Area-Efficient Buffer Binding Based on a Novel Two-Port FIFO Structure

Kyoungseok Rha

Samsung Electronics Co., Ltd.
Yongin, Kyunggi-do 449-711, Korea

contron@poppy.snu.ac.kr

Kiyoung Choi
School of EECS
Seoul National University
Seoul 151-742, Korea
+82-2-880-6768
kchoi@azalea.snu.ac.kr

ABSTRACT

In this paper, we address the problem of minimizing buffer storage requirement in buffer binding for SDF (Synchronous Dataflow) graphs. First, we propose a new two-port FIFO buffer structure that can be efficiently shared by two producer/consumer pairs. Then we propose a buffer binding algorithm based on this two-port buffer structure for minimizing the buffer size requirement. Experimental results demonstrate 9.8%~37.8% improvement in buffer requirement compared to the conventional approaches.

Keywords

Buffer binding, buffer sharing, SDF, scheduling

1. INTRODUCTION

Synchronous dataflow (SDF) graphs have been widely accepted as a powerful computation model for DSP applications. An SDF graph consists of a set of nodes (also called actors) representing tasks and a set of arcs (also called channels) representing communications between tasks. A communication is modeled as a sequence of tokens passing through an arc. The temporal sequences of values passed along the communication channels can be viewed as streams.

In DSP systems, buffer memory size is a critical factor that must be considered during the design process because of limited on-chip memory constraints. Many researchers have proposed efficient scheduling algorithms for buffer size reduction. E. A. Lee and D. A. Messerschmitt propose 'block scheduling' for buffer size reduction [4]. But the scheduling algorithm does not consider pipelining, thereby produces a scheduled graph having poor throughput.

A linear programming formulation (called the Optimal Schedule Buffer Allocation (OSBA) formulation) to obtain rate-optimal buffer-optimal schedule for homogeneous graphs is proposed in [7]. The OSBA algorithm converts an original SDF graph to an equivalent homogeneous SDF graph before it schedules the graph. The limitation is that it takes long time and the buffer size result can be worse for the original SDF graph, even if it is optimal for the homogeneous graph.MBRO algorithm proposed by R. Govindarajan and G. R. Gao also gives rate-optimal schedule while reducing buffer size requirement [8]. They formulate the problem as a unified linear programming problem to shorten the execution time.

Many researchers propose buffer sharing algorithms for buffer size reduction. However, they allow buffer sharing only for communications that are guaranteed to not overlap with each other in life-time.

In this paper, we propose a new two-port FIFO buffer structure that can improve the buffer utilization by allowing two simultaneous accesses to the same buffer. Based on the buffer structure, we propose a buffer binding algorithm that minimizes the buffer size requirement for a scheduled SDF graph . The rest of this paper is organized as follows. The following section motivates our work with the help of an example. In Section 3, we discuss the proposed two-port FIFO buffer structure. Section 4 describes the method of calculating the buffer size requirement, and Section 5 deals with the proposed buffer binding algorithm. In Section 6, we report experimental results. Concluding remarks are presented in Section 7.

2. MOTIVATIONAL EXAMPLE

Consider the SDF graph shown in Fig. 1. Let the execution time of actor a be 3 time units while those of other actors be unity. Throughout this paper, we do not allow two instances of an actor to fire concurrently. This is because the actors typically maintain some internal state, which may affect the subsequent firing.

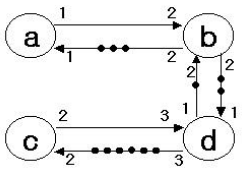

Fig. 1. Motivational example.

After rate-optimal schedule presented in [8], which minimizes the buffer size requirement, the number of tokens on each edge at every time step is shown in Fig. 2 (without loss of generality, we assume that each token

has the same amount of data).

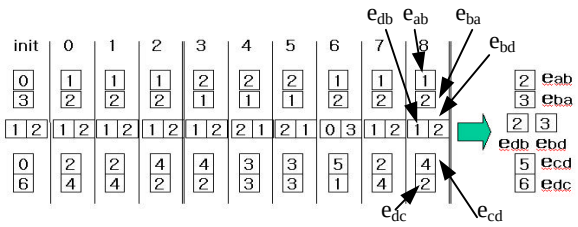

Fig. 2. The number of data on each edge at every time step.

The size of a buffer mapped to an edge must be the maximum number of tokens on the edge at every time step. Denoting the required size of a buffer mapped to edge $\mathbf{e_{uv}}$ as b(u,v),

b(a,b)=2; b(b,a)=3; b(d,b)=2; b(d,b)=3; b(c,d)=5; b(d,c)=6.

Therefore the total buffer size is 2+3+2+3+5+6=21. However, if we use a two-port buffer that can be shared by two streams of data on two different edges, the total buffer size can be reduced further.

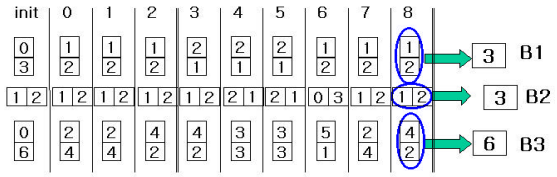

Fig. 3. The use of a two-port buffer.

The size of each two-port buffer must be the maximum number of tokens on both of the two edges at every time step. If each neighboring pairs of edges in Fig. 1 are bound to each two-port buffer as shown in Fig. 3, the total buffer size is B1+B2+B3=12. So, if we do not consider the overhead of the two-port buffer, the buffer size requirement is reduced by 43%.

3. TWO-PORT FIFO BUFFER

3.1 Implementation of the Two-Port FIFO Buffer

The two-port FIFO buffer is realized by extending the one-port FIFO buffer of shift-register style. The structure of the proposed two-port FIFO buffer is shown in Fig. 4.

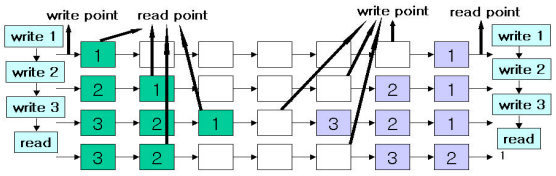

Fig. 4. The structure of the proposed two-port FIFO buffer.

In the case of port 1 (left port), write point is fixed and read point is floating. And in the case of port 2 (right port), write point is floating and read point is fixed. Data shift in each port can occur independently. Data shift in port 1 occurs on write access to port 1, and data shift in port 2 occurs on read access to port 2. To implement this independent data shift, we divide the register into 16 parts and control the shift of each part separately (Fig. 5).

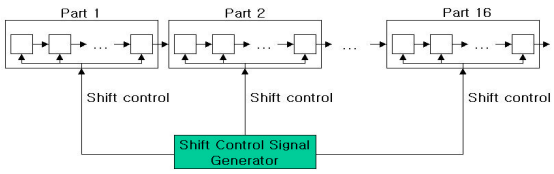

Fig. 5. Partial shift control of the two-port FIFO buffer.

If shift control signal to a part is logic 1, the part is shifted when there exists write access to port 1. If the shift control signal is logic 0, the part is shifted when there exists read access to port 2. Shift control signal generator watches the tail register of port 1, and assigns logic 1 to shift control signals to the parts that store port 1 data, and assigns logic 0 to shift control signals to the parts that store port 2 data. We add one dummy part to the buffer to avoid the case where port 1 data and port 2 data coexistin the same part.

3.2 Area Overhead of the Two-Port FIFO Buffer

We implemented both one-port FIFO buffer and two-port FIFO buffer in VHDL and synthesized to obtain gate-level netlists using the Synopsys Design Compiler.

Table 1. Area Overhead of the Two-Port FIFO Buffer

Buffer size	Area		
	One-port	Two-port	Overhead
32 x 32	8570.12	10344.65	20.7%
64 x 32	17013.52	21070.70	23.8%
128 x 32	33908.52	42140.96	24.3%
256 x 32	67980.67	84333.80	24.1%

There is about 20%~24% overhead in the proposed two-port FIFO buffer compared to the conventional one-port FIFO buffer which has the same capacity. The overhead (average 22%) is taken into account for all discussions in the following sections.

4. BUFFER SIZE CALCULATION

In this section, we explain how to calculate the buffer size requirement for a scheduled SDF graph. In our approach, an edge in the SDF graph can be bound to an one-port FIFO buffer or a two-port FIFO buffer, so we must calculate the buffer size of both types. After the calculation, we perform buffer binding (Section 5) to decide how to bind buffers (one-port or two-port) to edges in the SDF graph.

4.1 Size Calculation for the One-Port FIFO Buffer

Suppose that the schedule for the SDF graph of Fig. 6 has already been finished, and actors A and B are concurrently fired at time step t

according to the schedule.[1] And suppose that all the edges are bound to one-port FIFO buffers. According to the previous work [8], the number of data in buffer a does not change at time step t, because the same number (10) of tokens are created and consumed at the same time. However, this calculation is based on the premise that the consumption occurs always earlier than the production (Fig. 7.(a)).

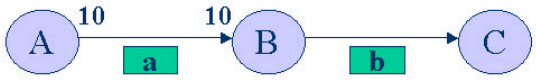

Fig. 6. An example of SDF graph.

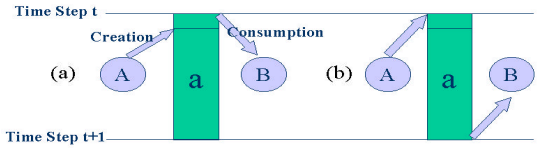

Fig. 7. Two extreme cases: (a) consumption occurs earlier than productionand (b) production occurs earlier than consumption (worst case).

If we don't know whether the production is earlier than the consumption or not, we must calculate the buffer size as in the worst case (Fig. 7.(b)) to avoid buffer overflow. In this regard, we make the following assumption. Input tokens remain on the incoming edge until the activation (firing) of the consumer is completed and output tokens are produced (all at once) at the start of the firing of the producer. This assumption gives worst calculation of the size of each one-port FIFO buffer but guarantees correct operation.

4.2 Size Calculation for the Two-Port FIFO Buffer

Let's consider a two-port FIFO buffer shared by the input and output streams of tokens of an actor.

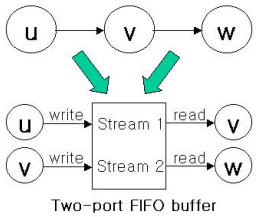

Fig. 8. Master actors on the two-port FIFO buffer.

[1] We assume that the actors are implemented in hardware or software with separate IPs or modules. If two or more actors are implemented with one IP, then they can be merged into one 'super-actor'.

There are two actors that write to the buffer and two actors that read from the buffer. Fig. 8 shows an example where actor U and actor V write to a buffer, and actor V and actor W read from the buffer. When U writes to the buffer, we calculate the buffer size as in the worst case (Sub-section 4.1), that is, we add the number of tokens produced by actor U when the firing of actor U starts. When W finishes, we subtract the number of tokens consumed by actor W from the buffer size.

For the contribution of actor V to the buffer size, if we don't know the buffer access pattern of actor V, we must calculate it as in the worst case. That is, at the start of firing of actor V, we add to the buffer size the number of tokens produced, and at the completion of actor V, we subtract the number of tokens consumed.

However, if we know the buffer access pattern of actor V, we can reduce the two-port buffer size, still maintaining correct operation.
In the best case, if actor V is known to produce all output tokens only after it consumes all input tokens, we can add the number of tokens produced to the buffer size after we subtract the number of tokens consumed.

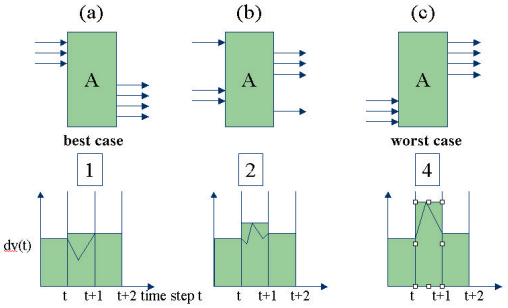

Fig. 9. Three buffer access patterns: (a) the best case, (b) a case in between, and (c) the worst case.

Fig. 9 shows three different cases and the corresponding traces of buffer usage, dv(t).The peak value of dv(t) is the required buffer size.As shown in Fig. 9., due to the firing of actor V, we add 1, 2, and 4 to the buffer size for the cases of (a), (b), and (c), respectively.

5. BUFFER BINDING

Fig. 10 shows that there exist several buffer binding schemes for one SDF graph. Rectangles with 'd' represent dedicated (one-port) FIFO buffers and those with 's' represent shared (two-port) FIFO buffers. We restrict sharing such that only neighboring two edges can be bound to one two-port FIFO buffer to alleviate the routing problem at the layout level.

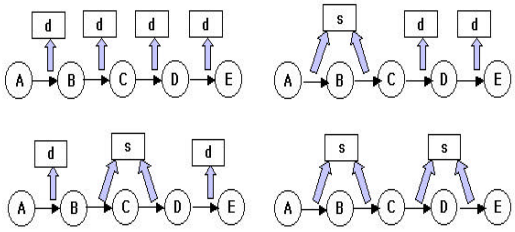

Fig. 10. Several buffer binding methods.

For the explanation of the proposed buffer binding algorithm, we first explain buffer binding algorithm for an SDF graph that has only one path, and then extend the algorithm to the case of multiple paths, which is the general case.

5.1 Buffer Binding for One-Path SDFGraph

Fig. 11 shows an example of SDF graph with only one path. The buffer binding problem can be transformed to an actor selection problem. For example, selecting actor C in Fig. 11 implies that the incoming edge and the outgoing edge of actor C are bound to a two-port FIFO buffer. If we do not select actor C, then it means that the incoming edge is bound to an one-port FIFO buffer, and the outgoing edge is bound to another FIFO buffer.

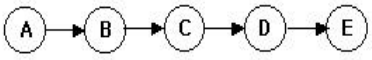

Fig. 11. An example of one-path SDF graph.

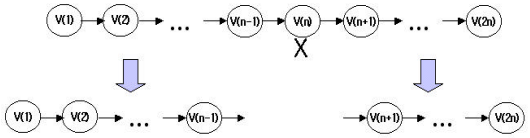

Fig. 12. When V(n) is not selected.

To solve the buffer binding problem, we take the divide and conquer approach. Fig. 12 shows an one-path SDF graph with 2n actors where actor V(n) is not selected. In this case, the buffer binding problem can be divided into that of left sub-graph (V(1)~V(n-1)) and that of right sub-graph (V(n+1)~V(2n)). Note that, once V(n) is decided not to be selected, the buffer binding of the left sub-graph and that of the right sub-graph are mutually independent. Therefore, the problem can be divided into two smaller problems.

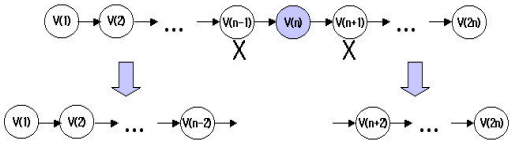

Fig. 13. When V(n) is selected

Fig. 13 shows the case where V(n) is selected. In this case, V(n-1) and V(n+1) cannot be selected. Therefore, the binding of the left sub-graph V(1)~V(n-2) and that of the right sub-graph V(n+2)~V(2n) are mutually independent. Except the exclusion of the two actors V(n-1) and V(n+1), the problem division process is the same as that in the case of Fig. 12.

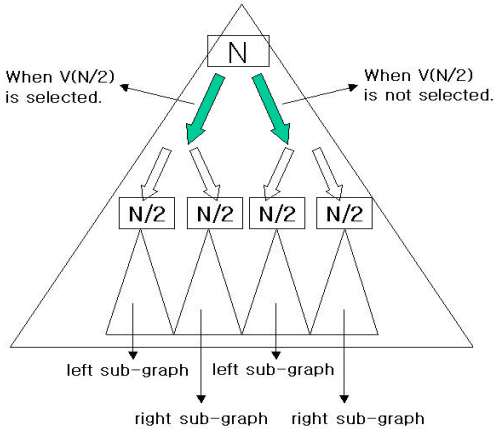

Fig. 14. Buffer binding algorithm for one-path SDF graph.

Fig. 14 depicts the proposed recursive buffer binding algorithm for a one-path SDF graph. The buffer binding algorithm finds an optimum solution to the problem with N actors by trying both cases: the one where V(N/2) is selected and the other where V(N/2) is not selected. Each case is divided into two sub-problems: buffer binding of the left sub-graph and that of the right sub-graph.

Theorem: The computational complexity of the proposed buffer binding algorithm is $O(N^2)$.
Proof: Let the computational complexity of the algorithm for N actors be BB(N). Then,

$$BB(N) = O(4 \times BB(N/2)) = O(4^2 \times BB(N/2^2))$$
$$= \ldots\ldots = O(4^{\log_2 N} \times BB(1)) = O(4^{\log_2 N}) = O(N^2) \quad \blacksquare$$

5.2 Buffer Binding for a Multi-Path SDF Graph

Fig. 15 shows an example of SDF graph with multiple paths. In this case, we first divide the SDF graph into several one-path sub-graphs as shown in Fig. 15. Actor a (b) has several output (input) ports. We call the actor that has several input or output ports *multi-port actor*.

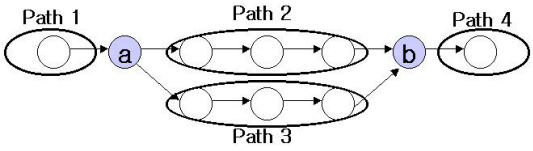

Fig. 15. An example of multi-path SDF graph.

The total buffer size requirement is much affected by how to bind the edges incident to/from the multi-port actors to one-port or two-port FIFO buffers.

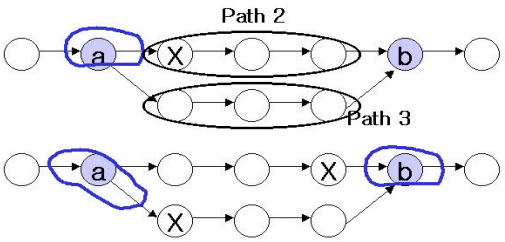

Fig. 16. Several choices that bind the buffers to the edges of multi-port actors

If we bind the edges as shown in the upper case of Fig. 16(the incoming edge and one outgoing edge of actor a are bound to a two-port FIFO buffer and each of the edges of actor b is bound to a separate one-port FIFO buffer), the left most actor of path 2 cannot be ' selected' (Sub-section 4.1). Similarly in the lower case of Fig. 16, the right most actor of path 2 and the left most actor of path 3 cannot be ' selected' .

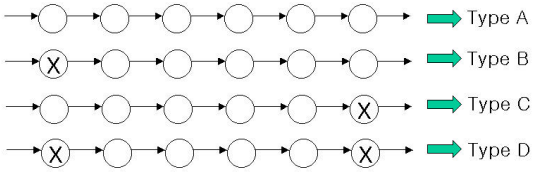

Fig. 17. Four types of the restriction on the neighboring one-path sub-graphs

We classify such effects on an one-path sub-graph into 4 types as shown in Fig. 17. Now the proposed general buffer binding algorithm is as follows. First, we divide the SDF graph into several one-path sub-graphs. Secondly, we run the buffer binding algorithm for an one-path SDF graph (Subsection 5.1) on all one-path sub-graphs , for all of the 4 types. Thirdly, for each combination of buffer binding for the multi-port actors, we select suitable types of one-path sub-graphs and add up all optimal costs to obtain the total buffer size requirement. Finally, we select the combination that gives the best solution.

Because the algorithm exhaustively searches all combinations, the complexity is exponential in the number of multi-port actors. However, since most of the computations in the main loop body are simple additions, the algorithm runs relatively fast. Besides, in actual cases, there are a few multi-port actors in SDF graphs (Section 6).

6. EXPERIMENTAL RESULTS

Table 2 shows the information about five DSP applications used for our experiments. As we mentioned in Subsection 5.2, there are a few multi-port actors in the examples. So the execution time of our buffer binding algorithm is very short (less than 1 second for all examples on a Sun workstation).

Table 3. DSP Applications

Examples	No. of Actors	No. of Edges	No. of Multi-port Actors
JPEG	3	2	0
OverlapAddFFT	7	7	2
Analytic	8	7	1
CD2DAT	4	3	0
Satellite Receiver	22	26	6

The procedure of our experiments is as follows. First, we ran the MBRO algorithm [8], which is a rate-optimal compile-time scheduling algorithm that minimizes buffer storage requirement without sharing, on five examples (second column in table 3). Then we compared the improvement in buffer size by the proposed method (4-th ~ 6-th column in table 3) with that by the traditional buffer sharing (third column in table 3) that allows buffer sharing only between buffers that are guaranteed to not overlap with each other in life-time.

For the approach that we propose, we tried three different cases: the 'Worst Case' , the ' Actual Case' , and the ' Best Case' (refer to Subsection 4.2). For the ' Actual Case' , we analyzed source codes written in C to obtain the buffer access pattern for all actors in SDF graphs, and applied that information to our algorithm. We do not have data for the Satellite Receiver because the source code was not available. In reality, the

Table 3 Buffer Size Requirement

Examples	Using Only One-Port Buffers (MBRO)	Traditional Buffer Sharing (Improvement)	Our Approach		
			Worst Case (Improvement)	Actual Case (Improvement)	Best Case (Improvement)
JPEG	256	256 (0%)	256 (0%)	231 (9.8%)	231 (9.8%)
OverlapAddFFT	644	580 (9.96%)	644 (0%)	505 (21.6%)	505 (21.6%)
Analytic	1441	1429 (0.84%)	1429 (0%)	1147 (20.4%)	1147 (20.4%)
CD2DAT	496	496 (0%)	449 (9.95%)	446 (10.1%)	446 (10.1%)
Satellite Receiver	1570	1544 (1.66%)	1561 (0.7%)	-	976 (37.8%)

‘Actual Case’ and the ‘Best Case’ have the same results in most cases. The reason is that most actors produce output tokens only after they consume all input tokens.

The overhead (22%) of the proposed two-port buffers has already been added to the data in the table. In the case that there is no improvement (0%) as in the ‘Worst Case’, there is no buffer sharing and only one-port buffers are used. Therefore, there is no overhead added to the data. In this case, the traditional buffer sharing method looks better than the worst case of our approach. Note, however, that the traditional buffer sharing method can also be applied after our buffer binding algorithm. But considering the routing overhead due to the sharing, we do not expect significant improvement.

Given the information on the buffer access patterns of the actors in an SDF graph, the experimental results show that our method can reduce the buffer size requirement by 9.8%~37.8% compared to the traditional approach.

7. CONCLUSIONS

In this paper, we proposed a new two-port FIFO buffer structure that can be used for an efficient buffer sharing. We also proposed a buffer binding algorithm that exploits the use of this two-port buffer to minimize the buffer size requirement for a scheduled SDF graph.

The weak point of this paper is that the complexity of the buffer binding algorithm for a multi-path SDF graph is exponential in the number of multi-port actors. This remains as a future research.

REFERENCES

[1] V. Madisetti, *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*, Bufferworth-Heinemann, Boston, MA, 1995.

[2] R. Karp and R. Miller, "Properties of a model for parallel computations: determinancy, termination, queueing," *SIAM Journal of Applied Math.*, Vol. 14, No. 6, Nov. 1996.

[3] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of IEEE*, Vol. 75, No. 9, pp. 1235~1245, Sept. 1987.

[4] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, Vol. C-36, No. 1, pp. 24~35, Jan. 1987.

[5] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: a framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, Vol. 4, No. 2, pp. 155~182, April 1994.

[6] S. Bhattacharyya, P. Murthy, and E. Lee, "APGAN and RPMC : complementary heuristics for translating DSP block diagrams into efficient software implementations," *Journal of Design Automation for Embedded Systems*, Vol. 2, No. 1, pp. 33~60, Jan. 1997.

[7] Q. Ning and G. R. Gao, "A novel framework of register allocation for software pipelining," *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. On Principles of Programming Languages*, pages 29-42, Charleston, South Carolina, Jan. 10-13, 1993. ACM SIGACT and SIGPLAN.

[8] R. Govindarajan, G. R. Gao, and P. Desai, "Minimizing memory requirements in rate-optimal schedules," *Proc. of the 1993 Intl. Conf. on Application Specific Array Processors*, pages 77-88, Venice, Italy, October 1993.

[9] R. Govindarajan, G. R. Gao, "Multi-rate optimal software pipelining for regular dataflow networks," *ACAPS Technical Memo 61*, School of Computer Science, McGill University, Montreal, Que., 1993.

[10] V. Van Dongen, G. R. Gao, and Q. Ning, "A polynomial time method for optimal software pipelining," *Proc. of the Conference on Vector and Parallel Processing*, pages 613-624, Lyon, France, Sept. 1992. Also in *LNCS-634*.