Complexity of Scheduling in Synthesizing Hardware from Concurrent Action Oriented Specifications

Gaurav Singh[1], S. S. Ravi[2], Sumit Ahuja[3] and Sandeep Shukla[4]

[1] FERMAT Lab, Virginia Tech, Deptt. of Electrical and Computer Engineering, Blacksburg, VA 24060, USA.
`gasingh@vt.edu`
[2] Univ. at Albany - State Univ. of New York, Department of Computer Science, Albany, NY 12222, USA.
`ravi@cs.albany.edu`
[3] FERMAT Lab, Virginia Tech, Deptt. of Electrical and Computer Engineering, Blacksburg, VA 24060, USA.
`sahuja@vt.edu`
[4] FERMAT Lab, Virginia Tech, Deptt. of Electrical and Computer Engineering, Blacksburg, VA 24060, USA.
`shukla@vt.edu`

Abstract. Concurrent Action Oriented Specifications (CAOS) formalism such as *Bluespec Inc.'s* Bluespec System Verilog (BSV) has been recently shown to be effective for hardware modeling and synthesis. This formalism offers the benefits of automatic handling of concurrency issues in highly concurrent system descriptions, and the associated synthesis algorithms have been shown to produce efficient hardware comparable to those generated from hand-written Verilog/VHDL. These benefits which are inherent in such a synthesis process also aid in faster architectural exploration. This is because CAOS allows a high-level description (above RTL) of a design in terms of atomic transactions, where each transaction corresponds to a collection of operations. Optimal scheduling of such actions in CAOS-based synthesis process is crucial in order to generate hardware that is efficient in terms of area, latency and power. In this paper, we analyze the complexity of the scheduling problems associated with CAOS-based synthesis and discuss several heuristics for meeting the peak power goals of designs generated from CAOS. We also discuss approximability of these problems as appropriate.

Keywords. Hardware Synthesis, Concurrent Action Oriented Specifications (CAOS), Scheduling, Complexity, Peak Power.

1 Introduction

High-level synthesis is the process of converting a behavioral (algorithmic) specification of a design into its register transfer level (RTL) description [1]. The three most important phases of any high-level synthesis process are *operation scheduling, resource allocation and resource binding.* During operation scheduling, each operation of a design is mapped to an appropriate control step in which it will be executed. Resource allocation determines the number of various resources (adders, multipliers, registers) that should be used to implement the design. During resource binding, each operation is bound to a particular resource which will be used to implement it.

Scheduling affects allocation and binding phases and hence efficient scheduling of the operations of a design is very important in producing designs that are efficient in terms of area, latency and power. The problem of scheduling during traditional high-level synthesis using CDFGs (control data flow graphs) can manifest in different forms as per the design requirements. A resource-constrained scheduling problem involves the minimization of the total number of control steps required to execute the operations of a design given the fixed number of each resource. On the other hand, time-constrained scheduling problems involve minimizing the number of instances of each resource required given the fixed number of control steps. Both these versions of the scheduling problem are known in general to be **NP-complete**. The best known algorithms for solving these problems have exponential time complexity, and hence heuristics are often used to solve such problems.

Recently, a new approach to high-level synthesis from Concurrent Action Oriented Specifications (CAOS) [2,3] has been proposed which is based on the idea that any hardware design can be described in terms of concurrent atomic actions or transactions [4]. During the execution of the designs generated from CAOS-based synthesis, one or more actions can be scheduled to execute in a given time slot. Each time slot corresponds to one cycle of a hardware clock. Since power management is essential in contemporary designs, efficient scheduling of the actions of a design can be used to minimize its **peak** as well as **dynamic power** consumption. Peak power of a design can be defined as the maximum total power consumed (due to the switching activity) during one time slot, whereas dynamic power is the total power consumed during the overall execution of the design.

In this paper, we analyze the complexity of the scheduling problems associated with CAOS-based synthesis. We focus on the peak power requirements of a design, and propose several heuristics for efficient scheduling of the actions of a CAOS-based design such that its peak power constraints are satisfied. The paper is organized as follows. Section 2 presents related work done in the area of low-power high-level synthesis. Section 3 briefly describes CAOS-based synthesis approach. In Section 4, we provide some preliminary definitions related to the complexity theory of algorithms. Section 5 presents the complexity analysis of the scheduling problem associated with CAOS-based synthesis without any peak power constraints. This paves the way to the analysis under peak power constraint in the next section. In Section 6, various versions of the scheduling problem under peak power constraints are discussed and efficient heuristics are presented in each case. Section 7 concludes this chapter with a brief summary.

2 Related Work

References [2,3] formulate the problems of power-optimal synthesis for CAOS and present heuristics targeting the minimization of peak power and dynamic power in designs generated from CAOS. The low-power heuristics presented in [2,3] are based on re-scheduling and factorization of actions. In this paper, we further extend the preliminary analysis of the peak power minimization problem in [2,3] by presenting a thorough analysis of various peak power optimization problems related to CAOS-based design.

In the past, most of the other work done in low-power behavioral synthesis is based on the techniques which use CDFGs as inputs to the synthesis process [5,6,7,8]. In [5], an integer linear programming model (ILP) for peak power minimization under latency constraints is presented. In [6], a power management technique targeted towards high-level synthesis of data-dominated behavioral descriptions is proposed. Reference [7] presents a scheduling algorithm which aims to maximize the idle times for functional units. Reference [8] proposes a controller re-specification technique based on re-designing the controller logic used to reduce the activity in the components of the datapath. Reference [9] presents a comprehensive high level synthesis system for reducing power consumption in control-flow intensive as well as data-dominated circuits. In [10], the authors address the issue of managing transient power consumption through the choice of appropriate architectures during high-level synthesis. A framework for simultaneous reduction of energy and transient power components during behavioral synthesis is proposed in [11]. [12] formally describes various algorithms on power macro-modeling based on regression analysis and power minimization through behavioral transformations, scheduling, resource assignment and hardware/software partitioning and mapping.

Given that the above mentioned low-power techniques [5,6,7,8,9,10,11,12] are related to the CDFG-based models, the work presented in this paper is distinct. Our work is related to the CAOS-based synthesis approach. One advantage of CAOS is that it does not lose the parallelism/concurrency inherent in the specification, and allow the synthesis mechanism to infer more parallelism.

The problem of resource-constrained scheduling for low-power objective has been addressed in [13,14]. These approaches use CDFGs to first determine the mobility of various operations based on the ASAP and ALAP schedules. Using the computed mobilities and other relevant factors, priorities are assigned to various operations. Based on these priorities, the operations are then scheduled in each time slot such that the power consumption of the design is reduced. However, as mentioned earlier, the strategies presented in this paper are not based on the static schedule (CDFGs) and hence do not have knowledge of the future time slots due to the nature of CAOS. In each time slot, appropriate actions are dynamically chosen to be scheduled based on the CAOS semantics such that the low-power goals of the design are met maintaining the correct functionality.

3 Concurrent Action Oriented Specifications

In the **CAOS** formalism, the behavior of the design is described using a collection of guarded atomic actions (transactions) at a level of abstraction higher than RTL. Each action consists of an associated condition (called the *guard* of that particular action) and a body. An action executes only when its associated guard is true, and the body of an action operates on the state of the system. The state can be explicitly defined by the designer in the high-level description of the system or, in other cases, can be inferred from that description. *Bluespec Compiler* [15] is an example CAOS-based synthesis. In *Bluespec*, there is no need to infer the state because the designer explicitly instantiates all the state elements of the system (like registers, FIFOs, memories etc.). This model then undergoes synthesis to generate the RTL code [15].

An *action A* in the concurrent action-based description of a design can be written as,

$$\begin{aligned} &\mathit{Action}\ A : g(s) \rightarrow \{ \\ &s_1 = \ b_1(s,i); \\ &s_2 = \ b_2(s,i); \\ &s_3 = \ b_3(s,i); \} \end{aligned}$$

Here, *g(s)* is the guard associated with the action A. s is the set of state elements of the design such that $s_1 \in s$, $s_2 \in s$ and $s_3 \in s$. The body of the action contains three statements of the form $s_j = b_j(s,i)$. $b_j(s,i)$ computes the subset of the next state of the system using the current values of the elements of s and the current input i.

The actions are atomic in the sense that either all the computations corresponding to the body of an action finish successfully, or none of them executes. Two actions are said to be in *conflict* with each other if they update one or more of the same state elements. Multiple actions can execute concurrently as long as they do not conflict, thus exploiting maximum parallelism present in the design. The system execution stops when no *guard* evaluates to *True*.

Following is an example from [2], which illustrates how a specific hardware can be described in *Bluespec System Verilog* (which is based on CAOS) and also provides an idea of the execution semantics of CAOS.

Example

Figure 1 shows an example of an *action-oriented* description of a LPM (Longest Prefix Match) module [15]. It takes 32-bit IP addresses, looks up the destination (32-bit data) for each IP address in a table in a memory, and returns the destinations. The module is pipelined and the results are returned in the same order as requests. The memory is also pipelined and has a fixed latency of L cycles. One of the possible implementations of a LPM module is in the form of a Circular Pipeline as shown below.

```
              **Action 1 (Input):**
(true) → {
        IPaddr = fifo1.deq();
        token = compBuffer.getToken();
        fifo2.enq(token,IPaddr[15:0],state0);
        RAM.readRequest(baseAddr + IPaddr[31:16]); }

              Action 2 (Complete):
(isLeaf(d = RAM.readResult())) → {
        (token, IPaddr, s) = fifo2.deq();
        compBuf.done(token, d);
        RAM.readAck(); }

              Action 3 (Circulate):
(!(isLeaf(d = RAM.readResult()))) → {
        (token, IPaddr, s) = fifo2.deq();
        fifo2.enq(token, IPaddr, s+1);
        RAM.readRequest(computeAddr(d,s,IPaddr));
        RAM.readAck();}
```

Fig. 1. Circular pipeline specification of the LPM module design using concurrent actions.

In the above example, *Action 1* represents the *Input* stage which takes an IP address *IPaddr* from *fifo1* and a token from the *Completion Buffer*. It then places a tuple (token, IPaddr[15:0], state0) into *fifo2* and enqueues a memory request using the 16 bits of the IP address. Based on the memory response *d* and the first tuple (token, IPaddr, s) in fifo2, either the *Completion* (*Action 2*) or *Circulate* (*Action 3*) stage executes. If the lookup is done the *Completion* stage forwards the tuple containing the memory response *d* and the token to the *Completion Buffer*, else the *Circulate* stage places the tuple (token, IPaddr, s+1) into *fifo2* and launches another memory request.

In this example, since IP addresses need varying numbers of memory references, each IP address goes around the pipe as many times as the number of needed memory references. This means that the requests finish out of sequence. Thus, tokens from the *Completion Buffer* are used to make sure that the results arrive in the right order.

After the parts of an action corresponding to combinational logic have executed, all the other parts of an action will occur in parallel in order to achieve maximum parallelism. For example, in *Action 1 (Input)*, after IP address and token are fetched (combinational parts), the other two parts of the actions which place a tuple into *fifo2* and enqueue a memory request will occur in parallel. Similarly, in the (*Action 3 (Circulate)*), the portion of the action that places a tuple in *fifo2*, enqueues a memory request, and acknowledges the reading of result will be executed in parallel after the part that dequeues the tuple from *fifo2* has executed. This kind of parallelism is quite common in hardware design.

However, the use of concurrent actions (multiple actions executing in the same time slot) in the above example reflects the additional amount of parallelism that can be exploited through synthesis from the action-oriented specification style.

The example illustrates that using action-oriented specifications relieves designers from worrying about global coordination, thus allowing them to focus on the much simpler task of local correctness. Larger examples of similar kind may be found in [16], [17].

Confluent Set of Actions

A CAOS-based design can be composed of conflict-free actions. Two actions are said to be conflict-free if none of the actions updates the state elements accessed (including the state elements accessed in the guards) by the other action [4]. A set of actions where all the pairs of actions are conflict-free is said to be *confluent* [18]. The important thing to note about a confluent set of actions is that all the possible orders of the execution of such actions will result in the same final state of the design.

Synthesis

Hardware synthesis using the concurrent actions can be achieved by implementing each g and b as a combinational logic. Also, a control circuit will be needed that picks up a maximum number of actions (among those having their guards true) to be executed concurrently in each time slot (clock cycle).

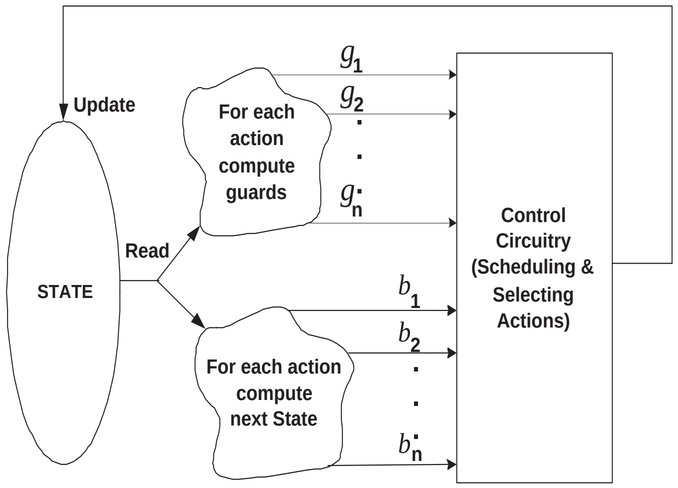
Update
For each action compute guards
g_1
g_2
g_n
Read
STATE
For each action compute next State
b_1
b_2
b_n
Control Circuitry (Scheduling & Selecting Actions)

Fig. 2. Synthesis from Concurrent Actions [15].

Figure 2 shows the schema for a translation from the actions into the hardware. The circuit shown is generated in the concurrent action-based synthesis flow. The guards (*g's*) and update functions (*b's*) are computed for each action using a combinational circuit. The scheduler is designed to select a maximal subset of applicable actions under the constraint that the outcome of a scheduling step can be explained as atomic firing of actions in some order. In each time slot, the *Control Circuit* selects as many actions as possible to execute and updates the current state with the resulting values of those actions. Thus, there exists an almost direct translation from the action-oriented specification of the design to its hardware. Note that the synthesis is done without any constraints on the resources of the design.

Peak Power Constraint

As mentioned earlier, during the execution of a CAOS-based design, multiple non-conflicting actions can be selected for execution from all the actions enabled (guards evaluated to *True*) in a particular time slot. Let A be such a set of non-conflicting actions. Each action $a_i \in A$ is composed of a set of operations and the power expended during each operation of a_i contributes to power p_i needed to execute a_i.

Total power consumed during a particular time slot can be estimated as the summation of the power consumed by various actions executing in that time slot. Thus, executing large number of actions in a time slot may increase the peak power of the design beyond the acceptable limits, which is undesirable. A peak power constraint P on a design can be specified by putting a limit on the total power that can be consumed in any time slot during the execution of the design. The scheduling of various actions of the design should then be done such that the peak power constraint of the design is not violated.

4 Terminology Concerning Approximation Algorithms

A **heuristic** for a combinatorial optimization problem is a polynomial time algorithm that produces a feasible, but not necessarily optimal, solution to all instances of the problem.

For any $\rho \geq 1$, a **ρ-approximation algorithm** for a combinatorial optimization problem is a heuristic that produces a solution which is within a factor ρ of the optimal solution value. To further clarify this notion, suppose the optimal solution value for an instance I of an optimization problem is denoted by $\mathrm{OPT}(I)$. For each instance I of a *minimization* problem, a ρ-approximation algorithm produces a solution whose value is *at most* $\rho\,\mathrm{OPT}(I)$. For each instance I of a *maximization* problem, a ρ-approximation algorithm produces a solution whose value is *at least* $\mathrm{OPT}(I)/\rho$. A ρ-approximation algorithm is also referred to as an algorithm that provides a **performance guarantee** of ρ.

For any *fixed* $\epsilon > 0$, a **polynomial time approximation scheme** (PTAS) for a combinatorial optimization problem provides a performance guarantee of

$1+\epsilon$. Thus, by an appropriate choice of ϵ, a PTAS can produce solutions that are arbitrarily close to the optimal value. As can be expected, the running time of a PTAS increases as the value of ϵ is decreased.

For some combinatorial problems, it is possible to design algorithms whose running times are polynomial in the size of the problem instance and the *maximum* value that occurs in the instance. Such an algorithm is called a **pseudo polynomial time algorithm**. For example, consider the SUBSET SUM problem: Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of integers and an additional integer B, is there a subset S' of S such that the sum of all the integers in S' is equal to B? We may assume without loss of generality that B is the largest integer among the input values. (Integers in S which are larger than B cannot be part of S'.) The SUBSET SUM problem can be solved in $O(nB)$ time using dynamic programming [19]. This is a pseudo polynomial time algorithm for the SUBSET SUM problem. On the other hand, for problems such as MINIMUM VERTEX COVER and MAXIMUM INDEPENDENT SET, there is no pseudo polynomial algorithm, unless **P** = **NP** [19].

5 Scheduling Problems without a Peak Power Constraint

This section considers two scheduling problems related to CAOS-based synthesis in which there is no constraint on the amount of power that can be used in any time slot; that is, no peak power constraint exists. The first problem (considered in Section 5.1) is to construct a largest subset of non-conflicting actions from the set of actions enabled in a time slot. The second problem (considered in Section 5.2) concerns the construction of a minimum length schedule for all the actions.

5.1 Selecting a Largest Non-conflicting Subset of Actions

During the execution of a CAOS-generated design, two actions conflicting with each other cannot be executed in the same time slot. Given a set of actions enabled in a time slot and pairs of conflicting actions, this section considers the problem of finding a largest subset of pairwise non-conflicting actions. The idea is that all the actions in such a subset can be scheduled in the same time slot. We call it the *Maximum Non-conflicting Subset of actions (MNS)* problem and its formal definition is presented below.

Maximum Non-conflicting Subset of actions (MNS)
Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of actions; a collection C of pairs of actions, where $\{a_i, a_j\} \in C$ means that actions a_i and a_j conflict; that is, they cannot be scheduled in the same time slot; an integer $K \leq n$.
Question: Is there subset $A' \subseteq A$ of such that $|A'| \geq K$ and no pair of actions in A' conflict?

In subsequent sections, we present complexity and approximation results for the MNS problem.

Complexity Results for the General Case - The following result points out that the MNS problem is, in general, computationally intractable. In particular, the result points out that the MNS problem corresponds to the well studied Maximum Independent Set problem for undirected graphs.

Proposition 1. *The* MNS *problem is* ***NP****-complete.*

Proof: The MNS problem is in **NP** since one can guess a subset A' of A and verify in polynomial time that $|A'| \geq K$ and that no pair of actions in A' conflict.

To show that MNS is **NP**-hard, we use reduction from the Maximum Independent Set (MIS) problem which is known to be **NP**-complete [19]. An instance of the MIS problem consists of an undirected graph $G(V, E)$ and an integer $J \leq |V|$. The question is whether G has an independent set V' of size $\geq J$ (i.e., a subset V' of V such that $|V'| \geq J$ and there is no edge between any pair of nodes in V').

The reduction is straightforward. Given an instance I of the MIS problem, we construct an instance I' of the MNS problem as follows. The set $A = \{a_1, a_2, \ldots, a_n\}$ of actions is in one-to-one correspondence with the node set V, where $n = |V|$. For each edge $\{v_i, v_j\}$ of G, we construct the pair $\{a_i, a_j\}$ of conflicting actions. Finally, we set $K = J$. Obviously, the construction can be carried out in polynomial time. From the construction, it is easy to see that each independent set of G corresponds to a non-conflicting set of actions and vice versa. Therefore, G has an independent set of size J if and only if there is a subset A' of size $K = J$ such that the actions in A' are pairwise non-conflicting. ■

The above reduction shows that there is a direct correspondence between the MNS and MIS problems. Thus, for any $\rho \geq 1$, a ρ-approximation algorithm for the MNS problem can also be used as a ρ-approximation algorithm for the MIS problem. It is known that for any $\epsilon > 0$, there is no $O(n^{1-\epsilon})$-approximation algorithm for the MIS problem unless the complexity classes[1] **NP** and **ZPP** coincide [21]. Thus, we have the following observation.

Observation 1 *For any $\epsilon > 0$, there is no $O(n^{1-\epsilon})$-approximation algorithm for the* MNS *problem, unless the complexity classes* ***NP*** *and* ***ZPP*** *coincide.* ■

Approximation Algorithms for a Special Case of MNS- As mentioned above, a polynomial time approximation algorithm with a good performance guarantee is unlikely to exist for general instances of the MNS problem. However, for special cases of the problem, one can devise heuristics with good performance guarantees by exploiting the close relationship between the MNS and MIS problems. We now present an illustrative example.

Consider instances of the MNS problem in which every action conflicts with at most Δ other actions, for some constant Δ. A simple heuristic which provides a performance guarantee of $\Delta + 1$ for this special case of the MNS problem is

[1] For definitions of complexity classes, see [20].

shown in Figure 3. Step 1 of this heuristic transforms the special case of the MNS problem into a special case of the MIS problem where the underlying graph has a maximum node degree of Δ. It is a simple matter to verify that each independent set of the resulting graph corresponds to a subset of pairwise non-conflicting actions. Step 3 describes a greedy algorithm which provides a performance guarantee of $\Delta + 1$ for the restricted version of the MIS problem [22,23]. Because of the direct correspondence between independent sets of G and non-conflicting subsets of actions, the heuristic also serves as a $(\Delta + 1)$-approximation algorithm for the special case of the MNS problem. This result is stated formally below.

Observation 2 *For instances of the* MNS *problem in which each action conflicts with at most Δ other actions for some constant Δ, the approximation algorithm in Figure 3 provides a performance guarantee of $\Delta + 1$.* ■

For the class of graphs whose maximum node degree is bounded by a constant Δ, a heuristic which provide a better (asymptotic) performance guarantee of $O(\Delta \log\log \Delta / \log \Delta)$ is known for the MIS problem [22,23]. However, that heuristic is harder to implement compared to the one shown in Figure 3.

Efficient approximation algorithms with good performance guarantees are known for the MIS problem for other special classes of graphs such as planar graphs, near-planar graphs and unit disk graphs [24,25,26]. When the MNS instances correspond to instances of the MIS problem for such graphs, one can use the approximation algorithms for the latter problem to obtain good approximations for the MNS problem.

```
1. From the given instance of the MNS problem, construct a graph $G(V, E)$ as follows:
   The node set $V$ is in one-to-one correspondence with the set of actions $A$. For each
   conflicting pair $\{a_i, a_j\}$ of actions, add the edge $\{v_i, v_j\}$ to $E$.
2. Initialize $A'$ to $\emptyset$. (Note: At the end, $A'$ will contain a set of pairwise non-conflicting
   actions.)
3. while $V \neq \emptyset$ do
     (a) Find a node $v$ of minimum degree in $G$.
     (b) Add the action corresponding to node $v$ to $A'$.
     (c) Delete from $G$ the node $v$ and all nodes which are adjacent to $v$. Also delete
         the edges incident on those nodes.
     (d) Recompute the degrees of the remaining nodes.
4. Output $A'$.
```

Fig. 3. Steps of the Heuristic for the Special Case of the MNS Problem

Application to Bluespec

Bluespec Compiler (BSC) allows a maximal set of non-conflicting actions to execute in each time slot which aids in reducing the latency of the design. Reference [27] presents an algorithm used in *BSC* which selects actions based on their priorities. The algorithm first orders the set A of all the actions in terms of their priorities. Let $A^{'} = \{a_1, a_2, ..., a_n\ \}$ be the ordered set of actions such that a_i is more urgent (higher priority) than a_j iff $i < j$. The algorithm then computes a set S which contains all the actions that can be executed in a time slot as follows.

1. Let S be the empty set.
2. If $A^{'}$ is empty, stop and return S.
3. Let a_k be the highest priority action in $A^{'}$.
4. If a_k is enabled and no action a_j, $1 \leq j < k$ conflicting with a_k is enabled, then $e = 1$, else $e = 0$.
5. If $(e == 1)$ then add a_k to S.
6. Remove a_k from $A^{'}$.
7. Go to step 2.

Thus, the above algorithm selects the set of non-conflicting actions based on their priorities. Between two conflicting actions, an action with higher priority is always chosen to execute whenever both the actions are enabled in the same time slot. If multiple actions conflict with each other then the action with the highest priority is preferred for execution over all the other conflicting actions. Such a high priority action is selected even if it conflicts with large number of other actions.

Note that step 3 of the heuristic shown in Figure 3 selects the minimum degree node from the remaining set of nodes. Thus, during the selection of independent nodes, the heuristic gives low preference to a node connected to large number of other nodes. With regard to the MNS problem, this implies that if an action conflicts with large number of other actions then the heuristic gives a low preference to that action while selecting the set of non-conflicting actions. It gives priority to actions conflicting with minimum number of other actions, and thus attempts to select as many actions as possible. This may further reduce the latency of the design. On the other hand, in *BSC*, if the designer does not specify a priority between two conflicting actions, the compiler assigns an arbitrary priority to both of them. Thus, the algorithm used in *BSC* does not attempt to select a large set of non-conflicting actions, unlike the heuristic shown in Figure 3.

If we change step 3 of the heuristic to select nodes based on their priorities, as done in *BSC*, then the result of the heuristic in Figure 3 will be same as the algorithm used in *BSC*; that is, if enabled an action with the highest priority will always be chosen to execute.

5.2 Constructing Minimum Length Schedules

The previous section considered the problem of choosing a non-conflicting set of maximum cardinality. Here, we consider the problem of partitioning a given set of actions into a minimum number of subsets so that no subset contains a pair of conflicting actions. When this condition is met, the actions in each subset can be scheduled in the same time slot. Thus, the partitioning problem models the problem of constructing a minimum length schedule for the set of actions. A formal definition of the scheduling problem is given below.

Minimum Length Schedule Construction (MLS)
Instance: A set $A = \{a_1, a_2, \dots, a_n\}$ of actions; a collection C of pairs of actions, where $\{a_i, a_j\} \in C$ means that actions a_i and a_j conflict, that is, they cannot be scheduled in the same time slot; an integer $\ell \leq n$.
Question: Is there a partition of A into r subsets A_1, A_2, …, A_r, for some $r \leq \ell$, such that for each i, $1 \leq i \leq r$, the actions in A_i are pairwise non-conflicting?

In what follows, we present complexity and approximation results for the MLS problem.

Complexity Results for the General Case The following result points out that the MLS problem is, in general, computationally intractable. In particular, our result points out that the MLS problem corresponds to the *minimum coloring* problem for undirected graphs.

Proposition 2. *The* MLS *problem is* ***NP****-complete.*

Proof: It is easy to see that the MLS problem is in **NP** since one can guess a partition of A into at most $r \leq \ell$ subsets and verify in polynomial time that no pair of actions in any subset conflict.

To show that MLS is **NP**-hard, we use reduction from the MINIMUM K-COLORING (K-COLORING) problem which is known to be **NP**-complete [19]. An instance of the K-COLORING problem consists of an undirected graph $G(V, E)$ and an integer $K \leq |V|$. The question is whether the nodes of G can be colored using at most K colors so that for each edge $\{v_i, v_j\} \in E$, the colors assigned to v_i and v_j are different. We note that in any valid coloring of G, each color class (i.e., the set of nodes assigned the same color) forms an independent set.

The reduction is straightforward. Given an instance I of the K-COLORING problem, we construct an instance I' of the MLS problem as follows. The set $A = \{a_1, a_2, \dots, a_n\}$ of actions is in one-to-one correspondence with the node set V, where $n = |V|$. For each edge $\{v_i, v_j\}$ of G, we construct the pair $\{a_i, a_j\}$ of conflicting actions. Finally, we set $\ell = K$. Obviously, the construction can be carried out in polynomial time. From the construction, it is easy to see that each valid coloring set of G corresponds to a partition of the set A into non-conflicting subsets and vice versa. (Each color class corresponds to a subset of actions that can be scheduled in the same time slot and vice versa.) Therefore, G has a valid coloring with $r \leq K$ colors if and only if there is a partition of A into r subsets such that the actions in each subset are pairwise non-conflicting. ■

Let us denote the problem of coloring a graph with a minimum number of colors by MinColor. The above reduction shows that there is a direct correspondence between the MLS and MinColor problems. Thus, for any $\rho \geq 1$, a ρ-approximation algorithm for the MLS problem can also be used as a ρ-approximation algorithm for the MinColor problem. It is known that for any $\epsilon > 0$, there is no $O(n^{1-\epsilon})$-approximation algorithm for the MinColor problem unless the complexity classes **NP** and **ZPP** coincide [28]. Thus, we have the following observation.

Observation 3 *For any $\epsilon > 0$, there is no $O(n^{1-\epsilon})$-approximation algorithm for the* MLS *problem, unless the complexity classes **NP** and **ZPP** coincide.* ■

Observations Concerning Special Cases of MLS As mentioned above, the MLS problem is, in general, hard to approximate. To identify some simpler versions of the MLS problem, we exploit the relationship between the MLS and MinColor problems. In particular, the MLS problem can be reduced to the MinColor problem by constructing a graph G in which each node corresponds to an action and each edge corresponds to a pair of conflicting actions. It is easy to see that any valid coloring of G with r colors corresponds to a schedule of length r. This reduction of MLS to the MinColor problem is useful for several reasons. First, it points out that in practice, one can use known heuristics for graph coloring in constructing schedules of near-minimum length. Although the coloring problem is hard to approximate in the worst-case [28], heuristics that work well in practice are known (see for example [29,30]). In addition, the reduction to the MinColor problem also points out that the following two special cases of the MLS problem can be solved efficiently.

(a) Consider instances of the MLS problem in which the upper bound on the length of the schedule is two. This special case of the MLS problem corresponds to the problem of determining whether a graph is 2-colorable. Efficient algorithms are known for this problem [31].
(b) Consider instances of the MLS problem in which each action conflicts with at most Δ other actions, for some integer Δ. For such instances, a schedule of length at most $\Delta + 1$ can be constructed in polynomial time. To see this, we note that the graph corresponding to such instances of the MLS problem has a maximum node degree of Δ. By a well known result in graph theory, called Brooks's Theorem, any graph with a maximum node degree of Δ can be colored efficiently using at most $\Delta + 1$ colors [32]. Such a coloring gives rise to a schedule of length at most $\Delta + 1$.

Application to Bluespec

In *Bluespec*, multiple conflicting actions can get enabled in each time slot. But only a non-conflicting subset of such actions can be allowed to execute in a given time slot. Thus, the heuristics for MLS problem can be used for partitioning such a set of actions into multiple subsets of non-conflicting actions. Each of these subsets can then be scheduled in a separate time slot.

Note that executing actions of one subset may disable (guards evaluate to *False*) the actions of other subsets. This may happen if the state elements updated by the actions belonging to a subset (executing in the present time slot) are accessed by the guards of the actions belonging to the other subsets (which are scheduled to execute in future time slots). However, if the guards of these actions of other subsets do not access the state elements updated by the subset of actions executing in the present time slot, then such partitioning of actions can be used in *Bluespec* to schedule each non-conflicting subset in a different time slot.

6 Scheduling Problems Involving a Power Constraint

In this section, we study scheduling problems taking into consideration the amount of power consumed by various actions. In particular, we assume that a constraint on peak power, that is, the amount of power that can be consumed in any time slot, is specified. The goal is to construct schedules satisfying this constraint. We first consider (Section 6.1) the problem of finding a largest subset of actions from a set of non-conflicting actions that can be scheduled in a given time slot under the peak power constraint. We show that this problem can be solved efficiently. Next, a generalization of this problem, where there is a utility value associated with each action and the goal is to choose a subset of actions that maximize the total utility while satisfying the peak power constraint is shown to be **NP**-complete (Section 6.2). Finally (Section 6.3), we consider several versions of the problem of minimizing the schedule length subject to the peak power constraint. We present both complexity and approximation results.

6.1 Packing Actions in a Time Slot under Peak Power Constraint

Due to the peak power constraint, it might not be possible to execute all the actions belonging to a set of non-conflicting actions that are enabled in a particular time slot. This section considers the problem of packing a maximum number of actions into a time slot without violating the constraint on peak power. We present a simple algorithm with a running time of $O(n \log n)$ for the problem. We begin with a formal statement of the problem.

Maximum Number of Actions in a Time Slot Subject to Peak Power Constraint (MNA-PP)

Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of non-conflicting actions; for each action a_i, the power p_i needed to execute that action; a positive number P representing the peak power, that is, the maximum power that can be used in any time slot.

Requirement: Find a subset $A' \subseteq A$ such that the total power needed to execute all the actions in A' is at most P and $|A'|$ is a maximum over all subsets of A that satisfy the constraint on total power.

```
1. Sort the actions in $A$ into non-decreasing order by the amount of power needed
   for each action. Without loss of generality, let $\langle a_1, a_2, \ldots, a_n \rangle$ denote the resulting
   sorted order.
2. **Comment:** Keep adding actions in the above order as long as the total power is
   constraint is satisfied.
   (a) Initialization: Let $A' = \emptyset$, $i = 1$ and $R = P$. (**Note:** $R$ denotes the remaining
       amount of power.)
   (b) **while** $(i \leq n)$ **and** $p_i \leq R$ **do**
         (i) Add $a_i$ to $A'$.
         (ii) $R = R - p_i$.
         (iii) $i = i + 1$.
3. Output $A'$.
```

Fig. 4. Steps of the Algorithm for the Mna-PP Problem

An efficient algorithm for Mna-PP is shown in Figure 4. The following lemma shows the correctness of the algorithm.

Lemma 1. *The algorithm in Figure 4 computes an optimal solution to the* Mna-PP *problem.*

Proof: Let $\langle a_1, a_2, \ldots, a_n \rangle$ denote the list of actions in nondecreasing order of power values. We assume that P is sufficient to execute a_1; otherwise, none of the actions can be executed in any time slot. Let A' be the set of actions produced by the algorithm in Figure 4, and let $|A'| = k$. Thus, $A' = \{a_1, a_2, \ldots, a_k\}$, and the remaining power is not enough to add action a_{k+1} to A'.

Suppose A' is not an optimal solution. Thus, there is another solution A^* such that $|A^*| = r \geq k+1$. Let $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_r} \rangle$ denote the actions in A^* arranged in the sorted order chosen by the algorithm. It is easy to see that, for $1 \leq j \leq k$, the power needed to execute action a_j is no more than that of a_{i_j}. Thus, the remaining power after adding actions a_{i_1}, a_{i_2}, …, a_{i_k} is no more than that after adding the actions a_1, a_2, …, a_k. Further, the power needed for action $a_{i_{k+1}}$ is at least that needed for action a_{k+1}. Thus, the optimal solution A^* cannot accommodate the action $a_{i_{k+1}}$ without violating the peak power constraint. This is a contradiction and the lemma follows. ■

Theorem 4. *The* Mna-PP *problem can be solved in $O(n \log n)$ time, where n is the number of actions.*

Proof: The correctness of the algorithm in Figure 4 follows from Lemma 1. To estimate the running time of the algorithm, we note that the sorting step uses $O(n \log n)$ time and the other steps use $O(n)$ time. So, the overall running time is $O(n \log n)$. ■

Application to Bluespec

As already mentioned, in *Bluespec*, maximal subset of non-conflicting actions are executed in each time slot. If large number of actions are executed in the same time slot, then it can lead to the violation of peak power constraint of the design. In that case, only a subset of such actions should be allowed to execute in a given time slot to meet the peak power constraint.

Thus, in *Bluespec*, the problem of selecting a largest subset of actions such that the peak power constraint of the design is satisfied can be directly mapped to the MNA-PP problem described above. Algorithm in Figure 4 can then be used to solve the peak power problem optimally.

6.2 Maximizing Utility Subject to a Power Constraint

We now consider a generalization of the MNA-PP problem considered in the previous subsection. Suppose for each action $a_i \in A$, there is a **utility** value u_i, in addition to the power value p_i. We can define the utility of a subset A' of actions to be the sum of the utilities of the actions in A'. It is of interest to consider the problem of selecting a subset of maximum utility, subject to the constraint on peak power. A formal statement of the decision version of this problem is as follows.

Maximizing Utility Subject to Peak Power Constraint (MU-PP)
Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of non-conflicting actions; for each action a_i, the power p_i consumed during the execution of that action and the utility u_i of that action; a positive number P representing the peak power, that is, the maximum power that can be used in any time slot; a positive number Γ representing the required utility.
Question: Is there a subset $A' \subseteq A$ such that the total power needed to execute all the actions in A' is at most P and the utility of A' is at least Γ?

Note that the MNA-PP problem is a special case of the MU-PP problem, with the utility value each action being 1. However, while the MNA-PP problem is efficiently solvable, the MU-PP problem is **NP**-complete, as shown below.

Proposition 3. *The* MU-PP *problem is* ***NP****-complete.*

Proof: The MU-PP problem is in **NP** since one can guess a subset A' of A and verify in polynomial time that the utility of $|A'|$ is at least Γ and that the total power needed to execute the actions in A' is at most P.

To show that MU-PP is **NP**-hard, we use reduction from the KNAPSACK problem which is known to be **NP**-complete [19]. An instance of the KNAPSACK problem consists of a set S of items, an integer weight w_i and an integer profit value u_i for each item $s_i \in S$, and two positive integers B and Π. The question is whether there is a subset S' of S such that the total weight of all the items in S' is at most B and the profit of all the items in S' is at least Γ.

The reduction is straightforward. Given an instance I of the KNAPSACK problem, we construct an instance I' of the MU-PP problem as follows. The set $A = \{a_1, a_2, \ldots, a_n\}$ of actions is in one-to-one correspondence with the set S, where $n = |S|$. For each action a_i, the power value p_i and the utility u_i are set respectively to the weight w_i and the profit u_i of the corresponding item $s_i \in S$, $1 \leq i \leq n$. Finally, we set the power bound $P = B$ and the utility bound $\Gamma = \Pi$. Obviously, the construction can be carried out in polynomial time. From the construction, it is easy to see that any solution to the KNAPSACK problem with a profit of α corresponds to a set of actions whose utility is α and vice versa. Therefore, there is a solution to the KNAPSACK instance I if and only if there is a solution to the MU-PP instance I'. ■

Approximation Algorithms for MU-PP

It is easy to see that the optimization version of the MU-PP problem can be transformed into the KNAPSACK problem. Each action a_i is represented by an item s_i; the utility u_i and the power value p_i are respectively the weight and the profit values for the item. The peak power value P represents the knapsack capacity. Because of this transformation, known algorithms for the KNAPSACK problem can be directly used to solve the MU-PP problem. For example, a pseudo polynomial algorithm for the MU-PP problem follows from the corresponding algorithm for the KNAPSACK problem [19]. Further, any approximation algorithm for the KNAPSACK problem can be used as an approximation algorithm with the same performance guarantee for the optimization version of MU-PP. For example, a known 2-approximation algorithm for KNAPSACK [19] implies a similar approximation for the optimization version of the MU-PP problem. Also, when the weights and profits are integers, there is a polynomial time approximation scheme (PTAS) for the KNAPSACK problem [19]. Therefore, when the power and utility values for each action are integers, one can obtain a PTAS for the optimization version of the MU-PP problem.

Application to Bluespec

The utility value of a given action represents the usefulness of the execution of that action. In *Bluespec*, one such measure of the utility value of an action is the number of actions having a data dependency on it [3]; that is, number of actions accessing the state elements updated by a particular action. Based on such a measure, an action having a large number of other actions dependent on it can be assigned a high utility value.

Thus, when actions are assigned different utility values in a *Bluespec* design, the solutions to the MU-PP problem described above can be used to solve the following *Bluespec* peak power problem - selecting the largest subset of non-conflicting actions such that peak power constraint of the design is satisfied and the utility of the selected subset of actions is maximized.

6.3 Combination of Makespan and Power Constraint

Peak power constraint of a design may not allow all the actions in a set of non-conflicting actions to execute in a single time slot. In this subsection, our focus is on scheduling such a set of actions over a small number of time slots while keeping the peak power value as small as possible. The number of slots used by a schedule is called the **makespan**. Two optimization problems can be studied in this context. First, the problem of minimizing makespan subject to a peak power constraint can be formulated as follows.

Minimizing Makespan Subject to Peak Power Constraint (MM-PP)
Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of non-conflicting actions; for each action a_i, the power p_i needed to execute that action; a positive number P representing the peak power, that is, the maximum power that can be used in any time slot.
Requirement: Find a schedule of minimum length for the actions in A such that the total power needed to execute the actions in each time slot is at most P.

The dual problem of minimizing peak power subject to a constraint on the schedule length can be formulated as follows.
Minimizing Peak Power Subject to a Makespan Constraint (MPP-M)
Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of non-conflicting actions; for each action a_i, the power p_i needed to execute that action; a positive integer L representing the makespan.
Requirement: Find a schedule of length at most L for the actions in A such that the maximum total power used in any time slot is a minimum over all schedules of length at most L.

We note that the decision versions of the two problems MM-PP and MPP-M are identical. A formal statement of the decision version is as follows.

Minimizing Makespan and Peak Power – Decision Version (MPP-DECISION)
Instance: A set $A = \{a_1, a_2, \ldots, a_n\}$ of non-conflicting actions; for each action a_i, the power p_i needed to execute that action; a positive number P representing the peak power that can be used in any time slot; a positive integer L representing the makespan.
Question: Is there a schedule of length at most L for the actions in A such that the total power used in any time slot is at most P?

We now show that MPP-DECISION is **NP**-complete, even when the makespan is fixed at 2.

Proposition 4. *Problem* MPP-DECISION *is* ***NP****-complete.*

Proof: The MPP-DECISION problem is in **NP** since one can guess a schedule for the actions in A and verify in polynomial time that the schedule length and the peak power constraints are satisfied.

To show that MPP-Decision is **NP**-hard, we use reduction from the Partition problem which is known to be **NP**-complete [19]. An instance of the Partition problem consists of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n integers. The question is whether S can be partitioned into two subsets S_1 and S_2 such that the sum of the elements in S_1 is equal to the sum of the elements in S_2. (We may assume without loss of generality that $\sum_{i=1}^{n} s_i$ is even; otherwise, there is no solution to the Partition problem.)

Given an instance I of the Partition problem, we construct an instance I' of the MPP-Decision problem as follows. The set $A = \{a_1, a_2, \ldots, a_n\}$ of actions is in one-to-one correspondence with the set S of items. For each action a_i, the power value p_i is set equal to s_i, $1 \leq i \leq n$. Finally, we set the bound P on peak power to $(\sum_{i=1}^{n} s_i)/2$ and the schedule length L to 2. This completes the construction. Obviously, the construction can be carried out in polynomial time. We now show that the Partition instance I has a solution if and only if the MPP-Decision instance I' has a solution.

Part 1: Suppose the Partition instance I has a solution given by subsets S_1 and S_2. Consider the schedule of length 2 which includes all the actions corresponding to the numbers in S_1 in the first time slot and the remaining actions in the second time slot. Since the sum of the integers in S_1 and S_2 are both equal to $P = (\sum_{i=1}^{n} s_i)/2$, the total power used in each time slot is equal to P. Thus, we have a schedule of length two satisfying the peak power constraint; in other words, there is a solution for the MPP-Decision instance I'.

Part 2: Suppose the MPP-Decision instance I' has a solution, that is, a schedule of length at most 2 such that in each time slot, the power used is at most P. We first note that the schedule cannot be of length 1; if so, the peak power used would be equal to $\sum_{i=1}^{n} s_i$, which is greater than P. Thus, the schedule is of length 2. For $i = 1, 2$, let S_i be the set of integers corresponding to the actions scheduled in time slot i. We claim that the sum of the integers in S_1 is equal to that of S_2. To see this, note that the sum of the integers in S_1 cannot exceed $P = (\sum_{i=1}^{n} s_i)/2$, since that will violate the peak power constraint in the first time step. Likewise, if the sum of the integers in S_1 is less than P, then the sum of the integers in S_2 would exceed P; that is, the peak power constraint would be violated in the second time step. Thus, sum of the integers in S_1 must be equal to that of S_2. In other words, we have a solution to the Partition instance I. This completes the proof of Proposition 4. ■

The above proof shows that determining whether there is a schedule of length 2 satisfying the peak power constraint is **NP**-complete. In contrast, we note that determining whether there is a schedule of length 1 satisfying the peak power constraint is trivial; we need only check whether the total power for all the actions in A is at most the peak power value.

While the above proof shows that the MPP-Decision problem is **NP**-complete even for fixed values of schedule length, it leaves open the possibility of a pseudo-polynomial algorithm for the problem. We now present a different reduction to show that MPP-Decision is **strongly NP**-complete, when the schedule length is part of the problem instance. Thus, in general, there is no pseudo-polynomial algorithm for the MPP-Decision problem, unless **P** = **NP**.

Proposition 5. *Problem* MPP-Decision *is strongly* ***NP****-complete when the schedule length is part of the problem instance.*

Proof: We use a reduction from the 3-Partition problem, which is known to be strongly **NP**-complete [19]. An instance I of this problem consists of a positive integer B, a positive integer m, a set $S = \{s_1, s_2, \ldots, s_{3m}\}$ of $3m$ positive integers such that $B/4 < s_i < B/2$, $1 \leq i \leq m$, and $\sum_{i=1}^{3m} s_i = mB$. The question is whether the set S can be partitioned into m subsets such that the sum of the values in each subset is exactly B. (The constraint on the value of each $s_i \in S$ ensures that when there is such a partition, each subset in the partition has exactly three elements.)

Given an instance I of 3-Partition, an instance I' of MPP-Decision can be constructed as follows. The set $A = \{a_1, a_2, \ldots, a_{3m}\}$ of actions is in one-to-one correspondence with the set S of items. For each action a_i, the power value p_i is set equal to s_i, $1 \leq i \leq n$. We set the bound P on peak power to B and the schedule length L to m. This completes the construction. Obviously, the construction can be carried out in polynomial time. The proof that the 3-Partition instance I has a solution if and only if the MPP-Decision instance I' has a solution is similar to that presented in the proof of Proposition 4. ■

6.4 Approximation Algorithms for MM-PP

Recall that in the MM-PP problem, we are required to minimize the schedule length subject to a constraint on the peak power value. Since this problem is **NP**-hard, it is of interest to investigate approximation algorithms with provable performance guarantees. One can obtain such approximation algorithms by reducing the problem to the well known Bin Packing problem and using known approximation algorithms for the latter problem.

In the Bin Packing problem, we are given a collection C of n items, where item i has a size $x_i \in (0, 1]$. The goal is to pack these items into a *minimum* number of bins, each of unit capacity. This minimization problem is known to be **NP**-hard [19]. However, several approximation algorithms with good performance guarantees are known for this problem [33]. For example, the problem admits a PTAS. However, the algorithm is somewhat complicated and its running time is exponential in $1/\epsilon$, where ϵ is the fixed accuracy parameter. A much simpler algorithm, called **First Fit Decreasing** (FFD), provides a performance guarantee of 11/9 [33]. The idea is first to sort the items in non-increasing order of their sizes and then assign each item to the first bin in which it will fit. The steps of this approximation algorithm are shown in Figure 5. A straightforward implementation of the algorithm Figure 5 runs in $O(n^2)$ time. However, a more sophisticated implementation reduces the running time to $O(n \log n)$ [19].

We note that the MM-PP problem can be reduced to the Bin Packing problem as follows. Let P denote the given bound on peak power. For each

```
1. Sort the items into non-increasing order of their sizes. Without loss of generality,
   let $\langle c_1, c_2, \ldots, c_n \rangle$ denote the resulting sorted order.
2. **Comment:** Assign each item to the first bin in which it will fit.
   (a) Initialization: Let $j = 1$. (The variable $j$ denotes the number of bins.) Let $B_1$
       denote the initial bin.
   (b) **for** $i = 1$ **to** $n$ **do**
       (i) Find the first bin, say bin $B_k$, among $B_1$ through $B_j$ in which item $i$ will
           fit.
       (ii) If there is no such bin, increment $j$ by 1 and create a new bin $B_j$. Let
            $k = j$.
       (iii) Add item $i$ to bin $B_k$.
3. Output $j$ (the number of bins) and the packing generated above.
```

Fig. 5. Steps of the First Fit Decreasing Algorithm for the Bin Packing Problem

action a_i with power value p_i, we create an item with size $= p_i/P$. Since $p_i \leq P$, the size of each item is at most 1. Now, each time slot can be thought of as a bin (of unit capacity) so that any packing of the items into q bins corresponds to a valid schedule of length q and vice versa. Thus, any approximation algorithm for the Bin Packing problem can be used as an approximation algorithm for the MM-PP problem with the same performance guarantee. A formal statement of this result is as follows.

Observation 5 *Any ρ-approximation algorithm for the* Bin Packing *problem is also a ρ-approximation algorithm for the* MM-PP *problem.* ■

From the above discussion, we can conclude that there are efficient approximation algorithms with good performance guarantees for the MM-PP problem.

Application to Bluespec

In *Bluespec*, if the execution of a large set of non-conflicting actions in a single time slot leads to the violation of peak power constraint, then such a set of actions can be re-scheduled to execute over multiple time slots. Instead of executing all the actions of the set, some of the actions can be postponed to execute in the future time slots in order to meet the peak power constraint. Such a re-scheduling problem of *Bluespec* is equivalent to the MM-PP problem, and hence the approximation algorithms for the MM-PP problem can be used to re-schedule a set of actions of a *Bluespec* design to meet the peak power constraint.

Note that executing only a subset of actions in the present time slot may disable (guards evaluate to *False*) the remaining actions of the set in future time slots. This may happen if the state elements updated by the subset of actions executing in the present time slot are accessed by the guards of the actions postponed to future time slots. In such cases, using the approximation algorithms of the MM-PP problem for re-scheduling of the actions may result in changing the output of the design (though the output will still be functionally correct) [3]. However, for designs described in terms of confluent set of actions, the output will not be affected. This is because, as mentioned earlier, a confluent set of actions can be executed in any order without changing the final state of the design. Thus, for a design described using confluent set of actions, solutions of the MM-PP problem can be used to re-schedule a set of non-conflicting actions to arrive at a minimum length schedule under the peak power constraints.

6.5 Approximation Algorithms for MPP-M

In the MPP-M problem, we are given a bound on the schedule length and the goal is to minimize the maximum power used in any time slot. Interestingly, when the power needed to execute each instruction is an integer, the MPP-M problem can be transformed into a classical multiprocessor scheduling problem [19] In the classical scheduling problem, we are given a collection $T = \{T_1, T_2, \ldots, T_n\}$ of tasks, where each task T_i has an integer execution time e_i. The tasks are independent; that is, there are no precedence constraints among the tasks. The problem is to schedule the tasks in a non-preemptive fashion on m identical processors so as to minimize the makespan. To see the correspondence between the MPP-M problem and the classical scheduling problem, we think of each action a_i as a task and the power value p_i as the execution time of the corresponding task. Further, we think of the number of time slots ℓ as the number of available processors. Now, for the resulting scheduling problem, it can be seen that any schedule with makespan P on ℓ processors corresponds to a solution to the MPP-M problem using ℓ time slots and a peak power value of P. (The set of tasks scheduled to run on the same processor corresponds to the set of actions to be performed during the same time slot.)

In view of the above relationship between the MPP-M problem and the multiprocessor scheduling problem, any ρ-approximation algorithm for the latter is a ρ-approximation algorithm for the former. In particular, the following is a 4/3-approximation algorithm for the multiprocessor scheduling problem [34].

1. Construct a list of the tasks in non-increasing order of execution times.
2. Whenever a processor becomes available, assign the next job from the list to that processor. (If several processors become available at the same time, ties are broken arbitrarily.)

In terms of the MPP-M problem, the above approximation algorithm corresponds to sorting the actions in non-increasing order of their power requirements and assigning each action to a time slot for which the total power used is the smallest at that time. Clearly, this approximation algorithm can be implemented to run in $O(n \log n)$ time.

A PTAS is also known for the multiprocessor scheduling problem [34]. However, as the running time of the corresponding algorithm is exponential in $1/\epsilon$, where $\epsilon > 0$ is the chosen accuracy parameter, this may not be suitable in practice.

Application to Bluespec

For some *Bluespec* designs, latency is of prime concern. For such designs, actions can be re-scheduled using the approximation algorithms for MPP-M problem in order to minimize the peak power of the design under the given latency constraints.

Here again, the re-scheduling of the actions of a design may result in the disabling of some of the actions, thus changing the output of the design. However, for designs described in terms of confluent set of actions, such re-scheduling is suitable (since it will not change their output) and hence can be used to minimize the peak power.

7 Conclusion

Keeping the peak power of a design under acceptable limits is important for designs generated from CAOS. In this paper, we discussed the complexity and approximability of a variety of problems involving peak power and schedule length encountered in CAOS-based synthesis. Exploiting the relationships between these problems and classical optimization problems such as bin packing and multiprocessor scheduling, we showed how one can develop efficient approximation algorithms with provable performance guarantees. In the future, we are planning to investigate whether the approximation algorithms can be extended to the scheduling problems in which there are precedence constraints among the actions.

References

1. Raghunathan, A., K.Jha, N., Dey, S.: High-Level Power Analysis And Optimization. Kluwer Academic Publishers (1998)
2. Singh, G., Shukla, S.K.: Algorithms for Low Power Hardware Synthesis from CAOS - Concurrent Action Oriented Specifications. Special Issue of International Journal of Embedded Systems on Power/Energy/Thermal topics (IJES'06) (2007)
3. Singh, G., Shukla, S.K.: Low-Power Hardware Synthesis from TRS-based Specifications. International Conference on Formal Methods and Models for Codesign (MEMOCODE'06) (2006)
4. Hoe, J.C., Arvind: Hardware Synthesis from Term Rewriting Systems. Proceeding of VLSI'99 Lisbon, Portugal (1999)
5. Shiue, W.T.: High level synthesis for peak power minimization using ilp. In Proceedings of the IEEE International Conference on ASSAP (2000) 103–112
6. Lakshminarayana, G., Raghunathan, A., Jha, N.K., Dey, S.: A Power Management Methodology for High-Level Synthesis. International Conference on VLSI Design (1998) 24–29

7. J.Monteiro, S.Devadas, P.Ashar, A.Mauskar: Scheduling techniques to enable power management. Proceedings of Design Automatin Conference (1996) 349–352
8. Raghunathan, A., Dey, S., Jha, N., K.Wakabayashi: Power Management techniques for Control-flow intensive designs. Proceedings of Design Automation Conference (1997)
9. Khouri, K.S., Lakshminarayana, G., Jha, N.K.: High-Level Synthesis Of Low Power Control-Flow Intensive Circuits. IEEE Transactions on Computer-Aided Design (TCAD'99) **18** (1999)
10. Raghunathan, V., Ravi, S., Raghunathan, A., Lakshminarayana, G.: Transient Power Management Through High Level Synthesis. In Proceedings of the ICCAD (2001) 545–552
11. Mohanty, S.P., Ranganathan, N.: A framework for energy and transient power reduction during behavioral synthesis. In Proceedings of the International Conference on VLSI Design (2003) 539–545
12. Chang, J.M., Pedram, M.: Power Optimization And Synthesis At Behavioral And System Levels Using Formal Methods. Kluwer Academic Publishers (1999)
13. Shiue, W.T., Chakrabarti, C.: Low-Power Scheduling with Resources Operating at Multiple Voltages. IEEE Transactions On Circuits and Systems - II: Analog and Digital Signal Processing **47** (2000) 536–543
14. Kumar, A., Bayoumi, M.: Multiple Voltage-based Scheduling Methodology for Low-power in the High-level Synthesis. IEEE International Symposium on Circuits and Systems **1** (1999) 371–374
15. Arvind, Nikhil, R., Rosenband, D., Dave, N.: High-level synthesis: An Essential Ingredient for Designing Complex ASICs. Proceedings of the International Conference on Computer Aided Design (ICCAD'04) (2004) 775–782
16. Kurki-Suonio, R.: A Practical Theory of Reactive Systems: Incremental Modeling of Dynamic Behaviors. Springer (1998)
17. Misra, J.: A Discipline of Multi-Programming. Springer (2001)
18. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
19. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco, CA (1979)
20. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley, Reading, MA (1993)
21. Hastad, J.: Clique is Hard to Approximate Within $n^{1-\varepsilon}$. Acta Mathematica **182** (1999) 105–142
22. Berman, P., Fujito, T.: Approximating Independent Sets in Degree 3 Graphs. Proc. 4th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. **955** (1995) 449–460
23. Halldorsson, M.M.: Approximations of Weighted Independent Set and Hereditary Subset Problems. In: Proc. 5th Ann. Int. Conf. on Computing and Combinatorics, Lecture Notes in Comput. Sci. Springer-Verlag (1999) 261–270
24. Baker, B.S.: Approximation Algorithms for NP-complete Problems on Planar Graphs. J. ACM **41** (1994) 153–180
25. Hunt-III, H.B., Marathe, M.V., Radhakrishnan, V., Ravi, S.S., Rosenkrantz, D.J., Stearns, R.E.: NC-Approximation Schemes for NP- and PSPACE-hard Problems for Geometric Graphs. Journal of Algorithms **26** (1998) 238–274
26. Hunt-III, H.B., Marathe, M.V., Radhakrishnan, V., Ravi, S.S., Rosenkrantz, D.J., Stearns, R.E.: Parallel Approximation Schemes for a Class of Planar and Near Planar Combinatorial Problems. Information and Computation **173** (2002) 40–63

27. Liz, M., et al.: Efficient Generation of Schedulers for Guarded Atomic Actions. Technical Memo, Bluespec Inc. (2005) http://www.bluespec.com/.
28. Feige, U., Kilian, J.: Zero Knowledge and the Chromatic Number. J. Computer and System Sciences **57** (1998) 187–199
29. Hertz, A., de Werra, D.: Using Tabu Search Techniques for Graph Coloring. Computing **39** (1987) 345–351
30. Campers, G., Henkes, O., Leclerq, P.: Graph Coloring Heuristics: A Survey, Some New Propositions and Computational Experiences on Random and Leighton's Graphs. Proc. Operational Research '87, Buenos Aires (1987) 917–932
31. Cormen, T., Leiserson, C.E., Rivest, R., Stein, C.: Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, Cambridge, MA (2001)
32. West, D.B.: Introduction to Graph Theory, Second Edition. Prentice Hall, Inc., Englewood Cliffs, NJ (2001)
33. Coffman-Jr., E.G., Garey, M.R., Johnson, D.S.: Approximation Algorithms for Bin-Packing - A Survey. In: Approximation Algorithms for NP-hard Problems. Edited by D. S. Hochbaum, PWS Publishing Company, Boston, MA (1997) 46–93
34. Hall, L.A.: Approximation Algorithms for Scheduling. In: Approximation Algorithms for NP-hard Problems. Edited by D. S. Hochbaum, PWS Publishing Company, Boston, MA (1997) 1–45