International Journal of Parallel Programming, Vol. 21, No. 6, 1992

Optimal Loop Storage Allocation for Argument-Fetching Dataflow Machines

Qi Ning[1] and Guang R. Gao[2]

Received April 1992; revised August 1993

In this paper, we consider the optimal loop scheduling and minimum storage allocation problems based on the argument-fetching dataflow architecture model. Under the argument-fetching model, the result generated by a node is stored in a unique location which is addressable by its successors. The main contribution of this paper includes: for loops containing no loop-carried dependences, we prove that the problem of allocating minimum storage required to support rate-optimal loop scheduling can be solved in polynomial time. The polynomial time algorithm is based on the fact that the constraint matrix in the formulation is totally unimodular. Since the instruction processing unit of an argument-fetching dataflow architecture is very much like a conventional processor architecture without a program counter, the solution of the optimal loop storage allocation problem for the former will also be useful for the latter.

KEY WORDS: Dataflow architecture; loop scheduling; storage allocation; polynomial algorithm.

1. INTRODUCTION

In scientific computations, loops are the most time consuming part of the programs. How to schedule the operations in a loop so that the execution can achieve maximum computation rate while only allocating minimum amount of high-speed storage spaces (e.g. registers) enough to support such optimal rate is among the most challenging problems faced by compiler writers.

[1] Centre de Recherche Informatique de Montreal, 1801 McGill College Ave, Bureau 800, Montreal, Quebec, Canada.
[2] School of Computer Science, McGill University, Montreal, Quebec, Canada.

0885-7458/92/1200-0421$06.50/0 © 1992 Plenum Publishing Corporation

The method developed in this paper is quite different from many conventional register allocation methods proposed and implemented in compilers. The goal of the conventional register allocation is to minimize the total number of registers used for a sequential execution of the program.[1] Many of these register allocation algorithms are based on the coloring of *interference graphs* representing overlapping relations of the live ranges of program variables in a sequential execution model.[2, 3] Under the sequential execution model, the scheduling of the operations is uniquely determined by the sequential order, so is its interference graph. Our goal is to study the storage allocation problem under a parallel dataflow execution model and to achieve the minimum storage allocation while not reducing the maximal achievable speedup. The optimal scheduling of operations and the relation between live ranges of program variables that they access are closely related, and need to be characterized together in a unified model.

In our paper,[4] we have studied the minimum storage allocation problem for loops under an *idealized dataflow machine model*, and derived the result that the problem can be solved in polynomial time. The architecture we considered in that paper was the FIFO argument-flow dataflow model, which is a generalization of the static dataflow model.[5–7] The FIFO dataflow model described in Ref. 4, like many other dataflow models, is based on the *argument-flow dataflow principle* (a term first coined in Ref. 8) where storage spaces (FIFO queues) are associated with arcs and tokens will "flow" along the arcs when they are produced at the tails of the arcs and are consumed at the heads of the arcs. The argument-flow model, although used to describe the data-driven principle due to its simplicity,[5, 9] has its weakness in terms of storage efficiency, caused by the model of allocating a FIFO queue for each arc.[8]

To overcome the space inefficiency of the argument-flow dataflow models, the argument-fetching dataflow model was proposed,[8, 10] which will be introduced in Section 2. The argument-fetching dataflow model achieves space efficiency by letting all the output arcs of a node share the same FIFO queue. Therefore, result tokens will no longer be flowing along the arcs. Instead the successor nodes will *fetch* the values from the FIFO queues of their predecessors. In an argument-fetching dataflow architecture, the FIFO queue is realized by the high-speed data memory in the processor. This is very much like the conventional von Neumann computation model where an instruction is fetching its operands from registers or memory. Alike its conventional architecture counterpart, the allocation of such high-speed memory in an argument-fetching dataflow architecture is crucial in the compiler design.

We believe that this is the first paper to deal with compile-time storage allocation for argument-fetching architectures. The paper gives a unified

mathematical formulation and polynomial time algorithm. Although, there is earlier work on compile-time storage allocation for dataflow machines,[4, 11–15] all of them are focused on argument-flow models.

The significance of this work goes beyond dataflow architectures. In fact, dataflow graphs are directly useful as an intermediate form for compile-time optimization such as software pipelining on conventional architectures. For instance, in Ref. 16, our software pipelining technique is based on dataflow graphs with *static* dataflow semantics. Although the architecture in the present paper is different from the one used in Ref. 16, the underline techniques for scheduling and allocation are similar. The argument-fetching dataflow graph model used in this paper is particularly interesting since it is based on an operational semantics that maintains the advantage of the dataflow model without using a token-pushing style execution, and supports a general updatable storage model. One important observation is that, since the instruction processing unit of an argument-fetching dataflow architecture is very much like a conventional processor architecture without a program counter, the solution of the optimal loop storage allocation problem for the former should also be useful for the latter. The idea is fully developed in our paper,[17] which can handle loops with loop-carried dependences on conventional architectures.

In this paper, we will solve the problem of minimizing the amount of storage spaces among all optimal schedules. The main contribution of this paper is: for loops containing no *loop-carried dependences*, the problem of minimizing storage spaces required to support rate-optimal loop scheduling can be solved in polynomial time. We show that the problem can be directly formulated into an integer programming problem. We then show that the integer programming problem can be solved in polynomial time. Since the architecture model in this paper is different from the one in Ref. 4, the problem and its formulation here are also different from those in Ref. 4. For loops with loop-carried dependences, the same optimization problem has been proven to be NP-complete.[18]

The paper is organized as follows: In Section 2, we formally state our minimum storage allocation problem for the argument-fetching architecture. In Section 3, we formulate the problem into an integer linear programming problem. In Section 4, we show that the integer programming problem is solvable in polynomial time. We actually prove that the constraint matrix is *totally unimodular* and therefore the integer programming problem can be reduced to the corresponding linear programming problem. Section 5 gives a more efficient algorithm for the solution. It reduces the original problem into a minimum cost network flow problem which has a $O(n^3 \log n)$ algorithm, where n is the number of nodes in the graph representing the loop body. Section 6 shows an example. In Section

7, we include some discussions of the issues involved in generating codes for conventional architectures in this paper. In Section 8, we relate our work with others and point out the differences in our objectives. We give our conclusion in the last section.

2. ARCHITECTURE MODEL AND PROBLEM STATEMENT

In this section, we outline the architecture model and formulate the problem to be studied in the rest of this paper.

2.1. Architecture Model

Our architecture model is an extension of the FIFO dataflow model (as presented in Ref. 4), to adapt the *argument-fetching principle.*

2.1.1. Argument-Fetching Dataflow Principle: A Brief Review

The argument-fetching dataflow principle was first proposed in Ref. 8. The advantages of the argument-fetching principle, as pointed out in the introduction section, is to obtain space efficiency through sharing FIFO queues among all the output arcs of a node. As in conventional dataflow architectures (those based on the *argument-flow dataflow principle* such as proposed in Refs. 5, 9, and 19), a program in an argument-fetching machine is expressed as a directed graph and the execution is based on the data-driven principle: a node becomes *enabled* and can be *fired* only when its operand values are produced. However, in an argument-fetching dataflow architecture, the data (arguments) are not organized as tokens traversing along the arcs in the program graph. Instead, a node is associated with some storage location which is used to store the result value generated by the corresponding operation, in a way very similar to an instruction execution in a conventional von Neumann processor architecture. In other words, under the argument-fetching principle, a result value never needs to be duplicated and sent to destination nodes via its output "arcs," thus eliminating the overhead of excessive token traffic in argument-flow dataflow architectures. As an example of dataflow architectures based on the argument-fetching principle, the readers are referred to the McGill Dataflow Architecture Model.[10, 20, 21]

2.1.2. FIFO Argument-Fetching Dataflow Model

In this paper, we extend the FIFO dataflow model presented in Ref. 4 to use the argument-fetching principle. The major features of the FIFO argument-fetching model are:

- A FIFO queue is associated with each node in the dataflow graph. Such a FIFO queue is used to store the result values produced by the corresponding node. There is no pre-assumed upper bound on the size of the FIFO queue. The successors of a node will "fetch" its operands (arguments) from its result FIFO queue. Therefore, the model is based on the argument-fetching principle.
- There may be more than one activation of a node in simultaneous execution, governed by the firing rule to be described later. The FIFO queue provides both the storage for their results and the mechanism to keep them in the order to be consumed by the successors. Therefore, it has extended the original argument-fetching static dataflow model as described in Ref. 8.

The firing rule of the FIFO argument-fetching model is presented here.

- A node is enabled if the FIFO queues associated with its input predecessor nodes are not empty, and its own result FIFO queue has at least one empty slot (i.e. not full).
- An enabled node can be fired and the firing of the node involves the following steps:
  1. fetch a value from the head of the FIFO queue associated with each of its input predecessor nodes,
  2. perform the operation using the operands just fetched,
  3. save the result of the operation into its own FIFO queue (at the tail).

The FIFO queue associated with a node i has a single writer (node i itself), but may have multiple readers (the successor nodes). The FIFO queue is shared by its readers in such a way that a reader node j can access the FIFO queue independently as if a FIFO queue is logically dedicated to the communication between node i and j. The implementation of the FIFO queue mechanism to efficiently support multiple readers, although interesting, is beyond the scope of this paper. Finally, the FIFO dataflow model inherits nice properties of the ordinary dataflow models (such as well-behavedness[5, 9]), as discussed in Ref. 22.

2.2. Problem Statement

The program model considered in this paper is an innermost loop, which does not contain *loop-carried dependences*. Therefore, the dataflow graph representing the loop body is a directed *acyclic* graph (DAG) $G = (N, E)$ which represents the data dependence graph for one iteration of

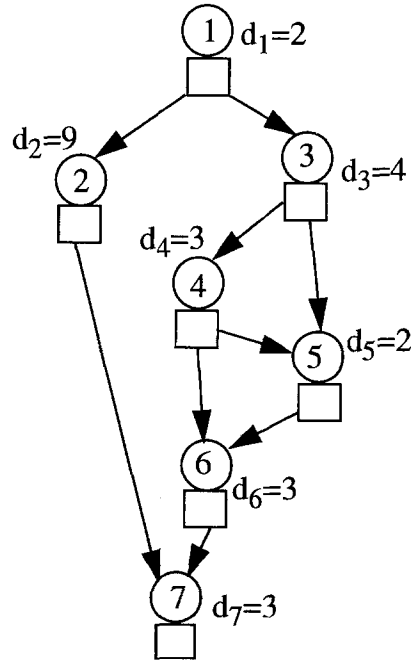
the loop body. N is the set of nodes and E is the set of arcs. Figure 1 shows an example loop and its dataflow graph for one iteration. There are seven nodes in the graph numbered from 1 to 7. Node 1 is the starting node and node 7 is the terminating node. The number beside an arc is the delay of that arc. Notice that the longest path from node 1 to node 7 is the path $\{(1, 3), (3, 4), (4, 5), (5, 6), (6, 7)\}$, and its length is 14.

The fine-grain parallelism in a loop (such as the previous example) can be easily exploited by dataflow software pipelining. The technique of dataflow software pipelining involves the arrangement of machine code such that successive computations can follow each other through one copy of the code. If we supply a sequence of values to the inputs (e.g. $X[i]$'s and $Y[i]$'s) in the dataflow graph, these values can flow through the program in a pipelined fashion.

For the loop L in Fig. 1, if not enough memory is allocated to the nodes, then it cannot support the maximum computation rate allowed by the idealized argument-fetching dataflow architecture, which is obviously 1,

```
      for i = 1 to N do
S1:   a[i] = x[i] + y[i];
S2:   b[i] = a[i] / 3;
S3:   c[i] = a[i] * z[i];
S4:   e[i] = c[i] - w[i];
S5:   f[i] = c[i] + e[i];
S6:   g[i] = f[i] * e[i];
S7:   h[i] = b[i] * g[i];
      endfor
```

(a)



(b)

Fig. 1. An example loop L.

i.e. one iteration initiation per time step. For example, node 1 could be fired at time step 0, and produce its result at the end of time step 1 (because $d_1 = 2$). Then node 2 could be fired at time step 2 and produce its result at the end of time step 10 ($d_2 = 9$). For the second iteration, we can start executing node 1 just one time step after node 1 in iteration 1 is initiated. The same is true for node 2. That is to say, node 1 should be fired the second time at time step 1, producing a result at the end of time step 2. Similarly, node 2 should be fired the second time at time step 3 and output its result at the end of time step 11. At this time, node 2 has produced two results (which should be stored in the registers). But node 7, which uses the results, can not be fired before time step 14 because of the longer data dependency paths along the other nodes from node 1 to node 7. Therefore, if the number of storage spaces allocated to node 2 is not enough, we have to choose not to fire node 2 consecutively. This will make the computation rate slower. Therefore, our problem statement can be presented as follows: Given a loop as previously described, how do we allocate the minimum amount of high-speed storage spaces to the nodes such that the graph will be executed at the maximum computation rate?

3. FORMULATION

In this section we formulate the minimum storage problem into an integer programming problem. We then show that the problem can be solved in polynomial time in the next section.

Let T_i denote the time when the first instance of node i is scheduled (i.e., execution in the first iteration). Once we have determined the T_i's for all the nodes in the graph, we can schedule the nodes for their second instances in the second iteration (which is represented by the same graph) at time $T_i + 1$, and their third instances in the third iteration at time $T_i + 2$, etc. In general, the kth instance of node i in the kth iteration will be scheduled at time $T_i + k - 1$.

Therefore the T_i's determine a schedule for all the instances of the instruction in the dataflow graph, which initiates one iteration per clock cycle. Our goal is to find such feasible T_i's such that the schedule it determines uses the minimum amount of storage spaces.

Next we investigate the lower bound on the size of the FIFO queue to be allocated for a node i.

Suppose that node j is a successor of node i and $e = (i, j)$ is an arc from i to j. Let T_i, T_j denote the scheduled time for the first instances of nodes i and j, respectively. Then they must satisfy the following timing constraint:

$$T_j - T_i \geq d_i \tag{1}$$

where d_i is the number of clock cycles to execute node i. Now the node i will be fired (scheduled) for the second time, third time, fourth time, etc. at time T_i+1, T_i+2, T_i+3 etc. The results of the consecutive firings of node i have to be stored in the FIFO queue allocated to node i. When node i is fired at time T_i, it will produce a result at time T_i+d_i. That value will be consumed by node j at time T_j. Therefore the time difference between producing the value and its consumption is:

$$T_j - T_i - d_i$$

So, the number of results produced by the consecutive firings of node i before they are consumed by node j is:

$$T_j - T_i - d_i + 1$$

which in turn means that we should allocate at least that amount of memory spaces for the FIFO queue of node i.

When considering all the successors of node i, we must make sure that the amount we allocate to its FIFO queue be not smaller than the amount we calculated earlier for each individual output arc. Let B_i be the size of the FIFO queue for node i, then we must ensure that:

$$B_i \geqslant T_j - T_i - d_i + 1, \qquad \forall j \in \delta^+(i) \tag{2}$$

where $\delta^+(i)$ is the set of nodes j in the graph that are connected *from* the node i by arcs $(i, j) \in E$, and E is the set of arcs in the dependence graph. Similarly, $\delta^-(i)$ can be defined to be the set of nodes j in the graph that are connected *to* the node i by arcs $(j, i) \in E$.

We assume that there exists a unique starting node s and a unique terminating node t and all the other nodes are on some path(s) from the starting node s to the terminating node t in the graph. The starting node s will be executed before any other node in any iteration. If the original graph has multiple starting nodes, we can add a new starting node s and add arcs from s to each of the original starting nodes. We assume that the execution time (or delay) of the new starting node equals zero. A similar procedure can be applied to the terminating nodes if the original graph has multiple terminating nodes. Therefore, the unique terminating node is the last one to be executed in any iteration and its execution time can be assumed to be zero (since it has no effect on the scheduling of the other nodes in the graph).

Now let us use L to denote the length of the longest dependence path from s to t. Then the optimal schedule for the loop should finish any iteration in L time cycles. Therefore the minimum storage allocation problem

for an optimal schedule of the loop without loop-carried dependences can be formulated into the following integer programming problem, which combines the timing constraints in Eq. (2) and the size constraints in Eq. (1).

Problem (I):

$$\min \sum_{i \in N} B_i$$

subject to

$$\begin{aligned} B_i + T_i - T_j &\geqslant 1 - d_i, \qquad \forall (i, j) \in E \\ T_j - T_i &\geqslant d_i, \qquad \forall (i, j) \in E \\ T_t - T_s &= L \\ T_i, B_i &\text{ integers}, \qquad \forall i \in N \end{aligned}$$

The solution of this programming problem will give a valid optimal schedule determined by the T_i's and a supporting storage allocation for the nodes determined by the B_i's. We will show in the next section that this formulation can be solved in polynomial time.

4. POLYNOMIAL TIME SOLUTION

We will show next that **Problem (I)** can be treated like a linear programming problem instead of an integer programming problem. From the known polynomial time algorithms, i.e., ellipsoid method[23] and interior point method,[24] the linear programming problem can be solved in polynomial time. Thus the original integer programming **Problem (I)** can be solved in polynomial time.

Actually, we will show that the constraint matrix of **Problem (I)** is totally unimodular. The total unimodularity property of the constraint matrix will guarantee that the linear relaxation problem has integer optimal solution.

First, we give some definitions.

Definition 4.1. The O-incidence matrix A^+ of (directed) graph G is a matrix with the rows indexed by arcs, the columns indexed by nodes, and the elements defined by:

$$a^+_{ev} = +1, \qquad \forall v \in N,\ \forall e \in E \text{ and } v \text{ is the tail of arc } e$$

Similarly, we can define the I-incidence matrix.

Definition 4.2. The I-incidence matrix A^- of graph G is a matrix with rows indexed by arcs, the columns indexed by nodes, and the elements defined by:

$$a_{ev}^- = -1, \qquad \forall v \in N, \forall e \in E \text{ and } v \text{ is the head of arc } e$$

With these definitions, we can see that each row of A^+ (or A^-) has exactly one $+1$ (-1) at the column indexed by the tail (head) of the arc. And for each column of A^+ (or A^-), the number of $+1$'s (-1's) is the out-degree (in-degree) of the node indexing that column.

Definition 4.3. The incidence matrix of graph G is:

$$A = A^+ - A^-$$

So, each row of the incidence matrix has one $+1$ and one -1 in the columns indexed by the tail node and head node of that arc, respectively.

Next, we cite the definition of totally unimodular matrix and a necessary and sufficient condition to test whether a matrix has such a property.

Definition 4.4. A matrix is called totally unimodular (TUM) if the determinant of every square submatrix of the matrix is either 0 or $+1$ or -1.

Since each element of the matrix is considered as a one-by-one square submatrix, we can see that each element of a totally unimodular matrix must be either 0 or $+1$ or -1.

Testing whether a given matrix has the TUM property is not easy in general, although complicated procedures have been developed to do that in polynomial time.[25] There are some known necessary and sufficient conditions in the literature for testing the TUM property. Here, we only list one which will be used later in our proof.

Definition 4.5. A submatrix of a $\{0, \pm 1\}$ matrix is called Eulerian if the sum of the elements in each row and in each column of the submatrix is even.

Theorem 4.1 (Camion[26]). A $\{0, \pm 1\}$ matrix is totally unimodular if and only if the sum of elements in each Eulerian submatrix can be divided by 4.

Now, we are ready to prove our key lemma.

Lemma 4.1. The constraint matrix in **Problem (I)** is totally unimodular.

Proof. The constraint matrix in **Problem (I)** has very strong relations with the incidence matrix of the graph. Actually, the constraint matrix is the following matrix:

$$\begin{bmatrix} A^+ & A \\ O & -A \\ O & R \end{bmatrix}$$

where O are submatrices of proper sizes of all 0 (zero) elements, and R is a row vector indexed by the nodes in which all the elements are 0 except at two positions: at position indexed by s, the element equals -1, and at position indexed by t, the element equals $+1$. A^+ is the O-incidence matrix, and A the incidence matrix.

Let us index the constraint matrix in the following way: the first n columns are indexed by $v'_1, v'_2, \ldots, v'_n$, and the remaining n columns are indexed by $v_1, v_2, \ldots, v_n$, where n is the number of nodes in the graph; the first m rows are indexed by $e'_1, e'_2, \ldots, e'_m$, and the next m rows are indexed by $e_1, e_2, \ldots, e_m$, where m is the number arcs in the graph. The last row is indexed by e_{st}. Here we assume that v'_i and v_i represent the same node i in the graph. Similar assumptions for arcs are also true.

Therefore the v''s index the columns of the submatrix $\begin{pmatrix} A^+ \\ O \\ O \end{pmatrix}$ in the constraint matrix. Similarly, the n's index the columns of the submatrix $\begin{pmatrix} A \\ -A \\ R \end{pmatrix}$, e's index the rows of the submatrix $(A^+ \quad A)$, and e's index the rows of the submatrix $(O \quad -A)$.

Let H be an arbitrary Eulerian submatrix of the constraint matrix. In general, we can assume that some rows of H are indexed by some e''s and some others are indexed by some e's and H may or may not have a row indexed by e_{st}. Similarly, we can assume that some columns of H are indexed by some v''s and others are indexed by some v's.

Consider a column p' of H indexed by an v'_i. Note that p' can only contain zeros and $+1$. Since H is Eulerian, the number of $+1$'s in p' must be even.

If column p indexed by v_i (which represents the same node as v'_i does) is also in H, then at the rows where p' has $+1$'s, p must also has $+1$'s, because the $+1$'s actually indicate the tails of the same arcs in the graph. Therefore, the number of $+1$'s in p' and the corresponding $+1$'s in p can be divided by 4.

Now, consider the case where the column indexed by v_i is not in H. Since H is Eulerian, the sum of elements in each row of H should be even. Let us assume that there is a $+1$ in column p' at row q' indexed by e'_j. Note that row q' contains only one $+1$ because the column indexed by v_i is not in H in this case. But H is Eulerian, and hence the sum of elements in row q' must be even. This means that there must be a -1 in row q'. So row q' contains one $+1$ and one -1. The sum of elements for row q' is zero which can be divided by 4.

Next, let w be a row of H in which there is no $+1$ on columns indexed by the v''s. Then, w can contain at most one $+1$ and at most one -1. Since H is Eulerian, w either contains only zeros or contains exactly one $+1$ and one -1. In all cases, the sum of elements in row w is zero, which can be divided by 4.

Since we have counted all the nonzero elements in H, we conclude that the sum of elements in H can be divided by 4. Hence, by Camion's Theorem 4.1, the constraint matrix of **Problem (I)** is totally unimodular. □

By a result in integer programming (see Ref. 25), if a problem has a totally unimodular constraint matrix, then the linear relaxation of the original problem will always has an optimal solution in integer values if the right-hand sides of the constraints are also integral. For **Problem (I)**, its linear relaxation is the following linear programming problem which is just the formulation of **Problem (I)** without the integer requirements:

Problem (II):

$$\min \sum_{i \in N} B_i$$

subject to

$$B_i + T_i - T_j \geqslant 1 - d_i, \qquad \forall (i, j) \in E$$

$$T_j - T_i \geqslant d_i, \qquad \forall (i, j) \in E$$

$$T_t - T_s = L$$

Lemma 4.1 actually implies the following theorem by the fact that a totally unimodular constraint matrix guarantees integer solution:

Theorem 4.2. The integer linear programming **Problem (I)** is equivalent to **Problem (II)** which can be solved in polynomial time.

Proof. The number of constraints in **Problem (II)** is $2m + 1$ where m is the number of arcs in the graph, and the number of variables in **Problem (II)** is $2n$ where n is the number of nodes in the graph.

Therefore, the linear relaxation **Problem (II)** can be solved in polynomial time by the ellipsoid method or the interior point method. Now, since the constraint matrix of the linear programming problem is totally unimodular, the optimal solution will always be integral. Therefore, the optimal solution is also a solution to the integer programming **Problem (I)**. □

5. A MORE EFFICIENT SOLUTION

In the previous section, we showed that the minimum storage allocation problem for a loop can be solved in polynomial time. But, the argument there is based on the fact that the general linear programming problem can be solved in polynomial time by either the Khachian's ellipsoid method[23] or the Karmarkar's interior point method.[24] However, both of these two methods are not quite efficient for our practical application in an optimizing compiler.

In this section, we show a more efficient algorithm to solve **Problem (I)**. The algorithm is a number of transformations of the problem to the minimum cost flow problem. Since the minimum cost flow problem can be solved more efficiently by the so called combinatorial algorithms, this will imply that our original **Problem (I)** can also be solved more efficiently.

Let us first write down the linear dual of **Problem (II)**:

$$\max \sum_{(i,j)\in E} ((1-d_i)\lambda_{ij}) + \sum_{(i,j)\in E} d_i\pi_{ij} + L\alpha_{st}$$

subject to

$$\sum_{j\in\delta^+(i)} \lambda_{ij} = 1, \qquad \forall i \in N$$

$$\sum_{j\in\delta^+(s)} \lambda_{sj} - \sum_{j\in\delta^+(s)} \pi_{sj} - \alpha_{st} = 0$$

$$\sum_{j\in\delta^+(i)} \lambda_{ij} - \sum_{j\in\delta^-(i)} \lambda_{ji} - \sum_{j\in\delta^+(i)} \pi_{ij} + \sum_{j\in\delta^-(i)} \pi_{ji} = 0, \qquad \forall i \in N - \{s, t\}$$

$$-\sum_{j\in\delta^-(t)} \lambda_{jt} + \sum_{j\in\delta^-(t)} \pi_{jt} + \alpha_{st} = 0$$

$$\lambda_{ij} \geqslant 0,\ \pi_{ij} \geqslant 0, \qquad \forall (i,j) \in E$$

If we reorganize the variables in the objective function, then the objective function can be written as:

$$\sum_{(i,j)\in E}((1-d_i)\lambda_{ij})+d_i\pi_{ij}+L\alpha_{st}$$

$$=\sum_{(i,j)\in E}(\lambda_{ij}-d_i\lambda_{ij}+d_i\pi_{ij})+L\alpha_{st}$$

$$=\sum_{(i,j)\in E}\lambda_{ij}-\sum_{(i,j)\in E}(d_i\lambda_{ij}-d_i\pi_{ij})+L\alpha_{st}$$

$$=\sum_{i\in N}\sum_{j\in\delta^+(i)}\lambda_{ij}-\sum_{(i,j)\in E}(d_i\lambda_{ij}-d_i\pi_{ij})+L\alpha_{st}$$

$$=\sum_{i\in N}1-\sum_{(i,j)\in E}(d_i\lambda_{ij}-d_i\pi_{ij})+L\alpha_{st}$$

$$=n-\sum_{(i,j)\in E}d_i(\lambda_{ij}-\pi_{ij})+L\alpha_{st}$$

The first term n in the last equation is a constant, so can be ignored in the objective function. The variables in the constraints can also be rearranged. After these rearrangements and the change of sign of the objective function, the dual problem can be written in the following form which will be called **Problem (III)**:

Problem (III):

$$\min\sum_{(i,j)\in E}d_i(\lambda_{ij}-\pi_{ij})+L\alpha_{st}$$

subject to

$$\sum_{j\in\delta^+(i)}\lambda_{ij}=1,\qquad \forall i\in N$$

$$\sum_{j\in\delta^+(s)}(\lambda_{sj}-\pi_{sj})-\alpha_{st}=0$$

$$\sum_{j\in\delta^+(i)}(\lambda_{ij}-\pi_{ij})-\sum_{j\in\delta^-(i)}(\lambda_{ji}-\pi_{ji})=0,\qquad \forall i\in V-\{s,t\}$$

$$-\sum_{j\in\delta^-(t)}(\lambda_{jt}-\pi_{jt})+\alpha_{st}=0$$

$$\lambda_{ij}\geqslant 0,\ \pi_{ij}\geqslant 0,\qquad \forall(i,j)\in E$$

Next, we want to show that the dual **Problem (III)** is equivalent to a minimum cost network flow problem. Now, we do a variable substitution for **Problem (III)**:

$$f_{ij}=\lambda_{ij}-\pi_{ij},\qquad \forall(i,j)\in A \tag{3}$$

With this variable substitution, **Problem (III)** becomes the following:

Problem (IV):

$$\min \sum_{(i,j)\in E} d_i f_{ij} + L\alpha_{st}$$

subject to

$$\sum_{j\in\delta^+(i)} f_{ij} = \sum_{j\in\delta^+(i)} \pi_{ij} + 1, \qquad \forall i \in N$$

$$\sum_{j\in\delta^+(s)} f_{sj} - \alpha_{st} = 0$$

$$\sum_{j\in\delta^+(i)} f_{ij} - \sum_{j\in\delta^-(i)} f_{ji} = 0, \qquad \forall i \in V - \{s, t\}$$

$$-\sum_{j\in\delta^-(t)} f_{jt} + \alpha_{st} = 0$$

$$f_{ij} \text{ unrestricted}, \ \pi_{ij} \geqslant 0, \qquad \forall (i, j) \in E$$

The first set of constraints in **Problem (IV)** can be further simplified by the following argument. Since π_{ij}'s are nonnegative variables and they do not appear in either the objective function or the other constraints in **Problem (IV)**, we can see that the first set of constraints is equivalent to the following:

$$\sum_{j\in\delta^+(i)} f_{ij} \geqslant 1, \qquad \forall i \in N \tag{4}$$

Lemma 5.1. Constraints

$$\sum_{j\in\delta^+(i)} f_{ij} = \sum_{j\in\delta^+(i)} \pi_{ij} + 1, \qquad \forall i \in N \tag{5}$$

$$f_{ij} \text{ unrestricted}, \ \pi_{ij} \geqslant 0, \qquad \forall (i, j) \in E$$

and constraints:

$$\sum_{j\in\delta^+(i)} f_{ij} \geqslant 1, \qquad \forall i \in N \tag{6}$$

$$f_{ij} \text{ unrestricted}, \qquad \forall (i, j) \in E$$

are equivalent.

Proof. Since $\pi_{ij} \geqslant 0$, Eq. (5) implies Eq. (6). Now we only need to show how to obtain a set of values for the π_{ij}'s given a solution of f_{ij} of Eq. (6).

For each node i, choose a unique $j_i \in \delta^+(i)$. Then define:

$$\pi_{ij_i} = \sum_{j \in \delta^+(i)} f_{ij} - 1, \qquad \forall (i, j_i) \in E$$

and define

$$\pi_{ij} = 0, \qquad \forall j \in \delta^+(i) - \{j_i\}$$

Such defined π_{ij}'s are nonnegative and they satisfy Eq. (5). □

Hence, **Problem (IV)** is equivalent to the following problem, in which the first set of constraints has been replaced by Eq. (4):

Problem (V):

$$\min \sum_{(i,j) \in E} d_i f_{ij} + L\alpha_{st}$$

subject to

$$\sum_{j \in \delta^+(i)} f_{ij} \geqslant 1, \qquad \forall i \in N$$

$$\sum_{j \in \delta^+(s)} f_{sj} - \alpha_{st} = 0$$

$$\sum_{j \in \delta^+(i)} f_{ij} - \sum_{j \in \delta^-(i)} f_{ji} = 0, \qquad \forall i \in V - \{s, t\}$$

$$- \sum_{j \in \delta^-(t)} f_{jt} + \alpha_{st} = 0$$

$$f_{ij} \text{ unrestricted}, \qquad \forall (i, j) \in E$$

The first set of constraints in **Problem (V)** gives a lower limit on the sum of output flow for each node, which does not appear in an ordinary minimum cost flow problem formulation. We will show how we can split the nodes in the graph to make the current formulation fit into the ordinary minimum cost flow problem. Actually, we can replace each node i in the original graph by two nodes i' and i''. The original input arcs to node i are now directed to node i'. The original output arcs from node i are now going out from node i''. We also add a new arc from node i' to node i''. Now consider the ordinary minimum cost flow problem on this

split graph. Let N' be the set of i' nodes and N'' be the set of i'' nodes. We use E' to denote the set of arcs in the split graph.

It is easy to see that the following minimum cost flow **Problem (VI)** is equivalent to **Problem (V)**.

Problem (VI):

$$\min - \sum_{(u,v)\in E'} d'_{uv} f'_{uv} + L\alpha_{st}$$

subject to

$$\sum_{v\in\delta^+(s'')} f'_{s''v} = \alpha_{st}$$

$$\sum_{v\in\delta^+(u)} f'_{uv} - \sum_{v\in\delta^-(u)} f'_{vu} = 0, \qquad \forall u \in N' \cup N''$$

$$\sum_{v\in\delta^-(t')} f'_{vt'} = \alpha_{st}$$

$$f'_{uv} \geqslant 1, \qquad \forall (u,v) \in E'$$

where we define the cost coefficients in the objective function by

$$d'_{uv} = \begin{cases} d_i, & \text{if } u = i'' \in N'' \text{ and } v = j' \in N' \\ 0, & \text{if } u = i' \in N' \text{ and } v = i'' \in N'' \end{cases}$$

Lemma 5.2. Given an optimal solution of **Problem (VI)** $\{f'_{uv}\}_{(u,v)\in E'}$ and α_{st} the $\{f_{ij}\}_{(ij)\in E}$ defined by the following formula together with α_{st} is an optimal solution of **Problem (V)**:

$$f_{ij} = f'_{uv}, \qquad \text{if } u = i'' \in N'' \text{ and } v = j' \in N' \text{ and } i \neq j$$

Similarly, given an optimal solution of **Problem (V)** $\{f_{ij}\}_{(i,j)\in E}$ and α_{st}, the following defined $\{f'_{uv}\}_{(u,v)\in E'}$ together with α_{st} is an optimal solution of **Problem (VI)**:

$$f'_{uv} = \begin{cases} f_{ij}, & \text{if } u = i'' \in N'' \text{ and } v = j' \in N' \text{ and } (i,j) \in E \\ \sum_{j\in\delta^+(i)} f_{ij}, & \text{if } u = i' \in N' \text{ and } v = i'' \in N'' \end{cases}$$

The proof of the lemma is straightforward and left to the reader.

It is well known that the minimum cost flow problem can always obtain an optimal solution in integer values if all the capacity constraints on the arcs are integral.[25] The capacity constraints on the arcs in **Problems (V)** and **(VI)** are integral, therefore they have optimal solutions in integer values. Actually, the efficient out-of-kilter algorithm (see Ref. 26)

and its variants will give such an optimal integer solution when it is applied to **Problem (VI)**.

Now we summarize our procedure to get the integer optimal solution of **Problem (I)**.

Step 1. Solve **Problem (VI)**. This is an ordinary minimum cost network flow problem which can be solved in $O(n^3 \log n)$ time by the out-of-kilter algorithm,[27] where n is the number of nodes in the graph. The solution obtained is *integral*, which is guaranteed by the totally unimodular property of the constraint matrix and the out-of-kilter algorithm.

Step 2. Transform the solution obtained in Step 1 into an optimal solution of **Problem (V)** by the formula in Lemma 5.2. The transformation preserves the integrality of the variables.

Step 3. Transform the solution obtained in Step 2 into an optimal solution of **Problem (IV)** by defining a set of π_{ij}'s satisfying Eq. (4) by the method in proof of Lemma 5.1. This will also preserve the integrality of the variables.

Step 4. Then, using the reverse variable substitution of Eq. (3):

$$\lambda_{ij} = f_{ij} - \pi_{ij}, \qquad \forall (i, j) \in E$$

we obtain the solution of **Problem (III)** which is the dual of **Problem (II)**. Read off the primal optimal solution of **Problem (II)** from the solution of **Problem (III)**. This will also guarantee to get an integer optimal solution of **Problem (II)**. The B variables are the required values for the minimum storage allocation scheme of **Problem (I)**.

Theorem 5.1. The minimum storage allocation **Problem (I)** or **(II)** can be solved in $O(n^3 \log n)$ time, where n is the number of nodes in the graph representing the loop.

6. EXAMPLE CONTINUED

In this section, we solve the integer linear programming problem for the simple example loop L we have seen in Fig. 1 and illustrate how we can make best use of these FIFO queues.

The integer programming **Problem (I)** of the graph in Fig. 1 is:

$$\min B_1 + B_2 + B_3 + B_4 + B_5 + B_6$$

subject to

$$B_1 + T_1 - T_2 \geqslant 1 - 2 = -1$$
$$B_1 + T_1 - T_3 \geqslant 1 - 2 = -1$$
$$B_2 + T_2 - T_7 \geqslant 1 - 9 = -8$$
$$B_3 + T_3 - T_4 \geqslant 1 - 4 = -3$$
$$B_3 + T_3 - T_5 \geqslant 1 - 4 = -3$$
$$B_4 + T_4 - T_5 \geqslant 1 - 3 = -2$$
$$B_4 + T_4 - T_6 \geqslant 1 - 3 = -2$$
$$B_5 + T_5 - T_6 \geqslant 1 - 2 = -1$$
$$B_6 + T_6 - T_7 \geqslant 1 - 3 = -2$$
$$T_2 - T_1 \geqslant 2$$
$$T_3 - T_1 \geqslant 2$$
$$T_4 - T_3 \geqslant 4$$
$$T_5 - T_3 \geqslant 4$$
$$T_5 - T_4 \geqslant 3$$
$$T_7 - T_2 \geqslant 9$$
$$T_6 - T_4 \geqslant 3$$
$$T_6 - T_5 \geqslant 2$$
$$T_7 - T_6 \geqslant 3$$
$$T_7 - T_1 = 14$$
$$B_1, \ldots, B_6, T_1, \ldots, T_7 \text{ integers}$$

An optimal solution of this problem is listed here:

$$T_1 = 0, \quad T_2 = T_3 = 2, \quad T_4 = 6, \quad T_5 = 9, \quad T_6 = 11, \quad T_7 = 14$$
$$B_1 = 1, \quad B_2 = 4, \quad B_3 = 4, \quad B_4 = 3, \quad B_5 = 1, \quad B_6 = 1$$

The storage allocation of FIFO queues for the instructions in the loop L is represented in Fig. 2. We did not calculate the amount of storage for node 7 since that amount depends on the subsequent instructions outside the loop which will use the results of node 7.

Now, let us see how these FIFO queues are used in the execution process. In Table I, the cycle time coordinate is on the vertical axis, and the iteration coordinate is on the horizontal axis. The first iteration of the loop is executed according to the schedule produced by the linear programming solution. So, we can see that s_1 is executed at time 0, s_2, s_3 are executed at time 2, etc. We can also see in Table I that iterations are scheduled exactly one cycle after iteration one is initiated.

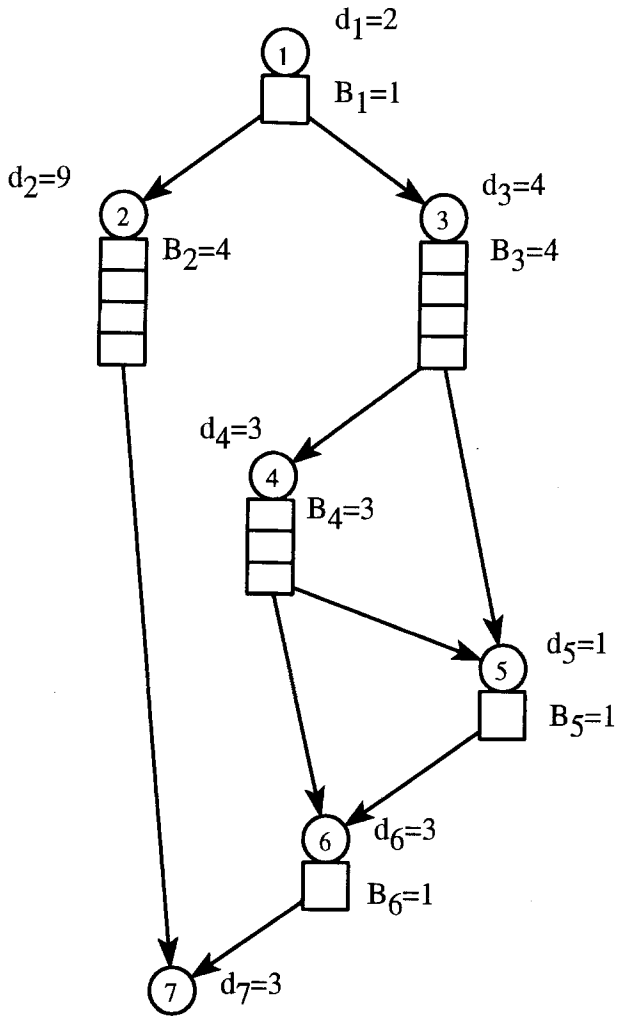


Fig. 2. Optimal memory allocation for the loop L.

Table I. Execution of the loop *L*

	iter 1	iter 2	iter 3	iter 4	iter 5	...
0	s_1					
1		s_1				
2	s_1, s_2		s_1			
3		s_1, s_2		s_1		
4			s_1, s_2		s_1	
5				s_1, s_2		
6	s_4				s_1, s_2	
7		s_4				
8			s_4			
9	s_5			s_4		
10		s_5			s_4	
11	s_6		s_5			
12		s_6		s_5		
13			s_6		s_5	
14	s_7			s_6		
15		s_7			s_6	
16			s_7			
17				s_7		
18					s_7	

For iteration 1, s_1 is executed at time 0, and its execution time will last 2 cycles, therefore at the end of cycle 1, s_1 will output its result into the FIFO queue element allocated to it so that at the beginning of cycle 2, s_2, s_3 can read the data in that slot. Then, s_2 is executed at cycle 2, and it will produce its result at the end of time 10 since the execution time of s_2 is 9. The result value is stored in the first empty slot of its FIFO queue of size 4. s_7 will use this result to do its own computation, but s_7 is scheduled to be executed at time 14. So, there is a 4 cycle-delay between the availability of the data and the time it is actually used. Since the second iteration is started one cycle after iteration 1, s_2 in iteration 2 will output its result at time 11. Now this result of s_2 in iteration 2 must use a new slot in its FIFO queue because the old result of s_2 in iteration 1 has not yet been used. Actually for iteration 3, s_2 will output its result at time 12, and this result also needs an empty slot. In iteration 4, s_2 will output its result at time 13 and another new empty slot is needed. Up to now, we see that the FIFO queue for s_2 should be at least of size four. However, in iteration 5, s_2 will output its result at the end of time step 14, and this result does not need a new slot since at the beginning of time step 14, s_7 is executed using the result produced by s_2 in iteration 1, which frees a slot in the FIFO queue of s_2 so that the result of the fifth execution of s_2 can be put into that freed slot. That is why we need to allocate a FIFO queue of size exactly four to

s_2. Similarly, we can see that s_3 needs a FIFO queue of size four and so on.

From our analysis of node s_2, we see that a FIFO queue of size four is allocated to it and all locations in the FIFO queues are all busy all the time. This is because the results are produced one per cycle. Therefore, different FIFO queues can not share their element(s). However, if the results can not be produced at one per cycle rate, which is the case if the loop contains loop-carried dependences, then sharing the elements in the FIFO queue is possible. When sharing of memory spaces is allowed, the problem of minimum allocation becomes substantially harder. It is our future research direction to study the loops that contain loop-carried dependences and how the sharing would save more spaces.

7. DISCUSSIONS

7.1. Implementation Issues

The argument-fetching architecture model that we used in this paper, as in many classical dataflow architecture models, does not make specific assumptions on register storage. In fact, our basic assumption is that the execution units are equipped with some operand memory. Under our scheme, some of the operand memory space is used as FIFI buffers. The goal of this paper is to propose a method to determine the minimum FIFO storage needed for maintaining a maximum speedup schedule under our model, and to determine an assignment of the buffers to the nodes in the program. There is an implementation issue on how to build a FIFO queue so that the successors can access it at the right addresses. A solution of this problem can be found in Refs. 17 and 18.

If the number of buffers obtained from the solution of our method is bigger than the number of buffers available on the machine, then two alternatives can be adapted:

1. Slow down the initiation rate of successive iterations. In this paper, we have taken the initiation period of iterations to be one per cycle in order to maximize the speedup. However, if the available buffers can not support this maximum speedup, we can try longer initiation periods so that the storage requirement of the program will eventually fit in the given memory space.
2. Or, we can introduce spill code in a way similar to that is used in traditional register allocation. In this situation, we should compute the live ranges of the buffers, then spill codes can be introduced accordingly.

7.2. Applications Beyond Dataflow Machines

Note that an argument-fetching dataflow machine permits an updatable memory model in the architecture similar to conventional architecture models. The idea of applying the method developed in this paper to conventional von Neumann architectures is fully presented in Ref. 17, which can also handle loops with loop-carried dependences. In the following we give a brief insight.

In Ref. 17, we propose a framework in which register allocation for software pipelining is solved in two steps. During the first step, as in the present paper, it determines a time-optimal schedule for a software pipelined loop by allocating certain FIFO buffers to the program. An important notion here is that symbolic registers are organized as FIFO buffers, one FIFO queue for each variable defined in the loop. Intuitively, such a buffer queue is used to "extend" the lifetimes of the corresponding loop variable generated in successive iterations, permitting multiple iterations to be overlapped in concurrent executions. We show that the minimum buffer allocation and the time-optimal scheduling problem can be formulated together as an integer programming problem and a polynomial time solution is developed along a line similar to this paper. The second step is to map the symbolic registers of the FIFO buffers into physical registers. Since a schedule is already derived in the first step, a coloring algorithm can be applied to minimize the number of physical registers required to implement the buffers. Code generation schemes with or without special hardware support are also discussed. This scheme has been implemented and encouraging experiment results can be found in Ref. 18.

8. RELATED AND FUTURE WORK

8.1. Optimal Storage Allocation for Dataflow Machines

The minimum storage allocation for static dataflow computers based on an argument-flow dataflow model has been treated in Refs. 11–13. The work is mainly focused on acyclic dataflow graphs under the term *balancing*. To our knowledge, the balancing problem is first formulated as a linear programming problem and solved using network flow methods in Ref. 11. In Refs. 14 and 15, the concept of *limited balancing* is introduced, where the focus is to reduce the storage allocation if the rate of operation is bounded by some constraints external to the dataflow graphs. In Ref. 16, static dataflow graphs with cycles have been studied, and a polynomial algorithm is proposed to derive a schedule to achieve the optimal computation rate—determined by the *critical cycles* in the graph. In Ref. 4, the minimum

storage requirement for dataflow graphs with cycles has been investigated, and the target machine model is a FIFO dataflow model where an arc can hold multiple tokens. It has been shown that the problem of minimum token storage allocation for a loop to execute at optimal rate can be formulated into a linear programming problem and a polynomial time solution has been proposed. The target architecture model, however, is still based on the argument-flow dataflow model.

Loop unraveling under a pure dynamic dataflow model can initiate as many iteration as possible, limited only by data dependences.[9] A main challenge is to minimize the storage used by dynamically "unraveled" concurrent iterations. By far, the most successful scheme to control the storage requirement is the *loop bounding* scheme by Culler.[28] One limitation of this scheme is that a fixed number of storage frames (one per iteration) are allocated to a loop, and this amount of storage may not be optimal. Recently, a method of compile time loop scheduling under dynamic loop unraveling has been presented in Ref. 29.

8.2. Loop Storage Allocation for Conventional Architectures

Dataflow software pipelining is also related to conventional software pipelining which performs loop scheduling by computing a static parallel schedule to overlap instructions of a loop body from different iterations. An advantage of software pipelining is that it provides a direct way of exploiting fine-grain parallelism across all iterations of the loop. This is achieved without the explicit use of loop unrolling, and results in highly compact object codes. Software pipelining has been proposed for synchronous parallel machines as well as pipelined machines.[30–35]

However, in real life compilers, register allocation is often treated as a separate phase from the code scheduling—such as software pipelining. Two approaches have been suggested to treat the register allocation problems in conjunction with code scheduling for pipelined machine architectures. In the first approach, it is assumed that a large number of registers is available, hence the code scheduling can be handled independently of the register constraints. After the scheduling is done, the global register allocation can be performed, assuming that there will be enough registers.[36] In the second approach, register allocation is done before the scheduling phase. Such register allocation is usually done using classical algorithms such as graph coloring techniques,[2, 3] which has the goal to minimize the storage usage in a sequential program, and does not take advantage of the iterative nature of loops. Unfortunately, as pointed out in Ref. 37, the two phases have conflicting goals.[36] There has been no clear criteria on how

the two parts can be integrated under a unified compiling framework to achieve the desired goals of both time and space efficiency.[38]

In this paper, we study the problem of rate optimal execution and minimum storage allocation within the same framework. Since the instruction processing unit of an argument-fetching dataflow architecture is very much like a conventional processor architecture without a program counter, the solution of the optimal loop storage allocation problem for the former will also be useful for the latter as shown in Section 7. Bradlee *et al.*,[39] and Goodman *et al.*,[40] has studied some heuristic approaches to combine register allocation and instruction scheduling for sequential von Neumann architecture.[17, 18, 41, 42] has studied the combined approach on superscalar architectures.

8.3. Retiming Synchronous Circuits

Retiming is a circuit transformation in which registers are added at some points and removed from others in such a way that the functional behavior of the circuit as a whole is preserved. A polynomial time retiming algorithm has been proposed to solve the problem of pipelining combinatorial circuitry with smallest possible clock period and minimum register cost.[43, 44]

Although there are some similarities in the problem formulation, our computation model is very different from what is used in retiming; ours is asynchronous in nature, while the retiming model is synchronous. Therefore, the objectives and formulations are different.

9. CONCLUSIONS

In this paper, we have proposed a polynomial time solution to the optimal loop storage allocation problem for rate optimal execution under an argument-fetching dataflow architecture model. The problem is formulated into an integer programming problem. Then we proved that the constraint matrix has the totally unimodular property which implies that an integer programming problem can be reduced to a linear programming problem. A more efficient algorithm is given to transform our problem to a minimum cost flow problem, which gives an $O(n^3 \log n)$ algorithm. This work can be viewed as an extension of our work on balancing static dataflow graphs for dataflow software pipelining. In our recent work, we extend our consideration to loops with loop-carried dependences, in which the corresponding dataflow graphs will contain cycles. Preliminary results will be published in Ref. 17.

Due to the similarity of the instruction processing, our results should be useful to the design of register allocators in a compiler for a von Neumann architecture. For details, see Refs. 17 and 18.

REFERENCES

1. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers—Principles, Techniques and Tools*, Addison-Wesley Publishing Co. (1986).
2. G. J. Chaitin, Register allocation and spilling via graph coloring, *ACM SIGPLAN Symp. on Compiler Construction*, pp. 98–105 (1982).
3. G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein, Register allocation via coloring, *Computer Languages*, **6**:47–57 (January 1981).
4. G. R. Gao and Qi Ning, Loop storage optimization for dataflow machines, Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua (eds.), *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, *Proc. of the Fourth Int'l. Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, Santa Clara, California, pp. 359–373 (1992).
5. J. B. Dennis, First version of a data flow procedure language, Technical Memo MIT/LCS/TM-61, MIT Laboratory for Computer Science, Cambridge, Massachusetts (1975).
6. J. B. Dennis, Data flow for supercomputers, *Proc. of the CompCon* (March 1984).
7. Jack B. Dennis, The evolution of "static" data-flow architecture, Jean-Luc Gaudiot and Lubomir Bic, (eds.), *Advanced Topics in Data-Flow Computing*, Chapter 2, Prentice-Hall, Englewood Cliffs, New Jersey (1991).
8. J. B. Dennis and G. R. Gao, An efficient pipelined dataflow processor architecture, *Proc. of Supercomputing '88*, IEEE Computer Society and ACM SIGARCH, Orlando, Florida, pp. 368–373 (November 1988).
9. Arvind and K. P. Gostelow, The U-Interpreter, *IEEE Computer*, **15**(2):42–49 (February 1982).
10. G. R. Gao, A flexible architecture model for hybrid dataflow and control-flow evaluation, *Proc. of the Int'l. Workshop: Data-Flow, A Status Report*, Eilat, Israel (1989).
11. G. R. Gao, A pipelined code mapping scheme for static dataflow computers, Technical Report TR-371, MIT Laboratory for Computer Science (1986).
12. G. R. Gao, Algorithmic aspects of balancing techniques for pipelined data flow code generation, *Journal of Parallel and Distributed Computing*, **6**:39–61 (1989).
13. G. R. Gao, *A Code Mapping Scheme for Dataflow Software Pipelining*, Kluwer Academic Publishers, Boston, Massachusetts (December 1990).
14. G. R. Gao, H. H. J. Hum, and Y. B. Wong, An efficient scheme for fine-grain software pipelining, *Proc. of the CONPAR '90-VAPP IV Conf.*, Zurich, Switzerland, pp. 709–720 (September 1990).
15. G. R. Gao, H. H. J. Hum, and Y. B. Wong, Limited balancing—an efficient method for dataflow software pipelining, *Proc. of the Int'l. Symp. on Parallel and Distributed Comput., and Syst.*, New York, New York (October 1990).
16. G. R. Gao, Y. B. Wong, and Qi Ning, A Petri-Net model for fine-grain loop scheduling, *Proc. of the SIGPLAN '91 Conf. on Programming Language Design and Implementation*, ACM SIGPLAN. Toronto, Ontario, pp. 204–218 (June 1991).
17. Q. Ning and G. R. Gao, A novel framework of register allocation for software pipelining, *Proc. of 20th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '93)*, Charleston, South Carolina, pp. 29–42 (January 1993).

18. Qi Ning, Register Allocation for Optimal Loop Scheduling, PhD thesis, School of Computer Science, McGill University, Montreal, Canada (May 1993).
19. J. B. Dennis, Data flow supercomputers, *IEEE Computer*, **13**(11):48–56 (November 1980).
20. G. R. Gao, H. H. J. Hum, and Y. B. Wong, Towards efficient fine-grain software pipelining, *Conf. Proc., Int'l. Conf. on Supercomputing*, ACM, Amsterdam, The Netherlands, pp. 369–379 (June 1990).
21. G. R. Gao, H. H. J. Hum, and J. M. Monti, Towards an efficient hybrid dataflow architecture model, E. H. L. Aarts, J. van Leeuwen, and M. Rem, (eds.), *Proc. of PARLE '91—Parallel Architectures and Languages Europe*, Eindhoven, The Netherlands, Springer-Verlag, Lecture Notes in Computer Science, pp. 505–506 (June 1991).
22. G. Gao, R. Govindarajan, and Prakash Panangaden, Well-behaved dataflow for dsp computation, *ICASSP-92, Int'l. Conf. on Acoustics, Speech, and Signal Processing* (March 1992).
23. L. G. Khachian, A polynomial algorithm in linear programming, *Soviet Math. Doklady*, **20**:191–194 (1979).
24. N. Karmarkar, A new polynomial-time algorithm for linear programming, *Combinatorica*, **4**:373–395 (1984).
25. A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons (1986).
26. P. Camion, Characterizations of totally unimodular matrices, *Proc. Amer. Math. Soc.*, **16**:1068–1073 (1965).
27. Eugene L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Saunders College Publishing, Ft Worth, Texas (1976).
28. D. E. Culler, Managing parallelism and resources in scientific dataflow programs, PhD thesis, Technical Report TR-446, MIT Laboratory for Computer Science (1989).
29. Micah Beck, Keshav K. Pingali, and Alex Nicolau, Static scheduling for dynamic dataflow machines, Technical Report TR 90-1076, Department of Computer Science, Cornell University, Ithaca, New York (January 1990).
30. A. Aiken, Compaction-based parallelization, PhD thesis, Technical Report 88-922, Cornell University (1988).
31. A. Aiken and A. Nicolau, Optimal loop parallelization, *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation* (June 1988).
32. A. Aiken and A. Nicolau, A realistic resource-constrained software pipelining algorithm, *Proc. of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, California (August 1990).
33. K. Ebcioğlu, A compilation technique for software pipelining of loops with conditional jumps, *Proc. of the 20th Ann. Workshop on Microprogramming* (December 1987).
34. K. Ebcioğlu and A. Nicolau, A global resource-constrained parallelization technique, *Proc. of the ACM SIGARCH Int'l. Conf. on Supercomputing* (June 1989).
35. Monica Lam, Software pipelining: An effective scheduling technique for VLIW machines, *Proc. of the ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, Atlanta, Georgia, pp. 318–328 (June 1988).
36. J. R. Larus and P. N. Hilfinger, Register allocation in the SPUR Lisp compiler, *Proc. of the ACM Symp. on Compiler Construction*, Palo Alto, California, pp. 255–263 (June 1986).
37. T. R. Gross, Code Optimization of Pipeline Constraints, PhD thesis, Computing System Lab., Stanford University (1983).
38. D. Bernstein and I. Gertner, Scheduling expressions on a pipelined processor with a maximal delay of one cycle, *ACM Trans. on Programming Languages and Syst.*, **11**(1):57–66 (January 1989).
39. D. G. Bradlee, S. J. Eggers, and R. R. Henry, Integrating register allocation and instruc-

tion scheduling for RISCs, *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pp. 122–131 (April 1991).
40. J. R. Goodman and W. Hsu, Code scheduling and register allocation in large basic blocks, *Int'l. Conf. on Supercomputing*, pp. 442–452 (July 1988).
41. R. A. Huff, Lifetime-sensitive modulo scheduling, *Proc. of ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, Albuquerque, New Mexico, pp. 258–267 (June 1993).
42. S. S. Pinter, Register allocation with instruction scheduling, *Proc. of ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, Albuquerque, New Mexico, pp. 284–257 (June 1993).
43. C. E. Leiserson, Optimizing synchronous systems, Technical Memo 215, MIT Laboratory for Computer Science (1982).
44. C. E. Leiserson, *Area-Efficient VLSI Computation*, MIT Press, Cambridge, Massachusetts (1983).