[33] 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.

[34] S Sakai and et al. An architecture of a dataflow single chip processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 46–53, Israel, 1989.

[35] I. Watson and J. Gurd. A practical data flow computer. *IEEE Computer*, 15(2):51–57, February 1982.

[36] T. Yuba and et al. Sigma-1: A dataflow computer for scientific computations. *Computer Physics Communications*, 37:141–148, 1985.

[20] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[21] G. R. Gao. An efficient hybrid dataflow architecture model. *Submitted for publication,* Journal of Parallel and Distributed Computing, January 1990.

[22] G. R. Gao, H. H. J. Hum, and Y. B. Wong. An efficient scheme for fine-grain software pipelining. In *Proceedings of the CONPAR '90–VAPP IV Conference*, Zurich, Switzerland, September 1990.

[23] 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.

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

[25] T.R. Gross. *Code Optimization of Pipeline Constraints.* PhD thesis, Computing System Lab., Stanford University, 1983.

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

[27] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 1984.

[28] R. M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multiprocessing system. In *AFIPS Conference Proceedings, vol. 48*, pages 613–622, 1979.

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

[30] J. R. Larus and P. N. Hilfinger. Register allocation in the SPUR Lisp compiler. In *Proceedings of the ACM Symposium on Compiler Construction*, pages 255–263, Palo Alto, CA, June 1986.

[31] E. Lawler. *Combinatorial Optimization Networks and Matroids.* Holt, Rinehart, and Winston, 1976.

[32] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the Seventeenth Annual International Symposium of Computer Architecture, Seattle, WA*, pages 82–91, 1990.

[6] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, MA, 1988.

[7] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, January 1989.

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

[9] D. E. Culler. Managing parallelism and resources in scientific dataflow programs, Ph.D thesis. Technical Report TR-446, Laboratory for Computer Science, MIT, 1989.

[10] J. B. Dennis. First version of a data flow procedure language. Technical Report MIT/LCS/TM-61, Laboratory for Computer Science, MIT, 1975.

[11] J. B. Dennis. Data flow for supercomputers. In *Proceedings of the 1984 CompCon*, March 1984.

[12] J. B. Dennis. Evolution of the static dataflow architecture. In *Advanced Topics in Dataflow Computing.* Prentice-Hall, 1991.

[13] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Proceedings of the Supercomputing '88 Conference*, pages 368–373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[14] J. B. Dennis, G. R. Gao, and K. W. Todd. Modeling the weather with a data flow supercomputer. *IEEE Transactions on Computers*, C-33(7):592–603, 1984.

[15] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *The Second Annual Symposium on Computer Architecture*, pages 126–132, January 1975.

[16] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 1972.

[17] L. R. Ford and D. R. Fulkerson. *Flow in Networks.* Princeton University Press, Princeton, NJ, 1962.

[18] D.R. Fulkerson. An out-of-kilter method for minimal cost flow problems. *J. SIAM*, 1961.

[19] G. R. Gao. A pipelined code mapping scheme for static dataflow computers. Technical Report TR-371, Laboratory for Computer Science, MIT, 1986.

may limit possible reordering of the instructions, as reported in [24, 26]. As pointed in [25], the two phases often have conflicting goals [25]. There has been no clear criteria on how the two parts can be integrated under a unified compiling framework to achieve the desired of both time and space efficiency [7].

We believe that the limited balancing method is a promising first step toward a solution to such problems in the context of loop scheduling. A compiler may use the procedure developed here to estimate the minimum amount of register space required for a loop. The loop can be scheduled when there is enough register reserved during the compilation. We expect that the number of registers on a processor chip will increase continuously in the future, and such a techniques may well be feasible.

7 Conclusions

In this paper, we have outlined a new scheme for determining the minimum loop storage allocation to support maximum computation rate under dataflow software pipelining. A limited balancing scheme has been presented to perform the storage allocation for a class of loops. We demonstrated that the algorithm of limited balancing can be performed in polynomial time. We argue that our method provides a new basis to handle loop storage allocation for both static and dynamic dataflow models. A comparison with other related work in loop scheduling and storage allocation is presented.

References

[1] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[2] Arvind and et al. The tagged token dataflow architecture (preliminary version). Technical report, Laboratory for Computer Science, MIT, Cambridge, MA., August 1983.

[3] Arvind and K. P. Gostelow. The U-Interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[4] Arvind, K. P. Gostelow, and W. Plouffe. *An Asynchronous Programming Language and Computing Machine*. Department of Information and Computer Science, University of California, Irvine, December 1978.

[5] Arvind and R. A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 426–436, 1983.

Theorem 5.1 *The linear relaxation* **Problem I** *of the minimum memory requirement* **Problem Main** *can be solved in polynomial time* $O(|E|^2 \log |N|)$.

The solution for problem (I) can be round-up to give an integer solution for the **Problem Main**. In our example of Figure 3, the optimal solution of the circulation flow problem will be that the amount of flow on each arc of the graph is 1. Therefore problem (II) will get an optimal solution:

$$z_{C_{A(B)DA}} = z_{C_{ACA}} = z_{C_{CD(E)C}} = 1.$$

This solution can be transformed to a solution of problem (I) as

$$x_{DA} = 1, x_{DC} = 1, \text{and } x_{CA} = \tfrac{2}{3}.$$

When round-up to an integer solution, it is

$$x_{DA} = 1, x_{DC} = 1, \text{and } x_{CA} = 1,$$

which is optimal.

Although we have shown that **Problem I** can be solved in polynomial time, the integer version of it **Problem Main** remains open to be solved in polynomial time.

6 Related Work

In this section, we briefly compare our method with other related work.

The method developed in this paper has addressed the limitations of the loop storage management of both static and dynamic dataflow models, and see Section 2 for more information.

The limited balancing method developed in this paper provides a framework where the loop scheduling and storage allocation are studied under a unified framework.

In real life compilers, register allocation is treated as a separate phase from the code scheduling itself. For example, two approaches have been suggested to treat the register allocation problems in conjunction with code scheduling for pipelined machine architectures. In the first approach, it is assumed that a large number of registers are available, hence the code scheduling can be handled independently of the register constraints. After the scheduling is done, the global register allocation can be performed using graph coloring methods, assuming there will be enough registers [30]. In the second approach, register allocation is done before the scheduling phase. This may introduce new constraints which

First let us notice that for such defined $\{z_c\}$, the following property holds:

$$\sum_{C \ni e} z_c = f_e, \forall e \in E.$$

For any arc $e \in E^-$, the constraint for e in problem (II) demands that the sum of the z_c's for the cycles C containing e is bounded above by 1. This is true by the property stated in the above equation since $f_e \leq 1$ for $e \in E^-$.

Now the objective value of such defined feasible $\{z_c\}$ is

$$\sum_{C \in \mathcal{C}} b_c z_c = \sum_{C \in \mathcal{C}} \left\{ K(C) \frac{P(C^*)}{K(C^*)} - \sum_{e \in C} m_e \right\} z_c = \sum_{C \in \mathcal{C}} \left\{ \sum_{e \in C} L_e \frac{P(C^*)}{K(C^*)} - \sum_{e \in C} m_e \right\} z_c$$

$$= \sum_{C \in \mathcal{C}} \sum_{e \in C} p_e z_c = \sum_{e \in E} \sum_{C \ni e} p_e z_c = \sum_{e \in E} p_e \sum_{C \ni e} z_c = \sum_{e \in E} p_e f_e$$

So the objective value of problem (II) is the same as that of problem (III).

On the other hand, if $\{z_c\}$ is a feasible solution of (II), then it is easy to check that the flow defined by the following formula is a feasible solution to (III):

$$f_e = \sum_{C \ni e} z_c, \forall e \in E.$$

Of course, if one of the feasible solution is optimal, the other is also optimal since they will produce the same objective value. □

Therefore (II) and (III) are equivalent in the sense that the solution of one problem also gives the solution of the other. We can see from our formulation that (III) has $2|N|$ constraints and $|E|$ variables. Both of these two numbers are in polynomial to the original size of the graph. As a matter of fact, the circulation flow problem (III) can be solved by a polynomial time algorithm [18, 16, 31] much better than the general methods of simplex algorithm, ellipsoid algorithm [29] or the Karmarkar's algorithm [27].

Lemma 5.2 *Circulation flow* **Problem (III)** *can be solved in* $O(|E|^2 \log |N|)$ *time [31], where* $|E|$ *is the number of arcs of the graph and* $|N|$ *number of nodes in the graph.*

The final step is to solve problem (I). Since the dual of (I) is (II), and (II) has a polynomial time algorithm which will return an optimal solution for (II), the optimal solution for (I) can be read off from the tableau constructed using the simplex method in polynomial time [8]. Hence we have proved the following theorem.

The general circulation flow problem is very similar to the minimum cost flow problem. The distinction is that in minimum cost flow problem, it has a source node and it has a sink node. And the objective is to send a fixed amount of flow from the source to the sink so that certain objective function is minimized. Where as in circulation flow problem, it has no source node and sink node. So flow will just circulate in the graph. Now let us consider the circulation flow problem formulation that fits our need. The problem is that given a graph, find a (circular) flow so that certain objective function is optimized (maximized or minimized). The formulation of the circulation flow problem is:

Problem (III)

$$\max \sum_{e \in E} p_e f_e$$

subject to

$$\sum_{e \in \delta^+(n)} f_e - \sum_{e \in \delta^-(n)} f_e = 0, \forall n \in N$$

$$f_e \leq 1, \forall e \in E^-$$

$$f_e \geq 0, \forall e \in E$$

where p_e is the cost coefficients which is defined as $p_e = L_e \frac{P(C^*)}{K(C^*)} - m_e$, $\delta^+(n)$ is the set of arcs going out from n, $\delta^-(n)$ is the set of arcs going into n, f_e is a variable which indicates the amount of flow along the arc e, and $E = E^+ \cup E^-$.

Lemma 5.1 *Given a (feasible) optimal solution of (III), we can construct a (feasible) optimal solution of (II), and vice versa.*

Proof of Lemma: Let $f^1 = \{f_e^1\}$ be a feasible solution of (III). Let $S(f^1)$ be the support set of f^1, that is, the subset of arcs $\{e\}$ such that $f_e^1 > 0$. Choose any directed cycle C_1 in $S(f^1)$. Define

$$z_{c_1} = min\{f_e^1; e \in C_1\},$$

and

$$f_e^2 = \begin{cases} f_e^1 - z_{c_1}, & \text{if } e \in C_1, \\ f_e^1, & \text{if } e \notin C_1. \end{cases}$$

It is easy to check that $f^2 = \{f_e^2\}$ is another feasible circulation flow of (III). But the number of arcs in $S(f^2)$ is strictly less than that in $S(f^1)$. We can repeat the above procedure to define another z_{c_2} and $\{f^3\}$ etc, until the support set of the circulation flow is empty.

By that time, we have defined $z_c > 0$ for several cycles. We define $z_c = 0$ for all the other directed cycles in E. We show next that such defined $\{z_c\}$ is a feasible solution of (II).

subject to

$$\frac{\sum_{e \in C^-} x_e + \sum_{e \in C^+} m_e}{K(C)} \geq \frac{P(C^*)}{K(C^*)}, \forall C \in \mathcal{C}$$

$$x_e \geq 0, \quad x_e \text{ integer}, \forall e \in E^-$$

where $\mathcal{C}$ is the set of all cycles in S-DFSP-C, $C^- = C \cap E^-$, $C^+ = C \cap E^+$, and C^* is a critical cycle in the original DFSP (a critical cycle can be found in $O(|N|^3 \log |N|)$ time [31]).

The problem can be reformulated into the following form:

Problem (Main)
Original problem:

$$min \sum_{e=(n_i,n_j) \in E^-} x_{ij}$$

subject to

$$\sum_{e \in C^-} x_e \geq b_c, \forall C \in \mathcal{C}$$

$$x_e \geq 0, \forall e \in E^-$$

$$x_e \text{ integer}, \forall e \in E^-$$

Problem (I)
Its linear relaxation:

$$min \sum_{e=(n_i,n_j) \in E^-} x_{ij}$$

subject to

$$\sum_{e \in C^-} x_e \geq b_c, \forall C \in \mathcal{C}$$

$$x_e \geq 0, \forall e \in E^-$$

Problem (II)
Its linear dual:

$$\max \sum_{C \in \mathcal{C}} b_c z_c$$

subject to

$$\sum_{C \ni e} z_c \leq 1, \forall e \in E^-$$

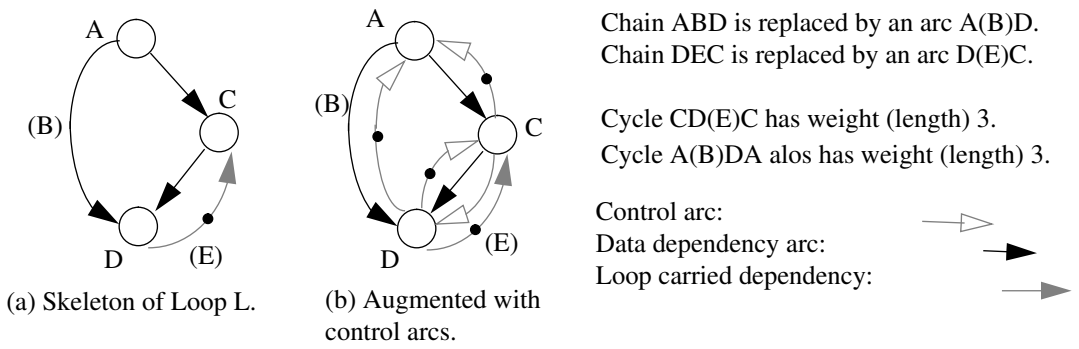
$$z_c \geq 0, \forall C \in \mathcal{C}$$

where $b_c = K(C)\frac{P(C^*)}{K(C^*)} - \sum_{e \in C} m_e$.

We should notice that there could be an exponential number of cycles in the S-DFSP-C (or in other words, an exponential number of constraints in the primal problem). Therefore the dual problem could have an exponential number of variables. The consequence is that the methods for solving the linear programming are expensive and may not directly applicable to solve **Problem I** or **Problem II**.

5 A Polynomial Time Solution

In this section, we show that the primal problem of the linear program **Problem I** can be solved in polynomial time. We will show that the dual problem **Problem II** is equivalent to the so-called *circulation flow problem* [17]. Since circulation flow problem can be solved in polynomial time [16, 31] we need to show that the optimal solution of the circulation flow problem can be translated into an optimal solution of the dual problem *in polynomial time*.



(a) Skeleton of Loop L.

(b) Augmented with control arcs.

Figure 3: A example of DFSP and its augmented S-DFSP-C

E^- has a length of 1 which reflect the timing assumption we made in Section 3 that each node will use one time step to take a token from its input arc and is ready to take the next token after this step. The augmented (by control arcs) graph is called S-DFSP-C. See Figure 3 for an example.

DFSPs have the property that during their execution the sum of tokens on any cycle does not change [23]. Each arc we just added in, together with its original dependence arc, forms a cycle with a sum of $x_{ji} + m_{ij}$ tokens on it, where m_{ij} is the number of original tokens on the arc e_{ij}. Therefore the number of tokens arc e_{ij} can hold in the S-DFSP-C is at most $x_{ji} + m_{ij}$. From here we can see that the total memory required for the execution of the DFSP is

$$\sum_{e=(n_i,n_j)\in E^-} x_{ij} + \sum_{e=(n_i,n_j)\in E^+} m_{ij}.$$

where the second sum in the above formula is a constant.

The introduction of the additional arcs also introduces many more new cycles in the S-DFSP-C that do not appear in the original DFSP. The B-ratio of the new cycles might be smaller than that of the original critical cycles if the x_{ij}'s are not properly chosen. In order to maintain the maximum computation rate we must keep enough tokens (or equivalently allocate enough memory spaces) in all cycles such that the B-ratio of the new cycles is not smaller than that of the original critical cycles. The following is the mathematical formulation of the minimum memory allocation problem to maintain the maximum computation rate.

$$\min \sum_{e=(n_i,n_j)\in E^-} x_{ij}$$

4.1 Simplify a DFSP with Chain Replacement

Let's first state what is a chain.

Definition 4.1 *Given a DFSP, if a node n has only one input arc and only one output arc, then it is called simple. A path is called a chain if all the nodes lying internally in the path (i.e. not including the two end nodes) are simple.*

Our method will allocate storage to a DFSP on a chain by chain bases. For instance, consider a chain CH of length m. Our algorithm may assign a total of n storage cells to the chain CH. Conceptually, the n storage cells will be shared by the tokens traveled along CH and operated on by all operations along the chain. This is very important, as the minimum value n may be less than m. As shown by our example in Figure 2 (b), the chain ABD has length $m = 2$, and is assigned $n = 1$ storage cell.

For simplicity in later formulation, we can replace each chain by an arc.

Definition 4.2 *Given a DFSP, and a chain Q in it. Let n_i be the starting node of chain Q, and n_j the end node of Q. A chain replacement of Q is to replace Q by a new arc joining n_i and n_j. The weight of the new arc is the sum of the weights of arcs along the chain Q. This procedure is called the chain replacement.*

After the chain replacement, we get a new DFSP in which no node is simple. We will call this new DFSP, denoted by S-DFSP, the *skeleton* of the old DFSP.

4.2 Linear Programming Formulation

After we do the chain replacement for a given DFSP we get the skeleton. In the following we will use S-DFSP to denote the skeleton.

To limit the number of memory spaces allocated on the arcs in the S-DFSP, we introduce a new storage "control arc" for each old arc in the DFSP. A control arc will have the effect to limit the number of tokens that can reside on an arc at any moment. But control arcs are only used for the purpose of calculating the amount of memory that should be allocated to each individual arc in the original DFSP. In actual execution of the DFSP, control arcs do not exist.

For an arc $e_{ij} = (n_i, n_j) \in E^+$, add a control arc $e'_{ji} = (n_j, n_i)$ with x_{ji} initial tokens (which is a variable to be determined later). The set of control arcs will be denoted by E^- as they are in the opposite direction of their corresponding dependence arcs. Each arc in

Definition 3.2 *Given a DFSP graph G = (N, D, LC; M,L). Let $\mathcal{C}(G)$ be the set of all directed cycles in G. For any directed cycle $C \in \mathcal{C}(G)$,*

1. $P(C) = \sum_{e \in C} m_e$.
2. $K(C) = \sum_{e \in C} L_e$.
3. $\frac{P(C)}{K(C)}$ *is called the B(alance)-ratio of C.*
4. *Cycles with minimum B-ratio are called critical cycles.*

We state the following theorem about the computation rate. The reader can find its proof in [23] or [33].

Theorem 3.1 *Given a DFSP graph G = (N, D, LC; M,L) which represents a loop. The maximum computation rate (after an initial transient period) of every node in the graph is $\frac{P(C^*)}{K(C^*)}$, where C^* is a critical cycle in the DFSP.*

In our example loop L in Figure 1 (c), there is only one cycle $CDEC$, therefore it is also a critical cycle. The number of tokens on it is 1. Its length is 3. Therefore the B-ratio is $\frac{1}{3}$. This cycle will dominate the computation rate of the whole graph so that every node will be fired once at a maximum rate of every 3 time steps. Although node A is not on the cycle, it can not be fired every time step.

4 The Limited Balancing Problem

From Theorem 3.1 in the last section we know that the maximum computation rate of a loop is dominated by its critical cycles. Therefore, there is no need to allocate more storage to other non-critical cycles beyond what is bounded by the B-ratio of the critical cycles (the critical B-ratio). In this paper, we propose that a compiler should be able to determine the storage allocation such that all cycles have the same balancing ratio as that of the critical cycle. For historical reasons, we call this procedure *limited balancing* [22].

In this section we will show how to formulate the limited balancing problem into a linear programming problem. There are two steps in the formulation process. The first step is called *chain replacement*, i.e. replace each chain in the DFSP with a single arc. The second step is to derive a linear programming formulation which will optimize the memory requirements. The solution of the linear programming problem will be the subject of Section 5.

2. *M is a function $M : LC \cup D \rightarrow Z$, where Z is the integer set. We will also use m_e to denote $M(e)$. For each arc $e \in LC \cup D$, m_e is the number of initial tokens on arc e.*

3. *L is a length function on the arcs. If an arc e comes out of a node n, the L_e is the execution time of node n.*

Thus, an arc in a DFSP can hold multiple tokens and there is no *a priori* bounds on the number of tokens that arc can hold. This implies that appropriate amount of storage must be allocated to each arc to accommodate the tokens which may present.

We assume that a DFSP G is operating under the following initial conditions: (1) the input/output environment for G is *maximally loaded*, i.e., it can produce input values for G and consume the output values from G without any delay (this will allow us to use a simple representation of the dataflow graph for loops in which the select actors and loop bound test actor can be omitted); and (2) the loop will run through a sufficient large number of iterations, thus the start-up and termination time can be ignored.

Finally, in a dataflow graph, a conditional expression is represented by the so-called *well-formed conditional dataflow graph* [20]. Switch and merge nodes are used in the implementation. These nodes have special firing rules which depend on run time decisions. To overcome these problems, the firing rules of the switch nodes and the merge nodes are altered to produce and consume *dummy tokens* on their unselected branches. A detailed discussion can be found in [23]. Under this treatment, the switch and the merge nodes have the same firing rule as the other regular nodes. Hence, a conditional dataflow graph can be handled as an ordinary DFSP.

3.3 Maximum Computation Rate

Given a DFSP representing a loop, the computation rate of an actor in this loop under some scheduling scheme is the average number of firings of the actor over a long period of time. Assuming that we have an *idealized dataflow machine*, i.e., the resources of the machine are unlimited, we can achieve the maximum computation rate by adopting *the earliest firing rule* for the scheduling. The earliest firing rule of a DFSP is to fire an actor as soon as all its operands have arrived (all of its input FIFO's are not empty). Obviously, under the earliest firing rule of the idealized machine model, a DFSP will achieve its maximum computation rate.

An important observation is that: the maximum computation rate of the entire DFSP is determined by the maximum possible computation rate of the nodes on "slowest cycles" (known as *critical cycles*). To formally state this result, let us first introduce the following definition.

3.1 The FIFO dataflow model

We consider an arc in the dataflow graph as a pool of memory locations which can hold data tokens. In particular, if the loop does not contain recursive function calls, we may assume that the pool is a FIFO queue. In our present paper, we don't handle loops that contain function calls, so the FIFO model is enough for us.

The firing rule of an actor in the FIFO model is:

- The actor is enabled if each input FIFO of the actor contains at least one token.
- An enabled actor can be fired and the firing of an actor will remove one token from the head of each input FIFO and put one token at the tail of each of the output FIFOs.

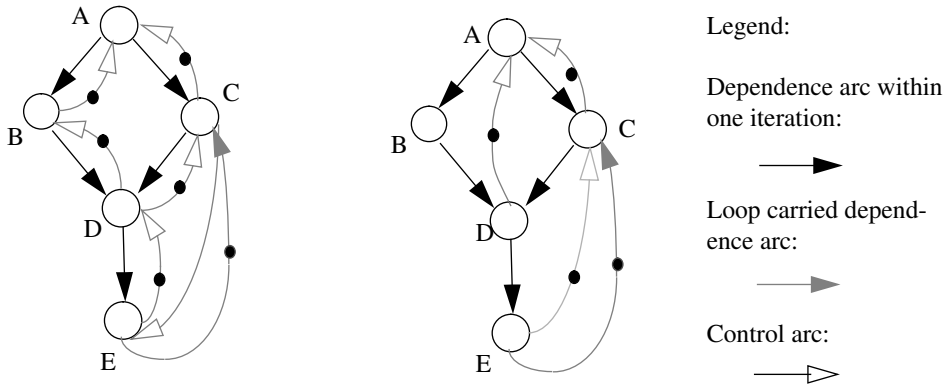
When an actor is fired, it removes the input token and starts the current activation. Meanwhile, the actor is ready to begin the next activation after certain number k of time steps. For simplicity, we assume that $k = 1$ in this paper. However, the results can be generalized straightforwardly to the cases where k is greater than 1. As long as the input queues are not empty, an actor can continue to be fired more than once, thus allow multiple activations in concurrent execution. This essentially has the effect of *loop unraveling* in dynamic dataflow model, but the unraveling is done in a "fine-grain" manner — i.e. the synchronization between activations of different iterations is performed at each individual actor, instead of at the boundary of individual iterations.

3.2 Dataflow Software Pipeline (DFSP)

In this section, we will define the class of loops known as the *dataflow software pipelines* (DFSP). The class of loops which are of interest are non-nested loops. The loops may contain both loop-independent and loop-carried dependencies.

Definition 3.1

1. *A DFSP is a directed graph (N, D, LC; M, L), where N is the set of the nodes representing the actors, D the set of arcs representing data dependencies within one iteration (which will be represented by solid arcs in the diagrams), and LC the set of arcs representing the loop carried dependencies (which will be represented by dotted arcs in the diagrams). For convenience, we will use E^+ to denote the union of D and LC, that is $E^+ = D \cup LC$.*



(a) Static dataflow graph with acknowledgment arcs.

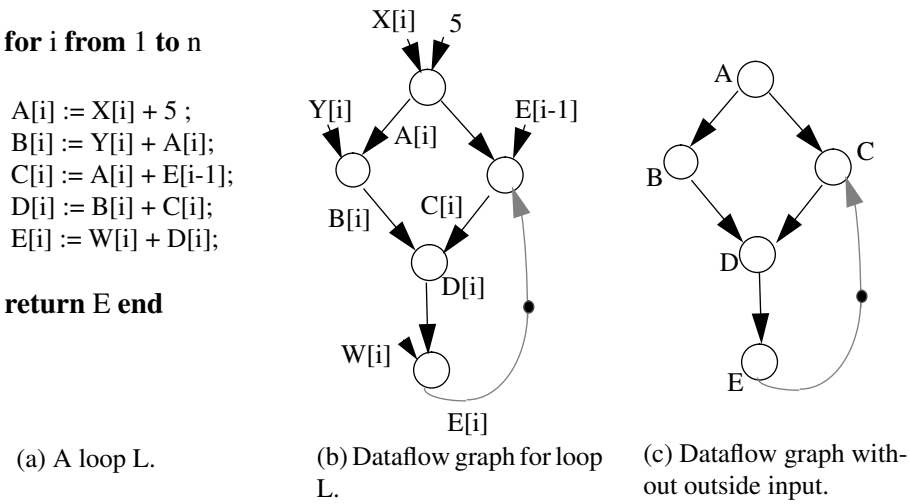
(b) Minimum memory allocation pattern.

Figure 2: Minimize Storage Allocation

Before we proceed, let us give an intuitive explanation of the method to be developed in the rest of this paper. Recall that the one-token-per-arc memory requirement for static data-flow graph is implemented by adding an acknowledgment arc for each pair of successor and predecessor nodes. There will be an initial token on such an acknowledgment arc, if it is the acknowledgment arc of an arc in D, denoting the fact that one storage cell is required by this arc. If the acknowledgment arc is for a loop carried dependency arc, then there is no initial tokens on this acknowledgment arc. The result graph is shown in Figure 2 (a). We can see from Figure 2 (a) that there are more cycles in the graph which are newly formed by the addition of the acknowledgment arcs. A key observation is that we can reduce the memory allocated to other (non-critical) cycles without decreasing the maximum computation rate dominated by the critical cycle ($CDEC$). For example, the cycles C_1 (ABA), and C_2 (BDB) are not critical cycles, we can reduce their storage allocation as shown in Figure 2 (b). The optimal computation rate remains unchanged, but the total memory allocation of the loop is reduced by 40%!

3 The Framework

In this section, we develop the FIFO dataflow model and state a maximum computation rate result without proof (proof can be found in [23] or [33]).



(a) A loop L.

(b) Dataflow graph for loop L.

(c) Dataflow graph without outside input.

Figure 1: Dataflow graph for loop L

The solid arcs in the dataflow graph represent the data dependencies within one iteration, and the dotted arcs represent loop carried data dependencies across the iterations. There are some initial tokens on the dotted arcs. The number of the initial tokens on a dotted arc corresponds to the *dependence distance* [6].

Dataflow software pipelining was originally proposed to exploit fine-grain parallelism in loops on static dataflow computers [20, 19]. A main limitation of static dataflow model is that it may not be able to fully exploit the parallelism in the loop. Under dynamic dataflow model, such limitation is eliminated via *loop unraveling* [1]. As pointed out in [9], however, exploiting more parallelism will invariably increase the resource requirement of a program. The challenge is not to allow the loop to consume more storage than necessary to fully exploit the parallelism in a loop.

Let us ask the following question: what is the minimum amount of storage that the loop (L) needs to run at the *maximum computation rate?* Let's illustrate this by the example loop L. Note that, the cycle C^* in L, $CDEC$, contains the loop carried data dependence arc. The maximum computation rate of L is dominated by C^*, which is the so-called *critical cycle* of L [23]. Thus the maximum computation rate of L is $1/3$, where we assumed that the execution time for each actor is 1 unit of time. In the static dataflow computation model, each solid arc in the graph need one memory location. Therefore, **five** memory locations are allocated for the loop L. However, **three** memory locations are enough to achieve the maximum computation rate, thus reducing the memory requirement by 40%. The rest of this paper will develop a procedure to achieve the minimization of the storage requirement.

proposed: the static dataflow model [15] and the tagged-token model [5, 35]. The dynamic model has evolved into the development of the "modern" dynamic dataflow machines such as the Monsoon dataflow machine at MIT and the SIGMA-I/EM-4 dataflow machines in Japan [32, 34, 36]. Meanwhile some "modern" static dataflow architecture proposals can be found in [21, 12].

One of the long standing issues in loop execution on dataflow machines is how to manage the fine-grain parallelism and the storage requirement supporting such parallelism. Under static dataflow model, the storage for a loop is completely determined at compile-time — simply the storage required for one instance of the loop body. A main restriction of this approach is that it may not be able to fully exploit the parallelism in the loop, although *dataflow software pipelining* has been proposed to organized the code such that several iterations may be proceeding simultaneously [19, 20]. The number of concurrent iterations is bounded by the amount of storage allowed for one copy of the loop body. Under pure dynamic dataflow model, such restriction has been eliminated. Instead, the loop is "unraveled" dynamically at runtime, and the execution can initiate as many iterations as possible, limited only by data dependencies [3]. Although the model provides opportunity to fully exploit fine-grain parallelism in the loop, managing the amount of storage it needs to do so has been a challenge.

In the present paper, we have developed a framework to determine, at compile-time, the minimum storage requirement enough to fully exploit the fine-grain parallelism in a loop. Our formal framework is developed under a *FIFO* dataflow model where each arc in the dataflow graph is a FIFO queue of certain size. We study the storage allocation problem for a class of loops called *dataflow software pipeline* (DFSP). The maximum computation rate of a DFSP is bounded by the *critical cycles* in the dataflow graph [23].

2 Loop Storage Minimization: An Example

In this section, we give an example to show the challenge of minimizing storage requirement while keeping maximum computation rate of loop execution on dataflow machines. Let's consider the *for* loop with *loop-carried dependencies* as shown in Figure 1 (a).

Figure 1 (b) shows the dataflow graph with external input arcs which is omitted from Figure 1 (c). Note that a complete translation of L also contains loop control actors, such as switch and merge actors [11]. For simplicity, we omit them from Figure 1. [1]

[1] We assume that the loop is executed a very large number of iterations. Therefore, it is reasonable to assume that the switch and merge actors for loop control will always take a fixed branch path except the start and termination of the loop.

Abstract

In scientific computation, loops are frequently used to compute large quantities of data organized in arrays. On a dataflow machine, the main challenge is how to maximally exploit fine-grain parallelism to speed up loop execution while not incurring excessive storage space overhead than what is necessary.

The main contributions of this paper include:

- The minimum storage requirement to support the maximum computation rate is analyzed and a storage minimization scheme called *limited balancing* is introduced. The basic intuition is that, since maximum computation rate is dominated by critical cycles in the loop, we should not allocate extra storage beyond a certain limit bounded by the ratios of the critical cycles. In other words, all cycles should be *balanced* to have the same balancing ratio.
- The limited balancing problem is formulated as a integer linear programming problem. An efficient solution of the problem is presented. It reduces the problem to a network flow problem called "minimum circulation flow" problem. Therefore, a polynomial time algorithm is established for the solution of the linear relaxation of the limited balancing problem.

Our formal framework is developed under a *FIFO* dataflow model where each arc in the dataflow graph is a FIFO queue of certain size. we establish the maximum computation rate of a loop under *earliest firing schedule*, and show that the maximum computation rate is dominated by the *critical cycles* of the dataflow graph. We discuss how our results may be applied to both the static dataflow model and the dynamic dataflow model.

1 Introduction

Efficient execution of loops is among one of the most important challenges for high-performance computer architectures. This is true for both conventional computers and dataflow computers [4, 2, 10, 15, 28].

Under the dataflow model, a computation is described by a dataflow graph—a directed graph with nodes representing actors and arcs transmitting tokens which carry the values to be processed by the actors. Actors become activated for execution when tokens are present at their input arcs. Unlike von Neumann computers, dataflow computers have no program counter or other form of centralized sequential control mechanism. Consequently, the order of instruction execution is restricted only by data dependencies within the dataflow programs.

In recent years much research has been conducted on dataflow architectures [1, 11, 14, 13, 32]. Two major approaches to the architectures of dataflow computers have been

List of Figures

1 Dataflow graph for loop L 6
2 Minimize Storage Allocation 7
3 A example of DFSP and its augmented S-DFSP-C 12

Contents

1 Introduction **4**

2 Loop Storage Minimization: An Example **5**

3 The Framework **7**

3.1 The FIFO dataflow model 8

3.2 Dataflow Software Pipeline (DFSP) 8

3.3 Maximum Computation Rate 9

4 The Limited Balancing Problem **10**

4.1 Simplify a DFSP with Chain Replacement 11

4.2 Linear Programming Formulation 11

5 A Polynomial Time Solution **13**

6 Related Work **16**

7 Conclusions **17**

McGill University
School of Computer Science

Loop Storage Optimization for Dataflow Machines

ACAPS Technical Memo 23
August 29, 1991

Qi Ning
G.R. Gao

Advanced Computer Architecture and Program Structures Group

3480 University St • Montreal • Canada • H3A 2A7