Force-Driven Minimum Latency Resource-Constrained Scheduling (ML-RCS) Algorithm

Silvia Bonenti and Please Lukau
ECE/CS 565: Physical Design Automation

Fall 2025

0.1 Introduction

High-Level Synthesis (HLS) involves transforming behavioral descriptions of digital systems into register-transfer level (RTL) implementations. One of the key steps in this process is *scheduling*, which determines when each operation in a data flow graph (DFG) should execute, given constraints on available hardware resources and latency requirements. The **Minimum Latency Resource-Constrained Scheduling (ML-RCS)** problem represents a more general and realistic form of the classical **Minimum Resource Latency-Constrained Scheduling (MR-LCS)** problem discussed in class.

From MR-LCS to ML-RCS

In the MR-LCS formulation each operation is assumed to have a fixed latency. The objective is to find an optimal schedule that minimizes the total number of functional units (FUs) of each type, while ensuring that the overall latency does not exceed a predefined latency constraint.

The ML-RCS problem, conversely, inverts the objective and the constraint. This formulation, which is often more representative of real-world synthesis scenarios, begins with a fixed resource budget (a maximum number of FUs of each type). Consequently, the scheduler must determine the start time for each operation in such a way as to minimize the total latency required for execution, while respecting data dependencies and the limited resource availability.

The Need for Dual Forces

To produce high-quality schedules under these conditions, the algorithm must balance two competing objectives:

1. **Resource Usage:** The utilization of each functional unit type must remain within its capacity limit, denoted a_k, for each cycle. Overutilization leads to resource contention and infeasible schedules.
2. **Latency Optimization:** The schedule must minimize the overall completion time (critical path length).

In classical Force-Directed Scheduling (FDS), a *resource force* is computed to distribute operations probabilistically across cycles to balance FU usage. In the ML-RCS context, this model must be extended: it must take into account at the same time of two different objective. A local one, related to managing the contention of resources in the current clock cycle, and a global one, related to the effects on the total downstream latency.

Motivation

By integrating resource and latency forces, the proposed algorithm aims to avoid locally optimal but globally suboptimal scheduling choices. For example, scheduling an operation early may reduce its own downstream delay but increase congestion for its resource type, potentially extending the critical path of successor operations. The combination of forces enables a more balanced distribution of operations in time, leading to better convergence between achievable latency and feasible resource utilization.

In summary, the ML-RCS problem captures a key challenge in modern digital synthesis: achieving timing closure while maintaining efficient hardware utilization. The algorithm developed in this report extends the FDS framework with probabilistic and congestion-aware reasoning to address these dual goals effectively.

0.2 Conceptual Design

The proposed algorithm is an iterative List Scheduler to handle the Minimum Latency Resource-Constrained Scheduling (ML-RCS) problem. Its design integrates both probabilistic reasoning about functional unit (FU) usage and delay-aware congestion analysis to achieve a balance between resource utilization and latency minimization.

Overview of the Algorithmic Structure

The algorithm follows an iterative refinement process, where each iteration consists of three major phases:

1. **Priority Computation:** Using current timing bounds, each operation is assigned a priority based on its urgency and congestion cost. This step captures both local and global scheduling constraints.
2. **List-Based Scheduling:** Operations are scheduled according to their computed priorities while respecting data dependencies and FU availability.
3. **Latency Update:** After each iteration, the achieved schedule latency L_{final} is compared against the target latency L_{target}. The target is then updated and the process repeats until convergence.

Iterative Latency Adjustment

The iterative structure allows the algorithm to progressively approach a feasible and efficient schedule. Initially, the target latency L_{target} is set optimistically to the critical path length obtained from an ASAP analysis. During each iteration, the list scheduler produces a new schedule that reflects the current balance between resource constraints and delay forces. If the resulting latency L_{final} differs from L_{target}, the target is updated to the new value, and a fresh set of probabilities and forces is computed. This feedback mechanism continues until the algorithm stabilizes, typically within a small number of iterations.

Algorithm's Forces

This design implicitly unites the two principal forces of the ML-RCS formulation: (1) (1) the latency force, captured directly by the operation's scheduling mobility, and (2) the resource force, derived from a non-myopic, probabilistic analysis of functional-unit congestion, both of which interact through the unified priority function $F(u)$.

0.3 Formal Definitions

This section defines the mathematical quantities used throughout the algorithm. Each term characterizes a fundamental concept in high-level scheduling, from time-frame bounds to probabilistic resource utilization and priority computation.

ASAP and ALAP Times

For each operation u in the data-flow graph (DFG), the algorithm computes two timing bounds:

- **ASAP time:** The earliest possible start cycle of operation u, given data dependencies:
$$ASAP(u) = \max_{v \in Pred(u)} \big(ASAP(v) + Latency(v)\big)$$
where $Pred(u)$ denotes the set of immediate predecessors of u.
- **ALAP time:** The latest possible start cycle of u without exceeding the target latency L_{target}:
$$ALAP(u) = L_{\text{target}} - DownLen(u) + 1$$
where $DownLen(u)$ is the longest path length from u to the SINK node.

The range $[ASAP(u), ALAP(u)]$ defines the feasible time frame during which operation u can be scheduled, also called **Mobility Range**.

Resource Utilization Probability

For each functional unit (FU) type k, the algorithm maintains a probabilistic occupancy profile across all clock cycles m:

$$q_k(m) = \sum_{u\,:\,type(u)=k} p_u(m)$$

where $p_u(m)$ denotes the probability that operation u executes in cycle m. These probabilities are assumed to be simple uniform distributions, derived from the mobility range of each operation:

$$p_u(m) = \begin{cases} \dfrac{1}{ALAP(u) - ASAP(u) + 1}, & \text{if } ASAP(u) \le m \le ALAP(u) \\ 0, & \text{otherwise} \end{cases}$$

The term $q_k(m)$ thus reflects the expected utilization of FU type k at cycle m.

Average Congestion Along the Critical Path

The congestion cost $C(u)$ quantifies the cumulative contention expected along the critical successor path of operation u:

$$C(u) = \frac{1}{|P(u)|} \sum_{v \in P(u)} \frac{q_{type}^{max}(v)}{a_{type}(v)}$$

where $P(u)$ is the set of operations along the critical path starting from u. $q_{type}^{max}(v)$ is the maximum Probability distribution of the operation along its mobility range. $a_{type(v)}$ denotes the number of available units of that type. This averaging captures the expected delay propagation effects that arise when downstream resources are overutilized, ensuring that operations with high downstream congestion receive lower scheduling priority.

Why a Non-Myopic Congestion Cost? An Example

To illustrate the weakness of a myopic scheduler, consider the Data Flow Graph (DFG) in Figure 1. This graph presents two parallel paths with different length (`s-a-b-c-g-z`: 4 cycles, `s-d-e-f-z`: 3 cycles) that will compete for a single ALU resource.

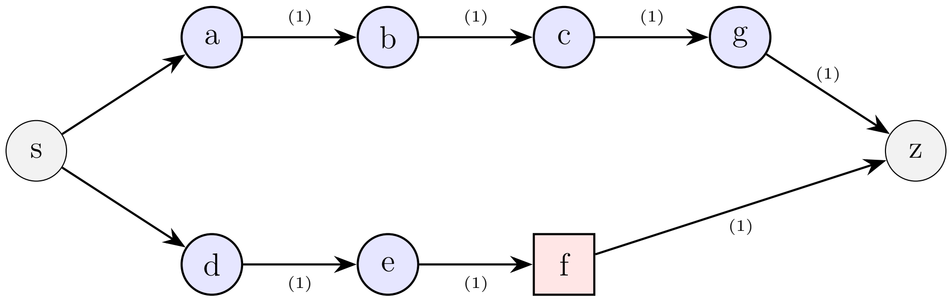


Figure 1: A test DFG with two different paths competing for one ALU. Path 1 is a pure 4-ALU chain. Path 2 is a 2-ALU, 1-MUL chain.

In Cycle 1, the `ReadyList` contains `a` and `d`. The choice of the starting node between those two is not important, as it can be seen that it is not what makes the difference in the long run. Below is the analysis of the two different approaches

Case 1: Myopic Scheduling ("Luck")

A myopic scheduler, will see `C(a)` = `C(d)`, so it will choose choose depending on the `S` value, which is lower for `a` since it is on the critical path. Since the values of `C` for `a,b,c,d` are the same, the priority is given to the nodes on the critical path (lower `S` value)), thus executing the whole critical path (along with `g`, since `C(g)` is 0). As a consequence, as it can be seen in Figure 2, the entire `d` path is starved, and its MUL operation (`f` node) is pushed to the very end. This results in a **final latency of 7 cycles**.

Case 2: Non-Myopic Scheduling (Our Algorithm)

In this case, the nodes `a,b,c` have the highest `C` values, being the path full of ALUs with an high congestion. As a consequence, there will not be a single path prevailing on the other, but they will operate in an interleaved way. In particular, the non-critical path is helped by having a congestion-free node at its end, the `f` node. In this way, as it can be seen in Figure 3, the parallelism between resources is exploited: once `f` starts executing, also `a` can be executed at the same time, resulting in the end in a **final latency of 6 cycles**.

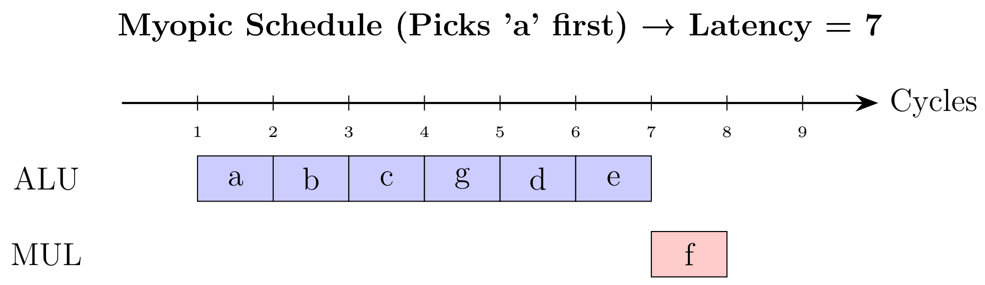


Figure 2: A Myopic scheduler executes the path `a` first. This choice serializes the execution of the two main paths, resulting in a suboptimal latency of 7 cycles.

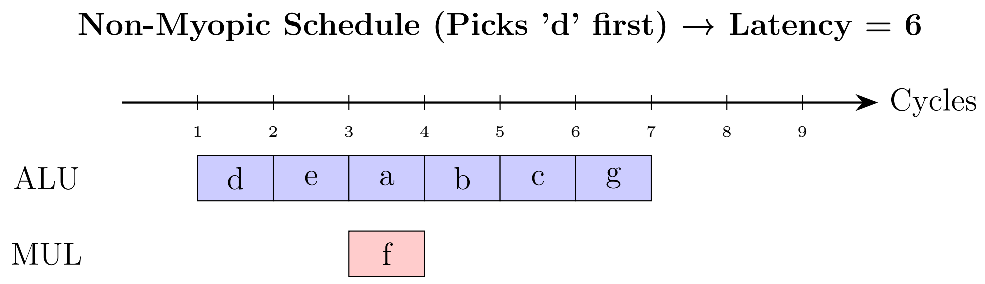


Figure 3: Our Non-Myopic scheduler correctly identifies `d` as the higher priority task. This choice allows the MUL operation `f` to be scheduled in parallel ("hidden"), resulting in a faster schedule (6 cycles vs. 7).

Priority Function

The final priority function $F(u)$ integrates both the timing slack and congestion cost into a unified measure:

$$F(u) = S_{norm}(u) \times \big(C_{norm}(u) + \varepsilon\big)$$

where $S_{norm}(u)$ and $C_{norm}(u)$ are the normalized slack and congestion terms respectively, and ε is a small constant added for numerical stability. Lower values of $F(u)$ correspond to higher scheduling priority. This formulation encourages early scheduling of operations that are both time-critical (low mobility) and located in low-congestion regions of the graph, leading to balanced and delay-minimal solutions.

These formal definitions form the foundation of the proposed ML-RCS scheduling algorithm, linking the probabilistic and latency-aware aspects of force-directed scheduling within a unified mathematical framework.

0.4 Algorithm Description

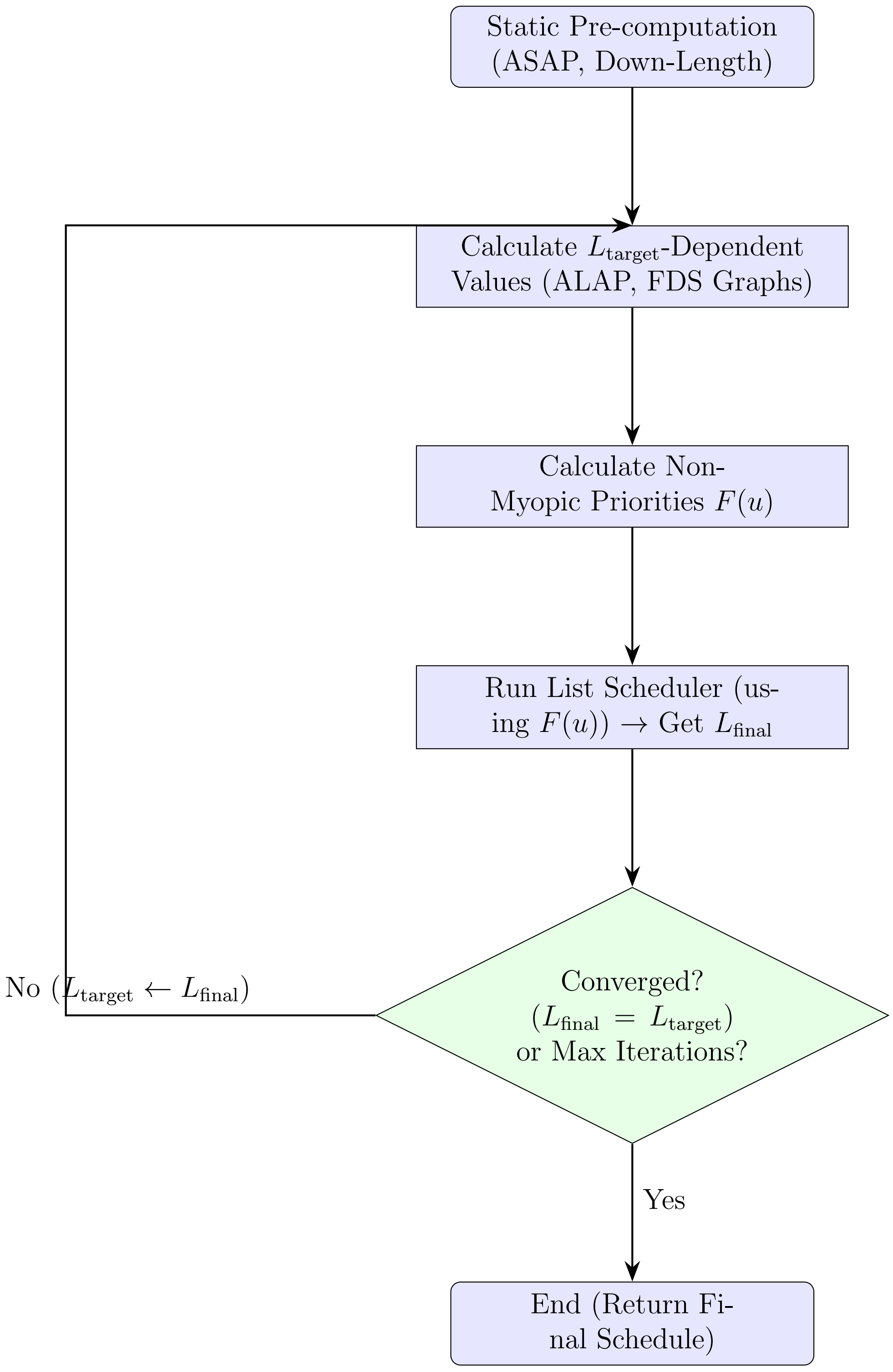
This section provides a detailed description of the proposed Force-Directed Scheduling (FDS)-based algorithm for solving the Minimum-Latency Resource-Constrained Scheduling (ML-RCS) problem. The algorithm integrates probabilistic resource modeling with delay-aware congestion estimation to produce a balanced and feasible schedule under limited functional unit (FU) availability.

Overview

The algorithm operates in four major stages:

1. **Main Scheduler:** Coordinates the iterative refinement loop, progressively adjusting the target latency until convergence.
2. **Priority Calculation:** Computes operation priorities based on both slack (urgency) and congestion forces.
3. **Congestion Cost:** Estimates the cumulative congestion along each operation's critical path, capturing non-local timing effects.
4. **List Scheduler:** Schedules operations according to computed priorities, while enforcing resource and dependency constraints.

High-level Flowchart



Static Pre-computation (ASAP, Down-Length)
Calculate L_{target}-Dependent Values (ALAP, FDS Graphs)
Calculate Non-Myopic Priorities $F(u)$
Run List Scheduler (using $F(u)$) $\to$ Get L_{final}
Converged? ($L_{\text{final}} = L_{\text{target}}$) or Max Iterations?
No ($L_{\text{target}} \leftarrow L_{\text{final}}$)
Yes
End (Return Final Schedule)

Figure 4: The high-level flowchart of the iterative scheduling algorithm.

Main Scheduler

The `Main_Scheduler` function performs an initial static analysis, followed by iterative refinement of the target latency. Each iteration recomputes priorities and runs the list scheduler until the achieved latency stabilizes.

```
FUNCTION Main_Scheduler(operations, dependencies, resources)
    // Initial Setup done only at start:
    // ASAP, Initial Target Latency and down stream lenght are computed
    graph <- Build_Graph_With_Source_Sink(operations, dependencies)
    Calculate_ASAP(graph)
    Calculate_Down_Length(graph)
    L_crit <- graph.SINK.ASAP

    // Iterative  Scheduling Loop
    L_target <- L_crit
    L_final <- 0
    final_schedule <- NULL

    // Loop until convergence of latency is reached
    WHILE (L_target != L_final) AND (under max iterations)
        IF this is not the first loop THEN
            L_target <- L_final
        ENDIF

        // Redo priorities calculation at each iteration
        priorities <- Calculate_All_Priorities(graph, resources, L_target)

        // Run LS using the new priority values
        (L_final_new, schedule) <- Run_List_Scheduler(graph, resources, priorities)

        // Update current iteration results
        L_final <- L_final_new
        final_schedule <- schedule
    ENDWHILE

    RETURN final_schedule, L_final
ENDFUNCTION
```

This loop allows the scheduler to progressively adjust the timing horizon. If the achieved latency exceeds the target, the target is increased, and new forces are recalculated until convergence.

Priority Calculation

This step combines mobility range and congestion into a unified force metric. Probabilities are computed for each operation's possible execution cycles, and a congestion-aware force is derived.

```
FUNCTION Calculate_All_Priorities(graph, resources, L_target)
    // Compute ALAP values for each node
    Calculate_ALAP(graph, L_target)
    // Compute resource probabilities distributions across cycles
    dist_graphs <- Calculate_FDS_Graphs(graph)

    // 3. Compute average congestion cost per path
    Calculate_Congestion_Cost(graph, dist_graphs, resources)

    // Computer raw priority terms
    // S = mobility + 1
    // C = congestion cost along critical successor path
    FOR each node u (excluding SOURCE/SINK)
        S_raw(u) <- (u.ALAP - u.ASAP) + 1
        C_raw(u) <- u.congestion_cost
    ENDFOR

    // Compute normalized priority terms
    S_norm <- Normalize(all_S_raw_values)
    C_norm <- Normalize(all_C_raw_values)

    // Calculate final priority value for each node
    FOR each node u
        Priority(u) <- S_norm(u) * (C_norm(u) + EPSILON)
    ENDFOR

    RETURN all Priority values
ENDFUNCTION
```

The computed priorities reflect both timing flexibility and expected congestion. A lower priority value corresponds to higher scheduling urgency.

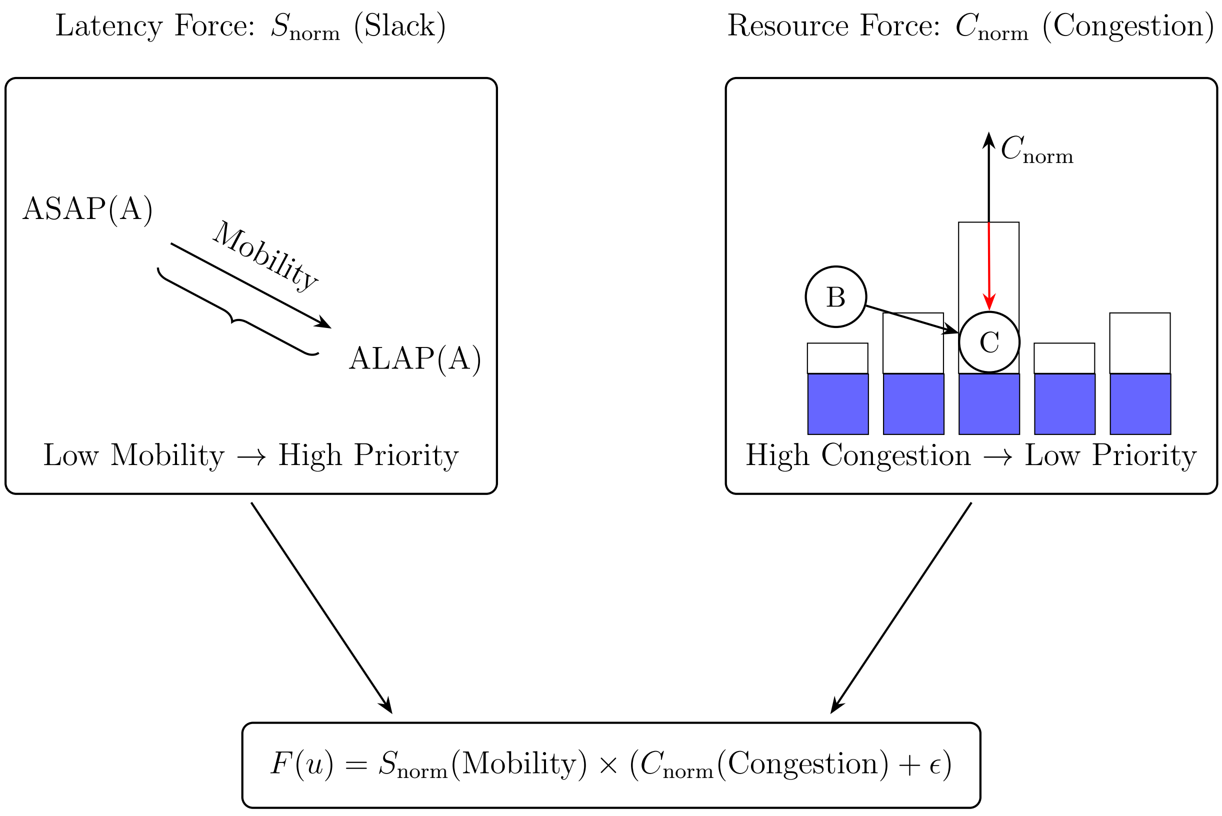

Latency Force: S_{norm} (Slack)
Resource Force: C_{norm} (Congestion)
ASAP(A)
Mobility
ALAP(A)
Low Mobility → High Priority
C_{norm}
B
C
High Congestion → Low Priority
$F(u) = S_{\text{norm}}(\text{Mobility}) \times (C_{\text{norm}}(\text{Congestion}) + \epsilon)$

Figure 5: The "Dual-Force" Priority Function ($F(u)$). The final priority is the product of two competing forces. The Latency Force (S_{norm}) is derived from the operation's mobility (or slack), prioritizing nodes on tight time-frames. The Resource Force (C_{norm}) is a probabilistic measure of congestion, penalizing nodes that must be scheduled in highly-contended resource cycles.

Congestion Cost Estimation

The congestion cost captures the *non-myopic* effect—how congestion in downstream nodes affects the critical path. This step processes nodes in reverse topological order, from SINK to SOURCE.

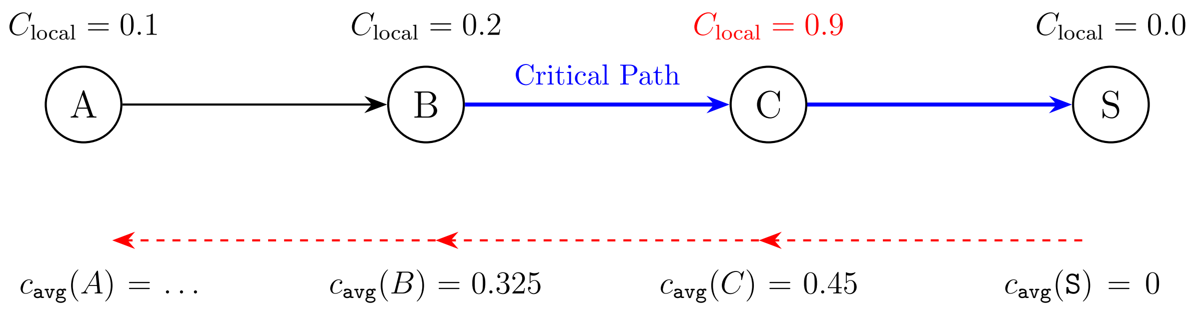
```
FUNCTION Calculate_Congestion_Cost(graph, dist_graphs, resources)
    FOR each node u (starting from the SINK)
        C_local(u) <- 0.0

        // Compute congestion for current node (q_k/a_k)
        IF u is not SOURCE or SINK THEN
            q_max <- Find max probability (q) for u's resource
            C_local(u) <- q_max / (number of available resources)
        ENDIF

        // Compute congestion for critical successor node
        // Keep track of critical path length, for final average computation
        v <- u.critical_successor
        IF v exists THEN
            C_inherited_sum <- v.c_path_sum
            inherited_len <- v.c_path_len
        ENDIF

        // Update congestion and path length for current node
        u.c_path_sum <- C_local(u) + C_inherited_sum
        u.c_path_len <- 1 + inherited_len

        // Compute congestion cost for current node
        u.congestion_cost <- u.c_path_sum / u.c_path_len
    ENDFOR
ENDFUNCTION
```



$C_{\text{local}} = 0.1$
$C_{\text{local}} = 0.2$
$C_{\text{local}} = 0.9$
$C_{\text{local}} = 0.0$
Critical Path
A
B
C
S
$c_{\text{avg}}(A) = \ldots$
$c_{\text{avg}}(B) = 0.325$
$c_{\text{avg}}(C) = 0.45$
$c_{\text{avg}}(\texttt{S}) = 0$

Figure 6: Explcit Example: Non-myopic congestion cost calculation (C_{avg}). Costs are computed in reverse topological order (from SINK to SOURCE), allowing bottlenecks (like the high local congestion of node C) to propagate backwards along the critical path and influence the priority of upstream nodes (like B).

This reverse traversal ensures that each operation's cost reflects both its own resource contention and that of its successors, producing a more globally informed scheduling heuristic.

List Scheduler

The list scheduler executes operations cycle-by-cycle while ensuring resource constraints and dependency satisfaction.

```
FUNCTION Run_List_Scheduler(graph, resources, priorities)

    // Initialization: start from CC 1 with all resources available,
    // Successors of the source in the ready list and InProgressList initialized to a
    CurrentCycle <- 1
    available_resources <- copy of all resources
    ReadyList <- SOURCE's successors
    InProgressList <- empty list

    // Loop until there is no operation left to schedule
    WHILE (SINK not finished)
        // Free finished operations
        FOR each op finished before this cycle
```

```
            Free resources, mark successors ready
        ENDFOR

        // Schedule ready operations under resource constraints

        // Sort the list of ready operations by our custom priority force
        CandidateList <- ReadyList sorted by priority
        FOR each op in CandidateList
            IF resource available THEN
                Schedule op at CurrentCycle
                Occupy resource, mark operation in progress
            ENDIF
        ENDFOR

        CurrentCycle <- CurrentCycle + 1
    ENDWHILE

    // Calculate final obtained latency
    L_final <- SINK.start_time - 1
    RETURN (L_final, final schedule)
ENDFUNCTION
```

This cycle-based list scheduling ensures that the most urgent and least-congested operations are prioritized, leading to efficient resource usage and minimized total latency.

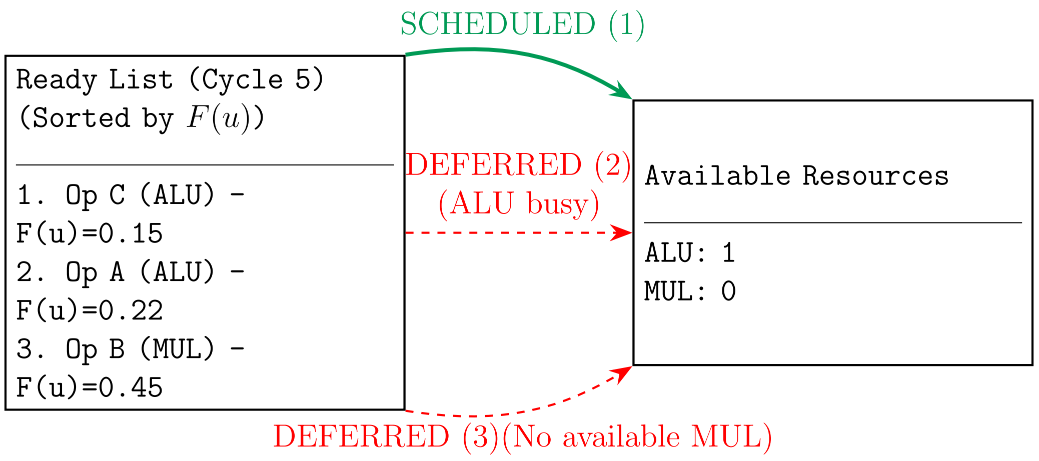

SCHEDULED (1)
Ready List (Cycle 5)
(Sorted by $F(u)$)
1. Op C (ALU) - F(u)=0.15
2. Op A (ALU) - F(u)=0.22
3. Op B (MUL) - F(u)=0.45
DEFERRED (2)
(ALU busy)
Available Resources
ALU: 1
MUL: 0
DEFERRED (3)(No available MUL)

Figure 7: A snapshot of the list scheduler in a single cycle: operations from the ready list are scheduled if the requested resource is available.

In summary, the algorithm combines local and global forces to iteratively refine schedules under minimum-latency constraints. Probabilistic resource balancing, congestion propagation, and iterative feedback allow the algorithm to heuristically search for high-quality timing solutions.

0.5 Time Complexity Analysis

To estimate the computational cost of the proposed scheduling algorithm, we use the following notation:

- N: number of operations (nodes) in the data-flow graph (DFG),
- E: number of dependencies (edges) in the DFG,
- K: number of iterations of the outer loop in `Main_Scheduler` (typically small),
- $L_{\max}$: maximum latency of the final schedule, i.e., number of clock cycles.

5.1 Main_Scheduler

The `Main_Scheduler` performs a one-time setup followed by at most K refinement iterations.

Setup phase.

- `Build_Graph_With_Source_Sink`: visits each node and edge once to construct the DFG $\Rightarrow O(N + E)$.
- `Calculate_ASAP`: forward traversal of the DFG $\Rightarrow O(N + E)$.
- `Calculate_Down_Length`: reverse traversal of the DFG $\Rightarrow O(N + E)$.

Each of these is linear in the graph size, so the one-time setup cost is:

$$T_{\text{setup}} = O(N + E).$$

Iterative phase. In each iteration, `Main_Scheduler` calls:

- `Calculate_All_Priorities`,
- `Run_List_Scheduler`.

If we denote the costs of these two routines by $T_{\text{Priorities}}$ and $T_{\text{Scheduler}}$, respectively, then the total cost of all iterations is:

$$T_{\text{iter}} = K \times \big(T_{\text{Priorities}} + T_{\text{Scheduler}}\big).$$

5.2 Calculate_Congestion_Cost

This function calculates the non-myopic congestion cost (C_{avg}) for all nodes.

- The function is built on a `reverse topological sort` (using a queue and `out_degree`), which visits each node and each edge once $\Rightarrow O(N + E)$.
- *However*, inside the main traversal loop (which executes N times, once per node), the function performs a second internal loop to find the congestion peak $q_{\max}$. This loop iterates from `ASAP` to `ALAP` and is bounded by $L_{\max} \Rightarrow O(N \times L_{\max})$.

The function's total cost is the sum of these two workloads: the graph traversal cost ($O(N + E)$) plus the total cost of all internal $q_{\max}$ searches ($O(N \times L_{\max})$).

$$T_{\text{congestion}} = O(N + E) + O(N \times L_{\max}) = O(N \times L_{\max} + E)$$

5.3 Calculate_All_Priorities

This stage computes the priority $F(u)$ for each node under the current target latency L_{target}.

(1) ALAP computation. `Calculate_ALAP` performs a reverse topological pass over all nodes and edges. Each node and edge is processed a constant number of times:

$$O(N + E).$$

(2) FDS probability graphs. For each operation, its time frame spans at most $L_{\max}$ clock cycles. Distributing or updating probabilities $p_u(m)$ over $[ASAP(u), ALAP(u)]$ costs $O(\text{width of time frame}) \leq O(L_{\max})$ per node, hence:

$$O(N \times L_{\max}).$$

(3) Congestion cost computation. The congestion cost is computed by a reverse traversal of the DFG. The cost has been computed before to be:

$$O(N \times L_{\max} + E)$$

(4) Normalization and priority combination. Slack values and congestion values are normalized and combined: this is a linear pass over all nodes:

$$O(N).$$

Combined complexity. Dominated by the congestion cost computation, the complexity of `Calculate_All_Priorities` is:

$$T_{\text{Priorities}} = O(N \times L_{\max} + E).$$

5.4 Run_List_Scheduler

The list scheduler advances cycle by cycle until all operations, including the SINK, are scheduled.

- The outer loop runs for at most $L_{\max}$ cycles.
- In each cycle:
  - Updating finished operations, ready list and schedule candidate take all $O(N)$
  - The dominant per-cycle work is sorting the ready list. The ready list can contain up to $O(N)$ operations, so sorting costs $O(N \log N)$ in the worst case for that cycle, thus dominating the $O(N)$ terms.

Hence, the total time complexity of this function is:

$$T_{\text{Scheduler}} = O(L_{\max} \times N \log N).$$

5.5 Overall Time Complexity

Combining setup and all iterations:

$$\begin{aligned}
T_{\text{Total}} &= T_{\text{setup}} + T_{\text{iter}} \\
&= O(N+E) + K \times \big(T_{\text{Priorities}} + T_{\text{Scheduler}}\big) \\
&= O(N+E) + K \times \Big((NL_{\max} + E) + (L_{\max} N \log N)\Big).
\end{aligned}$$

Retaining the dominant terms (for large N and $L_{\max}$):

$$T_{\text{Total}} = O\big(K \times \big(L_{\max} N \log N + E\big)\big)\,.$$

5.6 Space Complexity

The algorithm requires storage for:

- the graph structure and dependency lists: $O(N+E)$,
- ASAP, ALAP, down-length, priority, and congestion arrays: $O(N)$,
- distribution graphs for resource usage profiles over the horizon (assuming a constant number of FU types): $O(L_{\max})$.

Therefore, the total space complexity is:

$$\text{Space} = O(N + E + L_{\max}).$$

5.7 Discussion

The runtime is pseudo-polynomial, because it does not only depend on the input graph size (N, E) (polynomial), but also on the numeric parameter $L_{\max}$, which reflects the final schedule length. The multiplicative factor K accounts for the outer refinement loop; in practice K is small (e.g., 1–3), so the iterative overhead is modest.

The most expensive components are:

- the probability and congestion computations over the scheduling horizon $\big(O(NL_{\max})\big)$,
- the cycle-by-cycle list scheduling with sorting $\big(O(L_{\max} N \log N)\big)$.

This matches expectations for an iterative list scheduler that incorporates a computationally expensive, pseudo-polynomial priority function.

0.6 Theoretical Results

This section states basic correctness and structural properties of the proposed ML-RCS scheduling algorithm. These results are intentionally modest: the general resource-constrained scheduling problem is NP-hard, so we do not claim global optimality. Instead, we show that the algorithm always produces a valid schedule, terminates, and is optimal in a simple but important special case.

0.7 Problem Formulation

Given a Data-Flow Graph $G = (V, E)$, a set of resource types K, and a resource limit a_k for each type, the ML-RCS problem tries to find a start time t_i for each operation $v_i \in V$ minimizing the total schedule latency, subject to all constraints.

Define x_{il} as a binary decision variable that is 1 if operation v_i starts at cycle l, and 0 otherwise. Define d_i as the latency (duration) of operation v_i.

Then, a feasible schedule must satisfy the following constraints:

1. **Unique Start:** Each operation must be scheduled exactly once.

$$\sum_l x_{il} = 1 \quad \forall v_i \in V$$

2. **Precedence (Dependency):** An operation v_i can start only after all of its predecessors $v_j \in Pred(v_i)$ have completed.

$$\sum_l l \cdot x_{il} \geq \sum_l l \cdot x_{jl} + d_j \quad \forall (v_j, v_i) \in E$$

3. **Resource:** At any cycle l, the number of active operations of type k must not exceed the available resource limit a_k.

$$\sum_{i:T(v_i)=k} \sum_{m=l-d_i+1}^{l} x_{im} \leq a_k \quad \forall l, \forall k \in K$$

The `Run_List_Scheduler` algorithm is a heuristic method for finding a high-quality, feasible solution to this NP-hard problem. Its properties are analyzed below.

Feasibility of the Schedule

Claim 1. The schedule returned by `Run_List_Scheduler` is feasible with respect to (i) precedence constraints and (ii) resource constraints.

Proof.

- *Precedence constraints.* An operation u is inserted into the `ReadyList` only after all its predecessors have been marked finished. The scheduler selects operations to execute exclusively from the `ReadyList`. Therefore, by construction, every scheduled operation starts at a control step strictly greater than or equal to the completion steps of all its predecessors. Hence, all data dependencies are respected.

- *Resource constraints.* For each functional unit (FU) type k, the algorithm maintains a counter of currently available units. An operation of type k is scheduled in a given cycle only if the corresponding availability counter is strictly positive; upon scheduling, this counter is decremented, and it is incremented again when the operation finishes. Since at most one unit is consumed per decrement and the counter never becomes negative, the number of simultaneously active operations of type k never exceeds a_k. Thus, all resource constraints are satisfied.

Therefore, the returned schedule is precedence-feasible and resource-feasible. □

Termination of the Iterative Scheme

Claim 2. The outer loop in `Main_Scheduler` terminates.

Proof. Termination is guaranteed by the explicit upper bound on the number of iterations (the `MAX_ITERATIONS` condition). This ensures the algorithm stops, regardless of whether the scheduling heuristic converges: the algorithm's ideal stopping condition is $L^{(i)}_{\text{final}} = L^{(i)}_{\text{target}}$. Since the list scheduler and priority computation are deterministic, a given L_{target} will always produce the same L_{final}. However, the sequence of latencies $\{L^{(i)}_{\text{target}}\}$ is not guaranteed to be monotonic and could theoretically oscillate between a finite set of values. Therefore, the `MAX_ITERATIONS` bound is the true necessary guarantee of termination. □

Lower Bound Respect

Claim 3. The algorithm never returns a latency smaller than the critical path length.

Proof. The ASAP and down-length computations ensure that for each operation u, its earliest feasible start time and its contribution to any path respect data dependencies. The SINK node's ASAP time equals the critical path length L_{crit}. Any valid schedule must assign start times that respect these dependencies, so its latency is at least L_{crit}. Since `Run_List_Scheduler` produces only precedence-feasible schedules (**Claim 1**), the returned latency L_{final} satisfies $L_{\text{final}} \geq L_{\text{crit}}$. □

A Special-Case Optimality: Optimality in the unconstrained case

Claim 4. If the resource bounds a_k are sufficiently large such that, for every cycle, all ready operations of type k can be scheduled without conflict, then the algorithm produces an optimal schedule with latency equal to the critical path length.

Proof. In the absence of effective resource constraints, the list scheduler is free to schedule every ready operation as early as its dependencies allow. Because the algorithm never violates precedence constraints and always schedules ready operations at the earliest available cycle when resources are plentiful, the resulting schedule coincides with the ASAP schedule. The latency of this schedule equals the critical path length L_{crit}, which is the lower bound for any feasible schedule. Therefore, in this special case, the algorithm is optimal. □

Discussion

These properties formalize the behavior of the proposed ML-RCS algorithm: it always returns a valid schedule, it terminates, it respects fundamental lower bounds, and it is provably optimal when resource constraints do not bind. In the general constrained setting the method remains heuristic, but its dual-force design, which balances scheduling mobility against probabilistic resource congestion, is theoretically aligned.

0.8 Conclusion

This report presented an Iterative List Scheduling algorithm for solving the Minimum Latency Resource-Constrained Scheduling (ML-RCS) problem. The proposed approach borrows concepts from Force-Directed Scheduling (FDS), such as probabilistic resources modeling, to construct a non-myopic priority function. Through this integration, the algorithm effectively balances two competing goals in high-level synthesis: minimizing overall latency while maintaining feasible utilization of functional units. The design introduces a priority function that unifies two forces:

- the *latency force*, captured by the operation's scheduling mobility (slack)
- the *resource force*, derived from a probabilistic and non-myopic analysis of functional-unit congestion along the critical path.

This priority function guides a standard list scheduler, while an outer iterative refinement loop—driven by updates to the target latency L_{target}—allows the algorithm to heuristically search for a high-quality schedule.

Theoretical analysis established that the algorithm:

1. always produces a valid and resource-feasible schedule,
2. is guaranteed to terminate,
3. respects the critical-path lower bound, and
4. achieves optimality when resources are unconstrained.

Its pseudo-polynomial computational complexity, $T_{\text{Total}} = O(K \times (L_{\max} N \log N + E))$, and space complexity, $O(N + E + L_{\max})$, are consistent with iterative, probabilistic schedulers more complex than simple list-based heuristics. Ultimately, this algorithm is a practical tool for scheduling. It finds high-quality (though not perfect) answers.