A Multi-Pattern Scheduling Algorithm

Yuanqing Guo Cornelis Hoede Gerard J.M. Smit
University of Twente, Department of EEMCS
P.O. Box 217, 7500AE Enschede, The Netherlands

Abstract

In a coarse-grained reconfigurable architecture, the function of resources such as Arithmetic Logic Units (ALU) can be reconfigured at run-time. The traditional scheduling algorithms always assume a fixed resource pattern. In this paper, we propose an efficient scheduling algorithm assuming that different resource patterns are given. The multi-pattern scheduling algorithm is based on the list scheduling algorithm. The complexity of the proposed algorithm is linear in the number of given patterns.

1 Introduction

The scheduling problem is concerned with associating tasks of a Control Data Flow Graph (CDFG) to clock cycles such that certain constraints are met. This is an optimization problem, and is specified in several ways depending on the constraints: (1) Unconstrained scheduling finds a feasible schedule that obeys the precedence constraints on the graph. (2) Time-constrained scheduling minimizes the number of required resources when the number of clock cycles is fixed. (3) Resource-constrained scheduling minimizes the number of clock cycles when the number of resources is given.

There are many resource-constrained scheduling algorithms existing in the literature. Most of them assume that the constraint is given in terms of the number of tasks of each type. For example, there are two multipliers and four adders in the system. If the combination of concurrent existing resources is called *a pattern*, the traditional work on resource-constrained scheduling algorithms uses only one fixed pattern.

Recently reconfigurable systems have drawn more and more attention for its combination of flexibility with programmability. In a reconfigurable system, the types and number of resources can be changed by reconfiguring the resources at run-time. In a coarse-grained reconfigurable system such as the Montium [1], designed by the University of Twente, different patterns are allowed when the total number of computational cores is fixed. A Montium tile has five computational cores which, for instance, can be configured to compute two additions and three multiplications during the first clock cycle, and one addition, two substractions and two bit-or operations during the second clock cycle. One Montium tile allows up to 32 different configurations, i.e., patterns. In this paper we propose an efficient scheduling algorithm assuming given multiple resource patterns.

Most scheduling problems are NP-complete problems. To solve the scheduling problems, exact algorithms, which find optimal solutions, or heuristic algorithms, which find feasible (possibly suboptimal) solutions have been used. Most exact algorithms employ Integer Linear Programming (ILP) to compute the optimal solutions [5]. Two commonly used heuristic algorithms are: list scheduling [2][3] and force-directed scheduling[4]. These algorithms iteratively select and schedule one operation at a time within an appropriate clock cycle. Among them, list scheduling is more often used in resource constrained scheduling problems.

2 Problem Description

On a CDFG a node represents an operation and a directed edge denotes a conditional dependency between two operations.

The *depth of a node* is defined as the length of the largest path of the node in the data flow graph. We assume that the node without successors has depth equal to 1.

Given a set of patterns $P_1, P_2, \cdots P_R$, the scheduling problem is to assign nodes on a CDFG to clock cycles such that the conditional dependencies between nodes are satisfied, and within each clock cycle, the needed resources are according to the resources defined by one of the given patterns, and furthermore, the number of clock cycles is minimized.

3 A Multi-Pattern List Scheduling Algorithm

3.1 Algorithm Description

A list based algorithm maintains a candidate list of candidate nodes, i.e., nodes whose predecessors have already been scheduled. The candidate list is sorted according to a priority function of these nodes. In each iteration, nodes with higher priority are scheduled first and lower priority nodes are deferred to a later clock cycle. Scheduling an node within a clock cycle makes other successor nodes candidates, which will then be added to the candidate list.

For multi-pattern scheduling, for one clock cycle, not only nodes but also a pattern should be selected. The selected nodes should not use more resources than the resources presented in the selected pattern. For a specific candidate list C and a pattern P_i, a *selected set* $S(P_i, C)$ is defined as the set of nodes from C that will be scheduled if the resources are given by P_i.

The multi-pattern scheduling algorithm is given in Figure 1. In total two types of priority functions are defined here, the *node priority* and the *pattern priority*. The former is for each node in the graph and the latter is for scheduling a candidate list by one specific pattern.

1. Compute the priority function for each node in the graph.
2. Get the candidate list.
3. Sort the nodes in the candidate list according to their priority functions.
4. Schedule the nodes in the candidate list from high priority to low priority using each given pattern.
5. Compute the priority function for each pattern and keep the one with highest priority function.
6. Update the candidate list.
7. If the candidate list is not empty, go back to 3; else end the program.

Figure 1: Multi-Pattern List Scheduling Algorithm

3.2 Node Priority

In the algorithm, the following priority function for graph nodes is used:

$$f(n) = s \times depth + t \times \#\text{direct\_successors} + \#\text{all\_successors} \quad (1)$$

Here #direct_successors is the number of the successors that follow the node directly, and #all_successors is the number of all successors. Parameter s and t are used to distinguish the importance of the factors. s and t should satisfy the following conditions:

$$\begin{aligned} s &\geq \max\{t \times \#\text{direct\_successors} \\ &\qquad + \#\text{all\_successors}\} \\ t &\geq \max\{\#\text{all\_successors}\} \end{aligned} \quad (2)$$

These conditions guarantee that the node with largest depth will always have the highest priority; For the nodes with the same depth, the one with more direct successors will have higher priority; For the nodes with both the same depth and the same number of direct successors, the one with highest number of successors will have highest priority.

3.3 Pattern Priority

Intuitively for each clock cycle we want to choose the pattern that can cover most nodes in the candidate list. This leads to a definition of the priority function for a pattern P corresponding to a candidate list C.

$$F(P, C) = \text{number of nodes in} S(P, C). \quad (3)$$

On the other hand, the nodes with higher priorities should be scheduled before those with lower priorities. That means that we prefer the pattern that covers more high priority nodes. Thus we define the priority of a pattern as the sum of priorities of all nodes in the selected set.

$$F(P,C) \quad = \sum_{n \in S(P,C)} f(n). \qquad (4)$$

3.4 Example

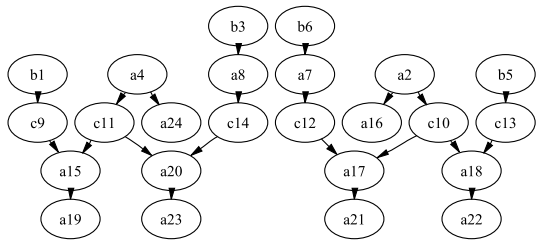

Figure 2: Multi-pattern scheduling example: 3FFT algorithm

Table 2: Final scheduling

clock cycle	scheduled nodes
1	a2,a4,b6
2	a7,a24,b3,c10,c11
3	a8,a16,b5,c12
4	a17,b1,c13,c14
5	a18,a20,a21,c9
6	a15,a22,a23
7	a19

We explain the algorithm with the help of the 3-point Fast Fourier Transform (3FFT) algorithm. The CDFG of 3FFT consists of additions, subtractions and multiplications, as shown in Figure 2. The nodes denoted by "a"s are additions; while those with "b" represent subtractions and the nodes with "c" multiplications. Two patterns are assumed to be given here: pattern1 = "aabcc" and pattern2 = "aaacc". The scheduling procedure is shown in Table 1. Initially, there are six candidates: {a2, a4, b1, b3, b5, b6}. If we use pattern1 {a2, a4, b6} will be scheduled, and if we use pattern2 {a2, a4} will be scheduled. Because the priority function of pattern1 is larger than that of pattern2, pattern1 is selected. For the second clock cycle, pattern1 covers nodes {a7, a24, b3, c10, c11} while pattern2 covers {a7, a16, a24, c10, c11}. The difference between the use of the two patterns lies in the difference between b3 and a16. If we use the pattern priority function of Equation (3), the two patterns are equally good. The algorithm will pick one at random. If we use Equation (4) as pattern priority function, pattern1 will be chosen because the depth of b3 is larger than that of a16. The final scheduling result using Equation (4) as pattern priority is shown in Table 2.

3.5 Complexity comparison with fixed-pattern list scheduling

For each clock cycle, the multi-pattern scheduling algorithm schedules the nodes in the candidate list using every given pattern. The rest is the same as in the traditional resource constrained list scheduling algorithm which uses a fixed pattern. The computational complexity of the multi-pattern scheduling algorithm is therefore $O(R \times$ *Complexity of fixed-pattern list algorithm*$)$, where R is the number of given patterns.

4 Experimental Results

We ran the multi-pattern scheduling algorithm on 3-, 5- and 15-point Fast Fourier Transform (3FFT, 5FFT and 15FFT) algorithms. For each algorithm two different pattern priorities were tested. See Table 3 for the experimental results. The number gives the number of clock cycles needed. From the simulation results we have the following observations:

- As more patterns are allowed the number of needed clock cycles gets smaller. This is the benefit we get from reconfiguration.
- In most cases the pattern priority function Equation (4) leads to better scheduling than the priority function given by Equation (3). However because of the greedy strategy of the list algorithm, there is no single priority function which can guarantee to find the best solution.

Table 1: Scheduling Procedure

clock cycle	candidate list	pattern1 =“aabcc”	pattern2 =“aaacc”	selected pattern
1	a2,a4,b1,b3,b5,b6	a2,a4,b6	a2,a4	*1*
2	b1,b3,b5,c11,a24, a16,c10,a7	a7,a24,b3,c10, c11	a24,a16,a7,c11, c10	*1*
3	a8,a16,b1,b5,c12	a8,a16,b5,c12	a8,a16,c12	*1*
4	b1,c14,a17,c13	a17,b1,c13,c14	a17,c13,c14	*1*
5	a18,a20,a21,c9	a18,a20,c9	a18,a20,a21,c9	*2*
6	a15,a22,a23	a15,a22	a15,a22,a23	*2*
7	a19	a19	a19	*1*

Table 3: Experimental results: Number of Clock cycles in Final Scheduling

	3FFT		5FFT		15FFT	
number of nodes	24		62		544	
pattern priority	Eq. (3)	Eq. (4)	Eq. (3)	Eq. (4)	Eq. (3)	Eq. (4)
“aabcc”	8		17		153	
“aabcc”,“aaacc”	7	7	16	16	139	139
“aabcc”,“aaacc”,“aaaac”	7	7	16	15	143	134
“aabcc”,“aaacc”,“aaaac”,“aabbc”	6	7	14	14	127	117
“acccc”,“abbbc”,“aaaaa”,“aabbc”					119	110
”abcaa”,“ccccc”,“aaaaa”,”bbbbb”					109	109

- The selection of patterns has very strong influence on the scheduling results!

5 Conclusions and Future Work

This paper presents an efficient scheduling algorithm for reconfigurable architectures where multiple resource patterns are allowed. The multi-pattern scheduling algorithm is a list based algorithm which chooses the best pattern as well as the best nodes according to priority functions for each clock cycle. Two pattern priority functions have been tested in the simulation. One of them shows better results than the other. The algorithm has low computational complexity, i.e., it is linear in the number of patterns. Due to the use of more patterns, fewer clock cycles are needed for an application.

From the simulation we can see that the choice of patterns is very important for scheduling! This leads to another research topic: pattern selection, which will be addressed in our future work.

References

[1] Paul M. Heysters, Gerard J.M. Smit, E. Molenkamp: “A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems”, *The Journal of Supercomputing*, Vol 26, No. 3, Kluwer Academic Publishers, Boston, U.S.A., November 2003, ISSN 0920-8542.

[2] B.M. Pangrle and D.D. Gajski, “Design Tools for Intellegent Compilation,” *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No. 6, Nov.1987, pp. 1098-1112.

[3] T.C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research,* Vol.9, No.6, Nov.1961. pp. 841-848.

[4] P.G. Paulin and J.P. Knight, “Algorithms for High-Level Synthesis,” *IEEE Design and Test of Computers,* Vol.6, No.4, Dec. 1989, pp.18-31.

[5] L.J.Hafer and A.C. Parker, “A Formal Method for the specification, Analysis, and Design of Register-Transfer Level Digital Logic,” *IEEE Trans. Computer-Aided Design,* Vol. CAD-2, No.1, Jan. 1983, pp. 4-18.