Branch-and-Bound Style Resource Constrained Scheduling using Efficient Structure-Aware Pruning

Mingsong Chen*, Saijie Huang, Geguang Pu*
Shanghai Key Lab of Trustworthy Computing
East China Normal University, Shanghai, China
Email: {mschen, sjhuang, ggpu}@sei.ecnu.edu.cn

Prabhat Mishra
Department of Computer & Information Sci. & Eng.
University of Florida, Gainesville, FL, USA
Email: prabhat@cise.ufl.edu

Abstract—Branch-and-bound approaches are promising in pruning infeasible search space during the resource constrained scheduling (RCS). However, such heuristic approaches only compare the estimated upper and lower bounds of an incomplete schedule to the length of the best feasible schedule at that iteration. This paper proposes an efficient pruning technique which can identify the fruitless search space based on the detailed structural scheduling information of the obtained best feasible schedule. The proactive nature of our pruning technique enables the pruning of the space which cannot be identified by the state-of-the-art branch-and-bound techniques. The experimental results demonstrate that our approach can drastically (up to two orders-of-magnitude) reduce the overall RCS time under a wide variety of resource constraints.

I. INTRODUCTION

High-Level Synthesis (HLS) enables rapid generation of RTL hardware designs to satisfy performance, cost, area and power requirements [1]. In HLS, Electronic System Level (ESL) behavior descriptions are converted into Data Flow Graphs (DFGs), which are used as the intermediate representation by HLS algorithms. HLS involves two major tasks: scheduling and allocation. Scheduling refers to the assignment of operations to control steps (c-steps), and allocation binds the computation operations in DFGs to hardware resources. Currently, the scheduling problem is a major challenge in HLS, because it needs to make the trade-off among various constraints and explore a large number of possible alternatives to find an optimal or near-optimal design. In this paper, we focus on HLS under resource constraints, called Resource Constrained Scheduling (RCS). Given a DFG and a pre-defined set of functional or non-functional resources with specified overheads, RCS tries to find a schedule with minimum overall c-steps. Essentially, RCS is a scheduling problem with constraints of computation precedence and resource limits [9].

Since RCS is a NP-Complete problem, instead of enumerating all possible solutions, various approaches [3], [14] are proposed to efficiently prune infeasible or inferior candidates during the RCS search. Branch-and-bound (B&B) approaches [3] are widely used in existing RCS approaches to prune the search space. The B&B methods update the upper bound information of the schedule dynamically when encountering a better design candidate during the search. It utilizes such information to prune the inferior candidates which is worse than the best candidate obtained so far. Although the existing B&B algorithms are efficient in pruning the search space, they only investigate the upper and lower bound information of the scheduling to guide the search. In fact, its performance can be significantly improved by further pruning the search space using other potential avenues besides the upper and lower bounds.

In this paper, we propose an efficient algorithm for the RCS problem based on B&B methods, which investigate the DFG structure information of the up-to-date optimal scheduling. By comparing the scheduling time of partial operations, our approach can discard the incomplete schedules whose estimated lower-bound length equals to the length of the up-to-date optimal schedule. Therefore it can prune the search space in a proactive manner. Moreover, due to early detection of fruitless search, our approach can efficiently avoid deep recursive search that leads to drastic reduction in scheduling time.

This paper is organized as follows. Section II introduces the related work of RCS. Section III presents the related background and motivates the need for structure-aware pruning. Section IV proposes our approach which considers the structure information of the pruning techniques in detail. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. RELATED WORKS

Unlike non-optimal HLS heuristic methods (e.g., list scheduling [3], force directed scheduling [5]) which can achieve near-optimal schedules with less overhead, this paper focuses on how to quickly obtain an optimal RCS result in HLS. As an early popular approach, Integer Linear Programming (ILP) models [2], [6], [7] are widely used in HLS scheduling. However, the number of variables in ILP models increases very fast with the size of DFGs. Therefore, solving tight resource constrained problems using ILP models may need prohibitively long time.

The execution interval analysis approach is another widely investigated approach for HLS scheduling. The basic idea is to perform the lower bound estimation before real scheduling. Since it can prune inferior designs, the overall scheduling time can be reduced. In [4], Ohm et al. presented a comprehensive technique for lower bound estimation. Shen and Jong [8] proposed a stepwise refinement algorithm for resource estimation based on execution interval analysis. Their approach can handle loop folding and conditional branches at the same time. Therefore it can produce a tight bound quickly. However, most of these methods can only give near-optimal solutions rather an optimal one.

To achieve an optimal resource constrained HLS schedule, an obvious and time-consuming way would be to enumerate all the possible designs. To effectively avoid unnecessary searching during scheduling, Narasimhan and Ramanujam [3] presented a B&B approach called BULB using both lower- and upper-bound information to prune the search space. Consequently, the overall scheduling time can be drastically reduced. It is important to note that, in [3], the lower-bound cost of the subproblem cannot be added to the partial schedule, which may lead to worse lower-bound values of the complete problem. Hansen and Singh [13] proposed an efficient B&B approach to reduce the scheduling time under multi-resource constraints. In [14], Wu et al. presented a novel in-place search algorithm based on a systematic offspring generation algorithm,

* Corresponding authors.
which requires only a constant storage space during the traversal of the search tree.

Our approach is inspired by the work of [3], which can efficiently prune the search space in a B&B manner. To the best of our knowledge, our approach is the first attempt to utilize the structure information of the best schedule searched so far to prune the search space. A comparison between the BULB method [3] and our approach is presented in Section V.

III. BACKGROUND AND MOTIVATION

A. Graph-Based Notations of RCS Problem

RCS employs DFGs (Data Flow Graphs) to describe the dependence of operations. A DFG is a DAG (Directed Acyclic Graph) \( G = (V, E) \), where \( V \) is a set of vertices (nodes) designating functional operations with different types, \( E \) is a set of directed edges describing operation dependencies between nodes. For any two nodes \( v_i, v_j \in V \), \( (v_i, v_j) \in E \) indicates that the operation of \( v_j \) must complete before the start of the operation of \( v_i \). Consider the example DFG in Figure 1 that consists of 5 nodes and 4 directed edges. In a DFG, each \( v_i \) is tied with an operation \( op_i \), and \( type(op_i) \) indicates the type of functional unit that will be occupied by \( op_i \). delay(\( op_i \)) is used to denote time delay of \( op_i \). An operation without any predecessors is an input operation, and an operation without any successors is an output operation.

Fig. 1. An example of an HLS DFG

In RCS, the start time of an operation is regarded as the first \( c\text{-step} \) of its execution. The As-Soon-As-Possible (ASAP) notation estimates the start time of \( op_i \). Since resource constraints cannot be taken into account before the real scheduling, in most approaches, ASAP(\( op_i \)) estimates the earliest possible \( c\text{-step} \) when \( op_i \) can be started under the precedence constraints. Alternatively, the As-Late-As-Possible (ALAP) notation estimates the latest \( c\text{-step} \) when the operation \( op_i \) can be started. The notation [ASAP(\( op_i \)),ALAP(\( op_i \))], which is introduced in [3], indicates the execution intervals of operation \( op_i \).

Besides basic graph notations, various graph theory notations are used to enable the RCS analysis. In this paper, we use \( G' \) to represent the transpose graph of \( G \) by reversing all edge orientation. \( G' = (V', E') \) is a sub-graph of \( G = (V, E) \) if both \( V' \subseteq V \) and \( E' \subseteq E \). The sub-graph including nodes \( v_i \) and all its direct and indirect predecessors is denoted by \( G_{pre}(v_i) \). The sub-graph with \( v_i \) as its source node is denoted as \( G(v_i) \). The length of a path in a DFG indicates the number of nodes along the path. The weighted length of a path is the sum of operation delays of the nodes along the path, while the delay is determined by the type of nodes. The path in \( G \) with the longest length is called its critical path and the weighted path with the longest length is called weighted critical path. We use \( CPw(G) \) to denote the length of the weighted critical path of \( G \), and \( CPc(G) \) for the length of the critical path of \( G \). During the recursive B&B search of \( G \), the scheduling order \( \rho(G) \) of all operations is determined by the length of weighted critical path \( CPw(G(v_i)) \). As \( i \leq N \) in a non-ascending manner. As an example shown in Figure 1, either \( \rho_1(G) = \langle v_1, v_2, v_3, v_4, v_5 \rangle \) or \( \rho_2(G) = \langle v_2, v_4, v_3, v_1, v_5 \rangle \) can be a candidate of scheduling orders for \( G \), since \( CPw(G(v_1)) = 4 \), \( CPw(G(v_2)) = 4 \), \( CPw(G(v_4)) = 3 \), \( CPw(G(v_3)) = 3 \), and \( CPw(G(v_5)) = 1 \).

Level is an important notion to describe the structure of the RCS. In a DFG, the level of a node \( v \), denoted by Level(\( v \)), indicates the longest length from input nodes to \( v \), that is, Level(\( v \)) = \( CPc(G(pre(\( v \)))) \). For the example in Figure 1, the nodes are partitioned into 3 levels, and the level of \( v_3 \) is 3. When the level information of each node is fixed before the scheduling, we mark each node with an index from small level to large level. For a node \( v \), \( L_i(v) \) and \( L_d(v) \) denote the smallest and largest indices of the nodes within the same level of \( v \) respectively. For example, the DFG in Figure 1 is already marked using the level information. For the nodes \( v_1 \) and \( v_2 \), we can get \( L_i(v_1) = L_i(v_2) = 1 \) and \( L_d(v_1) = L_d(v_2) = 3 \).

B. Scheduling Based on ASAP and ALAP

In RCS, c-step is the basic time unit. An operation will occupy a specific number of continuous c-steps for execution on corresponding functional unit during the scheduling. A feasible scheduling for a DFG tries to dispatch operations under the operation dependence constraints posed by the DFG and the limited resources by the implementation requirement. As described in Definition 3.1, a schedule for a given DFG is an assignment function \( S \) which dispatches each operation \( op_i \) at c-step \( S(op_i) \in Z^+ \). Let \( S \) be a feasible schedule, its length \( le(S) \) is the largest finished time of all the operations, i.e., \( le(S) = \max \{ S(op_i) + delay(op_i) \mid op_i \in V \} \). A schedule is optimal if it is the up-to-date best with smallest length during the scheduling exploration. Among all possible schedules, the schedule with minimal length is regarded as the global optimal schedule.

Definition 3.1: Let \( G = (V, E) \) be a DFG of a behavior specification, and \( OP \) be the set of operations corresponding to \( V \), where \( |V| = |OP| = N \). Assuming that the target implementation supplies \( M \) types of functions, \( \Sigma = \{ \pi_1, \ldots, \pi_M \} \), and num(\( \pi_i \)) indicates the number of functional units of type \( \pi_i \) (\( 1 \leq i \leq M \)). A function \( S : OP \rightarrow Z^+ \) is a feasible schedule of \( G \), iff it satisfies all the following conditions:

1. If \( \{ op_i, op_j \} \in E \) where \( 1 \leq i, j \leq N \), then \( S(op_i) + delay(op_i) \leq S(op_j) \) holds.
2. For any time \( t \) and any operation of type \( \pi_j \), \( | \{ op_i \mid type(op_i) = \pi_j \wedge (S(op_i) + delay(op_i)) \in [t, t+1) \} | \leq \text{num}(\pi_j) \).

Here condition (1) gives the precedence constraint posed by given DFG, and condition (2) indicates the resource constraints during the scheduling of DFG operations. We use \( (op_i, S(op_i)) \) to denote the scheduling pair for operation \( op_i \). For example, the binary relation \( \{ (op_1), (op_2), (op_3), (op_4), (op_5) \} \) is a feasible schedule with length \( 7 \) for the DFG in Figure 1. And the binary relation \( \{ (op_1), (op_2), (op_3), (op_4), (op_5) \} \) is an optimal schedule with length \( 5 \).

C. Motivation

Based on the permutation of c-steps on the [ASAP,ALAP] of all operations, B&B approaches [3, 13] can prune useless search space effectively. Besides [ASAP,ALAP] intervals which restrict the search space of each operation, B&B approach uses other two important data structures to prune infeasible schedules: i) the optimal schedule \( S_{opt} \) which keeps the up-to-date optimal schedule, and ii) current schedule \( S \) which holds current (incomplete) schedule.

Figure 2a) illustrates how the B&B search makes the pruning. Since it is impossible to determine the optimal schedule length before
all the enumerations are done, to restrict search space, the upper bound and lower bound of the optimal schedule are estimated. We use $\omega$ to record the upper-bound length of $S_{opt}$. Initially, $\omega$ equals to the length of a feasible schedule determined using the list scheduling approach. Then $\omega$ decreases dynamically when a faster feasible schedule is found during the RCS exploration. GlobalLow is the lower bound length of $S_{opt}$, which is calculated using the approach proposed in [10]. In Figure 2, optimal indicates the length of the global optimal schedule, which is in the range $[\text{GlobalLow}, \omega]$. The current schedule $S$, which is an incomplete enumeration, also has two bound estimations. We use lower and upper to denote the lower-bound estimate and upper-bound estimate for $S$ respectively. Note that $\omega$ is the length of a complete schedule $S_{opt}$, but in $S$, not all the operations are scheduled. Therefore, the lower and upper of $S$ also can be considered as the lower-bound and upper-bound respectively for the schedules which have the same c-steps on each scheduled operations of $S$. In the B&B approach, if lower is larger than $\omega$, the dispatching of the unscheduled operations will not continue, since optimal should be in the range $[\text{GlobalLow}, \omega]$. However, if upper is smaller than $\omega$, a schedule which is better than $S_{opt}$ is found. Consequently, $S_{opt}$ will be replaced by the new schedule.

**A. Level-Bound Pruning Heuristics**

To further improve the pruning performance, we proposed a novel structure-aware pruning technique called level-bound heuristic which fully utilizes the precedence relation information of DFG operations.

Figure 3 depicts the basic idea of our level-bound pruning method, where a) presents an optimal schedule searched so far of a DFG $G$, and b) shows an incomplete schedule of $G$. Each node in the figures is marked with a number, which indicates the dispatch time of the corresponding operation. Suppose that in $S$ the nodes of levels lower than or equal to level $X$ have been scheduled and the other nodes (i.e., in levels larger than $X$) have not been determined. For traditional B&B approaches, they only compares the value between $le(S_{opt})$ and the value of both the lower and upper of $S$ for pruning. Note that, in Figure 3, the dispatching time of all $S_{opt}$’s operations in the $X_{th}$ level is better than the one of $S$. Such structural scheduling information (i.e., the c-steps of partial operations and corresponding operation precedence relations) can be used to prune the search space.

**Definition 4.1:** For a given DFG $G$, a cut is an edge set $EG = \{e_1, ..., e_n\}$ where all the following conditions hold:

1) For each edge $e_i$ ($1 < i \leq n$), there exists a path from some source node (i.e., a node without any incoming edges) to $e_i$.
2) For each source node in $G'$, there exists a path from the source node to some $e \in EG$.

For a given cut, $\text{Pre}(\text{CUT})$ denotes its precedent node set; $\text{Imme}(\text{CUT})$ denotes the adjacent input nodes of the edges in the cut; and $\text{Post}(\text{CUT})$ denotes the set of descendant nodes. A cut that separates the nodes in level $X$ and the nodes in the levels lower than $X$ is called an $X_{th}$-level cut of $G$. Assuming that cut$_X$ is the $X_{th}$-level cut of $G$, $\text{Imme}(\text{cut}_X)$ is the $X_{th}$ complete level of $G$.

In RCS, checking two schedules node by node is neither efficient nor necessary. Alternatively, we introduce the cut notation which enables the fast comparison and pruning during RCS. A cut divides a DFG into two disjoint parts where the first part is the set of all the precedent nodes of the cut while the second part is the set of all the descendant nodes of the cut. Consider the example in Figure 1, $\{e_3, e_4\}$ is the second level cut of $G$. However, $\{e_4\}$ is not a cut, since there is no path from $e_4$ to $v_5$ in $G'$. In $G$, $\text{Pre}(\{e_3, e_4\}) = \{v_1, v_2, v_3, v_4\}$, $\text{Imme}(\{e_3, e_4\}) = \{v_3, v_4\}$ and $\text{Post}(\{e_3, e_4\}) = \{v_5\}$. The second complete level of $G$ is $\{v_3, v_4\}$.

For a given DFG $G$ and some given cut, if we have two schedules (i.e., $S_1$ and $S_2$) which have just finished the scheduling of the node set $\text{Pre}(\text{CUT})$, our level-bound heuristics can be used to determine whether some incomplete schedule can be abandoned only based on the comparison of the scheduling information of the same cut of $S_1$.
and $S_2$. Before proving the correctness of our level-bound heuristics (Theorem 4.1), we define local optimal schedule which is essentially based on the result of an incomplete schedule.

**Definition 4.2:** Let $OP$ be the node set of a given DFG $G$. Assume that $S$ is an incomplete schedule of $G$ and node set $OP_{\text{sch}} = \{op_1, op_2, \ldots, op_n\}$ is a subset of scheduled operations. A local optimal schedule $S_{\text{op, sch}}$ is a complete schedule which has fixed values for $OP_{\text{sch}}$ and the optimal scheduling for the unscheduled operations (i.e., $OP \setminus OP_{\text{sch}}$). That means $S_{\text{op, sch}}$ is a global optimal schedule in the search space $\Pi_{\text{pre}}[\Pi_{\text{post}}[\Pi_{\text{sch}}(S_{\text{op}})]] \times \Pi_{\text{op, \text{sch}}}(S_{\text{op, sch}})$. \hfill $\blacksquare$

The local optimal schedule $S_{\text{op, sch}}$ denotes when a set of operations $OP_{\text{sch}}$ is assigned with fixed $c$-steps in advance, it searches for the "global" optimal solution from the rest of the nodes. Obviously, $S_{\text{op, sch}}$ is an optimal schedule but may not be the globally optimal schedule of the given DFG.

**Theorem 4.1:** Assume that $S_1$ and $S_2$ are two incomplete schedules of a given DFG $G = (V, E)$, and $CUT$ is a cut of $G$. Let $S_{1, \text{Pre}(CUT)}$ and $S_{2, \text{Pre}(CUT)}$ be two different local optimal schedules based on $\text{CUT}$, and $\text{Imme}(\text{CUT})$ be an operation set $\{op_1, op_2, \ldots, op_n\}$. Assume that $o_1 = \text{le}(S_{1, \text{Pre}(CUT)})$ and $o_2 = \text{le}(S_{2, \text{Pre}(CUT)})$. We can conclude that $\text{Imme}(\text{CUT}) \subseteq S_{1, \text{Pre}(CUT)}$ and $\text{Imme}(\text{CUT}) \subseteq S_{2, \text{Pre}(CUT)}$. Thus, $S_{1, \text{Pre}(CUT)} \cup \text{Imme}(\text{CUT})$ and $S_{2, \text{Pre}(CUT)} \cup \text{Imme}(\text{CUT})$ are two incomplete schedules of $G$.

Proof: From the definition of local optimal schedule, we know that $S_{1, \text{Pre}(CUT)}$ is obtained from the search space $\Pi_{\text{op, \text{Pre}(CUT)}}[\Pi_{\text{op, \text{Post}(CUT)}}[\Pi_{\text{op, \text{sch}}}(S_{1, \text{Post}(CUT)})]] \times \Pi_{\text{op, \text{Post}(CUT)}}[\Pi_{\text{op, \text{sch}}}(S_{1, \text{Post}(CUT)})]$, while $S_{2, \text{Pre}(CUT)}$ is obtained from the search space $\Pi_{\text{op, \text{Pre}(CUT)}}[\Pi_{\text{op, \text{Post}(CUT)}}[\Pi_{\text{op, \text{sch}}}(S_{2, \text{Post}(CUT)})]] \times \Pi_{\text{op, \text{Post}(CUT)}}[\Pi_{\text{op, \text{sch}}}(S_{2, \text{Post}(CUT)})]$. Ignoring the scheduled nodes in $\text{Pre}(\text{CUT})$, we schedule nodes in $\text{Post}(\text{CUT})$ independently. Based on the scheduled nodes in $\text{Pre}(\text{CUT})$, the ASAP values on the $\text{Post}(\text{CUT})$ nodes of $S_1$ and $S_2$ can be dynamically updated. Due to the fact that if $\bigwedge_{1 \leq j \leq k} \text{Imme}(\text{CUT}) \subseteq S_{1, \text{Post}(CUT)}$ and $\bigwedge_{1 \leq j \leq k} \text{Imme}(\text{CUT}) \subseteq S_{2, \text{Post}(CUT)}$, we can get that $\text{ASAP}(\text{op}) \leq \text{ASAP}(\text{op})$ and $\text{ASAP}(\text{op}) \leq \text{ASAP}(\text{op})$.

Algorithm 1 describes the implementation of our level-bound pruning approach. Step 1 checks whether all the operations in current level have been scheduled. Step 2 obtains the \text{(level(op),)} \text{sch} complete level. Step 3 asserts that $S_{\text{op}}$ outperforms $S$ on the level-bound condition, and step 4 asserts otherwise. Note that, to save time, the level bound based approach will be invoked only when all the operations in the \text{(level(op),)} \text{sch} complete level are dispatched. \hfill $\blacksquare$

\begin{algorithm}
\caption{Our Level-Bound Pruning Algorithm}
\begin{algorithmic}
  \Input\ 
  \item $S_{\text{opt}}$, which is the best schedule searched so far
  \item $S$, which stores the current incomplete schedule
  \item $op_k$, which is the last dispatched operation of $S$
  \Output\ 
  \item $S_{\text{op}}$ outperforms $S$
  \item $S_{\text{op}}$, which is the last dispatched operation of $S$
  \Function{LevelBound}{$S_{\text{ss}}, op_k$}
  \If{$i \neq \text{le}(op_k)$}
    \State return false;
  \Else
    \State $OP = \{op_1, op_2, \ldots, op_k\}$ complete level
    \State $\bigwedge_{1 \leq j \leq k} \text{Imme} \subseteq S_{\text{op}}(op_k)$ \& \& $\bigwedge_{1 \leq j \leq k} \text{Imme} \subseteq S_{\text{op}}(op_k)$
    \If{$\bigwedge_{1 \leq j \leq k} \text{Imme}(op_k) \subseteq S_{\text{op}}(op_k)$ \& \& $\bigwedge_{1 \leq j \leq k} \text{Imme}(op_k) \subseteq S_{\text{op}}(op_k)$}
      \State return true;
    \Else
      \State return false;
  \EndIf
  \EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

Algorithm 1 describes the implementation of our level-bound pruning approach. Step 1 checks whether all the operations in current level have been scheduled. Step 2 obtains the \text{(level(op),)} \text{sch} complete level. Step 3 asserts that $S_{\text{op}}$ outperforms $S$ on the level-bound condition, and step 4 asserts otherwise. Note that, to save time, the level bound based approach will be invoked only when all the operations in the \text{(level(op),)} \text{sch} complete level are dispatched. \hfill $\blacksquare$

**B. HLS Scheduling using Our Approach**

Compared to B&B approach which uses $\text{value}$ only for pruning, our level-bound based heuristic employs the structure information of both current and optimal schedules. The level-bound pruning approach can be combined with B&B approach to explore more pruning chances, which can improve the RCS performance drastically.

Algorithm 2 presents our RCS approach that uses both B&B and level-bound heuristics. Step 1 initializes the variable $\text{suTimes}$ which indicates the times of the successful occupation of the resource by $op_k$. If $op_k$ is the final unscheduled operation in \text{(level(op),)} \text{sch} and LevelBound($S, S_{\text{op}}, op_k$) returns true, it means that the current schedule stored in $S$ is worse than $S_{\text{op}}$. Then step 2 sets the global Boolean variable $jump$ to be true, which indicates that the level-bound checking may lead to some backtrack of more
than one operations. Step 3 stops the c-step enumeration of \( op \). If \( \text{LevelBound}(S, S_{opt}, op) \) returns false, the operation \( op \) will be scheduled. Step 4 indicates that the operation successfully gets the intended resource. Steps 5 and 6 calculate the lower and upper for the current schedule. If \( \text{upper} \) is smaller than \( \text{lower} \), then \( \text{upper} \) and \( S_{opt} \) will be updated in steps 7 and 8. Changing \( \text{upper} \) value will trigger the checking of early termination condition (i.e., achieving a schedule whose length equals to \( \text{GlobalLow} \)) in step 9. Step 10 does the runtime space shrinking by update the ALAP values of operations. If \( \text{lower} \) is smaller than \( \text{lower} \), current operation will be scheduled. Step 11 assigns a c-step to operation \( op \). Step 12 reserves resources required by the operation. Then \( op_{p+1} \) is processed recursively in step 13. When the search backtracks, the resource occupied by \( op_{p+1} \) is released in step 14. Steps 15 and 16 deal with the backtracks caused by the level-bound checking. When \( \text{succTimes} \) equals to 1, it means that the current step is the smallest for \( op \) based on the scheduled operations. Since all the subsequent operations in the dispatching order have been enumerated, the search by increasing step by 1 is fruitless. Therefore, step 15 indicates that the search of \( op \) can be stopped. Step 16 continues to enumerate the following c-steps of the current operation. Finally, the algorithm returns one global optimal schedule and its length.

**Algorithm 2: Structure-Aware B&B Pruning**

**Input:**
1. \( D \), which is an HLS DFG with resource constraints;
2. \( op = \{op_1, \ldots, op_n\} \) in dispatching order;
3. \( S_{opt} \), which is a feasible schedule of length \( s \);
4. \( S \), which stores the current incomplete schedule;
5. \( \text{jump} = \text{false} \), which is a global Boolean variable;

**Output:** An optimal HLS schedule and its length \( (S', s') \)

OurMethod\((D, OP, i, N, S_{opt}, s, \text{jump})\) begin

1. \( \text{sucTimes} = 0; \)
   for \( \text{step} = \text{ASAP}(op) \) to \( \text{ALAP}(op) \) do
   if \( \text{LevelBound}(S, S_{opt}, op) \) then
   2. \( \text{jump} = \text{true}; \)
   3. return \( (S_{opt}, s); \)
   end
   if \( \text{precedence}(op) \land \text{resAvailable}(\text{step}, \text{type}(op)) \) then
   4. \( \text{sucTimes} = +1; \)
   5. \( \text{lower} = \text{LBound}(op); \)
   6. \( \text{upper} = \text{ListScheduling}(OP, op); \)
   if \( \text{upper} < \text{lower} \) then
   7. \( \omega = \text{upper}; \)
   8. \( S_{opt} = \text{ListScheduling}(OP, op); \)
   if \( \omega = \text{GlobalLow}(D) \) then
   9. \( \text{Terminate}(S_{opt}, \omega); \)
   end
   10. \( \text{UpdateALAP}(); \)
   end
   if \( \text{lower} < \omega \) then
   /* Dispatch the current operation */
   11. \( \text{ResOccupy}((\text{step}, \text{type}(op), \text{delay}(op)); \)
   12. \( \text{ResOccupy}((\text{step}, \text{type}(op), \text{delay}(op)); \)
   13. \( \text{OurMethod}(D, OP, i + 1, N, S_{opt}, s); \)
   14. \( \text{ResRestore}((\text{step}, \text{type}(op), \text{delay}(op)); \)
   if \( \text{jump} \) then
   15. \( \text{return}(S_{opt}, s); \)
   else
   16. \( \text{jump} = \text{false}; \)
   end
   end
   end
   return \( (S_{opt}, s); \)

V. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our approach, we conducted various experiments with different kinds of resource constraints. We collected the following benchmarks from the MediaBench benchmark [11], which is a standard DSP benchmark suite: i) ARFilter with 28 nodes and 30 edges, ii) Cosine I with 66 nodes and 76 edges, iii) Collapse with 56 nodes and 73 edges, and iv) Feedback with 53 nodes and 50 edges. We also used the benchmark FDCT with 42 nodes and 52 edges from [16]. By using the C programming language, we implemented both the BULB method and our approach that integrates the structure-aware pruning heuristic. For comparison, we also derived ILP models for RCS using IBM ILOG CPLEX CP Optimizer [12], which adopts the branch-and-cut heuristic method for efficient searching. All the experimental results were obtained on a Linux machine with Intel 2.0GHz processors and 3 GB RAM.

### Table I

**The Settings of the Functional Units**

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Operation Class</th>
<th>Delay (unit)</th>
<th>Power (unit)</th>
<th>Energy (unit)</th>
<th>Area (unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD/NDB</td>
<td>4/</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>MUL/DIV</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>MEM</td>
<td>LD,STR</td>
<td>1</td>
<td>15</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;,&gt;&gt;</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Other</td>
<td>...</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

In this experiment, we consider functional unit constraints as well as non-functional constraints (i.e., power and area). Table I lists the corresponding settings for various types of functional operations used in the experiment.

### Table II

**Scheduling Results under Functional Unit Constraints**

<table>
<thead>
<tr>
<th>Design</th>
<th># of ( \tau ), ( x )</th>
<th>ILP [12] (sec.)</th>
<th>BULB (sec.)</th>
<th>Ours (sec.)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARFilter</td>
<td>1, 3</td>
<td>0.34</td>
<td>0.14</td>
<td>2.43</td>
<td></td>
</tr>
<tr>
<td>Cosine</td>
<td>1, 2</td>
<td>0.86</td>
<td>0.26</td>
<td>3.31</td>
<td></td>
</tr>
<tr>
<td>Feedback</td>
<td>4, 4</td>
<td>2.32</td>
<td>0.01</td>
<td>&lt;0.01</td>
<td>1.00</td>
</tr>
<tr>
<td>Collaps</td>
<td>2, 1</td>
<td>105.67</td>
<td>23.38</td>
<td>4.52</td>
<td></td>
</tr>
<tr>
<td>Collaps</td>
<td>2, 2</td>
<td>611.75</td>
<td>65.91</td>
<td>9.28</td>
<td></td>
</tr>
<tr>
<td>Cosine</td>
<td>2, 2</td>
<td>0.02</td>
<td>&lt;0.01</td>
<td>&gt;2.00</td>
<td></td>
</tr>
<tr>
<td>Feedback</td>
<td>4, 5</td>
<td>171.67</td>
<td>154.94</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>Feedback</td>
<td>5, 5</td>
<td>5.53</td>
<td>4.96</td>
<td>1.11</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>2, 1</td>
<td>38.05</td>
<td>23.52</td>
<td>1.62</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>2, 2</td>
<td>210.22</td>
<td>18.67</td>
<td>11.26</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>2, 4</td>
<td>21.26</td>
<td>4.12</td>
<td>5.16</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>2, 4</td>
<td>4.31</td>
<td>2.00</td>
<td>2.16</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>2, 5</td>
<td>0.99</td>
<td>0.61</td>
<td>1.62</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>3, 4</td>
<td>0.64</td>
<td>0.51</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>FDCT</td>
<td>4, 4</td>
<td>0.13</td>
<td>0.02</td>
<td>6.50</td>
<td></td>
</tr>
</tbody>
</table>

* Timeout means the scheduling time is larger than 3600 seconds.*

Table II presents the experimental results carried out with different functional unit constraints on the five benchmarks. The first column of the table has two sub-columns. The first sub-column indicates the name of the benchmarks. The second sub-column presents the functional unit constraint for the design. We use notation “\( \tau \), \( x \)” to denote that only \( x \) adders and \( y \) multipliers are adopted for the RCS. For example, in the first row of ARFilter design with 2 adders and 3 multipliers can achieve the optimal result within the time limit. The third column presents the scheduling time using the CPLEX CP Optimizer. In all benchmark items, only one of them (i.e., ARFilter design with 2 adders and 3 multipliers) can achieve the optimal result within the time limit. The fourth column shows the results using our structure-aware approach. The last column indicates overall speedup using our approach over the BULB approach, i.e.,...
From this table, we can find that our approach can not only outperform the state-of-the-art ILP solver with the branch-and-cut heuristic, but also it can drastically improve the RCS performance using the B&B style searching. For instance, when handling the design Collapse under the constraint of two adders and one multiplier, BULB method needs more than 3600 seconds for scheduling, which is not acceptable in many scenarios. However, our approach requires only 234.76 seconds. Generally, during RCS, the cost of the detection and the check of level-bound information is negligible. However, it can effectively prune the space where the search is fruitless. Therefore, in most cases, our approach can achieve a significant improvement.

The area and the power are two key issues in hardware design. The scheduling under these two constraints can be considered as a variant of the time-minimum resource scheduling [13]. Since both area and power can be treated as special kinds of resources, our proposed structure-aware pruning can also be used to promote the scheduling performance under such non-functional constraints. We did the experiment with the designs given in Table II using these two constraints. Due to the space limit, we only present the results for the FDCT design. All the other designs have the similar conclusion based on their results.

![Fig. 4. Scheduling results with an area of 140 units](image)

![Fig. 5. Scheduling results with an area of 100 units](image)

Figure 4 shows the RCS result using the BULB approach and our approach. Under the area constraint of 140 units, we check the RCS with different power constraints (from 60 to 120 with an increment of 10). We can find that when the power is taken into consideration during the RCS (i.e., the case with a power constraint 60), our approach can get the maximum speedup (more than 22 times). It is important to note that, for the design of FDCT with a 140-unit area constraint, the threshold of the power is 80 units. When the power limit is larger than 80 units in this case, the power constraint will not alter the outcome. Therefore, we can find that the last five instances in Figure 4 show the similar results. We also tried to do RCS for the FDCT design with a smaller area constraint (i.e., 100 units). As shown in Figure 5. We can find a similar trend in Figure 4. In this case with a power constraint of 50 units, none of the BULB approach and our method can figure out any optimal schedule within the time limit (i.e., 3600 seconds). Although it shows the same result in this case, in fact the results cannot be compared. From above figures, we can find that our approach is very promising in pruning the searching space of designs with power and area constraints. Our approach can achieve up to 101 times improvement (for the case with 100-unit area and 40-unit power) compared to the BULB approach [3].

VI. CONCLUSION

This paper proposes a structure-aware approach to prune search space efficiently during resource-constrained HLS scheduling. Unlike existing B&B algorithms which are based on the comparison of upper and lower bounds between optimal schedule searched so far and incomplete searching schedules, our approach investigates structural scheduling information of the optimal schedule candidates during the search to prune other non-optimal schedules in a proactive manner. Our approach is orthogonal with the existing B&B approaches. The synergy of them can promote the overall RCS performance drastically by several orders of magnitude. Experimental results using various benchmarks with different resource constraints show that our method can achieve significantly better performance than existing state-of-the-art B&B techniques.

ACKNOWLEDGMENT

This work was partially supported by the grants from Natural Science Foundation of China (No. 61202103, 91118008, and 91118007), Open Project of Software/Hardware Co-design Engineering Research Center of MoE (No. 2013001), Shanghai Knowledge Service Platform Project (No. ZF1213), and Civil Aerospace Project 125.

REFERENCES