Complexity of Scheduling in High-Level Synthesis

C.A.Mandal[1] *P.P.Chakrabarti*[2] *S.Ghose*[2]

[1] Department of Computer Science & Engineering
Jadavpur University
Calcutta 700032, INDIA
[2] Department of Computer Science & Engineering
Indian Institute of Technology
Kharagpur 721302, INDIA

Abstract

This work examines the complexity of scheduling for high level synthesis. It has been shown that the problem of finding the minimum time schedule for a set of chains of operations of two types using two processors, one of each type, is NP-complete. However, for two chains only, a polynomial time algorithm can been obtained for scheduling with two processors. The problem of scheduling a rooted binary tree of two operation types on two processors, one of each type, has been shown to be NP-complete. It has also been proved that absolute approximations for schedule length minimization or processor minimization are NP-complete. A related resource constrained scheduling problem has also been shown to be NP-hard.

Key Words: Scheduling, NP-complete problems, High-Level Synthesis

1 Introduction

The problem of scheduling is an important one in the automation of VLSI design. It is a primary problem in high-level synthesis (HLS) of VLSI systems [1]. The scheduling problem surfaces soon after the behavioural specifications have been converted to the intermediate form [2] which is usually in the form of a set of data flow graphs. These may be in the form of directed acyclic graphs (DAGs) which contain the dependencies between the operations. A DAG contains nodes which denote the type of operation to be performed (like +, - , etc.) and the precedence constraints on the operations. The DAGs are themselves connected by directed edges to form a flow graph, which depicts the flow of control between them. Scheduling is applied to each DAG in the flow graph. The purpose of scheduling is to assign the operations to time steps so that the precedence constraints are satisfied. The objective may be to minimize of the number of operators given a deadline for finishing all the tasks or it may be to minimize the schedule length given a limit on the number (and type) of processors. Several heuristic algorithms have been proposed for such types of problem formulations [3, 4] in VLSI design. There are also a few approximate algorithms with non-trivial error bounds. Scheduling problems, in general, have been hard to solve optimally in the sense that many of them have been proved to be NP-complete. This paper addresses the issue of complexity of the scheduling problem in high-level synthesis.

Extensive studies have been made on the complexity and algorithms for problems in job shop scheduling [5, 6]. A variety of scheduling problems are encountered in HLS, some of which are a little different from the ones studied in job shop scheduling. A common feature of scheduling for HLS is the presence of precedence constraints and the non-premptive character of the operations; the execution times of the operations are sometimes integral, rather than unit. Operations may be of one type or of multiple types. The functional units (FU), which play the role of processors, may be simple, i.e., capable of realizing only one type of operation or complex, i.e., capable of realizing multiple types of operations. They may also be heterogenous, i.e., two FUs having non-identical capabilities.

There are several complexity results in scheduling theory which concern scheduling with precedence constraints [6, 7, 8]. They are, however, mostly related to DAGs which have only one type of operation. It is known that for DAGs with only one operation type the problem of minimum length scheduling with one or two processors can be solved in polynomial time; in fact it can be done in linear time [8]. But the problem for arbitrary m ($m > 1$) processors is NP-complete [6, 7], which usually signifies that there is little or no chance that a polynomial time algorithm exists [7]. For a fixed number of processors the complexity issue is still open. However, if the DAG is a forest (a set of disjoint trees) then the problem can be solved in polynomial time [7].

On the other hand, in VLSI scheduling we usually have more than one type of operation in the DAGs and the operators (or functional units) may either perform a particular operation or a set of them. If all operators are capable of performing all types of operations then the problem reduces to the original scheduling problem mentioned above. However, this is usually never the case. Moreover, in many situations where shared resources exist, resource constrained scheduling is an important problem to be tackled. For example, if several operations have to read data form a read only memory (ROM) having only one port we have a case of resource constraint.

In this paper the complexity issue of scheduling multiple operation DAGs on heterogenous processors has been considered. Specifically, we have considered the simple case where the DAG is a collection of chains (which are DAGs where every node has at most one immediate predecessor and at most one immediate successor), there are two types of operations and two processors, one for each type of operation. This problem has been shown to be NP-complete for the general case of m, $(m > 2)$ chains. For $m = 1$, the problem is trivial, for $m = 2$, a polynomial time solution has been shown to exist. It may be noted that the problem of scheduling DAGs on homogenous processors can be solved in polynomial time for upto two processors.

The above problem has been identified as a special case of a host of other scheduling problems, all of which have thus been shown to be NP-hard. In section 3, the proof technique used to derive the previous result has been used to derive the complexity of two other problems which are: i) Schedule length minimization of rooted binary trees of two operations using two processors, one of each type; and ii) Schedule length minimization of single operation chains in the presence of a single resource which an operation either uses wholly or not at all, during the period of its execution. A previous result for the second problem [6] states that this problem is NP-hard for a DAG.

Subsequent to this, the complexity of approximation of various scheduling problems concerning schedule length minimization as well as processor minimization have been considered. A few open problems have also been discussed.

2 The Complexity of Scheduling Two Operation Chains

2.1 The Problem

We are given a set of tasks $t_1, t_2, \ldots, t_n$ each of which takes one time unit. The tasks are of two types, 1 and 2. There are two processors, one of each type. An operation can execute only on the processor of its type. There is a precedence constraint on the tasks which is restricted to be a collection of chains. There is a deadline D for completion of all the tasks. We wish to solve the decision problem of whether all the tasks can be scheduled on appropriate processors satisfying the precedence constraints and meeting the deadline D.

2.2 The Reduction

The reduction is from Exact Cover by three-sets [7]. Given a set A=$\{a_1, a_2, \ldots, a_n\}$, n is divisible by 3, and a collection of sets $B_1, B_2, \ldots, B_n$ where each B_i is a subset of A and has three elements, we construct a graph G consisting of a set of $(n+1)$ chains of nodes of two types 1 and 2. The total number of nodes is $6n^2 + 42n$. We then show that G has a schedule of length $3n^2 + 21n$ if and only if there are $n/3$ sets in $B_1, B_2, \ldots, B_n$ whose union is A.

We use a reduction technique which is similar to the one proposed by Berger and Lenore [8] for proving the NP completeness of scheduling with $<$ and $=$ constraints for DAGs. A chain of the graph is defined by a sequence of the type 1-1-2-1 which means a type 1 node followed by a type 1 node followed by a type 2 node followed by type 1 node. We define the following types of chain structures. These structures are illustrated in figures 1 and 2. In these figures a type 1 operation is represented by a single circle, while a type 2 operation is represented by two concentric circles.

Ring This consists of three nodes of the type 1-1-2.

Ring Slot This consists of three nodes of the type 2-2-1. (Note that Ring Slot is the dual of a Ring in the sense that they can be scheduled in tandem.)

Key We shall have n different types of Keys, one for each element a_i. Each Key consists of $n+6$ nodes. The first five nodes of a Key are of the pattern 1-1-1-2-2. The next $n+1$ nodes are all of type 2 except the i-th one which is 1 for the i-th Key.

Key Slot This is the dual of the Key. There are again n different such types. Each consists of $n+6$ nodes. The first five are of the type 2-2-2-1-1. The next $n+1$ nodes are all of type 1 except the i-th one which is 2 for the i-th Key.

We now describe the set of chains formed by the reduction. The constraint forest consists of a set of $(n+1)$ chains. For each set $B_i = \{a_{k_i}, a_{l_i}, a_{m_i}\}$, $k_i < l_i < m_i$, in the set cover problem we have a chain C_i which consists of a Ring (which we call the header of a chain) followed by three Keys of the types k_i, l_i and m_i respectively. This construction ensures that the Key chains corresponding to a_{k_i}, a_{l_i}, and a_{m_i} of B_i must be scheduled in that order. This gives us n chains. The final structure, i.e., the $(n+1)-$th chain, is the special chain called *time-line* consisting of $3n^2 + 21n$ nodes having the following four conceptual stages, appearing in sequence:

Figure 1: A Ring chain and a Ring Slot chain.

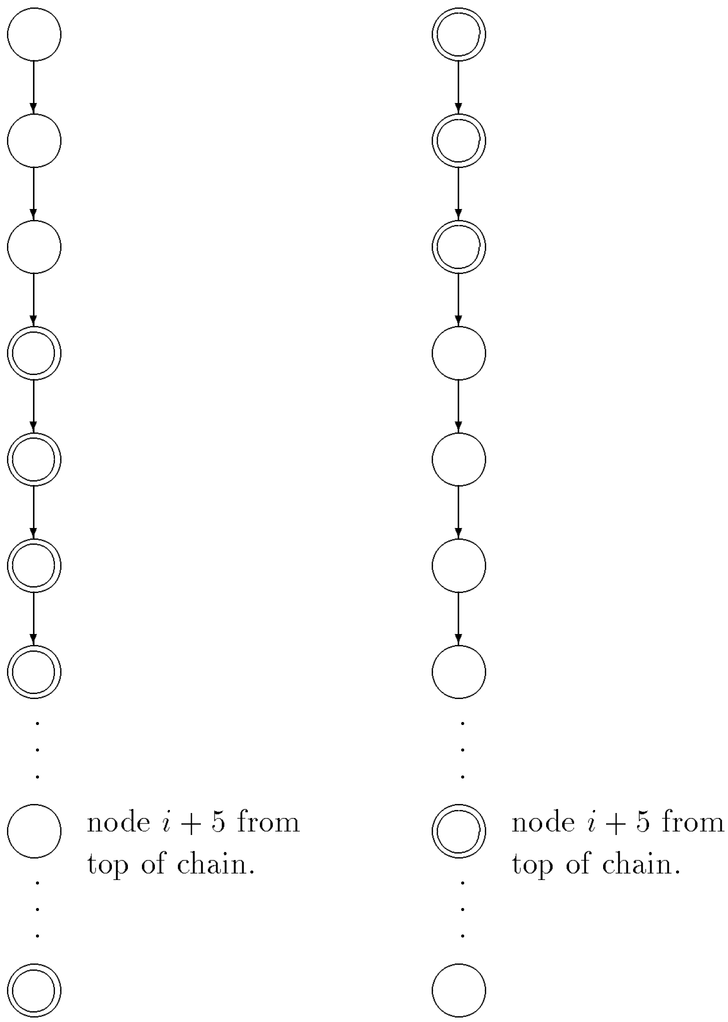

Figure 2: A Key chain and a Key Slot chain for a_i.

Stage 1:[READY] It consists of a chain of n/3 Ring Slots.

Stage 2:[FIT] It consists of a Key Slot for each $a_1, a_2, \ldots, a_n$ in that order.

Stage 3:[RELEASE] It consists of a chain of 2n/3 Ring Slots.

Stage 4:[PACK] This consists of a set of 2n Key Slots. For each a_i (in the order $a_1, a_2, \ldots, a_n$) there are $d_i - 1$ Key Slots of type i where d_i is the number of sets among $B_1, B_2, \ldots, B_n$ where a_i occurs.

The length of the time-line is $3n^2 + 21n$. We define $D = 3n^2 + 21n$. We may note here that by the above construction, we are asking for an "exact schedule" which implies that at no time step will we have any processor idle. Based on the above construction we now prove that the chain scheduling problem is NP-complete.

Lemma 1 *If the set A has an exact cover by 3 sets, then we can schedule the chains within the deadline D.*

Proof: We perform the scheduling in the following manner. We have to schedule the time-line as it has been given. We first schedule the headers (the Rings) of those chains C_i which correspond to the sets B_i occurring in the exact cover in stage 1. These chains are then scheduled perfectly in stage 2 of the time-line. The remaining chain headers are scheduled in stage 3 and the rest of the chains are scheduled in stage 4. □

Lemma 2 *If we can schedule in the given deadline D then A has an exact cover by 3 sets.*

Proof: We shall prove this by showing that the only way in which we can obtain a schedule of length D (if it exists) is the manner in which it has been described in the previous lemma. We shall show that if there is a deviation from this then we cannot get a schedule of length D in the sense that we will never obtain a perfect schedule. We shall prove it case by case for every stage considering the first place where deviation occurs.

Consider stage 1. This is a sequence of n/3 2-2-1 chains (Ring Slots). Suppose it is correctly scheduled up to the $(i-1)$-th such Ring Slot. Thus, till the end of the $(i-1)$-th Ring Slot, where a type 1 node occurs in the time-line, we must have exactly $(i-1)$ Rings from the headers of the chains C_1, C_2, …, C_n scheduled. Now consider the i-th Ring Slot (2-2-1) at this stage of the time-line. If for the 2-2 portion of this chain we have a deviation then there may occur four cases. The *first* is where two type 1 nodes from headers of two different C_i's are scheduled. The *second* is one where two type 1 nodes are scheduled one from a header and another from the first 1 of a ready chain. (A ready chain is one whose header Ring is already scheduled). The *third* case is one where two type 1 nodes from the same ready chain is scheduled and the *fourth* case is one where two type 1 nodes from two different ready chains are scheduled. In each of the cases we can show that the type 2 processor corresponding to the Slot adjacent to the 1 in the 2-2-1 sequence of the i-th Ring Slot in the time-line (in stage 1) will go empty (that is no task will be schedulable here). Thus we will not get a perfect schedule.

If there is no deviation for the 2-2 portion of the 2-2-1 chain then the 1 portion must also be free of deviation.

Consider stage 2. Similarly, here we will again assume that the schedule is correct (that is as mentioned in the proof of Lemma 1) up to the Key Slot corresponding to a_{i-1}. Let it deviate at the Key Slot corresponding to a_i. Now consider the 2-2-2 part of the Key Slot of this stage, in the time-line. The following deviations might occur.

1. Only type 1 nodes from more than one Key are scheduled on the idle type 1 processor in these three time steps. In this case the type 2 processor in the first time step of the succeeding 1-1... part of the time-line, will go idle.

2. Exactly one type 1 node from a Ring of one of the remaining 2n/3 chains is scheduled and type 1 nodes from one or two ready chains are scheduled. The type 2 processor in the next time step goes empty.

3. Two type 1 nodes from a Ring of one of the remaining 2n/3 chains and one type 1 node from a ready chain Key are scheduled. Only the type 2 node of the Ring is available for scheduling on a type 2 processor. Depending on whether it is scheduled the next time step or the next to next time step, the type 2 processor goes empty in one of the two time steps.

4. Three type 1 nodes, one each from the Rings of three of the remaining 2n/3 chains are scheduled. The type 2 processor goes empty in the next time step.

If there is no deviation in the 2-2-2 part of the time-line then there is no scope for deviation of the subsequent 1-1-...-1 part till the appearance of the first 2. It is clear that any deviation will cause the type 1 processor to go empty when the 2 appears on the time-line. Again there is no scope of deviation in the remaining 1-1-...-1 part of this Key Slot. As explained while considering stage 1, if a processor goes empty then the schedule cannot be perfect.

In the absence of any deviation in the READY and FIT parts of the time-line, the exact cover may be easily extracted. The arguments for the other two stages follow along similar lines. □

We are therefore ready to state the resultant theorem.

Theorem 1 *The problem of scheduling a set of chains corresponding to two different types of operations in two processors (one for each type of operation) given a deadline D is NP-complete.*

Proof: That this problem is in NP can easily be shown. That it is NP-hard follows from the construction and lemmas 1 and 2. □

We now present a small example to illustrate the generation of an instance of a scheduling problem from an instance of an exact problem.

Example 1 Consider the set $A = \{abcdef\}$, and the sets $B_1 = \{abc\}$, $B_2 = \{acf\}$, $B_3 = \{bcd\}$, $B_4 = \{adf\}$, $B_5 = \{ace\}$ and $B_6 = \{def\}$.
For each B_i we construct a chain C_i as follows:
$C_1 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.1.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2,
$C_2 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.2.2.1.2,
$C_3 =$ 1.1.2. 1.1.1.2.2. 2.1.2.2.2.2.2. 1.1.1.2.2. 2.2.1.2.2.2.2. 1.1.1.2.2. 2.2.2.1.2.2.2,
$C_4 =$ 1.1.2. 1.1.1.2.2. 1.2.2.2.2.2.2. 1.1.1.2.2. 2.2.2.1.2.2.2. 1.1.1.2.2. 2.2.2.2.2.1.2,

$C_5 = 1.1.2.\ 1.1.1.2.2.\ 1.2.2.2.2.2.2.\ 1.1.1.2.2.\ 2.2.1.2.2.2.2.\ 1.1.1.2.2.\ 2.2.2.2.1.2.2$ and
$C_6 = 1.1.2.\ 1.1.1.2.2.\ 2.2.2.1.2.2.2.\ 1.1.1.2.2.\ 2.2.2.2.1.2.2.\ 1.1.1.2.2.\ 2.2.2.2.2.1.2,$

The time-line T is as follows:

$T = 2.2.1.\ 2.2.1.$
2.2.2.1.1. 2.1.1.1.1.1.1. 2.2.2.1.1. 1.2.1.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.1.2.1.1.1.
2.2.2.1.1. 1.1.1.1.2.1.1. 2.2.2.1.1. 1.1.1.1.1.2.1.
2.2.1. 2.2.1. 2.2.1. 2.2.1.
2.2.2.1.1. 2.1.1.1.1.1.1. 2.2.2.1.1. 2.1.1.1.1.1.1.
2.2.2.1.1. 2.1.1.1.1.1.1. 2.2.2.1.1. 1.1.2.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.2.1.1.1.1.
2.2.2.1.1. 1.1.2.1.1.1.1. 2.2.2.1.1. 1.1.1.2.1.1.1.
2.2.2.1.1. 1.1.1.2.1.1.1. 2.2.2.1.1. 1.1.1.1.2.1.1.
2.2.2.1.1. 1.1.1.1.1.2.1. 2.2.2.1.1. 1.1.1.1.1.2.1.

The first line of T is the *ready* part, the next three lines correspond to the *fit* part, the fifth line is the *release* part, while the remaining lines correspond to the *pack* part. In this example B_1 and B_6 exactly cover A, this is also reflected in the existence of an exact schedule where C_1 and C_6 are scheduled in the *ready* and *fit* part of the time-line, while the remaining chains are scheduled in the *release* and *pack* part of the time-line. The schedule has been shown in figure 3. The time-line starts with a Ring Slot, indicated as R_s and the i-th chain starts with a Ring indicated as R^i. For B_i the corresponding Key and Key Slot are indicated as K_i and K_i^s, respectively, in figure 3. □

3 Related Results

Theorem 1 imples several other results. It follows from the theorem that scheduling chains of k types of operations (k-operation chains), $k \geq 3$, on k processors, one of each type, is NP-hard. This can be proved by simply augmenting the construction used for the proof with $k-2$ processors for the new types of operations. Now only the first two processors are required not to go idle in any time step. We do not elaborate here. Also if jobs have un-equal processing times, the problem is also NP-hard. However, since results similar to these generalizations (but not to the original 2-operation chain scheduling problem) are already available in scheduling theory [5, 6, 7], we will not discuss them here. We will discuss some other results which can be derived from the construction used and the result proved in theorem 1.

The corresponding optimization problem of finding the smallest length schedule is, therefore, also NP-hard. Since this problem is hard it becomes obvious that even when the operators are mixed (in the sense they can perform a set of operations) the problem is NP-hard. It is, therefore, a direct corollary that the problem of optimally scheduling a single two operation tree (or DAG) using two processors, one of each type, is NP-hard. This can be constructed from the previous problem of chains by constructing a single tree or a DAG from the chains.

3.1 Scheduling Rooted Binary Trees of Two Operations on Two Processors

Since most operations are binary operations, we consider the special case when the DAG is a binary tree. We consider dependencies which satisfies the following properties.

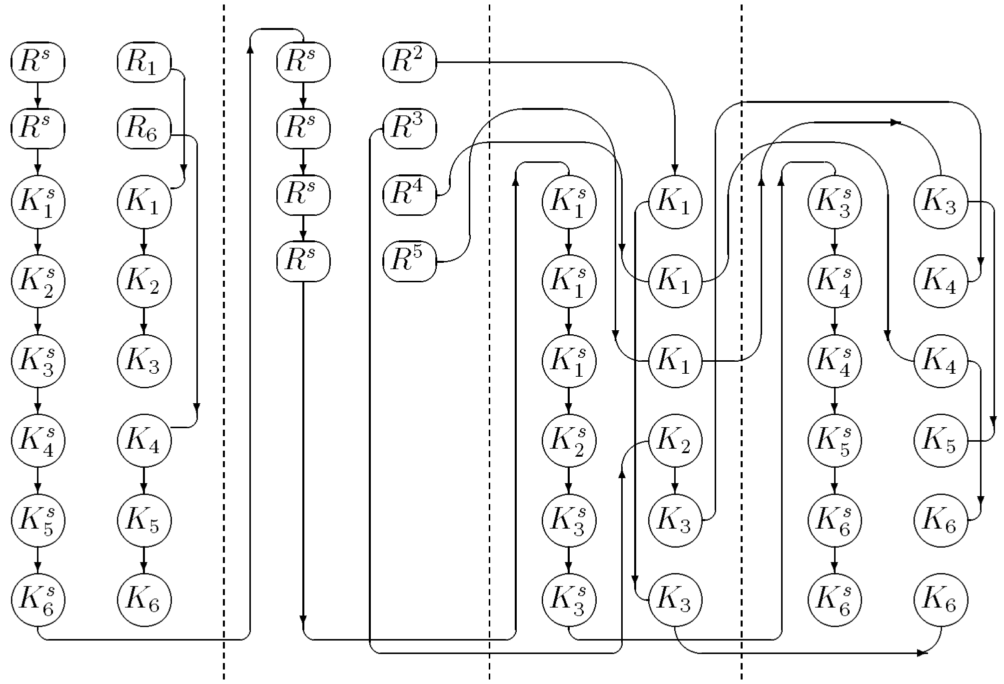

Figure 3: Schedule of time-line and other chains for two processors one of each type.

A node can have up to two successors and all but the a designated node called the root node will have exactly one predecessor. The root node does not have any predecessor. The corresponding graph of such depedencies will be referred to as a rooted binary tree. For the case of scheduling a rooted binary tree the following result may be obtained.

Theorem 2 *The problem of scheduling a single rooted binary tree of two operation types on two processors, one of each type, is NP-complete.*

Proof: A rooted binary tree may, in fact, be constructed from the chains that have been described above, as follows. Let the root node of the tree be R. R has two successors T_1 and C_1. A node T_i has one successor T_{i+1}, if $i < n$. The successor of T_n is the first node of the time-line. A node C_i has two successors, C_{i+1} and the first node of the i-th chain, if $i < n$. The successor of C_n is the first node of the n-th chain. The nodes T_i, $1 \leq i \leq n$ are of type 1. The nodes C_i, $1 \leq i \leq n$ are of type 2. The root node is arbitrarily chosen to be of type 1. The deadline is now taken as $D' = 3n^2 + 22n + 1$. As before a schedule is attempted on two processors. In all schedules of this rooted binary tree, the type 2 processor in the first time step, where the root node is scheduled, will remain idle. In the lines of lemma 1 and lemma 2, the hardness result can be proved by considering a perfect schedule in the remaining $D' - 1$ time steps of the deadline. $\square$

In practice we are likely to encounter such dependency structures where the dependency relation is just the reverse of a rooted binary tree. Such a dependency structure may be referred to an inverted rooted binary tree.

Corollary 3 *The problem of scheduling an inverted rooted binary tree of two operation types on two processors, one of each type, is NP-complete.*

Proof: Follows along similar lines as theorem 2. □

It may be noted that single operation DAGs can be scheduled on upto two processors in polynomial time optimally.

3.2 Resource Constrained Scheduling

Next we consider a resource constrained scheduling problem (RCS). As mentioned earlier, the operators themselves are generally not considered to be resources. An important sub-problem in this category is that of *scheduling with two processors, unit execution times, and one resource with limit 1* [6]. In this problem some of the operations will use the resource while the others will not. As before the operations are considered to have unit execution times. A previous result for this problem (in [6]) states that this problem is NP-hard for a DAG. The method used to prove theorem 1 may be used to prove the hardness of this problem, for the case where the scheduling constraints are a set of chains.

Theorem 4 *Scheduling m chains having only one type of operation with two processors, unit execution times, and one resource with limit 1, when the precedence constraints are a set of chains, is NP-complete.*

Proof: The proof may be derived in the lines of theorem 1 using exactly the same construction. The operations that require the resource are labeled as type 1 operations, while operations which do not require the resource are labeled as type 2 operations. □

A practical version of RCS comes up in DPS scheduling of DAGs where input/output needs to be performed. Due to pin limitations it becomes necessary to restrict the number of ports in the system. The port, now, is a system resource, some of the operations that need to do input/output will use these resources while the others will not. If there is only one port then we are directly faced with the 0/1 RCS problem described above.

3.3 Scheduling Only Two 2-operation Chains on Two Processors

We now consider the problem of scheduling 2-operation chains using one processor of each type where we have exactly two such chains. We show that this problem can be solved optimally using a polynomial time algorithm.

Let the two chains be $X = < x_1, x_2, \ldots, x_m >$ and $Y = < y_1, y_2, \ldots, y_{m'} >$ where $m + m' = n$ and $t(x_i)$ or $t(y_i)$ denotes the operation type associated with the respective node.

Observation 5 *The problem of finding a minimum length schedule of two chains using two processors, one of each type, is equivalent to finding the maximum compatible subsequences of the two chains. Two nodes are said to be compatible if $t(x_i)$ is not equal to $t(y_j)$.*

The observation is due to the fact that we map the nodes on the maximum compatible subsequences to the same time steps. Then the two chains can always be scheduled in $m+m'-C$ steps where C is the total length of the maximum compatible subsequences of X and Y. Infact, for any schedule, in any time step where both the processors are busy, the pair of nodes of the two chains which are mapped on the processors are of different types and can participate in the formation of a compatible subsequence. Therefore, no schedule is possible where both the processors are busy for C' steps, where $C' > C$, because this would violate the premise that the length of the maximum compatible subsequences is C.

The compatibility problem may be solved using the following recursive decomposition, similar to that used in [9] for the Longest Common Sequence (LCS) problem:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } t(x_i) \neq t(y_j) \\ max\ (c[i, j-1], c[i-1, j]) & \text{if } i,j > 0 \text{ and } t(x_i) = t(y_j) \end{cases}$$

where c[i,j] is the length of the maximum compatible subsequences of $< x_1, x_2, \ldots, x_i >$ and $< y_1, y_2, \ldots, y_j >$.

This recursion equation can be solved by dynamic programming in O(mm') time using the the dynamic programming technique in [9]. This shows that the two chain problem is solvable in polynomial time. However, the problem of r-chains for a fixed r $(r > 2)$ still remains open.

4 The Question of Approximation and Other Open Problems

The result proved in Theorem 1 makes it quite apparent that in practice optimality has to be discarded if we are looking for polynomial time solutions. We therefore have to rest content with getting sub-optimal solutions in polynomial time. But here again we wish to seek guarantees in the solution.

We know that list scheduling provides a very good bound on the quality of solutions in the sense that the schedule it provides for single operation type DAGs, given p processors, never takes more than twice the number of time steps of the optimal. In the case of DAGs with k types of operations, however, with p processors for each operation type, if we use list scheduling then we can do the following:

1. We may convert the DAG to a single operation type one.
2. Schedule the DAG by list scheduling with kp processors to obtain a list schedule of no more than twice the optimal length.
3. Consider those time steps where more than p nodes of the same type have been scheduled in the same time step.
4. Sequence these nodes in different time steps using no more than p processors of a particular type at one time step.

This way we can schedule a k operation type DAG using p processors of each type in no more than $2k$ time steps of the optimal schedule. However, can we get a bound which is better than the above obvious generalization of list scheduling, especially a bound which is independent of k? This remains an open problem.

The other important problem is in the area of minimization of the number (or cost) of processors. In such cases we wish to find algorithms which can schedule in a given number of time steps with minimum number of processors. Can we obtain an approximate scheme which requires no more than a small constant times the minimum number required? Very interestingly, even in the case of single processor DAGs such results have not come to our notice. Even for popular algorithms like FDS [3] there are no known theoretical bounds on performance.

Consider DAGs with two types of operation for which an optimal schedule, exists in D time steps using two processors, one for each type of operation. We can obtain a schedule in D time steps with four operators, two of each type, as follows. We can convert the DAG to a single operation type one. Then we can optimally schedule this in linear time using the HLS algorithm [8] of Gabow. This will give a schedule in time $T \leq D$. Using two more processors, one for each operation type, we can easily find a schedule within D steps as required.

However, the general case remains quite open if really strong bounds are to be obtained.

There is another type of approximation which is known as absolute approximation. In such cases we expect to obtain solutions which are bounded by opt + k where opt is the optimal and k is a constant. These are more difficult to obtain and in this particular case it is not difficult to show that they are NP-hard for nearly all types of scheduling problems.

Theorem 6 *Absolute approximation of scheduling DAGs is NP-hard for the problem of minimization of schedule length.*

Proof: Suppose a k approximation exists for the schedule length minimization problem. Take any DAG D. Make $k+1$ copies of it. Chain them up so that they must be scheduled one after another. Set the number of processors to m > 2. If this gives a k absolute approximate schedule then at least one of the $k+1$ copies of D must have been scheduled optimally for m processors. This proof is valid even for single operation type DAGs. □

Theorem 7 *Absolute approximation of scheduling DAGs with multiple operation types, given a deadline, is NP-hard for the problem of minimization of the number of processors.*

Proof: We take any DAG D, of a single operation type. Assume a k absolute approximate algorithm exists. Make $k + 1$ copies of D resulting in a set of disjoint DAGs. Give each DAG a different operation type; all nodes of a particular DAG have the same operation type. Try to find a schedule satisfying the deadline. If a k absolute approximate solution exists then one of the DAGs must have been scheduled with the minimum number of processors. □

However, the complexity issue of absolute approximation of single operation DAGs where we wish to minimize the number of processors remains another open problem.

5 Conclusion

In this paper we raise the issue of the complexity of various scheduling problems, which have practical application, and their approximation algorithms. The problems are different from normal scheduling problems in the sense that there are distinct types of operations which cannot be assigned to all processors. We have proved the NP-hardness for a simple case of scheduling of chains on two processors. A dynamic programming solution is suggested for the two chain scheduling problem. The problem of 0/1 resource constrained scheduling on single operation chains has also been shown to be NP-hard. We have also shown that absolute approximation of scheduling is NP-hard for both schedule length minimization or processor minimization. In the case of approximations we give a simple relative approximation using the well known list scheduling technique and an even better bound for two processors (section 4). However, we acknowledge that these algorithms are rudimentary, if not obvious, and stress the need for having much more improved bounds and deeper results. In the case of processor minimization no proper bounds have come to our notice. Though algorithms which appear to perform well have been proposed [3] , there has been no proper theoretical analysis of their performance. This remains a very important area of future work with several open questions remaining, some of which have been listed in this paper.

References

[1] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *COMPILERS Principles, Techniques and Tools*. Addison-Wesley Publishing Company, June 1987.

[3] P. G. Paulin and J. P. Knight, "Force-directed scheduling for asics," *IEEE Transactions on Computer Aided Design*, June 1989.

[4] A. C. Parker, J. T. Pizarro, and M. Mlinar, "Maha: A program for data path synthesis," *Proceedings of the 23rd Design Automation Conference*, 1986.

[5] E. W. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, *Sequencing and Scheduling: Algorithms and Complexity, in Handbook in Operations Research and Management Sciences*, vol. 4: Logistics of Production and Inventory. North-Holland, 1993.

[6] E. G. Coffman Jr eds, *Computer and Job Shop Scheduling Theory*. John Wiley & Sons., 1976.

[7] M. Garey and D. Johnson, *A guide to the theory of NP-completeness*. Freeman, San Fransisco, 1979.

[8] B. Berger and L. Cowen, "Complexity results and algorithms for $\{<, \leq, =\}$– constrained scheduling," in *Proceedings of 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, C. A.*, pp. 137–147, 1991.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.