2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems

Efficient Two-Phase Approaches for Branch-and-Bound Style Resource Constrained Scheduling

Mingsong Chen, Fan Gu, Lei Zhou, Geguang Pu* and Xiao Liu
Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
Email: {mschen,fgu,lzhou,ggpu,xliu}@sei.ecnu.edu.cn

***Abstract*—In high-level synthesis (HLS), the resource constrained scheduling (RCS) tries to explore a time-minimum schedule for low-level hardware implementations under specific resource constraints. To achieve such an optimal schedule quickly, branch-and-bound (B&B) approaches are widely investigated to prune the fruitless search space. However, due to the lack of approaches that can obtain a tight initial feasible schedule, RCS generally starts with an incompact search space, which is not time-efficient. This paper proposes an efficient two-phase approach, which can quickly shrink the search space using a smaller upper-bound for efficient B&B RCS search. The experimental results demonstrate that our approach can drastically reduce the overall RCS time.**

I. Introduction

High-Level Synthesis (HLS) enables the rapid generation of optimized RTL (Register-Transfer Level) designs from ESL (Electronic System Level) specifications [1], [2], [3]. In 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. As one of key tasks in HLS, scheduling assigns each operation of a DFG with a control step (c-step) which indicates the start execution time of the operation. This paper focuses on the HLS scheduling under resource constraints, called Resource Constrained Scheduling (RCS). Given a DFG and a pre-defined set of functional or non-functional resources (e.g., function units, power, area and etc.) with specified overheads, RCS tries to find a schedule of operations with minimum overall c-steps.

Essentially, RCS is an NP-Complete scheduling problem with constraints of computation precedence and resource limits. To avoid forcefully enumerating all possible schedules, many approaches [4], [13] are proposed to reduce the optimal schedule derivation time. The basic idea is to remove infeasible or inferior schedules during the HLS search as much as possible. The B&B RCS methods [4] 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 of the optimal schedule 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 in turn results in a huge search space. Furthermore, current B&B approaches search the state space in a recursive manner. When the remaining operations cannot be used to derive an optimal schedule, the loose dispatch range of operations and deep recursive search can easily cause the *stuck-at-local-search* problem, which is the main reason for the long search time.

To avoid the stuck-at-local-search problem and achieve a tight initial feasible schedule, we propose several promising partial-search heuristics which can narrow down the search space as well as escape easily from the stuck-at-local-search. By coarsely exploring the state space, our partial-search heuristics can obtain a tight upper-bound of the schedule length with small overhead. Based on the results of the partial-search heuristics, we present a two-phase approach which can drastically reduce the overall RCS search time.

This paper is organized as follows. Section II introduces the related works of HLS scheduling. Section III presents the preliminary RCS notations. Besides introducing our partial-search techniques in detail, Section IV proposes our two-phase B&B RCS framework. Section V compares our approach to the state-of-the-art B&B approach. Finally, Section VI concludes the paper.

II. Related Works

Unlike non-optimal HLS heuristics (e.g., list scheduling [4]), this paper focuses on how to quickly achieve optimal schedules for RCS. To improve the performance of HLS scheduling, various heuristic approaches were proposed. One commonly used method in HLS scheduling is the *Integer Linear Programming* (ILP) model [5]. However, the number of variables in ILP model increases very fast with the size of DFGs. Therefore, solving tight resource constraint problems with ILP models may need extremely large amount of time.

The *execution interval analysis* approaches were widely studied to scale down the complexity of HLS scheduling. The basic idea is to narrow down the dispatching time estimation for each functional operation, thus it can reduce the overall scheduling search space. Based on the execution interval analysis, Shen and Jong [6] proposed a stepwise refinement algorithm for resource estimation. It can produce a tight bound in a reasonable computation time. However, this method can only achieve a near-optimal solution rather than an optimal one. To further reduce the complexity when using execution interval analysis, various B&B approaches were proposed. In [4], Narasimhan and Ramanujam gave an efficient B&B algorithm BULB based on both lower-bound and upper-bound information. Hansen and Singh [10] presented an efficient B&B approach to reduce the scheduling time for asynchronous systems under multi-resource constraints. Chen et al. [11] proposed an efficient pruning approach based on the structural information of schedules. A parallel version of BULB approach was also proposed to improve the RCS time [12]. Although B&B approaches are promising in pruning inferior schedules, most of them adopts the heuristics such as list scheduling to achieve a feasible schedule first, which cannot always guarantee a tight initial search space, thus it can easily cause a long search time. Target to reducing the overall RCS time, this paper tries to reduce the upper-bound length estimation of the initial feasible schedule.

III. Preliminary Knowledge

HLS scheduling employs DFGs to describe its behavior. A DFG is a DAG (Directed Acyclic Graph) $G = (V, E)$, where V is a set of vertices (nodes) designating functional operations with different types, and E is a set of directed edges describing operation dependencies. For any two nodes $v_i, v_j \in V$, $\langle v_i,\ v_j\rangle \in E$ means that the operation of v_i must be complete before the start of the operation of v_j. As the example in Figure 1, the DFG consists of 5 nodes and 5 directed edges. In an HLS DFG, each v_i is bound to an operation

This work was supported by NSF of China (61202103, 91118007), Innovation Program of Shanghai Municipal Education Commission (14ZZ047), Open Project of SW/HW Co-design Engineering Research Center of MoE (2013001), and Shanghai Knowledge Service Platform Project (ZF1213).

* Corresponding author.

op_i, where $type(op_i)$ indicates the functional unit type occupied by op_i and $delay(op_i)$ is used to denote time delay of op_i. An operation without any predecessors is an *input operation*, and an operation without any successors is an *output operation*.

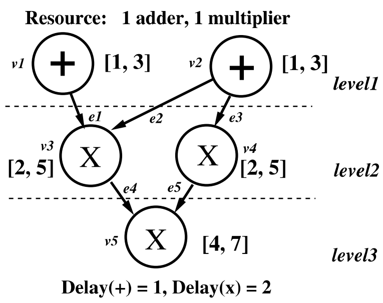

Fig. 1. An example of an HLS DFG

Various graph theory notations are used to enable the HLS scheduling analysis. In this paper, we use $G' = (V', E')$ to represent a *sub-graph* of $G = (V, E)$ if both $V' \subseteq V$ and $E' \subseteq E$. The sub-graph including nodes v_i and all its direct and indirect predecessors is denoted by $G_{pre}(v_i)$. The sub-graph with v_i as its source node is denoted as $G(v_i)$. A *path* is a sequence of nodes, which 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 the *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 $CP_w(G_{pre}(v_i))$. Operations with smaller $CP_w(G_{pre}(v_i))$ will be dispatched earlier. 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 $op_1, op_2, op_4, op_3, op_5$ is the dispatching order of operations in Figure 1, we can get $L_s(op_3) = L_s(op_4) = 4$ and $L_e(op_3) = L_e(op_4) = 3$, since op_4 is the first dispatched operation and op_3 is the last dispatched operation in level 2.

In RCS, the interval $[ASAP(op_i), ALAP(op_i)]$ is used to denote lower and upper bounds of the start c-step of operation op_i. To achieve a better RCS performance, it is required that the interval needs 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

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

And $ALAP_G(op_i)$ 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

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

Note that $le(S)$ has to be determined before calculating the *ALAPs* of operations. As an efficient method, the list scheduling algorithm [4] can achieve such a feasible schedule quickly.

In HLS, the *c-step* is the basic time unit. An operation will occupy a specific number of continuous c-steps for execution on corresponding function unit during the scheduling. As described in 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) gives the precedence constraint posed by the given DFG, and condition (2) indicates the resource constraints during the scheduling of DFG operations at any time. 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 the state space has been explored.

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, $\Sigma = \{\pi_1, ..., \pi_M\}$, and there are $num(\pi_i)$ units of π_i $(1 \leq i \leq M)$. 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 $S(op_i) + delay(op_i) \leq S(op_j)$ holds.

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

In Definition 3.2, 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 $(op_i, S(op_i))$ to denote the scheduling pair for operation op_i. The binary relation $\{(op_1, 1), (op_2, 2), (op_3, 3), (op_4, 5), (op_5, 7)\}$ is a feasible schedule with length 8 for the DFG in Figure 1. The binary relation $\{(op_1, 2), (op_2, 1), (op_3, 4), (op_4, 2), (op_5, 6)\}$ is an optimal schedule with length 7.

IV. Our Two-Phase Approach

A. Motivation

Besides $[ASAP, ALAP]$ intervals which restrict the search range of operations, B&B approaches [4], [10] use two other important data structures to prune inferior schedules: i) S_{bsf} which keeps the best schedule searched so far, and ii) S which indicates current schedule with unscheduled operations.

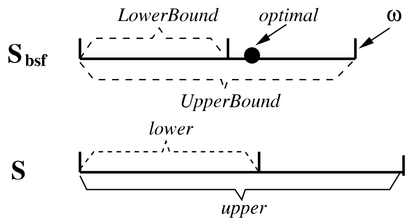

Fig. 2. Pruning scenario in B&B approach

Figure 2 presents the scenario which shows how the B&B search makes the pruning. Since it is impossible to determine the optimal schedule length before all the enumerations are done, to restrict search space, the upper-bound and lower-bound lengths of the optimal schedule are estimated. We use ω to record the upper-bound length of S_{bsf} (i.e., $UpperBound$). Initially, ω equals 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 HLS scheduling exploration. *LowerBound* is the lower-bound length of S_{bsf}, which is calculated using the approach proposed in [7]. In Figure 2, *optimal* indicates the length of the global optimal schedule, which is in the range of $[LowerBound, \omega]$. The current schedule S, which is an incomplete enumeration, also has two bound estimations. We use *lower* and *upper* to denote the lower- and upper-bound of schedule lengths respectively based on the scheduled operations of S.

In B&B approaches, if *lower* is larger than ω, the scheduling for the unexplored operations can be terminated, since *optimal* should be in the range $[LowerBound, \omega]$. We say that "the current schedule can be pruned". However, if *upper* is smaller than ω, it indicates that a schedule better than S_{bsf} is found. In other words, S_{bsf} will be replaced by the new schedule for further pruning. If ω equals *LowerBound*, the whole B&B search can be terminated. Based on the above discussion, we can find that ω plays an important role in determining the performance of HLS scheduling. Before the scheduling starts, its value equals to the length of a feasible schedule. During the search, ω indicates the length of a best feasible schedule explored so far. A wise use of ω can not only tighten the initial [*ASAP*, *ALAP*] interval which in turn compact the search space of operations, but also can reduce the chance of stuck-at-local-search and accelerate the pruning of the inferior schedules.

In most B&B approaches, the initial ω value is calculated based on the list scheduling approach, which often fails to get a tight value for ω. Based on Definition 3.1, the estimation of *ALAP* is derived from the initial ω value. 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 $\omega > \omega'$. The search space SS corresponding to ω is much larger than SS' corresponding to ω′. Since SS is larger than SS', the chance of vain search in S is higher. Consequently, the reduction rate of ω value is slower. Furthermore, B&B approach counts all the DFG node intervals in the recursive HLS searching, and the search space is huge in general. If the ω value drops slowly, the search space will shrink slowly, which will easily result it deep recursive procedure calls. In other words, a large initial search space can disable the pruning efficiency due to the slow ω convergence to optimal value. Therefore, how to quickly get a tight initial search space determines the HLS scheduling performance.

B. Two-Phase Search Space Reduction

In RCS, ω plays an important role in determining the pruning efficiency. The smaller the ω is, the faster the search is. However, it is hard to achieve a tightest ω before the real scheduling starts. Inspired by the observation in Section IV-A, we propose a two-phase approach shown in Figure 3 which aims to improve the overall searching performance.

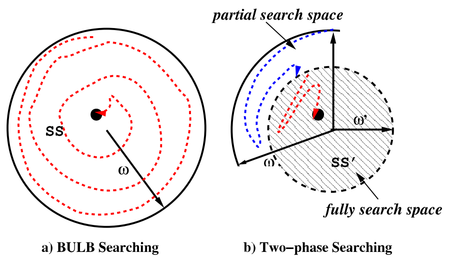

Fig. 3. Comparison between B&B and 2P approaches

Figure 3a) shows the searching using the classical B&B approach algorithm. To improve the initial search space, we partition the B&B approach algorithm into two phases: *partial-search* phase and *full-search* phase. Partial-search indicates that the search tries to explore a small part of the search space, and the full-search does the same job as the original B&B approach. As shown in Figure 3b), partial-search using blue dashed arrow line tries to quickly figure out a better initial schedule (i.e., the schedule with length ω′). And the following full-search uses the red dashed arrow line.

Figure 4 analyzes the time performance of our two-phase approach. For a given RCS problem, assume that B&B approach needs time T to get a global optimal solution. If we can find some partial-search heuristic that can coarsely search on a reduced search space or backtrack in a non-chronological way, due to the reduction of stuck-at-local-search scenarios, the time cost T_1 of the partial-search is generally smaller than T, i.e., $T_1 < T$. If a smaller ω is found during the partial-search, the time cost T_2 of full-search will be drastically reduced, i.e., $T_2 << T$. Consequently, we may have $T_1 + T_2 < T$. On the other hand, if the partial-search fails to find a better ω, T_2 will be approximate to T. Consequently the overall cost $T_1 + T_2$ will be worse than the cost of B&B approach algorithm (i.e., T). Therefore it is required that T_1 should be as small as possible. The less overhead spending in the partial-search phase, the better the overall performance we can achieve.

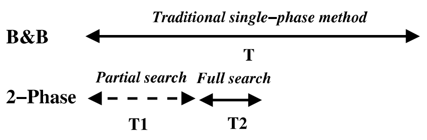

Fig. 4. Performance analysis of our 2P approach

C. Partial-search Heuristics

Since the partial-search time T_1 has a dominant effect on the overall performance, it should be as small as possible. Therefore, how to quickly find a small ω in partial-search phase is becoming a key problem in B&B search. By our observation, for the B&B search, the long-time search is mainly caused by the lack of pruning chances. When $\omega \in [lower, upper]$, the remaining unscheduled operations need to be investigated. Especially when the search goes deep down, the backtrack will become difficult. In other words, stuck-at-local-search needs a long time to hit a better schedule. To ease the escape from the stuck-at-local-search, this subsection will introduce three kinds of partial-search heuristics which can quickly find a tight upper-bound length for the following full-search.

1) Bounded Operations: To enable the efficient partial-search, it is required that a schedule whose upper-bound estimation $upper < \omega$ can be found quickly. However, due to the stuck-at-local-search, the value of ω cannot be reduced quickly. The delay of the ω update will cause the long search time. 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. Moreover, in B&B approach algorithm, the checking of operation dependence and resource allocation (i.e., *Precedence* and *ResAvailable* in Algorithm 2) in each iteration is very costly. If such checking can be optimized, T_1 can be further reduced.

Based on the above observation, our bounded operation based partial-search (B.O.) tries to avoid the deep recursive search by limited the number of the operations enumerated in the recursive search. It does not mean that we do not consider all operations for the scheduling. In our approach, the unenumerated operations will be estimated in an incomplete way. We adopt the list scheduling method to find a feasible schedule based on the enumeration of the bounded operations. 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 smaller than the original complete search.

As an example shown in Figure 1, assume that the operations are dispatched in the order op_1, op_2, op_3, op_4, and op_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 7 c-steps. This happens only when the dispatch time of op_1 equals to 2. It means that, when the c-step of op_1 does not equal to 2, the search is unfruitful. Even if the B&B pruning is used, it needs quite a long time when $S(op_1)$ equals to 2. In our approach, we only incompletely search the bounded operations, i.e., op_1 and op_2. The estimation of the upper-bound schedule length can be quickly reduced to 7 when op_1 dispatched at *c-step* 2. The whole partial-search can be terminated quickly as well. In the following full-search phase, the initial feasible schedule with an upper-bound 7 can make the tightest search space, and the full-search performance on this space can be improve drastically.

2) Non-Chronological Backtrack: In B&B approaches, 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 the order op_1, op_2, op_3, op_4, and op_5 in the example shown in Figure 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 length of S' equals 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 search is stuck-at-local-search, 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 during the recursive search. 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 has 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 the schedule $\{(op_1,1), (op_2,2), (op_3,3), (op_4,5), (op_5,7)\}$ is a feasible schedule in Figure 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.

3) Search Space Speculation: The operation dispatching interval $[ASAP,ALAP]$ plays an important role during the search space exploration. Although the methods defined in Section III 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 interval for an undispatched operation is large, then the search time will be intolerant. Adversely, if the search range of some operation can be reduced to a half, the overall search space will be reduced half too. Based on this observation, our search space speculation based partial-search (S.S.) tries to speculate better schedules by halving the search range of each operation on-the-fly. By adopting a greedy strategy, our speculation approach assumes that the global optimal result will be always located in the first half of the range of the current dispatching operation.

Our speculation process is as follows. During the B&B search, the ASAP value of each operation can be changed. As an example in Figure 1, assume that $S' = \{(op_1,1), (op_2,2)\}$ is the current incomplete schedule. Since the dispatching of op_3, op_4 and op_5 depends on the execution of op_2, their ASAP values can be updated to be 3, 3, 5 respectively. Assume that op_1 is dispatched first. Since op_1 is the current dispatching operation, during the recursive search, we restrict its search range to be the first half of the range $[1,3]$, i.e., $[1,2]$. Similarly, the dispatch of the second operation (i.e.,op_2) will be within the range $[1,2]$. Since $op2$ finishes at c-step 2, the execution range of op_3 will be changed to $[3,5]$. During the partial-search, we will only investigate its first half of the range, i.e., $[3,4]$. Since the search range will be always halved, the overall search work will be much smaller than the complete B&B search. If one shorter schedule can be hit, it will be beneficial to the overall B&B search.

D. Our Two-Phase Scheduling Approach

Algorithm 1 describes the skeleton of our partial-search based B&B algorithm. It has three inputs: i) an RCS DFG whose operations are sorted using the approach described in Section III, ii) resource constraints for operations, and iii) the partial-search strategy. In Algorithm 1, step 1 computes the initial *ASAP* values for each operation of D based on the Definition 3.1. Step 2 tries to achieve an initial feasible scheduling S by applying the list scheduling method. Step 3 uses the reduced length of S to update the initial *ALAP* values. Steps 4 conducts the partial-search. Steps 5-6 deal with the full-search on the compact search space based on the reduced ALAP values. It is important to note that both partial-search and full-search use the same procedure *BBM*. Finally the algorithm reports one global optimal scheduling for D under constraints C.

Algorithm 1: Two-Phase RCS Exploration

```
Input: i) An RCS DFG $D = (V,E)$, where $V$ has been sorted;
       ii) Resource constraints for all kind of operations, $C$;
       iii) The partial-search strategy type $T$.
Output: An optimal scheduling $S_{bsf}$ for $D$ and its length
TwoPhase($D$, $C$, $T$) begin
    1. $ComputeInitialASAP(D)$ ;
    2. ($S$, $\omega$) = $ListScheduling(D,op_1)$ ;
    3. $ComputeInitialALAP(D,le(S))$;
    /*partial-search*/
    4. $(S',le(S')) = BBM(D,C,T,(S,le(S)),1,true)$;
    /*full-search*/
    5. $UpdateALAP(D,le(S'))$;
    6. $(S_{bsf},le(S_{bsf})) = BBM(D,C,T,(S',le(S')),1,false)$;
    Return $(S_{bsf},le(S_{bsf}))$.
end
```

Algorithm 2 presents our B&B approach for both partial-search and full-search. It is important to note that the function *ResAvaible* can be used for checking the availability of various kinds of resources (function units, power, area, etc.). Therefore our approach can be applied for both functional and non-functional resources.

In Algorithm 2, steps 1 and 2 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 step 1. Steps 3 and 4 deal with the search space speculation based partial-search. If the partial-search is applied, step 3 will half the range of the current searching operation. Since our approach adopts the dynamic ASAP update, we need to save the original ASAP values of all operations of $G(op_i)$ first in step 5. Steps 6 and 7 calculate the *lower* and *upper* for current schedule. If *upper* is smaller than ω, then ω and S_{bsf} will be updated in steps 8 and 9. Changing ω value will trigger the checking of early termination condition in step 10 followed by the runtime space shrinking (step 11). If op_i is the last dispatched operation of some level and the level check condition holds, when the non-chronological backtrack is enabled, steps 12 and 13 will backtrack to the first dispatched operation in the same level. If the lower bound of current schedule (i.e., *lower*) is smaller than ω, the current operation will be scheduled. After the new c-step assignment of operation op_i in step 14, step 15 updates all the ASAP values of the operations in $G(op_i)$. Step 16 takes resources required by the operation op_i. Then op_{i+1} is processed recursively in step 17. When the search backtracks, the resource occupied by op_{i+1} is released in step 18. Steps 19 and 20 check whether the non-chronological backtrack has finished or not. When the search of $S(op_i)$ is done, step 21 restores the ASAP values of $G(op_i)$ saved in step 5. Finally, step 22 reports the results when the recursive search is complete.

Algorithm 2: Our B&B Pruning Algorithm

```
Input: i) An RCS DFG $D = (V,E)$, where $V$ has been sorted;
       ii) $C$, resource constraints for operations;
       iii) $T$, partial-search strategy;
       iv) $(S_{bsf}, \omega)$ = a feasible schedule and its length;
       v) $i$, index of the current operation in search;
       vi) $PS$ indicates if partial-search should be conducted.
Output: A schedule and its length
BBM($D$, $C$, $T$, ($S_{bsf}$, $\omega$), i, $PS$) begin
    if $PS$ & $T$==$B.O.$ then
        1. $N = |input\ operations\ of\ D|$;
    else
        2. $N = |V|$;
    end
    if $i \leq N$ then
        if $PS$ & $T$==$S.S.$ then
            3. $ALAP = \lceil \frac{(ASAP(op_i)+ALAP(op_i))}{2} \rceil$;
        else
            4. $ALAP = ALAP(op_i)$;
        end
        5. $SaveASAP(D, op_i)$;
        for $step = ASAP(op_i)$ to $ALAP$ do
            if $Precedence(op_i) \wedge ResAvaible(step, res(op_i))$ then
                6. $lower = LBound(op_i)$;
                7. $upper = le(ListScheduling(D, op_i))$;
                if $upper < \omega$ then
                    8. $\omega = upper$;
                    9. $S_{bsf} = ListScheduling(D, op_i)$;
                    if $\omega == LowerBound(D)$ then
                        10. Terminate($S_{bsf}$, $\omega$);
                    end
                    11. $UpdateALAP(D, \omega)$;
                end
                if $PS$ & $T$==$N.C.$ & $i == L_e(op_i)$ & $i \neq L_s(op_i)$
                & $LevelCheck(op_i)$ then
                    12. $return\_id = L_s(op_i)$;
                    13. Return ($S_{bsf}$, $\omega$);
                end
                if $lower < \omega$ then
                    /* Dispatch the current operation */
                    14. $S(op_i) = step$;
                    15. $UpdateASAP(D, op_i)$;
                    16. $ResOccupy(step, type(op_i), delay(op_i))$;
                    17. $BBM((D, C, T, (S_{bsf}, le(S_{bsf})), i+1, PS)$;
                    18. $ResRestore(step, type(op_i), delay(op_i))$;
                    if $return\_id \neq -1 \& return\_id \neq i$ then
                        19. Return ($S_{bsf}$, $\omega$);
                    else
                        20. $return\_id = -1$;
                    end
                end
            end
        end
        21. $RestoreASAP(D, op_i)$;
    end
    22. Return ($S_{bsf}$, $\omega$).
end
```

V. Experimental Results

To evaluate the effectiveness of our approaches, we collected the benchmarks *ARFilter*, *Cosine1*, *Collapse*, and *Feedback* from the *MediaBench* benchmark [8], which is a standard DSP benchmark suite. And we got the benchmark *FDCT* from [14]. We implemented our approaches using the C programming language. For comparison, we generated and solved the ILP models for each benchmark items using IBM ILOG CPLEX CP Optimizer [9], which utilizes the non-naïve *branch-and-cut* for efficient ILP solving. All the experimental results were obtained on a Linux machine with Intel Xeon 3.3GHz processors and 8GB RAM.

In this experiment, we only consider the constraints of functional units and the non-functional constraints (i.e., power and area). Table I lists the corresponding settings for all different types of functional operations used in the experiment.

TABLE I
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/DIV	×/÷	2	20	40	40
MEM	LD/STR	1	15	15	20
Shift	<</>>	1	10	10	5
Other	...	1	10	10	10

Table II 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 constraints for the design. As an example, "2a, 3m" denotes that two adders and three multipliers are used for the given design. Due to the space limit, we do not give the number of other functional units. The third column gives the lower and upper bound estimations of the optimal schedule before the scheduling, and the fourth column presents the lengths of global optimal schedules achieved by the scheduling. The fifth column presents the ILP solving time using the CPLEX CP Optimizer. The sixth column presents the scheduling timing using the BULB approach [4]. The columns 7-9 present the results of our two-phase search using the bounded operation based partial-search. The column 7 presents the time spent in the partial-search. The column 8 indicates the tightest bound found during the partial-search. The column 9 gives the total time for the whole two-phase search. The columns 10-12 and columns 13-15 present the results using our non-chronological and space speculation partial-search heuristics respectively. Since most modern computers have multiple cores, to achieve the maximum speedup over BULB, we can run two-phase methods with different partial-search heuristics as well as BULB approach on different cores in parallel. If one approach stops first, then the whole RCS can be terminated. By using this strategy, we can achieve the maximum speedup (i.e. $\frac{column6}{Min(column9, column12, column15)}$) as shown in the last column.

It can be found that our approaches outperform both the state-of-the-art ILP and the BULB approaches. Especially when the initial length of the feasible schedule can be reduced by the partial-search, our approaches can drastically reduce the RCS time (i.e., 20000 times improvement over BULB in *Collapse* design with the constraint "2a,1m"). Generally, it is hard to distinguish which partial-search heuristic is the best, since the performance is mainly determined by the design itself. For example, the bounded operation method works best for the *FDCT* design, but the 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 RCS problems that the other heuristics cannot solve (e.g., *Feedback* with "4a,5m"). It is important to note that the tighter the initial bound can achieve, the shorter the full-search time will be.

TABLE II
RCS RESULTS UNDER FUNCTIONAL UNIT CONSTRAINTS

Design				CP	BULB	Bounded Operation			Non-Chronological			Space Speculation			Max
Name	a, m #	[L,U]	le.	[9]	[4]	T_1	ω'	T_{total}	T_1	ω'	T_{total}	T_1	ω'	T_{total}	Impr.
ARFilter	1, 3	[14,16]	16	TO	0.16	0.08	16	0.22	<0.01	16	0.16	0.10	16	0.25	1.00
	1, 4	[14,16]	16	TO	0.40	0.13	16	0.49	<0.01	16	0.40	0.27	16	0.62	1.00
	1, 5	[14,16]	16	TO	0.38	0.14	16	0.49	<0.01	16	0.38	0.26	16	0.62	1.00
	2, 3	[14,15]	15	1.40	0.01	<0.01	15	0.02	<0.01	15	0.01	<0.01	15	0.01	1.00
Collapse	2, 1	[22,23]	22	TO	TO	162.20	22	162.20	TO	22	TO	0.18	22	0.18	>2.00e4
	2, 2	[21,23]	NA	TO	TO	TO	NA	TO	TO	NA	TO	TO	NA	TO	NA
Cosine1	1, 2	[28,29]	28	TO	63.51	<0.01	28	<0.01	0.06	28	0.06	20.94	28	20.94	1.59e4
	2, 2	[20,23]	20	TO	377.58	15.66	20	15.66	0.03	22	330.23	21.70	20	21.70	24.11
	3, 3	[16,17]	16	TO	<0.01	<0.01	16	<0.01	<0.01	16	<0.01	<0.01	16	<0.01	1.00
FDCT	1, 2	[26,27]	26	TO	20.30	<0.01	26	<0.01	18.56	26	18.56	0.26	26	0.26	5.08e3
	2, 2	[18,22]	18	TO	113.92	0.07	18	0.07	19.87	18	19.87	0.65	18	0.65	1.63e3
	2, 3	[14,17]	14	TO	11.17	0.63	14	0.63	1.94	14	1.94	2.03	14	2.03	17.73
	2, 4	[13,15]	13	TO	2.49	0.03	13	0.03	0.23	13	0.23	3.76	13	3.76	83.00
	2, 5	[13,14]	13	TO	0.48	<0.01	13	<0.01	0.12	13	0.12	0.16	13	0.16	120.00
	3, 4	[11,13]	11	TO	0.34	0.21	11	0.21	0.03	11	0.03	0.14	11	0.14	11.33
	4, 4	[11,12]	11	TO	0.07	0.01	11	0.01	0.06	11	0.06	0.02	11	0.02	7.00
Feedback	4, 4	[13,14]	13	TO	85.74	77.63	13	77.63	77.84	13	77.84	18.38	13	18.38	4.66
	4, 5	[13,15]	13	TO	TO	TO	NA	TO	265.30	13	265.30	TO	NA	TO	>13.60
	5, 5	[13,14]	13	TO	2.72	2.48	13	2.48	2.45	13	2.45	0.58	13	0.58	4.70

* All scheduling time is measured in seconds. "TO" means that the scheduling time is larger than 3600 seconds. "NA" indicates that the result is not available.

For the *ARFilter* design, since the length of optimal schedule equals to the initial upper-bound, the performance is a little 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 small overhead. Note that, in the *Cosine1* design with "2a,2m", we can find that although the non-chronological method can find a tighter bound (i.e., 22), the full-search still needs a long time since the initial upper-bound is not the tightest (i.e., 20).

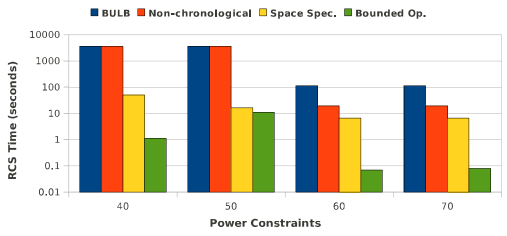

Fig. 5. RCS Results with an area of 100 Units

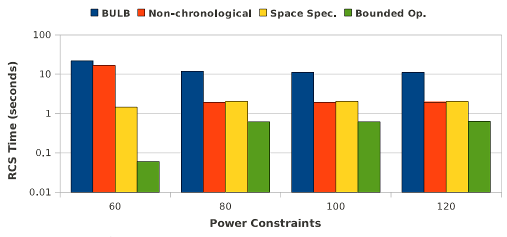

Fig. 6. RCS Results with an area of 140 Units

The power and area are two key non-functional constraints of the hardware design. Figure 5 and Figure 6 show the RCS results for *FDCT* design under different power and area constraints using different partial-search heuristics. Under the area constraints of 100 units and 140 units, we conduct the RCS with different power constraints (from 40 to 70 with an increment of 10 power units, and from 60 to 120 with an increment of 20 power units respectively). In Figure 5, when the area is 100 units and the power equals to 40 or 50 units, we cannot obtain the RCS results using the non-chronological backtrack based approach due to the timeout (i.e., 3600 seconds). Although we set 3600 seconds as the RCS time for these scenarios in Figure 5, in fact they cannot be compared with other approaches. Overall, from the above two figures, we can find that our two-phase approach using different partial-search heuristics outperforms the BULB approach significantly. It can achieve several orders of magnitude improvement compared to state-of-the-art BULB approach [4]. The bounded operation heuristic achieves the best performance in *FDCT* design, which is consistent with the results in Table II.

VI. CONCLUSION

This paper presents an efficient two-phase B&B approach that can quickly achieve optimal solution for resource-constrained scheduling problems. By adopting various partial-search heuristics with small overhead, the first phase of our approach targets to get a more accurate estimation of the upper-bound length for the optimal schedule. Due to the compact search space derived from the result of the first phase, the performance of the full-search in the second phase can be drastically improved. Consequently, the overall RCS time can be saved. Experimental results show that our method can achieve better performance than the state-of-the-art B&B method BULB by several orders of magnitude.

REFERENCES

[1] G. Martin and G. Smith, "High-level synthesis: past, present, and future", *Design & Test of Computers*, 26(4):18-25, 2009.
[2] P. Coussy, D. Gajski, M. Meredith and A. Takach, "An introduction to high-level synthesis", *Design & Test of Computers*, 26(4):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 TCAD*, 30(4):473-491, 2011.
[4] M. Narasimhan and J. Ramanujam, "A fast approach to computing exact solutions to the resource-constrained scheduling problem", *ACM TODAES*, 6(4):490-500, 2001.
[5] F. Su and K. Chakrabarty. High-level synthesis of digital microfluidic biochips. *ACM JETC*, 3(4), 2008.
[6] Z. Shen and C. Jong, "Lower bound estimation of hardware resources for scheduling in high-level synthesis", *Journal of Computer Science and Technology*, 17(6):718-730, 2002.
[7] G. Tiruvuri and M. Chung, "Estimation of lower bounds in scheduling algorithms for high-level synthesis", *ACM TODAES*, 3(2):162–180, 1998.
[8] Media Benchmarks. Available at: http://express.ece.ucsb.edu/benchmark/.
[9] IBM ILOG CPLEX CP Optimizer V12.3. Available at: http://www-01.ibm.com /software/commerce/optimization/cplex-cp-optimizer/index.html.
[10] J. Hansen and M. Singh, "A fast branch-and-bound approach to high-level synthesis of asynchronous systems", in *Proc. of ASYNC*, 107–116, 2010.
[11] M. Chen, S. Huang, G. Pu and P. Mishra, "Branch-and-bound style resource constrained scheduling using efficient structure-aware pruning", in *Proc. of ISVLSI*, 2013, accepted.
[12] 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. of CODES+ISSS*, 2013, accepted.
[13] C. Yu, Y. Wu and S. Wang, "An in-place search algorithm for the resource constrained scheduling problem during high-level synthesis", *ACM TODAES*, 15(4), 2010.
[14] H. Steve and B. Forrest, "Automata-based symbolic scheduling for looping DFGs", *IEEE Trans. on Computers*, 50(3):250–267, 2001.