PIPELINING: JUST ANOTHER TRANSFORMATION

Miodrag Potkonjak
C&C Research Laboratories, NEC, 4 Independence Way, Princeton, NJ 08540

Jan Rabaey
Dept. of EECS, University of California, Berkeley, CA 94720

ABSTRACT

A simple formulation of pipelining: "Pipelining with N stages is equivalent to retiming where the number of delays on all inputs or all outputs, but not both, is increased by N" is used as the basis for a convenient and efficient treatment of pipelining in design of application specific computers.

Classification of pipelining according to the optimization goal (throughput and resource utilization) and the latency is introduced. For polynomial complexity pipelining classes, optimal algorithms are presented. For other classes both proof of NP-completeness and efficient probabilistic algorithms are presented. Both theoretical and experimental properties of pipelining are discussed. In particular, a relationship with other transformations is explored. Due to close relationship between software pipelining and pipelining presented here, all results can be easily modified for use in compilers for general purpose computers. Also, as the side result, the exact bound (solution) for iteration bound is derived.

1.0 Introduction

Pipelining is an implementation technique where different instances of programs primitives are partially overlapped during execution. Depending on what is considered as a basic building block of a program, three forms of pipelining are most often discussed: suboperational (also called bit level pipelining, structural pipelining or just pipelining), control loop pipelining (software pipelining, loop winding and loop folding) and functional pipelining. The program primitive in suboperational pipelining is an operation, in control pipelining it is a program loop and in functional pipelining the instance is the whole program which is executed iteratively. Pipelining is probably the most often used transformation in DSP ASIC design and high level synthesis as well as the most discussed one in the DSP VLSI and CAD literature. An excellent, engineering oriented reference is the book by Kogge [Kog81]. He discusses many important technical questions in detail. Another excellent references include Chapter 6 of Hennessy's and Patterson's book "Computer architecture" [Hen89, Pat89], Chapter 3 in "High-performance computer architecture" by Stone [Sto90] and Chapter 3 in "Computer architecture and parallel processing" by Hwang and Briggs [Hwa84]. These books contain an extensive discussion in the field of pipelining research in the context of general computing and long lists of additional refer-

0-8186-2967-3/92 $03.00 © 1992 IEEE

ences. Therefore, we will limit our overview of previous work to the DSP ASIC design and high level synthesis treatment of pipelining.

This paper refers to functional pipelining, the technique of the partitioning of the control data flow graph into subgraphs that will be performed concurrently. Successive subgraphs, called pipeline stages, are streamed into the pipe so that different control data flowgraph (CDFG) instances are executed simultaneously on the same hardware. Since there is a close relationship between functional and control loop (software) pipelining all techniques and results presented here can be easily adapted for the software pipelining.

Suboperational pipelining is the most often discussed as a feature in scheduling algorithms. While Moritz and Chen [Che91] discuss the simultaneous application of both structural and functional pipelining techniques, the majority of the work addresses functional pipelining. The major reason why functional pipelining is getting high attention is due to its ability to sometimes significantly reduce the critical path, and to increase the scheduling freedom of operation.

Park and Parker produced the most comprehensive treatment of pipelining[Par88b, Par88c]. They discussed both the theoretical foundations and practical implementation issues. Later on, research at the University of Southern California has been continued with numerous studies on the effectiveness of pipelining and its relationship to other high level synthesis tasks [Jai89, Mli91]. Casavant and his co-workers at GE developed the PISYN- high level synthesis system for the application specific pipelined hardware [Cas91]. [Hwa91] used integer programming technique for the optimization of the pipelined designs. Kim [Kim91] discussed in detail how specific logic synthesis techniques can be used for synthesis and optimization of controller in pipelined design.

However, it is important to stress that in a significant part of the high level synthesis publications, only CDFG's without feedback edges are considered. Actually, even when the original examples do have feedback loops, those are sometimes ignored. This approach highly simplifies the study of pipelining and its application, but have a very limited application range. When it is applied to computations, which do have feedback in their CDFG, this results in a change in input-output relationship and therefore incorrect results.

In the digital signal processing community, pipelining is also a well discussed topic. The accent here is on how to manually apply pipelining in conjunction with other flow graph transformations for specific application areas (e.g. infinite response filters, dynamic programming based calculations such as the Viterbi algorithm), in order to achieve maximal speed-up [Par88a, Lin91, Fet90]. An excellent introduction to this type of research is again Kogge's book [Kog81].

In the compiler research, where a repeated execution of programs rarely occurs and where the most important transformations are related to program control, pipelining is most often discussed during its application on loops [Lam88, Lam89]. While this technique is often called loop winding or loop folding in high level synthesis, [Goo89, Gyr84] in the compiler literature it appears under the name of software pipelining. Jouppi [Jou89] discussed the amount of software pipelining potential in various classes of programs.

This paper treats functional pipelining as a high level synthesis transformation. This approach enables a simple and straightforward inclusion of pipelining in the HYPER transformational environment, and therefore its combination with other transformations, which significantly enhance pipelining power. HYPER is the high level synthesis system for the ASIC implementation of computationally intensive applications [Rab91]. In HYPER, structural pipelining is treated during scheduling. There is a very close relationship between software pipelining and functional pipelining. The only conceptual difference is that while software pipelining

operates on program control loops, functional pipelining operates on time loops. Also, from technical point of view, software pipelining includes additional important issues of loop prologue (initiation) and loop epilogue (finishing) effects [Goo89]. All results and techniques presented here can, therefore, be also easily applied on software pipelining.

The work presented here differs from the previously reported in both the scope and the used techniques. Four different forms of pipelining are identified and discussed. For the two of these problems the optimal polynomial complexity algorithms are presented. For the other two an NP-completeness proof is established and new probabilistic algorithms are used to generate a solution. The effects on the resulting hardware implementation for a number of diverse examples are reported and analyzed.

The rest of this paper is organized in the following way. After a definition of the four different forms of pipelining, their computational complexity is established. Next, an optimal polynomial algorithm is presented when the goal is the minimization of the critical path. For the case when the goal is the optimal resource utilization, the objective function during pipelining is defined, and a learning while searching algorithm is used for the optimization. After a discussion of experimental results, conclusions are drawn.

2.0 Problem Formulation

In the following section, the following simple, yet effective framework of the pipelining concept will be employed: "Pipelining with N stages is equivalent to retiming where the number of delays on all inputs or all outputs, but not both, is increased by N".

As we have already mentioned we treat pipelining just as yet another transformation. Strictly speaking pipelining is not really a transformation, as it changes the phase between input and output signals. However, since it preserves the input/output computational relationship, we will consider it as a transformation in the broad sense.

Depending on the goal and the level of the introduced latency, four different forms of pipelining can be identified. Those four pipelining forms, which are implemented in HYPER, are:

(1) pipelining for the minimization of the critical path;
(2) pipelining for the minimization of the critical path for a given number of pipeline stages;
(3) pipelining for resource utilization;
(4) pipelining for resource utilization for a given number of pipeline stages.

We will denote them as **CP**, **CP(n)**, **RP** and **RP(n)** respectively, where n stands for the number of pipeline stages. Before discussing the algorithms, a precise definitions of those classes are given first.

Pipelining for critical path (CP) produces a minimal stage time. The stage time is defined as the length of the critical path of the CDFG after pipelining.

Pipelining for critical path using n pipeline stages (CP(n)) produces a CDFG with a minimal stage time, but so that exactly n pipeline stages are introduced.

Pipelining for resource utilization (RP) produces a CDFG which realizes the minimum area for a given timing constraint.

Pipelining for resource utilization using n pipeline stages (RP(n)) produces a CDFG, which can be implemented using a minimum area, for a given timing constraint. The number of introduced pipeline stages should equal exactly n.

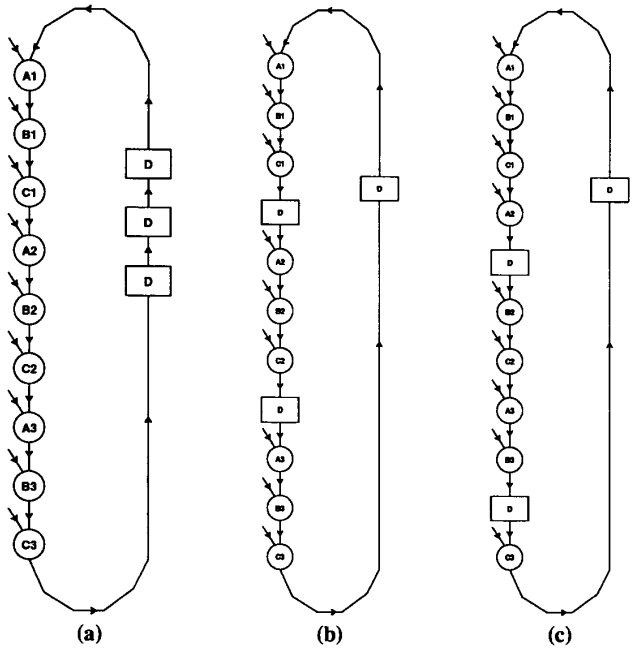


FIGURE 1. (a) Initial CDFG; (b) CDFG after the application of pipelining for the minimization of critical path; (c) CDFG after the application of pipelining for the resource utilization

Cycle	A	B	C
1	A1, A2	-	-
2	A3	B1, B2	-
3(a)	-	B3	C1, C2
4	-	-	C3

Table 1: Schedule after pipelining for critical path; () Schedule after pipelining for resource utilization

Cycle	A	B	C
1	A1	B2	C3
2	-	B1	C2
3	A3	-	C1
4	A2	B3	-

Table 2: Schedule after pipelining for resource utilization

It is sometimes argued that pipelining for critical path and pipelining for resource utilization are the same. Actually, until now difference between those two types of pipelining were not discussed in the literature. The discussion of the computational complexity of the associated optimization problems will denote that it is not true, but the following simple example provides intuitive insight into difference. Their relationship closely resembles the relationship between retiming for critical path and retiming for resource utilization.

Suppose that we have the computation shown in Figure 1(a), 1(b) and 1(c) in the flow graph format. Assume that each operation takes one control cycles, and that available time equals 4 control cycles. There are three different types of operations: A (operations A1, A2, A3), B (operations B1, B2, B3) and C (operations C1, C2, C3). For the sake of simplicity, we will only take the cost of the execution units into account. Due to the recursive bottleneck [Mes88], at most three pipeline stages can be introduced.

Pipelining for critical path will result in the transformed CDFG shown in Figure 1(b). The critical path is 3. It is easy to see that the optimal schedule needs at least 2 executional units of each type. A possible schedule is shown in Table 1.

The effect of pipelining for resource utilization on the CDFG is shown in Figure 1(c). The critical path is 4. However, although the critical path is longer than in the former case, only 3 executional units are needed for the realization. The schedule is shown in Table 2. Therefore, we can conclude, that pipelining for critical path and pipelining for resource utilization are indeed two different transformations.

3.0 The Computational Complexity of Pipelining

Sometimes a very small change in the formulation of a problem can result in a drastic change in its computational complexity. There are many pairs of similar problems, where one is NP-complete and the other can be solved in polynomial time [Gar79]. However, there are very few such pairs in high level synthesis. It is interesting that maximal pipelining and pipelining for resource utilization, as well as maximal pipelining for a given number of stages and pipelining for a resource utilization for a given number of pipeline stages, constitute two such pairs. While maximal pipelining problems have a polynomial complexity (see next section), pipelining for resource utilization problems belong to the class of NP-complete problems.

It is easy to prove that pipelining for resource utilization is an NP-complete problem. Actually, since we already proved that retiming for resource utilization is an NP-complete prob-

lem [Pot91a], we can use the simplest and most frequently applicable technique for proving NP-completeness: restriction [Gar79]. If we look at the NP-completeness proof for retiming for resource utilization we see that in the used CDFG it is impossible to introduce any new pipeline stages, since all operations are in recursive loops. So, if we have a polynomial complexity algorithm for either RP or RP(n) we will be able to solve the retiming for resource utilization problem, and therefore also the MAX-CUT problem, which is NP-complete [Gar79]. Therefore, we can conclude that both RP and RP (n) are at least as difficult as NP-complete problem. On the other hand, if we have a solution for either PR or RP(n) with a given assignment and schedule, we can easily check the cost of the proposed solution. Thus, both RP and RP(n) are NP-complete problems.

4.0 Pipelining Algorithms

4.1 Pipelining for minimization of the critical path for a given number of pipeline stages (CP (n))

The definition of pipelining as retiming where the number of delays on all inputs is increased by the number of pipeline stages directly leads to the efficient algorithm in the case of CP(n). The obviously correct and efficient algorithm can be obtain by applying the Leiserson-Saxe retiming algorithm to the CDFG where the number of delays on the input edges is increased by n. Since there is an excellent and extensive literature on the Leiserson-Saxe retiming algorithm, [Lei83, Goo86] we will not further elaborate this issue.

4.2 Pipelining for minimization of the critical path

Only a slightly more complex modification of the Leiserson-Saxe algorithm is needed to generate the fastest possible solution, using only pipelining as the transformational means. We can add on pipeline stages a very large number of delays (say as many as the number of operations in the CDFG), and obtain the CDFG which will have the minimum stage time achievable by pipelining. If we are interested in the minimum number of pipeline stages needed for the maximal minimization of the critical path, we can use a binary search over the number of introduced delays. In that manner, we can detect the situation where we still achieve the minimal critical path, with the smallest number of pipeline stages. Of course, during the binary search, we can still use the Leiserson-Saxe retiming algorithm to solve the pipelining problem with a given number of pipeline stages.

A more elegant approach, but conceptually very similar, is to directly modify the Leiserson-Saxe retiming algorithm, so that either all inputs or all outputs (but again not both) are not connected to the host node. (See [Lei83] for a detailed and clear description of the algorithm). In this case the algorithm will automatically introduce the necessary minimum number of pipeline stages. The correctness proof of the introduced modification is straightforward.

The interesting and important side result of the just described algorithm is that it also answers the questions about fast derivation of optimum pipelining rate in systolic array processors design [Kun88], as well as iteration bound for ASIC designs [Mes88, Par89]. The previous techniques required the search over all loops in CDFG, and since their number can be exponential with the respect to the number of nodes, all previous algorithms had an exponential complexity. The new approach has run time O (|V| |E| lg |V|), where |V| is the number of nodes in the

CDFG, and |E| is the number of edges. [Lei91]. Since all CDFG are sparse (each node has usually two inputs, or at most some small integer number of inputs), this procedure has the essentially quadratic run time, so that very large CDFG can be treated.

4.3 Pipelining for resource utilization with *n* stages

Again, as in the previous two cases, the formulation of pipelining as the retiming with the increased number of delays on all inputs provides a good starting point for the design of an efficient fast algorithm. It is easy to see that it is sufficient to increase the number of delays on all inputs simultaneously by n and then to apply retiming for resource utilization in order to design the algorithm for CP(n) [Pot91b]. However, in order to make the algorithm as efficient as possible, it is necessary to make several modifications in the algorithm for retiming for resource utilization. These modifications are needed because in the case of pipelining, especially in cases when a large number of pipeline stages are introduced, the number of delays in the graph grows extensively and therefore the solution space is far larger. Hence, the set of possible moves is growing, decreasing the effectiveness of the algorithm.

For the sake of completeness we will briefly describe the retiming for resource utilization learning while searching optimization algorithm.

The learning while searching algorithm is organized as a two phase process. In the first phase, the solution space is scanned in an organized fashion to detect areas where the objective function has a small value. Those areas are used in the second phase as the starting points for a more elaborate search towards a final solution.

The goal of the first phase is thus to discover (learn) k solutions where the objective function has a small value. In order to achieve this goal, we will traverse the search space a number of times, each time favoring one particular direction of traversal. For instance, we will first (probabilistically) favor moves in the forward direction (moving the delays from the inputs to the outputs for the retiming and favoring the forward associativity moves). After 1/4 of the optimization process, the preferred direction is reversed: moving the delays from output to input as well as the reverse associativity moves are now probabilistically favored. Finally, for the last 1/4 of the time, the forward moves are favored anew.

At every point in the optimization process, we select a move in a probabilistic fashion, proportional to the improvement in the objective function, while also accounting for the favored direction at that time. No selected moves are ever rejected. Moves, which increase the objective function, can be selected, but the changes for this to happen are inversely proportional to that increase.

The first phase results in k starting points for the second phase.

Those are used as the seeds for a steepest descent search towards the final solution. The objective function is now observed at each step and for all possible moves. The move offering the best decrease in the objective function is automatically selected. For each starting point, the search is concluded when a local minimum is reached. The best of those minima is selected as the final solution. We have set the number of starting points k to 10 for the examples discussed in the next section. The length of the first phase was set such that the number of moves during the first forward traversal equaled 10 times the number of nodes in the graph. Moves in the forward direction were preferred with a ratio 4:3. We have varied these values of large ranges and did not notice any significant changes in the quality of the solution, although the effects on the run time were outspoken.

CDFG, and |E| is the number of edges. [Lei91]. Since all CDFG are sparse (each node has usually two inputs, or at most some small integer number of inputs), this procedure has the essentially quadratic run time, so that very large CDFG can be treated.

4.3 Pipelining for resource utilization with *n* stages

Again, as in the previous two cases, the formulation of pipelining as the retiming with the increased number of delays on all inputs provides a good starting point for the design of an efficient fast algorithm. It is easy to see that it is sufficient to increase the number of delays on all inputs simultaneously by n and then to apply retiming for resource utilization in order to design the algorithm for CP(n) [Pot91b]. However, in order to make the algorithm as efficient as possible, it is necessary to make several modifications in the algorithm for retiming for resource utilization. These modifications are needed because in the case of pipelining, especially in cases when a large number of pipeline stages are introduced, the number of delays in the graph grows extensively and therefore the solution space is far larger. Hence, the set of possible moves is growing, decreasing the effectiveness of the algorithm.

For the sake of completeness we will briefly describe the retiming for resource utilization learning while searching optimization algorithm.

The learning while searching algorithm is organized as a two phase process. In the first phase, the solution space is scanned in an organized fashion to detect areas where the objective function has a small value. Those areas are used in the second phase as the starting points for a more elaborate search towards a final solution.

The goal of the first phase is thus to discover (learn) k solutions where the objective function has a small value. In order to achieve this goal, we will traverse the search space a number of times, each time favoring one particular direction of traversal. For instance, we will first (probabilistically) favor moves in the forward direction (moving the delays from the inputs to the outputs for the retiming and favoring the forward associativity moves). After 1/4 of the optimization process, the preferred direction is reversed: moving the delays from output to input as well as the reverse associativity moves are now probabilistically favored. Finally, for the last 1/4 of the time, the forward moves are favored anew.

At every point in the optimization process, we select a move in a probabilistic fashion, proportional to the improvement in the objective function, while also accounting for the favored direction at that time. No selected moves are ever rejected. Moves, which increase the objective function, can be selected, but the changes for this to happen are inversely proportional to that increase.

The first phase results in k starting points for the second phase.

Those are used as the seeds for a steepest descent search towards the final solution. The objective function is now observed at each step and for all possible moves. The move offering the best decrease in the objective function is automatically selected. For each starting point, the search is concluded when a local minimum is reached. The best of those minima is selected as the final solution. We have set the number of starting points k to 10 for the examples discussed in the next section. The length of the first phase was set such that the number of moves during the first forward traversal equaled 10 times the number of nodes in the graph. Moves in the forward direction were preferred with a ratio 4:3. We have varied these values of large ranges and did not notice any significant changes in the quality of the solution, although the effects on the run time were outspoken.

5.0 Experimental Results

For testing of the proposed pipelining algorithms we used a set of 15 different instances of 7 designs:

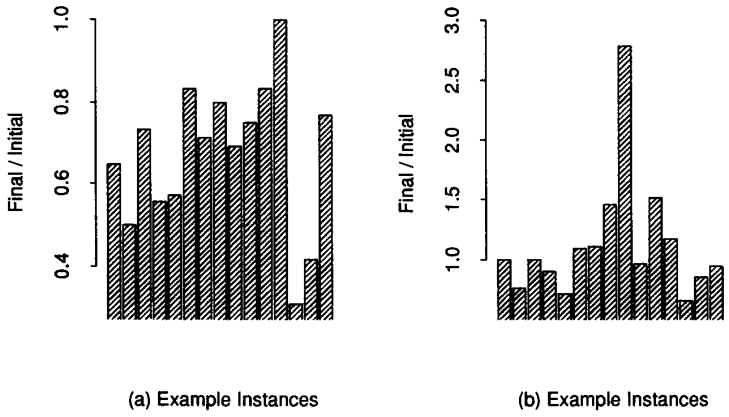
- iir7 - 7th order IIR filter,
- iir5 - 5th order IIR filter,
- fir11 - 11th order FIR filter,
- iir11 - 7th order IIR filter followed by 4th order equalizer,
- dct8 - DCT transform for 8 points,
- decby4 - decimation elliptic filter,
- volterra2 - second order Volterra filter.

	Initially						After CP						
Example	**CP**	**M**	**A**	**S**	**BS**	**R**	**CP**	**IPS**	**M**	**A**	**S**	**BS**	**R**
iir7	10	3	2	1	-	33	3	3	2	1	1	-	33
iir7	13	-	2	2	2	38	6	1	-	1	1	1	29
iir5	8	3	2	1	-	26	2	3	2	1	2	-	26
iir5	10	-	3	3	3	32	6	1	-	2	1	2	29
fir11	11	2	1	-	-	28	1	2	1	1	-	-	20
fir11	11	-	2	1	3	32	4	1	-	2	1	2	35
iir11	20	3	1	1	-	55	3	6	2	1	1	-	61
iir11	57	-	1	2	2	57	6	9	-	1	1	2	83
dct8	7	5	3	3	-	33	1	6	3	2	4	-	92
decby4	14	4	5	3	-	57	3	4	2	2	2	-	55
decby4	34	-	2	2	2	41	11	3	-	1	2	2	62
volterra2	12	2	1	1	-	23	12	0	2	1	1	-	27
volterra2	15	4	2	2	6	38	10	1	1	1	1	1	25
volterra2	10	7	5	3	-	48	5	1	3	2	1	-	41
volterra2	14	1	3	2	4	58	7	1	1	3	1	2	55

Table 3: The effect of the maximal pipelining

CP - Critical Path; M - number of Multipliers; A - number of Adders; S - Number of Subtractors; BS - number of Barrel Shifters; R - number of Registers; IPS - number of Introduces Pipeline Stages.

The available time in all examples was equal to the initial critical path. All designs (except dct8) were done in two ways: before and after the application of substitution of multiplications by constant with shifts and additions. In the two last cases of the Volterra filter, we also applied time loop unrolling. The effects of pipelining for minimization of the critical path are shown in Table 3 and Figure 2. The effects of pipelining for minimization of the resource utilization are shown in Figure 3



(a) Example Instances (b) Example Instances

FIGURE 2. The changes in (a) execution units area and (b) registers area due to the application of the pipelining for the minimization of critical path

In the case of CP, the average improvement in the area of execution units was 32.5%, the median reduction was 28.6%, best improvement was 69.2%, and in the worst case the area was unchanged. For the registers the situation was very different. The average register area was larger 12.9%, the median value was unchanged, the best improvement was only 33.2%, and in the worst case the register area increased 178.8%.

In the case of RP, the average improvement in the area of execution units was 39.1%, the median reduction was 33.3%, best improvement was 69.4%, and the smallest improvement was 16.7%. For the registers the situation was again very different. The average register area was only 5.9% smaller, the median saving was 9.4%, best improvement was 36.8%, and in the worst case the register area increased 36.4%.

In the final experiment we combined RP with the application of two other transformations: associativity and commutativity. Associativity was applied simultaneously with pipelining, while commutativity was applied independently after RP. The average improvement in the area of execution units was 46%, the median reduction was 46.7%, best improvement was 81.6% (the final implementation was more then 5 time smaller), and in the worst case improvement was

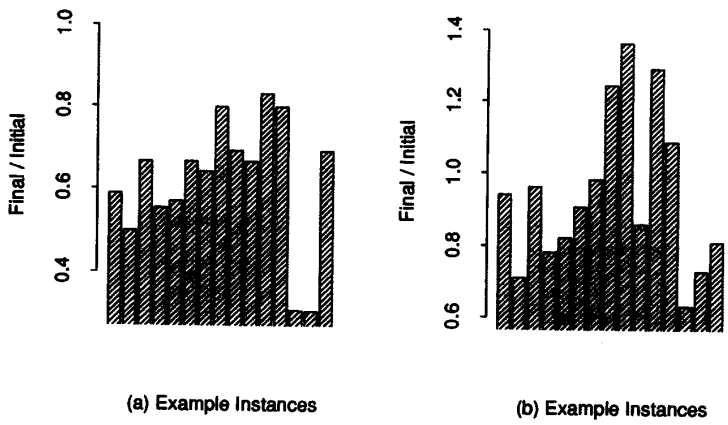


FIGURE 3. The changes in (a) execution units area and (b) registers area due to the application of the pipelining for the resource utilization.

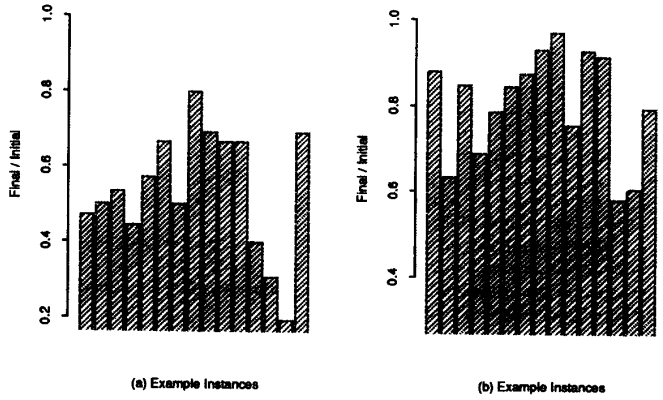


FIGURE 4. The changes in (a) execution units area and (b) registers area due to the application of the pipelining for the resource utilization, associativity and commutativity.

20%. The average, median, best and worst case improvement in the register cost was 19.9%, 16.4%, 43.1% and 3%. In this case pipelining with the aid of two algebraic transformations was able to improve in all cases both execution units and register cost

Analyzing experimental results we can draw many conclusions, but the most important are:

(1) That although pipelining is powerful transformation for both critical path reduction, as well as resource utilization improvement, on examples with feedback loops, which have many computational elements and few delays it has very limited efficiency.

(2) Actual prediction of the effects on pipelining application should be always carefully analyzed, since they heavily depend on the ratio of the operational elements' cost to the register cost.

(3) In order to fully explore the potential of pipelining, it is necessary to combine it with other transformations.

6.0 Conclusion

Classification of pipelining according to the objective (throughput and resource utilization) and the latency is introduced. For polynomial complexity pipelining classes, optimal algorithms are presented. For other classes both proof of NP-completeness and efficient probabilistic algorithms are presented. Both theoretical and experimental properties of pipelining are discussed. In particular, a relationship with other transformations is explored. Due to close relationship between software pipelining and functional pipelining presented here, all obtained results can be easily modified for software pipelining. Also, as the side result, the exact bound (solution) for iteration bound and optimal pipelining rate in systolic array processors are derived.

7.0 References

[Fet90] A. Fetweis, H. Meyr, L. Thiele: "Algorithm Transformations for Unlimited Parallelism", *IEEE International Symposium on Circuits and Systems*, pp. 1756-1759, New Orleans, 1990.

[Gar79] M.R. Garey, D.S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H.Freeman and company, New York, 1979.

[Goo86] G. Goossens, R. Jain, J. Vandewalle, H. De Man, "An optimal and flexible delay management technique for VLSI" in: C.I. Byrnes, A. Lindquist, "Computation and Combinational methods in system theory", pp. 409-418, North Holland, 1986.

[Goo89] G. Goossens, J. Wandewalle, H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", *26th Design Automation Conference*, pp. 826-831, Las Vegas, NV, 1989.

[Gyr84] E. Gyrczyc: "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Description", *Ph.D. Thesis*, Carleton University, 1984.

[Hen89] J.L. Henessy, D.A. Patterson: "Computer architecture: a quantitative approach", San Mateo, Calif.: Morgan Kaufman Publishers, 1989.

[Hwa84] K. Hwang, F.A. Briggs: "Computer architecture and parallel processing", McGraw-Hill, New York, NY 1984.

[Hwa91] C.-T. Hwang, J. -H. Lee, Y.-C. Hsu:"A Formal Approach to the scheduling problem in high level synthesis", *IEEE Trans. on CAD*, Vol. 10, No. 4, pp. 464-475, 1991.

[Jai89] R. Jain, "High-Level Area-Delay Prediction with Application to Behavioral Synthesis", *Technical Report 89-23*, University of Southern California, 1989.

[Jou89] N. Jouppi and D. Wall, "Available Instruction-Level Parallelism for Super-Scalar and Super-Pipelined Machines", *Proc. 3d International Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, pp.272-282, May 1989.

[Kim91] J.J. Kim, F.J. Kurdahi, N. Park: "Automatic-Synthesis of Time-Stationary Controllers for Pipelined Data Paths", IEEE International Conference on CAD, Santa Clara, CA, pp. 30-33, 1991.

[Kog81] P.M. Kogge: "The architecture of pipelined computers" Washington: Hemisphere Pub. Corp.; New York: McGraw-Hill, 1981.

[Kun88] S.Y. Kung: "VLSI Array Processors", Prentice Hall, Englewood Cliffs, NJ, 1988.

[Lam88] M.S. Lam: "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *ACM SIGPLAN*, 1988.

[Lam89] M. S. Lam: "A systolic array optimizing compiler", Boston: Kluwer Academic; Norwell, Mass., 1989.

[Lei83] C.E. Leiserson, F.M. Rose, J.B. Saxe, "Optimizing synchronous circuits by retiming", *Proceedings of the Third Conference on VLSI*, pp. 23-36, Computer Science Press, 1983.

[Lei91] C.E. Leiserson, J.B. Saxe: "Retiming Synchronous Circuitry", Algorithmica, Vol. 6, No. 1, pp. 5-35, 1991.

[Lin91] H.-P. Lin, D.G. Messerschmitt: "Finite State Machine has Unlimited Concurrency", *IEEE Trans. on Circuits and Systems*, Vol. 38, No. 5, pp. 465-475, 1991.

[Mes88] D. Messerschmitt, "Breaking The Recursive Bottleneck", in Performance Limits in Communication Theory and Practice, Kluwer Academic Publishers, 1988.

[Mli91] M.J.Mlinar, "Control Path/Data Path Trade-offs in VLSI Design", *Technical Report 91-16*, University of Southern California, 1991.

[Par88a] K. Parhi, "Algorithm and architecture design for high speed digital signal processing", *Ph.D. Dissertation*, University of California, 1988.

[Par88b] N. Park, A.C. Parker: "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on CAD*, Vol 7, No. 3, pp. 356-370, 1988.

[Par88c] N. Park, A.C. Parker: "Theory of Clocking for Maximum Execution Overlap of High-speed Digital Systems", *IEEE Trans. on Computers*, Vol. 37, No. 6, pp. 678-690, 1988.

[Par89] K.K. Parhi: "Algorithm transformation technique for concurrent processors", IEEE Proceedings of the IEEE, Vol. 77, No. 12, pp. 1879-1895. 1989.

[Pat89] D.A. Patterson, J.L. Henessy: "Computer architecture: a quantitative approach", San Mateo, Calif.: Morgan Kaufman Publishers, 1989.

[Pot91a] M. Potkonjak and J. Rabaey: "Optimizing the Resource Utilization Using Transformations", *Proc. IEEE ICCAD Conference*, Santa Clara, November 1991.

[Pot91b] M. Potkonjak: "Algorithms for High Level Synthesis: Resource Utilization Based Approach", *Ph.D. Dissertation*, University of California, Berkeley, 1991.

[Rab91a] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak: "Fast Prototyping of Data Path Intensive Architecture", *IEEE Design and Test*, Vol. 8, No. 2, pp. 40-51, 1991.

[Sto90] H. Stone: "High-performance computer architecture", 2nd ed. Addison-Wesley Pub. Co., Reading, MA, 1990.