Efficient Resource Constrained Scheduling Using Parallel Two-Phase Branch-and-Bound Heuristics

Mingsong Chen, *Member, IEEE*, Yongxiang Bao, Xin Fu, *Member, IEEE*, Geguang Pu, and Tongquan Wei, *Member, IEEE*

Abstract—Branch-and-bound (B&B) approaches are widely investigated in resource constrained scheduling (RCS). However, due to the lack of approaches that can generate a tight schedule at the beginning of the search, B&B approaches usually start with a large initial search space, which makes the following search of an optimal schedule time-consuming. To address this problem, this paper proposes a parallel two-phase B&B approach that can drastically reduce the overall RCS time. This paper makes three major contributions: i) it proposes three partial-search heuristics that can quickly find a tight schedule to compact the initial search space; ii) it presents a two-phase search framework that supports the efficient parallel search of an optimal schedule; iii) it investigates various bound sharing and speculation techniques among collaborative tasks to further improve the parallel search performance at different search phases. The experimental results based on well-established benchmarks demonstrate the efficacy of our proposed approach.

Index Terms—Resource constrained scheduling, branch-and-bound, parallel two-phase pruning, high-level synthesis

1 Introduction

Increasing complexity coupled with time-to-market constraints enlarge the gap between *Electronic System Level* (ESL) designs and *Register-Transfer Level* (RTL) implementations. To enable rapid generation of hardware designs while considering various requirements (e.g., performance, area and power), *High-Level Synthesis* (HLS) [1], [2] is proposed to automatically translate ESL designs to low-level RTL implementations. HLS has been widely adopted in many industry design fields, especially in the *Field-Programmable Gate Array* (FPGA) domain [3].

This paper focuses on HLS scheduling under resource constraints, called *Resource Constrained Scheduling* (RCS). For HLS, ESL specifications are converted into *Data Flow Graphs* (DFGs), which are used as an intermediate representation for the design exploration and performance estimation purpose. Scheduling assigns each operation of a DFG with a *control step* (c-step) which indicates the start execution time of the operation. Since RCS needs to explore a huge number of possible designs and make the trade-off among various resource constraints, it is a major challenge in HLS. Given a DFG and a pre-defined set of resources (e.g., number of function units, power, area) with specified overheads, RCS tries to find a schedule of operations with least overall c-steps.

Essentially, RCS is an NP-Complete problem with constraints of computation precedence and resource limits [4]. To avoid forcefully enumerating all possible schedules, many approaches [5], [6], [7] are proposed to reduce the searching time of optimal schedules. The basic idea is to remove as many infeasible or inferior schedules during the HLS search as possible. As a kind of promising RCS search paradigms, the B&B RCS methods [5] are widely investigated to prune the search space (i.e., the set of all combinations of operation assignments). During the search, B&B approaches update the upper-bound length estimation of the optimal schedule searched so far dynamically when encountering new better schedules. Such upper-bound length information can be used to determine the inferior schedules which are worse than the up-to-date best scheduling result. Although B&B approaches are efficient in pruning these inferior schedules, one major bottleneck is that they cannot guarantee a tight initial feasible schedule to restrict the search range of each operation, which can easily result in a huge initial search space. Furthermore, B&B approaches explore the state space in a recursive manner. If the remaining operations cannot be used to derive a better schedule, the loose dispatch range of operations and deep recursive search will result in the *stuck-at-local-search*, which is the main cause of the long search time.

Since more and more computers are equipped with multi-core CPUs, to avoid the stuck-at-local-search problem and improve the RCS performance, we propose a novel parallel B&B approach which can quickly narrow down the initial search space as well as search for optimal schedules in a collaborative manner. To fully utilize the capability of parallel

- *M. Chen, Y. Bao, G. Pu, and T. Wei are with the Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai 200062, China. E-mail: {mschen, yxbao, ggpu}@sei.ecnu.edu.cn, tqwei@cs.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.*

Manuscript received 11 Mar. 2016; revised 7 Sept. 2016; accepted 21 Oct. 2016. Date of publication 27 Oct. 2016; date of current version 12 Apr. 2017. Recommended for acceptance by M. Huebner.
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/TPDS.2016.2621768

search to quickly achieve an optimal schedule, our approach tries to address following three issues:

1) How to quickly achieve a better initial schedule than traditional B&B approaches with a small overhead? To achieve a tight initial feasible schedule, we introduce three efficient coarse-granularity partial-search heuristics that can easily escape from the stuck-at-local-search from different perspectives, thus reducing the search time for a shorter initial schedule.
2) How to organize parallel search tasks to enable efficient exploration of optimal schedules? Based on the proposed partial-search heuristics, we present a novel parallel two-phase B&B approach. By grouping search tasks with different partial-search heuristics and different operation enumeration order, the chance of early detection of better or optimal schedules increases. Therefore, our approach can reduce the overall RCS time.
3) How to achieve potential synergy among parallel tasks for the efficient search of optimal schedules? This paper proposes a collaborative framework that supports both the bound sharing and speculation mechanisms among parallel search tasks, which can further promote the pruning capability of parallel B&B approaches.

Although [22] proposes an efficient parallel RCS framework with the similar search task organization, it focuses on the utilization of parallel structure-aware pruning rather than the initial upper-bound optimization and bound speculation-based collaborative search. To further reduce the RCS time, our framework also incorporates the structure-aware pruning as an orthogonal approach to our proposed methods.

This paper is organized as follows. Section 2 introduces the related work on RCS. Section 3 presents the related background of B&B style RCS and motivates the needs of our parallel two-phase approach. Besides introducing our partial-search techniques which aim at finding tight initial schedules, Section 4 proposes our parallel two-phase B&B RCS framework in detail. Section 5 compares our approach with both the state-of-the-art sequential and parallel B&B approaches. Finally, Section 6 concludes the paper.

2 Related Work

Unlike non-optimal HLS scheduling methods (e.g., force directed scheduling [9]), this paper focuses on how to quickly obtain optimal HLS schedules. As a promising way to deal with HLS scheduling, *execution interval analysis* approaches perform the lower- and upper-bound estimation before real scheduling, which can reduce the RCS searching time. For example, Timmer and Jess [19] presented a unified approach for lower-bound functional area and cycle budget estimations based on relaxing precedence constraints in behavioral design descriptions. By calculating the minimal overlap among different execution intervals of operations, Sharma and Jain tried to estimate architecture resources and performance [10]. Ohm et al. [8] presented a comprehensive technique for lower-bound estimation. Shen and Jong [11] 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 to save the search time, most of existing methods are developed to find near-optimal solutions rather than optimal ones.

Branch-and-bound approaches are effective in avoiding unfruitful search, thus it is widely investigated in RCS. Narasimhan and Ramanujam [5] proposed an efficient B&B approach called BULB, which can prune fruitless or inferior schedules based on the estimation of the lower and upper bounds of optimal schedules. Hansen and Singh [16] gave an efficient B&B approach that can reduce the scheduling time considering various resource constraints. To enhance the pruning capability of B&B approaches, Chen et al. [17] proposed an efficient pruning approach based on the structural information of schedules. Although the above B&B approaches are promising in finding optimal results quickly, so far most B&B methods search for optimal RCS solutions in a sequential way using a single core.

Parallelism is an important topic in RCS. Various approaches are proposed to save the overall scheduling time. For example, to reduce the *Integer Linear Programming* (ILP)-based RCS models, various parallel *branch-and-cut* heuristics [21] have been investigated. Although such methods allow designers to describe RCS problems in a natural way, the number of variables in ILP models increases very fast with the size of graph-based models. Consequently, solving complex RCS problems using ILP models may need a prohibitively long time. As an alternative, Chen et al. [18] described a bound-oriented parallel B&B approach to improve the RCS time in HLS. By combining both bound speculation and search space partitioning heuristics among sub-search tasks, their approach can drastically reduce the overall search time of optimal schedules. In [22], Chen et al. proposed a parallel structure-aware pruning technique that can drastically reduce the overall RCS time by sharing level bound information among collaborative search tasks. Nevertheless, these B&B approaches do not consider how to achieve a tight feasible schedule at the beginning of the RCS to reduce the initial search space. Furthermore, they only considered the upper-bound speculation without studying the effect of the lower-bound update during the parallel searching.

Although B&B approaches are promising in pruning search space, most of them adopt the heuristics such as list scheduling [4] to achieve a feasible schedule first, which cannot always guarantee a tight initial search space. Therefore, it can easily cause a prohibitively long search time. To the best of our knowledge, our approach proposed in this paper is the first attempt to utilize the parallel tasks to conduct both the partial-search and bound-based search collaboration to further reduce the overall RCS time.

3 Preliminary Knowledge

This section presents the basic knowledge for the resource constrained scheduling in HLS, including the graph-based notations and a classic B&B RCS algorithm named BULB.

3.1 Notations for the RCS Problem

HLS scheduling generally employs DFGs to describe its behavior. A DFG is a Directed Acyclic Graph (DAG) $G = (V, E)$, where V is a set of vertices (nodes) designating

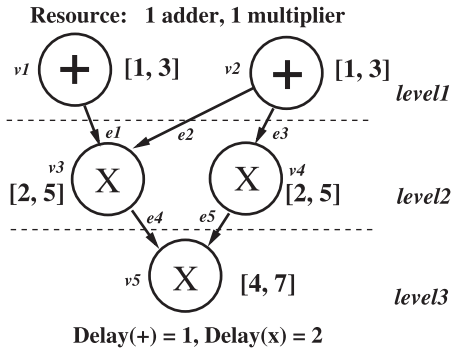


Fig. 1. An HLS DFG example.

different functional operations, and E is a set of directed edges describing operation dependencies. 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 be complete before the start of the operation of v_j. As an example shown in Fig. 1, the DFG consists of five nodes and five directed edges. In an HLS DFG, each v_i is associated with an operation op_i, where $type(op_i)$ indicates the functional unit type occupied by op_i and $delay(op_i)$ denotes the time delay of op_i. Operations without any predecessors are *input operations*, and operations without any successors are *output operations*.

Various graph notations are used to facilitate the HLS scheduling analysis. In this paper, we use $G' = (V', E')$ to represent a *sub-graph* of $G = (V, E)$ where $V' \subseteq V$ and $E' \subseteq E$. The sub-graph including the nodes v_i and all its direct and indirect predecessors is denoted by $G_{pre}(v_i)$. The sub-graph with the node v_i and all its connected successors is denoted as $G(v_i)$. A *path* is a sequence of nodes that starts from an input operation and ends with an output operation. The *size* of a path is the number of nodes along the path. We use $P_l(G)$ to denote the size of a path in G with maximum nodes. The *length* of a path is the sum of operation delays of the nodes along the path. The path with the longest length is called *critical path*. We use $CP_w(G)$ to denote the length of a critical path of G.

In a DFG, each node v is associated with *level* information, which indicates the largest size of all sub-paths that start from input nodes to v, i.e., $Level(v) = P_l(G_{pre}(v))$. During RCS, the dispatching order of operation op_i is determined by the value of $CP_w(G(v_i))$ in a non-ascending manner. As an example shown in Fig. 1, $\rho = <v_1, v_2, v_4, v_3, v_5>$ can be a candidate of scheduling orders, since $CP_w(G(v_1)) = 5$, $CP_w(G(v_2)) = 5$, $CP_w(G(v_3)) = 4$, $CP_w(G(v_4)) = 4$ and $CP_w(G(v_5)) = 2$. For each node v, $L_s(v)$ and $L_e(v)$ denote the indices of the first and last dispatched operations within the same level of v, respectively. Assuming that ρ is the dispatching order of operations in Fig. 1, we can get $L_s(v_3) = L_s(v_4) = 4$ and $L_e(v_3) = L_e(v_4) = 3$, since v_4 is the first dispatched operation and v_3 is the last dispatched operation in the level 2.

In HLS, the *c-step* is the basic time unit. An operation will occupy a specific number of continuous c-steps for execution on its corresponding function unit during the scheduling. The start time of an operation is regarded as the first c-step of its execution. To restrict the start time of operations, the B&B search assigns each operation with an interval $[ASAP(op_i), ALAP(op_i)]$, where $ASAP$ (*As-Soon-As-Possible*) and $ALAP$ (*As-Late-As-Possible*) indicate the earliest and latest dispatching time for operations, respectively. For example, in Fig. 1 the start c-step of op_3 should be within the interval $[2, 5]$. Since RCS search space can be represented by the Cartesian product of all the operation intervals, to achieve a better RCS performance, it is required that the intervals of operations need to be as tight as possible. The following definition presents two widely used approaches to calculate the initial $ASAP$ and $ALAP$ values.

Definition 3.1. *Let G be a DFG for RCS, 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*

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

$ALAP_G(op_i)$ 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 using the formula

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

Note that $le(S)$ has to be determined before calculating the $ALAPs$ of operations. As an efficient method, list scheduling [5] can achieve such a feasible schedule quickly. However, so far there is no approach that can guarantee a tight initial schedule before RCS.

According to Definition 3.2, a *schedule* is an assignment function S which dispatches each operation op_i at c-step $S(op_i) \in Z^+$. Here, the condition (1) indicates the precedence relation between operations, and condition (2) asserts that at any time the number of specific resource required by operations should be no more than available ones. 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 shortest one among all the explored feasible schedules so far. The *global optimal schedule* is the optimal schedule when all of the state space has been explored. During the RCS search, we use the *upper-bound* and *lower-bound* to estimate the maximum possible and minimum possible lengths of the global optimal schedule, respectively. Generally, $le(S)$ can be used as an upper-bound of the global optimal schedule. Based on the method proposed in [12], the lower-bound can be calculated assuming that there are unlimited resources.

Definition 3.2. *Let $G = (V, E)$ be a DFG, and OP be the set of operations corresponding to V, where $|V| = |OP| = N$. Assume that the target implementation supplies M types of functions, $\Sigma = \{\pi_1, \ldots, \pi_M\}$, and there are $num(\pi_i)$ units of π_i ($1 \le i \le M$). A function $S : OP \to Z^+$ is a feasible schedule of G, iff all the following conditions satisfy:*

(1) If $\langle op_i, op_j\rangle \in E$, then $S(op_i) + delay(op_i) \le S(op_j)$ holds.

(2) For any time t and any operation of type π_j, $|\{op_i \mid type(op_i) = \pi_j \wedge ([S(op_i),\ \ S(op_i) + delay(op_i)] \bigcap [t, t]) \neq \emptyset\}| \le num(\pi_j)$.

Let the pair $(op_i, S(op_i))$ denote the scheduling information for operation op_i and its dispatching time. Assume that there are only one adder and one multiplier for the implementation in the example shown in Fig. 1. The binary relation $\{(op_1, 1), (op_2, 2), (op_3, 3), (op_4, 5), (op_5, 7)\}$ represents a

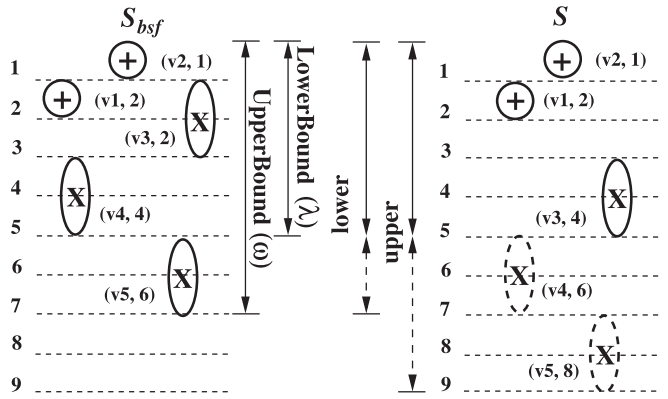


Fig. 2. A schedule pruning scenario in BULB.

feasible schedule with a length of 8 for the DFG. The binary relation $\{(op_1, 2), (op_2, 1), (op_3, 4), (op_4, 2), (op_5, 6)\}$ indicates a global optimal schedule with a length of 7.

3.2 BULB Approach

Based on the given $[ASAP, ALAP]$ intervals of operations, enumerating all the feasible schedules in RCS is extremely time-consuming. To reduce the fruitless search efforts, the BULB approach [5] was proposed to prune inferior schedules in a branch-and-bound manner. Besides $[ASAP, ALAP]$ intervals which restrict the search range of operations, B&B approaches [5], [16] use two other important data structures to prune inferior schedules: i) S_{bsf} which keeps the optimal schedule searched so far, and ii) S which is the current enumerating schedule with unscheduled operations. Based on the estimation of the upper- and lower-bounds of S_{bsf} and S, B&B approaches can avoid the unfruitful search during the RCS exploration.

Fig. 2 presents a typical scenario which illustrates how BULB makes the pruning on the example shown in Fig. 1. In this example, we use ω to specify the upper-bound of S_{bsf} (i.e., $UpperBound$). Initially, ω is equal to the length of a feasible schedule determined by the list scheduling approach. Then ω decreases dynamically when a shorter feasible schedule is found during the search of global optimal schedules. We use λ (i.e., $LowerBound$) to indicate the lower-bound length of S_{bsf}. According to [12], λ can be calculated assuming that there are unlimited resources. As shown in the right part, the current schedule S has only three operations (in solid ovals) dispatched. It also associates with two bound estimations, i.e., $lower$ and $upper$, which denote the lower- and upper-bound of S respectively based on the dispatched operations of S. Assuming that there exist unlimited resources, we can get $lower = 7$. In Fig. 2, the dotted ovals indicate the virtual dispatching of the remaining unscheduled operations, which can be used to infer the value of $upper$ (i.e., 9). In this example, we can find that $lower$ is equal to ω. Therefore, the scheduling for the unexplored operations of S can be terminated, since there is no chance that S can be shorter than S_{bsf}. Consequently, the operation enumeration based on S is useless. During the RCS search, if $upper$ is smaller than ω, it means that a schedule better than S_{bsf} has been found. Then, S_{bsf} will be replaced by the new schedule for the following pruning. If ω equals λ, it indicates that S_{bsf} is a global optimal schedule and the search will be terminated.

Algorithm 1 presents the details of the BULB approach. In this algorithm, S_{bsf} as well as ω are 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 satisfied. We use the procedure $Precedence(op_i)$ to check whether all the precedents of operation op_i are complete and use the procedure $ResAvailble(op_i, step)$ to check whether the resources required by op_i are enough at a given c-step. Note that the BULB approach can be extended to solve RCS problems under various kinds of non-functional constraints (e.g., area, energy, power) [16]. For example, if we want to incorporate power constraints in Fig. 1, we only need to put a checker in the procedure $ResAvailable(op_i, step)$ to detect whether the dispatching of op_i at c-step $step$ violates the power budget.

Algorithm 1. BULB Algorithm

Input: i) An HLS DFG D with resource constraints;
ii) Ordered operations set $OP = \{op_1, \ldots, op_N\}$;
iii) S_{bsf}, optimal schedule searched so far of D with length ω;
iv) S, which stores the current incomplete schedule.
Output: A global optimal schedule and its length for D

```
1  BULB(D, N, i, S, S_bsf, ω) begin
2    if i ≤ N then
3      for step = ASAP(op_i) to ALAP(op_i) do
4        if Precedence(op_i) ∧ ResAvailable(op_i, step) then
5          lower = le(LBound(S, i));
6          upper = le(UBound(S, i));
7          if upper < ω then
8            ω = upper;
9            S_bsf = UBound(S, i);
10           if ω == λ then
11             Return (S_bsf, ω);
12           end
13           UpdateALAP();
14       end
15       if lower < ω then
16           S(op_i) = step;
17           ResOccupy(step,type(op_i),delay(op_i));
18           BULB(D, N, i + 1, S, S_bsf, ω);
19           ResRestore(step,type(op_i),delay(op_i));
20         end
21       end
22     end
23   end
24   Return (S_bsf, ω).
25 end
```

In Algorithm 1, if both operation precedence and resource availability constraints hold, lines 5 and 6 will schedule undetermined operations in two different ways: i) $LBound(S, i)$ dispatches the undetermined operations without considering the resource constraints, hence it obtains the lower-bound length $lower$ for S; and ii) $UBound(S, i)$ generates a feasible schedule for the undetermined operations using the list scheduling method so that the upper-bound length $upper$ of S can be obtained. If $upper$ is smaller than ω, it means that $UBound(S, i)$ is shorter than S_{bsf}. Therefore, the S_{bsf} and ω will be updated based on $UBound(S, i)$ in lines 8 and 9. If ω equals λ, it indicates that an optimal schedule has been found. Therefore, line 11 will terminate the whole

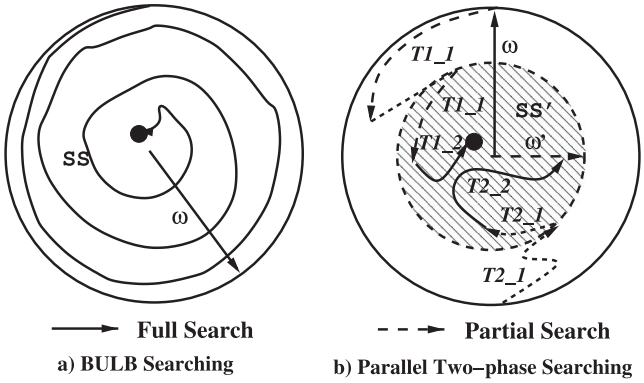


Fig. 3. Comparison between BULB and our approach.

BULB search. To further restrict the search space, line 13 updates the $ALAP$ values of operations with a smaller ω. If $lower$ is smaller than ω, lines 16-19 will dispatch the next ordered operation recursively. Otherwise, the current incomplete schedule S can be pruned safely. Finally, the algorithm returns a shortest schedule under the constraints.

4 Our Parallel Two-Phase Approach

4.1 Two-Phase Search Space Reduction

From Section 3.2, we can find that ω plays an important role in determining the BULB performance. A wise use of ω can not only compact the initial [$ASAP$, $ALAP$] intervals (i.e., RCS search space), but also can accelerate the pruning of the inferior schedules. In BULB, the initial ω value is calculated using the list scheduling approach, which often fails to get a tight value for ω. A large initial ω value will result in a huge initial search space. As an example of an RCS problem, let ω and ω' be two feasible initial scheduling candidates, where $\omega > \omega'$. Since B&B approaches count all the operation $[ASAP, ALAP]$ intervals in the recursive HLS searching, the search space SS corresponding to ω will be much larger than the search space SS' corresponding to ω'. Since SS is larger than SS', the chance of vain search in S is higher. Consequently, the search space will shrink slowly, which will easily result in deep recursive procedure calls, i.e., stuck-at-local-search. In other words, a large initial search space can disable the pruning efficiency due to the slow convergence of ω to its optimal value. Therefore, how to quickly get a tight initial search space determines the HLS scheduling performance. However, for the current version of BULB approach, it is hard to guarantee an enough tight initial feasible schedule.

To achieve a tight initial schedule, we propose a parallel two-phase approach as shown in Fig. 3 which aims to improve the overall RCS performance. Fig. 3a shows the searching using the classical BULB approach. Due to a large initial ω, the BULB search needs a quite long time to find the optimal schedule (denoted by the black dot). To tighten the initial search space, we partition the BULB approach into two phases: *partial-search* phase and *full-search* phase. Partial-search represents the search which tries to explore only a small subset of the search space, and the full-search does the same job as the original BULB approach. Fig. 3b illustrates an example of our parallel two-phase approach which requires much less time than the BULB approach. In Fig. 3b, there are two parallel collaborative search tasks $T1$ and $T2$, which start from different corners of the search space. We use TX_Y to indicate the Yth search phase of task TX. During the collaborative search, the partial-search $T2\_1$ quickly obtains a shorter initial schedule with a length of ω'. It notifies $T2$ about the newly found upper-bound immediately. Based on ω', the search space of both $T1$ and $T2$ can be drastically reduced to SS' at the same time and the pruning capability of both tasks can be improved. Assume that in the full-search $T1\_2$ can ensure it has found an optimal schedule earlier than $T2\_2$. Then both $T1\_2$ and $T2\_2$ can be terminated. In this example, due to the incomplete search space, the overhead of $T1\_1$ and $T2\_1$ is quite small. However, within the shrunken search space, the full-search of $T1\_2$ and $T2\_2$ requires much less time than the BULB approach as shown in Fig. 3a. Therefore the overall RCS time can be drastically reduced.

Based on the example shown in Fig. 3, the parallel search can benefit this two-phase heuristic. First of all, if different partial-search heuristics are employed in the parallel search, the chance of finding a shorter initial scheduled will increase. Next, the bound information sharing enables the search space reduction for both partial- and full-search phases, which in turn can reduce the overall search time. Moreover, if the BULB approach itself is one of the parallel search tasks, we can guarantee that the parallel two-phase search can be no worse than the BULB approach. Therefore, it is natural to adopt the parallel search tasks to conduct the two-phase search of optimal schedules. The following sections will introduce our parallel two-phase search framework in detail.

4.2 Partial-Search Heuristics

The partial-search time has a dominant effect on the overall performance. It should be as small as possible. Therefore, how to quickly find a schedule with small ω in partial-search phase is becoming a key challenge in our two-phase approach. Based on our observation, the long time search in BULB is mainly caused by the lack of pruning chances. When $\omega \in [lower, upper]$, the remaining unscheduled operations need to be investigated recursively. Especially when the search goes deep down, the backtrack will become difficult. To ease the escape from the stuck-at-local-search, this section will introduce three kinds of partial-search heuristics which can quickly find a tight upper-bound length with small overhead.

4.2.1 Bounded Operations

Generally, less DFG nodes involved in the recursive enumeration will lead to a quicker termination of partial-search, since this can efficiently avoid the deep recursion. Based on the above observation, our *bounded operation*-based partial-search (B.O.) tries to avoid the deep recursive search by limiting the number of the operations enumerated in the recursive search. It does not mean that we do not consider all the other operations for the scheduling. Our approach sets the input operations to be the *bounded operations*. In other words, during the partial-search, only the input nodes are investigated in the recursive enumeration. The other non-input nodes will be involved in the estimation of the upper-bound and lower-bound lengths of optimal schedules. Due to the incomplete

enumeration during the search, the partial-search will be much quicker than the original complete search.

As an example shown in Fig. 1, assume that the operations are dispatched in an order $\rho = < v_1, v_2, v_3, v_4, v_5 >$. In the bounded operation-based partial-search, only the input operations (i.e., op_1 and op_2) are involved in the recursive B&B search. For the remaining operations, we adopt the list scheduling method to achieve a feasible schedule based on the enumeration of dispatching time of op_1 and op_2. As aforementioned, the optimal scheduling needs seven c-steps. This happens only when the dispatch time of op_1 equals 2. It means that, when the c-step of op_1 is not equal to 2, the search is unfruitful. In this case, the BULB approach needs quite a long search time before $S(op_1)$ equals 2, since there are four operations that have to be fully enumerated. In our approach, we only completely enumerated the two bounded operations, i.e., op_1 and op_2. Therefore, the partial-search needs much less time to achieve the enumeration where $S(op_1) = 2$. For the following full-search phase, due to the tighter initial search space, the full-search performance on this space can be improved drastically.

4.2.2 Non-Chronological Backtrack

In BULB, operations are sorted and scheduled in a specific order. When the pruning happens, the exploration of the unscheduled operations will be terminated. Assume that the operations are dispatched in an order $\rho = < v_1, v_2, v_3, v_4, v_5 >$ in the example shown in Fig. 1 and the initial ω value is 8. Let $S' = \{(op_1, 1), (op_2, 2)\}$ be the current incomplete schedule. By using the list scheduling method on S', we can estimate that the upper-bound of S' is equal to ω (i.e., 8). Then the enumeration will be continued from op_3, and the new incomplete schedule will be $S'' = \{(op_1, 1), (op_2, 2), (op_3, 3)\}$. In this case, the stuck-at-local-search happens, since the following recursive search based on S'' will be unfruitful.

From the above example, we can find that current B&B methods cannot quickly approach to an optimal schedule due to the vain deep recursive search. To avoid such scenario and find a better schedule in the neighborhood of the current incomplete search, we adopt the non-chronological backtrack which can jump back to a non-adjacent operation. Our *non-chronological* partial-search (N.C.) is based on the DFG level structure. During the partial-search, when all the nodes of a DFG level have been scheduled, a check of backtrack condition (called *level-check condition*) will be triggered. Assume that the current incomplete schedule is S' and the current i_{th} level has the operations $op_{i_1}, op_{i_2}, \ldots, op_{i_k}$ in a sorted order. After the dispatching of op_{i_k}, we need to check whether for all the operations op_{i_j} $(1 \leq j \leq k)$ such that $S_{bsf}(op_{i_j}) \leq S'(op_{i_j})$. If the level-check condition is satisfied, a distant backtrack will be conducted. In original B&B approach, we will stay at op_{i_k} if $ALAP(op_{i_k}) > S'(op_{i_k})$ or backtrack to the last dispatched operation otherwise. In the non-chronological approach, we will jump back to the first dispatched operation of this level, i.e., op_{i_1}. Assume that $\{(op_1, 1), (op_2, 2), (op_3, 3), (op_4, 5), (op_5, 7)\}$ is a feasible schedule in Fig. 1. When the current incomplete schedule is $S' = \{(op_1, 1), (op_2, 2)\}$, the search will backtrack to op_1, and the new incomplete schedule will be $S'' = \{(op_1, 2), (op_2, 1)\}$. Therefore, the search of the optimal schedule $\{(op_1, 2), (op_2, 1), (op_3, 4), (op_4, 2), (op_5, 6)\}$ will be accelerated.

4.2.3 Search Space Speculation

The $[ASAP, ALAP]$ intervals of operations play an important role in RCS. Although the methods defined in Section 3 are promising to achieve tight $[ASAP, ALAP]$ intervals, generally it is hard to determine the tightest ASAP and ALAP values for each operation. During the B&B search, one major reason for the stuck-at-local-search is that it tries to enumerate all the c-step combinations of undispatched operations. If the intervals of undispatched operations are large, then the search time will be intolerant. Instead, if the search range of some operation can be reduced to a half, the overall search space will be reduced by half too. Based on this observation, our *search space speculation* based partial-search (S.S.) tries to speculate about better schedules by halving the search range of each operation on-the-fly.

Our search space speculation approach adopts a greedy strategy. It assumes that during the partial search the global optimal result will be always located in the first half of the range of the current dispatching operation. As an example in Fig. 1, during the partial-search the operation op_1 will be dispatched within the range $[1, 2]$. Similarly, the partial-search only considers the range $[2, 3]$ for operation op_4. Since the search range of all operations are halved, the partial-search need much less time than the full B&B search. If one shorter schedule can be obtained during the partial-search, it will be beneficial to the overall B&B search.

4.3 Collaborative Two-Phase Searching Framework

When adopting the two-phase RCS approach, *"how to guarantee that the worst case execution time of the two-phase search is no worse than the BULB approach?"* and *"how to reduce the time of both partial- and full-search as much as possible?"* are the two key issues. This section presents a parallel two-phase searching framework, which can fully explore the synergy among parallel search tasks in a collaborative manner to reduce the overall RCS time.

4.3.1 Generation and Organization of Search Tasks

The order of operation dispatching plays an important role in the parallel B&B search [22]. To maximize the effect of parallel search, parallel tasks should be scattered over the search space evenly, and the behavior variability among parallel tasks should be large. To achieve these two goals in task generation, our approach tunes the dispatching order of operations based on their level information. In BULB, each operation has a weight which indicates the dispatching priority. By adding a variance (generated by a random value in between $[0, 1]$) together with the level information (indicated by $CP_w(G)$), we can derive a new order for the operations. Let $B(i)$ be the binary format of the index of search task T_i. We tune the weight of operations using the following heuristic.

- If *i*==*0*, the operation weight will not be changed.
- If *i*!=*0* and the jth least significant bit of $B(i)$ equals 1, the weight of operations on the jth level will be increased by $(rand(0, 1) - j) \times CP_w(G)$, where

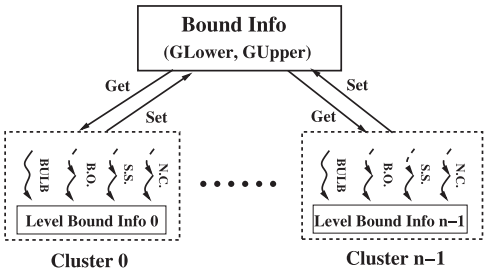


Fig. 4. Organization of our parallel search tasks.

$rand(0,1)$ generates a random number within the range $(0,1)$.

- If *i!=0* and the jth least significant bit of $B(i)$ equals 0, the operation weight on the jth level will be increased by $-j \times CP_w(G)$.

As an example in Fig. 1, the BULB approach can have an operation dispatching order $\rho = < v_1, v_2, v_3, v_4, v_5 >$ based on the $CP_w(G(v_i))$ $(1 \leq i \leq 5)$ values presented in Section 3.1. Assume the current search task is T_3. Since $B(3) = (011)_2$, the dispatching order of operations at the first and second levels will be tuned. For example, if the new weights of operations are $5 + (0.2 - 1) \times 5 = 1.0$, $5 + (0.3 - 1) \times 5 = 1.5$, $5 + (0.7 - 2) \times 5 = -1.5$, $5 + (0.8 - 2) \times 5 = -1.0$, $5 + (-3) \times 5 = -10.0$ for operations v1-v5, respectively. We can get $< v_2, v_1, v_4, v_3, v_5 >$ as a possible operation dispatching permutation for T_3 It is important to note that task T_0 has the same dispatching order as the original BULB approach.

We propose a framework that can manage tasks to collaboratively conduct the parallel search. Fig. 4 shows the organization of parallel collaborative search tasks. In this framework, parallel tasks with different partial-search heuristics are grouped into clusters, and each cluster has four tasks. To identify the partial-search type a task within a cluster, each task is assigned with a *CID*. Assuming that there are N search tasks (i.e., $T_0, T_1, \ldots, T_{N-1}$) in total, a task T_i $(0 \leq i \leq N-1)$ will have a *CID* $i\%4$ within the $(i/4)$th cluster. In a cluster, the task running the BULB search has a *CID* 0. The other three two-phase tasks in the same cluster are based on our proposed partial-search heuristics (the *CIDs* for B.O., S.S., and N.C. are 1, 2, and 3, respectively). To guarantee the RCS time of collaborative search tasks, we assume that the number of CPU cores is larger or equal to the number of parallel search tasks. In other words, during the RCS search the four tasks in a cluster are running on different CPU cores.

4.3.2 Bound Sharing Among Search Tasks

Sharing upper-bound information among parallel search tasks has been investigated in [18]. However, so far none of existing approaches considers the lower-bound sharing in parallel B&B RCS searching. As shown in Fig. 4, our framework has a global data structure called *global bound information* which keeps the current lowest upper-bound (i.e., *GUpper*) and highest lower-bound (i.e., *GLower*) of all the parallel search tasks during the search. Since in our approach each search task is created using a process, to facilitate the communication among tasks, the global bound information resides in an allocated shared memory. Along with the dynamic change of upper- and lower-bound estimation of the optimal schedule, the search space of parallel search tasks will be reduced synchronously. For example, if one search task finds one larger lower-bound, it will update the global bound information. By periodically querying the global bound information to obtain the latest bound information, other collaborative search tasks can shrink their search space earlier. Due to the tight bound estimations, the overall RCS performance can be improved.

Unlike traditional upper- and lower-bound based pruning approach in B&B searching, level-bound based pruning enables early termination of fruitless search by comparing the best schedule searched so far with the currently enumerated operations on a specific level [17]. The synergy between two pruning methods can further improve the overall RCS performance of the BULB approach. In parallel search with different operation dispatching order, search tasks may keep different best schedule searched so far with different structures. If all these information can be shared, the chance of level-bound pruning may increase. However, if one task needs to check the *level-bound condition* against the best schedules of all the parallel search tasks, the RCS performance may be degraded. Therefore, to enable the level-bound pruning with small overhead, our framework only allows the sharing of best schedules among the tasks in the same cluster. For each cluster with an index of c, we allocate a data structure $LBInfo_c$ in the shared memory to hold the best schedules searched so far for all the tasks within cluster c. During the level-bound condition checking, parallel tasks in cluster c need to compare their current incomplete schedule with the four best schedules searched so far saved in $LBInfo_c$. To simplify the construction of $LBInfo_c$, in our approach, the local level-bound information only contains the initial feasible schedules of the four search tasks within cluster c. There is no update of the level-bound information during the search.

Algorithm 2 presents the details of bound sharing operations among parallel collaborative search tasks. Functions *getGU*, *setGU*, *getGL*, and *setGL* are used to query and update the bound estimation information of global optimal schedules. To avoid the conflict of updating *GLower* and *GUpper* simultaneously, we use the globally defined mutexes *upper_bound_mutex* and *lower_bound_mutex* to ensure the exclusive update of bound information. Based on the level-bound pruning proposed in [17], $MLBCheck$ enables the structure-aware pruning. In the function, lines 22-24 determine whether op_i is the last undispatched operation in level $Level(op_i)$. Based on the operations achieved from line 25, lines 26-30 check the current schedule against all the collected schedules in $LBInfo$ using the operations in the $(Level(op_i))$th level. Note that $MLBCheck$ can be used to check both the level-check condition for N.C. partial-search and level-bound condition for structure-aware pruning (see details in Algorithm 3).

4.3.3 Bound Speculation Among Search Tasks

Without considering the level-bound sharing among parallel tasks, in the full-search phase a parallel search task needs to enumerate a huge set of feasible schedules, and the overlap of such sets between different search tasks is quite large. In other words, there exist many redundant search efforts among the parallel search tasks. Inspired by the work in [18], we use the static and dynamic upper-bound speculation to further reduce the repetitive search among the search

tasks in the same cluster. Note that our dynamic upper-bound speculation is different from [18]. It is driven by the lower-bound update.

Algorithm 2. Operations on Shared Bound Information

```
1  getGU() begin
2    Return GUpper;
3  end
4  setGU(upper) begin
5    if upper < GUpper then
6      upper_bound_mutex .lock();
7      GUpper = upper;
8      upper_bound_mutex .unlock();
9    end
10 end
11 getGL() begin
12   Return GLower;
13 end
14 setGL(lower) begin
15     if lower > GLower then
16       lower_bound_mutex .lock();
17       GLower = lower;
18       lower_bound_mutex .unlock();
19     end
20 end
21 MLBCheck(S, op_i, rank) begin
22     if i ≠ L_h(op_i) then
23       Return false;
24     end
25     OP = {op_{i_1}, ..., op_{i_k}} = (Level(op_i))_th level;
26     for each S' ∈ LBInfo_{rank/4} do
27     if
       ⋀_{1≤j≤k} (S'(op_{i_j}) ≤ S(op_{i_j})) & ⋁_{1≤j≤k}(S'(op_{i_j}) ≠ S (op_{i_j}))
28         Return true;
29     end
30   end
31   Return false;
32 end
```

In our approach, static upper-bound speculation is performed at the beginning of two-phase RCS. Since there are four tasks in a cluster, the upper-bound speculation divides the interval $[GLower, GUpper]$ into four parts evenly. Assuming that the search task has a *CID* of ci ($0 \leq ci \leq 3$), the speculation assigns the task with an upper-bound $\lfloor GUpper - \frac{(GUpper-GLower)*ci}{4} \rfloor$. Note that each cluster has a BULB search task (with *CID* = 0) that has an initial upper-bound $GUpper$. During the search, if some parallel search task finds a schedule shorter than $GUpper$, the value of $GUpper$ will be updated accordingly.

When adopting the upper-bound speculation, if one search task T_i (with *CID* $i\%4$ and upper bound ω_i) finishes the two-phase search and finds no better schedule than the speculated one with an upper bound ω_i, it means that the length of optimal schedules should be longer than ω_i. Therefore, the $GLower$ will be updated to $GLower' = \omega_i + 1$. Assume that each parallel task occupies one CPU core for the search. At this moment, if T_i finishes, its corresponding CPU core will be idle, which is a waste of computing resources. To fully utilize all available resources, our approach updates the bound information for all the parallel tasks using the following strategy:

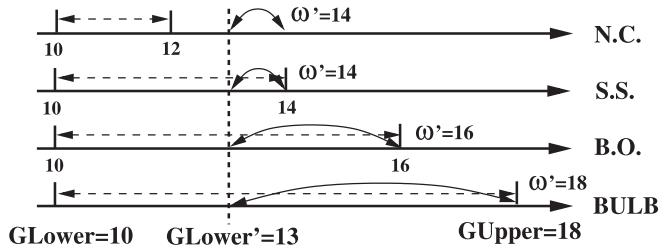


Fig. 5. An example of dynamic bound speculation.

- If a parallel task T_j has an upper-bound smaller than $GLower'$, it will terminate its current search and restart a new search with an upper-bound equal to $\lfloor GUpper - \frac{(GUpper-GLower')\times(j\%4)}{4} \rfloor$ and a lower-bound equal to $GLower'$.
- If a parallel task T_j ($j \neq i$) has an upper-bound ω_j and $\omega_j \geq GLower'$, it will set its upper-bound to $Min(\omega_j, \lfloor GUpper - \frac{(GUpper-GLower')\times(j\%4)}{4} \rfloor)$ and lower-bound to $GLower'$.

Fig. 5 shows an example of lower-bound update conducted by four tasks in a cluster. Similar to Fig. 4, the labels (e.g., $B.O.$ and $S.S$) in the figure indicate the tasks with different partial-search heuristics. It is important to note that the labels only denote the full-search (i.e., BULB search) followed by specific partial-search heuristics. In the figure, the dotted arrow lines indicate the original range of the optimal schedule length speculation. For instance, the BULB approach assumes that the length of optimal schedules is in the range $[10, 18]$. In this case, the $N.C.$ task has an upper-bound estimation of 12. Once the $N.C.$ task finishes its search within its speculated range and does not find any better schedules, it will update the $GLower$ to 13 and all the other tasks in the same cluster will update their lower-bound accordingly. Meanwhile, the $N.C.$ task will restart a new search task with an upper-bound speculation 14. For the $S.S.$ task, since the new speculated upper-bound $\lfloor 18 - \frac{(18-13)*2}{4} \rfloor = 15$ is larger than the original one, it will continue its search without changing the upper-bound. Due to the improvement of the lower-bound, the termination of the whole search becomes earlier.

In our approach, a search task periodically queries the global bound information, and speculates its bounds accordingly. Assume that the current task is T_i, and the upper-bound and lower-bound of current search task are ω and λ respectively. Let $f(i) = \lfloor GUpper - \frac{(GUpper-GLower')\times(i\%4)}{4} \rfloor$ be the procedure to calculate the speculative upper-bound of T_i. Fig. 6 shows the extended finite state machine (EFSM) for the bound speculation of the parallel task T_i. In our approach, each search task has three states:

- *Change* indicates that the task has found a better schedule since last speculation.
- *!Change* denotes that the task has not found a better schedule since last speculation.
- *Done* asserts the termination of the task.

The transitions between states specify the details of the collaborative speculation. Transition 1 initializes the upper- and lower-bounds of all the search tasks using the static speculation. Initially, since the task has not found any new

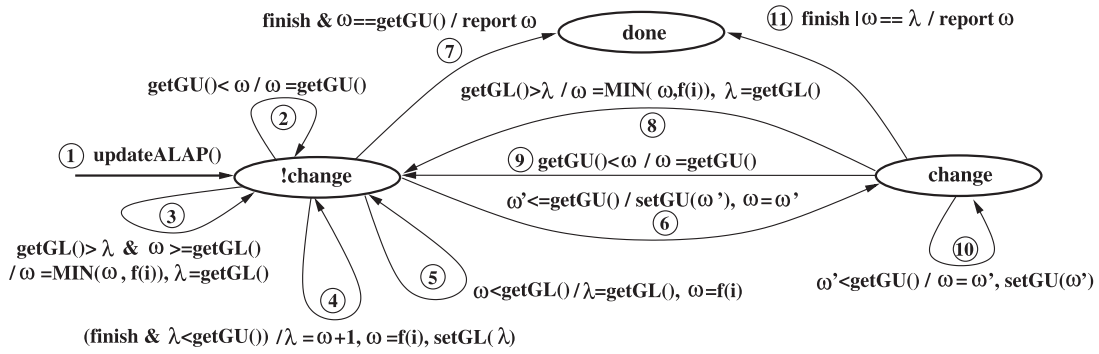


Fig. 6. EFSM of bound speculation for the parallel search task T_i.

better schedules, it will stay in the *!change* state. If it detects that other tasks have found a new better schedule, in transition 2 it will synchronize its upper-bound ω with the global information. If the task finds the change of the global lower-bound and the current lower-bound is no smaller than the global lower-bound, it will update its lower-bound and speculate it upper-bound in transition 3. In the *!change* state, if the current search is finished and $\lambda < getGU()$, it means that no better solution shorter than the speculated upper-bound is found in the given range. Therefore, transition 4 will terminate the current task and spawn a new task with a bound-speculation within the range $[\omega + 1, f(i)]$. Meanwhile, transition 4 will update the global lower-bound information. If the current upper-bound is smaller than the global lower-bound, transition 5 will terminate the current search and restart a new task with the speculation range $[getGL(), f(i)]$. During the search, if current task finds a better schedule, it will change its state to *change* and update the global upper-bound information in transition 6. If ω equals λ and the current search finishes, it indicates that ω is the length of the optimal schedule. Therefore, transition 7 will report the optimal result. If the task state is *change*, when other tasks find better lower-bound or upper-bound, it will switch to the *!change* state and update the upper- and lower-bounds accordingly as shown in transitions 8 and 9. If the current task finds a better schedule, it will stay in the *change* state as shown by transition 10. If the current task finishes in the *change* state, it will become the winner that finds the optimal schedule in transition 11.

4.4 Implementation of Our Approach

4.4.1 Implementation of a Parallel Search Task

Based on the proposed EFSM in Section 4.3.3 and BULB approach, Algorithm 3 presents the implementation details of a parallel B&B search task with an ID $rank$. In this algorithm, we use the following private global data structures to enable the B&B pruning: i) S_{bsf}, which is used to keep the optimal schedule searched by this task so far; ii) S, which is an enumerating incomplete schedule; iii) ω, which is equal to the length of S_{bsf}; and iv) λ, which indicates the lower-bound estimation of the optimal schedule of this task.

Since the bound-speculation of our approach is based on Fig. 6, Algorithm 3 implements the transitions in the proposed EFSM. In the algorithm, lines 2-6 check whether the bounded operation based partial-search is adopted or not. If yes, only the input nodes will be investigated in the partial-search as shown in line 3. Lines 8-12 deal with the search space speculation based partial-search. If the partial-search is applied, line 9 will halve the range of the current searching operation. Before operation dispatching, our method needs to check the global upper- and lower-bound information. If the global lower-bound is larger than ω, the current search task will be terminated and a new task will be created as shown in line 14. If the global upper-bound is smaller than ω, it indicates that some other task has found a better schedule. According to the transitions 2 and 9, lines 17-18 update the current ω information and set *change* to *false* to indicate that the current task is not the one that finds the global best schedule so far. If current task detects that the global lower-bound has been improved, according to the transitions 3 and 8, lines 21-23 will speculate the ω and update the λ, and set *change* to *false*. Since our approach adopts the dynamic ASAP update based on the current step assignment, we need to save the original ASAP values of all the operations of $G(op_i)$ first in line 25. Lines 27-32 conduct the level-bound checking as described in Section 4.3.2. Lines 34-35 calculate the $lower$ and $upper$ for current schedule, respectively. If $upper$ is no larger than ω, line 37 will set *change* to *true* to assert that it finds a schedule no worse than the global best schedule. If $upper$ is smaller than ω, then ω and S_{bsf} will be updated in lines 40-41. Changing ω value will trigger the checking of early termination condition in line 43 followed by the runtime space shrinking in line 45. Since a better schedule is found by the task itself, it needs to update the global upper bound-information in line 46. If op_i is the last dispatched operation of some level and the level-check condition holds, when the non-chronological backtrack is enabled, lines 49-50 will backtrack to the first dispatched operation in the same level. If the lower-bound of S (i.e., $lower$) is smaller than ω, the current operation will be scheduled. After the new c-step assignment of operation op_i in line 53, line 54 takes resources required by the operation op_i. Then op_{i+1} is processed recursively in step line 55. When the search backtracks, the resources occupied by op_i are released in step line 56. Lines 57-61 check whether the non-chronological backtrack has finished or not. When the search of $S(op_i)$ is done, line 65 restores the ASAP values of $G(op_i)$ saved in step 5. Finally, line 67 reports the results when the recursive search is complete.

Algorithm 3. Implementation of a Parallel Search Task

Input: i) An RCS DFG $D = (V, E)$, where V has been sorted;
ii) OP, a sequence of ordered operations;
ii) T, partial-search strategy;
iii) i, index of the current enumerating operation in the search;
iv) PS, indicating whether the partial-search is conducted.
Output: A schedule and its length

1 **PTask**(D, OP, T, i, PS) **begin**
2 **if** PS & T==*B.O.* **then**
3 N = |*input operations of* D|;
4 **else**
5 N= $|V|$;
6 **end**
7 **if** $i \leq N$ **then**
8 **if** PS & T==*S.S.* **then**
9 $ALAP = \lceil \frac{(ASAP(op_i)+ALAP(op_i))}{2} \rceil$;
10 **else**
11 $ALAP = ALAP(op_i)$;
12 **end**
13 **if** $getGL() > \omega$ **then**
14 **Return** (S_{bsf}, ω); **/*transition 5*/**
15 **end**
16 **if** $getGU() < \omega$ **then**
17 $\omega = getGU()$, $change = false$; **/*transitions 2, 9*/**
18 $UpdateALAP(D, \omega)$;
19 **end**
20 **if** $getGL() > \lambda$ & $\omega \geq getGL()$ **then**
21 $\lambda = getGL()$, $change = false$; **/*transitions 3, 8*/**
22 $\omega = MIN(\omega, \lfloor getGU() - \frac{(getGU()-getGL())\times(rank\%4)}{4} \rfloor)$;
23 $UpdateALAP(D, \omega)$;
24 **end**
25 $SaveASAP(D, op_i)$;
26 **for** $step = ASAP(op_i)$ *to* $ALAP$ **do**
27 **if** $!PS \mid (PS$ & $T! = N.C.)$ **then**
28 $S(op_i) = step$;
29 **if** $MLBCheck(S, op_i, rank)$ **then**
30 **Return** (S_{bsf}, ω);
31 **end**
32 **end**
33 **if** *Precedence*(op_i) $\wedge$ *ResAvailable*(op_i, $step$) **then**
34 $lower = LBound(op_i)$;
35 $upper = le(ListScheduling(D, OP, i))$;
36 **if** $upper <= \omega$ **then**
37 change = true;
38 **end**
39 **if** $upper < \omega$ **then**
40 $\omega = upper$; **/*transitions 6, 10*/**
41 $S_{bsf} = ListScheduling(D, OP, i)$;
42 **if** $\omega == LowerBound(D)$ **then**
43 **Report**(S_{bsf}, ω); **/*transition 11*/**
44 **end**
45 $UpdateALAP(D, \omega)$;
46 $setGU(\omega)$;
47 **end**
48 **if** PS & T==*N.C.* & $i == L_e(op_i)$ & $i \neq L_s(op_i)$ & $MLBCheck(S, op_i, rank)$ **then**
49 $return\_id = L_s(op_i)$;
50 **Return** (S_{bsf}, ω);
51 **end**
52 **if** $lower < \omega$ **then**
53 $S(op_i) = step$; **/* Dispatch** op_i ***/**
54 *ResOccupy(step,type(*op_i*),delay(*op_i*))*;
55 $PTask((D, OP, T, i + 1, PS)$;
56 *ResRestore(step,type(*op_i*),delay(*op_i*))*;
57 **if** $return\_id \neq -1$&$return\_id \neq i$ **then**
58 **Return** (S_{bsf}, ω);
59 **else**
60 $return\_id = -1$;
61 **end**
62 **end**
63 **end**
64 **end**
65 $RestoreASAP(D, op_i)$;
66 **end**
67 **Return** (S_{bsf}, ω).
68 **end**

Algorithm 4 describes the implementation of our parallel two-phase B&B RCS approach. To enable the parallel search, we adopt the OpenMPI library which implements each task as a process with an ID *rank*. In Algorithm 4, lines 5-10 create the data structures of the global bound information and local level-bound information in the shared memory, respectively. Line 11 parses the HLS inputs and figures out the DFG information as well as resource constraints. Line 12 sorts the operations using the method presented in Section 3.1. Lines 13-14 shuffle the operations based on the results of line 12 using the approach described in Section 4.3.1. Line 15 computes the initial $ASAP$ values for each operation of D based on the Definition 3.1. Line 16 tries to achieve an initial feasible schedule S_{bsf} by adopting the list scheduling method. Line 17 synchronizes all the search tasks. Since the S_{bsf} can be used for the level-bound pruning, it is inserted to the data structure of local level-bound information $LBInfo_{rank/4}$ in line 18. Line 19 calculates the upper- and lower-bound estimations of the optimal schedules for the current task. Line 20 updates the shared bound information among tasks. Lines 21-24 consult all the tasks and vote for the largest lower-bound λ and smallest upper-bound ω among all parallel tasks. Line 25 figures out the ALAP information for all the operations of the task. Line 26 sets the *change* to *false*. Lines 29 and 30 start a BULB search with *CID* 0 within each cluster (see transition 1 in Fig. 6). Line 32 conducts the partial-search with different strategies for the remaining tasks in clusters. When the partial-search finishes, line 33 queries the global upper- and lower-bound information. Line 34 speculates the new upper-bound for each task, and line 35 updates corresponding ALAP information (see transition 1 in Fig. 6). Followed by line 36 which denotes the beginning of the full-search, line 37 starts the B&B style full-search on the speculated upper-bounds. If the full-search finishes and no better schedules are found, according to the transition 4 in Fig. 6, it means that the optimal schedules do not have a length within range $[\lambda, \omega]$. Then, lines 39-40 will update the value of global lower-bound *GLower*. With the speculated upper-bound generated in line 41, line 42 restarts the new full-search for optimal schedules. If the current task does not find a better schedule but λ equals ω, the algorithm will terminate the whole parallel search, and report the optimal results saved in S_{bsf} and ω at line 45.

Algorithm 4. Parallel Two-Phase B&B RCS Approach

Output: An optimal schedule for D and its length

```
1: main(argc, argv) begin
2:     MPI_Init(&argc, &argv);
3:     MPI_Comm_size(MPI_COMM_WRD, &procnum);
4:     MPI_Comm_rank(MPI_COMM_WRD, &rank);
5:     if rank == 0 then
6:         CreateGlobalBoundInfo();
7:     end
8:     if rank%4 == 0 then
9:         LBInfo_{rank/4} = CreateLocalLBInfo();
10    end
11    D = ParseDFGAndConstraintsFromFile();
12    OP = {op_1, ..., op_N} = SortOperations(D);
13    code = Binary(rank, ⌈log_2(procnum)⌉);
14    OP_rank = Shuffle(D, OP, code);
15    ComputeInitialASAP(D) ;
16    S = S_bsf = InitialFeasibleSch(D);
17    MPI_Barrier(MPI_COMM_WRD);
18    LBInfo_{rank/4}.add(S_bsf);
19    (λ', ω') = (le(LBound(S, 1)), ListScheduling(D, OP_rank, 1));
20    setGU(ω') and setGL(λ');
21    MPI_Allreduce(&ω', &ω, 1, MPI_INT, MPI_MIN,
22                  MPI_COMM_WRD);
23    MPI_Allreduce(&λ', &λ, 1, MPI_INT, MPI_MAX,
24                  MPI_COMM_WRD);
25    ComputeInitialALAP(D, OP_rank, ω);
26    change = false;
27    /*Partial-search strategy type {BULB=0,B.O.=1,S.S.=2,
      N.C.=3})*/
28    if rank%4 == 0 then
29        UpdateALAP(D, ω); /*transition 1*/
30        PTask(D, OP_rank, BULB, 1, false);
31    else
32        PTask(D, OP_rank, rank%4, 1, true);
33        (λ, ω) = (getGL(), getGU());
34        ω = ⌊ω − ((ω−λ)×(rank%4))/4⌋);
35        UpdateALAP(D, ω); /*transition 1*/
36        change = false;
37        PTask(D, OP_rank, rank%4, 1, false);
38        while !change & λ < getGU() do
39            λ = ω + 1; /*transition 4*/
40            setGL(λ);
41            ω = ⌊getGU() − ((getGU()−λ)×(rank%4))/4⌋);
42            PTask(D, OP_rank, rank%4, 1, false);
43        end
44    end
45    Report (S_bsf, ω); /*transition 7*/
46    MPI_Abort(MPI_COMM_WRD, -1);
47 end
```

5 Experimental Results

To evaluate the effectiveness of our approaches, we collected the DOT files of designs *ARFilter*, *Cosine1*, *Collapse*, *Feedback*, and *Smooth Triangle* from the *MediaBench* benchmark [13], which is a standard DSP benchmark suite. We also got the benchmark *FDCT* from [20]. We implemented the BULB approach and both the sequential and parallel versions [23] of two-phase B&B RCS approaches using the C programming language and Open MPI library [15]. For the purpose of comparison, we generated and solved the ILP models for each benchmark item using *IBM ILOG CPLEX CP Optimizer* [14], which utilizes the parallel *branch-and-cut* [21] for efficient ILP solving. All the experimental results were obtained on a Linux sever with 96 Intel Xeon 2.4 GHz cores and 1 TB RAM. Since both the upper- and lower-bound estimation algorithms have a linear complexity, the initial bound estimations of the benchmarks can be achieved within less than 0.001 second, which is much smaller than the B&B search time. Therefore, we neglect the time elapsed in the estimation of these values. Note that our parallel two-phase approach generates search tasks based on randomly shuffled operations as described in Section 4.3.1. Therefore, we run each RCS problem instance in the benchmarks for five times for a fair comparison.

TABLE 1
Settings of Functional Units

Functional Units	Operation Class	Delay (unit)	Power (unit)	Area (unit)
ADD/SUB	+/-	1	10	10
MUL	×	2	20	40
$PMUL_1/PMUL_2$	$*_1/*_2$	1	10	20
DIV	÷	2	20	40
MEM	LD/STR	1	15	20
Shift	$<</>>$	1	10	5
Other	...	1	10	10

To enable the comparison with state-of-the-art HLS scheduling approaches, in this experiment we consider both the functional unit and power/area constraints which are the same as the ones used in [5], [7], [22]. Table 1 lists the corresponding settings for all different types of functional operations used in the experiment. Note that our approach can be directly applied on pipelined designs. This is because a pipelined operation can be divided into a sequence of sub-operations which correspond to different kinds of new resources. In this experiment, we assume that the pipelined multipliers denoted by "$*$" have two stages (i.e., $PMUL_1$ and $PMUL_2$) and each stage needs one c-step. Due to the limited space, we only present the results for the pipelined designs in Table 3.

5.1 Scheduling with Functional Constraints

To investigate the efficacy of the three partial-search heuristics individually, we tuned Algorithm 4 to check each of them using only one search task. We changed the *CID* of the investigated partial-search strategy to 0 and canceled all the other search heuristics in the same cluster. It is important to note that, in order to focus on the evaluation of each sequential two-phase approach separately, we did not employ the structure-aware pruning. Table 2 presents the experimental results carried out on a single core with different functional unit constraints on the six benchmarks.

The first column of the table indicates the benchmark and constraint information. The first sub-column gives the benchmark name. The second sub-column presents the functional unit constraints for the design. For example, "2 +,3×" denotes that only two adders and three non-pipelined multipliers are used in the implementation. Due to the space limitation, we only present the number of adders and multipliers in both Tables 2 and 3. The number of other functional units are fixed for both non-pipelined and pipelined designs. For the benchmarks *ARFilter*, *Collapse*, *Cosine*, and *Feedback*, we only adopted one functional unit for each

TABLE 2
Sequential RCS Results under Functional Unit Constraints

Design				BULB	LEVEL	Bounded Operation			Non-Chronological			Search Space Speculation		
Name	# of +,×	[L,U]	**len.**	[5]	[17]	$\mathbf{T}_1$	ω'	$\mathbf{T_{total}}$	$\mathbf{T}_1$	ω'	$\mathbf{T_{total}}$	$\mathbf{T}_1$	ω'	$\mathbf{T_{total}}$
ARFilter	1, 3	[14,16]	16	0.31	**0.15**	0.15	16	0.47	< 0.01	16	0.33	0.21	16	0.52
	1, 4	[14,16]	16	0.78	**0.25**	0.27	16	1.04	< 0.01	16	0.78	0.54	16	1.32
	1, 5	[14,16]	16	0.77	**0.24**	0.27	16	1.03	< 0.01	16	0.78	0.53	16	1.30
	2, 3	[14,15]	15	0.01	**0.01**	< 0.01	15	0.02	< 0.01	15	0.02	< 0.01	15	0.02
Collapse	2, 1	[22,23]	22	TO	TO	315.89	22	315.89	TO	NA	TO	0.34	22	**0.34**
	2, 2	[21,23]	NA	TO	TO	TO	NA	TO	TO	NA	TO	TO	NA	TO
Cosine1	1, 2	[28,29]	28	107.43	29.68	0.01	28	**0.01**	0.10	28	0.10	41.17	28	41.17
	2, 2	[20,23]	20	622.83	29.21	26.94	20	**26.94**	0.05	22	610.62	39.71	20	39.71
	3, 3	[16,17]	16	0.02	0.01	< 0.01	16	< **0.01**	< 0.01	16	< **0.01**	< 0.01	16	< **0.01**
FDCT	1, 2	[26,27]	26	36.91	26.63	< 0.01	26	< **0.01**	37.85	26	37.85	0.50	26	0.50
	2, 2	[18,22]	18	201.59	90.56	0.13	18	**0.13**	38.89	18	38.89	12.77	18	12.77
	2, 3	[14,17]	14	19.80	21.16	1.16	14	**1.16**	3.86	14	3.86	3.93	14	3.93
	2, 4	[13,15]	13	4.07	5.94	0.05	13	**0.05**	0.44	13	0.44	7.63	13	7.63
	2, 5	[13,14]	13	0.92	0.60	< 0.01	13	< **0.01**	0.22	13	0.22	0.32	13	0.32
	3, 4	[11,13]	11	0.55	0.48	0.39	11	0.39	0.05	11	**0.05**	0.26	11	0.26
	4, 4	[11,12]	11	0.12	0.03	0.01	11	**0.01**	0.11	11	0.11	0.04	11	0.04
Feedback	4, 4	[13,14]	13	154.18	155.74	156.87	13	156.87	156.30	13	156.30	35.43	13	**35.43**
	4, 5	[13,15]	13	TO	TO	TO	NA	TO	498.89	13	**498.89**	TO	NA	TO
	5, 5	[13,14]	13	4.87	4.92	4.96	13	4.96	4.97	13	4.97	1.13	13	**1.13**
Smooth Triangle	5, 5	[28,29]	28	0.45	0.41	0.41	28	0.41	0.40	28	**0.40**	TO	NA	TO
	6, 5	[28,29]	28	1921.54	1760.92	1767.49	28	1767.49	1753.19	28	**1753.19**	TO	NA	TO

**All RCS time is measured in seconds. "TO" means that the scheduling time is larger than 3,600 seconds. "NA" indicates that the result is unavailable.*

other function type. For the benchmarks *Cosine1* and *Smooth Triangle*, we used ten functional units for each other function type. The third sub-column gives the lower- and upper-bound estimations of the optimal schedule before the scheduling, and the fourth sub-column presents the lengths of global optimal schedules achieved by the scheduling.

TABLE 3
Parallel RCS Results under Functional Unit Constraints

Design Name	# of +,× / c-step	BULB [5]	Seq. 2P	CPLEX [14]	Hybrid8 [18]	ML8 [22]	Para. 2P w/o L.B.	Para. 2P w/ L.B.	# of +,∗ / c-step	BULB [5]	ML8 [22]	Para. 2P w/o L.B.	Para. 2P w/ L.B.
ARFilter	1, 3/16	0.31	0.33	TO	0.39	**0.13**	0.29 (< 0.01)	0.20 (< 0.01)	1, 1/19	< 0.01	< 0.01	< 0.01	< **0.01**
	1, 4/16	0.78	0.78	TO	1.01	**0.26**	0.75(< 0.01)	0.47 (< 0.01)	2, 1/19	< 0.01	< 0.01	< 0.01	< **0.01**
	1, 5/16	0.77	0.78	TO	1.03	**0.26**	0.73 (< 0.01)	0.46 (< 0.01)	2, 2/13	< 0.01	< 0.01	< 0.01	< **0.01**
	2, 3/15	0.01	0.02	1.93	< 0.01	< 0.01	< 0.01	< **0.01**	3, 1/19	< 0.01	< 0.01	< 0.01	< **0.01**
Collapse	2, 1/22	TO	0.34	TO	0.02	38.16	< 0.01	< **0.01**	2, 1/21	TO	TO	< 0.01	< **0.01**
	2, 2/21	TO	TO	TO	< 0.01	TO	0.02 (< 0.01)	< **0.01**	2, 2/21	TO	TO	< 0.01	< **0.01**
Cosine1	1, 2/28	107.43	0.01	TO	< 0.01	< 0.01	< 0.01	< **0.01**	1, 2/28	< 0.01	< 0.01	< 0.01	< **0.01**
	2, 2/20	622.83	26.94	TO	34.54	0.52	1.05 (0.42)	**0.36** (0.18)	2, 2/15	34.98	0.61	0.29 (0.44)	**0.09** (0.10)
	3, 3/16	0.02	< 0.01	TO	0.01	< 0.01	< 0.01	< **0.01**	3, 3/12	0.11	0.01	< 0.01	< **0.01**
FDCT	1, 2/26	36.91	< 0.01	TO	< 0.01	< 0.01	< 0.01	< **0.01**	1, 1/26	558.69	< 0.01	< 0.01	< **0.01**
	2, 2/18	201.59	0.13	TO	13.99	1.67	0.57 (0.42)	**0.26** (0.04)	1, 2/26	< 0.01	< 0.01	< 0.01	< **0.01**
	2, 3/14	19.80	1.16	TO	2.85	0.53	0.47 (0.10)	**0.25** (0.07)	2, 2/13	9.68	< 0.01	< 0.01	< **0.01**
	2, 4/13	4.07	0.05	TO	0.87	0.13	0.19 (0.08)	**0.09** (0.06)	2, 3/13	< 0.01	< 0.01	< 0.01	< **0.01**
	2, 5/13	0.92	< 0.01	TO	0.04	< 0.01	< 0.01	< **0.01**	3, 2/12	0.11	0.02	< 0.01	< **0.01**
	3, 4/11	0.55	0.05	TO	0.67	0.07	0.07 (0.02)	**0.03** (0.02)	3, 3/10	0.07	< 0.01	< 0.01	< **0.01**
	4, 4/11	0.12	0.01	TO	0.23	< 0.01	< 0.01	< **0.01**	4, 4/9	< 0.01	< 0.01	< 0.01	< **0.01**
Feedback	4, 4/13	154.18	35.43	TO	3.82	0.81	0.01	< **0.01**	4, 4/13	TO	3.67	0.04 (< 0.01)	**0.01**
	4, 5/13	TO	498.89	TO	4.50	1.13	0.01	< **0.01**	4, 5/13	TO	7.70	< 0.01	< **0.01**
	5, 5/13	4.87	4.96	TO	1.51	0.04	0.01	< **0.01**	5, 5/13	22.62	0..07	< 0.01	< **0.01**
Smooth Triangle	5, 5/28	0.45	0.40	TO	3.22	0.35	0.25 (0.14)	**0.06** (0.05)	2, 2/45	TO	< 0.01	< 0.01	< **0.01**
	6, 5/28	1921.54	1753.19	TO	15.86	1460.16	0.10 (0.06)	**0.04** (0.03)	2, 3/43	TO	< 0.01	< 0.01	< **0.01**

**All RCS time is measured in seconds. "TO" and "NA" have the same meaning as used in Table 2. * "×" and "∗" denote the non-pipelined and pipelined multipliers, respectively.*

The second column presents the RCS results using the BULB approach [5]. The third column presents the RCS results using the structure-aware B&B pruning technique (i.e., LEVEL [17]). The columns 4-6 present the two-phase search results based on the three proposed partial-search methods (i.e., bounded operation, non-chronological and search space speculation). For each column, we present the time spent in the partial-search, the tightest bound found during the partial-search, and the total time for the whole two-phase search, respectively. To facilitate the observation of performance comparison results, the best total RCS time of the five RCS approaches are marked in bold font.

It can be found that our partial-search based methods outperform both the BULB and structure-aware pruning approaches in most cases. Especially when the initial lengths of feasible schedules can be reduced by the partial-search, our approaches can drastically reduce the RCS time (e.g., the space speculation heuristic can achieve more than 10,588 times improvement over BULB in *Collapse* design with the constraint "2+,1×"). For the *ARFilter* design, since the length of optimal schedules equals the initial upper-bound estimation, the performance is worse than the BULB approach. However, all the other benchmarks benefit from our partial-search heuristics, since the coarse partial-search can locate the global optimal result with significantly less time. From this table, we can observe that the tighter the initial bound can achieve, the shorter the full-search time will be. In the *Cosine1* design with "2+,2×", we can find that although the non-chronological method can find a tighter upper-bound (i.e., 22), the full-search still needs a long time. This is because its initial upper-bound is not the tightest (i.e., 20) compared with other two approaches.

Generally, it is hard to state which partial-search heuristic is the best for a specific design before RCS, since the performance is mainly determined by the design itself. As shown in Table 2, the bounded operation method works best for the *FDCT* design, while the search space speculation method does best for the *Collapse* design. Although the non-chronological method cannot achieve the best results in most collected benchmark items, it can solve the problems (e.g., *Feedback* with "4+,5×") that cannot be solved by the other heuristics. All these observations inspired our parallel two-phase B&B approach, which combines the complementary search capability of the three proposed partial-search heuristics, to reduce the search time of short initial schedules.

Table 3 shows the results of our parallel two-phase B&B RCS approach. The first two columns of Table 3 contain the design information which is the same as the one shown in the first column of Table 2. The third column gives the RCS time using BULB approach. For the comparison purpose, the fourth column presents the best sequential RCS time of three proposed partial-search heuristics (i.e., $MIN(B.O.,\ N.C.,S.S)$). The fifth column presents the ILP solving time using the *CPLEX CP Optimizer* with eight cores. To compare with the state-of-the-art parallel RCS method, the sixth column denotes the RCS time using the hybrid method proposed in [18] with eight cores. The seventh column shows the RCS results using the parallel structure-aware approach introduced in [22] with eight cores. The eighth and ninth columns show the RCS time using our proposed parallel two-phase B&B RCS approach with eight cores. To show the effectiveness of the sharing of level-bound information, the eighth column presents the RCS time without the level-bound sharing and pruning, while the ninth column denotes the RCS time with the level-bound sharing and pruning. Since our approach relies on the random dispatching of operations as described in Section 4.3.1, in these two columns we give both the mean value and the standard deviation (in the parentheses) of the RCS time. Note that we omit the standard deviation when its value is smaller than 0.005. Since our approach can be directly applied on pipelined designs, we also conducted the experiment on the same benchmarks. The last five columns present the pipelined functional unit constraints, RCS time using the BULB approach [5], RCS time using the parallel structure-aware approach [22] (with eight cores), and RCS time using our parallel two-phase search method (with eight cores), respectively. Similar to the eighth and ninth columns, the last two columns give the RCS time without and with level-bound sharing and pruning, respectively. To facilitate the performance comparison, we marked the best RCS time of the parallel approaches for the non-pipelined and pipelined designs with bold font.

From this table, we can find that for the non-pipelined design our parallel two-phase approaches (columns 8 and 9) outperform the sequential two-phase approaches (column 3), the parallel CPLEX method [14] and hybrid method [18] for all of the benchmark items. This is mainly because of the following two reasons: i) the usage of the structure-aware pruning in the first-phase search; and ii) the adoption of the bound speculation and sharing introduced in Section 4.3.3. When dealing with the *ARFilter* design, the performance of our parallel two-phase approach is a little worse than the parallel structure-aware pruning method as presented in column 7. This is because the final c-step of the *ARFilter* design equals the initial upper-bound estimation. In other words, the first-phase search does not benefit the overall RCS time. Moreover, our parallel two-phase search method shares local level-bound checking information within a cluster, which may impose more burden on the level-bound pruning checking. Especially when the overall RCS time is small, this kind of overhead will be more obvious. However, when solving complex RCS problems, this time cost can be neglected. For instance, when handling the *Collapse* design with a constraint of "2+,2×", our parallel two-phase approach with level-bound sharing and pruning can achieve more than 3,600/0.01 = 360,000 times improvement than the parallel structure-aware pruning approach [22]. Compared with the parallel structure-aware pruning approach [22], in this experiment our parallel two-phase approach (with level-bound sharing and pruning) can achieve better RCS time in 18 out of 21 benchmark items. For the pipelined benchmark items shown in the last four columns, we can observe that our two-phase approaches in the last two columns significantly outperform the BULB approach as well as the parallel structure-aware pruning approach. For example, for the *Collapse* design, our parallel two-phase approach can figure out the problems quickly, while both the BULB and parallel structure-aware pruning approaches are timed out. In our experiment, we assume that a pipelined multiplier has two stages (i.e., $PMUL_1$ and $PMUL_2$) and each stage needs one c-step. Therefore, during

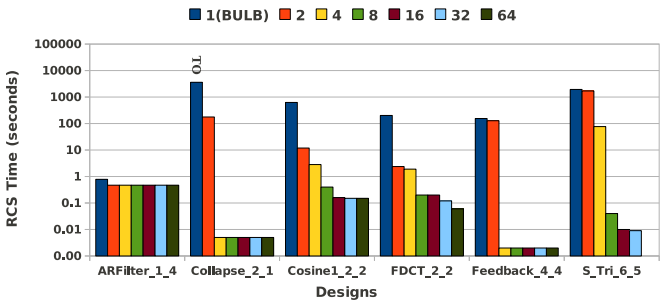


Fig. 7. Parallel RCS results with different number of cores.

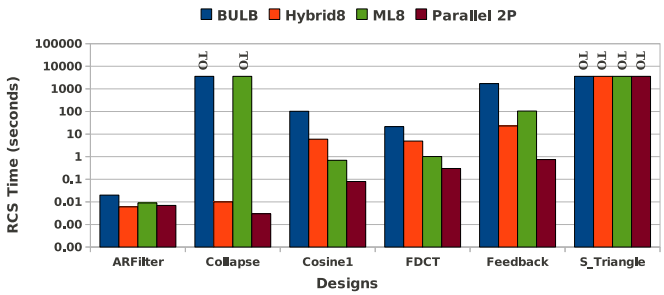


Fig. 9. Parallel RCS results under given constraints.

the operation dispatching we only need to check the availability of function units of type $PMUL_1$. The sub-operation $PMUL_2$ can be dispatched at the c-step next to the sub-operation $PMUL_1$. Due to the overlapped execution of pipelined units, the search space of the RCS for pipelined resources is typically smaller than the one of the RCS for non-pipelined resources. From Fig. 3 we can find that pipelined designs needs much less RCS time than the their non-pipelined counterparts, which is consistent to the results in [5].

To check the scalability of our parallel two-phase approach, we investigated the effect of task number on the RCS performance. Fig. 7 shows the RCS results of different benchmark items with different number of cores. In this figure, the label *design_x_y* indicates the non-pipelined design with a constraint of x adders and y multipliers. Due to the space limitation, we only present one item in each benchmark. From the other unlisted benchmarks, we can observe a similar trend. It is important to note that, in Fig. 7, the search with one core indicates the BULB approach, and two cores indicates the cooperation between the BULB and *B.O.* approach. It can be found that when there are more than four cores involved in the parallel two-phase search, we can achieve several orders of magnitude improvement over the BULB approaches. Especially for complex RCS problems (e.g., *S_Tri_6_5*), more cores will lead to a higher speedup.

5.2 Scheduling with Area and Power Constraints

Since both area and power can be treated as special kinds of resources, our approach can also be used to promote the RCS performance under such constraints. To investigate the effect of our approach under both power and area constraints, we assume that the target hardware platform has a reconfigurable part only for adders and multipliers with a power constraint of 100 units and an area constraint of 140 units. We assume that the number of all the other functional units are fixed. We assume that the number of all the other functional units are fixed. The goal of this experiment is to find a combination of adders and multipliers which can achieve the smallest overall c-steps. Fig. 8 shows the RCS results for the six designs under the constraints using different sequential approaches. From this figure, we can find that for the designs of *Cosine1*, *FDCT* and *Feedback* our two-phase approaches using different partial-search heuristics outperform both the BULB and LEVEL approaches significantly (by several orders of magnitude). For the *ARFilter* design, since the initial upper-bound cannot be reduced in the first phase, the overall performance is a little worse than the approaches BULB and LEVEL. For the designs of *Collapse* and *Smooth Triangle*, we cannot obtain the comparison results due to the timeout (i.e., 3,600 seconds) of all approaches.

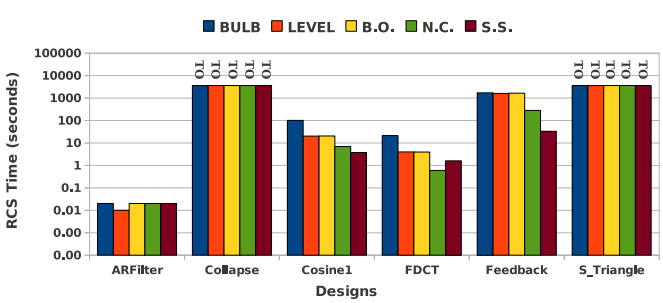


Fig. 8. Sequential RCS results under given constraints.

Fig. 9 presents the RCS results using our parallel two-phase approach together with the state-of-the-art parallel B&B style RCS approaches (i.e., Hybrid8 [18] and ML8 [22]) under the same power and area constraints as Fig. 8. From this figure, we can find that our parallel two-phase approach can achieve the best RCS performance for all the benchmarks except for the design *Smooth Triangle*. For the *ARFilter* design, due to the level-bound sharing and pruning techniques, our parallel two-phase approaches can have better performance than the BULB approach. For the *Smooth Triangle* design, the comparison fails due to the timeout of all the four approaches. For the other four designs, our parallel two-phase approach outperforms the Hybrid8 and parallel structure-ware pruning approach (i.e., ML8) methods by several orders of magnitude. This is mainly because our parallel two-phase approach can quickly find shorter initial schedules. Such shorter feasible schedules together with speculated upper-bounds, updated lower-bounds and level-bound pruning technique enable the early termination of the RCS search. For example, from the *Collapse* design, we can find that the ML8 approach cannot figure out an optimal schedule within 3,600 seconds. Note that in *Collapse* design the partial search heuristics are not helpful to find a shorter initial schedule, since all the three sequential two-phase heuristics fail to find optimal solutions within the specified time limit (i.e., 3,600 seconds) as shown in Fig. 8. However, based on the bound speculation and structure-ware pruning, our parallel two-phase approach can find an optimal schedule within less than 0.01 second. This is because optimal schedules are located in the smallest speculated search range.

6 Conclusion

This paper presented a parallel two-phase B&B approach that can quickly achieve optimal solutions for RCS

problems. By adopting the proposed partial-search heuristics in the first phase, parallel search tasks can quickly achieve a more accurate estimation for the upper-bound length of optimal schedules. In the second phase, by using our proposed bound sharing and speculation techniques, the speed of optimal schedule detection can be accelerated based on the collaboration between parallel search tasks. Due to the reduced initial upper-bound length and the parallel collaborative search, our approach can drastically reduce the overall RCS time. Experimental results using well-known benchmarks demonstrate that our approach can achieve better performance than both the state-of-the-art sequential and parallel B&B methods by several orders of magnitude.

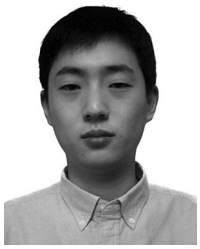
Acknowledgments

This work was partially supported by the grants from Natural Science Foundation of China (Nos. 91418203, 61672230 and 61572197), US National Science Foundation grants CCF-1351054 (CAREER), Innovation Program of Shanghai Municipal Education Commission 14ZZ047, and Shanghai Municipal NSF 16ZR1409000. Tongquan Wei is the corresponding author.

References

[1] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Des. Test Comput.*, vol. 26, no. 4, pp. 18–25, 2009.
[2] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An introduction to high-level synthesis," *Des. Test Comput.*, vol. 26, no. 4, pp. 8–17, 2009.
[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. A. Vissers, and Z. Zhang, "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] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design*. Norwell, MA, USA: Kluwer Academic, 1992.
[5] 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. 60, no. 4, pp. 490–500, 2001.
[6] 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. 27th Int. Conf. VLSI Des.*, 2014, pp. 162–167.
[7] 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, Art. no. 29.
[8] S. Y. Ohm, F. J. Kurdahi, and N. Dutt, "Comprehensive lower bound estimation from behavioral descriptions," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1994, pp. 182–186.
[9] P. Paulin and J. Knight, "Force-directed scheduling for behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 8, no. 6, pp. 661–679, Jun. 1989.
[10] A. Sharma and R. Jain, "Estimating architectural resources and performance for high-level synthesis applications," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 1, no. 2, pp. 175–190, Jun. 1993.
[11] 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.
[12] 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.
[13] Media Benchmarks. (2016). [Online]. Available: http://www.ece.ucsb.edu/EXPRESS/benchmark/
[14] IBM ILOG CPLEX CP Optimizer V12.3. (2014). [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/index.html
[15] Open MPI. (2014). [Online]. Available: http://www.open-mpi.org
[16] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems," in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst.*, 2010, pp. 107–116.
[17] 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. Int. Symp. VLSI*, 2013, pp. 224–229.
[18] 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. 9th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis Des.*, 2013, pp. 1–10.
[19] A. H. Timmer and J. A. G. Jess, "Execution interval analysis under resource constraints," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1993, pp. 454–459.
[20] H. Steve and B. Forrest, "Automata-based symbolic scheduling for looping DFGs," *IEEE Trans. Comput.*, vol. 50, no. 3, pp. 250–267, Mar. 2001.
[21] T. K. Ralphs, "Chap3: Parallel branch and cut," in *Parallel Combinatorial Optimization*. Hoboken, NJ, USA: Wiley, 2006, pp. 53–101.
[22] M. Chen, X. Zhang, G. Pu, X. Fu, and P. Mishra, "Efficient resource constrained scheduling using parallel structure-aware scheduling techniques," *IEEE Trans. Comput.*, vol. 65, no. 7, pp. 2059–2073, Jul. 2016.
[23] Two-Phase Parallel RCS Tool. (2016). [Online]. Available: http://faculty.ecnu.edu.cn/chenmingsong

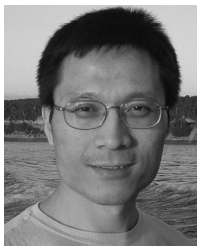
Mingsong Chen (S'08-M'11) received the BS and ME degrees from the Department of Computer Science and Technology, Nanjing University, Nanjing, China, in 2003 and 2006 respectively, and the PhD degree in computer engineering from the University of Florida, Gainesville, in 2010. He is currently a professor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



Yongxiang Bao received the BE degree from the Department of Computer Science and Technology, Anhui University of Technology, Anhui, China, in 2014. He is currently working toward the master's degree in the Department of Embedded Software and System, East China Normal University, Shanghai, China. His research interests include design automation of embedded systems, statistical model checking, and software engineering.

Xin Fu (S'05-M'10) received the PhD degree in computer engineering from the University of Florida, Gainesville, in 2009. She was an NSF Computing Innovation fellow in 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, University of Kansas, Lawrence. Currently, she is an assistant professor in 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.

Geguang Pu received the PhD degree in mathematics from Peking University, in 2005. Currently, he is a professor in 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 a PC members in a number of international academic conferences, including ATVA, ICFEM, ICTAC, etc.



Tongquan Wei (S'06-M'11) received the PhD degree in electrical engineering from Michigan Technological University, in 2009. He is currently an associate professor in the Department of Computer Science and Technology, East China Normal University. His research interests include green and reliable embedded computing, cyber-physical systems, parallel and distributed systems, and cloud computing. He serves as a regional editor of the *Journal of Circuits, Systems, and Computers* since 2012. He also served as guest editors of several special sections of the *IEEE Transactions on Industrial Informatics* and the *ACM Transactions on Embedded Computing Systems*. He is a member of the IEEE.