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

Silvia Bonenti and Please Lukau

ECE/CS 565: Physical Design Automation

Fall 2025

Outline

Introduction: High-Level Synthesis and Scheduling

- High-Level Synthesis (HLS) converts algorithmic descriptions into RTL hardware.
- **Scheduling** decides *when* each operation in a Data Flow Graph (DFG) executes.
- Performance depends on:
    - **Latency** – total execution time.
    - **Resource usage** – number of functional units (FUs).
- The **ML-RCS problem** generalizes the classical **MR-LCS** formulation.

The Need for Dual Forces

- Scheduling in HLS must balance two competing goals:
  1. **Resource usage:** keep FU utilization $\leq a_k$ per cycle.
  2. **Latency optimization:** minimize overall completion time.
- Classical Force-Directed Scheduling (FDS) considers only **resource forces** to spread operations over time.
- This scheduling algorithm for ML-RCS introduces an additional **latency force** to prioritize operations on the critical path.
- Combined, these two forces guide the scheduler to produce feasible yet fast schedules.

Key Idea

Resource force prevents overuse of functional units, while **latency force** reduces critical-path delay. Their interaction defines a balanced scheduling priority.

Iterative List Scheduling Framework

- This scheduling algorithm extends list scheduling into an iterative refinement loop.
- Each iteration includes three core phases:
    1. **Priority Computation** — derive urgency from mobility and congestion.
    2. **List-Based Scheduling** — assign operations respecting FU limits.
    3. **Latency Update** — compare L_{final} and L_{target}.
- The process repeats until latency converges or the iteration limit is reached.

Essence

Priority → *Schedule* → *Update* → *Repeat* ensures progressively improved latency-aware scheduling.

Algorithm Forces

- The algorithm defines two interacting forces for each operation:
  1. **Latency Force (S)** — measures timing urgency (low mobility ⇒ high priority).
  2. **Resource Force (C)** — measures congestion impact (high usage ⇒ penalized).
- Both forces combine into a unified priority:

$$F(u) = S_{\text{norm}}(u) \cdot \left(C_{\text{norm}}(u) + \varepsilon \right)$$

- Lower $F(u)$ means higher scheduling urgency.

Summary

The dual-force interaction balances delay minimization and resource feasibility within a single priority metric.

Summary

- Proposed a suboptimal **Iterative List Scheduling** algorithm for the **ML-RCS problem.**
- Combines Force-Directed Scheduling (FDS) concepts with a dual-force priority model:
  - *Latency force* — urgency from scheduling mobility.
  - *Resource force* — probabilistic, non-myopic FU congestion.
- Iterative refinement of L_{target} balances delay and resource use.

Proven properties:

1. Produces valid, resource-feasible schedules.
2. Guaranteed termination.
3. Respects the critical-path lower bound.
4. Optimal when resources are unconstrained.

Complexity:

$$T_{\text{Total}} = O(K(L_{\max} N \log N + E)), \qquad \text{Space} = O(N + E + L_{\max})$$

Where we left off: Algorithm Limitations

- **Suboptimal by Design (Greedy Heuristic):**
  - Scheduling is NP-complete; we aim for a good solution, not a guaranteed optimal one.
  - Our algorithm is a **greedy** List Scheduler: it makes the best greedy choice (based on our heuristic mixing local urgency and global cost) and **never backtracks** to fix strategic mistakes.
- **Heuristic Limitations (e.g., False Ties):**
  - The priority formula ($F = S \cdot C$) is an estimate, not a perfect model of cost.
  - It can create *false ties* (like a vs. d in Example. 2), forcing an arbitrary (e.g., alphabetical) tie-break.

Requested Adjustments

- **Non-Myopic Congestion Cost**
  - Modified the Congestion Cost $C(u)$ calculation to use the **average expected usage** q^{avg} instead of the peak usage q^{max} over the critical path $P(u)$.
  - **Why?** Average gives a better estimate of the expected resource usage with respect to the worst case given by taking the maximum.
  - The new Congestion Cost $C(u)$ formula is the following:

$$C(u) = \frac{1}{|P(u)|} \sum_{v \in P(u)} \frac{q^{avg}_{type(v)}}{a_{type(v)}}$$

Requested Adjustments

- **Probability Distribution**
  - The calculation of the probabilistic FU usage $q_k(m)$ was corrected to accurately model operations with $\mathrm{Latency}(u) > 1$.
  - **Why?** While an operation has a uniform probability of starting in any cycle within its mobility window $[\mathrm{ASAP}, \mathrm{ALAP}]$, its actual occupation time of the resource lasts for *L* cycles.
  - **Impact:** When we sum these individual probabilities, the effective resource utilization profile $q_k(m)$ becomes non-uniform and peaked in the middle of the operation's time range, accurately reflecting the higher expected congestion in those central cycles. This provides a realistic basis for the Congestion Cost $C(u)$.

Requested Adjustments

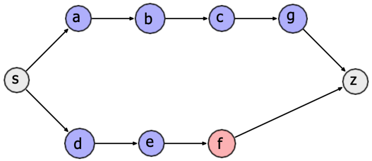
- **Relaxed Latency Initialization**
  - The outer loop now starts from a relaxed upper bound ($L_{target} \leftarrow$ LatencyParameter $\times$ $L_{standard-LS}$, where LatencyParameter $= 1.5$ and iteratively decrements the target latency by ($L_{target} \leftarrow L_{current-run} - 1$) to find the minimum feasible solution. If no solution is found, L_{target} is multiplied again by LatencyParameter, and the loop is run again, until a valid solution is found.
  - **Why?** Using an upper bound that can only be decreased, prevents oscillating behaviors and allows for a controlled convergence path

Requested Adjustments

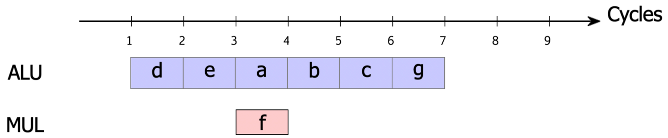
- **Relaxed Latency Initialization: Pseudo Code of Control Structure**

```
Calculate Upper_Bound_Latency using LS and multiply by LatencyParameter
Calculate Lower_Bound_Latency using ASAP
Repeat
    Success ← false
    Target_Latency ← Upper_Bound_Latency
    Repeat
        Run LS with current Target_Latency
        If valid_schedule is found:
            Save current result as Best
            Success ← true
            Target_Latency ← Current_Run_Latency - 1
    Until no valid_schedule or Target_Latency ≤ Lower_Bound_Latency
    If no valid_schedule found:
        Lower_Bound_Latency ← Upper_Bound_Latency + 1
        Upper_Bound_Latency ← Upper_Bound_Latency × Latency_Parameter
Until Success is true
```

Effect: Example 1



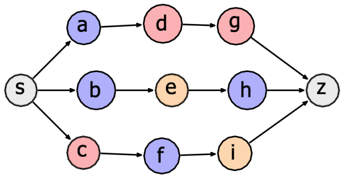
Resource	Symb	Latency	# Units
ALU	🟦	1 CC	1
MUL	🟥	1 CC	1
SRC/SINK	⚪	0 CC	-



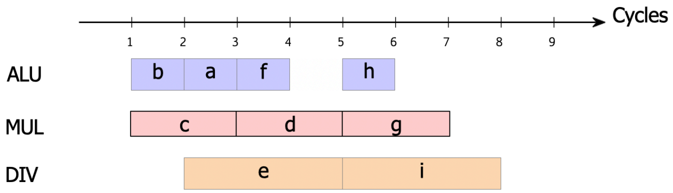
Result

The new changes still succeed to identify **d** as the higher priority task with respect to **a**. This choice allows the MUL operation f to be scheduled in parallel ("hidden"), resulting in an optimal latency of 6.

Effect: Example 2



Resource	Symb	Latency	# Units
ALU	🟦	1 CC	1
MUL	🟥	2 CC	1
DIV	🟧	3 CC	1
SRC/SINK	⬜	0 CC	-



Result

The new changes succeed also to identify **b** as the higher priority task with respect to a. This choice allows the DIV operations to be executed as soon as possible, resulting in an optimal latency of 7.

New Features: Silvia's idea - Motivation: The false tie problem

- **Context:** In the original ML-RCS scheduler, the priority function $F(u) = S(u) \times (C(u) + \epsilon)$ defines the main scheduling force.

- **Problem:** When two nodes have similar $F(u)$, the scheduler breaks ties arbitrarily (e.g., by node ID).
  This leads to unstable or suboptimal schedules, since some nodes are structurally harder to fit than others.

- **Goal:** Introduce a tie-breaking which makes more consistent scheduling decisions, thanks to structural awareness and path rigidity awareness.

New Features: Silvia's idea - Motivation for a Rigidity Metric

- The standard ML-RCS force function accounts for temporal urgency and resource congestion, but it ignores the structural rigidity of the DFG. $F(u) = S(u) \times (C(u) + \epsilon)$ defines the main scheduling force.
- In fact, Two nodes with similar $F(u)$ values can differ significantly:
  - One may lead to a long, sequential downstream chain that is difficult to place later.
  - Another may lead to short or parallel branches that are easier to accommodate.
- To capture this structural aspect, a rigidity metric called ***stiffness*** is introduced, quantifying how rigid or inflexible the paths following a node are, based on the squared latency of its successors.

New Features: Silvia's idea - Mathematical Definition of Stiffness

- $\Pi(u)$ = all the paths starting from a child of u and ending at sink, each containing operations v with latency $\mathrm{latency}(v)$
- $P(u) = \max_{p \in \Pi(u)} \sum_{v \in p} [\mathrm{latency}(v)]^2$

- In this way, the squared term penalizes long-latency operations more strongly, while the maximum selects the most rigid successor path.

- Interpretation: if $P(u)$ is higher, it means that the path is more rigid and less flexible, so it should be scheduled earlier

- Normalization: $P_{norm}(u)$ is obtained by normalizing $P(u)$ to have a value between 0 and 1, and then performing *(1 - normalized version)* in order to get higher priority (lower value) for higher stiffness

New Features: Silvia's idea - Implementation

- calculate_stiffness_term(available_ops)

```
For each node in available_ops
    If node has no children, stiffness ← 0
    Else find max stiffness across children.
Return  1 -  normalized(stiffness) // List
```

- get_stiffness(node)

```
If already computed for this node, return it
Else
    Compute latency squared for this node
    If node has no children, return it
    Else recursively find max across its children
```

New Features: Silvia's idea - Integration into the Force Function

- Rather than using stiffness only as a tie-breaker, it is directly embedded into the main force: $F(u) = S(u) \times (C(u) + \epsilon) \times P(u)$
    - $S(u)$: temporal urgency (slack)
    - $C(u)$: resource congestion
    - $P(u)$: normalized stiffness term
- **Effect:** Nodes with more rigid successors slightly decrease their total force, causing them to be scheduled earlier.

New Features: Silvia's idea - Descendant-Based Tie-Breaker (Priority 3)

- When two nodes still have still equal $F(u)$ and thus the tie-breaking is arbitrary, we apply an additional criterion based on unlock potential:
  $P_2 = -N,$
  Where $N = N_{desc}(u) = |\{v \in V \,|\, u \text{ is an anchestor of v}\}|$

  In other words, it is the number of immediate children.
- **Effect:** Having more children gives a higher potential to unlock parallelism. The negative sign ensures that the scheduler, which sorts ascending, assigns higher priority to nodes with more descendants.

New Features: Silvia's idea - Final sorting hierarchy

- **Level 1:** $F(u)$ - Original force enhanced with rigidity awareness. It is minimized.
- **Level 2:** P_2 - Prefer nodes that unlock more successors.
- **Level 3:** *Node ID* - Stable ordering in case of full tie.

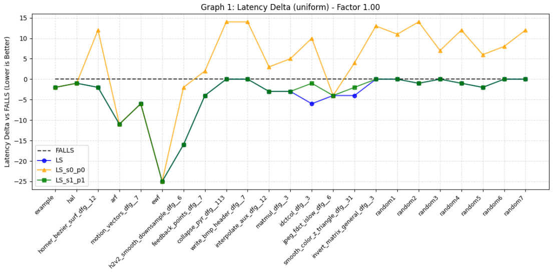
- **Effect:** This hierarchy resolves ties consistently while incorporating structural information and parallelism potential.

New Features: Silvia’s idea - Outcome

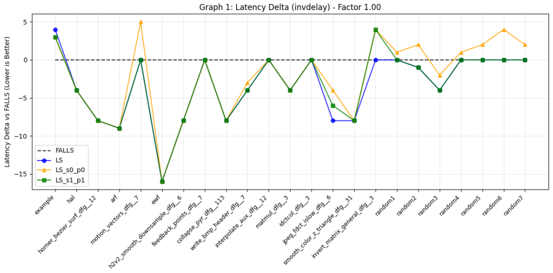
- Eliminates random tie behavior.
- Captures rigidity of successor paths via stiffness.
- Promotes parallelism using descendant-based tie-breaking.
- Keeps complexity low through memoization and local computations.

Results: Latency improvement vs DFGs (LS-std, LS-enhanced, FALLS, FDS)

Uniform



InvDelay



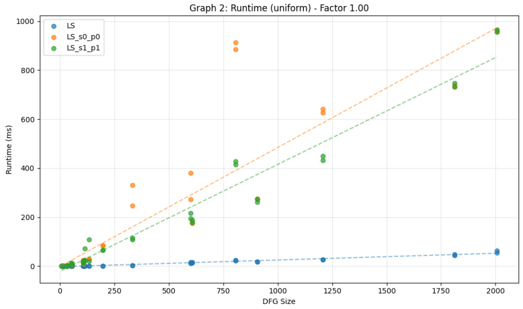
Results Analysis: Latency Improvement - Benchmark comparison

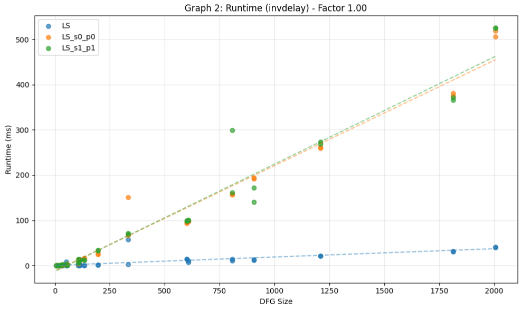
- **Standard LS dominance:** the results clearly show that the standard LS algorithm, with Slack-based priority, achieves better results in almost any DFG in the benchmark.
- **Silvia's Feature Success:** despite achieving sometimes worse latency than standard LS, the new feature significantly improves the Midterm solution, achieving equal or better latency in all benchmark's DFGs.
- **Superiority over FALLS:** Both Standard LS and the Final's version consistently outperform the FALLS algorithm: for these constrained benchmarks, greedy heuristics are more effective than complex methods like FALLS.

Results: Runtime vs DFGs size (LS-std, LS-enhanced, LS-enhanced-feat, FDS)

Uniform

InvDelay



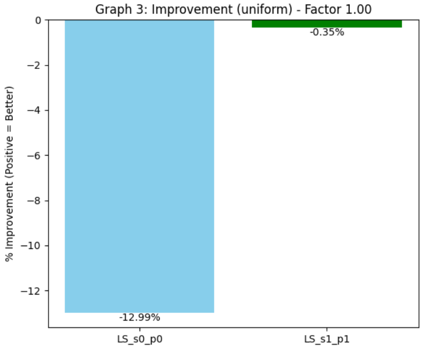


Results Analysis: Runtime Analysis & Scalability

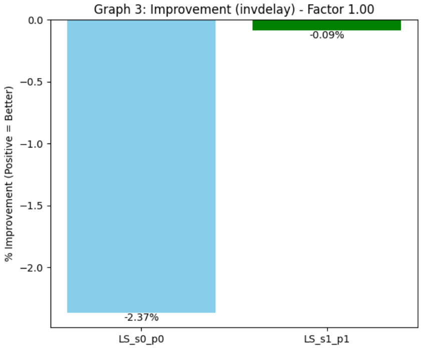
- **Linear Scaling:** All three variants demonstrate linear runtime behavior relative to the number of operations.
- **Minimal Overhead:** The curves for LS_s1_p1 (Final) and LS_s0_p0 (Midterm are nearly indistinguishable, while the one for LS is slightly lower.
- **Observation:** The implementation of Silvia's Stiffness metric and the Force calculations add negligible computational cost.

Results: % Improvement of LS-enhanced and LS-enhanced-feat wrt LS-std

Uniform



InvDelay



Results Analysis: LS-enhanced-feat and LS-enhanced wrt LS

- **The Noise of Force-Directed Scheduling (Midterm):** In the *Uniform* benchmark, using Forces without structural awareness introduced significant decision noise. The scheduler broke ties arbitrarily, leading to a massive 12.99% performance drop compared to simple Slack Priority.

- **The Stabilizing Role of Stiffness (Final):** Adding the Stiffness metric successfully filtered out this noise. By identifying the true structural bottlenecks, it forced the scheduler to make smarter decisions, recovering almost all the lost performance (from -12.99% to -0.35%).

- **Conclusion:** While Standard LS remains the most robust heuristic, Silvia's Stiffness metric is essential for making Force-Directed approaches viable. It transforms an unstable algorithm (Midterm) into a robust one that is comparable to standard LS.

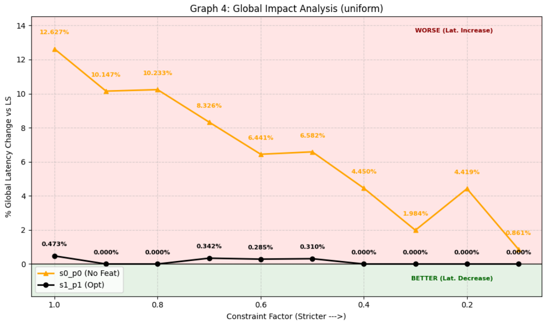
Results Analysis: LS-enhanced-feat and LS-enhanced wrt LS

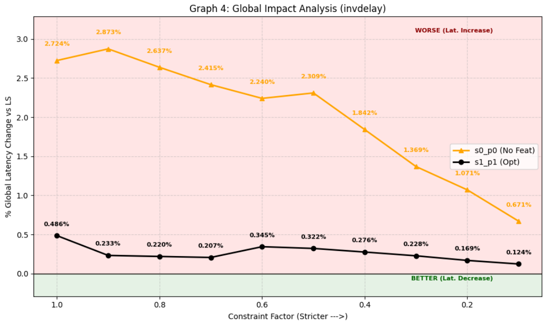
- Even though the results achieved adding the Stiffness metric are far superior to the ones without it, the results are still slightly worse than the ones obtained with the standard LS.
- Being the resource constraints very loose, an interesting analysis is to try to make these constraints more tight, to see whether in this case the enhanced LS can outperform the standard LS.
- The analysis is performed by scaling the available resources from 1.0 (standard case) to 0.10, with a step of 1.0.

Results: % Improvement of LS-enhanced wrt LS scaling the resource constraints

Uniform

InvDelay





Results Analyis: LS-enhanced wrt LS scaling the resource constraints

- **Uniform Benchmark (Final version)**
  - **Observation:** As resources decrease, the performance gap often vanishes completely.
    - Loose Constraints (1.0): Small penalty (- 0.47%).
    - High Stress (0.4 - 0.1): The gap hits 0%: LS final matches LS standard perfectly.
  - **Analysis:** Under extreme pressure, the new Stifness metric aligns with the optimal greedy choice, eliminating its own decision noise as options narrow.

Results Analyis: LS-enhanced wrt LS scaling the resource constraints

- **InvDelay Benchmark (Final version)**
  - **Observation:** The performance gap shrinks consistently as the problem becomes harder.
    - Loose Constraints (1.0): Gap is - 0.49%
    - High Stress (0.4 - 0.1): Gap narrows to - 0.12%
  - **Analysis:** Tighter constraints reduce the margin for error. Instead of degrading, the final LS version adapts, proving that its logic holds becomes more reliable in corner conditions.

Results Analyis: LS-enhanced wrt LS scaling the resource constraints

- **Midterm version**
  - **Observation:** The performance gap consistently shrinks in both Uniform and InvDelay benchmarks.
    - Loose Constraints (1.0): Gap is - 12.67% (Uniform) and - 2.74% (invDelay)
    - High Stress (0.4 - 0.1): Gap narrows to - 0.86% (Uniform) and - 0.67% (invDelay)
  - **Analysis:** Tighter constraints reduce the margin for error also for the version without the Stiffness metric. Even though results get better as the pressure increases, this version still gives worse performance with respect to the Final one.

Conclusion

- **The Noise of Force-Directed Scheduling (Midterm):** In the Uniform benchmark, using FDS without structural awareness introduced significant decision noise. The scheduler broke ties arbitrarily, leading to a massive 12.99% performance drop compared to simple Slack Priority.

- **The Stabilizing Role of Stiffness (Final):** Adding the Stiffness metric successfully filtered out this noise. By identifying true bottlenecks, it forced the scheduler to make smarter decisions, recovering almost all the performance loss (from - 12.99% to - 0.35%).

- **Final Conclusion:** While the standard List Scheduling remains a robust heuristic, Silvia's Stiffness metric is essential for making Force-Directed approaches viable. It transforms an unstable algorithm into a competitive one.