SCAR-DA: A Stable, Congestion-Aware Routing Algorithm with Deadlock-Avoidance

An Algorithm Description

1 Objective

The **SCAR-DA (Stable, Congestion-Aware Routing with Deadlock-Avoidance)** algorithm is a robust pathfinding algorithm designed for networks where path quality depends on dynamic congestion, and where network safety (deadlock-freedom) is a hard constraint. This is particularly relevant for Network-on-Chip (NoC) architectures.

It addresses three primary challenges:

- **Congestion:** Naive shortest-path routing (e.g., standard Dijkstra) creates bottlenecks by overloading the same shortest-hop paths.
- **Instability (Oscillation):** Simple load-aware routing often suffers from "route flapping," where all traffic rushes to a newly free path, congesting it, and then rushing away again.
- **Deadlock:** Adaptive or load-aware routing paths can easily violate the network's deadlock-avoidance (DA) turn model, leading to catastrophic network failure.

SCAR-DA integrates three concepts into a modified Dijkstra's shortest-path search:

1. A multi-factor cost function that considers both link and node congestion.
2. An Exponentially Weighted Moving Average (EWMA) for stable load metrics.
3. A constraint check to prune any path that violates the network's DA ruleset.

2 Algorithm Components

2.1 Input Parameters

$G = (V, E)$ The network graph, where V are routers and E are links.

s, t The source and target nodes for the path.

R_{DA} The Deadlock-Avoidance Ruleset (e.g., an XY, West-First, or Odd-Even turn model).

α, β Weighting coefficients that define the "penalty" for link and node congestion, respectively.

γ_L, γ_N Smoothing factors for the EWMA of link and node loads. A smaller γ results in more smoothing (slower reaction).

2.2 Network State: EWMA Load Smoothing

To prevent route flapping, the algorithm does not use instantaneous load metrics. Instead, it relies on a smoothed load value for each link e and node v, which is updated periodically by a separate network monitoring process (see Algorithm 2).

Let $L_{inst}(t)$ be the instantaneous load (e.g., buffer occupancy, bandwidth utilization) at time t. The EWMA load, $L_{ewma}(t)$, is calculated as:

$$L_{ewma}(t) = (\gamma \times L_{inst}(t)) + ((1 - \gamma) \times L_{ewma}(t - 1))$$

Where γ is the smoothing factor (γ_L for links, γ_N for nodes). The pathfinding algorithm (Algorithm 1) reads these stored L_{ewma} values.

2.3 Multi-Factor Dynamic Cost Function

The core of the algorithm is its cost function. The "cost" of traversing from node u to a neighbor v is a combination of static latency and dynamic, smoothed congestion costs for both the link (u, v) and the destination node v.

2.3.1 Node Cost

The cost associated with the destination router v:

$$C_{node}(v) = Lat_{node}(v) \times (1 + \beta \times L_{node}^{ewma}(v))$$

Where $Lat_{node}(v)$ is the static processing latency of router v and $L_{node}^{ewma}(v)$ is its smoothed buffer load (a value from 0.0 to 1.0).

2.3.2 Link Cost

The cost associated with traversing the link (u, v):

$$C_{link}(u, v) = Lat_{link}(u, v) \times (1 + \alpha \times L_{link}^{ewma}(u, v))$$

Where $Lat_{link}(u, v)$ is the static latency of the link (e.g., hop count, wire delay) and $L_{link}^{ewma}(u, v)$ is its smoothed bandwidth utilization.

2.3.3 Total Dynamic Cost

The total cost added to the path when moving from u to v is the sum of these two components:

$$\mathbf{C_{dyn}(u, v) = C_{link}(u, v) + C_{node}(v)}$$

2.4 Deadlock-Avoidance Constraint

The algorithm enforces safety by pruning any path segment that violates the network's turn model, R_{DA}. This is managed by a helper function:

`IsLegalTurn(prev_node, current_node, next_node,` R_{DA}`)`

This function returns `false` if the turn from $prev \rightarrow curr \rightarrow next$ is forbidden (e.g., a Y-to-X turn in an XY routing scheme), and `true` otherwise. For the first hop from the source, *prev_node* is null, and the function always returns `true`.

3 SCAR-DA Pathfinding Algorithm

The core algorithm is a modified Dijkstra's, shown in Algorithm 1. It finds the lowest-cost path that is also deadlock-free.

Algorithm 1 SCAR-DA Pathfinding Algorithm

Require: Graph $G = (V, E)$, source s, target t, ruleset R_{DA}, α, β

Let $dist[v] \leftarrow \infty$ for all $v \in V$
Let $prev[v] \leftarrow$ null for all $v \in V$
$dist[s] \leftarrow 0$
$pq \leftarrow \text{PriorityQueue}(V)$ ▷ Min-priority queue, stores $(v, dist[v])$
$pq.\text{add}(s, 0)$

while pq is not empty **do**
$u \leftarrow pq.\text{pop\_min}()$
if $u = t$ **then**
return $\textsc{BuildPath}(prev, t)$ ▷ Found the best path
end if
$prev\_node \leftarrow prev[u]$
for each neighbor v of u **do** ▷ — 1. Deadlock-Avoidance Constraint Check —
if $\textsc{IsLegalTurn}(prev\_node, u, v, R_{DA}) =$ **false then**
continue ▷ Prune this path; it's an illegal turn
end if
▷ — 2. Multi-Factor Cost Calculation —
$L_{link} \leftarrow G.\text{get\_link\_load}(u, v)$ ▷ Returns L_{link}^{ewma}
$L_{node} \leftarrow G.\text{get\_node\_load}(v)$ ▷ Returns L_{node}^{ewma}
$Lat_{link} \leftarrow G.\text{get\_link\_latency}(u, v)$
$Lat_{node} \leftarrow G.\text{get\_node\_latency}(v)$
$C_{link} \leftarrow Lat_{link} \times (1 + \alpha \times L_{link})$
$C_{node} \leftarrow Lat_{node} \times (1 + \beta \times L_{node})$
$dynamic\_cost \leftarrow C_{link} + C_{node}$
▷ — 3. Standard Dijkstra Relaxation Step —
$new\_dist \leftarrow dist[u] + dynamic\_cost$
if $new\_dist < dist[v]$ **then**
$dist[v] \leftarrow new\_dist$
$prev[v] \leftarrow u$
$pq.\text{add\_or\_update}(v, new\_dist)$
end if
end for
end while

return null ▷ No deadlock-free path found

4 Network State Update Process

This process (Algorithm 2) is not part of the pathfinding call. It is a separate, concurrent process that runs periodically (e.g., every N clock cycles) to update the smoothed load values that Algorithm 1 relies on.

Algorithm 2 Periodic EWMA State Update Process

Require: Graph G, γ_L, γ_N

procedure UpdateNetworkEWMA ▷ Run this procedure at a fixed interval
    for each link $e = (u, v)$ in $G.E$ **do**
        $L_{inst} \leftarrow G$.get_instantaneous_link_load(e)
        $L_{old} \leftarrow e.L_{ewma}$
        $L_{new} \leftarrow (\gamma_L \times L_{inst}) + (1 - \gamma_L) \times L_{old}$
        $e.L_{ewma} \leftarrow L_{new}$ ▷ Store the new smoothed value
    end for

    for each node v in $G.V$ **do**
        $L_{inst} \leftarrow G$.get_instantaneous_node_load(v)
        $L_{old} \leftarrow v.L_{ewma}$
        $L_{new} \leftarrow (\gamma_N \times L_{inst}) + (1 - \gamma_N) \times L_{old}$
        $v.L_{ewma} \leftarrow L_{new}$ ▷ Store the new smoothed value
    end for
end procedure