A Fast and Effective Lookahead and Fractional Search Based Scheduling Algorithm for High-Level Synthesis

Shantanu Dutt and Ouwen Shi
University of Illinois at Chicago
Email: {dutt, oshi2}@uic.edu

Abstract – We present a latency-constrained iterative list scheduling type algorithm, FALLS, to minimize the total number of functional units (FUs) allocated, and thus the total area, in high-level synthesis designs. The algorithm incorporates a novel lookahead technique to selectively schedule available operations by allocating the needed FUs earlier or reserving available FUs for scheduling more timing-urgent operations later, such that no additional FU is needed and higher FU utilization is obtained. Further, a fractional search framework is developed to iteratively estimate the number of FUs of each function type required in the final design based on the current scheduling solution and FU utilization, and reiterate the lookahead-based list scheduling with the new FU allocation estimate to further increase FU utilization. Extensive experiments conducted over several DFGs and a wide range of latency constraints demonstrate that FALLS is much more effective than other approximate state-of-the-art algorithms in both number of FUs and total FU area, and has a much smaller runtime. Results also show that FALLS has only an average 5.5% optimality gap compared to an optimal integer linear programming (ILP) formulation, but is 278k times faster. FALLS also performs much better in architectural (FU + mux/demux + register) area, interconnect congestion and number of interconnects than approximate algorithms, and is at most 4.0% worse in them than the ILP method.

I. Introduction

High-level synthesis (HLS) tools schedule operations in the design specification to clock cycles (cc's), bind and allocate the operations to functional units (FUs) to optimize an objective function subject to various design constraints. Latency-constrained scheduling to minimize the total number of FUs allocated in a synthesized design has been an objective of interest for decades, as it has a strong correlation to area and leakage power optimization.

There are many operation scheduling works focusing on FU minimization [1-12]. The integer linear programing (ILP) formulation proposed in [1] [2] provides optimal scheduling solutions for FU minimization, but it is impractical for large designs due to the exponential runtime complexity. Force-directed scheduling (FDS) presented in [3] [4] schedules operations iteratively by choosing the scheduling option that best balances the operation execution distribution across all cc's using the concept of minimum "force", and thereby minimizes the number of FUs required. The sub-optimality of FDS stems from its greedy and sequential scheduling option selection and lack of lookahead. In addition, the high runtime complexity of $O(n^3)$, where n is the number of operations, motivates several refinements [5] [6] that reduce the complexity to $O(n^2)$. A versatile technique called SDC proposed in [7] models the scheduling problem as a linear programming formulation. Though it can solve different types of HLS problems, only timing problems like latency minimization can be solved exactly, and only these results are presented. Stochastic methods for FU minimization have also been widely studied. A simulated annealing (SA) approach is proposed in [8] and its move set guarantees that the complete solution space can be explored. In [9], an ant-colony based algorithm ACO is developed to gradually approach a good solution by iteratively and probabilistically generating scheduling solutions based on which the scheduling probabilities are updated. In [10], the scheduling order of operations is determined by a genetic algorithm. The above stochastic methods [8] [9] [10] require high runtime for achieving good quality solution, which prevents them from being effective for large problem sizes.

List scheduling (LS) is a classical scheduling algorithm for latency-constrained FU minimization. It schedules operations in as early cc's as possible if FUs are available, while greedily avoiding allocating new FUs unless it is mandatory for satisfying the latency constraint. Though the solution of LS is far from optimal as discussed later, its complexity of $O(n \log n)$ is very scalable. Therefore, several works like [10] [11] [12] that use LS or LS-type algorithms as an internal sub-routine to achieve good optimization quality. Lookahead in LS has been studied in scheduling problems in non-HLS fields [13] [14]. Early lookahead in instruction scheduling like [13] helps LS with a bad "precedence function" (different from the one we will discuss) to avoid failed scheduling by tentatively scheduling some or all unscheduled operations. Recent works like [14] in heterogeneous computing use a lookahead approach to exhaustively evaluate all candidate resources that can execute the scheduled operation, and chooses the resource that is most likely to lead to the smallest estimated latency. These lookahead techniques are not applicable to FU or area minimization in HLS.

In this paper, we propose a novel FrActional search and Lookahead based List Scheduling (FALLS) algorithm. We construct a fractional search framework, an estimate based extension of binary search, to gradually approach a good scheduling solution by pre-allocating an appropriate number of FUs at the beginning of a scheduling iteration. Internally, a scheduler with a novel and more streamlined lookahead technique than in past work is used to schedule non-0-slack operations for higher utilization, and thus minimized allocation, of FUs. Moreover, our algorithm maintains a complexity comparable to that of LS, and thus scales well for large HLS designs, which is also shown in our experiments.

The rest of the paper is organized as follows. We formulate the FU minimization scheduling problem and review the classical list scheduling in Section II. Our FALLS algorithm is discussed in greater detail in Section III. Experimental results comparing FALLS to state-of-the-art algorithms are presented

978-3-9819263-0-9/DATE18/©2018 EDAA

in Section IV, and we conclude in Section V.

II. Background

A. Problem Formulation

We consider the following well-known Minimum-Resources Latency-Constrained Scheduling (MR-LCS) problem. Given:

1. An unscheduled data-flow graph (DFG) $G(V, E)$, where V is the set of operations and E is the set of arcs representing data dependencies between the operations.
2. A legal upper-bound latency constraint L_c in number of cc's that is no smaller than the critical path delay.
3. An FU library R that includes one FU design for each function type (FT) that appears in the operations in V.

For any two operations $u, v \in V$ and a data dependency arc $(u, v) \in E$, if u and v are scheduled in cc's t_u and t_v, respectively, the *dependency constraint* is:

$$t_u + d_u \leq t_v \tag{1}$$

where $d_u \geq 1$ is the delay of u in number of cc's.

Our objective is to schedule each operation in V to a certain cc such that the *cost*, the total number of allocated FUs, which is strongly correlated to area as shown by empirical results, is minimized such that the achieved latency $L \leq L_c$ and all dependency constraints are satisfied.

B. Review of List Scheduling

Here we briefly discuss the classical latency-constrained list scheduling (LS) algorithm for the MR-LCS problem. In each cc, LS always schedules the most timing-urgent operations to available FUs. Starting with a minimum FU allocation, a new FU is only allocated when there is an available operation that needs to be scheduled immediately to satisfy the latency constraint, but there is no allocated FU of that FT currently available due to being busy executing other operations. By only allocating new FUs when it is mandatory, LS was expected to come close to minimizing the number of allocated FUs in the final scheduling solution. However, LS fails due to the low FU utilization mentioned later.

The pseudo code of LS is presented in Fig. 1. Initially, only one FU per FT is allocated. The *as late as possible* (ALAP) time t^L, the latest cc where an operation can be scheduled to satisfy the L_c, is computed for each operation. Let $pred(u)$ and $succ(u)$ denotes the set of predecessors and successors of operation u, respectively. The symmetric *as soon as possible* time t^S and the ALAP time t^L are recursively defined as:

$$t_u^S = \max_{v \in pred(u)} (t_v^S + d_v) \tag{2}$$

$$t_u^L = \min_{w \in succ(u)} (t_w^L - d_u) \tag{3}$$

where $t_u^S = 1$ if $pred(u) = \emptyset$, and $t_u^L = L_c - d_u + 1$ if $succ(u) = \emptyset$. Then in each cc t in chronological order, for each FT k, an available unscheduled operation set $U_{t,k} \in V$, which includes all unscheduled operations of FT k whose predecessors have all finished execution, is determined. The *slack* s_u of each operation u in $U_{t,k}$ is then computed as:

$$s_u = t_u^L - t, \qquad u \in U_{t,k} \tag{4}$$

If $s_u = 0$, u is *0-slack* and must be scheduled in cc t, i.e., one additional FU needs to be allocated if all FUs of FT k are busy executing other operations. The other operations in $U_{t,k}$ are *non-0-slack*. If there are still available FUs after all 0-slack operations are scheduled, the non-0-slack operations are scheduled in cc t in slack-increasing order and bound to the available FUs. This slack-based scheduling process iterates for each cc t until all operations are scheduled. The time complexity of LS is $\Theta(n \log n)$, since each sorting or searching operations in a balanced binary search tree based on ALAP times (equivalent to slack) takes $\Theta(\log n)$ time, and the total number of searches is equal to the total number of available operations across all cc's, which is $\Theta(n)$.

Algorithm LS (DFG $G(V, E)$, latency constraint L_c, FU library R)
1. $\boldsymbol{r} = (1,1, ..., 1)$, $t = 1$ *//pre-allocate one FU per FT*
2. Compute the ALAP times t^L for L_c
3. **While** there are unscheduled operations **Do**
4. **For** each FT k **Do**
5. Determine the available unscheduled operation set $U_{t,k}$
6. Compute slack s_u for all $u \in U_{t,k}$ by (4)
7. Schedule 0-slack operations in $U_{t,k}$ to t, allocate new FUs if needed, update r_k if new FUs are allocated
8. Schedule non-0-slack operations in $U_{t,k}$ to t in slack-increasing order and bind them to remaining available FUs
9. **End For**
10. $t = t + 1$
11. **End while**
12. **Return** the scheduling solution

Fig. 1. The pseudo code of the classical list scheduling algorithm.

We term the FU allocation vector $\boldsymbol{r}$ (one element per FT) before any operation is scheduled as *pre-allocation*; it is only one FU per FT in LS; Similarly, the FU allocation vector $\boldsymbol{r}$ after all operations are scheduled is termed as *post-allocation*. In practice, the number of FUs in post-allocation is significantly more than that in pre-allocation in the solutions of LS, indicating many FUs are allocated in intermediate cc's. This results in the FUs allocated in later cc's being sparsely utilized. Due to the insufficient FU utilization, excessive FUs are likely to be allocated in the solutions of LS.

III. Our FALLS Algorithm

We present our FALLS algorithm in this section. FALLS schedules in chronological order of cc's and utilizes slack to determine the timing-urgency of available unscheduled operations, which are the beneficial aspects of the classical LS algorithm. However, to rectify the drawback of LS, we have made significant extensions as follows:

- We schedule non-0-slack operations following a novel lookahead technique that allocates new FUs earlier than they would be in LS or reserves available FUs in the current cc for scheduling future 0-slack operations, such that the average FU utilization is increased.
- An estimate based extension of binary search, which we call fractional search, is proposed to incrementally estimate the number of FUs required for the design and finally accurately pre-allocate FUs at the last scheduling iteration to further increase FU utilization.
- We use FU utilization rate as a general guideline to dynamically adjust pre-allocation, pre-allocation expansion technique for conservatively pre-allocating more FUs to increase FU utilization and pre-allocation pruning technique for eliminating redundant FUs.

A general view of FALLS is given first: the pseudo code is presented in Fig. 2. The internal scheduler of FALLS, Lookahead, is based on LS but improved by our lookahead technique. Nesting the enhanced scheduler (in line 4 and 9),

fractional search iteratively determines a more accurate pre-allocation by the pre-allocation expansion (line 5) and pruning (line 6 to line 16) technique, which are based on the pre-allocation and post-allocation of the previous iteration (call to Algorithm Lookahead). The final solution is latest solution after the last iteration where there is no improvement to the current solution after FU expansion and pruning techniques.

Algorithm FALLS (DFG $G(V, E)$, L_c, FU library R)
1. $soln.\boldsymbol{r}^{pre}$ = (1,1, ..., 1) *//pre-allocate one FU per FT*
2. Compute the ALAP times t^L for L_c
3. **Repeat** *//fractional search begins*
4. $soln$ = Lookahead (t^L, $soln.\boldsymbol{r}^{pre}$)
5. For each FT k where $soln.r_k^{post} > soln.r_k^{pre}$, increase $soln.r_k^{pre}$ by (10) ($r_k^{pre/post}$ is the k'th element of vector $\boldsymbol{r}^{pre/post}$)
6. **For** each FT k where $soln.r_k^{post} \leq soln.r_k^{pre}$ **Do**
7. Get $r\_major_k^{pre}$ by major pruning (Sec. III-D) of $soln.r_k^{pre}$
8. Temporarily update $soln.\boldsymbol{r}^{pre}$ with $r\_major_k^{pre}$
9. Get a new solution = Lookahead (t^L, $soln.\boldsymbol{r}^{pre}$)
10. **If** the cost of the new solution is improved **Do**
11. Linear search the range [$r_k^{pre-min}$, $r\_major_k^{pre}$] to determine a better $r\_minor_k^{pre}$, where $r_k^{pre-min}$ is the previous largest unsuccessful $soln.r_k^{pre}$ that was tried **Else**
12. Binary search the range ($r\_major_k^{pre}$, $soln.r_k^{pre}$] to determine a better $r\_minor_k^{pre}$
13. **End If**
14. Update $soln.\boldsymbol{r}^{pre}$ with the best $r\_minor_k^{pre}$ or $r\_major_k^{pre}$
15. **End For**
16. **Until** no improvement in $soln.\boldsymbol{r}^{post}$
17. **Return** the latest scheduling solution

Algorithm Lookahead (ALAP times t^L, pre-allocation vector $\boldsymbol{r}^{pre}$)
1. $\boldsymbol{r}^{post} = \boldsymbol{r}^{pre}$, $t = 1$
2. Unschedule all operations if they are scheduled
3. **While** there are unscheduled operations **Do**
4. **For** each FT k **Do**
5. Determine the available unscheduled operation set $U_{t,k}$
6. Compute slack s_u for all $u \in U_{t,k}$ by (4)
7. Schedule 0-slack operations in $U_{t,k}$ to t, allocate new FUs if needed, update r_k^{post} if an FU is used for the first time
8. Apply the lookahead technique (Sec. III-A) to schedule non-0-slack operations in $U_{t,k}$ to t, allocate new FUs if needed and update r_k^{post} if an FU is used for the first time
9. **End For**
10. $t = t + 1$
11. **End while**
12. **Return** $\boldsymbol{r}^{pre}$, $\boldsymbol{r}^{post}$ and the scheduling solution as $soln$

Fig. 2. Pseudo code of the FALLS algorithm.

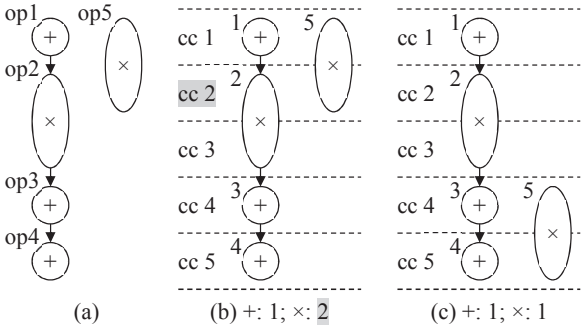

Fig. 3. Illustration of the advantage of reserving FUs for later use in the lookahead scheduling of FALLS. "opi" denotes operation i. FU allocation results are shown below solutions. (a) An unscheduled DFG; (b) The solution of LS; (c) The solution of lookahead scheduling.

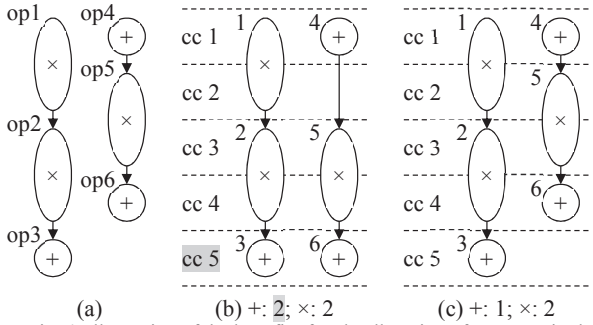

Fig. 4. Illustration of the benefit of early allocation of new FUs in the lookahead scheduling of FALLS. FU allocation results are shown below solutions. (a) An unscheduled DFG; (b) The solution of LS; (c) The solution of lookahead scheduling.

A. *Lookahead Scheduling*

The lookahead technique makes better scheduling decisions than LS for non-0-slack operations. In the current scheduling cc t, it detects operations that are currently unavailable and will become 0-slack in some near-future cc's. To allow these operations to be executed on currently available FUs when they are available and 0-slack, some available FUs are reserved for this purpose in the current cc t. Moreover, it aggressively allocates new FUs in cc t to schedule certain non-0-slack operations under the condition that if the operations are not scheduled in cc t, the same number of new FUs are still needed to be allocated for scheduling them in later cc's. By preventing allocating avoidable new FUs in later cc's and allocating new FUs earlier that are unavoidable later, the average FU utilization is increased and hence the number of FUs needed is minimized.

The advantage of reserving FUs for later use is illustrated by the example in Fig. 3. The DFG in Fig. 3(a) has only two FTs: addition with a 1-cc delay and multiplication with a 2-cc delay; L_c is 5 cc's. In Fig. 3(b), LS schedules op5 (slack = 3) in cc 1, since it is the only available multiplication operation in cc 1 and there is an available multiplier. The overlapping of execution time of op2 and op5 results in a new multiplier being allocated in cc 2. On the other hand, in cc 1, our lookahead scheduling detects that op2 will become 0-slack in cc 2 and hence reserves the multiplier for scheduling op2 in cc 2 to avoid the new multiplier being allocated, as in Fig. 3(c). As the scheduling proceeds, op5 eventually becomes 0-slack in cc 4, and the multiplier being busy in cc's 2-3 becomes available for op5. Therefore, by reserving the multiplier in cc 1 and scheduling op5 later, one multiplier is saved.

The other aspect of the lookahead scheduling, aggressive early new FU allocation, is illustrated by the example in Fig. 4 with the same set of FTs and L_c as in Fig. 3. The scheduling quality here solely depends on the allocation of adders. In Fig. 4(b), after op4 is scheduled in cc 1, LS schedules op5 in cc 3, since op5 is non-0-slack in cc 2 and there is no available multiplier then. Although a new multiplier must be allocated no matter where op5 is scheduled, LS fails to detect this situation due to the limited information provided by slack alone. This forces op6 to be scheduled in cc 5, where op3 is concurrently scheduled. This leads to a new adder to be allocated. Different from LS, our lookahead scheduling realizes that a new multiplier is unavoidable for scheduling op5, hence allocates it when op5 is first available in cc 2 and

schedules op5 there. Such scheduling decision makes no change in multiplier allocation, but reduces the number of adders by one: op6 can be scheduled one cc earlier and hence avoid being executed concurrently with op3.

Now we formulate our lookahead technique as follows. In any cc t during the scheduling process, after scheduling 0-slack operations of FT k, we explore the cc's in the cc range $R(t) = [t + 1, t + d_k - 1]$ of FT k, where d_k is the delay of FT k and $d_k > 1$. We explore this cc range because this is the maximum range for which any executing operation of FT k scheduled in a cc $\leq t$ will finish its execution in some cc in this range, and we will thus know exactly how many FT k FUs will be available in these cc's. This information, along with which operations will become 0-slack in these cc's, is needed to determine FU reservation and early new FU allocation in cc t. For cc $i \in [t, t + d_k - 1]$, we define: $a(i)$ to be the number of FUs that will be busy in $i - 1$ and become available in i; $z(i)$ to be the total number of 0-slack operations in i; $z'(i)$ to be the number of 0-slack operations in i that are available in cc $t < i$. Therefore, $z(i) - z'(i)$ is the number of 0-slack operations in i that are unavailable in t. Each of these parameters in the cc range can be determined in cc t. With these parameters, we can recursively compute $Avail(i)$, the number of available FUs in cc i after scheduling $z(i) - z'(i)$ operations in cc i, as:

$$Avail(i) = max\{0, Avail(i-1) + a(i) - [z(i) - z'(i)]\} \quad (5)$$

At the recursion boundary of cc t, $Avail(t)$ is the number of available FUs after scheduling all 0-slack operations in t. Based on $Avail(i)$, we can determine $new(i)$, the number of new FUs needed in i for scheduling the $z'(i)$ 0-slack operations of i, as:

$$new(i) = max\{0, z'(i) - Avail(i)\} \quad (6)$$

This needs to be followed by an update of $Avail(i)$ in order to compute $Avail(i + 1)$ by Eq. 5: $Avail(i) = 0$ if $new(i) > 0$, otherwise $Avail(i) = Avail(i) - z'(i)$. As is hopefully clear from the formulation, new FUs are only allocated for scheduling $z'(i)$ 0-slack operations when there are not enough available FUs after scheduling the $z(i) - z'(i)$ 0-slack operations. The updated $Avail(i)$ that accounts for scheduling $z'(i)$ operations becomes the number of available FUs in i after scheduling all its $z(i)$ 0-slack operations.

After all cc's in the cc range $R(t)$ are explored, we can determine $S(t)$, the maximum number of available non-0-slack operations to be scheduled in cc t by:

$$S(t) = max\left\{0, \min_{j \in [t, t+d_k-1]} Surplus(j)\right\} + \sum_{j=t+1}^{t+d_k-1} new(j) \quad (7)$$

where

$$Surplus(i) = Avail(t) + \sum_{j=t+1}^{t+d_k-1} \{a(j) - [z(i) - z'(i)]\} \quad (8)$$

$Surplus(i)$ is thus the number of available FUs in cc i after scheduling $z(j) - z'(j)$ 0-slack operations in each cc j in $[t + 1, i]$ without allocating any new FUs in any of these cc's; it can thus be negative. Equation (7) incorporates both aspects of the lookahead technique that are illustrated in Figs. 3 and 4. The first term with $Surplus(i)$ allows use of only $max\left\{0, \min_{j \in [t, t+d_k-1]} Surplus(j)\right\}$ of the available FUs in cc t for available non-0 slack operations in it and reserves the rest for later use in $R(t)$. The second term with $new(i)$ is for early allocation and use of the appropriate number of new FUs in cc t. The idea of $Surplus(i)$ is that if it is positive, and for the sake of argument we ignore other $Surplus(j)$ values, then we can schedule at most $min(Surplus(i), z'(i))$ available non-0-slack operations in cc t of the $z'(i)$ 0-slack operations of cc i, on the already available FUs in t (after its 0-slack operations are scheduled), without incurring any extra new FU in $R(t)$ compared to scheduling these operations in cc i. However, for this to be true for all cc's in $R(t)$, we can only schedule $\min_{j \in [t, t+d_k-1]} Surplus(j)$ (if it is positive) available non-0-slack operations in cc t (in slack increasing order—these are the operations that become 0-slack earliest among all the $z'(i)$ operations in $R(t)$) without allocating any extra new FUs in $R(t)$. If any more are scheduled in cc t, then the minimum positive $Surplus$ point r in $R(t)$ will become negative, meaning that extra new FU(s) will be needed to schedule some of the $z(r) - z'(r)$ 0-slack operations in cc r.

Thus, accounting for both the minimum $Surplus(i)$ and early allocation of new FUs in cc t, we schedule $S(t)$ available non-0 slack operations in slack increasing order in cc t.

B. *Fractional Search*

Our fractional search framework contains two sub-techniques: *pre-allocation expansion* and *pre-allocation pruning*. Both techniques rely on an indicator of FU utilization, utilization rate, to determine the number of FUs to be adjusted in the pre-allocation. The *utilization rate* (*ur*) of an FU is the fraction of cc's in which the FU is busy executing operations over the entire scheduling latency. For the p'th FU of FT k with $n_{k,p}$ operations bound to it, its utilization rate $ur_{k,p}$ is:

$$ur_{k,p} = \frac{n_{k,p} \times d_k}{L} \quad (9)$$

where d_k is the delay of FT k and L is the achieved latency of the current scheduling solution. Intuitively, FUs allocated in earlier cc's have a greater potential to have high utilization rates compared to those allocated in later cc's.

We first illustrate fractional search in Fig. 5. For any FT k of a solution, we determine the new pre-allocation by the pre-allocation and post-allocation of the previous iteration. If the former is smaller than the later, we expand the pre-allocation by adding the sum of utilization rates of new FUs (an optimistic estimate) to it. Otherwise, we attempt to prune pre-allocated FUs by a utilization rate based major pruning followed by minor binary or linear pruning steps to gradually approach the accurate pre-allocation. Unlike binary search, which iteratively eliminates half of the search space, fractional search makes the new estimate based on utilization rate to more efficiently locate the target value. Fractional search terminates when the latest round of prunings for each FT that satisfies the pruning condition, no further solution improvement can be obtained.

C. *Pre-allocation Expansion Technique*

Given a solution of an iteration, for any FT k, if its pre-allocation r_k^{pre} is smaller than its post-allocation r_k^{post}, the pre-allocation expansion technique is applied. The number of FUs to be increased in the pre-allocation r_k^{new} of FT k is:

$$r_k^{new} = \left\lceil \sum_{p \in FU\_new(k)} ur_{k,p} \right\rceil \quad (10)$$

where $FU\_new(k)$ is the set of new FUs of FT k allocated in the current scheduling iteration. The pre-allocation expansion is performed in a conservative way in which it only allocates the minimum number of FUs which can handle all operations

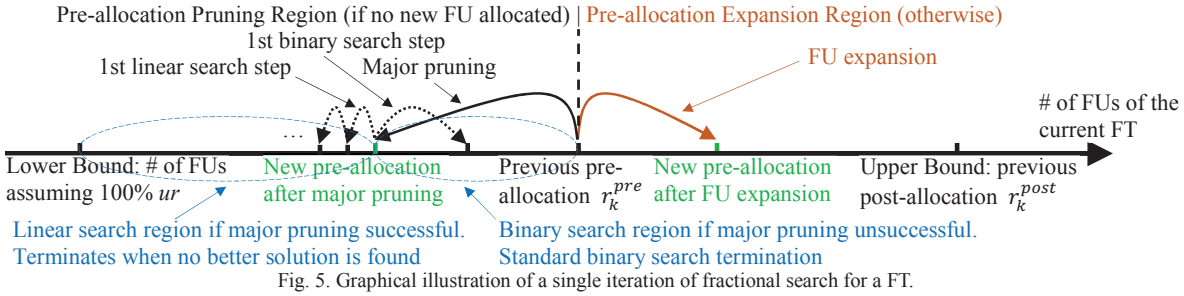

Fig. 5. Graphical illustration of a single iteration of fractional search for a FT.

bound to the new FUs with an *ideal utilization rate* of 100%. This allows fractional search to gradually approach the minimum number of required FUs.

D. *Pre-allocation Pruning Technique*

Given a solution of an iteration, for any FT k, if its pre-allocation r_k^{pre} is not smaller than its post-allocation r_k^{post}, the pre-allocation pruning technique is applied. The pruning is performed when $r_k^{pre} > r_k^{post}$, since the unutilized $r_k^{pre} - r_k^{post}$ pre-allocated FUs that have no operations bound to should obviously be pruned. Further, when $r_k^{pre} = r_k^{post}$, it is potentially beneficial to prune some of the utilized pre-allocated FUs, since these over-allocated FUs may be used to over-schedule less timing-urgent non-0-slack operations that have slacks greater or equal to the delay, making the FUs sparsely utilized in later cc's. This pruning includes a major pruning followed by minor prunings that are either based on binary or linear search.

Besides pruning the unused FUs, if any, the idea of major pruning is to adaptively increase the utilization rates of the most under-utilized FUs. Let the maximum and minimum utilization rate among all the used FUs in the current solution be ur_{max} and ur_{min}, respectively. We can evenly partition the range $[ur_{min}, ur_{max}]$ into α ($\alpha \geq 2$; $\alpha = 4$ in our experiments) partitions. The most under-utilized FUs are in the 1st partition, which is the range $urr_1 = [ur_{min}, ur_{min} + \frac{ur_{max} - ur_{min}}{\alpha})$. We attempt to increase the utilization rates of these FUs to the adjacent partition with a higher average utilization rate range uur_2 so that fewer FUs are in the pre-allocation and this is expected to translate to fewer FUs in the post-allocation by reducing over scheduling. To execute the same number of operations that were bound to the most under-utilized FUs whose utilization rates are in urr_1 with fewer but fully-utilized FUs whose utilization rates are in urr_2, the least number of FUs required m is determined as:

$$m = \frac{\sum_{p \in FU(k, urr_1)} ur_{k,p}}{ur_{avg}(k, urr_2)} \qquad (11)$$

where $FU(k, urr_1)$ is the set of FUs of FT k whose utilization rates are in uur_1 and $ur_{avg}(k, urr_2)$ is the average utilization rate of FUs whose utilization rates are in uur_2. The number of pruned FU is thus $|FU(k, urr_1)| - m$.

As illustrated in Fig. 5, after major pruning, there are a series of minor prunings using either linear or binary search. If major pruning leads to a lower-cost solution, we perform linear search to further prune the number of FUs of FT k by the smallest granularity of one and re-schedule, and iterate until no better solution is found. On the other hand, if the new solution is worse than the previous one, we perform binary search in the range of the current pre-allocation and the previous pre-allocation until the best lower-cost solution is found or no lower-cost solution can be found.

E. *Time Complexity*

The time complexity of the lookahead technique is $O(n \log n + nd_{max})$, where d_{max} is the maximum delay among all FTs. The $n \log n$ term comes from LS's time complexity, and the nd_{max} term from the fact that an FU executing an operation of FT k with delay d_k cc's, will be accessed d_k times to determine $a(i)$ and related parameters for lookahead processing. Further, if n_k is the number of operations of FT k and there are q FTs, fractional search will determine at most $O(\log n_k)$ new pre-allocations (and thus calls to lookahead scheduling) for FT k, and thus it overall makes $O(q \log n) \sim O(\log n)$ (q being a small constant compared to n) calls to lookahead scheduling. Thus, the total time complexity of FALLS is $O(n \log^2 n + (n \log n)d_{max}) = O(\max(n \log^2 n, (n \log n)d_{max}) \sim O(n \log^2 n)$ if d_{max} is a small constant.

IV. Experimental Results

We implemented FALLS in C++. Experiments were conducted on a machine with Core i7-4710HQ (3.5GHz) and16GB RAM. First, we make a direct comparison between

TABLE I: COMPARISON BETWEEN ACO AND FALLS

L_c Factor	idctcol (114, 164)					invert (333, 354)				
	ACO		FALLS		FU %	ACO		FALLS		FU %
	nFU	+, *	nFU	+, *	Improv	nFU	+, *	nFU	+, *	Improv
1.0	11	5, 6	11	6, 5	0.0%	47	25, 22	46	22, 24	2.1%
1.1	10	4, 6	10	6, 4	0.0%	42	23, 19	42	18, 24	0.0%
1.2	9	4, 5	9	5, 4	0.0%	36	20, 16	34	14, 20	5.6%
1.3	8	3, 5	8	5, 3	0.0%	34	19, 15	30	12, 18	11.8%
1.4	8	3, 5	7	4, 3	12.5%	30	17, 13	26	11, 15	13.3%
1.5	7	3, 4	7	4, 3	0.0%	28	16, 12	25	10, 15	10.7%
1.6	7	3, 4	6	4, 2	14.3%	26	15, 11	22	9, 13	15.4%
1.7	7	3, 4	6	4, 2	14.3%	25	14, 11	21	9, 12	16.0%
1.8	7	3, 4	5	3, 2	28.6%	23	13, 10	20	9, 11	13.0%
1.9	6	3, 3	5	3, 2	16.7%	23	13, 10	19	8, 11	17.4%
2.0	6	3, 3	5	3, 2	16.7%	22	13, 9	18	7, 11	18.2%
Avg	7.8	3.4, 4.5	7.2	4.3, 2.9	8.1%	30.5	17.1, 13.5	27.5	11.7, 15.8	9.8%

TABLE II
AVERAGE NUMBER OF FUS AND AREA RESULTS. AREA UNIT = 10^2
T. "*" MEANS A SET OF EXPERIMENTS RAN OUT OF MEMORY

	DFG	# of ops	LS		FDS		SA		ILP		FALLS	
			nFU	Area	nFU	Area	nFU	Area	nFU	Area	nFU	Area
1	hal	11	6.1	382.5	5.5	337.5	5.5	337.5	5.5	337.5	5.5	337.5
2	horner	18	6.5	355.8	5.7	309.9	5.3	315.0	5.1	300.0	5.1	300.0
3	arf	28	9.4	588.5	5.9	331.7	5.9	331.7	4.6	269.2	4.6	276.3
4	motion	32	20.9	1251.4	13.2	746.8	13.9	795.1	12.2	718.3	12.3	700.7
5	ewf	34	4.4	175.7	4.4	189.9	4.3	175.3	3.2	134.9	3.3	135.3
6	h2v2	51	10.3	356.2	7.2	293.9	7.4	228.9	6.3	218.8	6.5	233.8
7	feedback	53	21.5	1133.0	14.0	739.1	14.2	728.7	11.3	590.4	11.3	590.4
8	collapse	56	29.7	1595.7	13.5	800.1	13.7	764.7	11.4	684.3	11.6	692.0
9	write	106	70.4	2048.3	14.5	569.0	15.9	557.3	11.2	438.6	11.9	483.0
10	interpolate	108	41.8	2019.5	24.9	1256.4	27.9	1517.8	19.7	1055.9	20.2	1062.5
11	matmul	109	37.4	2188.5	20.4	1153.0	21.2	1224.6	15.5	931.5	16.3	956.9
12	idctcol	114	43.1	1826.9	22.3	990.8	15.8	738.0	10.8	497.0	12.1	533.2
13	jpeg	134	36.7	1617.8	25.9	1140.8	22.0	1108.8	14.6	703.3	15.4	704.9
14	smooth	197	75.9	3961.1	30.5	1737.6	31.6	1761.2	24.1	1403.2	26.2	1465.3
15	invert	333	100.2	5963.5	56.7	3328.9	55.9	3484.5	35.6	2216.7	39.5	2374.5
16	rand-1300	1300	520.7	59841.2	100.3	14271.1	115.7	16200.1	*	*	94.5	13961.7
DFG 1-15 Avg		92.3	34.3	1697.6	17.6	928.4	17.4	937.9	12.7	700.0	13.4	723.1
FALLS % Improv.			60.8%	57.4%	23.8%	22.1%	22.6%	22.9%	-5.5%	-3.3%	0.0%	0.0%
DFG 1-16 Avg		167.8	64.7	5331.6	22.8	1762.3	23.5	1891.8	*	*	18.5	1550.5
FALLS % Improv.			71.4%	70.9%	18.9%	12.0%	21.3%	18.0%	*	*	0.0%	0.0%

FALLS and ant colony optimization (ACO) in [9]. The trivial FU library in [9] has only two FTs of FUs: multiplier (d = 2 cc's) for multiplication and division, ALU (d = 1 cc), denoted for simplicity by "+", for the remaining FTs. For each DFG, the L_c is set to be a factor, called *L_c factor*, of the critical path delay. Perhaps due to the stochastic nature of ACO, it is hard to obtain consistent good results for all DFGs. Thus, to make a fair comparison, in Table I we compare FALLS to ACO results in [9] for two large DFGs which are the only ones for which FU allocation results with specified L_c's are presented in [9]. Across various L_c's, FALLS allocates an average of 8.1% and 9.8% fewer FUs for the two DFGs than ACO.

In Table II, we compare FALLS to LS, FDS [3] [4], SA [9] and ILP [1] [2] (implemented in CPLEX) for 15 DFG benchmarks from [15] and one randomly generated DFG rand-1300, to show its efficacy in FU minimization and the indirectly minimized total area. For this, we constructed a 16-bit non-trivial FU library that has eight FTs: adder/subtractor (d = 4 cc's), multiplier (d = 10 cc's), divider (d = 24 cc's), arithmetic and logical shift register (d = 1 cc), memory read and write (d = 1 cc), and logical AND (d = 1 cc). The delay and area of the FU designs from [16] are theoretically derived based on the number of gate inputs along the critical path and the total number T of transistors, respectively. Each number of FU (nFU) and area result in Table II is the average for a DFG for 11 L_c's with L_c factors in the range [1, 2] with a granularity of 0.1. The results show that FALLS reduces the total number of FUs by an average of 18.9% to 71.4% compared to LS, FDS and SA, and has similar area reductions. It also shows that FALLS has only a 5.5% optimality gap in the number of FUs and merely a 3.3% greater area compared to the optimal ILP method. We also performed the experiments with the trivial library used by [9]: FALLS' FU and area improvements range from 11.9% to 51.6% compared to the approximate algorithms while being only 1.2% worse than ILP. Mistakes made by the competing approximate algorithms are less costly in area in the trivial library as the area difference in the two types of FUs is much smaller than among FU types in the more complex library of Table II. This results in some shrinking of the % differences between the results of these techniques and FALLS compared to those for the complex library.

Further, though the following are not part of our optimization objective, we also determined the *architectural area* = the sum of the areas of FUs, mux's/demux's and registers, by using the left-edge algorithm [17] to bind operations to FUs (and thereby determine mux and demux sizes needed) and allocate registers post-binding. The results (not given per DFG in tables due to space constraints) show that FALLS has 32.4% to 60.8% average architectural area reduction compared to the competing approximate algorithms. Further, the average maximum congestion (max in + out degree of an FU) of FALLS is 3.5% to 14.7% smaller than these algorithms, and its average number of interconnects is 9.6% to 37.2% fewer. Also, FALLS has at most 4.0% more of the above architectural area and interconnect metrics compared to ILP. Similar results were obtained with the trivial library. These results show that a good FU minimization algorithm like FALLS is indirectly beneficial to other important architectural metrics.

Finally, the runtimes for the experiments of Table II show that FALLS is extremely fast, taking only 0.62 ms for the smallest and 69.85 ms for the largest DFG. The runtimes of SA and ILP are very high, preventing them from solving practical large-size problems. Across DFGs and L_c's, FALLS is merely about 3 times slower than the extremely fast but extremely sub-optimal LS, but is 68, 873 and 278k times faster than FDS, SA and ILP, respectively. Also, for the largest DFG rand-1300, CPLEX runs out of memory for even the smallest L_c (and thus smallest solution space). Considering that FALLS obtains solutions to the largest DFG with 1300 operations in a miniscule 69.85 milliseconds, and that it has an average optimality gap of only 5.5%, one can conclude that it has very good runtime and solution quality scalability.

V. Conclusions

We proposed a latency-constrained iterative list scheduling type algorithm FALLS to minimize the number of functional units (FUs) in high-level synthesis designs. We presented a novel lookahead technique to schedule some non-0-slack operations earlier to increase FU utilization or to reserve some currently available FUs for scheduling 0-slack operations in near-future clock cycles to avoid new FU allocations in them. Furthermore, a unique fractional search framework was developed to iteratively estimate the number of FUs required in the final design, and re-schedule with these initial FU allocations to further increase FU utilization and reduce the number of FUs. Extensive experiments demonstrated the significant effectiveness and efficiency of FALLS for FU area minimization, as well as for the beneficial side effects of reducing architectural area and important interconnect metrics.

References

[1] J.-H. Lee, Y.-C. Hsu and Y.-L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," *ICCAD,* Santa Clara, CA, Nov. 1989, pp. 20-23.

[2] C.-T. Hwang, J.-H. Lee and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *TCAD,* 1991.

[3] P. G. Pauline and J. P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," *DAC*, 1987.

[4] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *TCAD*, Jun. 1989.

[5] W. F. J. Verhaegh, E. H. L. Aarts and J. H. M. Korst, "Improved force-directed scheduling," in *EDAC*, Feb. 1991, pp. 430-435.

[6] W. F. J. Verhaegh, et al., "Efficiency improvements for force-directed scheduling," *ICCAD*, Santa Clara, CA, Nov. 1992, pp. 286-291.

[7] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," *DAC*. 2006.

[8] J. A. Nestor and G. Krishnamoorthy, "SALSA: a new approach to scheduling with timing constraints," *TCAD*, Aug. 1993.

[9] G. Wang, W. Gong, B. DeRenzi and R. Kastner, "Ant Colony Optimizations for Resource- and Timing-Constrained Operation Scheduling," *TCAD,* vol. 26, no. 6, pp. 1010-1029, Apr. 2007.

[10] M. J. M. Heijligers and J. A. G. Jess, "High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques," in *IEEE International Conference on Evolutionary Computation*, Perth, WA, Dec. 1995, pp. 56-61.

[11] A. Sharma and R. Jain, "InSyn: integrated scheduling for DSP applications," *IEEE Transactions on Signal Processing,* 1995.

[12] S. Gupta, N. Dutt, R. Gupta and A. Nicolau, "SPARK: a high-level synthesis framework for applying parallelizing compiler transformations," in *16th International Conference on VLSI Design*, New Delhi, India, Jan. 2003, pp. 461-466.

[13] S. J. Beaty, "List Scheduling: Alone, with Foresight, and with Lookahead," in *the First International Conference on Massively Parallel Computing Systems*, Ischia, Italy, May 1994, pp. 343-347.

[14] L. F. Bittencourt, R. Sakellariou and E. R. M. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Pisa, Italy, Feb. 2010, pp. 27-34.

[15] http://www.ece.ucsb.edu/EXPRESS/benchmark/.

[16] B. Parhami, Computer Arithmetic: Algorithms and Hardware Designs, Second Edition: Oxford University Press, Oct. 2009.

[17] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment withn large apertures," in *8th Design Automation Workshop*, Atlantic City, NJ, June 1971.