$$
\begin{aligned}
& \min\ b_a + b_b + b_c + b_d + b_e + b_f \\
\text{subject to} \quad & \\
& b'_a + t_a - t_b \geq 2 \\
& b'_a + t_a - t_f \geq 2 \\
& b'_a + t_a - t_a \geq 2 \\
& b'_b + t_b - t_c \geq 0 \\
& b'_b + t_b - t_e \geq 0 \\
& b'_c + t_c - t_d \geq 0 \\
& b'_d + t_d - t_e \geq 0 \\
& b'_e + t_e - t_f \geq 0 \\
& t_b - t_a \geq -1 \\
& t_f - t_a \geq -1 \\
& t_a - t_a \geq -1 \\
& t_c - t_b \geq 13 \\
& t_e - t_b \geq 13 \\
& t_d - t_c \geq 1 \\
& t_e - t_d \geq 2 \\
& t_f - t_e \geq 2
\end{aligned}
\tag{43}
$$

Solving (43) we obtain the following solution:

$$t_a = 1, t_b = 0, t_c = 13, t_d = 14, t_e = 16, t_f = 18; \tag{44}$$

$$b'_a = 20, b'_b = 17, b'_c = 2, b'_d = 3, b'_e = 3, b'_f = 0. \tag{45}$$

With the technique (20) in Section 4, we obtain the following buffer allocation:

$$b_a = 10, b_b = 8, b_c = 1, b_d = 1, b_e = 1, b_f = 0. \tag{46}$$

Since the live 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 [31].

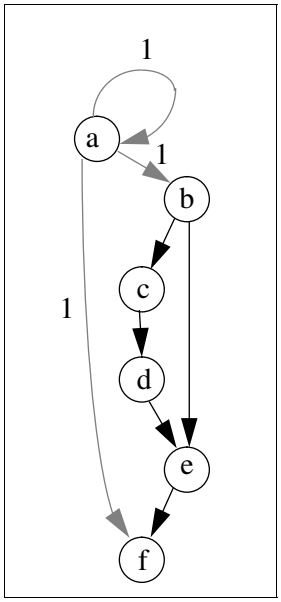

Figure 6: Data dependence graph of the low level code of Rau's example.

$$
\begin{aligned}
& \min\ b_a + b_b + b_c + b_d + b_e + b_f \\
\text{subject to}\quad & \\
& 2b_a + t_a - t_b \geq 2 \\
& 2b_a + t_a - t_f \geq 2 \\
& 2b_a + t_a - t_a \geq 2 \\
& 2b_b + t_b - t_c \geq 0 \\
& 2b_b + t_b - t_e \geq 0 \\
& 2b_c + t_c - t_d \geq 0 \\
& 2b_d + t_d - t_e \geq 0 \\
& 2b_e + t_e - t_f \geq 0 \\
& t_b - t_a \geq -1 \\
& t_f - t_a \geq -1 \\
& t_a - t_a \geq -1 \\
& t_c - t_b \geq 13 \\
& t_e - t_b \geq 13 \\
& t_d - t_c \geq 1 \\
& t_e - t_d \geq 2 \\
& t_f - t_e \geq 2
\end{aligned}
\tag{42}
$$

After variable substitution (9), we have:

C The Example from Rau Et Al's Paper

In this appendix, we look at the example loop given in Rau Et Al's paper [31]. The loop is shown below in (40).

$$\boxed{\begin{array}{l}\text{for i = 1 to n do}\\ s = s + a[i]\\ a[i] = s * s * a[i]\\ \text{enddo}\end{array}} \tag{40}$$

Its low level representation, like 3-address code, is shown below.

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

(41)

We will focus on the low level representation. The data dependence graph of the low level presentation is shown in Figure 6. 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 the following:

the total unimodularity property. However, we show next that the constraint matrix of (10) is totally unimodular. In fact, we will show that the condition in Theorem B.1 is satisfied.

Let us index the constraint matrix in the following way: the first n columns are indexed by $v'_1, v'_2, \cdots, v'_{|N|}$, and the remaining n columns are indexed by $v_1, v_2, \cdots, v_{|N|}$, where n is the number of nodes in the graph; the first m rows are indexed by $e'_1, e'_2, \cdots, e'_{|A|}$, and the next m rows are indexed by $e_1, e_2, \cdots, e_{|A|}$, where m is the number arcs in the graph. Here we assume that v'_i and v_i represent the same node i in the DDG representing the loop. A similar assumption for arcs is also true.

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

Let H be an arbitrary Eulerian square submatrix of the constraint matrix. In general, we can assume that some rows of H are indexed by some e''s and some e's. Similarly, we can assume that some columns of H are indexed by some v''s and 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) is also in H, then at the rows where p' has $+1$'s p must also have $+1$'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 that the column indexed by v_i is not in H. Since H is Eulerian, the sum of entries 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 as assumed. But H is Eulerian, and hence the sum of entries 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 entries 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 entries in row w is zero which can be divided by 4.

Since we have counted all the non-zero entries on H, we conclude that the sum of entries in H can be divided by 4. Hence by Camion's Theorem B.1, the constraint matrix of (10) is totally unimodular. □

So each row of the incidence matrix has one $+1$ and one -1 at 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 a matrix to be totally unimodular.

Definition B.3 *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 be 0 or $+1$ or -1.

Testing whether a given matrix having the TUM property is not easy in general although very complicated procedures have been developed to do that in polynomial time [33]. There are some known necessary and sufficient conditions in the literature for testing the TUM property. Here we only list one which we will use later in our proof.

Definition B.4 *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 B.1 (Camion (1965) [6]) *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 Theorem 4.1 in Section 4 that the constraint matrix in (10) 17 is totally unimodular.

Proof: The constraint matrix in (10) has very strong relations with the incidence matrix of the graph. Actually the constraint matrix is the following matrix:

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

where O are submatrices of proper sizes in which all entries are zero, U^+ is the Out-incidence matrix and U the incidence matrix.

Although it is well known that the incidence matrix of a directed graph is totally unimodular, it is not generally true that the combination of totally unimodular matrices preserves

Modified Optimal Schedule and Buffer Allocation (MOSBA) Problem:

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

subject to

$$\begin{aligned} Pb_i + t_i - t_j &\geq Pm_{ij} - d_i + 1, \forall (i,j) \in E \\ t_j - t_i &\geq d_i - Pm_{ij}, \forall (i,j) \in E \\ t_i, b_i &\text{ integer}, \quad \forall i \in N. \end{aligned} \tag{39}$$

Notice that the MOSBA formulation (39) and the OSBA formulation (8) only differ from the right hand sides of the constraints. Therefore the algorithms to solve them can be the same.

B Proof of Theorem 4.1

Definition B.1 *The O-incidence matrix A^+ of (directed) graph G is a matrix with the rows indexed by arcs and the columns indexed by nodes and the elements defined by*

$$a^+_{ne} = +1, \forall n \in N, \forall e \in E \text{ and } n \text{ is the tail of arc } e.$$

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

$$a^-_{ne} = -1, \forall n \in N, \forall e \in E \text{ and } n \text{ is the head of arc } e.$$

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 B.2 *The usual incidence matrix of graph G is just*

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

Appendices

A A Modified OSBA Problem

In this appendix, we give a modified formulation of OSBA problem (8) in Section 3, by assuming that the destination register of an instruction s_i is reserved at time $t_i + d_i - 1$, i.e. when it is at the output stage of the pipeline.

The formulation and the procedures to deduce it are almost identical to the one in Section 3. The solution of the formulation is also very similar as noted after we complete the formulation.

As in Section 3, consider an arc (i,j) in the DDG. Now since we commit a buffer to node i time instance $t_i + d_i - 1$ instead of t_i, the time span of the result value will become $t_j + Pm_{ij} - (t_i + d_i - 1)$ instead of $t_j + Pm_{ij} - t_i$.

Hence the lower bound on the number b_i of buffers for node i becomes:

$$b_i \geq \frac{t_j + Pm_{ij} - (t_i + d_i - 1)}{P}, \forall j \in \delta^+(i).$$

With these modifications, we formulate our modified optimal schedule and buffer allocation problem into an integer program problem as follows:

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

subject to

$$\begin{gathered} b_i \geq \frac{t_j + Pm_{ij} - t_i - d_i + 1}{P}, \forall (i,j) \in E \\ t_j \geq t_i + d_i - Pm_{ij}, \forall (i,j) \in E \\ t_i, b_i \ \text{ integer}, \quad \forall i \in N. \end{gathered} \tag{38}$$

In the following we rewrite the above formulation (38) so that all the variables appear on the left sides of the inequalities. We name it as *Modified Optimal Schedule and Buffer Allocation (MOSBA) Problem.*

[29] 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.

[30] 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.

[31] 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 Conference on Programming Language Design and Implementation, San Francisco, CA*, 1992. to appear.

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

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

[34] 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.

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

[18] 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*, February 1992.

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

[20] Mike Johnson. *Superscalar Microprocessor Design.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991.

[21] N. P. Jouppi and D. W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, Boston, MA, 1988.

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

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

[24] 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.

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

[26] 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.

[27] A. Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on fine-grained parallelism. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science 589, Proceedings of the 4-th International Workshop on Languages and Compilers for Parallel Computing, pages 359–373, Santa Clara, California, 1992. Springer-Verlag.

[28] Q. Ning and G. Gao. Optimal memory allocation for argument fetching dataflow machines. ACAPS Technical Memo 32, School of Computer Science, McGill University, Montreal, January 1992.

[5] 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.

[6] P. Camion. Characterizations of totally unimodular matrices. *Proc. Amer. Math, Soc.*, 16:1068–1073, 1965.

[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] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the ACM Symposium on Compiler Construction*, pages 37–47, June 1984.

[16] G.R. Gao, Y.B. Wong, and Q. Ning. A petri net model for loop scheduling. In *the Proceedings of ACM SIGPLAN'91, Toronto, Canada.* June 1991.

[17] 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.

10 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. A discussion of such extensions is left for the full paper.

11 Acknowledgment

We would like to thank Professors Laurie Hendren and David Avis for the constructive discussions with them during the development of the paper. We would also like to express our deep thank to Erik Alman who very carefully read the draft and corrected countless grammar errors.

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.

9 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 [27] 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, 26]. However, they did not consider the minimum register allocation problem as discussed in this paper.

In Lam's work on software pipelining [24], an interesting scheme called *modulo variable expansion* is proposed to allow 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 Section 8 that our OSBA formulation (8) includes Callahan et al's result as a special case.

In a recent paper by Rau et al [31], 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 [31]. Our method uses 21 registers, while Rau et al claimed that 28 was needed in their case. Our method in this case shows an improvement of 25%. Other related work can be found in [14, 10].

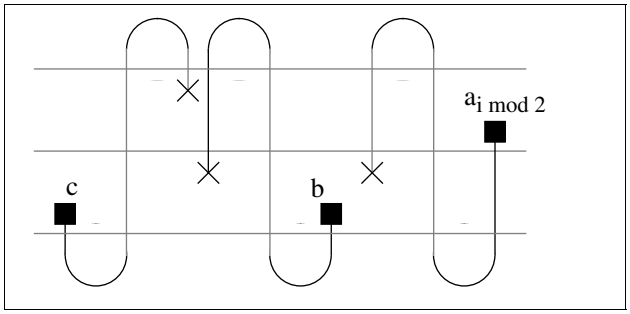

Figure 5: Life range intervals for code generated by DSC scheme.

8 Special Case

In this section we look at a special case of formulation (8), which is Callahan et al's result [5] for a fixed sequential program.

Callahan et al [5] considered the problem of using several registers for a subscripted variable to eliminate most of the loads and stores for that variable in a loop. They assume that the (sequential) execution order for the loop has already been fixed. Then they look at the dependence arcs outcoming from an instruction. They choose a number τ which is the longest dependence distance across iterations from this instruction. Then they allocate $\tau + 1$ registers to the instruction which writes to a subscripted variable. In this way they can eliminate the store of the subscripted variable in the current iteration and the load in the subsequent iterations which use this subscripted variable.

Now suppose we are given a fixed execution order for a sequential architecture. We can still think that the schedule for the sequential machine is periodic because the iterations are executed in a periodic way. However in this case the period P is the total execution time of a whole iteration. Therefore we have

$$t_i \leq P, \forall i \in N. \tag{36}$$

Therefore the lower bound on the number of buffers for node i becomes:

$$\begin{array}{rcl} b_i & \geq & \lceil \frac{t_j - t_i}{P} \rceil + m_{ij}, \forall j \in \delta^+(i) \\ & \geq & 1 + m_{ij}, \forall j \in \delta^+(i) \end{array} \tag{37}$$

which is exactly Callahan et al's formula to calculate the amount of registers needed to an instruction i.

In this sense our method is more general than theirs since we can also handle parallel schedules, and more importantly we do not fix schedule so that best schedule will be found which uses minimum amount of registers.

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 (28), and draw the live-range diagram of the variables in Figure 2 in Section 2.

We can draw the interference graph according to the circular arcs in Figure 2, 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, 18] 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 [18].

After the coloring algorithm the final code for the repeating pattern is shown in (35), in which c is replaced by a_1 since they have the same color.

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

(35)

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 is explicit register-shiftings now. For instance, the live ranges of the variables in the code (34) generated by the DSC scheme is shown in Figure 5. 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.

	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				$s_{3,4} : c = a_1 + b$

(32)

In (32) the repeating pattern is from clock cycle 2 to clock cycle 3, which is shown below in (33).

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

(33)

We can see from (33) that the pattern derived by using the DSC scheme contain 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 (34).

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

(34)

7 Reduce Register Requirement Further by Coloring: Step 2

In Section 6 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

6.2 Scheme II: Data Stationary Coding

The Data Stationary Coding (DSC) scheme proposed here is intended to avoid register shifting 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 (27) to calculate the positions to read in the FIFOs. Instead we must use modulo addressing according to the following lemma:

Lemma 6.2 *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:*

$$index'_{ij} = (Q_i - m_{ij}) \ mod \ b_i, \tag{30}$$

where b_i is the buffer size.

Proof: Suppose that the current iteration is K. Then node i writes to the position $(K-1)$ mod b_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 $(K - m_{ij} - 1)$ mod b_i. Substitute $K-1$ with Q_i, we obtain the formula (30). □

The code generated using modulo addressing is shown in (31). For example, at clock cycle $P(i-1)$, we have the instruction $a_{(i-1) \bmod 2} = X + c$ for iteration i.

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

(31)

Its expanded version is shown in (32).

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

so node s_3 should read from a_1.

If we examine line by line horizontally at the code in (26), we can discover a repeating pattern of the code from cycles 2 to 3, as shown below in (28).

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

(28)

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 uses only of P(=2) clock cycles, which means that in every 2 clock cycles a new iteration will start. That is the optimal rate we can obtain. The new parallel loop is shown is (29), in which the $||$ sign means "execute in parallel with".

prologue code

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

epilogue code

(29)

Generally, let

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

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

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.

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 6.1 *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:*

$$index_{ij} = \lceil \frac{t_j - t_i}{P} \rceil + m_{ij} - 1. \tag{27}$$

Proof: 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 (27). □

As an example, we use formula (27) to calculate the positions node s_2 and node s_3 read from the FIFO buffer of size 2 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

For node s_3, we have

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

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 by the producer and successors are performed with the modulo addressing method.

6.1 Scheme I: Access Stationary Coding

Under Access Stationary Coding (ASC) scheme, the code generated with register shifting for our example loop L_1 is given in (26). 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) shift 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.

	iteration 1	iteration 2	iteration 3	iteration 4
0	b-shift: $a_1 = a_0$ $s_{1,1} : a_0 = X + c$			
1	$s_{2,1} : b = a_0 * F$			
2		b-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			b-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				b-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				$s_{3,4} : c = a_1 + b$

(26)

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

which uses only 4 buffers.

The actual schedule for the loop L is shown in (2). 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.

6 Code Generation

In this section we discuss the code generation problem based on our solution of the schedule and registers allocation 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. 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 1 in Section 2.

In the rest of this section, we illustrate two schemes to generate code which implements the the 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.

sum of distances of it is 2. Therefore the B-ratios of the two cycles are $\frac{1}{2}$ and 1. Hence the critical B-ratio is $\frac{1}{2}$ which is also the optimal period P(=2) for our periodical scheduling.

The OSBA problem formulation (8) for loop L is in (21).

$$\begin{array}{rc} & \min b_1 + b_2 + b_3 \\ \text{subject to} & \\ & 2b_1 + t_1 - t_2 \geq 0 \\ & 2b_1 + t_1 - t_3 \geq 0 \\ & 2b_2 + t_2 - t_3 \geq 0 \\ & 2b_3 + t_3 - t_1 \geq 4 \\ & t_2 - t_1 \geq 1 \\ & t_3 - t_1 \geq 1 \\ & t_3 - t_2 \geq 2 \\ & t_1 - t_3 \geq -3 \end{array} \tag{21}$$

Then we do variable substitution (9), and obtain the following formulation:

$$\begin{array}{rc} & \min b_1 + b_2 + b_3 \\ \text{subject to} & \\ & b'_1 + t_1 - t_2 \geq 0 \\ & b'_1 + t_1 - t_3 \geq 0 \\ & b'_2 + t_2 - t_3 \geq 0 \\ & b'_3 + t_3 - t_1 \geq 4 \\ & t_2 - t_1 \geq 1 \\ & t_3 - t_1 \geq 1 \\ & t_3 - t_2 \geq 2 \\ & t_1 - t_3 \geq -3 \end{array} \tag{22}$$

Solving (22) for loop L, we obtain the following scheduling and guidelines for the buffer sizes:

$$b'_1 = 4, b'_2 = 3, b'_3 = 2. \tag{23}$$

$$t_1 = 0, t_2 = 1, t_3 = 3, \tag{24}$$

If we round up the solution for the b_i's in (23), we would end up with 5 registers. However if we use (20) to calculate the real need for the buffers that support the schedule, then we can have the following allocation:

$$b_1 = 2, b_2 = 1, b_3 = 1. \tag{25}$$

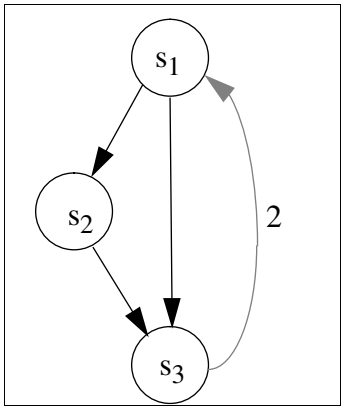

Figure 4: Data dependence graph of the example loop L.

4.2 Back Substitution

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

$$b_i = \frac{b'_i}{P}, \forall i \in N. \tag{19}$$

However such b_i's may not always be integral since we have done a divide operation in (19). If we simply round the b_i's to their integral ceilings, a suboptimal solution may result. We can solve 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 use the lower bound (6) and take the integer ceiling of it to obtain the integer lower bound on b_i. Thus we use the following formula to obtain the integer value for b_i:

$$b_i = \max \{\lceil \frac{t_j - t_i}{P} \rceil + m_{ij}, \forall j \in \delta^+(i)\}, \forall i \in N. \tag{20}$$

5 Example

In this section, we apply our OSBA procedure to the example loop given in Section 2. The loop is given in (1). Its data dependence graph is shown in Figure 4. The loop carried dependence is drawn with a dotted arc. The number beside it is the dependence distance.

There are two directed cycles in the data dependence graph. One cycle C_1 is $s_1, \rightarrow, s_2, \rightarrow, s_3, \rightarrow, s_1$. The other cycle C_2 is $s_1, \rightarrow, s_3, \rightarrow, s_1$. The length (sum of delays on the nodes along the cycle) of C_1 is 4 and the sum of distances of it is 2. The length of C_2 is 2 and the

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 (18) and formulation (17) are equivalent, that is, given an optimal solution $\{f'_{uv}\}_{(u,v)\in A'}$ of (18) then the $\{f_{ij}\}_{(i,j)\in A}$ defined by the following formula is an optimal solution of (17):*

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

Similarly, given an optimal solution $\{f_{ij}\}_{(i,j)\in A}$ of (17), the the following defined $\{f'_{uv}\}_{(u,v)\in A'}$ is an optimal solution of (18):

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

The proof of the lemma is straightforward, and is omitted.

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 [33]. The capacity constraints on the arcs in (17) and (18) are integral, therefore they have optimal integral solutions. Actually the efficient out-of-kilter algorithm (see [25]) and its variants will give such an optimal integer solution when it is applied to (18).

Theorem 4.2 *The problem (10) 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 (11) is the dual of (10). Therefore it is equivalent to solve either of them. We have has a series transformations to transform (11) to (17). Lemma 4.1 stated the equivalence between (17) and (18). Therefore (11) is equivalent to (18) which is a minimum cost flow problem. Using the Out-of-Kilter method in [25] tor 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 (11) can be solved in $O(|N|^3 \log |N|)$ time. □

the flow coming into a node must equal to the flow coming out of that node. If the first set of constraints (17b) have not appeared in (17), 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''. See Figure 3 for the illustration of splitting a node i.

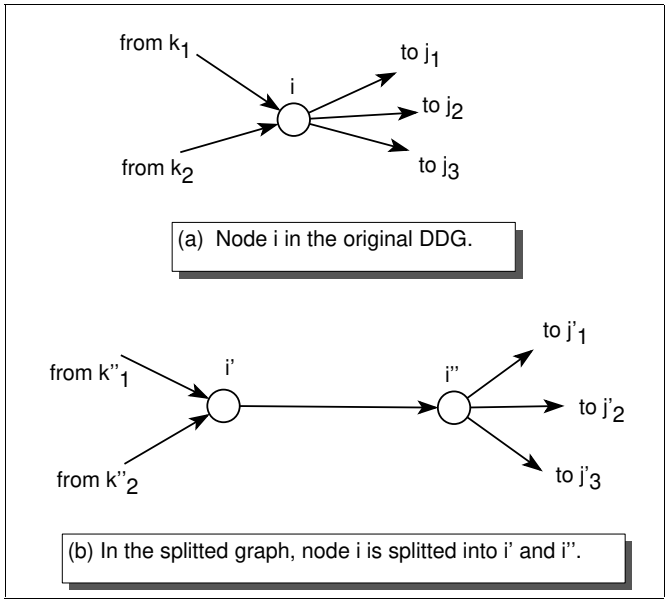

Figure 3: How node i is split into i' and i''.

Now consider the ordinary minimum cost flow problem on the result graph. Let N' be the set of i' nodes and N'' be the set of i'' nodes. We use A' to denote the set of arcs in the result graph.

It is easy to see that the following minimum cost flow problem (18) is equivalent to (17).

$$\min \sum_{(u,v)\in A'} d'_{uv} f'_{uv} \tag{18a}$$

subject to

$$\sum_{v\in\delta^+(u)} f'_{uv} - \sum_{v\in\delta^-(u)} f'_{vu} = 0, \forall u \in N' \cup N'' \tag{18b}$$

$$f'_{uv} \geq -1, \qquad \forall (u,v) \in A'. \tag{18c}$$

$$\max \; -\sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} \tag{16a}$$

subject to

$$\sum_{j\in\delta^+(i)} (\pi_{ij} - f_{ij}) = 1, \qquad \forall i \in N \tag{16b}$$

$$\sum_{j\in\delta^+(i)} f_{ij} - \sum_{j\in\delta^-(i)} f_{ji} = 0, \;\; \forall i \in N \tag{16c}$$

$$f_{ij} \geq -1, \;\; \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{16d}$$

Formulation (16) is not yet a minimum cost flow problem. Later we show that (16) can be further reduced to a minimum cost flow problem. Now let us notice that in (16) the objective function does not contain the variables π_{ij}, which only appear in constraints (16b). Therefore we can simplify constraints (16b) by the following technique. (16b) can be rewritten as

$$\sum_{j\in\delta^+(i)} f_{ij} \geq \sum_{j\in\delta^+(i)} \pi_{ij} - 1$$

and since π_{ij} are nonnegative variables and they do not appear in neither the objective function nor the constraints (16c), we can see that the above inequality is equivalent to the following:

$$\sum_{j\in\delta^+(i)} f_{ij} \geq -1$$

Hence formulation (16) is equivalent to the following (17).

$$\min \sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} \tag{17a}$$

subject to

$$\sum_{j\in\delta^+(i)} f_{ij} \geq -1, \qquad \forall i \in N \tag{17b}$$

$$\sum_{j\in\delta^+(i)} f_{ij} - \sum_{j\in\delta^-(i)} f_{ji} = 0, \forall i \in N \tag{17c}$$

$$f_{ij} \geq -1, \qquad \forall (i,j) \in A \tag{17d}$$

The first set of constraint (17b) gives a lower limit on the sum of output flow for each node. The second set of constraints (17c) is the conservation law for the flow meaning that

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{(i,j)\in A} d_i \pi_{ij} \tag{12b}$$

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} \sum_{j\in\delta^+(i)} d_i \pi_{ij} \tag{12c}$$

$$= \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij} \tag{12d}$$

The variables in the constraints can also be rearranged. After these rearrangement, the dual problem can be written in the following form:

$$\max \sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij} \tag{13a}$$

subject to

$$\sum_{j\in\delta^+(i)} \lambda_{ij} = 1, \qquad \forall i \in N \tag{13b}$$

$$\sum_{j\in\delta^+(i)} (\lambda_{ij} - \pi_{ij}) - \sum_{j\in\delta^-(i)} (\lambda_{ji} - \pi_{ji}) = 0, \quad \forall i \in N \tag{13c}$$

$$\lambda_{ij} \geq 0, \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{13d}$$

Now we do a variable substitution:

$$f_{ij} = \pi_{ij} - \lambda_{ij}, \forall (i,j) \in A. \tag{14}$$

With this variable substitution, the objective function in (13a) becomes:

$$\sum_{(i,j)\in A} Pm_{ij}(\lambda_{ij} - \pi_{ij}) + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \pi_{ij} \tag{15a}$$

$$= - \sum_{(i,j)\in A} Pm_{ij} f_{ij} + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} (f_{ij} + \lambda_{ij}) \tag{15b}$$

$$= - \sum_{(i,j)\in A} Pm_{ij} f_{ij} + \sum_{i\in N} \sum_{j\in\delta^+(i)} d_i f_{ij} + \sum_{i\in N} d_i \sum_{j\in\delta^+(i)} \lambda_{ij} \tag{15c}$$

$$= - \sum_{(i,j)\in A} (Pm_{ij} - d_i) f_{ij} + \sum_{i\in N} d_i \tag{15d}$$

Note that the last term in (15d) is also a constant and can be discarded. Then the formulation (13) becomes the following with the variable substitution (14):

Corollary 4.1 *When (10) 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 (10).*

To solve (10), we can use general linear programming algorithms like simplex methods [9], or ellipsoid method [23] or interior point methods [22]. We present an more efficient algorithm below which reduces the problem to a minimum cost flow problem on a network, which has a $O(n^3 \log n)$ algorithm.

4.1 More Efficient Algorithm for Solving OSBA

In this subsection, we show a more efficient algorithm to solve OSBA problem after variable substitution, which is (10). 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 (10) can also be solved more efficiently.

Let us first write down the linear programming dual of (10):

$$\max \sum_{(i,j)\in A} \{Pm_{ij}\lambda_{ij} + (d_i - Pm_{ij})\pi_{ij}\} \tag{11a}$$

subject to

$$\sum_{j\in\delta^+(i)} \lambda_{ij} = 1, \qquad \forall i \in N \tag{11b}$$

$$\sum_{j\in\delta^+(i)} (\lambda_{ij} - \pi_{ij}) - \sum_{j\in\delta^-(i)} (\lambda_{ji} - \pi_{ji}) = 0, \forall i \in N \tag{11c}$$

$$\lambda_{ij} \geq 0, \pi_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{11d}$$

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:

$$\sum_{(i,j)\in A} \{Pm_{ij}\lambda_{ij} + (d_i - Pm_{ij})\pi_{ij}\} \tag{12a}$$

Optimal Scheduling and Buffer Allocation (OSBA) Problem:

$$\begin{aligned} & \min \ \textstyle\sum_{i\in N} b_i \\ \text{subject to} & \\ & Pb_i + t_i - t_j \geq Pm_{ij}, \forall (i,j) \in E \\ & t_j - t_i \geq d_i - Pm_{ij}, \forall (i,j) \in E \\ & t_i, b_i \text{ integers}, \quad \forall i \in N. \end{aligned} \tag{8}$$

4 Solution of the OSBA Problem

In last section we obtained an integer programming formulation (8) of the OSBA problem. In this section we investigate its solution.

Here we do a variable substitution:

$$b'_i = Pb_i \tag{9}$$

and transform the formulation (8) into the following form:

OSBA Problem with Variable Substitution (9):

$$\begin{aligned} & \min \ \textstyle\sum_{i\in N} b'_i \\ \text{subject to} & \\ & b'_i + t_i - t_j \geq Pm_{ij}, \quad \forall (i,j) \in E \\ & t_j - t_i \geq d_i - Pm_{ij}, \quad \forall (i,j) \in E \\ & b'_i, t_i \text{ integers}, \qquad \forall i \in N. \end{aligned} \tag{10}$$

In [28], we proved the following theorem which is also used here to solve (10). The proof can also be found in Appendix B.

Theorem 4.1 *The constraint matrix in (10) 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 (10) are all integral numbers. From linear programming theory [33] 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 and the optimal solution is guaranteed to be integral. Hence we obtain the following corollary:

read these results. Let i be a node in DDG which will produce result data. We want to know how many buffers we should allocate to a node i. This number depends on the scheduling time 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 is shown is Appendix A.

Let us consider one node j of such successors, so that (i, j) is an arc from node i to node j.

When node i is scheduled for the first time, it is t_i. If the dependence distance between i and j is m_{ij}, then the result value produced by node i will be consumed by node j at time $t_j + Pm_{ij}$. Therefore the time span of the result value is at least $t_j + Pm_{ij} - t_i$. During this period of time, node i will be scheduled at p clock cycles. All these new results produced by node i have to be saved in buffers so that node j in later iterations can read them. Hence the number b_i of buffers for node i should satisfy the following inequality, in which j belongs to the set of immediate successors of node i:

$$b_i \geq \frac{t_j + Pm_{ij} - t_i}{P}, \forall j \in \delta^+(i). \tag{6}$$

Now we combine the timing constraints (5) for feasible schedules, stated in Lemma 3.1 and the buffer size constraints in (6). Together these two sets of constraints define all the feasible schedules with optimal speedup and all supporting buffer allocation schemes. Then we want to minimize the total sum of buffers among all these feasible schedules. Putting all together, we obtain the following integer program problem:

$$\begin{array}{ll} & \min \sum_{i \in N} b_i \\ \text{subject to} & \\ & b_i \geq \frac{t_j - t_i}{P} + m_{ij}, \quad \forall (i,j) \in E \\ & t_j \geq t_i + d_i - Pm_{ij}, \quad \forall (i,j) \in E \\ & t_i, b_i \ \text{ integers}, \quad \forall i \in N. \end{array} \tag{7}$$

In the following we rewrite the above formulation (7) so that all the variables appear on the left sides of the inequalities. We name it as *Optimal Schedule and Buffer Allocation (OSBA) Problem.*

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

$$t_j \geq t_i + d_i - Pm_{ij}, \forall (i,j) \in E \tag{5}$$

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

For some P the above system (5) may not have any feasible solution, for some others it has. The restriction for P to guarantee a feasible solution in the above system is that it is not less than the reciprocal of the critical B-ratio above in Definition 3.2. [1] The critical B-ratio can be computed efficiently by the algorithm of solving the minimum cost-to-time ratio problem (page 94, [25]). In this paper, we always assume P (the reciprocal of B-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 first and the result loop will have an integer period P. For example, if the critical B-ratio is $\frac{U}{V}$, with U and V relative prime integers, then we can unroll the loop body U times. the resulting loop now has a critical B-ratio $\frac{1}{V}$. Then the reciprocal of $\frac{1}{V}$ is the integer V. So the unrolled loop has a optimal period V. Or, we can round up the fraction to the nearest integer value for P but the optimal rate will be suffered. For example, if the critical B-ratio is $\frac{2}{5}$, then P can be 3, 4, 5, etc. If we take P = 3, and the result schedule will have a rate 3 cycles per iteration which is half cycle slower than the optimal rate.

3.4 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.

We will allocate a set of buffers for each node in DDG, so that the buffers are organized as FIFO queues and they can retain the results for several iterations before the consumers

[1] If the loop contains no l.c.d.'s, then the graph contains no directed cycles. In this case B-ratio is not defined, so that P can be taken as 1, which is the maximum possible speedup.

3. $\frac{M(C)}{D(C)}$ *is called the B(alance)-ratio of C.*

4. *Cycles with minimum B-ratio are called critical cycles.*

5. *Critical B-ratio is the B-ratio of a critical cycle.*

The following theorem characterizes the maximum rate one can obtain. Proofs can be found in [32], [16] or [29].

Theorem 3.1 (Reiter [32]) *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* $\frac{M(C^*)}{D(C^*)}$*, where* C^* *is a critical cycle in the DDG.*

3.3 Periodic Scheduling

The loop scheduling problem based on software pipelining can be formulated mathematically as an integer programming problem. In this and the next subsections, we extend the formulation to the problem of register allocation. In this subsection, we focus on the problem of *periodic scheduling* as defined below.

Definition 3.3 *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 [32] 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 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.

With periodic scheduling, the number of registers needed for different nodes may vary according to different assignment of the initial execution times t_i of the nodes. The number of registers to be allocated for a node i should be enough to hold the number of results generated by node i that have not been consumed by its successors in the process of computation.

Our first step is to characterize the set of all feasible periodic schedules. Using the notations in Definition 3.3, the following lemma gives the constraints on the feasible periodic schedules. The proof can be found in Reiter [32].

3.1 Timed Data Dependence Graph

We assume that a data dependence graph (DDG) supplemented with timing 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 such that node j can not be scheduled until node i has finished.

The *loop carried dependences* may cause cycles to occur in the DDG. We use dotted arcs to indicate loop carried dependences. For example, 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 of distance k, and we use m_{ij} to indicate it. So for a data dependence not crossing the iterations, the m_{ij} is always 0. For each node i, we use d_i (called delay) to indicate the number of 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,* $m = \{m_{ij}, \forall (i,j) \in E\}$ *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 is caused.

3.2 Maximum Computation Rate

The maximum computation rate of a DDG (loop) is the maximum average number of executions of its nodes per time cycle, restricted only by the data dependences in the graph. If a loop has no l.c.d., then the maximum computation rate is one since once a node is scheduled for the first time it can be scheduled consecutively cycle after cycle without worrying about the results of the previous iterations. However if the loop contains l.c.d., then the maximum rate could be $\frac{2}{3}$, $\frac{4}{9}$, etc. depending on the cycle lengths and the configuration of the dependence distances on the arcs in the graph.

Definition 3.2 *Given a DDG G = (N, E, m, d). Let* $\mathcal{C}(G)$ *be the set of all directed cycles in G. For any directed cycle* $C \in \mathcal{C}(G)$,

1. $M(C) = \sum_{(i,j) \in C} m_{ij}$.
2. $D(C) = \sum_{i \in C} d_i$.

So far, we are assuming that the buffers are represented by symbolic registers. 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 and its given schedule, the live ranges of the variables in the repeating pattern are indicated in Figure 2. Since the live ranges of F and X are the whole range of the loop, they are not shown in Figure 2. As shown in Figure 2, the live range of c does not intersect with the live range of a_1. So we can share the same physical register between c and a_1. Thus, the mapping of buffers to physical registers (step 2 of our method) is somewhat similar to the traditional register allocation problem. In Section 7, 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 below where the symbolic register names e.g. a_1 now represent physical registers.

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

(4)

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. In Section 3.1, we introduce *timed data dependence graph* as our representation of the loops to be studied in this paper. In Section 3.2, the maximum computation rate of a loop is given, based on a result from Reiter [32]. In Section 3.3, we introduce the basic notion of periodic scheduling for software pipelining. Section 3.4 presents an integer programming formulation of the *Optimal Scheduling and Buffer Allocation* (OSBA) problem.

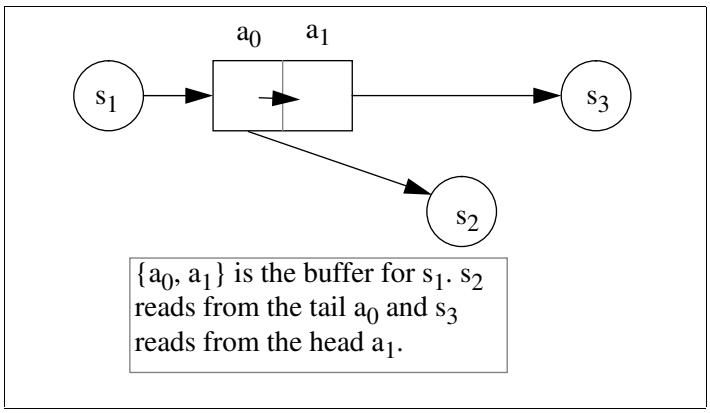

Figure 1: A multiple-head buffer.

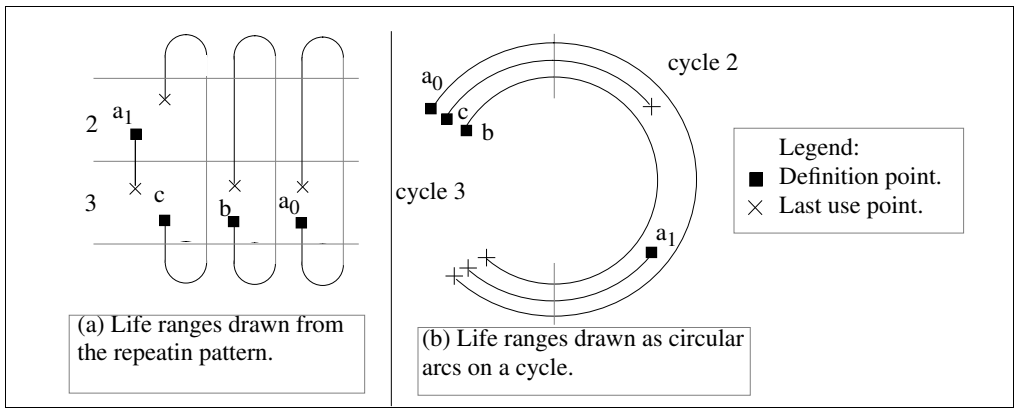

Figure 2: The live ranges of the variables for code generated by the ASC scheme.

buffer. The following pseudo code may be generated to ensure the correct accesses of the buffer positions, in which we have used the second index to indicate the iteration, i.e. $s_{2,1}$ means node s_2 in iteration 1.

	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				$s_{3,4} : c = a_1 + b$

(3)

Let us see how the successors s_2, s_3 of s_1 access the buffer of a. 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 $c[1] = a_1 + b$. 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 1.

The concept of FIFO buffer plays an important role here, as it captures the notion of lifetime of a loop variable "extended" into successive iterations. In Section 3, we formulate, as the first step of of method, the optimal scheduling and buffer allocation problem by relating the schedule of each generator and its successors to the size of the corresponding FIFO buffer allocated to the generator. The solution of the problem is a time-optimal schedule which uses minimal amount of buffers among all time-optimal schedules. In our example, the repeating pattern between cycles 2 and 3 in (3) is, in fact, the time-optimal schedule for software pipelining we expect to derive.

	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				$s_{3,4}$: $c[4]=a[4]+b[4]$

(2)

Note 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, conceptually 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. Here we assume these registers are symbolic registers. 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[0] 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 ($a[0]$) which is still needed in cycle 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. But before we do that we have to assume that the buffer has a mechanism to shift its contents towards its head so that the tail (a_0) is ready to get 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, the $a[1]$ is in a_1 and a[2] is in a_0.

At the code generation phase, we should generate appropriate code to implement the FIFO addressing mechanism. For example, to implement the FIFO buffer mechanism allocated for a, we have to figure out how the successors obtain the correct values of a[i] at the

different locations for different iterations. In Section 7, we show how to reduce the register requirement further through coloring algorithms for backend code generation. This is the second phase in our register allocation scheme. In Section 8 we show that Callahan et al's result [5] can be viewed as a special case of OSBA formulation. In Section 9 we compare our approach with related works.

2 Motivation

In this section, we motivate our work by studying register allocation for a simple loop L 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 below, loop L contains three instructions in its body.

$$\begin{array}{|l|l|}\hline L: & \text{for i = 1 to n do} \\ s_1: & a[i] = X + c[i-2]; \\ s_2: & b[i] = a[i] * F; \\ s_3: & c[i] = a[i] + b[i]; \\ & \text{enddo;} \\ \hline \end{array} \quad (1)$$

L contains 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. For instance, s_2 reads $a[i]$ which is produced by s_1 in the same iteration. And s_3 reads $a[i]$ and $b[i]$ produced by s_1 and s_2 in the same iteration.

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, we have to work out 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 below:

Step 1: Optimal Scheduling and 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 [32].

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* [18]. Code generation schemes with or without special hardware support are discussed.

The method developed in this paper is applicable to machines such as VLIW (Very Lang Instruction Word) [15], superscalar [20] and superpipelined architectures [21]. We have implemented the OSBA algorithm. In experiments using our method on the working example used in Rau et. al. [31], a reduction of register usage from 28 to 21 is achieved — a 25% improvement!

Since register coloring method is well understood, 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 stated in Section 4, and the proof and algorithm is provided in Appendix B. We return to our motivating example in Section 5 to illustrate the formulation and solution of the corresponding OSBA problem. In Section 6, we proposed two schemes of code generation for our (OSBA) scheme. One of the schemes is to shift the registers to simulate the effect of FIFO (First-In-First-Out) buffer. This scheme is called *Access Stationary Code* (ASC) where the accessing mode is fixed. The other scheme uses self-modifying code (if hardware supports) to eliminate the burden of register shifting. This second approach is called *Data Stationary Code* (DSC) where the data are not moved but the they have to be written into

1 Introduction

The efficient exploitation of fine-grain parallelism in loops has been a major challenge in the design of optimizing compilers for high-performance computer architectures. 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, as well as super-scalar and VLIW architectures [2, 3, 4, 12, 13, 24, 30, 34, 35].

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 the instruction scheduling (*post-pass scheduling*) may introduce new constraints due to the reuse of registers, which may limit possible reordering of the instructions, as reported in [17, 19]. On the other hands, if the instruction scheduling is done before (and independent of) register allocation (*prepass scheduling*), 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 [35].

The objective of this paper is to develop a unified scheduling-allocation framework for concurrent scheduling and register allocation, which is quite different from the conventional approach. The goal of the conventional register allocation is to minimize the number of registers under a given schedule. For example, many of these 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]. The objective of this paper is to develop a unified scheduling-allocation framework for concurrent scheduling and register allocation.

Our method is also different from other work on register allocation for loop variables with or without software pipelining. In particular, our method generalized the work by Callahan, Carr and Kennedy on *scalar expansion* as a register allocation method for subscript variables, the work by Lam on *modular variable expansion* for software pipelined loops [24], and the work by Rau et. al. on register allocation for modulo scheduled loops [31]. A comparison with related work is outlined in Section 9.

We propose a framework in which register allocation for software pipelining is solved in two steps.

Register Allocation for Time-Optimal Loop Scheduling

Qi Ning
Guang R. Gao

School of Computer Science
McGill University
Montreal, Quebec
Canada H3A 2A7
email: ning@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. The objective of this paper is to develop a unified scheduling-allocation framework for concurrent scheduling and register allocation to support time-optimal software pipelining. Register allocation for software pipelining is solved in two steps. The first step, called *Optimal Scheduling and Buffer Allocation*, determines the time-optimal schedule for a software pipelined loop and allocates symbolic registers organized as FIFO buffers, one 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 which is called *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 [32]. The second step is to map the symbolic registers of the FIFO buffers into physical registers. Since a time-optimal 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. Code generation schemes with or without special hardware support are discussed. In particular, our method generalized the work by Callahan, Carr and Kennedy on *scalar expansion* as a register allocation method for subscript variables, the work by Lam on *modular variable expansion* for software pipelined loops [24], and the work by Rau et al on register allocation for modulo scheduled loops [31].

List of Figures

1 A multiple-head buffer. 11

2 The live ranges of the variables for code generated by the ASC scheme. . . . 11

3 How node i is split into i' and i''. 21

4 Data dependence graph of the example loop L. 23

5 Life range intervals for code generated by DSC scheme. 32

6 Data dependence graph of the low level code of Rau's example. 43

10 Conclusions and Future Work **34**

11 Acknowledgment **34**

A A Modified OSBA Problem **38**

B Proof of Theorem 4.1 **39**

C The Example from Rau Et Al's Paper **42**

Contents

1 Introduction **6**

2 Motivation **8**

3 Formulation of the Optimal Scheduling and Buffer Allocation (OSBA) Problem: Step 1 **12**

3.1 Timed Data Dependence Graph 13

3.2 Maximum Computation Rate 13

3.3 Periodic Scheduling 14

3.4 Optimal Scheduling and Buffer Allocation Together 15

4 Solution of the OSBA Problem **17**

4.1 More Efficient Algorithm for Solving OSBA 18

4.2 Back Substitution 23

5 Example **23**

6 Code Generation **25**

6.1 Scheme I: Access Stationary Coding 26

6.2 Scheme II: Data Stationary Coding 29

7 Reduce Register Requirement Further by Coloring: Step 2 **30**

8 Special Case **32**

9 Related Work **33**

McGill University
School of Computer Science

Register Allocation for Time-Optimal Loop Scheduling

(Version 2.0)

ACAPS Technical Memo 42
January 7, 1993

Qi Ning
Guang R. Gao

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

Advanced Computer Architecture and Program Structures Group