E_{SCC}, the edge-set of $LDDG_{SCC-m}$ as E_{SCC-m}. Note that LDDG=(O, E, λ, δ) so the modified LDDG, $LDDG_m = (O, E - E_r - E_{SCC} + E_{SCC-m}, \delta)$.
4. Under the constraints of resources and $LDDG_m$, use list scheduling technique to generate

a valid schedule, σ.
5. According to σ, determine the row-numbers and the length of the new loop body, h. $rn(op_i) = \sigma(op_i)$.
6. Construct $LDDG^{\tau}$.
7. Generate a valid schedule of $LDDG^{\tau}$, σ^{τ}.
8. According to σ^{τ}, determine the column-numbers, $cn(op_i) = \sigma^{\tau}(op_i)$.
9. In terms of the row-numbers and the column-numbers, construct the new loop body.
10. Construct the prelude and the postlude.
$\diamond$

The correctness of the FRLC algorithm is guaranteed by Theorem 2.1 and the following Theorem 3.1, 3.2 and 3.3.

Theorem 3.1 For any cycle, C, of $LDDG_{SCC}$,

$$\sum_{\forall e \in C} d(e) \leq 0$$

Proof: For any cycle C, we have

$$h_0 \geq \delta(C)/\lambda(C), \quad \delta(C) = \sum_{\forall e \in C} \delta(e) \; and \; \lambda(C) = \sum_{\forall e \in C} \lambda(e).$$

Thus,

$$\sum_{\forall e \in C} \delta(e) - h_0 * \sum_{\forall e \in C} \lambda(e) \leq 0$$

That is

$$\sum_{\forall e \in C} d(e) \leq 0$$

$\square$

Theorem 3.2 If for any cycle C of LDDG, $\delta(C) > 0$, then $LDDG_{SCC-m}$ is acyclic.

Proof: We need to show that, for any cycle C, there must be at least an edge, $e = (op_i, op_j)$, such that

$$rn^d(op_i) + \delta(e) > rn^d(op_j)$$

Assume there is a cycle C, for any edge $e = (op_i, op_j)$

$$rn^d(op_i) + \delta(e) \leq rn^d(op_j)$$

Thus,

$$\sum_{\forall e \in C} \delta(e) \leq 0$$

That is , $\delta(C) \leq 0$.

Therefore, $LDDG_{SCC}$ is acyclic.

$\square$

Theorem 3.3 For each cycle $C \in LDDG^{\tau}$,

$$\sum_{\forall e \in C} \tau(e) \leq 0.$$

Proof: Let C be a cycle of $LDDG^{\tau}$ and $e_1, e_2, \ldots, e_i, e_{i+1}, \ldots, e_k$ be all edges of C where $e_{i+1}, \ldots, e_k$ remain and $e_1, e_2, \ldots, e_i$ are removed in $LDDG_m$.

Let $\sigma = (rn_0, cn_0, h_0)$ be the loop schedule by which $LDDG$ is modified and $LDDG_m$ is gotten. From Definition 2.2 and Theorem 2.1, we have

$$-\sum_{\forall e \in \{e_{i+1}, \ldots, e_k\}} \lambda(e) + \sum_{\forall e = (op, op') \in \{e_1, \ldots, e_i\}} (-\lambda(e) + \lceil (\delta(e) + rn_0(op) - rn_0(op'))/h_0 \rceil) \leq 0.$$

That is,

$$\sum_{\forall e = (op, op') \in \{e_1, \ldots, e_i\}} \lceil (\delta(e) + rn_0(op) - rn_0(op'))/h_0 \rceil \leq \sum_{\forall e \in C} \lambda(e).$$

Note that $\delta(e) + rn_0(op) - rn_0(op') > 0$, thus

$$i \leq \sum_{\forall e = (op, op') \in \{e_1, \ldots, e_i\}} \lceil (\delta(e) + rn_0(op) - rn_0(op'))/h_0 \rceil \leq \sum_{\forall e \in C} \lambda(e)$$

Now, for the new loop body, we can choose its length, h, such that $\forall e = (op, op') \in \{e_1, \ldots, e_i\}$, $(\delta(e) + rn(op) - rn(op'))/h \leq 1$, that means,

$$\sum_{\forall e = (op, op') \in \{e_1, \ldots, e_i\}} \lceil (\delta(e) + rn(op) - rn(op'))/h \rceil = i.$$

So,

$$\sum_{\forall e \in C} \tau(e) = -\sum_{\forall e \in \{e_{i+1}, \ldots, e_k\}} \lambda(e) + \sum_{\forall e = (op, op') \in \{e_1, \ldots, e_i\}} (-\lambda(e) + \lceil (\delta(e) + rn(op) - rn(op'))/h \rceil)$$

That is,

$$\sum_{\forall e \in C} \tau(e) = -\sum_{\forall e \in C} \lambda(e) + i \leq 0.$$

Finally, we must point out that we can always get a small h since the list scheduling technique is used to generate the new loop body.

□

Fig.3.2 gives a complete example for illustrating the FRLC algorithm. The original loop is given in Fig.3.1. We assume that the machine has one adder, one multiplier and one load/store unit which can operate in parallel and whose latencies all are one cycle.

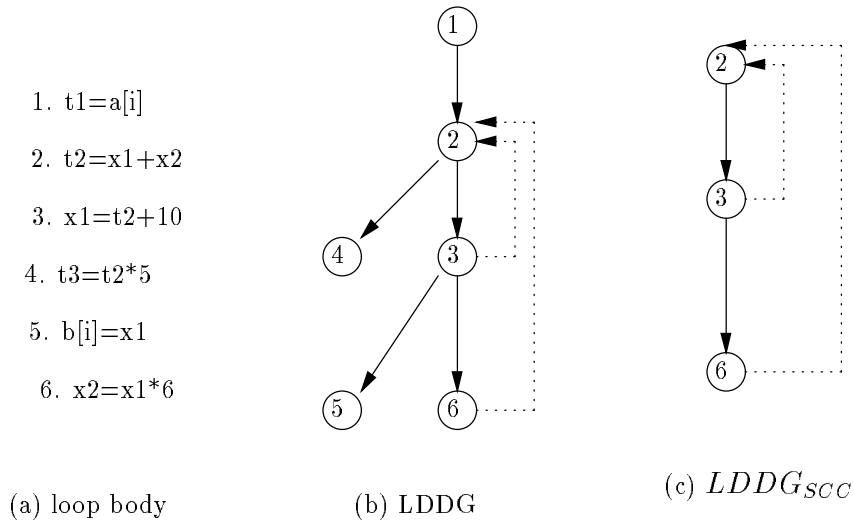


Fig. 3.1 A loop

3.2 FCLR Algorithm

We describe the FCLR algorithm as follows.

FCLR Algorithm:
1. Find $LDDG_{SCC}$ from LDDG by Algorithm 3.1.
2. Under unlimited resources, generate an optimal software pipelining result $\sigma_0 = (rn_0, cn_0, h_0)$ for $LDDG_{SCC}$ by Algorithm 3.2.
3. For each edge $e = (op_i, op_j) \in E$, if $rn_0(op_j) - rn_0(op_i) \geq \delta(e)$, then $cn(e) = -\lambda(e)$; if $rn_0(op_j) - rn_0(op_i) < \delta(e)$, then $cn(e) = -\lambda(e) + \lceil(\delta(e) + rn_0(op_i) - rn_0(op_j))/h_0\rceil$. For each edge $e = (op_i, op_j) \in E - E_{SCC}$, $cn(e) = -\lambda(e) + 1$.
4. Construct $LDDG^{cn}$.
5. Generate a valid schedule of $LDDG^{cn}$, σ^{cn}, by the finding longest path algorithm.
6. For each operation op of LDDG, $cn(op) = \sigma^{cn}(op)$;
7. For each edge $e = (op_i, op_j) \in E_{SCC}$, if $cn(op_j) - cn(op_i) > -\lambda(e)$, remove it from E_{SCC}. The resulted E_{SCC} is denoted as E'_{SCC}.
8. For each edge $e \in E - E_{SCC}$, remove it. So we get $LDDG^{cr} = (O, E'_{SCC}, \delta_{new})$.
9. Under the constraints of resources and $LDDG^{cr}$, use the list scheduling technique to generate a schedule of $LDDG^{cr}$, σ^{cr}.
10. For each operation op of LDDG, $rn(op) = \sigma^{cr}(op)$.
11. In terms of the row-numbers and the column-numbers, construct the new loop body.
12. Construct the prelude and the postlude.
$\diamond$

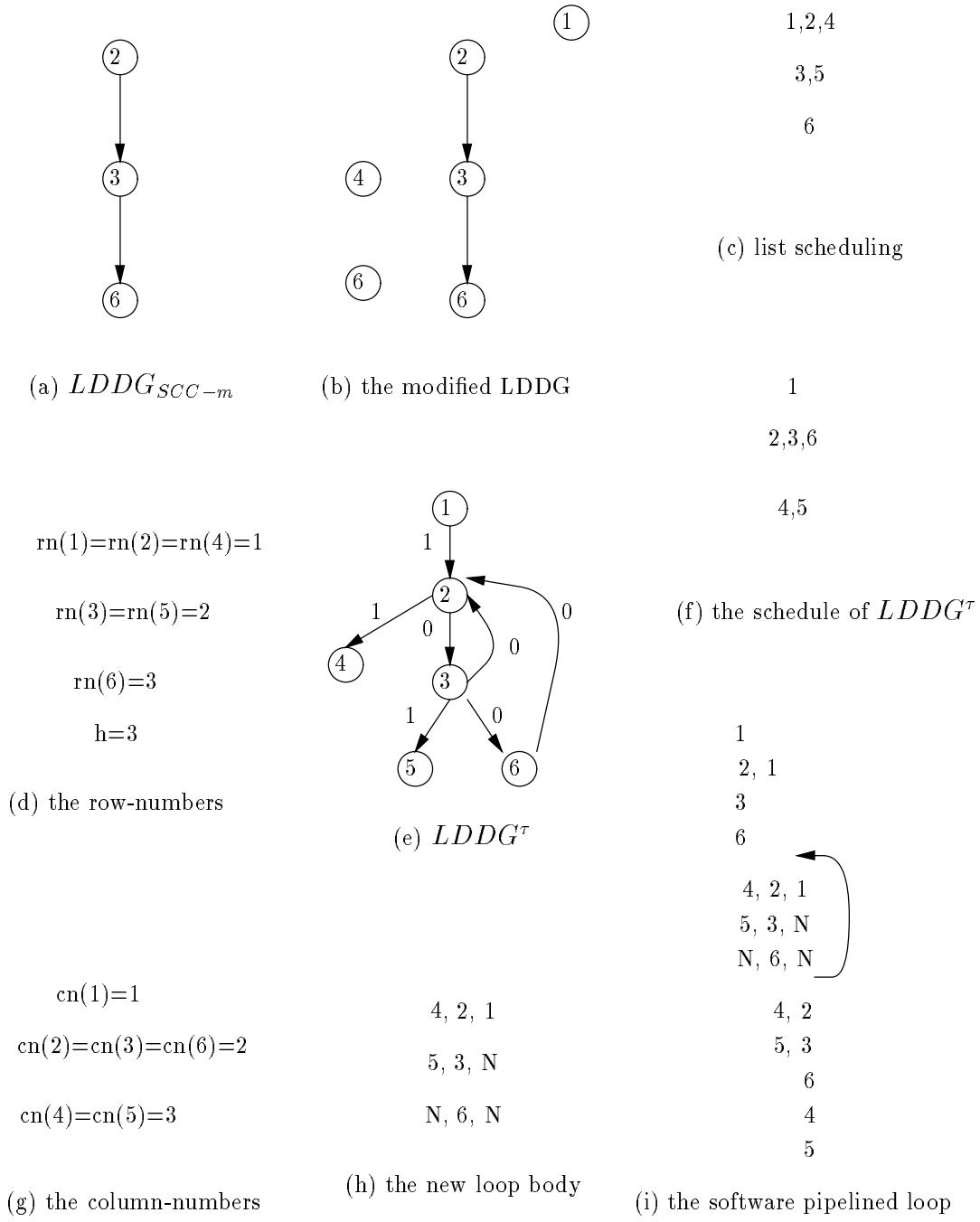


Fig. 3.2 Optimizing the loop of Fig.3.1 by the FRLC algorithm

The correctness of FCLR algorithm can be guaranteed by Theorem 2.2 and the following two theorems.

Theorem 3.4 For each cycle C of $LDDG^{cn}$,

$$\sum_{\forall e \in C} cn(e) \leq 0.$$

Proof: Note that the definition of $cn(e)$ in Subsection 2.3, for the optimal software pipelining $\sigma_0 = (rn_0, cn_0, h_0)$ of $LDDG_{SCC}$ in the absence of resource constraint, if $rn_0(op_j) - rn_0(op_i) \geq \delta(e)$, $cn(e) = -\lambda(e)$; if $rn_0(op_j) - rn_0(op_i) < \delta(e)$, $cn(e) = -\lambda(e) + \lceil(\delta(e) + rn_0(op_i) - rn_0(op_j))/h_0\rceil$.

Since σ_0 is a valid loop schedule, we have

$$\forall e = (op_i, op_j) \in LDDG_{SCC}, \quad rn_0(op_j) - rn_0(op_i) \geq \delta(e) - h_0 * (\lambda(e) + cn_0(op_j) - cn_0(op_i)).$$

If $rn_0(op_j) - rn_0(op_i) \geq \delta(e)$, then

$$cn_0(op_j) - cn_0(op_i) \geq -\lambda(e) = cn(e).$$

If $rn_0(op_j) - rn_0(op_i) < \delta(e)$, then

$$cn_0(op_j) - cn_0(op_i) \geq -\lambda(e) + \lceil(\delta(e) + rn_0(op_i) - rn_0(op_j))/h_0\rceil = cn(e).$$

Thus, for each cycle $C \in LDDG_{SCC}$,

$$\sum_{\forall e \in C} cn(e) \leq \sum_{\forall e = (op_i, op_j) \in C} (cn_0(op_j) - cn_0(op_i)) = 0.$$

□

Theorem 3.5 If for any cycle C of LDDG, $\delta(C) > 0$, then $LDDG^{cr}$ is acyclic.

Proof: We only need to show that for any cycle C, there is at least one edge, $e = (op_i, op_j)$, such that $cn(op_j) - cn(op_i) + \lambda(e) > \lceil\delta(e)/h_{lb}\rceil - 1$ (the condition under which an edge can be removed in FCLR, see Subsection 2.3).

Note that the column-numbers are determined by a valid loop schedule $\sigma_0 = (rn_0, cn_0, h_0)$. Since $\delta(C) > 0$, there must be at least an edge $e = (op_i, op_j) \in C$ for any cycle $C \in LDDG_{SCC}$, such that $rn_0(op_j) - rn_0(op_i) < \delta(e)$.

Thus, $cn(op_j) - cn(op_i) = -\lambda(e) + \lceil(\delta(e) + rn_0(op_i) - rn_0(op_j))/h_0\rceil$.

As $|rn_0(op_i) - rn_0(op_j)| < h_0$ *and* $h_0 \leq h_{lb}$, we have

$$cn(op_j) - cn(op_i) > -\lambda(e) + \lceil\delta(e)/h_{lb}\rceil - 1$$

□

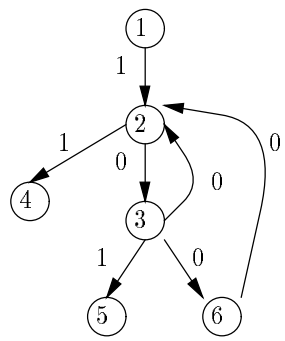
A complete example is given to illustrate FCLR algorithm in Fig.3.3.

2

3

6

(a) the software pipelining result for $LDDG_{SCC}$



(b) $LDDG^{cn}$

1

2,3,6

4,5

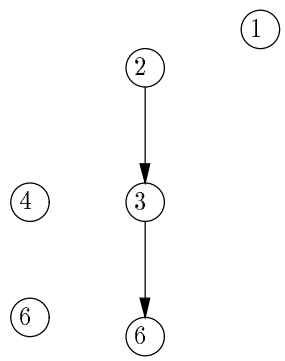
(c) the schedule of $LDDG^{cn}$

cn(1)=1

cn(2)=cn(3)=cn(6)=2

cn(4)=cn(5)=3

(d) the column-numbers



(e) $LDDG^{cr}$

1,2,4

3,5

6

(f) the schedule of $LDDG^{cr}$

rn(1)=rn(2)=rn(4)=1

rn(3)=rn(5)=2

rn(6)=3

h=3

(g) the row-numbers

4, 2, 1

5, 3, N

N, 6, N

(h) the new loop body

1

2, 1

3

6

4, 2, 1

5, 3, N

N, 6, N

4, 2

5, 3

6

4

5

(i) the software pipelined loop

Fig. 3.3 Optimizing the loop of Fig.3.1 by the FCLR algorithm

4 GDESP – An Extension of DESP

In this section, we will extend DESP (on the basis of FRLC) and develop Global DESP (denoted as GDESP) approach to exploit instruction level parallelism for loops with conditional jumps (branches). We first give some useful notations, then extend and redefine the row-number and the column-number, and finally present the basic idea of GDESP and discuss the major problems which we face when we develop GDESP approach. The correctness of GDESP is also theoretically proven.

4.1 Notations

This subsection gives some notations to model a loop with branches for which we must consider control dependences as well as data dependences. Flow graphs are the most common way to represent control dependences in which each node represents a basic block and the directed edge between two basic blocks represents the control dependence between these two basic blocks [Fis81, Ell86, Gas89]. A basic block is a sequence of instructions which has no jumps into the code except at the first instruction and no jumps out except at the end.

Definition 4.1 Let $FG = (BB, CD)$ be the flow graph of a loop, BB denotes the set of nodes and CD the set of edges. If $e = (B_{exit}, B_{entrance})$ is a loop back-edge, then $B_{entrance}$ is the loop-entrance basic block and B_{exit} the loop-exit basic block. A loop execution path is a simple path from $B_{entrance}$ to B_{exit}.

Definition 4.2 (branch data dependence) Branch data dependence is defined from conditional jump operations to other operations. Let basic blocks B_i and B_j be two immediate successors of a conditional jump operation (denoted as cj), p_i and p_j be two simple paths from B_i to B_{exit} and from B_j to B_{exit}, respectively. If data unit (e.g. variable, array element, register, etc.) d is live at the top of p_i, and there is an operation op of B_j which defines d, then there is a branch data dependence from cj to op.

Definition 4.3 We consider three kinds of data dependences: loop-independent data dependences, loop-carried dependences and branch data dependences. There are two non-negative integers $\lambda(e)$ and $\delta(e)$ associated to each data dependence $e = (op_i, op_j)$. $(\lambda(e), \delta(e))$ denotes that op_j can only be executed $\delta(e)$ cycles after op_i of the $\lambda(e)$th previous iteration has started executing.

Definition 4.4 (Global LDDG, denoted as **GLDDG)** Let $FG = (BB, CD)$ be the flow graph of a loop, $P = \{p_1, p_2, \cdots, p_k\}$ be the set of all loop execution paths of FG. For each $p_i \in P$, we can consider p_i as a basic block and construct its loop DDG (denoted as $LDDG(p_i)$). Let $LDDG(p_i) = (O(p_i), E(p_i))$, then GLDDG is defined as

$$(\bigcup_{\forall p_i \in P} O(p_i), \bigcup_{\forall p_i \in P} E(p_i))$$

In the following, we give a method for construction of GLDDG.

1. For each basic block B_i, construct its LDDG;
2. For each loop execution path p_i, (1) construct loop-independent data dependence edges

and loop-carried dependence edges among basic blocks; (2) for each conditional jump operation cj of p_i, let basic block B_j be an immediate successor off p_i (that is, B_j is not on p_i) and p' be a simple path from B_j to B_{exit}, if there is a data unit d being live at the top of p', then we consider that cj references d. Using this view as a basis, we can construct all branch data dependences from cj to other operations on p_i;

3. In terms of the LDDGs of all loop execution paths, we can easily construct the GLDDG of the given loop.

Fig.4.1 gives an example of GLDDG.

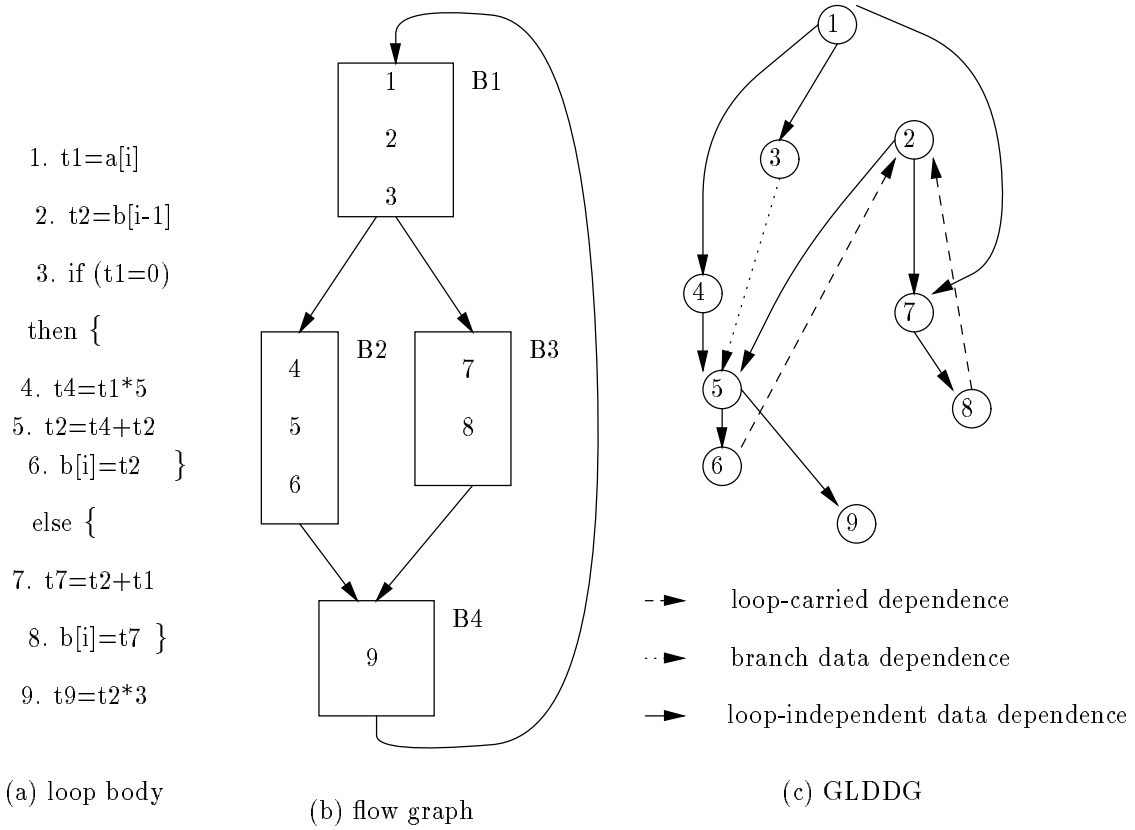


Fig.4.1 An example of GLDDG

4.2 Extended Row-number and Extended Column-number

Up to now, how to formally describe the global software pipelining problem is still an open problem. As there are more than one loop execution paths in the loops with conditional jumps and in compiling time it is not known which path will be chosen to execute in a certain iteration, it is extremely difficult to formally define the loop schedule in the case of global software pipelining.

In this paper, we do not try to formally describe the global software pipelining problem. Our goal is to solve the following three problems:

1. How to construct the global software pipelined loop if we get a correct global software pipelined loop body?

2. What does it mean by the correct global software pipelined loop body?

3. How to get a correct global software pipelined loop body?

To solve the above three problems, we first introduce the notations of extended row-number and extended column-number.

Let us see an example of global software pipelining shown in Fig.4.2. (a) is the original loop and (b) the global software pipelined loop. For more intuitive, we use the parallel program flow graph [Aik87, Gas89, Su91b] to represent the original loop and the global software pipelined loop. We pay more attention to the new loop body (shown in (c)) which is of the following two characteristics:

1. Global software pipelining may cause code motions such that some operations move across conditional jump or join and copies of these operations are generated [Fis81]. Thus, for an operation, there may be more than one its copies in the new loop body. We introduce the term of operation-copy to stand for the operations in the new loop body and use (op, p_i) to denote an operation-copy, where op is the operation and p_i is the loop execution path on which the copy is. For instance, in Fig.4.2(c), 1' can be denoted as (1,B1-B3-B4) and 1 as (1,B1-B2-B4).

2. From the prelude and the postlude shown in Fig.4.2(b), we know that the operations in the new loop body have different iteration numbers. For instance, if 1 (and 1'), 2 (and 2') are ith iteration's operations, then 3,4,5,6 (or 3,7,8) are $(i-1)$th iteration's operations and 9 (and 9'), 10 (and 10') are $(i-2)$th iteration's operations.

Therefore, it is insufficient to represent the global software pipelined loop body only by flow graph. For this reason, we introduce extended row-number and extended column-number.

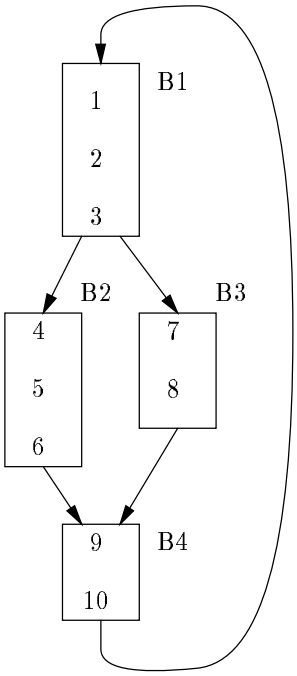
Definition 4.5 (extended row-number and extended column-number) For a global software pipelined loop body, we define extended row-number and extended column-number for each operation-copy:

(1) Let op be an operation-copy, its extended row-number, $ern(op) = (B(op), rn(op))$, where $B(op)$ is the basic block in which op is, and $rn(op)$ is the row-number of op in $B(op)$;

(2) For each operation-copy, op, its extended column-number is denoted as $ecn(op)$. Let op_i, op_j be two operation-copies, op_i is ith iteration's operation and op_j jth iteration's operation, then $ecn(op_j) - ecn(op_i) = i - j$. We also stipulate that

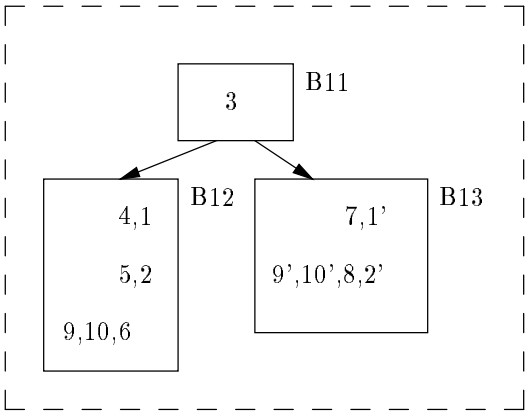
$$\min_{for\ all\ op}(ecn(op)) = 1$$

For the global software pipelined loop body shown in Fig.4.2(c), $ern(3) = (B11, 1)$, $ern(4) = ern(1) = (B12, 1)$, $ern(5) = ern(2) = (B12, 2)$, $ern(9) = ern(10) = ern(6) =$

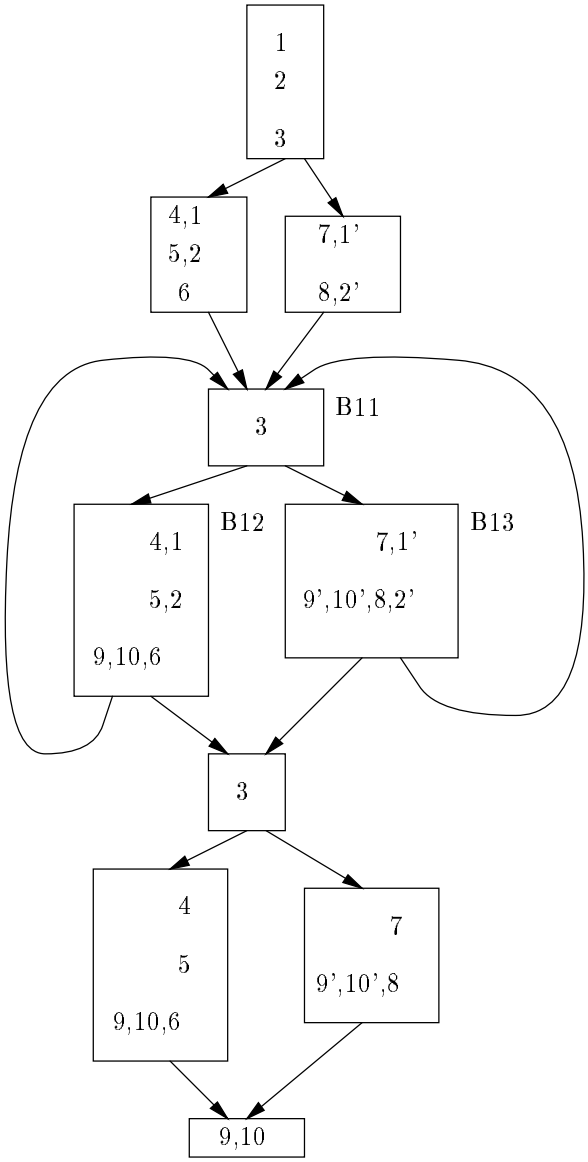


3 is conditional jump operation

(a) the original loop



(c) the global software pipelined loop body
(the new loop body)



(b) the global software pipelined loop

Fig.4.2 An example of global software pipelining

$(B12,3), ern(7) = ern(1') = (B13,1),\ ern(8) = ern(9') = ern(10') = ern(2') = (B13,2);$ $ecn(1) = ecn(2) = ecn(1') = ecn(2') = 1,\ ecn(3) = ecn(4) = ecn(5) = ecn(6) = ecn(7) = ecn(8) = 2,\ ecn(9) = ecn(10) = ecn(9') = ecn(10') = 3.$

In the remainder of this paper, we will use the notations of GLDDG, extended row-number and extended column-number to develop GDESP approach.

4.3 GDESP Approach

In this subsection, we present GDESP approach to exploit instruction level parallelism for loops with branches. Our basic ideas are as follows:

(1) We first use a global code scheduling technique (such as Trace Scheduling [Fis81], Percolation Scheduling [Nic85] or Region Scheduling [Gup87]) to schedule the operations of the original loop body. The scheduled loop body is the global software pipelined loop body (for short, the new loop body). Before scheduling the original loop body, we construct its GLDDG and modify the GLDDG by removing some data dependence edges. Especially, if the GLDDG is acyclic, we can remove all data dependence edges so that the instruction level parallelism can be globally exploited to a much great extent. The ideas behind this will be further discussed in Section 4.3.1. According to the new loop body we can easily find the extended row-numbers.

(2) We then determine the extended column-number for each copy of operation in the new loop body.

(3) Finally, in terms of the extended row-numbers and the extended column-numbers of all operation-copies in the new loop body, we construct the global software pipelined loop.

The major problems which GDESP faces will be discussed in details in the following subsections.

4.3.1 Construction of the New Loop Body

In DESP approach, we first find a subgraph of LDDG which consists of all strongly connected components of LDDG, denoted as $LDDG_{SCC}$; secondly, a software pipelining algorithm without resource constraint is applied to preprocess the $LDDG_{SCC}$ and remove some edges such that $LDDG_{SCC}$ becomes acyclic; thirdly, we remove all edges from LDDG which are not included in $LDDG_{SCC}$; finally we use a list scheduling algorithm to schedule the modified LDDG under resource constraint and get the new loop body. This idea can be almost directly extended to the global case. The only point that needs some attention is that branch data dependencies cannot be removed in the second step. This does not cause much trouble, since they can not belong to a dependence cycle by hypothesis.

Definition 4.6 For a given GLDDG, we define $GLDDG_{SCC}$ as a subgraph of LDDG which consists of all strongly connected components of LDDG.

We can use Algorithm 3.1 to find $GLDDG_{SCC}$ from GLDDG.

Now we consider a $GLDDG_{SCC}$ as a LDDG of a loop without branch and use a local software pipelining algorithm to pipeline it in the absence of resource constraint.

Algorithm 4.1

0. Preprocess the branch data dependence edges in $GLDDG_{SCC}$, such that these edges can not be removed by Algorithm 4.2. Let E_{BDD} be the set of branch data dependences edges;

1. Compute h_0, where

$$h_0 = \max_{\forall C \in LDDG} \lceil \delta(C)/\lambda(C) \rceil, \quad \delta(C) = \sum_{\forall e \in C} \delta(e) \ and \ \lambda(C) = \sum_{\forall e \in C} \lambda(e)$$

C denotes a cycle;

2. Add a weight d on each edge of $GLDDG_{SCC}$, $d(e) = \delta(e) - \lambda(e) * h_0$;

3. Add a source, s, to $GLDDG_{SCC}$ and connect a new edge with weight 0 from s to each operation of $GLDDG_{SCC}$;

4. Let σ be a loop schedule, denoting the local software pipelining result. Let $\sigma(s) = 1$;

5. Use the finding longest path algorithm to compute the longest distances from s to each operation op_i, denoted as $ld(s, op_i)$. Let $\sigma(op_i) = ld(s, op_i)$;

6. In the terms of the notations of row-number and column-number, σ can be expressed as (rn, cn, h_0).

$\diamond$

In the following, we use Algorithm 3.1 and 4.1 to modify $GLDDG_{SCC}$ such that it becomes acyclic.

Algorithm 4.2

1. Use Algorithm 3.1 to find $GLDDG_{SCC}$; If $GLDDG_{SCC} = \emptyset$, then return;

2. Use Algorithm 4.1 to generate σ;

3. For each edge $e = (op_i, op_j)$ of $GLDDG_{SCC}$, if $rn(op_i) + \delta(e) > rn(op_j)$, then remove e from $GLDDG_{SCC}$. We denote the modified $GLDDG_{SCC}$ as $GLDDG_{SCC-m}$.

$\diamond$

Definition 4.7 Let $GLDDG = (O, E)$ and $GLDDG_{SCC-m} = (O_{SCC-m}, E_{SCC-m})$, we define the modified GLDDG as $GLDDG_m = (O, E_{SCC-m} \cup E_{BDD})$.

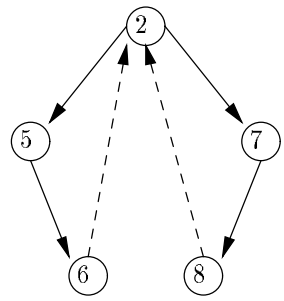
Fig.4.3 gives an example to illustrate how to get the $GLDDG_m$ from the GLDDG shown in Fig.4.1(c) by Algorithm 3.1, 4.1 and 4.2, where we assume that, for any edge e, $\delta(e) = 1$, and for any loop-carried dependence edge e, $\lambda(e) = 1$.

Note that $GLDDG_m$ is acyclic (see Theorem 4.1 in the following), thus, after we get $GLDDG_m$, we can use a global loop-free code scheduling technique (such as Trace Scheduling) to generate the new loop body under the constraints of resources, flow graph and $GLDDG_m$.

Theorem 4.1 If for any cycle C of GLDDG, $\delta(C) = \sum_{\forall e \in C} \delta(e) > 0$, then $GLDDG_m$ is acyclic.

Proof: We need to show that, for any cycle C, there must be at least an edge, $e = (op_i, op_j)$, such that

$$rn^d(op_i) + \delta(e) > rn^d(op_j) \quad (see\ step\ 3\ of\ Algorithm\ 4.2)$$



(a) $GLDDG_{SCC}$ by Algorithm 3.1

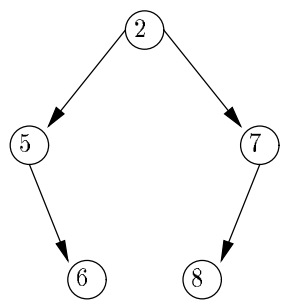
2

5,7

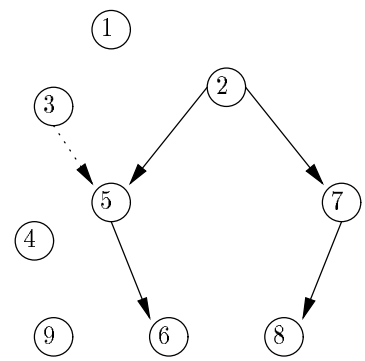
6,8

(b) a loop schedule

by Algorithm 3.2



(c) $GLDDG_{SCC-m}$ by Algorithm 3.3



(d) $GLDDG_m$

Fig4.3 How to get $GLDDG_m$ from GLDDG

Assume there is a cycle C, for any edge $e = (op_i, op_j)$

$$rn^d(op_i) + \delta(e) \leq rn^d(op_j)$$

Thus,

$$\sum_{\forall e \in C} \delta(e) \leq 0$$

That is , $\delta(C) \leq 0$.

Therefore, $GLDDG_m$ is acyclic.

□

4.3.2 Determination of the Extended Column-numbers

In DESP approach, after determining the row-numbers of all operations, we can compute the iteration distance τ between any two operations with data dependence and then construct $LDDG^\tau$ by Definition 2.2. In terms of $LDDG^\tau$, the column-numbers of all operations can

be easily determined by the finding longest path algorithm. This idea can be also directly extended to the global case.

In global case, it is difficult to define the length of the new loop body. In the flow graph of the new loop body, there may be more than one path from the loop entrances to the loop exits, each path is still called loop execution path. The length of a loop execution path is easily computed. For example, in Fig.4.2(c), the length of B11-B12 is 4 and the length of B11-B13 is 3 if the latencies of all operations are one cycle.

Definition 4.8 Let $GLDDG = (O, E, \lambda, \delta)$ be the GLDDG of a given loop and LB_{new} be the new loop body, we define the extended GLDDG of LB_{new}, denoted as $EGLDDG(LB_{new})$, as follows:

(1) Each operation-copy in LB_{new} is represented by a node;

(2) For any two operation-copies in the same loop execution path, say $op_i^{'}$ and $op_j^{'}$, let their original operations be op_i and op_j respectively, if $e = (op_i, op_j)$ is an edge of $GLDDG$, then construct $e^{'} = (op_i^{'}, op_j^{'})$ in $EGLDDG(LB_{new})$ and let $\lambda(e^{'}) = \lambda(e)$, $\delta(e^{'}) = \delta(e)$.

An example of extended GLDDG is given in Fig.4.4, where the original loop body and the new loop body are given in Fig.4.2(a) and (c) respectively. If the GLDDG is given in Fig.4.4(a) then its $EGLDDG(LB_{new})$ is shown in Fig.4.4(b).

Like $LDDG^{\tau}$, we define $EGLDDG^{\tau}(LB_{new})$.

Definition 4.9 let LB_{new} be a new loop body and $EGLDDG(LB_{new}) = (EO, EE, \lambda, \delta)$ be its extended GLDDG. We define $EGLDDG^{\tau}(LB_{new})$ to be $(EO, EE \cup E_c, \tau)$, where

(1) $E_c = \{e | e = (op, op^{'}),\ op, op^{'} \in EO\ and\ op, op^{'}\ are\ two\ copies\ of\ the\ same\ operation\}$;

(2) $\forall e \in E_c, \tau(e) = 0$;

(3) $\forall e = (op_i, op_j) \in EE$,
if $dern(op_j, op_i) \geq \delta(e)$, then $\tau(e) = -\lambda(e)$;
if $dern(op_j, op_i) < \delta(e)$, then $\tau(e) = -\lambda(e) + \lceil(\delta(e) - dern(op_j, op_i))/lsp\rceil$.

In (3), $dern(op_j, op_i)$ denotes the difference between the extended row-numbers of op_j and op_i, whose absolute value is actually the length of the path from op_i to op_j in the new loop body. $dern(op_j, op_i)$ can be easily computed by $ern(op_i)$ and $ern(op_j)$. lsp denotes the length of the shortest path from loop entrance to loop exits in the new loop body.

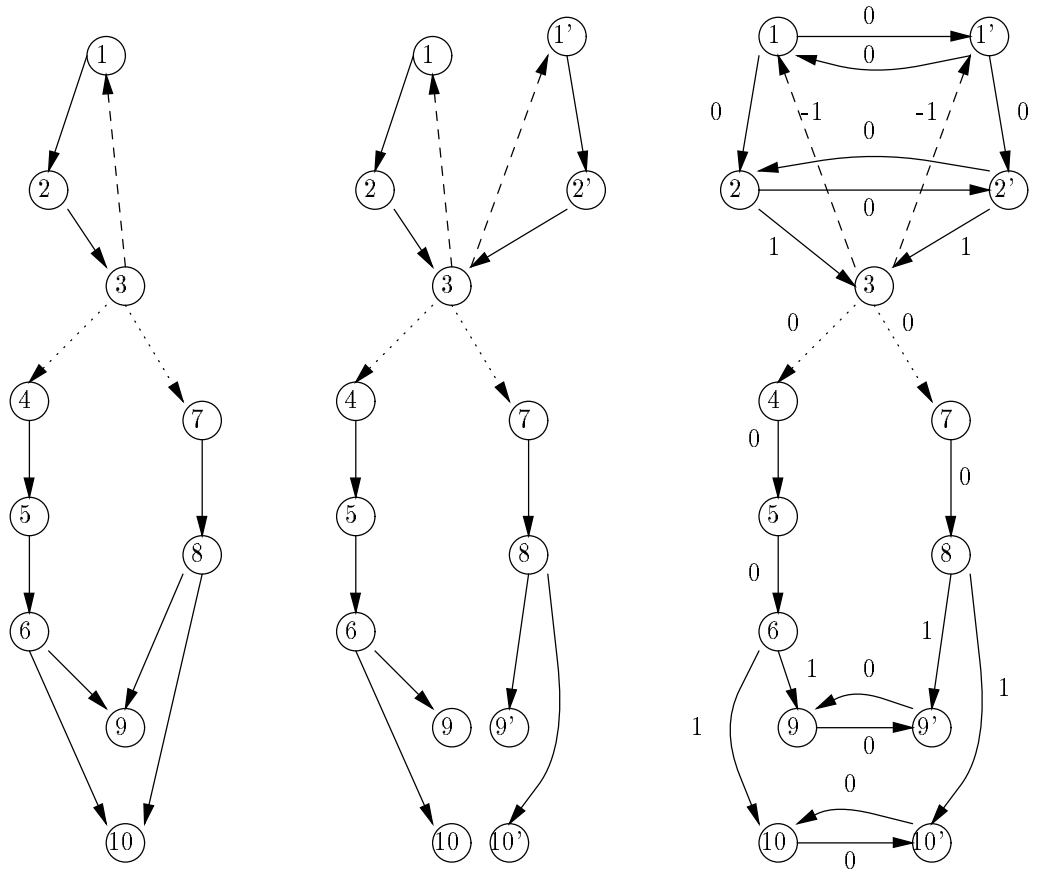
Let us see an example. For the new loop body shown in Fig.4.2(c), assume the latencies of all operations is one cycle, then $lsp = 3$, $dern(3, 2) = -2$, $dern(3, 2') = -2$, $dern(2, 1) = 1$, $dern(2', 1') = 1$, etc. Note that the GLDDG and the $EGLDDG(LB_{new})$ shown in Fig.4.4(a) and (b), we can get $EGLDDG^{\tau}(LB_{new})$ as shown in Fig.4.4(c).

Now we can compute the extended column-number by the following algorithm.

Algorithm 4.3:

1. Introduce a new node, called source (denoted as s), into $EGLDDG^{\tau}(LB_{new})$, from s to each node of $EGLDDG^{\tau}(LB_{new})$, we connect a new edge on which the value of τ is 0;

2. Let $ecn(s) = 1$;



(a) GLDDG (b) $EGLDDG(LB_{new})$ (c) $EGLDDG^{\tau}(LB_{new})$

Fig.4.4 An example of $EGLDDG$ and $EGLDDG^{\tau}$

3. Use the finding longest path algorithm to compute the longest distance from s to each operation-copy op', denoted as $ld(s, op')$, let $ecn(op') = ld(s, op')$.
$\diamond$

The above algorithm can work only if for each cycle C of $EGLDDG^{\tau}(LB_{new})$, $\sum_{\forall e \in C} \tau(e) \leq 0$. In the following, we will prove that the new loop body generated by the method of Subsection 4.3.1 satisfies this condition.

Theorem 4.2 For each cycle $C \in EGLDDG^{\tau}(LB_{new})$,

$$\sum_{\forall e \in C} \tau(e) \leq 0.$$

Proof: Let C be a cycle of $EGLDDG^{\tau}(LB_{new})$ and $e_1, e_2, \ldots, e_i, e_{i+1}, \ldots, e_k$ be all edges of C where $e_{i+1}, \ldots, e_k$ remain and $e_1, e_2, \ldots, e_i$ are removed in $GLDDG_m$.

Let $\sigma = (rn, cn, h)$ be the local software pipelining result by Algorithm 4.1. From Defi-

nition 2.2 and Theorem 2.1, we have

$$-\sum_{\forall e \in \{e_{i+1},\ldots,e_k\}} \lambda(e) + \sum_{\forall e=(op,op') \in \{e_1,\ldots,e_i\}} (-\lambda(e) + \lceil(\delta(e) + rn(op) - rn(op'))/h\rceil) \leq 0.$$

That is,

$$\sum_{\forall e=(op,op') \in \{e_1,\ldots,e_i\}} \lceil(\delta(e) + rn(op) - rn(op'))/h\rceil \leq \sum_{\forall e \in C} \lambda(e).$$

Note that $\delta(e) + rn(op) - rn(op') > 0$, thus

$$i \leq \sum_{\forall e=(op,op') \in \{e_1,\ldots,e_i\}} \lceil(\delta(e) + rn(op) - rn(op'))/h\rceil \leq \sum_{\forall e \in C} \lambda(e)$$

Now, for the new loop body, we can choose its lsp, such that $\forall e = (op, op') \in \{e_1, \ldots, e_i\}$, $(\delta(e) + desp(op', op))/lsp \leq 1$, that means,

$$\sum_{\forall e=(op,op') \in \{e_1,\ldots,e_i\}} \lceil(\delta(e) + dern(op', op))/lsp\rceil = i.$$

So, by Definition 4.9, we have

$$\sum_{\forall e \in C} \tau(e) = -\sum_{\forall e \in \{e_{i+1},\ldots,e_k\}} \lambda(e) + \sum_{\forall e=(op,op') \in \{e_1,\ldots,e_i\}} (-\lambda(e) + \lceil(\delta(e) + dern(op', op))/lsp\rceil)$$

That is,

$$\sum_{\forall e \in C} \tau(e) = -\sum_{\forall e \in C} \lambda(e) + i \leq 0.$$

Finally, we must point out that we can always get a small lsp since the global loop-free code scheduling technique (such as Trace Scheduling) is used to generate the new loop body. □

4.3.3 Construction of the New Loop

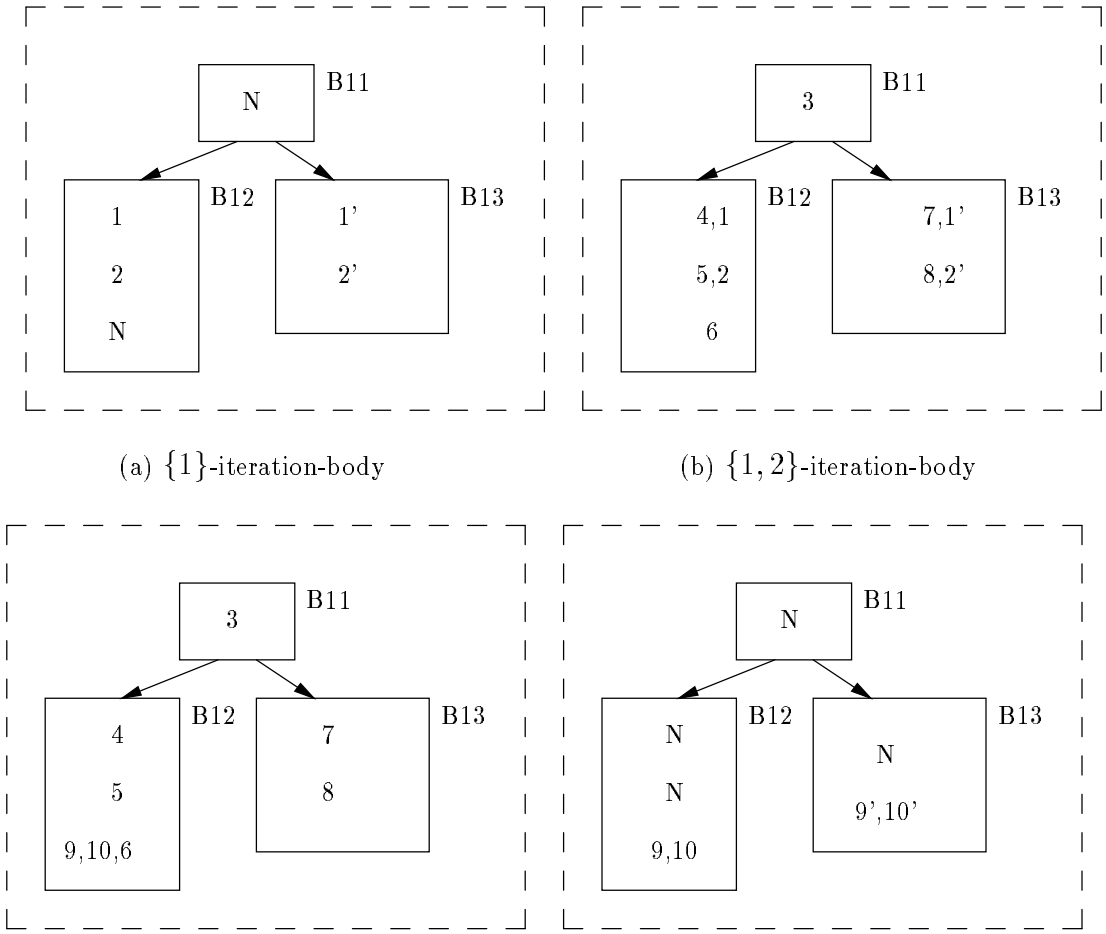
In this subsection, we will discuss how to construct the new loop body in terms of the flow graph of the new loop body, the extended row-numbers and the extended column-numbers.

In this paper, we do not consider the nested loop, so the original loop to be treated with has only one loop entrance. Since we use a global loop-free code scheduling technique to generate the new loop body, it has only one loop entrance (however, it may have more than one loop exits).

We first introduce a new notation, called iteration-body.

Definition 4.10 Let LB_{new} be a new loop body and $ECN = \{1, 2, \ldots, K\}$, where $K = \max_{\forall op_i \in LB_{new}}(ecn(op_i))$, we call LB_{new} ECN-iteration-body. Let ECN_s be a subset of ECN, if we remove all operation-copies whose extended column-numbers are the elements of ECN_s from LB_{new}, then we call the resulting loop body $(ECN - ECN_s)$-iteration-body.

For example, let us see the new loop body shown in Fig.4.2(c), it is $\{1, 2, 3\}$-iteration-body itself, its $\{1\}$-iteration-body, $\{1, 2\}$-iteration-body, $\{2, 3\}$-iteration-body and $\{3\}$-iteration-body are given in Fig.4.5.



(a) $\{1\}$-iteration-body (b) $\{1,2\}$-iteration-body

(c) $\{2,3\}$-iteration-body (d) $\{3\}$-iteration-body

Fig.4.5 Iteration-bodies

Now, using the notation of iteration-body as a basis, we present a method for construction of the new loop as follows:

1. Let LB_{new} be the new loop body and $K = \max_{\forall op_i \in LB_{new}}(ecn(op_i))$.

2. Find $\{1\}$-iteration-body.

3. Assume we have constructed $\{1, 2, \ldots, K'\}$-iteration-body, $K' < K$, we find $\{1, 2, \ldots, K'+1\}$-iteration-body and build an edge from each exit basic block of $\{1, 2, \ldots, K'\}$-iteration-body to the entrance basic block of $\{1, 2, \ldots, K'+1\}$-iteration-body. This work can be done until we finish the construction of $\{1, 2, \ldots, K\}$-iteration-body.

4. Now we finish the construction of the prelude and can construct the new loop body. In fact, $\{1, 2, \ldots, K\}$-iteration-body is the new loop body, so we only construct the loop

back-edges. We build an edge from each exit basic block of $\{1, 2, \ldots, K\}$-iteration-body to its entrance basic block.

5. The construction of the postlude: Assume we have constructed $\{K', K'+1, \ldots, K\}$-iteration-body, $K' < K$, we find $\{K'+1, \ldots, K\}$-iteration-body and build an edge from each exist basic block of $\{K', K'+1, \ldots, K\}$-iteration-body to the entrance basic block of $\{K'+1, \ldots, K\}$-iteration-body. This work can be done until we finish the construction of $\{K\}$-iteration-body.

6. Under the pipeline (timing) constraints, delete empty cycles and merge the abundant operations.

Fig.4.2(b) and (c) may be helpful to understand the above method whose correctness is guaranteed by the following Theorem 4.3.

First, we denote the execution instance of an operation-copy as (op, i), in which op is the operation-copy and i is the iteration number.

Theorem 4.3 In the global software pipelined loop generated by GDESP approach, for any $e = (op_i, op_j) \in EGLDDG$, if (op_i, k) is executed at cycle t_i and $(op_j, k + \lambda(e))$ is executed at cycle t_j, then $t_j - t_i \geq \delta(e)$.

Proof: If e is a branch data dependence edge, then $t_j - t_i \leq \delta(e)$, since we do not remove e from $GLDDG_m$. Thus, we can only consider that e is a loop-carried data dependence edge or a loop-independent data dependence edge. From Definition 4.10, if op_i belongs to $\{l\}$-iteration-body, then op_j belongs to $\{l + (ecn(op_j) - ecn(op_i))\}$-iteration-body. According to the method for construction of the global software pipelined loop, we have

$$t_j - t_i \geq dern(op_j, op_i) + lsp * (\lambda(e) + ecn(op_j) - ecn(op_i))$$

where $dern(op_j, op_i)$ and lsp are defined in Definition 4.9.

Now, according to Definition 4.9 and Algorithm 4.3, there are two cases:

Case 1: if $dern(op_j, op_i) \geq \delta(e)$, then $ecn(op_j) - ecn(op_i) \geq -\lambda(e)$.

Case 2: if $dern(op_j, op_i) < \delta(e)$, then
$ecn(op_j) - ecn(op_i) \geq -\lambda(e) + \lceil(\delta(e0 - dern(op_j, op_i))/lsp\rceil$.

For case 1, we have $\lambda(e) + ecn(op_j) - ecn(op_i) \geq 0$ and $dern(op_j, op_i) \geq \delta(e)$, that is,

$$dern(op_j, op_i) + lsp * (\lambda(e) + ecn(op_j) - ecn(op_i)) \geq \delta(e)$$

so, $t_j - t_i \geq \delta(e)$.

For case 2, we have $\lambda(e) + ecn(op_j) - ecn(op_i) \geq \lceil(\delta(e0 - dern(op_j, op_i))/lsp\rceil$, that is ,

$$dern(op_j, op_i) + lsp * (\lambda(e) + ecn(op_j) - ecn(op_i)) \geq \delta(e)$$

so, $t_j - t_i \geq \delta(e)$.
□

5 A GDESP Algorithm

We give the description of a GDESP algorithm as follows.

GDESP Algorithm:

1. Construct $GLDDG$ of the given loop;
2. Preprocess $GLDDG$:
   (a) Use Algorithm 3.1 to find $GLDDG_{SCC}$ from $GLDDG$;
   (b) Use Algorithm 4.2 to get a local software pipelining result for $GLDDG_{SCC}$ in the absence of resource constraint;
   (c) Use Algorithm 4.3 to modify $GLDDG_{SCC}$;
   (d) Find $GLDDG_m$;
3. Under the constraints of $GLDDG_m$, control flow and resources, use a global loop-free code scheduling technique (such as Trace Scheduling) to schedule (compact) the original loop body, so we can get the new loop body, denoted as LB_{new};
4. Compute the extended row-numbers for all operation-copies in LB_{new};
5. Compute extended column-numbers:
   (a) Find $EGLDDG(LB_{new})$; (see Definition 3.8)
   (b) Find $EGLDDG^{\tau}(LB_{new})$; (see Definition 3.9)
   (c) Use Algorithm 4.3 to compute the extended column-numbers for all operation-copies in LB_{new};
6. Construct the global software pipelined loop:
   (a) $K = \max_{\forall op_i \in LB_{new}}(ecn(op_i))$
   (b) Construct the prelude:
      i. Find $\{1\}$-iteration-body and let $K' = 1$;
      ii. While $(K' < K)$ do { Find $\{1, 2, \ldots, K'+1\}$-iteration-body and build an edge from each exit basic block of $\{1, 2, \ldots, K'\}$-iteration-body to the entrance basic block of $\{1, 2, \ldots, K'+1\}$-iteration-body. $K' = K' + 1$; }
   (c) Construct the loop back-edges: Build an edge from each exit basic block of $\{1, 2, \ldots, K\}$-iteration-body to its entrance basic block;
   (d) Construct the postlude:
      i. Let $K' = 1$;

ii. While $(K' < K)$ do { Find $\{K'+1, \ldots, K\}$-iteration-body and build an edge from each exist basic block of $\{K', K'+1, \ldots, K\}$-iteration-body to the entrance basic block of $\{K'+1, \ldots, K\}$-iteration-body. $K' = K'+1$;

7. Under the pipeline (timing) constraints, delete empty cycles and merge the abundant operations;

$\diamond$

A complete example is given to illustrate how GDESP algorithm works in which we globally software pipeline the loop program shown in Fig.4.1(a) by GDESP algorithm. We assume there are one adder, one multiplier and one store/load unit in the architecture which can be operated in parallel. For simplicity, we also assume the latencies of these functional units are one machine cycle.

Now, let us go step by step.

1. By Definition 4.4, $GLDDG$ is constructed as shown in Fig.4.1(c).

2. Preprocessing $GLDDG$, we can get $GLDDG_m$ (see Fig.4.3).

3. Under the constraints of resources, $GLDDG_m$ and control flow, we use Trace Scheduling algorithm to schedule the original loop body. B1-B2-B4 is chosen as the main trace. The scheduled loop body (that is, the new loop body) is shown in Fig.5.1(a).

4. Computing the extended row-numbers, we get: $ern(2) = ern(4) = ern(3) = (B11, 1)$, $ern(9) = ern(5) = ern(1) = (B12, 1)$, $ern(6) = (B12, 2)$, $ern(7) = ern(1') = ern(9') = (B13, 1)$, $ern(8) = (B13, 2)$.

5. By Definition 4.8, we construct $EGLDDG$ as shown in Fig.5.1(b).

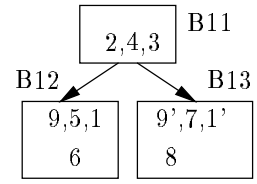
6. By Definition 4.9 and the extended row-numbers, we can construct $EGLDDG^{\tau}$ as shown in Fig.5.1(c).

7. Using Algorithm 4.3, we can get: $ecn(1) = ecn(1') = 1$, $ecn(2) = ecn(3) = ecn(4) = ecn(5) = ecn(6) = ecn(7) = ecn(8) = 2$, $ecn(9) = ecn(9') = 3$.

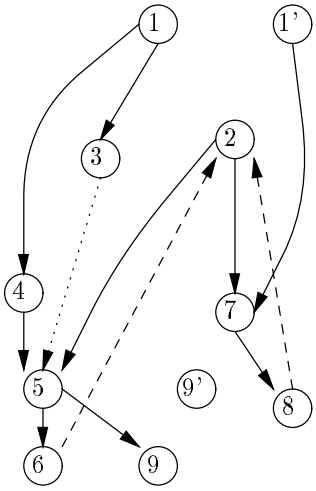
8. By Definition 4.10, we can get $\{1\}$, $\{1,2\}$, $\{1,2,3\}$, $\{2,3\}$, $\{3\}$-iteration-body (denoted as ib in Fig.5.1) as shown in Fig.5.1(d).

9. By steps of 6.2, 6.3 and 6.4 of GDESP algorithm, we can get the global software pipelined loop as shown in Fig.5.1(e).

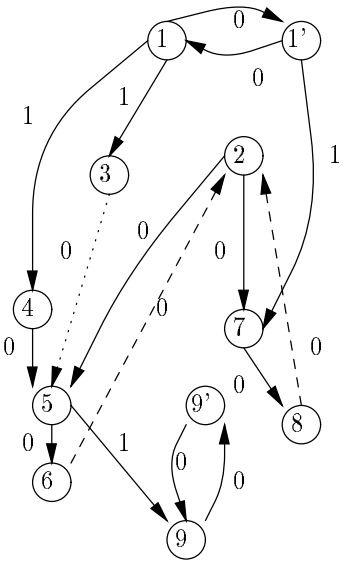
10. After deleting the empty cycles and merge the abundant operations, we get the final result (see Fig.5.1(f)).



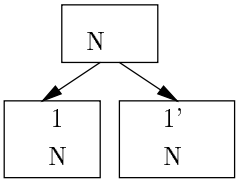
(a) the new loop body
($\{1,2,3\}$-ib)



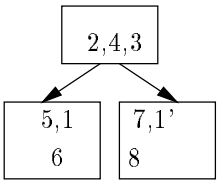
(b) $EGLDDG$



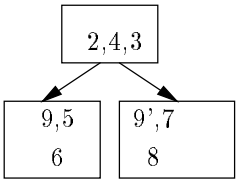
(c) $EGLDDG^{\tau}$



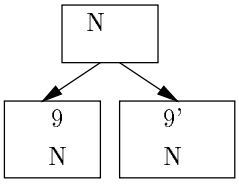
$\{1\}$-ib



$\{1,2\}$-ib

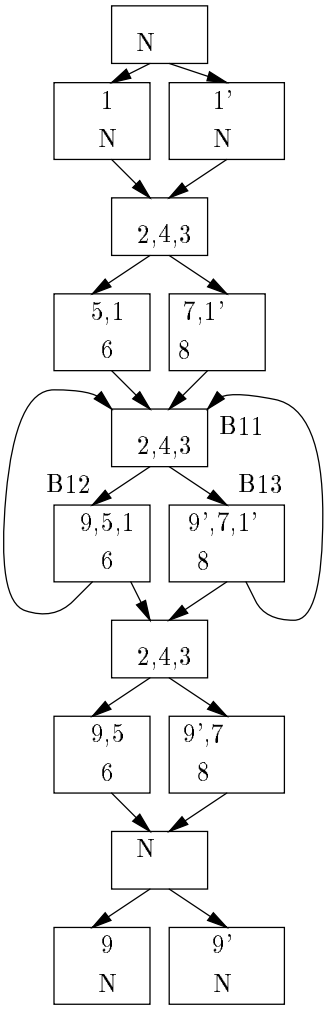


$\{2,3\}$-ib

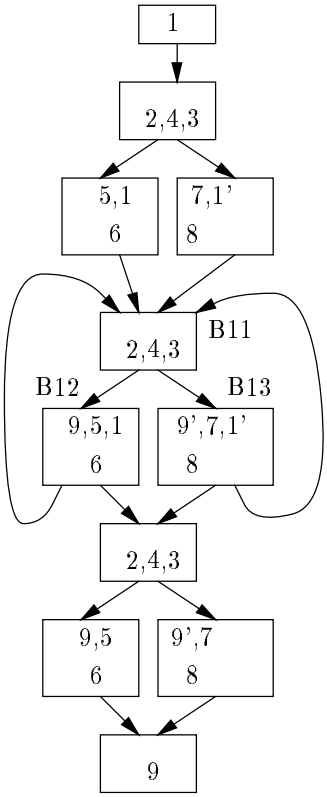


$\{3\}$-ib

(d) iteration-bodies



(e) the global software pipelined loop



(f) the final result

Fig.5.1 An example

6 Discussion and Comparisons

6.1 Local DESP Approach

The computation complexity: We first analyze the computation complexity of the FRLC algorithm. Let n be the number of operations in the given LDDG. It is easy to check that the complexity of step 6,8,9 and 10 is $O(n)$ and that of step 1 is $O(n^3)$. For step 3 and 4, it is also easy to check that the complexity is $O(n^2)$. Since the complexity of the finding longest path algorithm is $O(n^3)$, the complexity of Algorithm 3.2 and 3.3 and the method presented in the proof of Theorem 2.1 is $O(n^3)$. Thus, the complexity of step 2 and 7 is $O(n^3)$. For step 5, in order to check whether the determined row-numbers and h satisfy the condition of Theorem 2.1, we should find the critical cycle of $LDDG^{\tau}$, the complexity is $O(n^3)$. h can be chosen from $\max_{\forall op \in LDDG}(rn(op))$ to $\max_{\forall op \in LDDG}(rn(op) + exe(op))$. So the complexity of step 5 is also $O(n^3)$. Therefore, the complexity of the FRLC algorithm is $O(n^3)$.

By the similar analysis, the computation complexity of FCLR algorithm is also $O(n^3)$. It is not difficult to see that the complexity of step 6,10 and 12 is $O(n)$, that of step 1 is $O(n^3)$ and that of step 3,4,7,8,9 and 11 is $O(n^2)$. For step 2 and 5, since they use the method presented in the proof of Theorem 2.1, the complexity is $O(n^3)$.

Comparison with the Restricted Algorithms: In [Gas89] and [Su91a], software pipelining algorithms are classified into general and restricted algorithms. General algorithms, such as the Perfect Pipelining [Aik88], allow each loop body to have different schedules and initiation intervals. Restricted algorithms, such as URPR [Su86,87,91a,91b], require that each loop body has identical schedules and initiation intervals.

The main idea behind the existing restricted software pipelining algorithms is to use heuristic scheduling techniques to construct the new loop body simultaneously considering the constraints of resources, dependences and cyclicity. However, the loop-carried dependences and the resource constraint make the restricted software pipelining problem very complicated and difficult. Thus, the existing restricted algorithms can not obtain a satisfactory time efficiency with low complexity.

Our DESP approach decomposes the software pipelining into two steps: the determination of row-numbers and the determination of column-numbers. The time efficiency is mainly related with the determination of row-numbers. When we determine the row-numbers we only suffer from the constraints of resources and the modified LDDG which is acyclic. In the modified LDDG, almost all dependence edges are removed (especially, if the LDDG is acyclic, all edges can be removed). Hence, we can exploit the instruction level parallelism for the loop to a great extent, so DESP can get much better time efficiency than the existing restricted algorithms.

Comparison with the General Algorithms: Generally speaking, the general algorithms have much better time efficiency, but worse space efficiency and higher computation complexity than the restricted algorithms. The general algorithms first suppose that the loop is unrolled infinite times, then use a scheduling algorithm to compact the infinitely unrolled loop, finally try to find a "pattern" which is the software pipelined loop body. The "pattern" may be very long so the general algorithms have bad space efficiency. Moreover, finding a

"pattern" sometimes is very difficult (especially in the presence of resource constraint) and causes the high computation complexity.

DESP can achieve the same time efficiency as the general algorithms by loop unrolling. Before we use DESP to pipeline a loop, we can unroll the loop by a certain times [Su91a]. The number of unrolled loop bodies can be computed by

$$K_{unroll} = max(\lceil \max_{\forall C \in LDDG}(\delta(C)/\lambda(C))\rceil, 0)$$

Although the loop unrolling may lengthen the software pipelined loop body, the DESP with loop unrolling still has much better space efficiency than general algorithms.

Comparison with the Algorithm in [Eis92]: Eisenbeis and Windheiser present a new software pipelining algorithm to optimize the loops whose LDDGs are acyclic. The ideas behind the algorithm and our FRLC algorithm are very similar. The difference between these two algorithms is that our FRLC algorithm can optimize any loop, while their algorithm can only optimize those loops whose LDDGs are acyclic. In the presence of resource constraint, these two algorithms can get the optimal results for those LDDGs without any cycle.

Comparison with the Algorithm in [Gas92]: Recently, Gasperoni and Schwiegelshohn present a new loop scheduling algorithm [Gas92]. The ideas behind their algorithm and our FCLR algorithm are similar, while the heuristics by which the ideas are developed to an algorithm are very different. In [Gas92], the given loop is first preprocessed by the Bellman-Ford single source longest paths algorithm in conjunction with a binary search to generate a software pipelined loop without resource constraint; Secondly, if a dependence edge holds in the pipelined loop, it will remain in the modified LDDG. The modified LDDG becomes an acyclic one; Finally the modified LDDG is scheduled by a list scheduler under the constraint of resources. In our FCLR algorithm, we first preprocess the strongly connected components and explicitly determine the column-numbers; Secondly, in terms of the column-numbers, we remove some edges from $LDDG$ so $LDDG$ becomes acyclic. In particular, we can remove all edges which are not included in the strongly connected components; Finally, a list scheduler is used to generate the new loop body. Consequently, our FCLR algorithm is of lower computation complexity than their algorithm as we can only preprocess the strongly connected components. Also, our FCLR algorithm can often generate much better results than their algorithm since FCLR algorithm can remove more dependence edges than their algorithm does. Fig.6.1 and 6.2 give an example.

6.2 GDESP Approach

Computation Complexity of GDESP Approach: Let n be the number of operations in the given loop. In GDESP algorithm, Step 1 is the construction of $GLDDG$, its complexity is $O(n^2)$; The complexity of Step 2 is $O(n^3)$ since finding strongly connected components and finding longest distances need the computation time of $O(n^3)$; In Step 3, a global loop-free code scheduling technique is used, if we adopt Trace Scheduling, the complexity is $O(n^2)$; It is easy to see that step 4 needs $O(n)$; In Step 5, the finding longest distances algorithm is used, so its complexity is $O(n^3)$; In Step 6, we first find $(2K-1)$ iteration-bodies, it needs

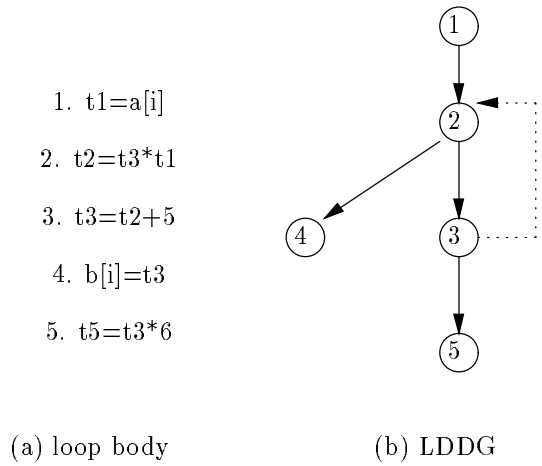


Fig.6.1 An example

the time of $O(K * n)$; then we connect these iteration-bodies one by one, it needs the time of $O(K * n^2)$. Note that K is bounded for a given architecture, so the complexity of Step 6 is $O(n^2)$. Therefore, the computation complexity of GDESP approach is $O(n^3)$ in the worst case.

Time Efficiency: GURPR [Su87] and $GURPR^*$ [Su91b] are two global software pipelining approaches which base on the ideas behind URPR [Su86,91a]. Due to the influence of loop-carried dependences, these two approaches often can not get the same good time efficiency as the existing general global approaches. Besides, they use the instruction insertion method to treat with the problem of resource conflicts and also cause a poor time efficiency. By using a hierarchical reduction, Lam's approach [Lam87,88] can handle the loops with branches, however, it extends the shorter branch in length to match the longer branch and results in a poor time efficiency. GDESP approach can efficiently solve the above problems from which GURPR, $GURPR^*$, and Lam's approach suffer. By using a global loop-free code scheduling technique to generate the global software pipelined loop body, GDESP can get the same good time efficiency as the existing general global approaches for most big loop programs. For small loop programs, loop unrolling can greatly increase the time efficiency of GDESP approach.

Code Explosion Problem: In order to fully exploit instruction level parallelism for loops with branches, the existing general global software pipelining approaches [Aik87,88, Ebc87] first assume the given loop being unrolled infinite times, then use a global loop-free code scheduling technique to compact the infinitely unrolled loop. However, this may cause the code explosion problem. During code motion across basic blocks, the patching code generated by bookkeeping can potentially be exponential in the size on the input loop program. Enhanced Pipeline Scheduling [Nak90], an improvement of Pipeline Scheduling [Ebc87], uses a software lookahead window to control the code explosion, but restricts the parallelism to be exploited. In GDESP, we only use a global loop-free code scheduling technique to compact

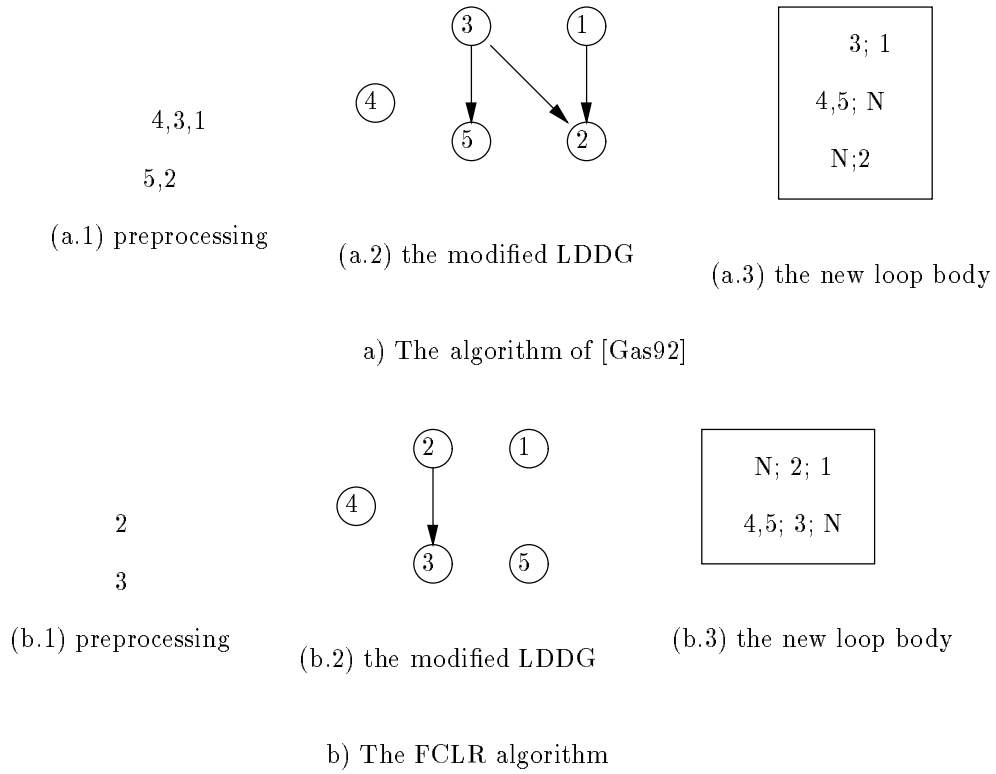


Fig.6.2 Comparison

the original loop body, so the code explosion can be effectively controlled.

Compatibility: We introduce the term of "compatibility" to evaluate a new code optimization approach in such a way: if the approach can be easily implemented into the existing practical optimizing compilers, we say that the approach is of a good compatibility; otherwise, a poor compatibility. GDESP actually decomposes the loop scheduling problem into the loop-free code scheduling problem and some simple problems which are independent of resource constraints and can be easily solved by the classical algorithms of graph theory. For the loop-free code scheduling problem, there have been many algorithms presented since 1979 when J. Fisher presented his Trace Scheduling. In particular, some of them have been used in practical optimizing compilers. Therefore, GDESP can be easily implemented into the existing practical optimizing compilers on the basis of its global loop-free code scheduling algorithm.

7 Conclusion

In this report, we present a new view on software pipelining and develop DESP approach and GDESP approach to exploit instruction level parallelism for loop programs without/with

conditional jumps. DESP/GDESP decomposes the local/global software pipelining problem into two subproblems. one is the loop-free code scheduling problem which can be effectively solved by list scheduling/Trace Scheduling technique; and another is independent of resource constraints and can be easily solved by the classical algorithms of graph theory.

Our future work is to implement DESP/GDESP approach on our compiler testbed and experimentally compare DESP/GDESP with the existing local/global software pipelining approaches. Also, using DESP/GDESP as a basis, we are developing a new technique to combine software pipelining and register allocation.

Reference

[Aik88] A. Aiken and A. Nicolau, Perfect Pipelining: A New Loop Parallelization Technique, In European Symposium on Programming, pp.221-235, Spring-Verlag, Lecture notes in Computer Science, No.300, June 1988.

[Alm89] G.S. Almasi and A. Gottlieb, Highly parallel Computing, The Benjamin/Cummings Publishing Company, Inc., 1989.

[Bod89] F. Bodin, Optimisation de microcode pour une architecture horizontale et synchrone: Etude et mise en oeuvre d'un compilateur, Thèse de Doctorat en Informatique, Irisa/Ifsic, Université de Rennes 1, Juin 1989.

[Cha81] A.E. Charlesworth, An Approach to Scientific Array Processing: The Architecture Design of the AP-120B/FPS-164 Family, Computer,9(1981), pp.18-27.

[Ebc87] K. Ebcioglu, A Compilation Technique for Software Pipelining of Loops with Conditional Jumps, Proc. of MICRO-20, pp.69-79, 1987.

[Eis88] C. Eisenbeis, Optimization of Horizontal Microcode Generation for Loop Structures, Proc. of the 1988 International Conference on Supercomputing, pp. 453-465, Saint-Malo, France, July, 1988.

[Eis92] C. Eisenbeis and D. Windheiser, A New Class of Algorithms for Software Pipelining with Resource Constraints, Rapport de Recherche INRIA, INRIA, 1992.

[Ell86] J.R. Ellis, Bulldog: A Compiler for VLIW Architectures, MIT Press, 1986.

[Fis81] J.A. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, IEEE Trans. on Computers, vol.C-30, No.7, 1981, pp.478-490.

[Fis84] J. Fisher, et al., Parallel Processing: A Smart Compiler and a Dumb Machine, Proc. of ACM SIGPLAN 1984 Symposium on Compiler Construction, pp. 299-305, June 1984.

[Gao91] G. R. Gao, et al., A Timed Petri-Net Model for Fine-Grain Loop Scheduling, Proc. of ACM SIGPLAN'91 Conference on PLDI, June, 1991.

[Gas89] F. Gasperoni, Compilation Techniques for VLIW Architectures, Technical Report 435, New York University, Mar. 1989.

[Gas92] F. Gasperoni and U. Schwiegelshohn, Scheduling Loops on Parallel Processors: A Simple Algorithm with Close to Optimum Performance, Lecture Note, INRIA, 1992.

[Gup87] R. Gupta and M.L. Soffa, Region Scheduling, Proc. of the 2nd International Conference on Supercomputing, pp. 141-148, May 1987.

[Lah83] J. Lah and D.E. Atkins, Tree Compaction of Microprograms, Proc. of MICRO-16, pp.23-33, Oct. 1983.

[Lam88] M.S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machine. Proc. SIGPLAN'88 Conference on PLDI, Atlanta, June, 1988.

[Lin83] J.L. Linn, SRDAG Compaction–A Generalization of Trace Scheduling to Increase the Use of Global Context Information, Proc. of MICRO-16, pp.11-22, Oct 1983.

[Mun91] A. Munier, Résolution d'un Problème d'Ordonnancement Cyclique à Itérations Indépendantes et Contraintes de Ressources, RAIRO-Recherche Opérationnelle, Vo.25/2, 1991, pp.161-182.

[Mun92] A. Munier and C. Hanen, A Study of the Cyclic Scheduling Problem on Parallel Processor, RR766, July 1992, LRI University of PARIS-SUD, Orsay.

[Nak90] T. Nakatani and K. Ebcioglu, Using a Lookahead Window in Compaction-Based Parallelizing Compiler, Proc. of MICRO-23, 1990.

[Nic85] A. Nicolau, Percolation Scheduling: A Parallel Compilation Technique, Technical Report TR-85-678, Cornell University, Department of Computer Science, Ithaca, NY, May 1985.

[Su84] B. Su, S. Ding and L. Jin, An Improvement of Trace Scheduling for Global Microcode Compaction, Proc. of MICRO-17, Dec. 1984.

[Su86] B. Su, et.al., URPR - An Extension of URCR for Software Pipelining, Proc. of MICRO-19, pp.104-108, 1986.

[Su87] B. Su, S. Ding, J. Wang and J. Xia, GURPR–A Method for Global Software Pipelining, Proc. of MICRO-20, 1987.

[Su91a] Bogong Su and Jian Wang, Loop-carried Dependence and the General URPR Software Pipelining Approach, Proc. 24th HAWAII International Conference on System Sciences, Kailua-Kona, Hawaii, Jan. 1991.

[Su91b] Bogong Su and Jian Wang, $GURPR^*$: A New Global Software Pipelining Algorithm, Proc. of MICRO-24, 1991.

[Tou84] R.F. Touzeau, A Fortran Compiler for the FPS-164 Scientific Computer, Proc. ACM SIGPLAN Symposium on Compiler Construction, 1984.