Bound-Oriented Parallel Pruning Approaches for Efficient Resource Constrained Scheduling of High-Level Synthesis

Mingsong Chen, Lei Zhou, Geguang Pu and Jifeng He
Shanghai Key Laboratory of Trustworthy Computing
East China Normal Univeristy, Shanghai, China
{mschen,lzhou,ggpu,jifeng}@sei.ecnu.edu.cn

ABSTRACT

As a key step of high-level synthesis (HLS), resource constrained scheduling (RCS) tries to find an optimal schedule which can dispatch all the operations with minimum latency under specific resource constraints. Branch-and-bound heuristics are promising to achieve such an optimal schedule quickly, since they can prune away large parts of infeasible solution space during the exploration. However, few of them are based on the prevalent multi-core platforms. Based on the bound information, this paper exploits the parallel pruning potentials from different perspectives and proposes various efficient techniques that can substantially reduce the overall RCS search efforts. The experimental results demonstrate that our approach can reduce the RCS time drastically.

Categories and Subject Descriptors

B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*automatic synthesis, optimization*

General Terms

Algorithms, Performance

Keywords

Parallel Pruning, Branch-and-Bound, High-Level Synthesis, Resource Constrained Scheduling

1. INTRODUCTION

High-level synthesis (HLS) approaches [15] are being gradually adopted in more and more complex industrial hardware designs [20, 5]. It enables rapid generation of register transfer level (RTL) implementations from behavior descriptions, while taking into account performance, cost, area and power requirements [7].

HLS involves three major steps: scheduling, allocation and binding. For a given behavior-level description, it can be partitioned into a set of operations organized using the *Data Flow Graphs* (DFGs). Based on such intermediate representations, scheduling refers to the operation assignment to specific control steps (c-steps), where each c-step can be performed in a single clock cycle in the hardware. The allocation and binding associate the computation operations in the DFGs to corresponding hardware components, such as adders and registers. Generally, the scheduling problem is a key challenge in HLS, since complex designs need to make the trade-off among various constraints. To achieve an optimal solution, a huge number of possible designs need to be explored and evaluated, which is time-consuming. This paper only focuses on the HLS scheduling under the resource constraints, termed Resource Constrained Scheduling (RCS) [17, 10]. Given a DFG of a high-level description and a fixed number of hardware resources with specified delays, RCS tries to find a schedule with minimum latency to dispatch all the operations.

Essentially, RCS is an NP-Complete combinatorial problem with constraints of computation precedence and resource limits [1, 25]. Instead of enumerating all possibilities, various heuristics [17, 28] are proposed to efficiently prune infeasible or inferior schedule candidates. Branch-and-bound (B&B) approaches are [10, 17] promising in pruning the search space. By computing both the lower- and upper- bounds of the optimal schedule, the B&B methods can accurately estimate the possible c-steps of each operation, and the fruitless search space (i.e., the schedules whose lengths are larger than the upper-bound) can be discarded efficiently. Although B&B approaches are promising in pruning search space, they only investigate the upper-bound and lower-bound information on a single-core platform. Since more and more computers are supporting multi-core and many-core computation, the pruning efficiency can be improved further by utilizing the parallelism.

Assume that a given HLS scheduling problem needs time t to achieve an optimal solution. The search time on n identical cores can be reduced to t/n when the whole search can be fully parallelized. However, this reduction cannot be guaranteed for the B&B algorithms, since the upper-bound estimation of the optimal schedule changes during the scheduling which strongly affects the search performance. In B&B approaches, when a smaller upper-bound is achieved, the pruning rate will be accelerated. In other words, more inferior schedules will be identified and an optimal result will be achieved quickly. As an alternative, assume that the search space can be equally divided and assigned to n cores. If the pruning of B&B heuristic methods does not take effect in one of the parallel parts, the overall search will be delayed. In this case, the scheduling time could be much larger than t/n. Adversely, if some part can find better schedules quickly, and such information can be informed globally to other search tasks, the overall search performance could be much smaller than t/n. Therefore, designing an efficient parallel RCS approach needs to figure out the following two questions.

1. How to efficiently decompose an HLS scheduling task for parallel processing?
2. What kind of learning can be shared between parallel sub-tasks to enable faster convergence to an optimal schedule?

To address above two questions, this paper makes two major contributions based on the bound information: i) it proposes promising space partitioning and bound speculation approaches that can efficiently decompose the overall search task into a set of parallel sub-tasks; and ii) it proposes a cooperation framework that shares the minimum upper-bound information among parallel sub-tasks to enable efficient pruning.

978-1-4799-1417-3/13/$31.00 ©2013 IEEE

This paper is organized as follows. Section 2 introduces the related work of HLS scheduling. Section 3 presents the preliminary notations for HLS scheduling. Besides introducing our parallel techniques in detail, Section 4 proposes a cooperation framework that can share the bound information between sub search tasks. Section 5 presents the experimental results using our parallel pruning approaches. Finally, Section 6 concludes the paper.

2. RELATED WORKS

Unlike non-optimal HLS heuristic methods (e.g., list scheduling [17], force directed scheduling [19]) which can achieve a near-optimal scheduling with less overhead, this paper focuses on how to quickly obtain an optimal RCS result in HLS. As an early popular approach, *Integer Linear Programming* (ILP) models [12] are widely used in HLS scheduling. Gebotys and Elmasry [9] proposed efficient formulas that can reduce the execution time of the ILP method. In [23], Rim and Jain derived lower-bounds of operations using a relaxed ILP formulation together with a greedy algorithm. Langevin and Cerny [14] improved Rim and Jain's algorithm by adopting a fast recursive technique for estimating lower-bound performance of data path schedules. However, the number of variables in ILP models increases very fast with the size of DFGs. Consequently, solving tight resource constraint problems using ILP models may need an extremely large amount of time.

The *execution interval analysis* approach is another way to deal with the HLS scheduling. By relaxing precedence constraints in designs' behavioral descriptions, Timmer and Jess [26] proposed a unified approach of lower-bound functional area and cycle budget estimations under resource constraints. By calculating the minimal overlap among different execution intervals of operations, Sharma and Jain proposed an approach to estimate architecture resources and performance [24]. In [18], Ohm et al. presented a comprehensive technique for lower-bound estimation. In addition to functional resources, their cost model also takes storage resources into account. Tiruvuri and Chung [27] improved the method in [24] by efficiently estimating completion time of partial schedules.

To effectively avoid unnecessary search in RCS, Narasimhan and Ramanujam presented a B&B algorithm called BULB [17] using both lower-bound and upper-bound lengths of an optimal schedule to prune the search space. It can efficiently prune away the schedules which are longer than the upper-bound estimation in an early stage. In [10], Hansen and Singh proposed an efficient B&B approach to reduce the scheduling time under multi-resource constraints. To enhance the performance of B&B approaches, Chen et al. [3] proposed a level-bound method that can prune the fruitless search space which cannot be detected by traditional B&B approaches. In [28], Wu et al. presented a novel In-Place search algorithm based on a systematic children-generating algorithm. It only requires a constant storage space during the traversal of the search tree. However, so far, most RCS optimization methods only use a single-core to figure out the solution.

Parallelism is widely investigated in behavior synthesis. In [6], Cordone et al. presented a methodology that can extract inherent parallelism from the control and data flow graphs of a sequential program. To reach optimal solutions in a reasonable time, Cherroun and Feautrier [4] tried to exploit parallelism and locality of a program. They proposed a B&B algorithm, where each evaluation is accelerated by both maximal and greedy clique computation. However, most of them focused on the parallelism inside behavior specifications themselves rather than the HLS scheduling. Our approach proposed in this paper is based on the B&B style BULB approach [17] with additionally various parallel pruning techniques. Although parallel B&B approaches have been successfully adopted in many domains [8], few of them were considered in HLS. To the best of our knowledge, our approach is the first attempt to utilize both B&B and parallel pruning approaches based on the bound information to further reduce the RCS efforts.

3. PRELIMINARY KNOWLEDGE

3.1 Graph-based Notations of RCS Problem

HLS scheduling employs DFGs as its intermediate format. A DFG is a DAG (Directed Acyclic Graph) $G = (V, E)$, where V denotes a set of vertices (nodes) designating functional operations, and E is a set of directed edges describing operation dependencies between nodes. $G' = (V', E')$ is a *sub-graph* of $G = (V, E)$ if both $V' \subseteq V$ and $E' \subseteq E$. The sub-graph with v_i as its source node is denoted as $G(v_i)$. The sub-graph including nodes v_i and all its predecessors is denoted by $G_{pre}(v_i)$. In a DFG, the *length* of a path is the sum of operation delays along the path, while the delays are determined by the types of nodes. The path in G with the longest length is called its *critical path*. We use $CP(G)$ to denote the length of the critical path of G.

For RCS, DFGs are used to describe the dependencies between operations. Assume that there are M different types of functional units (e.g., adders and multipliers). We use $\Sigma = \{\pi_1, ..., \pi_M\}$ to denote the set of function types. The number of available functional unit resources of type π_j is denoted by $num(\pi_j)$. In an HLS DFG, each node v_i is assigned with an operation op_i, and $type(op_i) \in \Sigma$ indicates the type of functional unit that will be occupied by op_i. We use $delay(op_i)$ to denote time delay of op_i.

In HLS, the control step (*c-step*) is the basic time unit. An operation may occupy a fixed number of continuous c-steps for execution on corresponding function units. Like scheduling on task graphs, a *feasible scheduling* for a DFG tries to dispatch operations under the constraints of data dependence posed by the DFG and the limited resources posed by the implementation requirements. A schedule for a DFG is an assignment function S which dispatches each operation op_i at c-step $S(op_i) \in Z^+$. Let S be a feasible schedule. Its length $le(S)$ is the largest finished time of all the operations, i.e., $le(S) = \max\{S(op_i) + delay(op_i) - 1 \mid op_i \in V\}$.

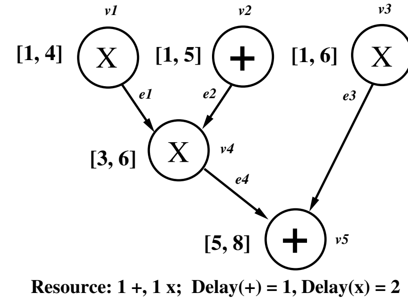

Figure 1: An example of an HLS DFG

The interval notation $[ASAP, ALAP]$ plays an important role in RCS. It denotes the earliest and latest possible start time for each operation respectively. Generally the narrower the interval $[ASAP, ALAP]$ is, the better RCS performance can be achieved. Before RCS, the initial interval $[ASAP, ALAP]$ for each operation op_i can be estimated as follows: $ASAP(op_i) = CP(G_{pre}(op_i)) - delay(op_i) + 1$, and $ALAP(op_i, le(S)) = le(S) - CP(G(op_i)) + 1$. It is important to note that a feasible schedule S needs to be obtained first before calculating the $ALAP$ time of operations. As an efficient method, the list scheduling algorithm [17] can be used to achieve such a feasible schedule. Since our approach only focuses on enumeration of operation c-steps within different $[ASAP, ALAP]$ inter-

vals, it can be applied on various designs, including chaining and pipelining problems [19].

As an example in Figure 1, the DFG consists of 5 nodes and 4 edges, and we assume that the resource of the target platform includes only one adder and one multiplication unit. In the DFG, there are two types of operations, i.e., addition (denoted by the symbol "+", with a delay of 1 c-step) and multiplication (denoted by the symbol "×", with a delay of 2 c-steps). The notation $[m,n]$ $1 \leq m \leq n$ indicates the execution interval for each operation. Let $(op_i, S(op_i))$ be the pair for the scheduling of operation op_i. The binary relation $\{(op_1,1), (op_2,3), (op_3,4), (op_4,6), (op_5,8)\}$ indicates a feasible schedule with length 8 for the DFG in Figure 1. The binary relation $\{(op_1,1), (op_2,1), (op_3,3), (op_4,5), (op_5,7)\}$ is one of the optimal schedules with length 7.

3.2 BULB Algorithm

The BULB algorithm proposed in [17] is a typical B&B heuristic, which can efficiently prune fruitless search space.

Algorithm 1: BULB Algorithm

```
Input: i) An HLS DFG D with resource constraints;
       ii) Operation set OP = {op_1, ..., op_N} in dispatching order;
       iii) S_bsf, which is a feasible schedule for D and its length is ω;
       iv) S, which stores the current incomplete schedule;
Output: An optimal schedule and its length for D
BULB(D, N, i, S, S_bsf, ω) begin
    if i ≤ N then
        for step = ASAP(op_i) to ALAP(op_i) do
            if Precedence(op_i) ∧ ResAvaible(step, type(op_i)) then
                1. lower = le(LBound(S));
                2. upper = le(UBound(S));
                if upper < ω then
                    3. ω = upper;
                    4. S_bsf = UBound(S);
                    if ω == globalLow then
                        5. Return (S_bsf, ω);
                    end
                    6. UpdateALAP();
                end
                if lower < ω then
                    /* Dispatch the current operation */
                    7. S(op_i) = step;
                    8. ResOccupy(step,type(op_i),delay(op_i));
                    9. BULB(D,N,i+1,S,S_bsf,ω);
                    10. ResRestore(step,type(op_i),delay(op_i));
                end
            end
        end
    end
    Return (S_bsf, ω).
end
```

Algorithm 1 presents the implementation of BULB algorithm in a recursive way. In this algorithm, S_{bsf} keeps the best feasible schedule searched so far with the length ω. S denotes the current incomplete search schedule; and *globalLow* indicates the lower-bound estimation of ω's length. Initially, S_{bsf} equals to a feasible schedule derived using the list scheduling approach. Before an operation can be dispatched, the precedence and resource constraints of the operation should be checked. In the BULB method, the procedure $Precedence(op_i)$ checks whether all precedents of operation op_i are finished already, and the procedure $ResAvailble(step,\ type)$ checks whether the resources required by op_i are available at the given c-step. If all such constraints hold, steps 1 and 2 try to schedule the undetermined operations in two different ways. $LBound(S)$ [27] dispatches the unscheduled operations by ignoring the resource constraints, thus it can be used to get the lower-bound length *lower* for S. By adopting the list scheduling method [17], $UBound(S)$ can obtain a feasible schedule for the undecided operations, thus it can be used to estimate the upper-bound length *upper* of S. If *upper* is smaller than ω, it means that $UBound(S)$ is better than S_{bsf}. Therefore, in steps 3 and 4, the S_{bsf} and ω will be updated. If the *upper* equals to *globalLow*, the whole BULB procedure will be returned in step 5 since an optimal schedule has been found. Otherwise, step 6 updates the *ALAP* for each operation, since ω becomes smaller. If *lower* is smaller than ω, steps 7-10 will dispatch a new operation, since it is impossible to determine the relation between S and S_{bsf}. Otherwise, if *lower* is not smaller than ω, the current schedule S can be pruned. Finally, the algorithm reports an optimal result.

It is important to note that the BULB approach can be easily extended to solve the scheduling with multiple kinds of constraints (e.g., area, energy, power, etc.) [10]. For example, when we want to incorporate power constraints in Figure 1, during the searching we only need to put a checker for the power availability in the procedure $ResAvaible(step, type(op_i)$ of Algorithm 1. Section 5.2 will present two examples of such non-functional constraints.

4. PARALLEL PRUNING APPROACHES

From Algorithm 1, we can find that there are two key factors that strongly affect the performance of B&B RCS approaches: i) the [*ASAP*, *ALAP*] intervals, which are lower-bound and upper-bound estimations of the start time of operations; and ii) the lower-bound length and upper-bound lengths (i.e., *globalLow* and ω in Algorithm 1) of the optimal schedule S_{bsf}. In fact, based on the definition, the *ALAPs* of operations rely on the value of $\omega = le(S_{bsf})$. If we can achieve a smaller ω, the size of the search space will be drastically reduced.

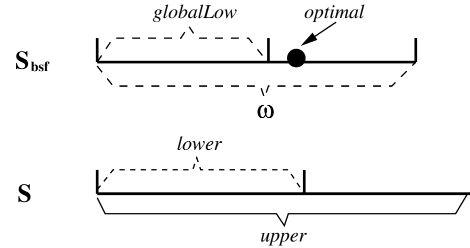

Figure 2: The analysis of pruning in B&B RCS

Figure 2 analyzes the pruning scenarios of the B&B RCS approaches. The notations S_{bsf} and S are the same as the ones defined in Algorithm 1. We use ω to indicate the upper-bound length of S_{bsf}. Initially, ω equals to the length of a feasible schedule. Then ω decreases dynamically when a shorter feasible schedule is found during the RCS exploration (i.e., steps 3 and 4 in Algorithm 1). *globalLow* is the lower-bound length of S_{bsf}, which is calculated using the approach proposed in [27]. Here, *optimal* indicates the position of a global optimal schedule, which is in the range of $[globalLow, \omega]$. The current schedule S, which is an incomplete enumeration, also has two bound estimations *lower* and *upper* (i.e., steps 1 and 2 in Algorithm 1). In BULB, if *lower* is larger than ω, the schedule S can be safely discarded, since *optimal* should be in the range $[globalLow, \omega]$. In the other hand, if *upper* is smaller than ω, a schedule which is better than S_{bsf} has been found. Consequently, the *ALAPs* of each operation will be updated, and the whole search space will be dynamically compacted [21].

From the above discussion, we can find that ω plays an important role in determining the performance of HLS scheduling. A wise use of ω can not only tighten the [*ASAP*, *ALAP*] intervals which in turn prunes the search space of operations, but also enable the fast pruning of inferior schedules during the RCS searching. However, existing B&B RCS approaches do not fully investigate the usage of ω. Consequently, loose estimations of ω can easily result in large

search space and long search time. Furthermore, current B&B RCS approaches only employ a single process to handle the "branch" and "bound" work, which is not time-efficient.

To quickly locate a tight upper-bound for ω, we developed two novel decomposition heuristics that can search the minimum ω value in parallel. We also developed a cooperation framework that can synchronize the smallest searched ω value so far among the sub-tasks. The following sub-sections will present them in detail.

4.1 Search Task Decomposition

To investigate the potential of the parallel search, this section presents two efficient approaches to decompose an entire search work into a set of sub search tasks and run them in parallel in order to achieve a better search performance.

4.1.1 Search Space Partitioning

During the search of optimal schedule using BULB approach [17], when the difference between the upper-bound and lower-bound of the schedule length (i.e., $|globalLow - \omega|$) is small, the RCS can be easily *stuck-at-local-search*, i.e., the RCS is trapped in the deep recursive search. This is because the less frequent update of the upper-bound length will result in less pruning opportunities.

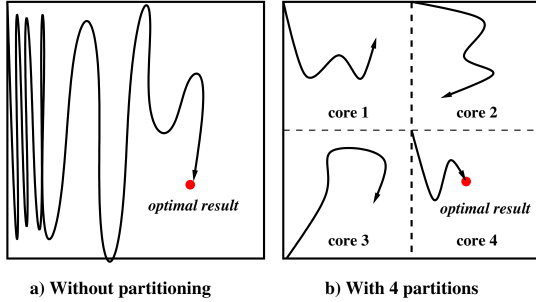

Figure 3: RCS search without and with partitioning

To overcome the *stuck-at-local-search* problem, we can divide the search space into several smaller parts. Figure 3 shows an example of RCS search with and without the space partitioning. Assume that the length of an optimal schedule equals to *globalLow*; the search time in Figure 3a) is T_l; and the search time in Figure 3b) is T_r. Without partitioning, the RCS search gets stuck at the left part of the search space because of no reduction of the ω value. Consequently, the search time T_l can be extremely long, since all the possible schedules should be enumerated. However, if we can divide the search space into 4 small disjoint pieces and check each of them using a different CPU core, the effect of local search can be drastically relieved. Assuming that we can get a new ω' such that $\omega' = globalLow$ at time T_r in the 4_{th} partition, the whole RCS search can be terminated since an optimal schedule has been found. If the search of the small partition terminates quickly, the overall search can be benefited and the search time can be drastically reduced. In other words, we can get $T_r << T_l/4$. However, before the scheduling starts, it is hard to obtain a tight lower-bound estimation for the optimal schedule S_{bsf} such that the length of S_{bsf} (i.e., $le(S_{bsf})$) is equal to *globalLow*. If $le(S_{bsf}) == globalLow$ does not hold, the whole search should wait for the termination of all the sub search tasks.

In our search space partitioning approach, we only divide the operations which are dispatched at the early stage of the search. In our approach, the operations are dispatched in a non-ascending order according to the length of the critical paths. As an example shown in Figure 1, the operations are dispatched in the order $<v_1, v_2, v_4, v_3, v_5>$, since $CP(G(v_1)) = 5$, $CP(G(v_2)) = 4$, $CP(G(v_4)) = 3$, $CP(G(v_3)) = 3$, and $CP(G(v_5)) = 1$. The $[ASAP, ALAP]$ values of each operations are as follows: $v1 : [1,4]$, $v2 : [1,5]$, $v3 : [1,6]$, $v4 : [3,6]$, $v5 : [5,8]$. Our approach halves the $[ASAP, ALAP]$ intervals of the investigated operations. If there are k operations involved in the partitioning, there will be 2^k partitions generated during the search. For example, if we handle the operations v_1 and v_2 only in Figure 1, there are four partitions generated as follows:

1. $v1 : [1,2]$, $v2 : [1,3]$, $v3 : [1,6]$, $v4 : [3,6]$, $v5 : [5,8]$
2. $v1 : [1,2]$, $v2 : [4,5]$, $v3 : [1,6]$, $v4 : [3,6]$, $v5 : [5,8]$
3. $v1 : [3,4]$, $v2 : [1,3]$, $v3 : [1,6]$, $v4 : [3,6]$, $v5 : [5,8]$
4. $v1 : [3,4]$, $v2 : [4,5]$, $v3 : [1,6]$, $v4 : [3,6]$, $v5 : [5,8]$

Although most present CPUs contain 4-8 cores, it does not require that the search space should be divided into 4 or 8 partitions. If the number of partitions is small (e.g., equals to the core number), during the practical search, we can often find that some cores are idle since they finish their sub search tasks earlier. This case is a waste of the computation power. To fully utilize the advantage of parallelization as well as avoid the *stuck-at-local-search* further, we can divide the search space into more partitions. Thanks to the capability of state-of-the-art parallel schedulers (e.g., OpenMP [2]), the parallel tasks can be dynamically scheduled on the limited cores with consideration of the load balance. Generally, if there are more operations involved in the partitioning, the search performance can be further improved.

4.1.2 Static Upper-Bound Length Speculation

The estimation of the upper-bound length of the S_{bsf} plays a key role in the RCS search. When using the BULB method on a single-core platform, it is difficult to achieve the "tightest" estimation for the length of S_{bsf}. For multi-core platforms, the situation is different. In this case, we can speculate the upper-bound length of the S_{bsf} to quickly achieve the optimal solution.

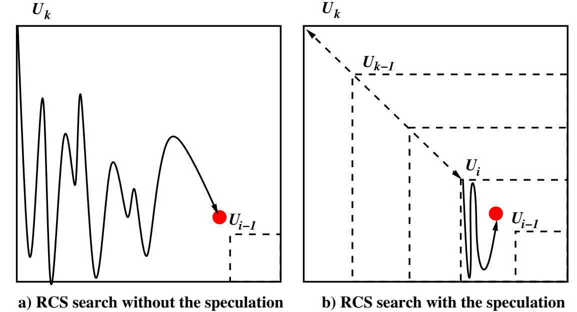

Figure 4: An illustration of upper-bound length speculation

Before the scheduling starts, we can get an interval $[globalLow, \omega]$ which is an estimation of the S_{bsf}'s length. Initially, the value of ω equals to the length of a feasible schedule. In the context of multi-core computation, we can speculate the length of S_{bsf} in the range of $[globalLow, \omega]$. The basic idea of our speculation method is to divide the interval $[globalLow, \omega]$ into K sub-intervals $[globalLow, U_1]$, $[globalLow, U_2]$, ..., $[globalLow, U_K]$ where $U_1 < U_2 < \ldots U_K$, $U_1 = globalLow$ and $U_K = \omega$. Assume that the K sub search tasks run on K cores in parallel with speculation. Due to the different initial estimation of the upper-bound length of S_{bsf}, the size of the search space on each core is different. If there are enough cores, we can guarantee that there exists one core that runs with the initial ω that equals to the length of

the optimal schedule. As an example shown in Figure 4, it assumes that $U_{i-1} \le le(S_{bsf}) \le U_i$. Generally the smaller the upper-bound length can be achieved, the more space will be pruned during the search. Therefore, the core running with the estimation $[globalLow, U_i]$ will quickly achieve the optimal solution.

In our speculation method, the number of the cores determines the performance of the overall search. Assume that we have n cores and the range of estimated $le(S_{bsf})$ is $[globalLow, \omega]$, we uniformly divide the range into segments with a size of $segment = \lceil (\omega - globalLow + 1)/(n-1) \rceil$. For each CPU core, we assign an index starting from 1 to n. On the i_{th} core, we estimate its upper-bound length within the range $[globalLow, MIN(globalLow + (i-1) \times segment, \omega)]$.

Unlike the partitioning method proposed in Section 4.1.1, the termination of the speculation method only needs one of the following conditions to be satisfied:

1. The condition $\omega == globalLow$ holds on some core.
2. The i_{th} core finishes the searching and there is a feasible solution found in $[globalLow, U_i]$. And all the searches on $[globalLow, U_j]$ $(j < i)$ have been completed and none of them find a possible solution smaller than U_j.

In the speculation method, the CPU cores compete with each other to achieve the best solution. If some core has won the search because one of the above conditions holds, all the other running sub search tasks will be terminated instantly.

4.1.3 A Hybrid Approach

The partitioning approach can reduce the chance of the *stuck-at-local-search*. However, when the global optimal schedule length is not equal to *globalLow*, all the partitions need to be checked. In this case, the performance of the partitioning approach is determined by the search time of the worst partitions (i.e., partitions with least pruning). To reduce the search time for each partition, the speculation method can be applied. Due to the orthogonality of the pruning effects between the partitioning method and the static upper-bound speculation, they can be combined as a hybrid approach to further reduce the searching time.

Assume that we have M ($M = 2^k$, where k is the number of operations involved in partitioning) partitions, and have N upper-bound speculations in the range of $[globalLow, \omega]$, then the hybrid approach will derive $M * N$ sub-tasks. Assume that $U_1 < U_2 < \ldots U_N$ are N upper-bound speculations generated using the approach in Section 4.1.2, where $U_1 = globalLow$ and $U_N = \omega$. Since the partition with the smallest upper-bound estimation can be checked using least time, we adopt the following strategy to check all the sub-tasks. In total there are N iterations of the search. In the i_{th} iteration, we check all the partitions with the speculated bound U_i. If some better schedule in the i_{th} iteration has been found, then the following iterations will be ignored. Otherwise, if no better schedules are found in the i_{th} iteration, then the *globalLow* will be safely set to be U_i for the following iterations. Note that the termination condition of the hybrid method is the same as the partitioning approach. Since the number of the sub-tasks using hybrid method is larger than the partitioning approach for the same partitions, it can strongly affect the overall search performance. To restrict the number of the generated sub-tasks in hybrid method, our approach allows at most four speculations in the range $[globalLow, \omega]$. The following formula shows the number of speculations when we adopt the static speculation strategy on partitions.

$$\#\,of\ speculations = \begin{cases} 1 & globalLow == \omega \\ \omega - globalLow + 1 & \omega - globalLow < 4 \\ 4 & \omega - globalLow \ge 4 \end{cases}$$

4.2 Parallel Search Task Cooperation

The search space partitioning approach divides the $[ASAP, ALAP]$ intervals to avoid the fierce local search. The bound speculation method tries to achieve the tightest initial ω for the optimal schedule searching, which can lead to a compact initial search space. Both parallel heuristic methods have a strong relation with the estimation of the upper-bound length of the S_{bsf} (i.e., ω). However, both methods assume that every sub-task manages its ω independently and searches the optimal solution without knowing the progress of other sub search tasks, which is not time-efficient. In this sub-section, we propose a cooperation framework which can exchange useful bound information (i.e., global smallest ω so far) among sub search tasks. By sharing such information, the overall search performance can be further reduced.

4.2.1 Minimum ω Synchronization

Assume that we have k cores to run the RCS search using the partitioning or the speculation method. During the search, the progress of the sub search tasks can be different. If one task explores a new schedule which has the smallest ω among all the sub-tasks, such information can be propagated to other sub search tasks to reduce their search space dynamically. This is because that the *ALAP* values of operations are determined by the ω. If the ω can be reduced on-the-fly, the search space can be dynamically shrunk [21].

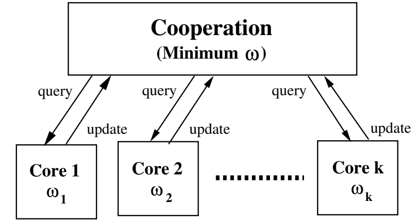

Figure 5: A framework for minimum ω synchronization

Based on this observation, we developed a cooperation framework that supports the *minimum* ω *synchronization* among sub search tasks. Figure 5 shows the overview of our implementation. The cooperation unit maintains the global minimum ω value. It supports the query and update operations of the global minimum ω value.

Algorithm 2: Update and Query Operations of Cooperation

```
Update(coreID, ω') begin
    omp_set_lock(&lock);
    if ω' < global_min then
        winner = coreID;
        global_min = ω';
    end
    omp_unset_lock(&lock);
end
QueryT() begin
    Return global_min;
end
QueryW() begin
    Return winner;
end
```

Algorithm 2 presents the details of the update and query operations for the cooperation unit in our framework. In this algorithm, all the sub search tasks compete with each other to be the winner of the search. The cooperation unit keeps both the current winner (denoted by *winner*) and globally minimum feasible schedule length (denoted by *global_min*). To achieve minimum ω synchronization, all the sub search tasks always monitor the change of the global minimum ω value by using the procedures *QueryT* and *QueryW*, and dynamically shrink the search space accordingly. When one sub task finds a schedule with a smaller length than the winner

schedule's, it will replace the current winner by invoking the *Update* procedure. To avoid the race condition when updating the *winner* and *global_min* information, we adopt the locking mechanism in the *Update* procedure.

4.2.2 Dynamic Upper-Bound Length Speculation

Section 4.1.2 presented a static speculation method for the upper-bound length. This method speculates upper-bound lengths of schedules at the beginning of the search. However, without collaboration between sub search tasks, the performance of the overall search using static upper-bound length speculation only depends on the i_{th} sub-task with the minimum initial range $[globalLow, \omega_i]$ where $\omega_{i-1} \leq le(S_{bsf}) \leq \omega_i$. Even if the minimum ω synchronization is allowed between sub search tasks, this statement still holds.

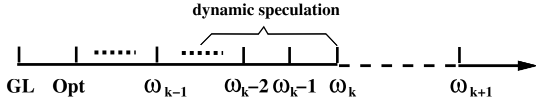

Figure 6: A scenario where dynamic speculation happens

In fact, during the search with collaboration, some upper-bound length speculation can be conducted dynamically to further reduce the overall search time. Figure 6 shows such a scenario. In this figure, *GL* denotes the *globalLow*; *Opt* indicates the length of an optimal schedule; and ω_i denotes the estimated upper-bound length for the i_{th} decomposed sub-tasks. Assume that the k_{th} sub search task finds an up-to-date best schedule with length ω_k. If the partitioning approach is employed, then the ω of all the search partitions can be speculated to be $\omega - 1$. The case of upper-bound speculation method is more subtle, since all the sub-tasks search on the whole search space. When the k_{th} sub-task finds a better schedule, all the search of the i_{th} $(i > k)$ task becomes useless. This is because that the search progress of the i_{th} sub-task falls behind the k_{th} sub-task. To avoid such situation, we dynamically speculate the upper-bound length for all the sub-tasks whose index is no smaller than k. For the k_{th} task, its new upper-bound ω'_k can be speculated to be $\omega_k - 1$. If the granularity (i.e., the interval $[\omega_k, \omega_{k+1}]$) of static speculation is large, proper further speculation on the interval $(\omega_{k-1}, \omega_k - 1)$ can be performed. To make the dynamic speculation easier, for the i_{th} sub-tasks whose index is larger than k (i.e., $i > k$), its new upper-bound ω'_i can be speculated to be $Max(\omega'_k - |i-k|, globalLow)$.

4.2.3 Implementation of Cooperative Sub Search Tasks

For the method using the search space partitioning, the initial *global minimum* ω in the cooperation unit and the initial ω values of each sub search task are same. During the searching, the only difference between sub search tasks is the disjoint search space. Therefore when any sub search task finds a smaller ω than the queried value, it will try to update the global minimum ω. If the update succeeds, the task who found the schedule will become the temporary winner of competition. All the other sub-tasks will be "notified" synchronously with the new better schedule length, and the search space of them will be shrunk accordingly. If some winner achieves a schedule whose size equals to *globalLow*, all the search process will be terminated and the winner will report the global optimal schedule which was found in its local area. Otherwise, the overall search process will be terminated until all the sub search tasks finish their search.

For the upper-bound speculation method, initially the global minimum ω in the cooperation unit is the same as the upper-bound estimation of the whole search space, which equals to the length of a feasible schedule. However, unlike the partitioning method, the initial ω value of each sub search task is different. Therefore, in the upper-bound speculation method the procedure *update* cannot be invoked in the same way as in the search space partitioning method. In the *update* procedure, the winner is the search task that finds the shortest schedule. However, for the sub search tasks in speculation method, initially ω only has a speculative value (i.e., there may not be a schedule of length ω). Therefore the ω value cannot be directly used to update the winner information. Only when the sub search task achieves a better schedule whose length is smaller than or equal to its initial ω, the sub search task is eligible to claim the winner of the search. To impose this constraint to the decomposed sub search tasks, a Boolean variable *change* is introduced to indicate the status of the sub search tasks.

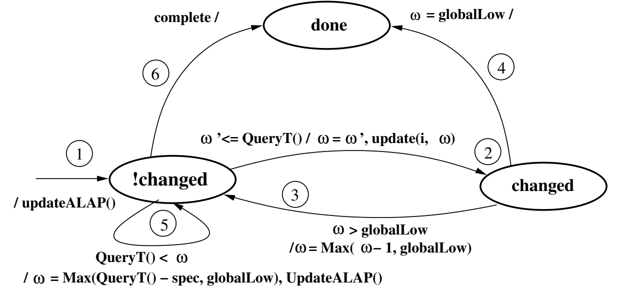

Figure 7: Extended state machine for the i_{th} search task

Figure 7 models the behavior of a sub search task. In this figure, ω indicates the upper-bound length found by the search task so far, and ω' indicates the length of a better schedule than the global minimum ω. There are three states indicating the possible status of the sub search task. The !*changed* state is an initial state, which indicates that no new better schedule has been found since the beginning of the search or since the last update of the *change* variable. The *changed* state denotes that the search task finds a schedule that is better than the query result. The state *done* represents the termination of the search task. There are 6 transitions in this state machine.

- Transition 1 initializes the search by shrinking the search space based on the initial ω.
- Transition 2 asserts that the search task finds a schedule which is better than the queried one.
- Transition 3 updates the global minimum ω and set the ω of current sub-task to be $\omega - 1$, which is a kind of dynamic speculation.
- Transition 4 terminates the search because an optimal schedule has been found.
- Transition 5 periodically queries the global minimum ω, and shrinks the search space accordingly. For dynamic speculation. *spec* equals to 1 when adopting partitioning method. Otherwise, when adopting static speculation method, *spec* equals to $|i - QueryW()| + 1$.
- Transition 6 asserts that all possible schedules have been explored.

By using this state machine to model the cooperation behavior of a sub search task, we can quickly achieve a small global minimum ω during the parallel search, which in turn prunes the search space effectively. It is important to note the state machine in Figure 7 can be applied on both the search space partitioning method and upper-bound speculation method for the cooperation among sub search tasks. Theorem 4.1 proves the correctness of the proposed cooperation framework.

THEOREM 4.1. *In RCS, by using the minimum* ω *synchronization and the sub-tasks modeled in Figure 7, one of the optimal solutions can be found finally.*

PROOF. Let $le(S_{optimal})$ be the length of the optimal schedules. If $le(S_{optimal})$ equals to *globalLow*, according to the definition of *globalLow*, the search will hit it and abort the whole search safely.

If $le(S_{optimal})! = globalLow$, at any time there should be a minimum range $[U_{i-1}, U_i]$ such that $U_{i-1} \leq le(S_{optimal}) \leq U_i$, where U_{i-1} and U_i are the upper-bound lengths estimated statically or dynamically by some sub-tasks. In this case, all the sub-tasks whose ω (upper-bound length) is smaller than U_{i-1} will always stay in state !*changed* until the search completes, since no better schedule can be found to trigger the transition 2. Only the sub-task whose ω is larger than or equal to $le(S_{optimal})$ has the chance to update the global minimum ω on transition 2. Since the state *changed* indicates that the sub-task find a global minimum upper-bound length ω so far, its following job on the unexplored search space is to find the one whose upper-bound is smaller than ω. Its new upper-bound estimation $\omega - 1$ can sufficiently guarantee that, if $\omega - 1 \geq le(S_{optimal})$, the optimal schedule will not be missed. Therefore, no matter which decomposition approaches are used, the cooperation framework can guarantee that one of the optimal solutions can be found finally. □

Algorithm 3: Algorithm for A Parallel Sub Search Task

```
Input: i) An HLS DFG D with resource constraints;
       ii) Operation set OP = {op_1, ..., op_N} in dispatching order;
       iii) Search space SP with specific [ASAP,ALAP] values;
       iv) S_bsf, which is a feasible schedule and its length is ω for D;
       v) S, which stores the current incomplete schedule;
       vi) The ID of sub search task; and vii) Parallel strategy T;
Output: An optimal schedule and its length for D
SubSearch(D, OP, SP, N, i, S, S_bsf, ω, ID, T) begin
  1. UpdateALAP();
  if i <= N then
    for step = ASAP_SP(op_i) to ALAP_SP(op_i) do
      if Precedence(op_i) ∧ ResAvaible(step, type(op_i)) then
        /* state !changed */
        if QueryT() < ω then
          if T == partitioning then
            2. ω = Max(QueryT() - 1, globalLow);
          else
            3. ω = Max(QueryT() - 1 - |ID -
               QueryW()|, globalLow);
          end
          4. UpdateALAP();
        end
        5. lower = le(LBound(S));
        6. ω' = upper = le(UBound(S));
        if ω' <= QueryT() then
          7. ω = upper;
          8. S_bsf = UBound(S);
          9. update(ID, ω);
          /* state changed */
          if ω == globalLow then
            /* state done */
            10. Terminate(S_bsf, ω);
          end
          11. ω = Max(ω - 1, globalLow);
        end
        if lower < Max(ω, globalLow + 1) then
          12. S(op_i) = step;
          13. ResOccupy(step,type(op_i),delay(op_i));
          14. SubSearch(D, OP, SP, N, i + 1, S, S_bsf, ω, ID, T);
          15. ResRestore(step,type(op_i),delay(op_i));
        end
      end
    end
  end
  Return(S_bsf, ω).
end
```

Based on the state machine presented in Figure 7, we developed the Algorithm 3. Assume that the ID of the sub search task is i, and tasks are running in parallel on different cores. In this algorithm, step 1 compacts the initial search space based on the initial ω of the specified sub search task. Steps 2-4 are involved in the !*changed* state. When the queried result is smaller than the current ω, if the partitioning method is applied, step 2 dynamically speculates the ω with the value $Max(QueryT() - 1,\ globalLow)$. If the dynamic speculation is applied, step 3 speculates the ω with the value $Max(QueryT() - 1 - |i - QeuryW()|,\ globalLow)$. Step 4 compacts the search space based on the reduced upper-bound length. Steps 5 and 6 estimate the lower-bound length and upper-bound length of the optimal schedule based on the dispatched operations. If the estimated upper-bound length is better than the queried result, the search task goes into the *changed* state. Steps 7 and 8 replace the up-to-date best schedule with the newly found one. Steps 9 tries to update the global minimum ω. In the *changed* state, if ω equals to *globalLow*, step 10 will terminate the overall search. If ω is larger than *globalLow*, step 11 will speculate the length of S_{bsf} to be $Max(\omega - 1, globalLow)$. The steps 12-15 try to recursively handle the unscheduled operations.

4.2.4 Our Parallel B&B Method

To parallelize the RCS search, we divide the overall search into k sub-tasks. Assuming that the parallel search adopts the decomposition strategy T. Based on T, the search space is divided into the search space set $SP = \{sp_1, sp_2, \ldots, sp_k\}$, and the corresponding initial upper-bound lengths are based on the upper-bound length set $\Omega = \{\omega_1, \omega_2, \ldots, \omega_k\}$.

Algorithm 4: Our Parallel B&B RCS Algorithm

```
Input: i) An HLS DFG D with resource constraints;
       ii) Operation set OP = {op_1, ..., op_N} in dispatching order;
       iii) S_bsf, which is a feasible schedule and its length is ω for D;
       iv) S, which stores the current incomplete schedule;
       v) Search space set SP = {sp_1, sp_2, ..., sp_k};
       vi) Upper-bound length set Ω = {ω_1, ω_2, ..., ω_k};
       vii) Parallel strategy T;
Output: An optimal schedule and its length for D
ParaBULB(D, N, S, S_bsf, ω, SP, Ω, T) begin
  1. globalLow = le(LBound(S));
  2. update(1, ω);
  #pragma omp parallel for schedule(dynamic)
  for i=1 to k do
    3. Task(i) = SubSearch(D, OP, SP_i, N, 1, S, S_bsf, ω_i, i, T);
    if Task(winner).global_min == globalLow then
      4.Return (Task(winner).S_bsf, global_min)
    end
    if T adopts speculation then
      if winner = i then
        5. Return (Task(winner).S_bsf, global_min);
      end
      if Task(i).ω > globalLow then
        6. globalLow = Task(i).ω + 1;
      end
    end
  end
  if T adopts partitioning then
    7. Return (Task(winner).S_bsf, global_min);
  end
end
```

Algorithm 4 describes our parallel B&B RCS approach. In this algorithm, step 1 calculates the estimated *globalLow*. Step 2 initializes the global minimum ω. Then a parallel for loop in OpenMP [2] format is started to handle the sub search invoked by step 3 in parallel. If in the half way a schedule whose size equals to *globalLow* is found, then step 4 will terminate the search and report the optimal solution. Otherwise, if the task decomposition adopts the upper-bound speculation strategy and the winner is the sub-task that finished the searching, then the whole search can be terminated in step 5. If the task is not the winner and its best solution is larger than *globalLow*, then step 6 will update the value of *globalLow* to be $Task(i).\omega + 1$. If the parallel search adopts the search space partitioning method, the search needs to wait for the completion of all the sub-searches, and then step 7 reports an optimal solution.

5. CASE STUDY

To evaluate the effectiveness of our proposed parallel pruning approaches, we conducted the experiments with different kinds of resource constraints. We collected the following benchmarks from the *MediaBench* benchmark [16] which is a standard DSP benchmark suite: i) *ARFilter* with 28 nodes and 30 edges, ii) *Cosine 1* with 66 nodes and 76 edges, iii) *Collapse* with 56 nodes and 73 edges, and iv) *Feedback* with 53 nodes and 50 edges. We also used the benchmark *FDCT* with 42 nodes and 52 edges from [11]. By using the C programming language and OpenMP APIs, we implemented a parallel B&B RCS prototype tool which incorporates our decomposition approaches and cooperation framework for parallel pruning. For comparison, we also derived constraint programming models for the above 5 benchmarks using *IBM ILOG CPLEX CP Optimizer* [13], which adopts the parallel branch-and-cut heuristic method [22] for efficient searching. All the experimental results were obtained on a Linux sever with 96 Intel Xeon 2.4GHz cores and 1 TB RAM.

5.1 Results for Functional Constraints

Table 1 presents the experimental results carried out with different functional unit constraints on the 5 benchmarks. In this experiment, we only investigated the parallelism with 8 cores. That means, when using our B&B approach, all parallel *for* loops were running on 8 parallel OpenMP threads. We only took 32 partitions for the partition-based decomposition method, and all the 32 subtasks were scheduled on the 8 threads dynamically. For the hybrid approach, there were 128 sub-tasks (32 partitions $\times$ 4 speculation tasks/partition) generated for the scheduling. To avoid the unexpected long searching time, we use the notation *NA* to indicate the RCS abortion due to the time limit (i.e., 10000 seconds)

In Table 1, column 1 presents the names of the scheduled benchmarks. Column 2 indicates the resource constraints of adders and multipliers which are denoted by a and m respectively. Due to the space limit, we do not show the constraints for other resource units. Columns 3 and 4 present the initial lower-bound and upper-bound length estimations of an optimal schedule. Column 5 shows the length of the optimal schedule achieved finally. It is important to note that the upper-bound lengths are achieved using the list scheduling approach, which is widely used to achieve a near-optimal result in many commercial tools due to the time efficiency. Although list scheduling needs less exploration time, the performance of the explored schedule cannot be guaranteed (e.g., for the FDCT design with 2 adders and 2 multipliers, the c-step achieved using list scheduling is 22, but the optimal c-step is 18). Column 6 presents the parallel solving time using the CPLEX CP Optimizer on 8 cores. It can be found that by using the commercial constraint solving engine, only one benchmark item can be solved within time limit. Column 7 presents the scheduling time using the non-parallel BULB approach [17]. By combining both the static and dynamic upper-bound speculation methods, column 8 gives the scheduling time using our speculation approach. Columns 9 and 10 compare the scheduling time of partitioning method using different dynamic upper-bound speculation strategies. The column 9 only does pure partitioning at the beginning of the search without incorporating the dynamic speculation of the bounds, while column 10 takes the dynamic upper-bound speculation into account. Both of them do not consider the static speculation. From this table we can find that for the simple cases (e.g., the benchmark items whose BULB searching time is less than 1 second), there is no obvious difference between these two approaches. However, when handling complex cases, the one with the dynamic speculation will be a better choice. For example, in the case of *Cosine* 1 with $2a, 2m$ resources, the partitioning method without dynamic speculation needs 781.34 seconds. But when using the one with dynamic speculation, it just requires 66.74 seconds. Column 11 gives the results using hybrid method (with both static and dynamic upper-bound speculations). To highlight the best scheduling results, in columns 7-11, they are marked with gray color. The last column shows the improvement of the marked results over the scheduling time using the BULB method. From Table 1, it can be found that our approaches can not only surpass the performance of the state-of-the-art parallelized constraint programming method, but also can improve the non-parallelized BULB heuristics by several orders of magnitude.

Due to the distinct decomposition strategies, the performance of the speculation method and the partitioning method is different, though both of them outperform the BULB approach. As an example in *ARFilter*, the speculation method outperforms the partitioning method and hybrid method, since smaller initial upper-bound estimation can efficiently compact the search space. But in the *Collapse* case, the speculation does not work, since the compacted search space is still large. The partitioning method can quickly achieve the optimal result in this case, because some optimal results exist at the boundary of some partition. Based on the synergy of the upper-bound speculation (static and dynamic) and search space partitioning heuristics, the hybrid approach can achieve the best overall performance. From Table 1, we can find that the hybrid approach outperforms other methods in 11 out of 20 benchmark items. Especially for the hard benchmark items which need more than 10 seconds using BULB, the hybrid approach achieves best results in 8 out of 11 benchmark items. And for the remaining 3 out of 11 hard benchmark items, the performance difference between the hybrid approach and the optimal approach is within an acceptable range (i.e., less than 2 times).

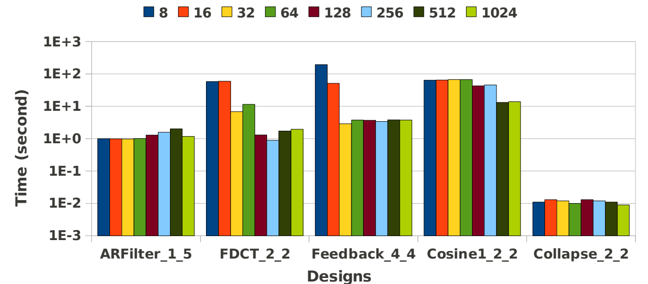

Figure 8: Partitioning method with different number of partitions

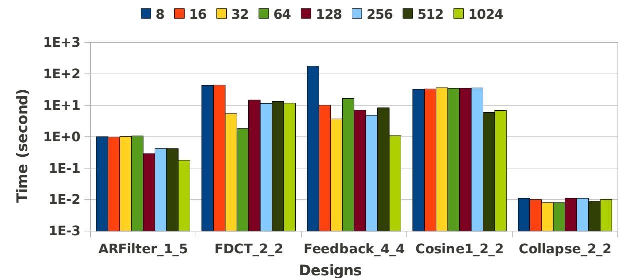

Figure 9: Hybrid method with different number of partitions

For the speculation method, there are at most $upper - lower + 1$ bound estimations. No further decomposition can be done. Hence, more cores may not bring more gains. However, for the partitioning method, if there are n nodes, there can be at most 2^n sub-tasks, which is far more than the speculation method. In this case, all the available cores can be fully utilized during the search. There-

Table 1: The RCS Results for the Collected Benchmarks with Limited Functional Units

Benchmark					CP [13]	BULB [17]	Spec. (sec.)	Partitioning (sec.)		Hybrid	Max
name	# of a, m	lower	upper	le($S_{opt.}$)	(sec.)	(sec.)	sspec+dspec	w/o dspec.	w/ dspec.	part.+spec.	speedup
ARFilter	1, 3	14	16	16	NA	0.31	0.02	0.40	0.39	0.39	15.50
	1, 4	14	16	16	NA	0.78	0.06	1.00	0.97	1.01	13.00
	1, 5	14	16	16	NA	0.77	0.07	0.98	0.98	1.03	11.00
	2, 3	14	15	15	1.93	0.01	0.01	0.04	0.03	<0.01	>1.00
FDCT	1, 2	26	27	26	NA	36.91	43.89	<0.01	<0.01	<0.01	> 3691.00
	2, 2	18	22	18	NA	201.59	58.31	9.90	6.95	13.99	29.00
	2, 3	14	17	14	NA	19.80	6.51	7.24	3.19	2.85	6.95
	2, 4	13	15	13	NA	4.07	5.06	2.69	2.22	0.87	4.68
	2, 5	13	14	13	NA	0.92	0.99	1.10	1.07	0.04	23.00
	3, 4	11	13	11	NA	0.55	0.50	0.78	0.77	0.67	1.10
	4, 4	11	12	11	NA	0.12	0.12	0.19	0.18	0.23	1.00
Feedback	4, 4	13	14	13	NA	154.18	176.43	2.92	2.88	3.82	53.53
	4, 5	13	15	13	NA	NA	NA	3.14	3.08	4.50	3246.75
	5, 5	13	14	13	NA	4.87	5.50	0.35	0.35	1.51	13.92
Cosine 1	1, 2	28	29	28	NA	107.43	137.36	<0.01	< 0.01	< 0.01	>1.00e4
	2, 2	20	23	20	NA	622.83	41.02	781.34	66.74	34.54	18.03
	3, 3	16	17	16	NA	0.01	< 0.01	0.05	0.04	0.04	> 1.00
Collapse	2, 1	22	23	22	NA	NA	NA	0.04	0.03	0.02	> 5.00e5
	2, 2	21	23	21	NA	NA	NA	< 0.01	0.02	< 0.01	> 1.00e6
	2, 3	21	23	21	NA	NA	NA	<0.01	<0.01	< 0.01	> 1.00e6
	2, 4	21	23	21	NA	NA	NA	<0.01	< 0.01	< 0.01	> 1.00e6

* NA means that the scheduling time is larger than 10000 seconds.

fore, in our partitioning method, the number of partitions strongly affects the scheduling performance. Figure 8 and Figure 9 analyze the effect of the number of partitions using both the search space partitioning method (with dynamic upper-bound speculation) and the hybrid approach. In these figures, we only choose one typical case from each design in the Table 1. We use the notation *design_m_n* to denote the design with the resource of *m* adders and *n* multipliers. Since we use 8 cores for parallelism in the comparison, the number of partitions starts from 8. It is important to note that the results with 8 partitions may have already been improved significantly compared to the BULB approach. From these figures, we can find that in most cases, the more partitions involved during the searching, the better RCS performance we can achieve. However, in practical implementation of our approaches, more partitions indicate more sub-tasks. The memory allocation and the preprocessing for a large set of sub-tasks may cause some overhead. Interestingly, from Figure 8 and Figure 9, we can find that there exists a *performance jump point* with respect to the number of partitions. At this point, even if more partitions are generated, the overall scheduling performance cannot be improved drastically further. Based on the results shown in Figure 8 and Figure 9, we can find that *the number of cores* $\times$ *8* (i.e., 64) could be used as such a performance jump point for all these five designs.

The core number (i.e., thread number in OpenMP) also plays an important role during the search using the partitioning and the hybrid approaches. Generally, the more cores are used for the parallel computation, the more speedup can be achieved. However, as mentioned in Section 1, this is not true in B&B style parallel RCS. Figure 10 and Figure 11 show the results with different number of cores, where the search space of each design is divided into 128 partitions. From these two figures, we can find that in most cases, when the number of cores is larger than 4, increasing the core number will be a waste of computing resources, since it cannot drastically reduce the searching time. To further reduce the scheduling time, when more cores are available, a wise strategy is to run distinct parallel pruning approaches individually on different sets of cores. For example, if there are 12 cores, we can equally divide them into 3 sets and run each of our approaches on one core set. If any one of the parallel searches finishes first, all the other two searches can be safely terminated. By using this strategy, we can always achieve the largest speedup over BULB as shown in Table 1.

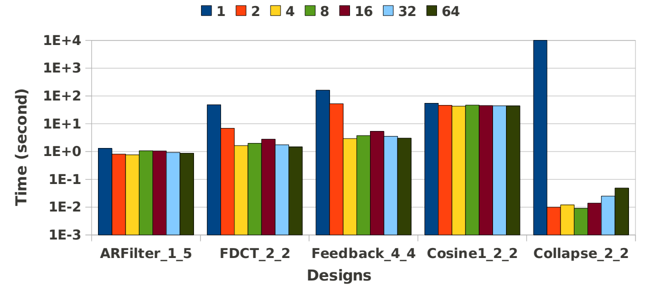

Figure 10: Partitioning method with different number of cores

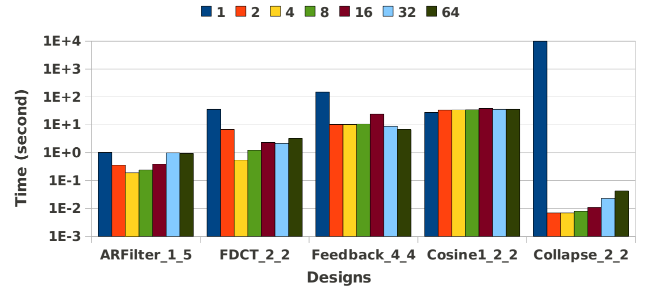

Figure 11: Hybrid method with different number of cores

5.2 Results for Area and Power Constraints

The power and area are two key factors during the hardware design. The scheduling under these two constraints can be considered as a variant of the time-minimum resource scheduling [10]. Since both area and power can be treated as special kinds of resources, our proposed approach can also be used to promote the scheduling performance under such non-functional constraints. We did the

experiment with the designs given in Table 1 using these two constraints. Due to the space limit, we only present the results for the FDCT design. All the other designs have similar results.

We evaluate our approach on FDCT design using both power and area constraints. FDCT only contains four types of functional operations (i.e., +, -, *, /). Table 2 lists the RCS settings for the functional units of these four operations.

Table 2: The Settings of the Functional Units in FDCT

Functional Unit	Operation Class	Delay (unit)	Power (unit)	Energy (unit)	Area (unit)
Adder	+	1	10	10	10
Subtracter	-	1	10	10	10
Multiplier	*	2	20	40	40
Divider	/	2	20	40	40

Figure 12 and Figure 13 show the scheduling time with the area constraints 140 units and 100 units, respectively. For the parallel scheduling, we use 8 cores and 32 partitions. In each figure, we use the notation *PX* to denote that the power constraint is *X*. From these two figures, we can find that our proposed approach (especially the hybrid approach) can achieve several orders of magnitude improvement over the BULB approach, which is consistent with the results under the functional unit constraints in Section 5.1.

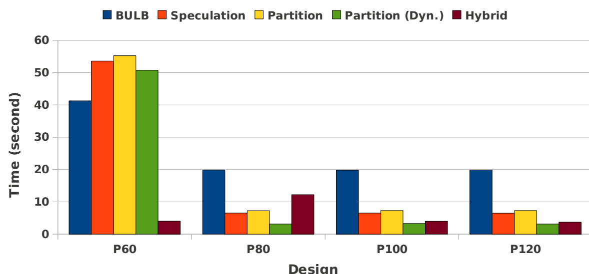

Figure 12: Scheduling results with an area of 140 units

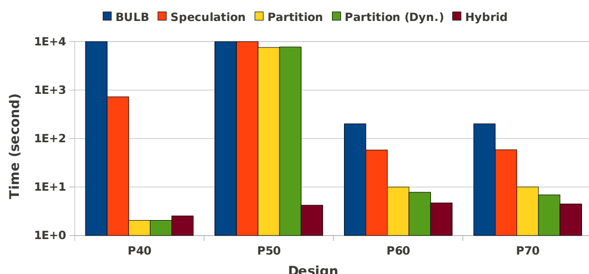

Figure 13: Scheduling results with an area of 100 units

6. CONCLUSIONS

This paper presented various promising bound-oriented parallel pruning approaches for branch-and-bound resource constrained scheduling during high-level synthesis. It proposed two decomposition approaches: i) the search space partitioning method which can avoid the *stuck-at-local-search* problem; and ii) the static and dynamic upper-bound speculation methods which can restrict the search space within a small bound. To further reduce the search complexity, this paper presented a cooperation framework which can share the search progress information among search tasks. Such cooperation can enable both space shrinking and dynamic speculation. Therefore, the overall search time under various constraints can be drastically reduced. Experimental results demonstrated that our method can outperform existing approaches by several orders of magnitude.

7. ACKNOWLEDGEMENTS

This work was partially supported by the grants from Natural Science Foundation of China (No. 61202103, 91118008, 91118007 and 61021004), State High-Tech Development Plan of China (No. 2011AA010101), Open Project of Software/Hardware Co-design Engineering Research Center of MoE (No. 2013001), and Shanghai Knowledge Service Platform Project (No. ZF1213).

8. REFERENCES

[1] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Wglarz. *Scheduling Computer and Manufacturing Processes*. Springer-Verlag, 1996.

[2] B. Chapman, G. Jost, and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[3] M. Chen, S. Huang, G. Pu and P. Mishra. Branch-and-bound style resource constrained scheduling using efficient structure-aware pruning. In *Proceedings of ISVLSI*, 2013. To appear.

[4] H. Cherroun and P. Feautrier. An exact resource constrained-scheduler using graph coloring technique. In *Proceedings of AICCSA*, pages 554–561, 2007.

[5] 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.

[6] R. Cordone, F. Ferrandi, M. D. Santambrogio, G. Palermo, and D. Sciuto. Using speculative computation and parallelizing techniques to improve scheduling of control based designs. In *Proceedings of ASP-DAC*, pages 898–904, 2006.

[7] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *Design & Test of Computers*, 26(4):8–17, 2009.

[8] T. G. Crainic, B. L. Cun, and C. Roucairol. Chap1: Parallel branch-and-bound algorithms. In *Parallel Combinatorial Optimization*, pages 1–28. John Wiley & Sons Inc., 2006.

[9] C. H. Gebotys and M. I. Elmasry. Global optimization approach for architectural synthesis. *IEEE TCAD*, 12(9):1266–1278, 1993.

[10] J. Hansen and M. Singh. A fast branch-and-bound approach to high-level synthesis of asynchronous systems. In *Proceedings of ASYNC*, pages 107–116, 2010.

[11] S. Haynal and F. Brewer. Automata-based symbolic scheduling for looping DFGs. *IEEE Transactions on Computers*, 50(3):250–267, 2001.

[12] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE TCAD*, 10(4):464–475, 1991.

[13] IBM ILOG CPLEX CP Optimizer V12.3. http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/index.html.

[14] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. In *Proceedings of ICCD*, pages 16–20, 1993.

[15] G. Martin and G. Smith. High-level synthesis: past, present, and future. *Design & Test of Computers*, 26(4):18–25, 2009.

[16] Media Benchmarks. http://express.ece.ucsb.edu/benchmark/.

[17] 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.

[18] S. Ohm, F. Kurdahi, and N. Dutt. Comprehensive lower bound estimation from behavioral descriptions. In *Proceedings of ICCAD*, pages 182–186, 1994.

[19] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE TCAD*, 8(6):661–679, 1989.

[20] N. Pothineni, P. Brisk, P. Ienne, A. Kumar, and K. Paul. A high-level synthesis flow for custom instruction set extensions for application-specific processors. In *Proceedings of ASP-DAC*, pages 707–712, 2010.

[21] G. Pu, J. He, and Z. Qiu. An optimal lower-bound algorithm for the high-level synthesis scheduling problem. In *Proceedings of DDECS*, pages 151–152, 2006.

[22] T. K. Ralphs. Chap3: Parallel branch and cut. In *Parallel Combinatorial Optimization*, pages 53–101. John Wiley & Sons Inc., 2006.

[23] M. Rim and R. Jain. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE TCAD*, 13(4):451–458, 1994.

[24] A. Sharma and R. Jain. Estimating architectural resources and performance for high-level synthesis applications. *IEEE TVLSI*, 1(2):175–190, 1993.

[25] J. A. Stankovic, M. Sprui, M. D. Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):15–25, 1995.

[26] A. H. Timmer and J. A. G. Jess. Execution interval analysis under resource constraints. In *Proceedings of ICCAD*, pages 454–459, 1993.

[27] G. Tiruvuri and M. Chung. Estimation of lower bounds in scheduling algorithms for high-level synthesis. *ACM TODAES*, 3(2):162–180, 1998.

[28] 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.