Efficient Resource Constrained Scheduling Using Parallel Structure-Aware Pruning Techniques

Mingsong Chen, *Member, IEEE*, Xinqian Zhang, Geguang Pu, Xin Fu, *Member, IEEE*, and Prabhat Mishra, *Senior Member, IEEE*

Abstract—Branch-and-bound approaches are promising in pruning fruitless search space during the resource constrained scheduling. However, such approaches only compare the estimated upper and lower bounds of an incomplete schedule to the length of the best feasible schedule at that iteration, which does not fully exploit the potential of the pruning during the search. Aiming to improve the performance of resource constrained scheduling, this paper proposes a parallel structure-aware pruning approach that can traverse the search space significantly faster than state-of-the-art branch-and-bound techniques. This paper makes three major contributions: i) it proposes an efficient pruning technique using the structural scheduling information of the obtained best feasible schedules; ii) it investigates how to perform parallel search to enable efficient multi-directional search and generation of effective fences by tuning the operation enumeration order; and iii) it presents a framework that supports the sharing of minimum upper-bound and fence information among different search tasks to enable efficient parallel structure-aware pruning. The experimental results demonstrate that our parallel pruning approach can drastically reduce the overall resource constrained scheduling time under a wide variety of resource constraints.

Index Terms—Resource constrained scheduling, branch-and-bound, parallel pruning, structure-aware pruning

1 Introduction

By raising the design abstraction to electronic system levels (ESLs), high-level synthesis (HLS) enables rapid generation of RTL hardware designs to satisfy performance, cost, area and power requirements [1], [2]. More and more HLS tools have been adopted for designing efficient systems in a wide variety of domains including field-programmable gate array (FPGA) based systems [3] and application specific multiprocessor designs [4]. HLS solutions show that they can achieve optimized designs as well as improve design productivity.

To enable the design space exploration and performance estimation, ESL behavior descriptions are converted into *data flow graphs* (DFGs), which are used as the intermediate representation by HLS algorithms. HLS involves three major tasks: scheduling, allocation and binding. Scheduling refers to the assignment of operations to control steps (*c-steps*). Allocation and binding map the computation operations in DFGs to hardware resources. In HLS, scheduling is a major challenge, because it needs to make the trade-off between 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) [5], [6]. Given a DFG and a pre-defined set of 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 [15].

RCS is an NP-Complete problem [5], [6]. Instead of enumerating all feasible schedules, various approaches [7], [8] are proposed to efficiently prune inferior solutions for RCS. Branch-and-bound (B&B) approaches [7], [9] are promising in existing RCS approaches to prune the search space. During the state space exploration, the B&B methods update the upper bound information of the optimal schedule dynamically when encountering a better schedule with shorter length. They utilize the updated upper-bound information to shrink the search space as well as to strengthen the pruning of inferior schedules. As an example, Fig. 1a shows a typical B&B search scenario. Due to a loose upper-bound estimation (i.e., ω), the search space denoted by the solid circle is large, and the B&B search indicated by the solid arrow line is slow. In this stage, the B&B search can only filter schedules whose lengths are larger than or equal to ω. When a better schedule (denoted by black dots) with shorter length (i.e., ω') is obtained, the space of the following search indicated by the dashed arrow line will be restricted by the new upper bound ω'. Consequently, the overall search space gets reduced. Meanwhile, the B&B search with shorter upper bound ignores all the schedules whose lengths are larger than or equal to ω', thus the pruning rate is accelerated. Although the above B&B search pattern is promising in pruning unfruitful search space, it only utilizes one search task, and only the upper and lower bound information is used for pruning. If such search pattern can

- *M. Chen, X. Zhang, and G. Pu are with the Shanghai Key Laboratory of Trustworthy Computing at East China Normal University, Shanghai 200062, China. E-mail: {mschen, xqzhang, ggpu}@sei.ecnu.edu.cn.*
- *X. Fu is with the Department of Electrical and Computer Engineering, University of Houston, Houston, TX 77204. E-mail: xfu8@central.uh.edu.*
- *P. Mishra is with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120. E-mail: prabhat@cise.ufl.edu.*

Manuscript received 6 Oct. 2014; revised 27 July 2015; accepted 2 Aug. 2015. Date of publication 12 Aug. 2015; date of current version 15 June 2016. Recommended for acceptance by L. Van.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2468230

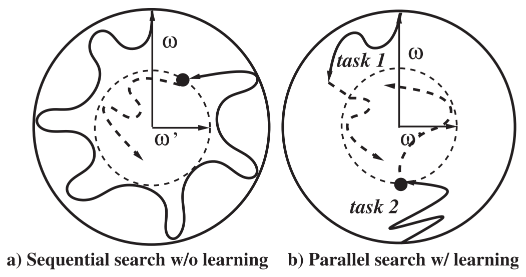
ω
ω'
task 1
ω
ω'
task 2
a) Sequential search w/o learning
b) Parallel search w/ learning

Fig. 1. Sequential versus parallel search.

be improved by using other potential avenues besides sequential search and bound-based pruning, the switching time between different pruning rate will be reduced. Therefore the overall searching time can be reduced drastically.

Since more and more computers are supporting multi-core or many-core computation, the pruning efficiency can be improved further by utilizing the parallelism. Fig. 1b shows a parallel search example with two search tasks (task 1 is at the top, and task 2 is at the bottom). We assume that search task 2 can find a better schedule earlier than task 1. Since the shorter length of the better schedule can be used as a new upper-bound for the scheduling, the search space of both tasks can be reduced simultaneously. Moreover, due to the early detection of a smaller upper-bound, the pruning rate is improved earlier than the case in Fig. 1a. Therefore, in Fig. 1b the B&B search of task 1 is accelerated. From this example, we can find that the interactions between different search tasks can benefit the overall search. Based on the above observations, designing an efficient parallel RCS approach needs to address the following three challenges.

1) How to spawn multiple search tasks to enable efficient parallel exploration on a given search space? In this paper, we investigate the tuning of operation enumeration order to enable multi-directional parallel search. It enables the parallel search and generation of fences at different parts of search space, which increases the chance of early detection of optimal schedules.
2) How to achieve a better schedule faster than traditional B&B approach? In this paper, we propose an efficient level-bound pruning technique for the RCS problem based on B&B methods, which exploits the DFG structure information of the up-to-date optimal schedules. By comparing the scheduling time of partial operations, our approach can discard the incomplete schedules whose estimated upper-bound length equals to the length of the up-to-date optimal schedule. Therefore, it can prune the search space in a proactive manner.
3) How to share learning information among different search tasks to accelerate the searching? This paper presents a collaborative framework that supports the sharing of fences and minimum upper-bound among different search tasks to enable efficient parallel level-bound pruning. Each parallel search task can have multiple "fences" to efficiently avoid deep recursive search that leads to drastic reduction in scheduling time.

This paper is organized as follows. Section 2 introduces the related work on RCS. Section 3 presents the related background and motivates the need for structure-aware pruning. Section 4 proposes our scheduling approach using structure-aware pruning. Section 5 presents our experimental results. Finally, Section 6 concludes the paper.

2 Related Work

Unlike non-optimal RCS heuristic methods (e.g., list scheduling [5], [6], Tabu search [11]), this paper studies how to quickly obtain a tight schedule under resource constraints. In the early stage, *integer linear programming* (ILP) based approaches [10], [13] were widely investigated in HLS scheduling. For example, Gebotys and Elmasry [10] proposed a set of efficient formulas that can drastically reduce the solving time of ILP methods. By using a relaxed ILP formulation together with a greedy algorithm, Rim and Jain [13] presented an approach that can optimize lower bounds of operations. Although ILP-based methods allow designers to describe the RCS problem naturally, the number of variables in ILP models increases very fast with the size of DFGs. Consequently, solving complex RCS problems using ILP models may need prohibitively long time.

Execution interval analysis is another popular approach for HLS scheduling. Its basic idea is to perform lower- and upper-bound estimation before real scheduling. Based on relaxing precedence constraints in behavioral design descriptions, Timmer and Jess [12] presented a unified approach for lower-bound functional area and cycle budget estimations. Shen and Jong [14] 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 quickly produce a tight bound. Although execution interval analysis can restrict the search within a small range, most of existing methods are invented to find near-optimal solutions.

To achieve an optimal resource constrained HLS schedule, an obvious but time-consuming way is to enumerate all the feasible designs [9]. To effectively avoid unnecessary enumeration of inferior schedules, Narasimhan and Ramanujam [7] presented a B&B approach called BULB using both lower- and upper-bound information to prune the search space. In [21], Hansen and Singh proposed an efficient B&B approach that can reduce the scheduling time considering various resource constraints. In [26], Chen et al. outlined a two-phase B&B approach that can achieve a shorter initial upper-bound estimation to enable the overall quick search within a new smaller search space. In [8], Yu et al. presented an in-place approach based on a systematic offspring generation algorithm, which requires only a constant storage space during the traversal of the search tree. However, so far, most RCS optimization methods only use a single-core to figure out the solution.

Parallelism is widely studied in HLS [22], [25]. Chen et al. [25] proposed a bound-oriented parallel B&B approach for HLS. Based on the best upper- and lower-bound information shared among parallel sub-search tasks, [25] can efficiently locate an optimal result by combining both bound speculation and search space partitioning techniques. However, it heavily relies on the analysis of bound information,

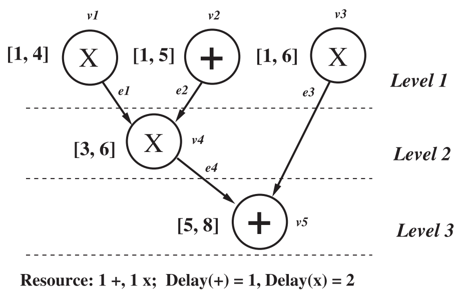

Fig. 2. An example of an HLS DFG.

and does not utilize other features for pruning. Moreover, it does not study the effect of operation ordering during the parallel search.

Although parallel B&B approaches have been successfully adopted in many domains [27], few of them were considered in HLS. The BULB approach [7] applied branch-and-bound for RCS but did not consider schedule structure or parallel pruning techniques to reduce the RCS efforts. To the best of our knowledge, our approach is the first attempt to utilize the structure information of best schedules obtained so far by sub-search tasks to prune the search space in parallel.

3 Preliminary Knowledge

This section introduces basic ideas for HLS scheduling, including graph-based notations, scheduling related terms, and a classic B&B RCS algorithm—BULB [7].

3.1 Graph-Based Notations of RCS Problem

RCS employs DFGs to describe the dependence of operations. A DFG is a Directed Acyclic Graph (DAG) $G = (V, E)$, 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. In a DFG, each v_i is tied with an operation op_i. Therefore in this paper, we use v_i and op_i interchangeably. For any two nodes $v_i, v_j \in V$, $\langle v_i,\ v_j\rangle \in E$ indicates that the operation of v_i must complete before the start of the operation of v_j. We use $type(op_i)$ to indicate the type of functional unit that will be occupied by op_i, and $delay(op_i)$ to denote the time delay of op_i. In our approach, we assume that the operations with the same type have the same delay. In other words, if $type(op_i) = type(op_j)$, we can get $delay(op_i) = delay(op_j)$. An operation without any predecessors is an *input operation*, and an operation without any successors is an *output operation*. Consider an example shown in Fig. 2. This DFG consists of six nodes and five directed edges. It uses two kinds of resources (i.e., adders and multipliers). For example, op_4 is a multiplication operation with a delay of two c-steps. This DFG has three input operations and one output operation. For instance, v_1 is an input operation, and v_6 is an output operation.

Besides basic graph notations, various graph theory notations are used to enable the RCS analysis. In this paper, we use G^{τ} 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 that includes the node v_i and all its connected successors 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 $CP_w(G)$ to denote the length of the weighted critical path of G, and $CP_l(G)$ 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 $CP_w(G(v_i))$ $1 \leq i \leq N$ in a non-ascending manner. For example in Fig. 2, either $\rho_1(G) = <v_1, v_2, v_3, v_4, v_5, v_6>$ or $\rho_2(G) = <v_1, v_3, v_2, v_4, v_5, v_6>$ can be a candidate of scheduling orders for G, since $CP_w(G(v_1)) = 6$, $CP_w(G(v_2)) = 5$, $CP_w(G(v_3)) = 5$, $CP_w(G(v_4)) = 4$, $CP_w(G(v_5)) = 3$ and $CP_w(G(v_6)) = 2$.

As an important notion to describe the RCS structure, the level of a node v (denoted by $Level(v)$) indicates the longest length from input nodes to v, i.e., $Level(v) = CP_l(G_{pre}(v))$. For the example in Fig. 2, the nodes are partitioned into three levels. For example, the level of v_3 is 1 and the level of v_5 is 2. Before the scheduling, we need to mark the level information for each node from small levels to large levels. For a node v, $L_l(v)$ and $L_h(v)$ denote the indices of the earliest and latest visited nodes within the same level of v respectively. For example, assume that in Fig. 2 the operation scheduling order for G is $<v_1, v_2, v_3, v_4, v_5, v_6>$. Since v_1, v_2 and v_3 are in the same level, and v_1 and v_3 are the first and last visited operation in that level respectively, we can get $L_l(v_1) = L_l(v_2) = L_l(v_3) = 1$ and $L_h(v_1) = L_h(v_2) = L_h(v_3) = 3$.

3.2 Scheduling Based on ASAP and ALAP

In RCS, an operation occupies a specific number of continuous *c-steps* for execution on corresponding functional unit during the scheduling. The start time of an operation is regarded as the first *c-step* of its execution. The *As-Soon-As-Possible* notation estimates the earliest start time of an operation op_i. Since resource constraints cannot be figured out before the real scheduling, in most approaches, $ASAP(op_i)$ estimates the earliest *c-step* only considering the operation precedence constraints. Alternatively, the *As-Late-As-Possible* time of operation op_i, denoted by $ALAP(op_i)$ estimates the latest *c-step* when the operation op_i can be started. To achieve a higher HLS scheduling performance, it is required that the intervals $[ASAP(op_i), ALAP(op_i)]$ are as tight as possible. The following definition outlines an approach to calculate the initial $ASAP$ and $ALAP$ values for each operation op_i in a graph G, which is adopted by most HLS approaches [7].

Definition 3.1. *Let $G = (V, E)$ be a DFG of the HLS scheduling, and op_i $(i \in [1, N])$ be the operation of node $v_i \in V$. $ASAP_G(op_i)$ denotes the earliest time when the operation op_i can be dispatched, where*

$$ASAP_G(op_i) = CP_w(G_{pre}(op_i)) + 1 - delay(op_i).$$

$ALAP_G(op_i)$ indicates the latest time when the operation op_i can be dispatched. Let $le(S)$ be the length of a feasible schedule S. It can be calculated by

$$ALAP_G(op_i, le(S)) = le(S) - CP_w(G(op_i)).$$

From the definition, $le(S)$ has to be determined before calculating the $ALAP$ of operations. As an efficient method, the list scheduling algorithm [5], [6] can achieve such a feasible schedule quickly.

A *feasible scheduling* for a DFG tries to dispatch operations within calculated $[ASAP, ALAP]$ intervals under the operation dependence constraints posed by the DFG and the limited resources by the implementation requirement. As described in Definition 3.2, 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 best one until that iteration with the smallest length during the scheduling exploration. Among all possible schedules, the schedule with minimal length is regarded as the *global optimal schedule*.

Definition 3.2. *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 π_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 $\langle op_i, op_j \rangle \in E$ 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 π_j, $|\{op_i \mid type(op_i) = \pi_j \wedge ([S(op_i),\ S(op_i) + delay(op_i)] \bigcap [t, t]) \neq \emptyset\}| \leq num(\pi_j)$.*

Here condition 1 is 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 in Fig. 2, assume that the RCS problem only has one adder and one multiplier. In this case, the binary relation $\{(op_1, 1), (op_2, 3), (op_3, 4), (op_4, 6), (op_5, 6), (op_6, 8)\}$ is a feasible schedule with length 9. The binary relation $\{(op_1, 1), (op_2, 1), (op_3, 3), (op_4, 5), (op_5, 5), (op_6, 7)\}$ is one of the optimal schedules with length 8.

3.3 BULB Algorithm

Naïve enumeration of all the possible schedules for a DFG within the given $[ASAP, ALAP]$ intervals is extremely time-consuming. To reduce the unnecessary search time, the BULB approach [7] was proposed to efficiently prune fruitless search space in a branch-and-bound manner.

Algorithm 1 presents the details of the BULB algorithm. In this algorithm, we use S_{opt} to keep the best feasible schedule searched so far with the length ω. S indicates the current incomplete enumeration of operations. $globalLow$ denotes the lower-bound estimation of S_{opt}'s length. S_{opt} is initialized with a feasible schedule obtained using the list scheduling approach. Before an operation can be dispatched, both the precedence and resource constraints of the operation should be checked. We use the procedure $Prec(op_i)$ to check whether all the precedents of operation op_i are complete, and use the procedure $ResAvailable(step,\ type(op_i))$ to check whether the resources required by op_i are available at the given c-step. Note that the BULB approach can be easily extended to solve the scheduling with multiple kinds of constraints (e.g., area, energy, power) [21]. For example, if we want to incorporate power constraints in Fig. 2, during the searching we only need to put a checker for the overall power consumption estimation at the given c-step $step$ in the procedure $ResAvailable(step, type(op_i))$. If all specified constraints hold, steps 1 and 2 will schedule the undetermined operations in two different ways: i) $LBound(S)$ [16] dispatches the unscheduled operations without considering the resource constraints, thus it can be used to get the lower-bound length $lower$ for S; and ii) $UBound(S)$ obtains a feasible schedule for the undecided operations using the list scheduling method [5], [6], thus it can be used to estimate the upper-bound length $upper$ of S. If $upper$ is smaller than ω, it means that $UBound(S)$ is better than S_{opt}. Therefore, the S_{opt} and ω will be updated in steps 3 and 4. If the $upper$ equals $globalLow$, step 5 will terminate the whole BULB procedure. This is because an optimal schedule has been found. Otherwise, step 6 will update the $ALAP$ for each operation with a smaller ω. If $lower$ is smaller than ω, steps 7-10 will dispatch a new operation recursively. Otherwise, if $lower$ is not smaller than ω, the current incomplete schedule S can be pruned. Finally, the algorithm reports an optimal result.

Algorithm 1. BULB Algorithm

```
Input: i) An HLS DFG $D$ with resource constraints;
       ii) Operations $OP = \{op_1, \ldots, op_N\}$ in dispatching order;
       iii) A feasible schedule $S_{opt}$ of $D$ and its length $\omega$;
       iv) $S$, which stores the current incomplete schedule;
Output: An optimal schedule and its length for $D$
BULB($D, N, i, S, S_{opt}, \omega$) begin
    if $i \leq N$ then
        for $step = ASAP(op_i)$ to $ALAP(op_i)$ do
            if $Prec(op_i) \wedge ResAvailable(step, type(op_i))$ then
                1. $lower = le(LBound(S))$;
                2. $upper = le(UBound(S))$;
                if $upper < \omega$ then
                    3. $\omega = upper$;
                    4. $S_{opt} = UBound(S)$;
                    if $\omega == globalLow$ then
                        5. Return ($S_{opt}, \omega$);
                    end
                    6. $UpdateALAP()$;
                end
                if $lower < \omega$ then
                    7. $S(op_i) = step$;
                    8. $ResOccupy(step, type(op_i), delay(op_i))$;
                    9. $BULB(D, N, i+1, S, S_{opt}, \omega)$;
                    10. $ResRestore(step, type(op_i), delay(op_i))$;
                end
            end
        end
    end
    Return ($S_{opt}, \omega$).
end
```

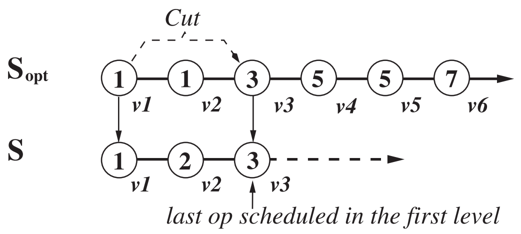

Fig. 3. A structure-aware pruning scenario.

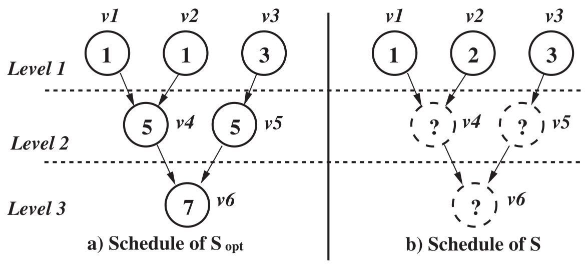

Fig. 4. Comparison of two schedules.

4 Parallel Structure-Aware Pruning

From Section 3.3, we can find that in B&B approach, the *lower*, *upper* and ω are the three main factors involved in pruning the useless search space. The pruning performance highly depends on the estimation algorithms for the factors *lower*, *upper* and the value of ω. However, the other useful information of the S_{opt} has not been fully investigated, especially when *upper* equals ω. For example, the structural scheduling information of S_{opt} (i.e., precedence constraint and assigned c-steps of operations) has not been exploited by the pruning strategies of B&B approach.

Based on the example shown in Figs. 2 and 3 shows the basic idea of the structure-aware pruning. In this figure, each node represents a functional operation, and the line indicates the dispatching order. Since the S_{opt} contains useful scheduling exploration information, by comparing partial operations (i.e., the operations on a *cut*, which will be introduced in Definition 4.1) between S_{opt} and S, the further search of optimal schedule based on S can be terminated under some condition. For example, let $\rho_1(G)$ of the design shown in Fig. 2 be the operation enumeration order, and assume that $S_{opt} = \{(op_1, 1), (op_2, 1), (op_3, 3), (op_4, 5), (op_5, 5), (op_6, 7)\}$ is the best schedule searched so far. In this case, $upper = 8$ and $globalLow = 6$. If only the operations op_1, op_2, op_3 have been dispatched in the current schedule S such that $S(op_1) = 1$, $S(op_2) = 2$, and $S(op_3) = 3$, the subsequent recursive search of the current schedule can be canceled. This is because that *cut* formed by operations op_1, op_2, and op_3 satisfies the level-bound pruning condition as proposed in Section 4.1. However, the traditional B&B approaches will continue the search, since $\omega = 8$ and the *upper* of the current search is also 8.

The above structural information is extremely useful when $\omega \in [lower, upper]$, since it extends the pruning capability of the traditional B&B approaches. The structure information can be used as a “fence” to prevent the deep fruitless recursive search in the B&B approaches. In other worlds, it can be used to prune a large part of the search space. Constructing more such fences will certainly reduce the overall search time. However, since the fence information is generated from the best schedule searched so far, the current version of sequential B&B approaches only maintains the best schedule as the fence, and updates it when encountering a better schedule during the search [24]. Resorting to the multicore platforms, multiple searches can be performed in different parts of the search space to construct fences. During the parallel search, the search space will be effectively pruned by the surrounding fences.

This section presents the details of our parallel structure-aware pruning approach. Section 4.1 proposes our level-bound pruning condition using the structural information of schedules. Next, Section 4.2 presents how to create multiple search tasks to explore different parts of the search space. Finally, Section 4.3 presents a framework that enables sharing of learning among different parallel tasks, which can significantly improve the pruning performance.

4.1 Level-Bound Pruning

Fig. 4 illustrates the basic idea of our level-bound pruning method based on the example shown in Fig. 2. Fig. 4a presents an optimal schedule searched so far of a DFG G, and Fig. 4b shows an incomplete schedule of G. Each operation in the figure is marked with a number indicating the dispatch time of the corresponding operation. Suppose that in S all the nodes of the first level have been scheduled and the other nodes have not been determined yet. To avoid fruitless search, traditional B&B approaches (e.g., BULB) only compare $le(S_{opt})$ with both the *lower* and *upper* of S for pruning. Note that, in Fig. 4, the dispatching time of all S_{opt}'s operations in the first level is no worse 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* $EG = \{e_1, \ldots, e_n\}$ *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^r, *there exists a path from the source node to some* $e \in EG$.

For a given cut, $Pre(CUT)$ *denotes its precedent node set;* $Imme(CUT)$ *denotes the adjacent input nodes of the edges in the cut; and* $Post(CUT)$ *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* Xth-level cut *of G. Assuming that* cut_X *is the* Xth-level cut *of G,* $Imme(cut_X)$ *is the* Xth 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 Fig. 2, $\{e_4, e_5\}$ is the second level cut of G. In G, $Pre(\{e_4, e_5\}) = \{v_1, v_2, v_3, v_4, v_5\}$, $Imme(\{e_4, e_5\}) = \{v_4, v_5\}$ and $Post(\{e_4, e_5\}) = \{v_6\}$. The second complete level of G is $\{v_4, v_5\}$.

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 $Pre(CUT)$, our level-bound pruning method can be used to determine whether some incomplete schedule can be abandoned 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 pruning method (Theorem 4.1), we define the *local optimal schedule* which is 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 $OP_{sch} = \{op_{i_1}, op_{i_2}, \ldots, op_{i_m}\}$ is a subset of scheduled operations. A* local optimal schedule *$S_{l,OP_{sch}}$ is a complete schedule which has fixed values for OP_{sch} and the optimal scheduling for the unscheduled operations (i.e., $OP \backslash OP_{sch}$). That means $S_{l,OP_{sch}}$ is a global optimal schedule in the search space $\Pi_{k=1}^{m}[S(op_{i_k}), S(op_{i_k})] \times \Pi_{op_j \in OP \backslash OP_{sch}}[ASAP(op_j), ALAP(op_j)]$.*

The local optimal schedule $S_{l,OP_{sch}}$ denotes that when a set of operations OP_{sch} are assigned with fixed *c-steps* in advance, it searches for the "global" optimal solution from the rest of the nodes. Obviously, $S_{l,OP_{sch}}$ 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 $S_{l_1,Pre(CUT)}$ and $S_{l_2,Pre(CUT)}$ be two different local optimal schedules based on CUT, and Imme (CUT) be an operation set $\{op_{i_1}, op_{i_2}, \ldots, op_{i_k}\}$. Assume that $\omega_1 = le(S_{l_1,Pre(CUT)})$ and $\omega_2 = le(S_{l_2,Pre(CUT)})$. We can conclude that $\bigwedge_{1\le j\le k}(S_1(op_{i_j}) \le S_2(op_{i_j}))$ && $\bigvee_{1\le j\le k}(S_{opt}(op_{i_j})! = S(op_{i_j})) \Longrightarrow \omega_1 \le \omega_2$.*

Proof. See the details in [24]. □

In RCS, dynamically deciding whether an edge set is a cut is time-consuming. Since the complete level structures introduced in Definition 4.1 imply the cut information, our approach adopts them for the *level-bound pruning* checking. Instead of comparing all scheduled operations between two schedules, in our level-bound pruning method, the comparison is triggered only when all operations in one complete level have been scheduled.

Definition 4.3. *For a given DFG G, S is an incomplete schedule and S_{opt} is the best schedule so far. Let OP_k be the operation set of the kth complete level. Assume that all the operations in set OP_k have been scheduled. Based on Theorem 4.1, the* level-bound pruning *can be enabled when the following conditions hold.*

1) $\forall op_i,\ op_i \in OP_k \rightarrow S(op_i) > 0$;
2) $\forall op_i,\ op_i \in OP_k \rightarrow S_{opt}(op_i) \le S(op_i)$;
3) $\exists op_i,\ op_i \in OP_k \rightarrow S_{opt}(op_i) < S(op_i)$.

In Definition 4.3, the condition 1 indicates that all the operations in kth complete level are dispatched. The condition 2 means that all the operations of S in kth complete level do not have *c-steps* smaller than the corresponding operations of S_{opt}. The condition 3 denotes that at least one operation in kth complete level has worse *c-step* than the corresponding operation in S_{opt}. The above conditions indicate that the comparison only needs to check all the operations in level k. For example in Fig. 4, when the last operation in the first level is scheduled, the level-bound checking will be invoked. Since all operations of S in the first complete level have larger or equal *c-steps* than the ones in S_{opt}, the level-bound pruning can be enabled in this case.

For a given DFG, assume that the ith complete level contains k operations, i.e., $OP_k = \{op_{i_1}, op_{i_2}, \ldots, op_{i_k}\}$, and all the operations in the ith complete level have been scheduled. Let S_{opt} be the optimal schedule so far, and let S be the current schedule. When the *level-bound pruning condition* holds for the ith complete level, there exists an operation op_{i_j} $(1 \le j \le k)$ such that $S(op_{i_j}) > S_{opt}(op_{i_j})$. In other words, the recursive procedure of B&B approach backtracks at least to the operation op_{i_j} with value $S_{opt}(op_{i_j}) + 1$. Let OP be the operation set which contains both OP_k and its precedent operations, and their *c-steps* are the same as the schedule S. According to Theorem 4.1, the local optimal schedule $S_{l,OP}$ cannot be better than S_{opt}. Since S_{opt} keeps the up-to-date best schedule along the B&B approach recursion, the current schedule S can be pruned.

Algorithm 2. Our Level-Bound Pruning Algorithm

```
Input: i) $S_{opt}$, the best schedule searched so far
       ii) $S$, the current incomplete schedule
       iii) $op_i$, the last dispatched operation of $S$
Output: Whether $S_{opt}$ outperforms $S$.
LevelBound($S, S_{opt}, op_i$) begin
    if i $\neq L_h(op_i)$ then
        1. return $false$;
    end
    2. $OP = \{op_{i_1}, \ldots, op_{i_k}\} = (level(op_i))$th complete level;
    if $\bigwedge_{1\le j\le k}(S_{opt}(op_{i_j}) \le S(op_{i_j}))$
    && $\bigvee_{1\le j\le k}(S_{opt}(op_{i_j}) \neq S(op_{i_j}))$ then
        3. return $true$;
    end
    4. return $false$;
end
```

Algorithm 2 describes the details of our level-bound pruning approach. Step 1 checks whether all the operations in current level have been scheduled. Step 2 identifies the $(level(op_i))$th complete level. Step 3 asserts that S_{opt} outperforms S based 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 $(level(op_i))$th complete level are dispatched.

4.2 Parallel Level-Bound Pruning

According to Definition 4.3, our level bound pruning approach enables the construction of fences to avoid fruitless search. In fact, each feasible schedule can be utilized as a fence to restrict the fruitless search. Since we focus on the pruning based on the operation scheduling information of some level (i.e., level-bound pruning), the fence can be decomposed into a set of sub-fences across different levels. When different parallel search tasks are running collaboratively for an RCS problem, multiple fences will be generated. Since each fence corresponds to one feasible schedule, if such information can be shared among parallel search tasks, the search space will be restricted further by different fences. Consequently, the overall search time can be

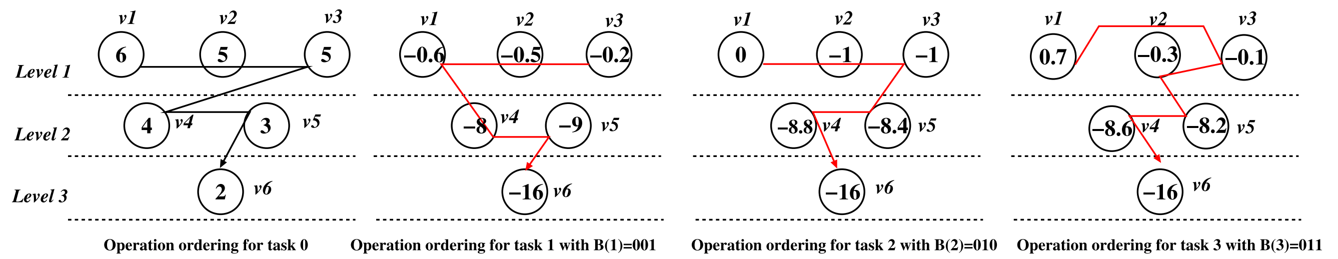
Level 1
Level 2
Level 3
v1
v2
v3
v4
v5
v6
6
5
5
4
3
2
–0.6
–0.5
–0.2
–8
–9
–16
0
–1
–1
–8.8
–8.4
–16
0.7
–0.3
–0.1
–8.6
–8.2
–16
Operation ordering for task 0
Operation ordering for task 1 with B(1)=001
Operation ordering for task 2 with B(2)=010
Operation ordering for task 3 with B(3)=011

Fig. 5. An illustration of different ordering for different search tasks.

reduced. For example in Fig. 2, assume that the length of an initial feasible schedule is 8, and there are two search tasks t_1 and t_2 using the operation order $\rho'_1 = < v_1, v_2, v_3, v_4, v_5, v_6 >$ and $\rho'_2 = < v_1, v_3, v_2, v_4, v_5, v_6 >$, respectively. When t_1 finds a better schedule S_1, it will create a sub-fence at first level as $\$(v_1, 1), (v_2, 1), (v_3, 3)\$$. Similarly, t_2 will create a sub-fence $\$(v_1, 3), (v_2, 1), (v_3, 1)\$$. When the search of t_1 advances to the enumeration $\{(v_1, 3), (v_2, 1), (v_3, 1), (v_4, ?), (v_5, ?), (v_6, ?)\}$ where '?' denotes undetermined c-steps, t_1 can be safely terminated, since the following search will be blocked by the fence set by t_2. Alternatively, t_2 will also benefit from the fence set by t_1. The search time could be further reduced if more high-quality fences are constructed to restrict the search space of each search task.

By instrumenting fences efficiently in the search space of collaborative search tasks, our approach can effectively avoid the deep recursive search, while increasing the chance to find an optimal schedule earlier than traditional B&B approaches. Based on this observation, to quickly find an optimal schedule with parallel search tasks, we need to address two following major issues: i) how to efficiently distribute parallel tasks over search space, and ii) how to quickly detect whether the enumeration of current incomplete schedule can be terminated? The following sections describe the details.

4.2.1 Derivation of Parallel Search Tasks

The parallel B&B style searching is quite different from the sequential B&B style search. In parallel search, if all search tasks have the same operation ordering and the same search space, parallel search is not beneficial using multi-cores. However, if different parallel search can have different operation enumeration order, the search will become "multi-directional" and the search time can be reduced drastically. Generally, there are two kinds of multi-directional search. First, the length of the optimal schedule equals to the lower bound estimation (i.e., $globalLow$). During parallel search, if one optimal schedule is confirmed by some search task, the overall search can be terminated. In other words, the search time is determined by shortest distance between parallel tasks and their nearby optimal schedules. Apparently, parallel search is suitable for quick identification of an optimal schedule. Secondly, the length of the optimal schedule is larger than the lower bound estimation. Each search task needs to explore the entire search space. However, if fences can be used and shared among search tasks, some part of the search space will be blocked by the generated fences among search tasks. Therefore, the overall search time can be reduced.

For recursive B&B style searching methods (e.g., BULB), it is required to check both the status of predecessor operations (using $Prec$ function in Algorithm 1). In other words, the operation precedence should be considered during the design of the operation enumeration order. If the operation precedence is violated, there will be a large number of long-distance backtracks to cancel the effect of the early wrong decisions. In this case, the B&B style searching performance can be degraded. Besides the long-distance backtrack, stuck-at-local-search problem [25] also needs to be avoided in the B&B style search. Therefore, the dispatching order of the enumerated operations is very important. If there are more different fences in the top levels (i.e., levels with small indices), the chance of local search will be reduced due to the early interception by sub-fences instrumented in top levels. Therefore, we need to create a proper number of sub-fences for the top levels to intercept potential unfruitful local search.

Based on the above observation, we design an operation enumeration ordering scheme for parallel search tasks. Assume that we have n search tasks (i.e., $t_0, \ldots, t_{n-1}$) in total. To guarantee better performance than the sequential approach, we set the operation ordering of task t_0 to be the same as the heuristic adopted in the traditional B&B approaches. For the remaining parallel tasks, to enable generation of multiple effective sub-fences in top levels, the operations in top levels should be shuffled as much as possible while keeping the operation precedence relation. In our approach, we conduct the guided-random shuffle of operations by tuning the length of weighted critical paths of operations which are sorted by the BULB approach. Let $B(i)$ be the binary number of an integer i, and $B(i)[j]$ $(i, j > 0)$ denotes the jth least significant digit of $B(i)$. For the ith parallel search task t_i, our approach uses the following strategy to shuffle the operations.

1) If $B(i)[j]$ equals to 1, then all the operations in the jth level of the t_i's DFG are shuffled in guided-random fashion. For each operation op in the jth level, its $CP_w(G(op))$ will be increased by $\frac{rand() \bmod N}{N+1} - CP_w(G) \times j$, where $rand()$ returns a random integer and N denotes the number of operations in the DFG.
2) If $B(i)[j]$ equals to 0, then the lengths of the weighted critical paths of all operations in the jth level will be decreased by $CP_w(G) \times j$.

Fig. 5 shows an example of four parallel tasks with different ordering for the design shown in Fig. 2. The leftmost figure shows the ordering using traditional B&B approach indicated by a solid arrow line. We mark each operation with a number, which indicates the tuned length of weighted critical path of that operation. By using our

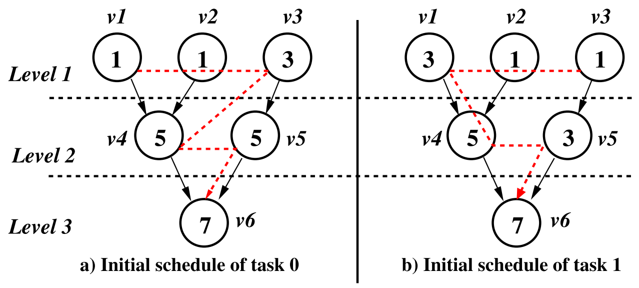

Fig. 6. An example of RCS searching with shared fences.

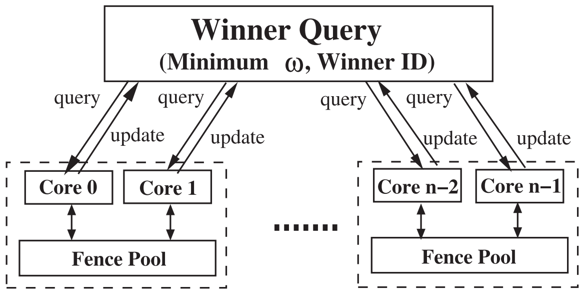

Fig. 7. Our RCS framework with bound and fence sharing.

enumeration strategy, the tasks whose ID is larger than 0 can dispatch the operations level by level using our proposed approach. The remaining figures show the operation enumeration order for tasks 1-3, respectively. Assume that the level of operation op is l, by reducing the length of critical paths of operations by $CP_w(G) \times l$, all the operations will be enumerated level by level. The notation $\frac{rand()\ mod\ N}{N+1}$ increases the length of critical paths of operations by a random value within $(-1, 1)$, which indicates the shuffle of operations in that level. As an example, for task 2, since B(2) equals $(010)_2$, the operations in the second level needs to be shuffled. For levels 1-3, the lengths of weighted critical paths for each operation are reduced by -6×1, -6×2 and -6×3 respectively, which impose the enumeration priority based on the level information of operations. While more tasks are involved in the parallel search, due to the shuffle and level-based priority of operations, a wide variety of sub-fences for top levels will be generated to prevent the deep recursive search.

4.2.2 Collaboration among Parallel Tasks

During the multi-directional search, if parallel tasks explore search space independently, when the length of optimal schedules is larger than the lower bound estimation, the RCS time will be equal to the time of the task which first finishes the scan of the search space. Based on the observation in [25], if the upper bound of best schedule searched so far can be shared among parallel tasks, the search space can be compacted on-the-fly. Therefore the overall RCS time can be significantly reduced. Since the sharing of bound is a common technique for parallel RCS searching, it can be easily combined with other optimization approaches.

Due to different operation ordering, parallel tasks may keep different best schedules searched so far during the search. Such information can be used as a fence whose level bound information can be used as sub-fences to avoid the unfruitful deep search. In fact, such fence information can be shared among parallel tasks to further avoid unfruitful search. Therefore, besides sharing of bound information, our approach also supports the sharing of the fence information.

Fig. 6 shows an example of sharing fences between tasks using the example shown in Fig. 2. We assume that there are only two search tasks with different operation ordering as shown in Fig. 5 and both tasks share their obtained fences. In this figure, due to different operation ordering indicated by the dashed arrow lines, we can get two different initial feasible schedules for the same RCS problem. It is important to note that these two schedules can be used as shared fences (i.e., f_1 and f_2) to benefit both the following RCS search. For example, in Fig. 6a, the current search will backtrack to operation v_1 based on its local fence f_1 safely without finding any better schedule. This is because the operation dispatching time of v_5, v_4, v_3 and v_2 cannot be smaller than 5, 5, 3 and 1, respectively. When v_1 is dispatched at c-step 2, the scheduling will fail because the length of $S_{l,\{v_1\}}$ is no smaller than the length of the initial schedule. Then v_1 will be dispatched at c-step 3. Based on the fence f_2, all the following enumeration will be fruitless. Therefore, the search of task 0 will terminate and assert that its initial schedule is one optimal result. Consequently, the search time of task 0 can be reduced based on the shared fences.

To effectively share bound and fence information, we developed a cooperation framework as shown in Fig. 7 that supports the *multi-level bound pruning* as well as *minimum ω synchronization* [25] among parallel search tasks. In this framework, we partitioned the parallel search tasks into clusters according to their task IDs. For example in Fig. 7, we cluster 2 tasks with adjacent IDs. The reason of choosing adjacent IDs is that they share the same ordering of lower level operations but different ordering of top level operations. Therefore, the shared fences have different dispatching time for top level operations, which can be used together to effectively avoid the deep recursive search. Each cluster has a fence pool to keep the shared fences among parallel search tasks in the same cluster. The fence pool is implemented by the data structure *vector*. A parallel task will periodically check the update of its shared cluster fence pool. If some new fences are shared, each task in the same cluster will update its local fence pool to include the newly generated fences. To enable the collaboration among all clusters, we construct a global data structure called *winner query* which enables the query and update of the length (i.e., minimum ω) and the ID of the task that has found the best schedule so far. Each parallel task can update the winner information if it finds a better schedule than the current winner. It can also periodically query the winner information, and update its local ω if the schedule of the winner is better. The update of the local ω enables the shrinking of the search space for the parallel task. Therefore, our framework can drastically reduce the parallel search time. The whole framework will terminate with the optimal schedule if any of the parallel search task stops.

Generally, it is not wise to keep a large set of fences in the local fence pool of a parallel task. This is because too many fences will cause a large number of invocations of level bound pruning, which is time-consuming. To control the number of fences, we adopt the following strategy in each cluster for fence sharing. At the beginning of parallel search, each task in a cluster will only figure out one feasible schedule and save it in its local fence pool for the level bound checking. Initially, the cluster fence pool is empty. During the search, when one task in the cluster finds a better schedule than its S_{opt}, it will update the cluster fence pool with this schedule. This newly added schedule will be propagated to other tasks in the same cluster to enable the multi-level bound pruning at top levels.

Algorithm 3. Update/Query Operations in Our Framework

```
UpdateWinner($S, rank$) begin
    if $len(S) < global\_min$ then
        framework_mutex.$lock()$;
        $global\_min = len(S)$;
        $global\_winner = rank$;
        framework_mutex.$unlock()$;
    end
end
QueryWinner() begin
    return $(global\_winner, global\_min)$;
end
UpdateFencePool($S, rank$) begin
    cluster_mutex.$lock()$;
    fencePoolSet[rank/clusterSZ].pushback(S);
    cluster_mutex.$unlock()$;
end
QueryNewFences(lfp, rank) begin
    fsize = fencePoolSet[rank/clusterSZ].size();
    while $fpmark < fsize$ do
        lfp.pushback(fencePoolSet[rank/clusterSZ][fpmark]);
        fpmark++;
    end
end
```

Algorithm 3 presents the details of the update and query operations for the information of global minimum bound and the fence information. We use an array $fencePoolSet$ to denote the set of fence pools. Two integers $global\_winner$ and $global\_min$ indicate the ID and weighted length of the best schedule searched so far. Since our framework is implemented using MPI [17], all tasks are implemented using processes, and all the shared data structures are constructed in shared memory which can be accessed by parallel processes. To achieve minimum ω synchronization, all the search tasks always monitor the change of the global minimum ω value by using the procedures *QueryWinner*, and dynamically shrink the search space accordingly. When one task finds a schedule with the length smaller than or equal to $global\_min$, it will try to update the fence pool based on the aforementioned conditions by invoking the *UpdateWinner* procedure. To avoid the race condition when updating the $winner\ query$, we create a global lock for all the search tasks. In each cluster, a task can update and query the shared fence pool for the newly updated fences using the functions *UpdateFencePool* and *QueryNewFences*. For each task, *lfp* indicates the local fence pool of the task. *fpmark* is a local iterator that keeps the index of the last index of monitored fences in the cluster fence pool. If there exists newly added fences, such iterator will be increased while the shared fences are included in *lfp*.

Algorithm 4. Multi-Level Bound Pruning

```
Input: i) $S$, which is an incomplete schedule;
       ii) $op_i$ which is an operation dispatched;
       iii) $rank$, which indicates the rank of current process;
Output: Whether to enumerate the remaining operations in S
MLevelPrune($S, op_i, rank$) begin
    QueryNewFences(lfp, rank);
    for each fence $S_F$ in lfp of $S$ do
        if LevelBound($S, S_F, op_i$) then
            return TRUE;
        end
    end
    return FALSE;
end
```

Algorithm 4 describes the details of our multi-level bound pruning approach based on the multiple fences in the local fence pool. When the operation op_i is the last dispatched operation in a level, *MLevelPrune* will first update its local fence pool *lfp* and then invoke the function *LevelBound* to check each fences in *lfp*. If some sub-fence works, the current incomplete enumeration will be terminated.

4.3 Our Parallel B&B Approach

Compared to traditional B&B approach which only uses ω value for pruning, our parallel RCS pruning approach utilizes the shared bound of multi-directional search and the structure information of schedules saved in shared fence pools. Our method can be effectively combined with traditional B&B style RCS approach to explore further pruning chances, which can improve the RCS performance drastically.

4.3.1 Implementation of Parallel RCS Search Tasks

Algorithm 5 presents the details of the implementation of a parallel RCS search task in our proposed framework as shown in Fig. 7. Step 1 initializes the variable $op_i.sucTimes$ which indicates the times of the successful occupation of the resource by op_i. Step 2 queries the current minimum length of all possible schedules. If op_i is the final unscheduled operation in $level(op_i)$ and $MLevelPrune(S, op_i, rank)$ returns $true$, it means that the current schedule stored in S is worse than some schedules in the fence pool. 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_i. If $MLevelPrune(S, op_i, rank)$ returns false, the operation op_i will be scheduled. Step 4 indicates that the operation successfully gets the intended resource again. Steps 5 and 6 calculate the $lower$ and $upper$ for the current schedule. If $upper$ is smaller than ω, then ω and S_{opt} will be updated in steps 7 and 8. Changing ω value will trigger the checking of early termination condition (i.e., achieving a schedule whose length equals to $GlobalLow$) in step 9. Step 10 updates the fence pool using the achieved schedule in step 8. If $lower$ is smaller than ω, current operation will be scheduled. Step 11 assigns a *c-step* to operation op_i. Step 12 reserves resources required by the operation. Then op_{i+1} is processed recursively in step 13. When the search backtracks, the resource occupied by op_i is released in step 14. Steps 15 and 16 deal with the backtracks caused by the level-bound checking. When $op_i.sucTimes$

equals to 1, it means that the current *step* is the smallest for op_i 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_i can be stopped. Finally, the algorithm returns a global optimal schedule and its length. Note that parallel search tasks compete with each other to find an optimal schedule. If one task finishes first, the other tasks will be terminated.

Algorithm 5. Parallel RCS Search Task Implementation using Structure-Aware B&B Pruning

```
Input: i) $D$, which is an HLS DFG with resource constraints;
       ii) $OP = \{op_1, \ldots, op_N\}$ in dispatching order;
       iii) $S_{opt}$, which is a feasible schedule of length $\omega$;
       iv) $S$, which stores the current incomplete schedule;
       v) $rank$, which indicates the rank of current process;
Output: An optimal HLS schedule and its length
ParaRCS($D$, $OP$, $i$, $N$, $S_{opt}$, S, $\omega$, $rank$) begin
    if $i \leq N$ then
        1. $op_i.sucTimes = 0$;
        if $\omega > QueryWinner().second$ then
            $\omega = QueryWinner().second$;
            $UpdateALAP()$;
        end
        for $step = ASAP(op_i)$ to $ALAP(op_i)$ do
            if $op_i == L_h(op_i) \wedge MLevelPrune(S, op_i, rank)$ then
            2. jump = $true$;
            3. return ($S_{opt}$, $\omega$);
        end
        if $Prec(op_i) \wedge ResAvailable(step, type(op_i))$ then
            4. $op_i.sucTimes++$;
            5. $lower = le(LBound(S))$;
            6. $upper = le(UBound(S))$;
        if $upper < \omega$ then
            7. $\omega = upper$;
            8. $S_{opt} = ListScheduling(OP, op_i)$;
            if $\omega == GlobalLow(D)$ then
                9. Report ($S_{opt}$, $\omega$);
                MPI_Abort(MPI_COMM_WRD, -1);
            end
            10. $UpdateFencePool(S_{opt}, rank)$;
        end
        if $lower < \omega$ then
            /* Dispatch the current operation */
            11. $S(op_i) = step$;
            12. ResOccupy(step, type($op_i$), delay($op_i$));
            13. ParaRCS(D, OP, i+1,N, $S_{opt}$, S, $\omega$, rank);
            14. ResRestore(step, type($op_i$), delay($op_i$));
            if jump then
                if $op_i.sucTimes$==1 then
                    15. return($S_{opt}$, $\omega$);
                else
                    16. jump = $false$;
                end
            end
        end
    end
  end
   end
   return ($S_{opt}$, $\omega$);
end
```

4.3.2 Implementation of Our Parallel B&B Method

Algorithm 6 describes our parallel B&B RCS approach. Since we adopt MPI [17] as our parallel search programming language library, Algorithm 6 shows the skeleton of our approach in the format of MPI. We use S_{opt} and S to indicate the best schedule search so far and the current incomplete schedule, respectively. ω is the length of S_{opt}. In this algorithm, step 1 parses the input file to figure out the DFG of a given RCS problem as well as the user input constraints for the problem. Step 2 sorts all the operations of the DFG based on the length of their weighted critical paths. Step 3 encodes the task index into binary format. If the current task has an index of 0, then it will construct the shared information (i.e., $fencePoolSet$ and winner information) in a shared memory which can be accessed by parallel tasks. In our approach, if the task index is 0, it will have the same operation order as the BULB approach. Otherwise, as shown in step 5, the operation order of other parallel tasks will be shuffled based on the level information of the DFG as well as the binary number of the task index. Step 6 searches for an initial feasible schedule for the task, and such schedule will be put into the local fence pool *lfp*. It is important to note that this initial schedule will not be inserted into the cluster fence pools in our approach. In steps 8 and 9, *globalLow* and ω indicate the lower and upper bound estimation of the optimal schedule, respectively. Based on the agreement of all parallel tasks in step 10, the smallest initial upper bound estimation among all tasks (i.e., ω) will be used to shrink the initial search space of all parallel tasks described in step 11. If any task finishes the search, one optimal schedule as well as its length will be reported in step 12, and the whole parallel search will be terminated.

Algorithm 6. Our Parallel B&B RCS Algorithm

```
Output: An optimal schedule and its length for $D$
main(argc, argv) begin
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WRD, &procnum);
  MPI_Comm_rank(MPI_COMM_WRD, &rank);
  1. $D = ParseDFGFromFile()$;
  2. $OP = \{op_1, \ldots, op_N\} = SortOperations(D)$;
  3. $code = Binary(rank, \lceil log_2 procnum \rceil)$;
  if rank==0 then
        4. Init(fencePoolSet,global_winner, global_min);
  else
        5. $OP_{rank} = Shuffle(D, OP, code)$;
  end
  6. $S = S_{opt} = InitialFeasibleSch(D)$;
  7. lfp.pushback(S);
  8. $globalLow = le(LBound(S))$;
  9. $\omega' = le(S_{opt})$;
  10. MPI_Allreduce(&$\omega'$, &$\omega$, 1, MPI_INT,
              MPI_MIN, MPI_COMM_WRD);
  11. $ParaRCS(D, OP_{rank}, 1, N, S_{opt}, S, \omega, rank)$;
  12. Report ($S_{opt}$, $\omega$, $rank$);
  MPI_Abort(MPI_COMM_WRD, -1);
end
```

5 Experimental Results

To evaluate the effectiveness of our approach, we conducted various experiments with different kinds of resource

TABLE 1
The Settings of the Functional Units

Functional Unit	Operation Class	Delay (unit)	Power (unit)	Energy (unit)	Area (unit)
ADD/SUB	+/-	1	10	10	10
MUL/PMUL	$\times/*$	2	20	40	40
DIV	/	2	20	40	40
MEM	LD, STR	1	15	15	20
Shift	$<<, >>$	1	10	10	5
Other	...	1	10	10	10

constraints. We collected the DOT files of following benchmarks from the *MediaBench* benchmark [18], which is a standard DSP benchmark suite: i) *ARFilter* with 28 nodes and 30 edges, ii) *Cosine 1* with 66 nodes and 76 edges, iii) *Collapse* with 56 nodes and 73 edges, iv) *EWF* with 34 nodes and 47 edges, and v) *Feedback* with 53 nodes and 50 edges. We also used the benchmark *FDCT* with 42 nodes and 52 edges from [23]. To enable the comparison with the in-place approach [8], we unfolded the *EWF* design for two times (i.e., *EWF2*) and three times (i.e., *EWF3*), respectively. We developed a DFG representation to unify the benchmark formats from different sources, which can be used as a part of the input (together with the resource constraints and parallel search settings) of our approach. For comparison, we also generated *ILP models* for RCS using IBM ILOG CPLEX CP Optimizer [19], which adopts the *branch-and-cut* method [28] for efficient searching.

By using the C programming language, we implemented the BULB method [7], the hybrid parallel pruning approach [25] and our approach [20] that incorporates the proposed parallel structure-aware pruning techniques. To enable the parallel searching, we use the MPI library [17] for constructing parallel processes. To avoid the overload of the level bound checking, we restrict the number of tasks in a cluster to be 4 in the experiment. All the experimental results were obtained on a Linux sever with 96 Intel Xeon 2.4 GHz cores and 1 TB RAM. Since our approach shuffles the operations in guided-random fashion in the same level, different operation ordering can easily cause the search time variation. Therefore, we run each RCS problem instance five times and record the mean value for the comparison. In this experiment, we consider functional unit constraints as well as power and area constraints during the scheduling. Table 1 lists the settings for various types of operations used in the experiment. It is important to note that by slightly modifying the *ResAvailable* function our level-bound pruning approach can be applied on the pipelined designs directly, since it only needs to check the dispatch time of the first sub-operations. In this experiment, we assume that the pipelined multipliers (i.e., *PMUL*) have two stages and each stage needs one c-step.

5.1 Scheduling under Functional Unit Constraints

Table 2 presents the experimental results carried out with different functional unit constraints on the five benchmarks. The first column of the table indicates the name of the benchmarks. The second column presents the functional unit constraint for the design. We use notation "x, y" to denote that only x adders and y non-pipelined multipliers are adopted for the RCS. For example, in the first row of $ARFilter$ design, "1, 3" denotes that only one adder and three non-pipelined multipliers are used for the given design. Due to the space limit, we did not provide the number of other functional unit types. The third column gives the optimal *c-steps* for the whole design under the specified resource constraints. The fourth column presents the ILP solving time using the CPLEX CP Optimizer. Among all benchmark items, only one of them (i.e., *ARFilter* design

TABLE 2
Scheduling Results under Functional Unit Constraints

Bench-	# of	# of	CPLEX	BULB	LEVEL	ML1	MD8	Hybrid8[25]/+L	ML8	# of	# of	BULB	ML8
Mark	$+, \times$	csteps	(sec.)	[7] (sec.)	[24] (sec.)	(sec.)	(sec.)	(sec.)	(sec.)	$+, *$	csteps	[7] (sec.)	(sec.)
ARFilter	1, 3	16	NA	0.31	0.15	0.13	0.29	0.39/0.18	0.13	1, 1	19	< 0.01	< 0.01
	1, 4	16	NA	0.78	0.25	0.26	0.75	1.01/0.28	0.26	2, 1	19	< 0.01	< 0.01
	1, 5	16	NA	0.77	0.24	0.26	0.73	1.03/0.27	0.26	2, 2	13	< 0.01	< 0.01
	2, 3	15	2.32	0.01	0.01	0.01	0.01	< 0.01/ < 0.01	< 0.01	3, 1	19	< 0.01	< 0.01
Collapse	2, 1	22	NA	NA	NA	118.81	38.14	0.02/0.02	38.16	2, 1	NA	NA	NA
	2, 2	21	NA	NA	NA	NA	NA	< 0.01/ < 0.01	NA	2, 2	NA	NA	NA
Cosine	1, 2	28	NA	107.43	29.68	19.83	< 0.01	< 0.01/ < 0.01	< 0.01	1, 2	28	< 0.01	< 0.01
	2, 2	20	NA	622.83	29.21	55.92	0.80	34.54/3.12	0.52	2, 2	15	34.98	0.61
	3, 3	16	NA	0.02	0.01	< 0.01	< 0.01	0.01/0.03	< 0.01	3, 3	12	0.11	0.01
FDCT	1, 2	26	NA	36.91	26.63	19.94	< 0.01	< 0.01/ < 0.01	< 0.01	1, 1	26	558.69	< 0.01
	2, 2	18	NA	201.59	90.56	40.78	2.12	13.99/0.19	1.67	1, 2	26	< 0.01	< 0.01
	2, 3	14	NA	19.80	21.16	6.04	1.37	2.85/2.18	0.53	2, 2	13	9.68	< 0.01
	2, 4	13	NA	4.07	5.94	2.01	0.14	0.87/0.11	0.13	2, 3	13	< 0.01	< 0.01
	2, 5	13	NA	0.92	0.60	0.52	< 0.01	0.04/0.04	< 0.01	3, 2	12	0.11	0.02
	3, 4	11	NA	0.55	0.48	0.43	0.15	0.67/0.64	0.07	3, 3	10	0.07	< 0.01
	4, 4	11	NA	0.12	0.03	0.03	< 0.01	0.23/0.08	< 0.01	4, 4	9	< 0.01	< 0.01
Feedback	4, 4	13	NA	154.18	155.74	127.22	1.14	3.82/5.30	0.81	4, 4	13	NA	3.67
	4, 5	13	NA	NA	NA	NA	3.62	4.50/3.72	1.13	4, 5	13	NA	7.70
	5, 5	13	NA	4.87	4.92	4.02	0.06	1.51/1.59	0.04	5, 5	13	22.62	0.07

† *NA means that the value is not achieved within time limit (3,600 seconds). "$\times$" and "$*$" indicate non-pipelined and pipelined multipliers, respectively.*

with two adders and three multipliers) can achieve the optimal result within the time limit (i.e., 3,600 seconds). The fifth column presents the scheduling timing using the BULB approach [7]. The sixth column shows the results using our structure-aware approach presented in [24] with a single core. It is important to note that BULB approach only compares the structure of incomplete schedules with the structure of the optimal schedule searched so far. In other words, it only keeps one fence during the search. Unlike [24], the seventh column gives the results using our structure-aware pruning approach with a single core and multiple fences. Here, ML stands for the checking with multi-level pruning. The eighth column adopts the parallel multi-directional search with eight cores. However, it does not instrument any fences during the search. To show the effectiveness of our proposed techniques, the ninth column shows the comparison between the hybrid approach [25] (running with eight cores) and its modified version, where each sub-task utilizes fences to conduct the efficient pruning. The tenth column presents the results using our parallel structure-aware pruning with eight cores. Since our approach can be directly applied on the pipelined designs, we also conduct the experiment on the same benchmark with different pipelined functional units. The last four columns present the pipelined functional unit constraints, *c-steps* of optimal schedules, RCS time using the BULB approach [7], and RCS time using our parallel structure-aware approach (with eight cores), respectively.

To show the efficacy of our multi-level bound pruning and multi-directional search separately, we present the ML (multi-directional search with fences) and MD (multi-directional search without fences) results in Table 2. For the designs with non-pipelined constraints (in columns 2-10), we can find that our approaches (i.e., $ML1$, $MD8$, $ML8$) can not only outperform the state-of-the-art ILP solver with the branch-and-cut heuristic, but also can drastically improve the RCS performance using the B&B style searching. As a sequential pruning method, $ML1$ outperforms both BULB and $LEVEL$. It is important to note that, for the *ARFilter* design, the performance of $ML1$ is a little bit worse than the performance of $LEVEL$. This is because $ML1$ has more fences for level bound checking. Since the overall search time is small, the overhead of multiple fence checking time will occupy a large portion of the overall search time. *MD8* is a parallel pruning approach with multi-directional search. It shares the bound information among sub-tasks without instrumenting any fences. For all the benchmarks except the *ARFilter* design, such parallel searching method outperforms all the sequential searching approaches listed in the table. The reason why *ARFilter* cannot benefit from the *MD8* search is that the length of optimal schedule equals to the upper bound estimation for *ARFilter* design with different constraints. Without fences, all the sub-tasks will search all the search space to ensure this fact. Therefore, $MD8$ cannot benefit in the case of *ARFilter*. $Hybrid8$ [25] is based on the search space partitioning and upper bound speculation. Based on the assumption that optimal results are evenly scattered in the search space, it is promising to quickly find one optimal result by some sub-search task. However, if the length of the optimal schedule equals to the upper bound estimation (see the example of ARFilter) or the partitioned

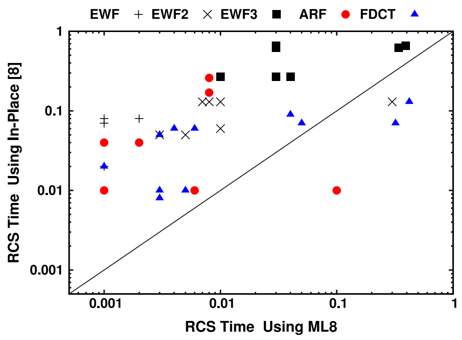

Fig. 8. Comparison with the in-place approach [8].

search space has no optimal schedules, then *Hybrid*8 will be easily stuck at the local search. Our structure-aware technique can be incorporated in *Hybrid*8 to promote its searching performance. We developed the *Hybrid*8+*L* approach, which enables the level bound checking for each sub-task. However, in order to avoid the unnecessary overload of level bound checking for lightweight sub-tasks, we do not share the fences among sub-tasks. For each sub-task in *Hybrid*8+*L* approach, it only checks the level bound information in the same way as the $LEVEL$ method [24]. By our observation, the structure-aware pruning can help *Hybrid*8 to reduce the overall searching time drastically. For the design *FDCT* under the constraint of two adders and two multipliers, $Hybrid8$ method needs 13.99 seconds for scheduling, while $Hybrid8 + L$ approach requires only 0.19 second. When combining both multi-level bound pruning and multi-directional search methods, our $ML8$ approach shows a better performance than $MD8$. In most cases, our $ML8$ approach can achieve a significant improvement over *MD*8. Moreover, our $ML8$ approach outperforms (no worse than) *Hybrid*8 approach for 17 out of 19 benchmark items. Even for the $Hybrid8 + L$ approach, our approach outperforms for 15 out of 19 benchmark items. Similarly, for the designs with pipelined functional units (in columns 11-14), we can find that our parallel pruning approach (i.e., $ML8$) outperforms the BULB approach by several orders of magnitude.

To compare with the latest RCS approach in HLS [8], we conducted the experiments on the designs (i.e., *EWF*, *EWF2*, *EWF3*, *ARF* and *FDCT*) using the same benchmarks as listed in [8]. Fig. 8 shows the comparison results between our approach *ML*8 (on a server with 96 2.4 GHz cores) and the in-place method (on a desktop with a 3.0 GHz CPU [8]). From this figure, we can find that our approach significantly outperforms the in-place method [8] for 33 out of 37 benchmark items. Our approach can achieve up to 30 times improvement for some benchmark items. Note that, due to the space limit, we only present the results for the non-pipelined designs. For the pipelined designs, we can observe the similar trend.

The core number (i.e., process number in MPI) plays an important role during the search. Fig. 9 shows the scheduling results for the non-pipelined designs with different number of cores. Generally, the more cores are used for the parallel search, the more speedup can be achieved. However, the performance of our parallel search approach is determined by the gap between the upper-bound estimation and the real length of the optimal schedules. As an example of $ARFilter\_1\_4$, with the increasing number of

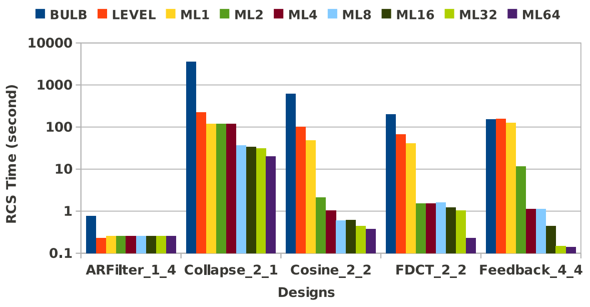

Fig. 9. Scheduling results with different number of cores.

cores, the performance does not change. This is because, in this example, the upper bound estimation equals to the length of optimal schedule. Therefore, during the multi-directional search, no new fences will be found and shared. Moreover, each sub-task needs to scan all the search space. So each of them uses the similar search time. Therefore, the search time cannot be reduced with more cores. For all the other cases, we can find that RCS with more cores will have better RCS performance. When 64 cores (i.e., $ML64$) are adopted, our approach can outperform both the BULB and $LEVEL$ approaches by several orders of magnitude for the given benchmarks.

5.2 Scheduling under Area and Power Constraints

The area and power are two key issues in hardware design. The scheduling under such constraints can be considered as a variant of the time-minimum resource constrained scheduling [21]. Since both area and power can be treated as special kinds of resources, our parallel structure-aware pruning can be used to promote the scheduling performance under such constraints. We did the experiment with the designs given in Table 2 using these two constraints. Due to the space limit, we only present the results for the non-pipelined $FDCT$ design. Our approach also provides similar results for the other designs. In the experiments, all the parallel methods use eight cores.

Fig. 10 shows the RCS result using the BULB approach, $LEVEL$ approach, $Hybrid$ approach and our structure-aware parallel pruning approaches. Under the area constraint of 140 units, we check the RCS with different power constraints (from 60 to 120 with an increment of 20). From this figure, we can find that our $ML8$ approach can achieve the best performance. Compared to the state-of-the-art B&B style pruning methods (i.e., $Hybrid8$ [25]), our approach can improve the pruning performance by up to 403 times. It is

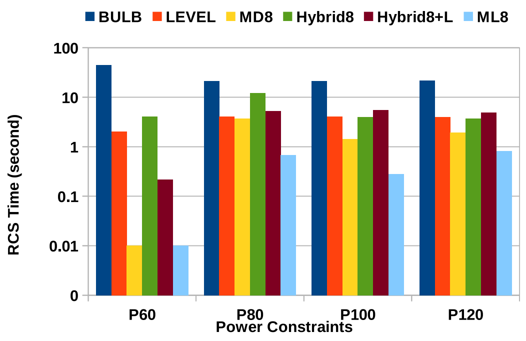

Fig. 10. Scheduling results with an area of 140 units.

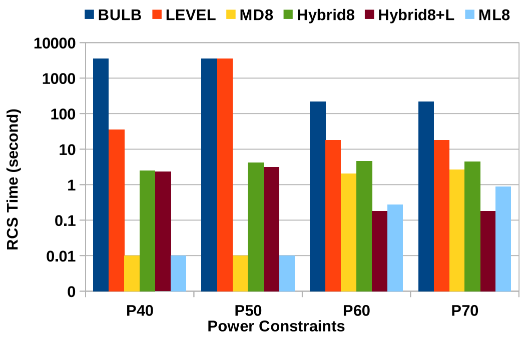

Fig. 11. Scheduling results with an area of 100 units.

important to note that, for the design of $FDCT$, when the RCS time using $MD8$ is small (e.g., 0.01 second under the power constraint of 60), $ML8$ does not outperform the $MD8$ too much. This is because $MD8$ can quickly find the optimal result with less help from $ML8$ approach. However, when $MD8$ failed to hit the optimal result within a short time, our $ML8$ approach can improve the search performance by up to 5.5 times. We also conducted RCS for the $FDCT$ design with a smaller area constraint (i.e., 100 units). As shown in Fig. 11, we check the RCS with different power constraints (from 40 to 50 with an increment of 10). From this figure, we can find a similar trend in Fig. 10. In this case with a power constraint of 40 units, the BULB approach cannot get an optimal schedule within the time limit (i.e., 3,600 seconds).[1] When the power constraint equals to 50 units, neither BULB nor $LEVEL$ can figure out an optimal schedule within the time limit. However, for both cases, our $ML8$ approach can achieve the optimal result within 0.01 second. When the power constraint is larger than 50, we can find that $ML8$ can achieve up to 422 times improvement (for the case with 100-unit area and 50-unit power) compared to the $Hybrid8$ approach [25]. Moreover, under the benefit of our structure-aware pruning technique, the performance of $Hybrid8$ approach can be further improved. We can find that the $Hybrid8+L$ approach can achieve up to 26.1 times improvement (for the case with 100-unit area and 60-unit power) compared to the $Hybrid8$ approach. Since both the results of $Hybrid8+L$ approach and $ML8$ approach are smaller than 1 seconds, both RCS time is quite similar and acceptable in practice.

6 Conclusion

This paper presented a novel parallel structure-aware pruning approach for efficient resource-constrained HLS scheduling. Unlike existing B&B style algorithms which only compare the upper- and lower-bounds between optimal schedule searched so far and current incomplete enumerating schedule, our approach investigated the structural scheduling information of optimal schedule candidates (i.e., fences) to prevent the search of non-optimal schedules in a proactive manner. Based on the collaborative multi-directional search and the shared fence and bound information among parallel search tasks, our approach can promote the

1. For the ease of comparison, we set the search time of the failed search to be 3,600 seconds in Fig. 11. In this case, the results of BULB cannot be compared with other search time results.

overall RCS performance drastically. Comprehensive experimental results using various benchmarks with different resource constraints demonstrated that, by integrating our structure-aware pruning method, the performance of existing B&B style approaches can be further improved. Compared with state-of-the-art parallel branch-and-bound techniques, our parallel structure-aware pruning method can improve RCS performance by several orders of magnitude.

ACKNOWLEDGMENTS

This work was partially supported by the grants from Natural Science Foundation of China (Nos. 91418203, 61202103 and 91118008), Innovation Program of Shanghai Municipal Education Commission 14ZZ047, Shanghai Knowledge Service Platform for Trustworthy Internet of Things (No. ZF1213), and NSF grants CCF-1351054 (CAREER). Geguang Pu is the corresponding author.

REFERENCES

[1] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *IEEE Des. Test Comput.*, vol. 26, no. 4, pp. 8–17, Jul./Aug. 2009.
[2] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Des. Test Comput.*, vol. 26, no. 4, pp. 18–25, Jul./Aug. 2009.
[3] J. Cong, B. Liu, S. Neuendorffer, and J. Noguera, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.
[4] J. Cong, K. Gururaj, G. Han, and W. Jiang, "Synthesis algorithm for application-specific homogeneous processor networks," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 9, pp. 1318–1329, Sep. 2009.
[5] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer, 1992.
[6] G. De Micheli, *Synthesis and Optimization of Digital Circuits.* New York, NY, USA: McGraw-Hill, 1994.
[7] M. Narasimhan and J. Ramanujam, "A fast approach to computing exact solutions to the resource-constrained scheduling problem," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 4, pp. 490–500, 2001.
[8] C. Yu, Y. Wu, and S. Wang "An in-place search algorithm for the resource constrained scheduling problem during high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 15, no. 4, 2010.
[9] G. Shobaki, K. Wilken, and M. Heffernan, "Optimal trace scheduling using enumeration," *ACM Trans. Archit. Code Optimization*, vol. 5, no. 19, 2009.
[10] C. H. Gebotys and M. I. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 12, no. 9, pp. 1266–1278, Sep. 1993.
[11] M. Sevaux, A. Singh, and A. Rossi, "Tabu search for multiprocessor scheduling: Application to high level synthesis," *Asia-Pacific J. Oper. Res.*, vol. 28, no. 2, pp. 201–212, 2011.
[12] A. H. Timmer and J. A. G. Jess, "Execution interval analysis under resource constraints," in *Proc. Int. Conf. Comput. Des.*, 1993, pp. 454–459.
[13] M. Rim and R. Jain, "Lower-bound performance estimation for the high-level synthesis scheduling problem," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 4, pp. 451–458, Apr. 1994.
[14] Z. Shen and C. Jong, "Lower bound estimation of hardware resources for scheduling in high-level synthesis," *J. Comput. Sci. Technol.*, vol. 17, no. 6, pp. 718–730, 2002.
[15] J. A. Stankovic, M. Sprui, M. D. Natale, and G. C. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Comput.*, vol. 28, no. 6, pp. 15–25, Jun. 1995.
[16] G. Tiruvuri and M. Chung, "Estimation of lower bounds in scheduling algorithms for high-level synthesis," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 3, no. 2, pp. 162–180, 1998.
[17] Open MPI [Online]. Available: http://www.open-mpi.org, 2015.
[18] Media Benchmarks [Online]. Available: http://express.ece.ucsb.edu/benchmark/, 2015.
[19] IBM ILOG CPLEX CP Optimizer V12.3 [Online]. Available: http://www-01.ibm.com /software/commerce/optimization/cplex-cp-optimizer/index.html, 2015.
[20] Parallel Structure-Aware RCS [Online]. Available: http://faculty.ecnu.edu.cn/chenmingsong, 2015.
[21] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems," in *Proc. Int. Symp. Asynchronous Circuits Syst.*, 2010, pp. 107–116.
[22] P. Prabhakaran and P. Banerjee, "Parallel algorithms for force directed scheduling of flattened and hierarchical signal flow graphs," *IEEE Trans. Comput.*, vol. 48, no. 7, pp. 762–768, Jul. 1999.
[23] H. Steve and B. Forrest, "Automata-based symbolic scheduling for looping DFGs," *IEEE Trans. Comput.*, vol. 50, no. 3, pp. 250–267, Mar. 2001.
[24] M. Chen, S. Huang, G. Pu, and P. Mishra, "Branch-and-bound style resource constrained scheduling using efficient structure-aware pruning," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2013, pp. 224–229.
[25] M. Chen, L. Zhou, G. Pu, and J. He, "Bound-oriented parallel pruning approaches for efficient resource constrained scheduling of high-level synthesis," in *Proc. Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2013, pp. 1–10.
[26] M. Chen, F. Gu, L. Zhou, G. Pu, and X. Liu, "Efficient two-phase approaches for branch-and-bound style resource constrained scheduling," in *Proc. Int. Conf. VLSI Des.*, 2014, pp. 162–167.
[27] T. G. Crainic, B. L. Cun, and C. Roucairol, "Parallel branch-and-bound algorithms," in *Parallel Combinatorial Optimization*, New York, NY, USA: Wiley, 2006, ch. 1, pp. 1–28.
[28] T. K. Ralphs, "Parallel branch and cut," *Parallel Combinatorial Optimization*. New York, NY, USA: Wiley, ch. 3, pp. 53–101, 2006.

Mingsong Chen (S'08-M'11) received the BS and ME degrees from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the PhD degree in computer engineering from the University of Florida, Gainesville, in 2010. He is currently an associate professor with the Software Engineering Institute of East China Normal University. His research interests are in the areas of design automation of cyber-physical systems, formal verification techniques and mobile cloud computing. Currently, he is an associate editor of *Journal of Circuits, Systems and Computers*. He is a member of the IEEE.

Xinqian Zhang received the BE degree from the Software Engineering Institute of East China Normal University in 2015. He is currently working toward the PhD degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests are in the areas of design automation of embedded systems, quantitative analysis of systems, statistical model checking, and software engineering.

Geguang Pu received the PhD degree from the Mathematics Department, Peking University, in 2005. Currently, he is a professor with the Software Engineering Institute, East China Normal University. His research interests include program analysis, formal modeling of business processes, automated testing, and verification. From 2006, he served as PC members in a number of international academic conferences, including ATVA, ICFEM, ICTAC, etc.

Xin Fu (S'05-M'10) received the PhD degree in computer engineering from the University of Florida, Gainesville, in 2009. She was the US National Science Foundation (NSF) Computing Innovation Fellow with the Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, from 2009 to 2010. From 2010 to 2014, she was an assistant professor in the Department of Electrical Engineering and Computer Science, the University of Kansas, Lawrence. Currently, she is an assistant professor at the Electrical and Computer Engineering Department, University of Houston, Houston. Her research interests include computer architecture, high-performance computing, hardware reliability and variability, energy-efficient computing, and mobile computing. She received 2014 NSF Faculty Early CAREER Award and 2012 Kansas NSF EPSCoR First Award. She is a member of the IEEE.

Prabhat Mishra (S'00-M'04-SM'08) received the BE degree from Jadavpur University, India, the MTech degree from the Indian Institute of Technology, Kharagpur, and the PhD degree from the University of California, Irvine all in computer science. He is currently an assistant professor with the Department of Computer and Information Science and Engineering, University of Florida. His research interests include design automation of embedded systems, security and reliability, and hardware/software verification. His research has been recognized by several awards including the US National Science Foundation (NSF) CAREER Award, two best paper awards, and 2004 EDAA Outstanding Dissertation Award. He currently serves as the deputy editor-in-chief of *IET Computers & Digital Techniques*. He is an associate editor of *ACM TODAES, IEEE TVLSI, IEEE Design & Test*, and *Journal of Electronic Testing*. He is a senior member of both the ACM and IEEE.