Parallelism Analysis and Extraction of Digital Signal Processing Algorithms[1]

Khaled M. Elleithy and Alaaeldin A. M. Amin

Computer Engineering Department
King Fahd University of Petroleum and Minerals
Dhahran 31262, Saudi Arabia

Abstract

In this paper a new approach for parallelism analysis and extraction of Digital Signal Processing algorithms is introduced. The high level description of the input is given in CIRCAL. A dependency graph of the problem is constructed to check existence of cycles. Loops in the dependency graph are parallelized. The approach is illustrated by an example.

1: Introduction

Parallel processing is a trend in computing that has shown great performance advantages over the conventional sequential schemes. Description of parallel architectures in a formal parallel language is an essential need for better understanding and designing of parallel systems. This paper presents an environment for the specification, parallelism detection and transformation techniques of Digital Signal Processing architectures.

The basic structural features of algorithms or problem descriptions are dictated by their data and control dependencies. These dependencies refer to the precedence relations between computations that need to be satisfied in order to compute a problem correctly. The absence of dependencies indicates the possibility of simultaneous computations.

The Dependency Graph (DG) of an algorithm or a problem description is a directed graph G(V,E) where V is a set of vertices corresponding to the algorithm statements (states) and E is a set of edges (arcs) representing dependencies between these statements (states). Dependencies (edges of the DG) can be either control dependencies or data dependencies. Control dependency exists between two statements (states) where a condition (event) in one statement (state) controls the execution of the other statement (state). Data dependency between two statements occur when one statement (state) uses a data variable that was generated by another statement (state). Furthermore, when the value of a variable produced in a statement (state) is used in a subsequent statement (state), the data dependency is called *data-flow* dependency. If the value of a variable produced in one statement (state) has been previously used in another, or even the same, statement (state), the dependency is called *data-anti-dependency*. When two statements (states) produce the value of the same variable, the dependency is called *data-output* dependency.

CIRCAL is a formal concurrent process model developed for the purpose of rigorously describing and analyzing properties of highly complex concurrent systems such as those found in integrated circuit hardware [1-3]. The CIRCAL model provides a description language as the single medium to specify the behavior of hardware under analysis and the required timing properties which that hardware should posses.

This paper introduces a new design environment for synthesizing and parallelizing CIRCAL based algorithms. In section 2, an overview of the environment is presented. In section 3, the development of the dependency graph is given. Existence of cycles and loops are examined in sections 4 and 5. Loop parallelization is presented in section 6. In section 7, a final loop reduction step is presented. An example using the methodology is given in section 8.

2: Parallelism analysis and Extraction

Figure 1 shows the various stages used in our design environment.. A problem description in CIRCAL is accepted as the input. A dependency graph of the problem is constructed using the algorithm *Dependency GRAph*

[1]This work was supported by King AbdulAziz City of Science and Technology (KACST) grant AR-13-11.

Building or DEGRAB. Cycles existence is checked out using the algorithm *CYcles CHEcKing* or CYCHEK. If cycles do exist, it is necessary to find out if they were caused by one or more loops so that loop parallelization may be performed. Checking for existence of loops is achieved using the algorithm *LOops CHEcKing* or LOCHEK. Loop parallelization is done by the algorithm *LOops PARAllelization* or LOPARA. Finally, the dependency graph is reduced to eliminate additional redundancy. This is done by the algorithm *DEpendency Graph REDuction* (DEGRED).

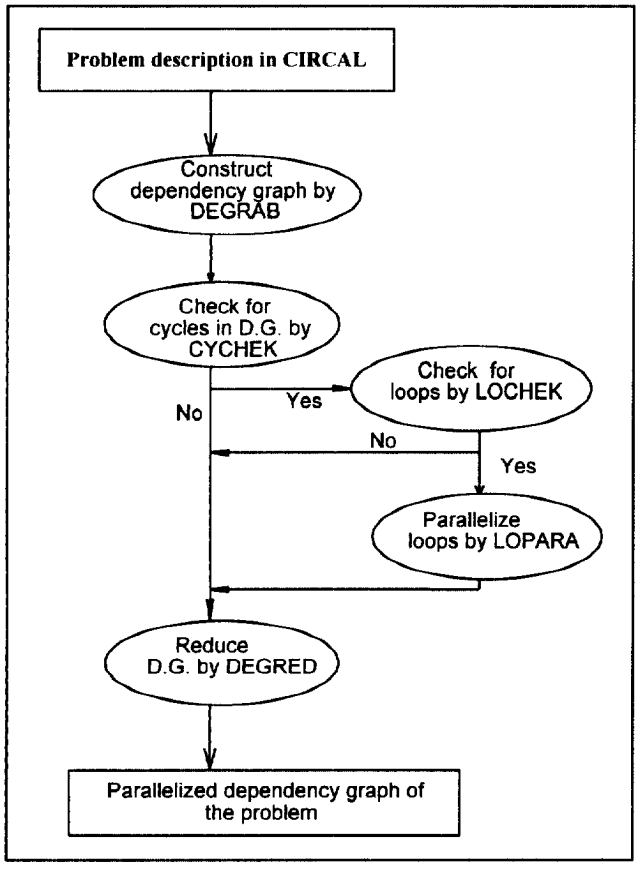

Figure 1. Overview of the Environment

3: Development of a Dependency Graph

This is the first stage as shown in Figure 1. The target here is to construct the dependency graph of the problem. In order to build a dependency graph of a problem described in CIRCAL, two approaches must be taken for this purpose. Each of these approaches deals with a different description level. The two levels are the *process level* and the *problem level*. The process level dependency approach finds out dependencies within a process. Dependency analysis here is visualized as a state transition analysis for the behavior of that process under analysis. The dependency analysis approach will track the state transition dependencies and build the dependency graph accordingly. This analysis is the first step of the overall dependency analysis of the problem. At the problem level, dependency in the whole problem is found out. Dependency analysis here is done over the processes that constitute the complete problem using CIRCAL composition operator. The dependency analysis approach will study the inter-process dependencies and build the dependency graph accordingly. This analysis is the second step of the overall dependency analysis of the problem.

The methodology used to build the dependency graph of a problem is called *DEpendency GRAph Building* or the DEGRAB algorithm. The algorithm starts at the process level to find out dependencies within each process. This is accomplished, for all processes of the problem, as a state transition analysis for the behavior of the process being studied and then building the dependency graph accordingly. This is followed by the problem level dependency approach which detects dependencies in the whole problem. The dependency analysis here is done over the processes that constitute the complete problem using CIRCAL composition operator. This analysis studies the inter-process dependencies and modifies the dependency graph accordingly.

In analyzing the complexity of the DEGRAB algorithm, as well as all other algorithms, we will consider both time and storage complexities. In addition, we will assume that the average number of states in a behavior of a process to be N, and the number of processes in the problem to be M. Furthermore, it is also assumed that when dealing with the DG at the process level, we assume that DG manipulations can be done simultaneously for all processes in the problem.

As for the DEGRAB algorithm, the first step will take time of O(N) since building DG for each process will require finding next states for every state (constant multiple of N). The second step, however, will take time of O(M) because the number of composition operators will be between two processes, which means that it is less than N, or a constant multiple of M in the worst case. Therefore, the total time taken by DEGRAB is of O(N+M). Looking at the storage complexity, building the DG will take a storage of O(NM) because the DG will have as many nodes as the number of states in all processes.

4: Cycles Existence in a Dependency Graph of a Problem

The target of this stage, as shown in Figure 1, is to check for the existence of cycles in the dependency graph of the problem. Cycles in a DG means that there is one or

more dependencies that are either *data-output* dependent, *data-anti-dependencies*, or control dependencies with control being passed back to some previous statement (state). These dependencies not only make it difficult to parallelize the problem but also cause a non-ordered sequence of the problem's operation. If cycles are found in a DG, the cause of their existence is checked to find out whether there is a loop in the problem description. If so, the loop is handled by LOPARA algorithm. Otherwise, the cycles are handled using DEGRED algorithm.

The methodology to check for cycles existence in the dependency graph of the problem is *CYcles CHEcKing* (or CYCHEK) algorithm. The algorithm goes through the DG and scans dependencies of all processes to find all existing cycles. Existence of a cycle in a process is found, by the algorithm, if we start with one node in the DG and follow all edges going out from it, and from its successors, continuously until we reach the same starting node . This cycle is saved for use in the other algorithms.

Analyzing the complexity of the CYCHEK algorithm, it will take time of $O(N^2)$ since following all edges from every node in the DG for each process, to check for cycles, will require time that is a constant multiple of N^2. Considering the storage complexity, saving of cycles requires a storage of $O(N^2)$ since for each node the number of cycles is less or equal to N with each cycle consisting of N states or less.

5: Loops existence in the problem description

As shown in Figure 1, the target of this stage is to check for loop existence in the problem description. As stated earlier, when cycles are found in a DG, these cycles must check to find out whether they are caused by a loop in the problem description. If this is the case, loop parallelization is accomplished using the LOPARA algorithm. The procedure used to check whether loops exist in the description of the problem is called *LOops CHEcKing* or the LOCHEK algorithm. The algorithm considers all cycles detected by CYCHEK and reviews the original description of each cycle to compare it with the loop construction in CIRCAL, which is one of the following:

(1) A parameter-change state: which is a state in which there is one or more variable parameters in its name. For example, S1(n) ⇐ ..... where n changes steadily in it, or in another state that leads to it.

(2) A condition state: which is a state in which there is one or more variables that is compared with a constant each time the state is executed. For example, S2 ⇐ ... if less(n,100) ... where n changes steadily in the state, or in another state that leads to it.

Analyzing the complexity of the LOCHEK algorithm, it will take $O(N^2)$ time since it studies each DG cycle for each process to check for loops which requires time that is a constant multiple of N^2. For the storage complexity, the algorithm will use storage which is O(N) to keep track of cycles and loops.

6: Loop Parallelization in the Problem Description

As shown in Figure 1, the target of this stage is to parallelize loops that exist in the problem description. If a loop body is detected in the problem description, it is parallelized as much as possible within the description. The corresponding DG will be modified accordingly using the LOPARA algorithm. After handling the cycles caused by loops in the DG, the resulting graph will be ready for reduction by the DEGRED algorithm.

The algorithm used for loop parallelization is called *LOops PARAllelization* or LOPARA. The algorithm takes the loops found by LOCHEK and uses the following parallelization techniques: loop fusion [4,5], alignment [6,7], cycle shrinking [8], interchanging [9,10], loop distribution [11,12], and vectorization [13,14]. Applying the appropriate parallization technique is determined by LOPARA. The parallelization algorithm will modify the CIRCAL description to remove the no longer needed loop elements, e.g. the loop index, step, etc. since these are covered by the resulting vectorized version of the loop.

The LOPARA algorithm studies each loop to check for the appropriate transformation technique, to apply that technique, to vectorize the loop, to modify the CIRCAL description, and to run the DEGRAB process level algorithm. Accordingly, the LOPARA algorithm has a time complexity of O(Y), where Y is the number of loops found by LOCHEK. Correspondingly, the storage complexity of LOPARA is only O(Y).

7: Dependency Graph Reduction

The target of this stage is to reduce the dependency graph of the problem. This is the last step in the complete methodology. The input to this stage is a DG whose loop dependency has been reduced as much as possible. The output is a reduced DG with minimum dependency.

The methodology to reduce a dependency graph of a problem is called the *DEpendency Graph REDuction* (DEGRED) algorithm. The algorithm will simplify the DG, and the corresponding CIRCAL description itself by eliminating the redundant states. It is the nature of CIRCAL that force the description to be concise enough. Perhaps the only redundancy that could exist is the one-state cycles where the event is a clock tick. Another

possibility is a state that has only one next-state caused also by a clock tick. Thus, the algorithm finds such states and modifies both the CIRCAL description and the DG.

The DEGRED algorithm has a time complexity of O(N), because it goes through all states in the DG to check if it has a redundancy. As for the storage complexity, the algorithm will not use storage more than O(N) since storage is needed for keeping track of cycles and next states.

8: Example

A CIRCAL-described problem is given in Figure 2. The problem in this example consists of two processes. The first process receives an input value on its input port ***a***. If the input value is not equal to 0, the process will output this value on its output port ***b*** after two clock ticks. The second process gets this value through its input port ***in*** and then adds this input to all contents of a two-dimensional array *c* of size 5 x 10. Then, it outputs the value true on its output port ***clk***.

```
PROCESS PROC1
   PROC1 ⇐ S1
   S1    ⇐ {t, a > X : noteq(X,0)} S2 + {t} S1
   S2    ⇐ {t} S3
   S3    ⇐ {t, b < X} ∧
END

PROCESS PROC2
   PROC2    ⇐ ST1
   ST1      ⇐ {t , in < Y} CLW(5,10)
   CLW(n,m) ⇐ if noteq(n,1) then {t , c(n,m) <
                (c(n,m) + in)} CLW(n-1,m)
                  + if and(eqs(n,1),noteq(m,1))
                then {true} CLW(5,m-1)
                  + if eqs(m,1) then {clk < true} ∧
END

PROB1 ⇐ PROC1 * PROC2 [ b / in ] - b
```

Figure 2: Initial CIRCAL-Description of the problem.

Applying the algorithm DEGRAB on the CIRCAL description gives the DG shown in Figure 3. The algorithm starts with the process level dependency to find out the dependency within each of the two processes. As shown in Figure 3, a node was constructed for every state and an edge was constructed for every transition, with the label on each edge showing the event or action causing this transition. The problem level dependency is then checked and an edge between the two processes is constructing. The labeling of this edge indicates the ports of both processes which were connected (***b*** and ***in***).

After applying algorithm DEGRED on the DG, modified by LOPARA, there is one one-state cycle, which is S1, and it is caused by the event {t}. Therefore, the edge representing this event was eliminated from the DG and the event and its next-state {t} S1 were eliminated from the CIRCAL description. In addition, the node corresponding to state S2 has only one outgoing edge with a label {t}. Therefore, this edge and the node from which it is outgoing were eliminated from the DG and the edge going to node S2 from node S1 was directed towards node S3 with addition of {t} to the label of this edge. Correspondingly, state S2 is eliminated from the behavioral CIRCAL description of process PROC1. Furthermore, references to S2 are replaced by state S3 preceded by an event {t} in the process description.

The modified CIRCAL description and the corresponding DG are shown in Figures 4 and 5, respectively.

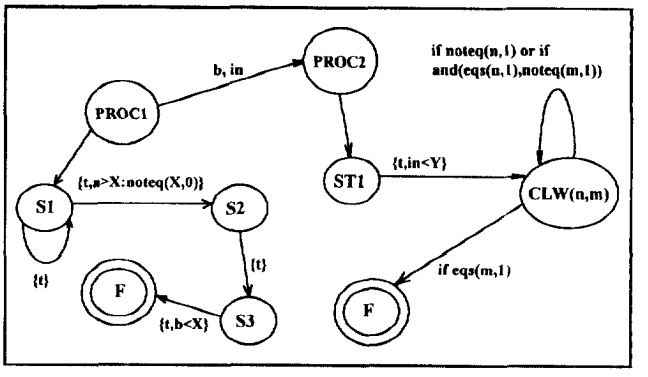

Figure 3. Initial DG of the problem.

```
PROCESS PROC1
   PROC1 ⇐ S1
   S1    ⇐ {t, a > X : noteq(X,0)} {t} S3
   S3    ⇐ {t, b < X} ∧
END

PROCESS PROC2
   PROC2    ⇐ ST1
   ST1      ⇐ {t , in < Y} CLW(5,10)
   CLW(5,10) ⇐ {t , c(5:1,10:1) < (c(5:1,10:1)
               + in)} {clk < true} ∧
END

PROB1 ⇐ PROC1 * PROC2 [ b / in ] - b
```

Figure 4: Final CIRCAL-Description of the problem.

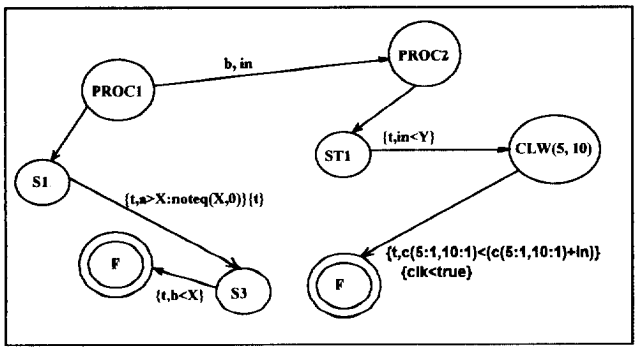

Figure 5. Final DG of the problem

9: Conclusions

In the process of designing a parallel architecture to realize some problem description, one of the required significant steps is to analyze the parallelism in the problem description and apply parallelization techniques to exploit the explicit, or the implicit, parallelism in the problem.

In this study, we have presented a new methodology for parallelism analysis and extraction of CIRCAL based algorithms.

Acknowledgments

The authors wish to acknowledge Mr. Muhammed Alhumagiani for designing the transformation algorithms used in this work.

References

1- Milne, G. J. CIRCAL: A Calculus for Circuit Description. *Integration, the VLSI Journal*, July 1983, pp. 121-160.

2- Milne, G. J. CIRCAL and the Representation of Communication, Concurrency, and Time. *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, April 1985, pp. 270-298.

3- Milne, G. J. The Formal Description and Verification of Hardware Timing. *IEEE Transactions on Computers*, vol. 40, no. 7, July 1991, pp. 811-826.

4- Loveman, D. Program Improvement by Source-to-Source Transformation. *Journal of ACM*, vol. 24, no. 1, January 1977, pp. 121-145.

5- Abu-Sufah, W., Kuck, D., and Lawrie, D. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Transactions on Computers*, vol. c-30, no. 5, May 1981, pp. 341-356.

6- Midkiff, S. and Padua D. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, vol. c-36, no. 12, December 1987, pp. 1485-1495.

7- Allen, R., Callahan, D., and Kennedy, K. Automatic Decomposition of Scientific Programs for Parallel Execution. *In Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, January 1987, pp. 63-76.

8- Polychronopoulos, C. D. Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design. *IEEE Transactions on Computers*, vol. 37, no. 8, August 1988, pp 991-1004.

9- Allen, R. and Kennedy, K. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, October 1987, pp. 491-542.

10- Wolfe, M. Advanced Loop Interchanging. *In Proc. 1986 Int. conf. on Parallel Processing*, St. Charles, III, Aug. 1986, pp. 536-543.

11- Banerjee, U., Chen, S., Kuck, D., and Towle, R. Time and Parallel Processor Bounds for FORTRAN-Like Loops. *IEEE Transactions on Computers*, vol. c-28, no. 9, September 1979, pp. 660-670.

12- Kennedy, K. and McKinley, K. Loop Distribution with Arbitrary Control Flow. *In Proc. Supercomputing '90*, IEEE Computer Society Press, Nov. 1990, pp. 407-417.

13- Banerjee, U., Eigenmann, R., Nicolau, A., and Padua, D. Automatic Program Parallelization. *Proceedings of the IEEE*, vol. 81, no. 2. Feb. 1993, pp. 211-243.

14- Wolfe, M. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press, 1989.