PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 40, NO. 2, PP. 77–85 (1996)

PIPELINED EXECUTION IN MULTI-USER SEQUENTIAL RECURSIVE LOOPS[1]

István JANKOVITS[2] and Tamás VISEGRÁDI[3]

Department of Process Control
Technical University of Budapest
H–1521 Budapest, Hungary

Received: June 10, 1995
Revised: January 8, 1996

Abstract

Recursive sections in a data path are constraints to the minimum value of data introduction latency (R) in that data path. Minimizing loop execution times is a way to increase the virtual data introduction frequency ($1/R$), with the minimum values set by the loops themselves. For a number of applications, another method is possible to decrease the restart time while actually increasing the total execution time of the loop (L_r). An advantage of this method is that the increase of execution time is an external constraint. This paper presents a description of such problems, with the steps of scheduling performed for an application of this type.

Keywords: high level synthesis, scheduling, recursive loops, constraints, scheduling methods.

Glossary[4]

Data path – A directed graph representation of data transitions in a problem. Graph nodes are operations, edges are data connections and dependencies.

Execution time – Time needed by an (elementary) operational unit to calculate its output value from its inputs. It is denoted by $t(i)$, where i is the number of operations.

Latency – Time difference between a set of data entering the data path and the output values *belonging to that set of input data* becoming available on the outputs (L).

Loop or Recursive loop – A section in a data path executed in an iterative way such that every iteration requires the result of the previous iteration (as initial value) and data from the data path. As the time between successive iterations of the loop may not be smaller than the total of all execution units in the loop (L_r), the loop takes data from the outside at most with the frequency equal to $1/L_r$.

Loop core – Hardware used in realizing the recursive part of the data path.

---

[1]The research outlined in this paper is supported by grant CP–940453
[2]Ph.D. student
[3]Graduate student
[4]More detailed definitions in [1]

Pipelined execution – Feeding a system with a restart time less than the total latency is available in some units. Any such unit works in an overlapped, *pipelined* way.

Restart time – The period time of new data input to the system (R).

Time steps or Time cycles – The time unit in time calculations, often expressed without dimension, i.e. 'a time of 3 [time cycles]'.

Client or User processes – Separate processes feeding data to the recursive cores.

Introduction

Recursive loops are considered to be unavailable to overlapped execution during the scheduling phase of ASIC design. This is caused by the special nature of recursive execution: an iterative algorithm may not be fed on the next data before the final result of the previous iteration is ready. As calculation of the next value requires a minimum of L_r cycles (L_r is the *total loop execution time*), a restart latency under $\max_{\substack{every \\ loop}} L_{r,i}$ time cycles is impossible in a data path containing recursive loops. However, there are some notable exceptions to the general case. In a special type of problems, recursive solutions are needed to calculate values of identical functions for different processes. Such an example is the centralized control of robots using the *computed torque technique.* Realizations of the computed torque technique require periodic calculations of a dynamic model for the robot joints to deal with changes in the environment. For the scheduling phase, this calculation may run simultaneously with torque calculations, in a *conditional execution branch* with a probability of $\frac{1}{N}$ if a calculation is required for every N complete cycle. For our study the probability-based conditional execution is not significant, so from now on the recursive subsection is treated on its own, without the actual realization of the conditional branches.

In the execution of a recursive loop, some parts of the loop are *busy* while the others are *idle.* As execution of such a loop is strictly sequential, the busy state 'propagates' along the data path in time (*Fig. 1*).

As time limits permit, it is possible to introduce new data to the start of the loop that runs through the loop without data conflict with the previous data. This overlapped execution exploits the inherent idle states of the loop. With such a structure, more than one process may use the recursive core, if a strict order of data introduction is maintained.

Pipeline Data-Flow Based Recursive Loop Scheduling

Most of the published methods handle the recursive loop latencies (L_r) as the minimum value of the restarting time (R). This constraint causes

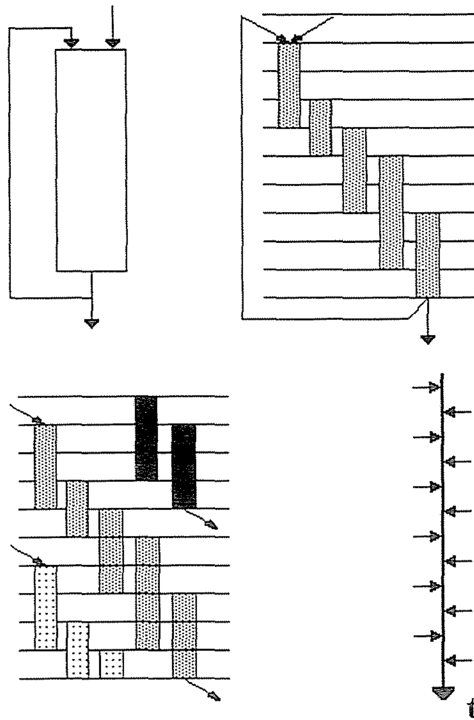

Fig. 1.

design methods to minimize the total execution time in loops. By using multiple-process recursive loops, we make possible for separate processes to share the same resources in such a way that the recursive core in process is realized only once. It means that the total loop sequence will be divided into smaller parts where each part can work parallel on a task and the parts rolling through the loop without breaking the rule of the recursion. To tune the loop and to control the periphery, the actual data ($d_j(i)$) and the dependent new data ($d_j(i+1)$) must arrive in the same time to the receptor elements in the loop. This means a synchronization problem in the sequences going through the loop-core (external synchronization) and an additional synchronization inside the loop (loop-core scheduling). Since the external synchronization is loop-core dependent this problem will be discussed after the classification of the recursive problems. As the basic synchronization tool is the delay (buffer), a recursive structure optimized for pipelined execution is likely to work slower than the non-pipelined loop.

This speed loss may be small, especially if loop execution time is much greater than 1, as the buffers cause a latency increase of 1 each.

Loop-Core Scheduling

It is obvious to discuss the loop-core scheduling problem for the simplest case when the whole graph consists of one loop core, therefore the external synchronization is eliminated (*Fig. 2*). As another facility, the graph contains just one receptor element at the beginning of the recursion. (It is easy to see that it does not mean any loss from the generality). The decision, whether the data must stay in the loop for another iteration or until the recursion has been done, will also be made by this receptor element. To avoid a data synchronization problem between the feedback of the loop output (initial value of the next iteration) and the new data, the loop latency (L_r) should be equal to an integer multiple (N) of the restarting period (R):

$$L_r = (N + 1)^* R. \tag{1}$$

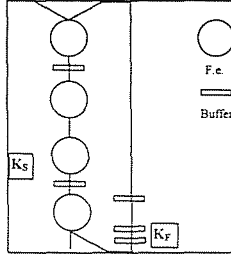

Fig. 2.

From [3] it is known that L_r is the function of R and R, N (N means number of processes using the loop) are given values, (1) will fulfil just for special cases, which causes the synchronization problem inside the loop. To solve this problem the difference between L_r and R must be realized as an extra delay in the loop-core. In this way (1) will be modified:

$$L_r + K = (N + 1)^* R, \tag{2}$$

where K is the number of the inserted buffers (delay element). The number of the inserted buffers are determined by 2 factors:

$$K = K_s + K_f \tag{3}$$

- *Scheduling the loop-core for the aimed restarting time* (K_s): for this problem any known scheduling method can be used ([1,3,6,7]) to tune the opened loop. With [1] K_s is the sum of the number of the inserted buffers to the places where the transfer scores make it necessary and the buffers are inserted before the multiplied elements.
- *Synchronization between the input and the feedback of the loop* (K_f): K_f can be calculated from (2) and (3):

$$K_f = (N+1)^* R - L_r - K_s, \tag{4}$$

where $L_r + K_s \leq (N+1)^* R$.

If $L_r + K_s \geq (N+1)R^*$ (the scheduling inserted more buffers than needed for the whole synchronization), then the feedback of the loop will be slower than the input side, so this additional problem must be taken into consideration during the external synchronization. In this case:

$$K_f = 0. \tag{5}$$

Classification of Recursive Problems

If we ignore the exact composition of a recursive loop core, data propagation has three main classes based on the number of iterations. The number of iterations a data set spends in a recursive loop (loop depth, γ_i) is either a finite (constant or variable) or an infinite value.

1) Iterative solutions of differential equations (*Fig. 3*) are remaining in an iterative state or a (usually) finite number of loop transitions, while the exact loop depth is data dependent. For these calculations no general data transition rule can be given. The time needed for the first data set to leave and a new set of data to be introduced to the iteration is

$$\min_k (R(\gamma(k)n + k - 1)) \tag{6}$$

2) The set of problems with a fixed finite number of iterations is typical for the equivalent of FOR loops (which generally run for a constant number of times). An example of this is the class of FIR filters (*Fig. 4*), where the number of iterations is related to the order of the filter, which is a constant based on the nature of the problem. As with these problems $\forall i : \gamma(i) \equiv \gamma_0$, total data transition time may be calculated as:

$$\frac{x}{n-1}\gamma n R + (n-1)R, \tag{7}$$

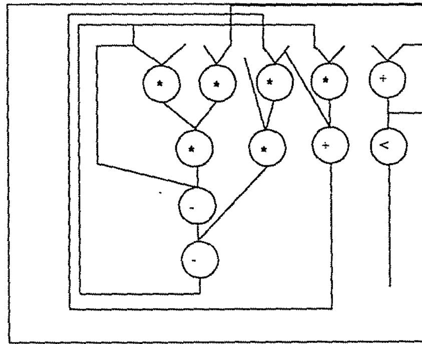

Fig. 3.

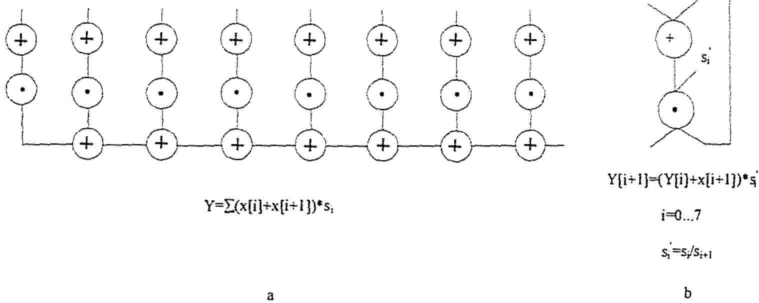

Fig. 4.

which is less than

$$\gamma L_H x \tag{8}$$

time required without loop overlapping (x is the number of data sets put through the recursive calculations). From (7) and (8) the condition can be derived, when our method is worth using. If the ratio of (8) and (7) is less than 1, then the new design solves the problem faster.

$$\frac{\frac{x}{n-1}\gamma nR + (n-1)R}{x\gamma L_r} << 1. \tag{9}$$

From the meaning of x it is obvious that x is much greater than the other values in the equation, therefore:

$$\frac{x}{n-1}\gamma nR >> (n-1)R. \tag{10}$$

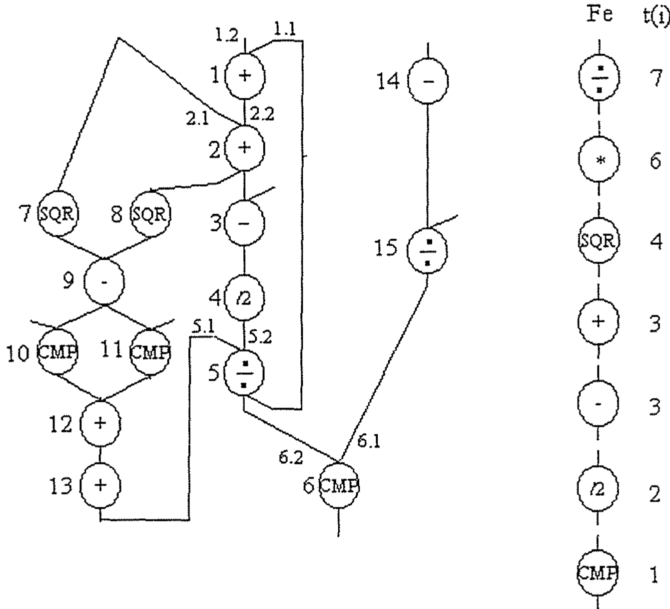

Fig. 5.

Neglecting the left side of (10) from (9), and applying some equivalent modifications:

$$\frac{R}{L_r} << \frac{n-1}{n}. \tag{11}$$

The right side of (11) shows that the more the restarting period is decreased against the latency, the more efficient structure will be achieved and from the left side it can be seen that as more and more data are introduced to the structure, the result can be more and more efficient.

3) Infinite loop depth is presented in the case of continuous calculations. Robots, for example, need to calibrate the dynamic model of their environment periodically, as long as the robot is moving. In this case the overlapped loop execution is usually slower than the non-pipelined version as modifying the loop core for overlapping increases L_H. One of the advantages is the reduced cost of hardware components, which is generally possible as the nature of calculations is identical for every process. Absolute time gain is therefore expressed in terms of realization costs, with a typical 3-process robot subsystem presenting only one fully utilized loop core instead of three partially idle systems.

External Synchronization

The external synchronization will be illustrated on an example (*Fig. 5*) which is used for data filtering in voice-transfer processes. This recursive algorithm contains all the possible data conflicts that can arise. The synchronization problems are negotiated here in the time order which eliminates any iterative steps during the scheduling process. In this example two different, but not independent loop-cores exist (The first follows the 1, 2, 3, 4, 5, and the second, the critical path, follows the 1, 2, 8, 9, 10, 12, 13, 5 sequence.) In the first step the shorter loop must be tuned to the critical recursive path, inserting extra buffers to the shorter path. The calculation of the buffer-number is described in [3]. In this example it means that 7 buffers must be placed before the 5th element, to the 5.2 part.

The synchronization problem can be simply solved by one of the algorithms published in [1,3–6] before the 2, 3, 9, 10, 11, 12 and the 15 elements, handling the whole graph with opened feed-backs. The value of the L_r used in (2) will be given just after this step, since the synchronization may introduce delay elements to the critical recursive path, which makes L_r longer. Another class of the synchronization problems is when the receptor element uses the result of the loop-core (6th receptor in *Fig. 5*). In this case the algorithm is loop-dependent, because the result of the loop is available in different time cycles depending on the type of the loop (see in section 'Classification of Recursive Problems'). This problem can be eliminated by inserting a shift register which can produce any contained data in any time step depending on the control of the system. The length of the shifter is determined by the worst case (the longest possible loop execution time). For the problems of the 1st class (see in section "Classification of recursive problems") it modifies the structure of the control logic in [1]. In this case the control logic picks up the suitable data from the shift register when the recursive loop finished an iteration.

Conclusion

Recursion, being one of the three fundamental description methods (Turing-machine, grammar and μ-recursive functions), is a feature often found in practical problems. Numerical methods, used in many applications are usually based on recursive solutions. Scheduling algorithms of recursion, as it is common to treat the loop as a solid execution block, maximizes data introduction (restart) frequency. This solid model does not enable the scheduler to increase system throughput.

The method presented in this paper deals with recursive systems in the other way: by separating loop components, in a special set of recursive systems it is possible to tune the loop for an overlapping mode. This pipelined execution enables the systems to process data at a higher frequency, increasing data throughput and decreasing processing time. The algorithm of recursive loop tuning is not bound to any of the scheduling methods. It is possible to insert it as an extension to most of the schedulers.

References

1. ARATÓ, P. – BÉRES, I. (1994): A High-Level Datapath Synthesis Method for Pipelined Structures. *Proc. of The 8th Symposium on Microcomputer and Microprocessor Application.*
2. ARATÓ, P. (1990): Logic Synthesis of Special-Purpose Hardware Structures Based on a Pipelined Dataflow Model. Department of Process Control, TUB, Report.
3. ARATÓ, P. – JANKOVITS, I. – RUCINSKI, A.: Time Scaled High-Level Synthesis for Pipelined Data-flow Structures, *Proceedings of ATW'94.*
4. CHENG-TSUNG HWANG – JIAHN-HURNG LEE – YU-CHIN HSU (1991): A Formal Approach to the Scheduling Problem in High Level Synthesis, IEEE Transactions on Computer-Aided Design, Vol. 10, No. 4, April 1991.
5. High-Level VLSI Synthesis, Edited by Raul Camposano & Wayne Wolf, Kulwer Academic Publisher, 1991.
6. ROSENSTIEL, W. – KRAMER, H.: Scheduling and Assignment in High Level·Synthesis.
7. CASAVANT, A. E. – KI SOO HWANG – MCNALL, K. N.: PISYN–High-Level Synthesis of Application Specific Pipelined Hardware, .
8. PAULIN, P. G. – KNIGHT, J. P.: Force-directed Scheduling for the Behavioral Synthesis of ASIC's, *IEEE Transactions on Computer-Aided Design,* 1989/6.