In the Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993, pages 29–42. Copyright ©1993 ACM (see notice below).

A Novel Framework of Register Allocation for Software Pipelining

Qi Ning
Guang R. Gao

School of Computer Science
McGill University
Montreal, Quebec
Canada H3A 2A7
email: ning@cs.mcgill.ca gao@cs.mcgill.ca

Abstract

Although software pipelining has been proposed as one of the most important loop scheduling methods, simultaneous scheduling and register allocation is less understood and remains an open problem [28]. The objective of this paper is to develop a unified algorithmic framework for concurrent scheduling and register allocation to support time-optimal software pipelining. A key intuition leading to this surprisingly simple formulation and its efficient solution is the association of maximum computation rate of a program graph with its *critical cycles* due to Reiter's pioneering work on Karp-Miller computation graphs [29]. In particular, our method generalizes the work by Callahan, Carr and Kennedy on *scalar expansion* [6], the work by Lam on *modular variable expansion* for software pipelined loops [20], and the work by Rau et al on register allocation for modulo scheduled loops [28].

1 Introduction

Software pipelining has been proposed as one of the most important fine-grain loop scheduling methods. It determines a parallel schedule with a periodic pattern which may overlap instructions of a loop body from different iterations. Software pipelining can be applied to high-performance pipelined processor architectures, super-scalar and VLIW architectures [2, 3, 4, 12, 13, 20, 27, 31].

Although much progress has been made in finding time-optimal schedules for software pipelining, simultaneous scheduling and register allocation is less understood and remains an open problem. In terms of instruction scheduling for RISC processor architectures, it is well recognized that performing register allocation before instruction scheduling (*postpass scheduling*) may introduce new constraints due to the reuse of registers, which may limit possible reordering of the instructions, as reported in [15, 17]. On the other hands, if instruction scheduling is done before (and independent of) register allocation (*prepass scheduling*) [5], more registers than necessary may be needed, which may cause unnecessary *register spills* and severely degrade the performance of the resulting code. Warren has described a technique which applies instruction scheduling twice: once before and once after the register allocation [32].

The objective of this paper is to develop a unified scheduling-allocation framework for concurrent scheduling and register allocation. We propose a framework in which register allocation for software pipelining is solved in two steps.

Step 1: Time-Optimal Scheduling and Minimal Buffer Allocation: The first step determines the time-optimal schedule for a software pipelined loop and allocates symbolic registers organized as FIFO buffers, one buffer for each variable defined in the loop. Intuitively, such a buffer 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 called the *Optimal Scheduling and Buffer Allocation* (OSBA) problem. An efficient polynomial time solution is presented based on a transformation of the OSBA problem into *min-cost flow problem.* A key intuition leading to this surprisingly simple formulation and its efficient solution is the association of maximum computation rate of a program graph with its *critical cycles* due to Reiter's pioneering work on Karp-Miller computation graphs [29].

Step 2: Mapping buffers to physical registers: The second step is to map the symbolic registers of the FIFO buffers into physical registers. Since a schedule is derived from the solution of the OSBA problem, a coloring algorithm can be applied to minimize the number of physical registers required to implement the buffers. In particular, we can apply a recently proposed method based on coloring of *cyclic interval graphs* [16]. Code generation schemes with or without special hardware support are discussed.

We have implemented the OSBA algorithm. In experiments using our method on the working example used in Rau et. al. [28], a reduction of register usage from 28 to 21 is achieved — a 25% improvement in register use without compromising the speed of the schedule.

The general problem of allocating minimum number of registers to support a class of loop schedules of certain rate, possibly optimal rate, is proven to be NP-complete in [25]. However the two step approach proposed here is a practical way of attacking this hard problem, and it has a time complexity $O(n^3 \log n)$.

Since the register coloring methods are described in other works, our discussion centers on the buffer allocation step and the code generation schemes. We organize the subsequent sections as follows. In Section 2 we present a simple example loop to motivate the concept of buffers and the two step approach for concurrent scheduling and register allocation. In Section 3 we formulate the Optimal Scheduling and Buffer Allocation (OSBA) problem. A polynomial time solution is presented in Section 4. The proof of the theorem in this section can be found in [25, 24]. In Section 5 we address the code generation problem. In Section 6, we explain how to reduce the register requirement further through coloring algorithms for backend code generation. This is the second step in our register allocation scheme. In Section 7 we compare our approach with related works. We also include an appendix which describes the example in Rau et al's paper [28], and our solution.

2 Motivation

In this section, we motivate our work by studying register allocation for a simple loop L, shown in Table 1, under software pipelining. Although a high level language representation of the loop is chosen here, it is intended only to give a simple description of our technical framework. There is no difficulty in applying our framework to the lower level representation of the code. As shown in Table 1, the loop L contains three instructions in its body. It also contains

L :	for i = 1 to n do
s_1 :	$a[i] = X + c[i-2]$;
s_2 :	$b[i] = a[i] * F$;
s_3 :	$c[i] = a[i] + b[i]$;
	enddo;

Table 1: An example loop L.

a *loop carried dependence* of distance 2 from s_3 to s_1. For instance, the value $c[i]$ generated by s_3 in iteration i is only used two iterations later by statement s_1 in iteration $i+2$. The other data dependences in the loop are all within the same iteration. The *Data Dependence Graph (DDG)* of L is shown in Figure 1, where the positive number beside (s_3, s_1) indicates its dependence distance.

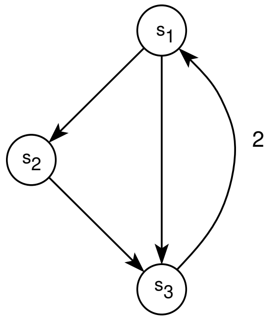

Figure 1: Data dependence graph of the example loop L.

Under software pipelining, the iterations are permitted to overlap so that the subsequent iterations may start before the previous iterations finish. Since there exist loop carried dependences, there should be a proper initiation delay interval P between successive iterations so that when the next iteration starts P cycles after the previous iteration started, no loop carried dependences are violated. In our example, assuming that the delay for Add is 1 cycle and the delay for Multiply is 2 cycles, then a delay of $P = 2$ cycles between the starting times of two consecutive iterations is optimal in the sense that the scheduled loop achieves the maximum speedup. A possible maximum speedup schedule is shown in Table 2, in which $s_{2,1}$ indicates the instruction s_2 in iteration 1.

Note in Table 2 that iteration 3 starts after iteration 1 produces c[1], and iteration 4 starts after iteration 2 produces c[2], etc. So the loop carried dependence is not violated. The schedule exploits parallelism since there are two instructions scheduled in parallel at cycle 3 (5, 7 etc.). Under the above time-optimal schedule, s_1 is executed twice before its successor s_3 is scheduled for the first time in cycle 3. Hence, in order to support the schedule, it is natural to provide a storage buffer of size more than one (if necessary) between the generator s_1 and its successors s_2, s_3. To enforce the correct order of the values produced, the buffer should behave like a first-in-first-out (FIFO) queue.

Let us examine this in some detail. Suppose that a FIFO buffer of two symbolic registers $\{a_0, a_1\}$ is allocated to a, such that a_0 is the tail and a_1 is the head of the queue. Each of the other variables is allocated a buffer of size 1, which is a single register. For convenience we will use the variable name as its register allocated if the buffer size is one. When a[1] is produced by s_1 in cycle 0, it is put into the tail a_0 of the buffer. At cycle 2, a new iteration starts and a[2] is produced before a[1] is used at cycle 3. At this moment, the new value can not be written into a_0, otherwise it would overwrite a value (a[1]) which is still needed in cycle

	iteration 1	iteration 2	iteration 3	iteration 4
0	$s_{1,1}$: $a[1]{=}X{+}c[-1]$			
1	$s_{2,1}$: $b[1]{=}a[1]{*}F$			
2		$s_{1,2}$: $a[2]{=}X{+}c[0]$		
3	$s_{3,1}$: $c[1]{=}a[1]{+}b[1]$	$s_{2,2}$: $b[2]{=}a[2]{*}F$		
4			$s_{1,3}$: $a[3]{=}X{+}c[1]$	
5		$s_{3,2}$: $c[2]{=}a[2]{+}b[2]$	$s_{2,3}$: $b[3]{=}a[3]{*}F$	
6				$s_{1,4}$: $a[4]{=}X{+}c[2]$
7			$s_{3,3}$: $c[3]{=}a[3]{+}b[3]$	$s_{2,4}$: $b[4]{=}a[4]{*}F$
8				
9				$s_{3,4}$: $c[4]{=}a[4]{+}b[4]$

Table 2: A possible maximum speedup schedule.

3. Now since we allocated two registers and organized them as a FIFO buffer, we can continue to write to the buffer at the tail at cycle 2. We assume that a mechanism is provided such that a buffer can shift its contents towards its head so that the tail (a_0) is ready to get a new value. In our case, we can assume that the old value in a_0 is shifted to a_1 at beginning of cycle 2. So at cycle 2, a[1] is in a_1 and a[2] is in a_0.

Conceptually, the pseudo code "buffer-shift" in Table 3 plays the role of such shifting. In practice, the buffer mechanism can be implemented either by explicit shifting using register moves, or by implicit shiftings based on a modulo addressing scheme. These schemes will be discussed in Section 5.

Let us see how the successors s_2, s_3 of s_1 access the buffer of a in Table 3. Since $s_{2,1}$ is scheduled at cycle 1, when it reads the value of a[1], we can assume it is still in register a_0. Therefore, the actual code for $s_{2,1}$ in iteration 1 could be $b = a_0 * F$. On the other hand, if we assume that the buffer shifts its contents at cycle 2, then at cycle 3 when $s_{3,1}$ is executed, the value of a[1] is shifted to a_1 in the buffer, hence $s_{3,1}$ should read from a_1. Therefore the code for $s_{3,1}$ should be $c = a_1 + b$. Hence $s_{2,i}$ always reads from the tail a_0 and $s_{3,i}$ always reads from the head a_1. This phenomenon is caused by the different scheduled timings of the successors of a producer. We call such a buffer *multiple-head* FIFO queue because the successors read the contents of the buffer at different places. The multiple-head buffer is illustrated in Figure 2.

Here the concept of FIFO buffer plays an important role, as it captures the notion of lifetime of a loop variable "extended" into successive iterations. The repeating pattern between cycles 2 and 3 in Table 3 is, in fact, the time-optimal schedule for the software pipelining we expect to derive. We will choose this repeating pattern as our new loop body, which is the first step of our method.

When the new loop body and the buffer allocation have been decided, we notice that the buffers allocated to individual instructions can share same physical registers if their live ranges in the schedule are non-overlapping. In our example loop L and its given schedule, the live ranges of the variables in the repeating pattern are indicated in Figure 3. Since the live ranges of F and X are the whole range of the

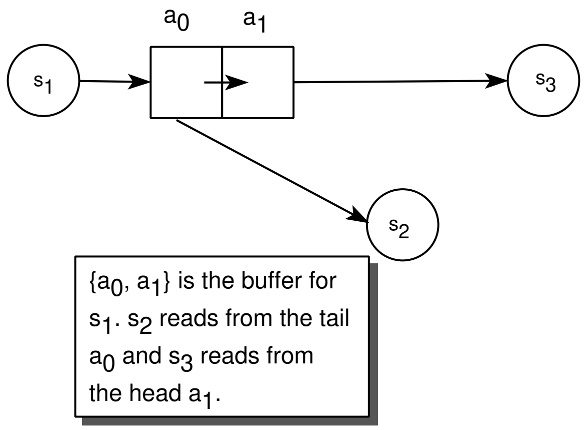

Figure 2: A multiple-head buffer.

loop, they are not shown in Figure 3.

As shown in Figure 3, the live range of c does not overlap with the live range of a_1, so they can share the same physical register. Thus, the mapping of buffers to physical registers (step 2 of our method) is somewhat similar to the traditional register allocation problem. In Section 6, we describe how to apply the register allocation method based on cyclic interval graphs for this step. After this procedure the code does not need the extra register c, which is replaced by a_1, as shown in Table 4 where the symbolic register names e.g. a_1 now represent physical registers, and where the sign $\|$ means "execute in parallel with".

3 Formulation of the Optimal Scheduling and Buffer Allocation (OSBA) Problem: Step 1

In this section we give a mathematical formulation of the scheduling and buffer allocation problem. Section 3.1 introduces the concept of periodic schedules and optimal computation rate. Section 3.2 presents an integer programming formulation of the *Optimal Scheduling and Buffer Allocation*

	iteration 1	iteration 2	iteration 3	iteration 4
0	buffer-shift: $a_1 = a_0$ $s_{1,1} : a_0 = X + c$			
1	$s_{2,1} : b = a_0 * F$			
2		buffer-shift: $a_1 = a_0$ $s_{1,2} : a_0 = X + c$		
3	$s_{3,1} : c = a_1 + b$	$s_{2,2} : b = a_0 * F$		
4			buffer-shift: $a_1 = a_0$ $s_{1,3} : a_0 = X + c$	
5		$s_{3,2} : c = a_1 + b$	$s_{2,3} : b = a_0 * F$	
6				buffer-shift: $a_1 = a_0$ $s_{1,4} : a_0 = X + c$
7			$s_{3,3} : c = a_1 + b$	$s_{2,4} : b = a_0 * F$
8				
9				$s_{3,4} : c = a_1 + b$

Table 3: Code generated for loop L.

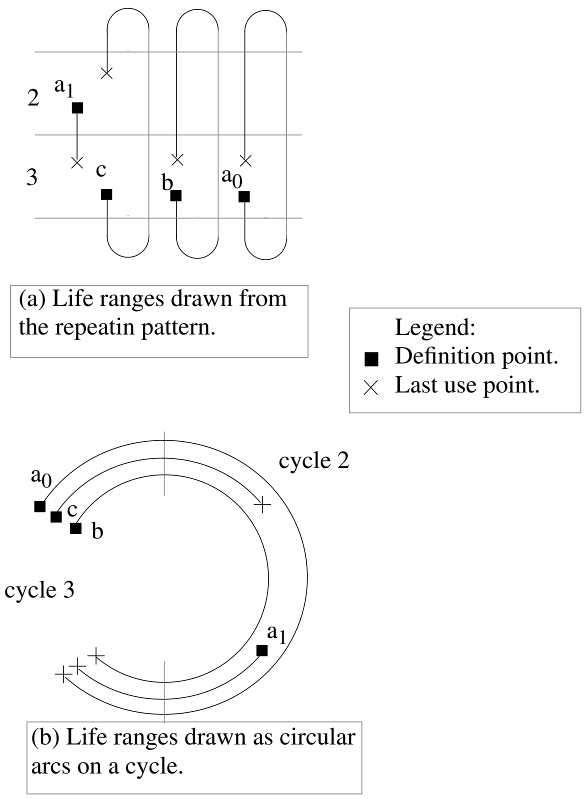

Figure 3: Live ranges of the variables for code generated by ASC scheme.

prologue code

for i = 1 to n-P do
$a_1 = a_0 \parallel a_0 = X + a_1$
$a_1 = a_1 + b \parallel b = a_0 * F$
enddo

epilogue code

Table 4: Code from repeating pattern.

(OSBA) problem.

3.1 Time-Optimal Schedules

We assume that a data dependence graph (DDG) supplemented with delay information is used to represent a loop. In a DDG, nodes represent the instructions in a program. An arc from node i to node j indicates that there is a data dependence between them. *Loop carried dependences* are indicated by arcs with positive numbers beside them indicating their dependence defined below. If node i produces a result at the current iteration and the result will be used k iterations later by node j, then we say that the arc (i,j) has a *dependence distance* k, and we use m_{ij} to indicate it. So for a data dependence not crossing iterations, the m_{ij} is 0. d_i (called delay) is used to indicate the number of clock cycles node i needs to finish its execution.

Definition 3.1 *A Timed DDG is a tuple (N, E, m, d) where N is the set of nodes, E is the set of arcs, $m = \{m_{ij}, \forall (i,j) \in E\}$ is the dependence distance vector on arc set E, and $d = \{d_i, \forall i \in N\}$ is the delay function on node set N.*

Later on in the paper, we will simply use DDG to mean Timed DDG whenever no confusion can be caused. We will consider a class of software pipelining schemes called periodic schedules.

Definition 3.2 *Let P be a positive integer. A schedule is called periodic with period P if for any node i, the times i is scheduled for its first instance, second instance, third instance,... are $t_i, t_i + P, t_i + 2P, \cdots$. Time t_i is called the initial execution time for node i.*

It is obvious that the computation rate of a periodic schedule with period P is $\frac{1}{P}$. Reiter [29] has shown that it is always possible to have an periodic schedule with optimal period P which equals the reciprocal of the maximum computation rate of the dependence graph. When a periodic schedule achieves this maximum computation rate, we call it *time-optimal*. The main objective of this paper is to find a time-optimal periodic schedule which uses a minimum number of registers. The next lemma characterizes the feasible periodic schedules.

Lemma 3.1 (Reiter [29]) *The initial execution times t_i are feasible for a periodic schedule with period P if and only if they satisfy the following inequalities:*

$$t_j \geq t_i + d_i - Pm_{ij}, \forall (i,j) \in E \tag{1}$$

where d_i is the delay of node i, P the period, and m_{ij} the dependence distance for arc (i,j).

The maximum computation rate of a DDG (loop) is the maximum average number of executions of its nodes per clock cycle, restricted only by the data dependences in the graph. For example, the loop represented in Figure 1 has a maximum computation rate of $\frac{1}{2}$, meaning that every node can be executed once in every two time cycles. The following theorem characterizes the maximum rate one can obtain. Proofs can be found in [29, 26].

Theorem 3.1 (Reiter [29]) *Given a DDG=(N, E, m, d) which represents a loop. The maximum computation rate of every node in the graph is the critical ratio,*

$$min\ \{\frac{\sum_{(i,j)\in C} m_{ij}}{\sum_{i \in C} d_i},\ \textit{for all cycles C in the DDG,}\}$$

The cycles that reach the min are called critical cycles.

The critical ratio can be computed efficiently by the algorithm of solving the minimum cost-to-time ratio problem [21]. In this paper, we assume P (the reciprocal of critical ratio) to be an integer. In the case when a fraction is derived for the period, we can always unrolled the loop a proper number of times and the resulting loop will have an integer period P, as illustrated in [25] or [24]. For loops without loop carried dependences or its DDG does not contain dependence cycles, the critical ratio is undefined, and can be considered as positive infinity. In this case we choose any positive number as period P which is always feasible. However a better choice of P should be obtained by analyzing the size of the loop body and the available resources. In general if the loop body is large and the resource is limited, the period should be larger.

3.2 Optimal Scheduling and Buffer Allocation Together

In general, there are many time-optimal schedules. One of our goals in this paper is to find the best schedule t_i such that it will need the least number of registers. In this section we look at the problem of providing the minimum number of buffers so that successive iterations can be initiated at the desired optimal rate. We do not assume that a fixed schedule is given here. Instead we will find one that can achieve both time-optimality and space-optimality. The time-optimal property is enforced by using an optimal period while not fixing the timings of the individual nodes.

A schedule may produce different numbers of results at different times during the execution. So our first objective is to minimize the maximum of the number of buffers required at different times during the execution. Let K be a given time instance. We want to know what should be the size of the buffer for node i at time K. This size depends on the timings of i's successors and also on the reservation scheme for the registers. Here we take the most conservative assumption that a register is reserved at the issue time of the instruction. However our analysis can be applied to other reservation schemes with only minor modification. For instance, if we assume that a register is reserved only at the output stage of the pipeline, then a modified formulation can be obtained. See [25, 24] for details.

Let us consider one node j of such successors, so that (i, j) is an arc from node i to node j. Suppose node i has started its k_1-th instance execution and the $(k_1 + 1)$-th instance has not been started. Hence we have the following relation:

$$t_i + P(k_1 - 1) \leq K < t_i + Pk_1. \tag{2}$$

Suppose that at time K node j has started executing its k_2-th instance, and the $(k_2 + 1)$-th instance has not been started. Hence we have the following relation:

$$t_j + P(k_2 - 1) \leq K < t_j + Pk_2. \tag{3}$$

Therefore at time K, the number of results produced by node i which have not yet been consumed by node j is $k_1 + m_{ij} - k_2$. Therefore we should allocated a buffer of size at least $k_1 - k_2 + m_{ij}$ to node i.

Next we calculate the bounds on k_1 and k_2 in terms of t_i and t_j. From inequalities (2) and (3), we can transform them into:

$$\frac{K - t_i}{P} < k_1 \leq \frac{K - t_i}{P} + 1, \tag{4}$$

and

$$\frac{K - t_j}{P} < k_2 \leq \frac{K - t_j}{P} + 1. \tag{5}$$

Let us consider (4) first. There is only one integer value in the half-open-half-closed interval:

$$(\frac{K - t_i}{P}, \frac{K - t_i}{P} + 1]. \tag{6}$$

Hence k_1 equals $\frac{K-t_i}{P} + 1$ if the latter is an integer, otherwise k_1 equals integer ceiling of the lower bound, i.e. $k_1 = \lceil \frac{K-t_i}{P} \rceil$. But in both cases, it is easy to verify that the following is always true:

$$k_1 = \lceil \frac{K - t_i + 1}{P} \rceil. \tag{7}$$

Similarly for k_2, it is always true that

$$k_2 = \lceil \frac{K - t_j + 1}{P} \rceil. \tag{8}$$

At time instance K, the number of results produced by node i which has not yet been consumed by node j is:

$$k_1 + m_{ij} - k_2 = \lceil \frac{K - t_i + 1}{P} \rceil + m_{ij} - \lceil \frac{K - t_j + 1}{P} \rceil. \tag{9}$$

Let b_i be the buffer size for node i. Then we should consider all the immediate successors of node i for the bounds in (9). When we consider all the outcoming arcs from node i, we have:

$$b_i \geq \lceil \frac{K - t_i + 1}{P} \rceil + m_{ij} - \lceil \frac{K - t_j + 1}{P} \rceil, \quad \forall (i,j) \in E, \forall K \in Z^+. \tag{10}$$

where Z^+ is the set of all positive integers.

The constraints in (10) are too complicated to handle in two aspects. First it must be true for all the positive values of K. This condition can be reduced to consider K for P consecutive values only, because of the periodicity of our schedule. Secondly, the right hand sides of the inequalities involve integral ceiling function which is not a linear function. In order to formulate the problem into an integer linear programming formulation, some approximation must be taken to deal with the integral ceiling function. Our approach here is to approximate k_1 in (7) by

$$k_1' = \frac{K - t_i}{P} + 1 \geq k_1, \tag{11}$$

and to approximate k_2 in (8) by

$$k_2' = \frac{K - t_j + 1}{P} \leq k_2. \tag{12}$$

With these approximations in (11) and (12) we can get a simplified constraints on b_i as the following:

$$\begin{array}{lll} b_i & \geq k_1' + m_{ij} - k_2' & \\ & = \frac{K-t_i}{P} + 1 + m_{ij} - \frac{K-t_j+1}{P}, & \forall (i,j) \in E, \forall K \in Z^+ \\ & = \frac{t_j - t_i - 1}{P} + m_{ij} + 1, & \forall (i,j) \in E. \end{array} \tag{13}$$

Let us notice that in (13) the time instance parameter K has disappeared. That is due to the result of the approximation. Now we are ready to formulate our optimal schedule and buffer allocation problem into an integer linear program problem by combining the constraints for feasible schedules and the constraints for buffer sizes, as follows:

Optimal Scheduling and Buffer Allocation (OSBA) Problem:

$$\begin{array}{lll} & \min \ \sum_{i \in N} b_i & \\ \text{subject to} & & \\ & P b_i + t_i - t_j \geq P(m_{ij} + 1) - 1, & \forall (i,j) \in E \\ & t_j - t_i \geq d_i - P m_{ij}, & \forall (i,j) \in E \\ & t_i, b_i \text{ integers}, & \forall i \in N. \end{array} \tag{14}$$

4 Solution of the OSBA Problem

In last section we obtained an integer programming formulation (14) of the OSBA problem. In this section we investigate its solution. Here we do a variable substitution:

$$b_i' = P b_i \tag{15}$$

and transform the formulation (14) into the following form:

OSBA Problem with Variable Substitution (15):

$$\begin{array}{lll} & \min \ \sum_{i \in N} b_i' & \\ \text{subject to} & & \\ & b_i' + t_i - t_j \geq P(m_{ij} + 1) - 1, & \forall (i,j) \in E \\ & t_j - t_i \geq d_i - P m_{ij}, & \forall (i,j) \in E \\ & b_i', t_i \text{ integers}, & \forall i \in N. \end{array} \tag{16}$$

We state the following theorem which is essential to solve (16). The proof can be found in [25, 24].

Theorem 4.1 *The constraint matrix in (16) is totally unimodular, that is to say, each of its square submatrices has a determinant equal to either 0 or 1 or -1.*

The right hand sides of (16) are all integral numbers. From linear programming theory [30] that if the constraint matrix is totally unimodular and the right hand sides are integral, then the integer programming problem can be solved as a linear programming problem. The optimal linear programming solution is guaranteed to be integral. Hence we obtain the following corollary:

Corollary 4.1 *When (16) is solved as a linear programming problem (dropping the integer requirement), the solution obtained is always integral, i.e. it is the integer programming solution of (16).*

To solve (16), we can use general linear programming algorithms like simplex methods [9], or ellipsoid method [19] or interior point methods [18]. Here we present an more efficient algorithm in next subsection.

4.1 More Efficient Algorithm for Solving OSBA

In this subsection, we show a more efficient algorithm to solve OSBA problem as expressed in (16). 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 (16) can also be solved more efficiently.

Let us first write down the linear programming dual of (16):

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

subject to

$$\sum_{j \in \delta^+(i)} \lambda_{ij} = 1, \forall i \in N \tag{17}$$

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

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

where $\delta^+(i), \delta^-(i)$ are the sets of immediate successors and immediate predecessors of i, respectively.

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

$$\sum_{(i,j)\in E} (P(m_{ij}+1)-1)\lambda_{ij} + \sum_{(i,j)\in E} (d_i - Pm_{ij})\pi_{ij}$$

$$= \sum_{(i,j)\in E} (P-1)\lambda_{ij} + \sum_{(i,j)\in E} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{(i,j)\in E} d_i \pi_{ij}$$

$$= (P-1)\sum_{i\in N}\sum_{j\in\delta^+(i)} \lambda_{ij} + \sum_{(i,j)\in E} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N}\sum_{j\in\delta^+(i)} d_i\pi_{ij}$$

$$= (P-1)\sum_{i\in N} 1 + \sum_{(i,j)\in E} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij}$$

$$= (P-1)n + \sum_{(i,j)\in E} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij} \quad (18)$$

Note that the $(P-1)n$ in (18) is a constant. The variables in the constraints can also be rearranged. After these rearrangement, the dual problem can be written in the following form:

$$\max \sum_{(i,j)\in E} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij}$$

subject to

$$\sum_{j\in\delta^+(i)} \lambda_{ij} = 1, \forall i \in N \quad (19)$$

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

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

Now we do a variable substitution:

$$g_{ij} = \pi_{ij} - \lambda_{ij}, \forall (i,j) \in E. \quad (20)$$

With this variable substitution, the objective function in (18) becomes:

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

$$= -\sum_{(i,j)\in E} Pm_{ij} g_{ij} + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} (g_{ij} + \lambda_{ij})$$

$$= -\sum_{(i,j)\in E} Pm_{ij} g_{ij} + \sum_{i\in N}\sum_{j\in\delta^+(i)} d_i g_{ij} + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \lambda_{ij}$$

$$= -\sum_{(i,j)\in E} (Pm_{ij} - d_i) g_{ij} + \sum_{i\in N} d_i \quad (21)$$

Note that the last term in (21) is also a constant. Then the formulation (19) becomes the following with the variable substitution 20:

$$\max \; -\sum_{(i,j)\in E} (Pm_{ij} - d_i) g_{ij}$$

subject to

$$\sum_{j\in\delta^+(i)} (\pi_{ij} - g_{ij}) = 1, \forall i \in N \quad (22)$$

$$\sum_{j\in\delta^+(i)} g_{ij} - \sum_{j\in\delta^-(i)} g_{ji} = 0, \forall i \in N$$

$$g_{ij} \text{ unrestricted}, \; \pi_{ij} \geq 0, \forall (i,j) \in E.$$

Formulation (22) is not yet a minimum cost flow problem. Next we show that (22) can be further reduced to a minimum cost flow problem. But let us first consider the following problem:

$$\min \sum_{(i,j)\in E} (Pm_{ij} - d_i) f_{ij}$$

subject to

$$\sum_{j\in\delta^+(i)} f_{ij} \geq -1, \forall i \in N \quad (23)$$

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

$$f_{ij} \geq -1, \forall (i,j) \in E.$$

The first set of constraints in (23) gives a lower limit on the sum of output flow for each node. The second set of constraints is the conservation restriction for the flow meaning that flow coming into a node must equal to the flow coming out of that node. If the first set of constraints have not appeared in (23), then it is the ordinary minimum cost network flow problem. We will show that how we can split the nodes in the graph to make the current formulation fitting 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 (24) is equivalent to (23).

$$\min \sum_{(u,v)\in E'} d'_{uv} f'_{uv}$$

subject to

$$\sum_{v\in\delta^+(u)} f'_{uv} - \sum_{v\in\delta^-(u)} f'_{vu} = 0, \forall u \in N' \cup N'' \quad (24)$$

$$f'_{uv} \geq -1, \forall (u,v) \in E'.$$

where we define the cost coefficients in the objective function by

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

Lemma 4.1 *Formulation (24) and formulation (23) are equivalent, that is, given an optimal solution* $\{f'_{uv}\}_{(u,v)\in E'}$ *of (24) then the* $\{f_{ij}\}_{(ij)\in E}$ *defined by the following formula is an optimal solution of (23):*

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

Similarly, given an optimal solution $\{f_{ij}\}_{(i,j)\in E}$ *of (23), the the following defined* $\{f'_{uv}\}_{(u,v)\in E'}$ *is an optimal solution of (24):*

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

The proof of the lemma is straightforward.

It is well known that the minimum cost flow can always obtain an optimal integer flow if all the capacity constraints on the arcs are integral [30]. The capacity constraints on the arcs in (23) and (24) are integral, therefore they have optimal integral solutions. Actually the efficient out-of-kilter algorithm (see [21]) and its variants will give such an optimal integer solution when it is applied to (24).

Now we are ready to proof the equivalence between (17) and (23).

Lemma 4.2 *Given an optimal integral solution of (23) we can transform it to obtain an optimal feasible solution for (17) and vice versa.*

Proof: Since (17) is equivalent to (22), we only need to show the equivalence of (22) and (23).

Let $\{f_{ij}\}_{(i,j)\in E}$ be an optimal integral solution of (23). First let $g_{ij} = f_{ij}, \forall (i,j) \in E$. Because of the first set of constraints in (23), and the fact that f_{ij}'s are integers, we have either

$$\sum_{j\in\delta^+(i)} f_{ij} = -1, \tag{25}$$

or

$$\sum_{j\in\delta^+(i)} f_{ij} \geq 0. \tag{26}$$

If (25) happens, we let $\pi_{ij} = 0, \forall j \in \delta^+(i)$. If (26) happens, we let $\pi_{ij} = 1 + g_{ij_i}$ for a particularly given $j_i \in \delta^+(i)$ and $\pi_{ij} = 0$ for all other $j \in \delta^+(i) - \{j_i\}$.

It is easy to verify that such defined $\{g_{ij}, \pi_{ij}\}$ is a feasible solution of (22). Furthermore, since π_{ij} does not appear in the objective function in (22), this solution will have the same objective value as that of (23) defined by the given f_{ij}.

Next we prove the reverse. Suppose that $\{g_{ij}, \pi_{ij}\}$ is an optimal solution of (22). Simply let $f_{ij} = g_{ij}, \forall (i,j) \in E$ will give a feasible solution of (23). Again because π_{ij} does not appear in the objective function in (22), such defined f_{ij} will give the same objective value of (23) as g_{ij}, π_{ij} gives for (22).

Since the feasible solutions for both problems have the same objective value, one of them is optimal implies that other must be optimal. □

Theorem 4.2 *The problem (16) can be solved in* $O(n^3 \log n)$ *time, where* n *is the number of nodes in the graph representing the loop.*

Proof: Recall that (17) is the dual of (16). Therefore it is equivalent to solve either of them. Lemma 4.2 proved the equivalence between (17) and (22). Lemma 4.1 proved the equivalence between (22) and (23). Therefore (17) is equivalent to (23) which is a minimum cost flow problem. Using the Out-of-Kilter method in [21] to solve the minimum cost flow problem, the algorithm for our case has a complexity $O(n^3 \log n)$. Hence our transformation procedures never use more than $O(n^3 \log n)$ time, we can conclude here that (17) can be solved in $O(n^3 \log n)$ time. □

4.2 Back Substitution

Our method reduces the problem to a minimum cost flow problem on a network, which has a $O(n^3 \log n)$ algorithm.

The t_i variables give the optimal schedule of the nodes. The b'_i variables have to be substituted back by the formula:

$$b_i = \frac{b'_i}{P}, \forall i \in N. \tag{27}$$

However such b_i's may not always be integral since we have done a divide operation in (27). If we simply round the b_i's to their integral ceilings, a suboptimal solution may result. We can resolve this problem by noticing that by this time the schedule is already produced. By fixing the schedule to be the t_i's produced by the solution of OSBA, we can recalculate the k_1 and k_2 by (7) and (8) only for P consecutive time instances of K, starting from the time it enters a steady state. It turns out that the starting time of the steady state can be the cycle time when the first iteration finishes its latest instructions. We use Q to denote the cycle time when the latest instruction finishes. Considering all the successors of node i, we should allocate a buffer of size R_i determined by the following formula (28):

$$R_i = \max \{\lceil \frac{K - t_i + 1}{P} \rceil + m_{ij} - \lceil \frac{K - t_j + 1}{P} \rceil,$$
$$\forall j \in \delta^+(i), \text{ for } K = Q, Q+1, \cdots, Q+(P-1), \} \tag{28}$$

where $\delta^+(i)$ is the set of immediate successors of i.

For our example loop in Section 2, its OSBA problem is:

$$\begin{array}{ll} & \min\ b_1 + b_2 + b_3 \\ \text{subject to} & \\ & 2b_1 + t_1 - t_2 \geq 1 \\ & 2b_1 + t_1 - t_3 \geq 1 \\ & 2b_2 + t_2 - t_3 \geq 1 \\ & 2b_3 + t_3 - t_1 \geq 5 \\ & t_2 - t_1 \geq 1 \\ & t_3 - t_1 \geq 1 \\ & t_3 - t_2 \geq 2 \\ & t_1 - t_3 \geq -3 \end{array} \tag{29}$$

Solving (29) for loop L, we obtain the following scheduling and guidelines for the buffer sizes:

$$t_1 = 0, t_2 = 1, t_3 = 3, \tag{30}$$

$$b_1 = 2, b_2 = \frac{3}{2}, b_3 = 1. \tag{31}$$

If we round up the solution for the b_i's in (31), we would end up with 5 registers. However if we use (28) to calculate the real need for the registers that support the schedule, then we can have the following allocation:

$$R_1 = 2, R_2 = 1, R_3 = 1. \quad (32)$$

which uses only 4 registers.

The actual schedule for the loop L is shown in (2) on page 31. In (2), notice that by the time node s_3 is first scheduled, its predecessor node s_1 has been executed 2 times. That is why we allocated a FIFO buffer of size 2 for node s_1.

5 Code Generation

In this section we discuss the code generation problem based on our solution of the schedule and register allocation problem in Section 4.

The unique aspect of our code generation method is the buffer registers allocated to each node as a FIFO (First-In-First-Out) queue which has multiple heads. If we allocated more than one register to a node, arranging them as a FIFO queue can make sure that the results are consumed in the same order as they are produced.

Conceptually, the new result produced by a node should always be written to the tail of the corresponding FIFO buffer, while the successors of the node should read the results at the proper places of the buffer. However, we must note that it is not always true that the successors should read the results from the same head of the FIFO buffer. In some cases it is possible a successor should read the result from a place in the FIFO buffer other than the head. In other word, the FIFO buffer should have multiple heads. The intuition is that these successor nodes are, in general, executing at different time points in the software pipelined schedule. And the buffer shifts its contents each time a new value is produced by the associated node. Therefore the successors need to read from different places of the buffer. Hence, a FIFO with multiple heads is required. Such a multiple-head buffer was illustrated in Figure 2 in Section 2.

In the rest of this section, we illustrate two schemes to generate code which implements a FIFO buffer using registers. The tradeoff of dedicated hardware architecture supports will also be discussed.

Scheme I: Access Stationary Coding (ASC). In this scheme, the FIFO buffer between a producer node and its successor nodes is directly accessed using fixed register assignment for the tail and the heads. This assignment is "stationary", and will remain the same during the entire execution. On the other hand, the data in the FIFOs are explicitly shifted each time the producer node is writing a new value to its tail. The "shifting" can be realized by issuing multiple register move instructions, or by special architecture support for register shifts.

Scheme II: Data Stationary Coding (DSC). In this scheme, instead of letting the registers in a buffer to shift their contents, we simply let the next iteration write to the next position in the corresponding FIFO buffer. Thus, data are kept *stationary*, while accesses to the registers of a FIFO buffer are performed with the modulo addressing method.

5.1 Scheme I: Access Stationary Coding

Under Access Stationary Coding (ASC) scheme, the code generated with register shifting for our example loop L is given in Table 5. In the table, at cycle time 0 (or cycles 2, 4, 6, etc) the FIFO buffer of two registers allocated to node s_1 (for variable a) shifts its contents, and a new value is written into its tail a_0. We assume that at the beginning of the cycle, all the old content are read off from the registers, and at the end of the same cycle new contents are written back into the registers. Therefore $a_1 = a_0, a_0 = X + c$ have the effect of shifting the old contents in a_0 to the register a_1 and the new result is written into the tail a_0 at the same clock cycle. It is safe to overwrite a_1 at this moment because the schedule and the supporting buffer allocation guarantee that the old content of a_1 is no longer used. We always align the shifting operation at the point when the corresponding instruction is issued.

To ensure that the successors also read the correct results from the right places, we have to calculate the positions for them to read in the FIFO buffer. We have seen the use of multiple-head buffer in Section 2. Here we give a lemma to calculate the positions for the successors to access the data from the buffers:

Lemma 5.1 *Let (i,j) be an dependence arc in the DDG, that is to say, that node i is the producer and node j is a consumer (successor). The formula to calculate the position from which node j should read in the FIFO buffer of node i is:*

$$index_{ij} = \lceil \frac{t_j - t_i}{P} \rceil + m_{ij} - 1. \quad (33)$$

Proof of Lemma 5.1: Node i writes the result to the tail of its FIFO buffer at time $t_i + (K-1) * P$ in iteration K. This result will be read by node j in iteration $K + m_{ij}$. Node j in iteration $K + m_{ij}$ is scheduled at cycle $t_j + (K-1+m_{ij}) * P$. Therefore the time difference between the production and the consumption is:

$$[t_j + (K - 1 + m_{ij}) * P] - [t_i + (K-1) * P]$$

$$= t_j - t_i + m_{ij} * P.$$

During this time interval, there are

$$\lceil \frac{t_j - t_i + m_{ij} * P}{P} \rceil - 1 = \lceil \frac{t_j - t_i}{P} \rceil + m_{ij} - 1$$

many register shiftings for the buffer allocated to node i. Also note that the above formula is independent of iteration K. Hence node j (in any iterations) should read from the buffer position indexed by the above formula which is (33). □

As an example, we use formula (33) to calculate the positions node s_2 and node s_3 read from the FIFO buffer of size 3 allocated for node s_1. For node s_2, we have

$$index_{1,2} = \lceil \frac{t_2 - t_1}{P} \rceil + m_{1,2} - 1$$

$$= \lceil \frac{1 - 0}{2} \rceil + 0 - 1$$

$$= 0,$$

so node s_2 should read from a_0

	iteration 1	iteration 2	iteration 3	iteration 4
0	buffer-shift: $a_1 = a_0$ $s_{1,1} : a_0 = X + c$			
1	$s_{2,1} : b = a_0 * F$			
2		buffer-shift: $a_1 = a_0$ $s_{1,2} : a_0 = X + c$		
3	$s_{3,1} : c = a_1 + b$	$s_{2,2} : b = a_0 * F$		
4			buffer-shift: $a_1 = a_0$ $s_{1,3} : a_0 = X + c$	
5		$s_{3,2} : c = a_1 + b$	$s_{2,3} : b = a_0 * F$	
6				buffer-shift: $a_1 = a_0$ $s_{1,4} : a_0 = X + c$
7			$s_{3,3} : c = a_1 + b$	$s_{2,4} : b = a_0 * F$
8				
9				$s_{3,4} : c = a_1 + b$

Table 5: Code generated by ASC scheme.

For node s_3, we have

$$index_{1,3} = \lceil \frac{t_3 - t_1}{P} \rceil + m_{1,3} - 1$$

$$= \lceil \frac{3-0}{2} \rceil + 0 - 1$$

$$= 1,$$

so node s_3 should read from a_1.

If we examine line by line horizontally at the code in Table 5, we can discover a repeating pattern of the code from cycles 2 to 3, as shown in Table 6.

iteration i	iteration i+1
	b-shift: $a_1 = a_0$ $s_{2,i+1} : a_0 = X + c$
$s_{3,i} : c = a_1 + b$	$s_{2,i+1} : b = a_0 * F$

Table 6: The repeating pattern from the ASC code.

We will use this repeating pattern as our new loop body. The original loop is now transformed into an new parallel loop body plus a prologue (lines 0 to 1) and an epilogue. The important fact is that the new loop body is only of P(=2) cycles, which means that in every 2 cycles a new iteration will start. That is the optimal rate we can obtain. The new parallel loop is shown is Table 7, in which the || sign means "execute in parallel with".

Generally, let

$$T = \max\ \{t_i : i \in N\},$$

then the pattern is formed from time instances $T - P + 1$ to T.

prologue code

for i = 1 to n-P do
 $a_1 = a_0 \parallel a_0 = X + c$
 $c = a_1 + b \parallel b = a_0 * F$
 enddo

epilogue code

Table 7: Code from the repeating pattern of the ASC code.

So far, we assume that the register shifting operation can be implemented using register moves (copying) in conventional architectures. However, it is also possible that a processor architecture supports register shifting directly in hardware. Such support allows the ALUs be devoted to other computation functions, thus improving performance.

5.2 Scheme II: Data Stationary Coding

The Data Stationary Coding (DSC) scheme proposed here is intended to avoid explicit register shiftings in the previous ASC scheme. Instead of letting the registers to shift their contents, we simply let the next iteration write to the next position in the corresponding FIFO buffer. For the successor nodes, we can not simply use formula (33) to calculate the positions to read in the FIFOs. Instead we must use modulo addressing according to the following lemma:

Lemma 5.2 *For a dependence arc (i,j), if in the current iteration node i is writing to Q_i (where Q_i is the index) of its buffer, then node j in current iteration should read from position:*

$$index'_{ij} = (Q_i - m_{ij})\ mod\ R_i, \tag{34}$$

where R_i is the buffer size.

Proof of Lemma 5.2: Suppose that the current iteration is K. Then node i writes to the position $(K-1) \bmod R_i$ since the data are not to be moved. In current iteration K, node j should read the result produced by node i in iteration $K - m_{ij}$. Therefore node j in current iteration should read from position $(K - m_{ij} - 1) \bmod R_i$. Substitute $K-1$ with Q_i, we obtain the formula (34). □

The code generated using modulo addressing scheme is shown in Table 8. For example, at cycle $P(i-1)$, we have the instruction $a_{(i-1) \bmod 2} = X + c$ for iteration i.

	iteration i
$P(i-1)$	$s_{1,i} : a_{(i-1) \bmod 2} = X + c$
$P(i-1)+1$	$s_{2,i} : b = a_{(i-1) \bmod 2} * F$
$P(i-1)+2$	
$P(i-1)+3$	$s_{3,i} : c = a_{(i-1) \bmod 2} + b$

Table 8: Code for one iteration generated by DSC scheme.

Its expanded version is shown in Table 9 on the next page.

In Table 9 the repeating pattern is from cycle 2 to cycle 3, which is shown in Table 10.

iteration i	iteration i+1
	$s_{1,i+1} : a_{i \bmod 2} = X + c$
$s_{3,i} : c = a_{(i-1) \bmod 2} + b$	$s_{2,i+1} : b = a_{i \bmod 2} * F$

Table 10: Repeating pattern from the DSC code.

We can see from Table 10 that the pattern derived by using the DSC scheme contains less instructions than that of the ASC scheme, which is due to the elimination of the register shifting operations. The new parallel loop body is shown in Table 11.

prologue code

for i = 1 to n-P do
$a_{i \bmod 2} = X + c$
$c = a_{(i-1) \bmod 2} + b \parallel b = a_{i \bmod 2} * F$
enddo

epilogue code

Table 11: New loop body from the DSC code repeating pattern.

6 Reduce Register Requirement Further by Coloring: Step 2

In Section 5 we showed how to generate code from a repeating pattern. That finished the first step of our register allocation scheme. At this point, the FIFO buffer sizes and the schedule are all determined. However we still have the chance to share the buffer elements if their live ranges do not overlap with each other for this fixed schedule. Hence second step of our register allocation scheme is to apply the conventional coloring algorithm(s) to further reduce the register requirement.

For each of the instructions that we allocated a buffer of size 1, the register may be thought as a symbolic register. Each such symbolic registers may be reused. For an instruction we allocated a buffer of size more than 1, the chance to reuse the registers in the buffer is only for the head register in the FIFO because all the other registers are live in the entire range of the loop cycle.

In our example loop L, suppose that we use the ASC scheme to generate code, then we choose the repeating pattern in (6), and draw the live-range diagram of the variables in Figure 3 in Section 2.

We can draw the interference graph according to the circular arcs in Figure 3, and color the interference graph with 3 colors. For instance, the following is a legal coloring with 3 colors: color_1 = $\{a_1, c\}$, color_2= $\{b\}$, color_3 = $\{a_0\}$. Therefore the actual number of registers required for the repeating pattern is 3 for these 3 colors, plus 2 extra for loop invariants X and F, which totals 5 registers. In general we can use the coloring algorithms [8, 7, 16] to obtain the minimum number of registers used in the new loop body. In this paper, we apply a recent method of cyclic interval graph coloring [16].

After the coloring algorithm the final code for the repeating pattern is shown in Table 4 in Section 2, in which c is replaced by a_1 since they have the same color.

The coloring algorithm can also be applied to the code produced by DSC scheme. However the live ranges of the registers in a buffer of size more than one may last for several repeating patterns (new iterations) because there are no explicit register-shiftings now. For instance, the live ranges of the variables in Table 11 generated by the DSC scheme is shown in Figure 4. In the picture no two variables can be colored the same color. Hence the code already uses minimum number of registers and we do not need to change the code again for this example.

7 Related Work

The early work by Aiken and Nicolau [2, 3, 4] did not consider register allocation problem. In a recent paper, Nicolau et al [23] considered the register allocation problem by renaming for the compaction-by-percolation based algorithms. Ebcioglu et al have proposed the technique of *enhanced software pipelining* with resource constraints [12, 13, 11, 22]. However, they did not consider the minimum register allocation problem as discussed in this paper.

In Lam's work on software pipelining [20], an interesting scheme called *modulo variable expansion* is proposed to al-

	iteration 1	iteration 2	iteration 3	iteration 4
0	$s_{1,1} : a_0 = X + c$			
1	$s_{2,1} : b = a_0 * F$			
2		$s_{1,2} : a_1 = X + c$		
3	$s_{3,1} : c = a_0 + b$	$s_{2,2} : b = a_1 * F$		
4			$s_{1,3} : a_0 = X + c$	
5		$s_{3,2} : c = a_1 + b$	$s_{2,3} : b = a_0 * F$	
6				$s_{1,4} : a_1 = X + c$
7			$s_{3,3} : c = a_0 + b$	$s_{2,4} : b = a_1 * F$
8				
9				$s_{3,4} : c = a_1 + b$

Table 9: Expanded code from DSC scheme.

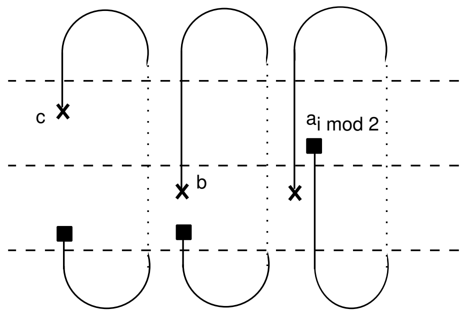

Figure 4: Life range intervals for code generated by DSC scheme.

low a scalar loop variables be expanded to use more than one location so that the unnecessary precedence constraints due to scalar variable in different iterations can be removed. However modulo variable expansion is only performed after the schedule has been fixed. The work described in in this paper can be considered as an extension to modulo variable expansion in the sense that it is incorporated in a unified framework of time-optimal scheduling, and minimizes the amount of storage for scalar variable expansion and array variable shrinkage.

Callahan, Carr and Kennedy have studied register allocation for subscripted variables. In their method, array references which are live across several iterations are recognized and a source-to-source transformation called *scalar replacement* is performed such that they can be handled by coloring-based register allocators. However, their work is aimed at sequential loop execution and does not consider loop scheduling such as software pipelining. We have shown in [25, 24] that our OSBA formulation (14) includes Callahan et al's result as a special case.

In a recent paper by Rau et al [28], a method of register allocation for software pipelining was presented. In this method, register allocation is performed after the so-called *modular scheduling phase.* Successive iterations are initiated at a fixed *initiation interval.* The register allocation problem is formulated as a bin-packing problem of *vector lifetimes* on a cylinder surface. A heuristic algorithm has been proposed for the register allocation and has been demonstrated to be quite effective by experimental results. However, the paper did not attempt to describe a complete concurrent scheduling-allocation strategy for software pipelining. In fact we have calculated the number of registers needed by our method for the example in Rau et al [28]. Our method uses 21 registers, while Rau et al claimed that 28 was needed in their case, a 25% improvement in register usage without compromising the speed. Other related work can be found in [14, 10].

8 Conclusions and Future Work

Many conventional register allocation algorithms are based on the coloring of *interference graphs* representing overlapping relations of the life ranges of program variables given a sequential execution schedule [8, 7, 1]. In this paper, we formulate the register allocation problem as part of the joint schedule-allocation problem. Intuitively, we are using register allocation as a constraint to the software pipelining scheduling process — to derive, among all time-optimal schedules, the ones which have the potential to use the minimum number of registers. Once such a schedule is fixed, we can use conventional methods to further minimize the register requirements. This paper has provided a new framework for simultaneous scheduling and register allocation for software pipelining. We plan to extend this framework to real applications in our future work. One scenario is that the number of available registers is given. Another scenario is that the loop body may contain conditionals.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison-Wesley Publishing Co., 1986.

[2] A. Aiken. Compaction-based parallelization. (PhD thesis), Technical Report 88–922, Cornell University, 1988.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[4] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[5] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 122–131, April 1991.

[6] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.

[7] G. J. Chaitin. Register allocation & spilling via graph coloring. *ACM SIGPLAN Symp. on Compiler Construction*, pages 98–105, 1982.

[8] G. J. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages 6*, pages 47–57, January 1981.

[9] V. Chvatal. *Linear Porgramming.* W.H. Freeman and Company., 1983.

[10] E. Duesterwald, R. Gupta, and M.L. Soffa. Register pipelining: An integrated approacn to register allocation for scalar and subscripted variables. Technical report, Department of Computer Science, University of Pittsburgh, 1991.

[11] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelization loops with unpredictable branches on a VLIW architecture. Technical report, IBM, 1990.

[12] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[13] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, June 1989.

[14] Christine Eisenbeis, William Jalby, Daniel Windheiser, and Francois Bodin. A strategy for array management in local memory. In *Third Workshop on Programming Languages and Compilers for Parallel Computing.* University of California, Irvine, 1990. To be published by Pitman/MIT Press.

[15] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, June 1986.

[16] L. Hendren, G.R. Gao, E. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *Lecture Notes in Computer Science 641*, pages 176–191, October 1992.

[17] J. Hennessy and T. Gross. Postpass code optimization of pipelined constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[18] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[19] L. G. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, 20:191–194, 1979.

[20] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[21] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, Ft Worth, TX, 1976.

[22] T. Nakatani and K. Ebcioglu. Using a lookahaed window in a compaction–based parallelizing compiler. In *Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitectures*, pages 57–68, 1990.

[23] A. Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on fine-grained parallelism. In U. Banerjee et al., editor, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, pages 359–373, Santa Clara, California, 1992. Springer-Verlag.

[24] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. Technical Report ACAPS Technique Memo 42, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1992.

[25] Qi Ning. *Optimal Register Allocation to Support Time-Optimal Scheduling for Loops.* PhD thesis, in preparation, School of Computer Science, McGill University, 1992.

[26] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri Nets. *IEEE Transactions on Computers*, pages 440–448, September 1980.

[27] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[28] B.R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register allocation for modulo scheduled loops: Strategies, algorithms and heuristics. In *Proceedings of SIGPLAN '92 Conf. on Programming Language Design and Implementation, San Francisco, CA*, 1992.

[29] R. Reiter. Scheduling parallel computations. *Journal of ACM*, 15:590–599, October 1968.

[30] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1986.

[31] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.

[32] H.S. Warren. Instruction scheduling for the IBM RISC System/6000 processor. *IBM J. Res. Develop.*, 34(1), January 1990.

Appendix

A Example from Rau Et Al's Paper

In this appendix, we look at the example loop given in Rau Et Al's paper [28]. The loop is shown in Table 12.

```
for i = 1 to n do
s = s + a[i]
a[i] = s * s * a[i]
enddo
```

Table 12: Example from Rau et al's paper.

Its low level representation, like 3-address code, is shown in Table 13.

a :	$vr33 = vr33 + vr32$	% vr33 is address of a[i] %
b :	$vr34 = \text{load}m(vr33)$	% vr34 = a[i]%
c :	$vr35 = vr35 + vr34$	% vr35 = s %
d :	$vr36 = vr35 * vr35$	
e :	$vr37 = vr36 * vr34$	% vr37 = new a[i] %
f :	$\text{store}(vr37, m(vr33))$	
	branch if $i \leq n$	

Table 13: Low level intermediate representation.

We will focus on the low level representation and generate code for it. The data dependence graph of the low level presentation is shown in Figure 5. The delay for Add and Store is 1, the delay for Multiply is 2 and delay for Load is 13.

There is only one directed cycle in the dependence graph, which is a self-loop from node a to a. The B-ratio of it is 1. Therefore we can generate a schedule with period of 1. However since Rau et al used 2 as their period, we will also use 2 as our period for generating the schedule and the register allocation. The OSBA formulation of the low level representation is in 35.

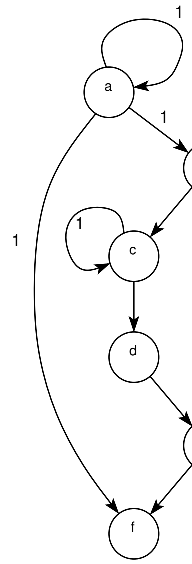

Figure 5: Data dependence graph of the intermediate representation.

$$\begin{aligned}
&\min\ b_a + b_b + b_c + b_d + b_e \\
\text{subject to}\quad & \\
& b_a + t_a - t_b \geq 4 \\
& b_a + t_a - t_f \geq 4 \\
& b_a + t_a - t_a \geq 4 \\
& b_b + t_b - t_c \geq 2 \\
& b_b + t_b - t_e \geq 2 \\
& b_c + t_c - t_d \geq 2 \\
& b_d + t_d - t_e \geq 2 \\
& b_e + t_e - t_f \geq 2 \\
& t_b - t_a \geq 13 \\
& t_f - t_a \geq -1 \\
& t_a - t_a \geq -1 \\
& t_c - t_b \geq 13 \\
& t_e - t_b \geq 13 \\
& t_d - t_c \geq 1 \\
& t_e - t_d \geq 2 \\
& t_f - t_e \geq 2
\end{aligned} \tag{35}$$

Solving (35) we obtain the following schedule and buffer allocation:

$$t_a = 1, t_b = 0, t_c = 13, t_d = 14, t_e = 16, t_f = 18; \tag{36}$$

$$b_a = \frac{21}{2}, b_b = 9, b_c = \frac{3}{2}, b_d = 2, b_e = 2. \tag{37}$$

With the technique (28) in Section 4, we obtain the following buffer allocation:

$$R_a = 10, R_b = 8, R_c = 1, R_d = 1, R_e = 1. \tag{38}$$

Since the life ranges of the variables in the repeating pattern all overlap, our second step of coloring algorithm is not necessary now. So there is a total of 21 register allocated, while Rau et al used 28 registers in [28].