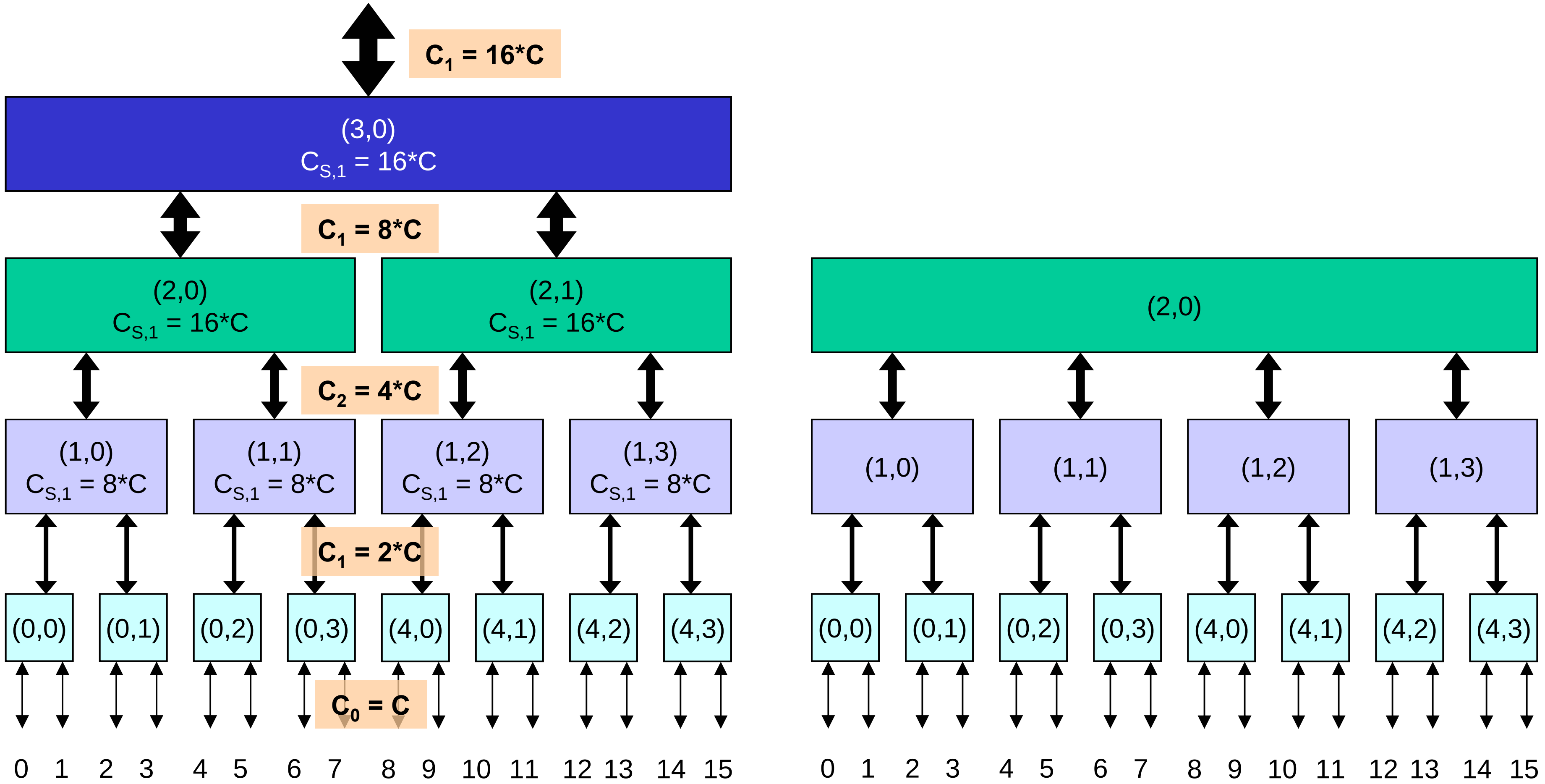
Multistage Interconnection Networks for Data Centers

Bidirectional Fat Tree Construction and Routing for IEEE 802.1au

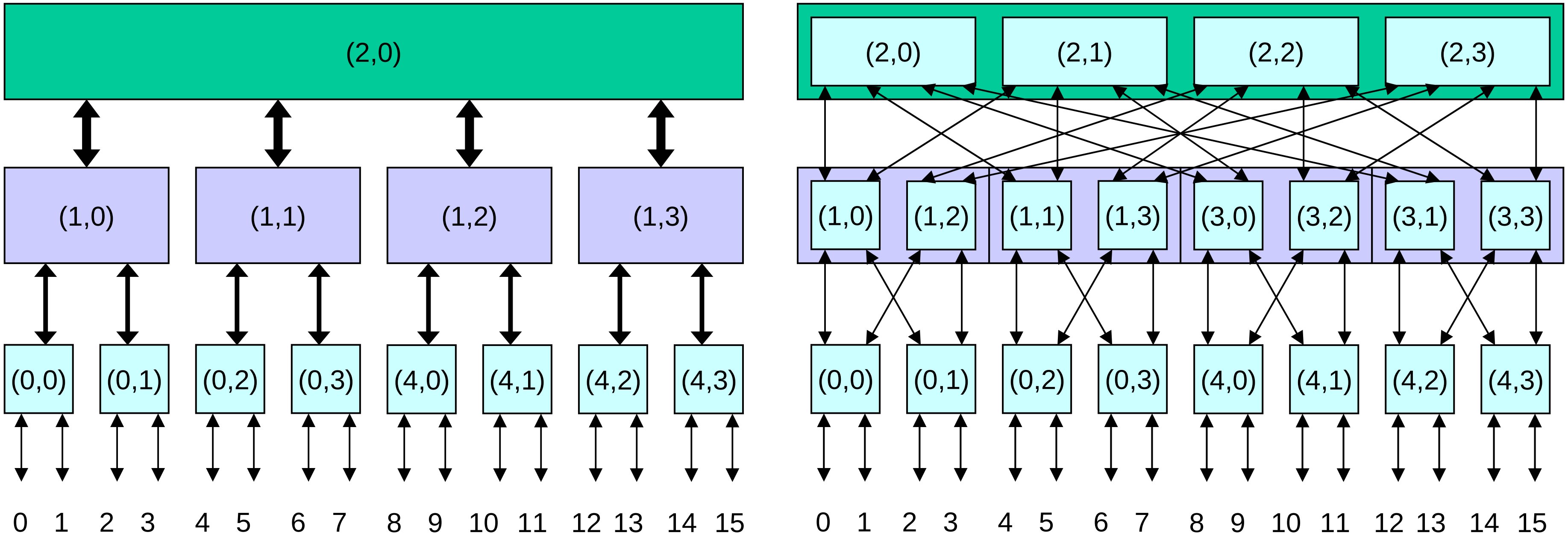
Cyriel Minkenberg & Mitch Gusat
IBM Research GmbH, Zurich

The origin of the term "fat tree"



- A fat tree's main characteristic: Constant bisectional bandwidth
- Achieved by increasingly "fatter" links in subsequent levels
- Top level can be eliminated if further expansion not desired

Transmogrification

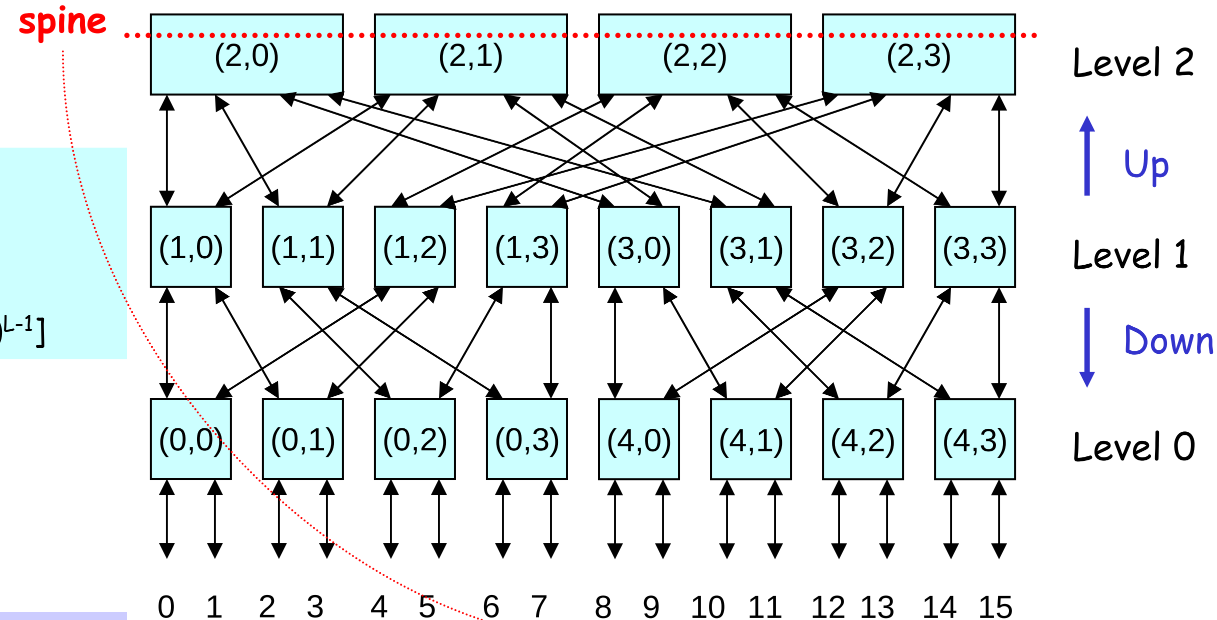


- Scalability of switching node is an issue
  - Aggregate capacity doubles with each level
- Construct fat tree from fixed-capacity switches
  - As shown on the right
  - Requires specific interconnection and routing rules

Fat tree ⇔ Beneš network

Switches are labeled (stageID, switchID):

- stageID ∈ [0, S-1]
- switchID ∈ [0, $(N/2)^{L-1}$]



Fat tree: Folded representation

Conventions

M = no. of end nodes = $N^*(N/2)^{L-1}$

N = no. of bidir ports per switch

L = no. of levels (folded)

S = no. of stages = 2L-1 (unfolded)

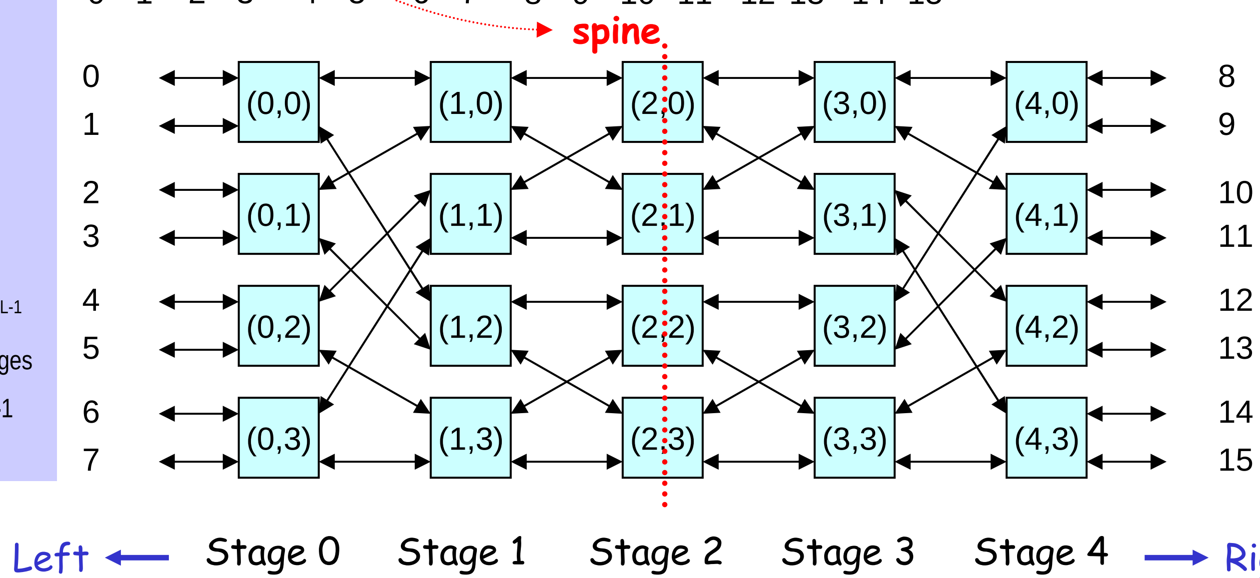
Number of switches per stage = $(N/2)^{L-1}$

Total number of switches = (2L-1) $*(N/2)^{L-1}$

Nodes are connected at left and right edges

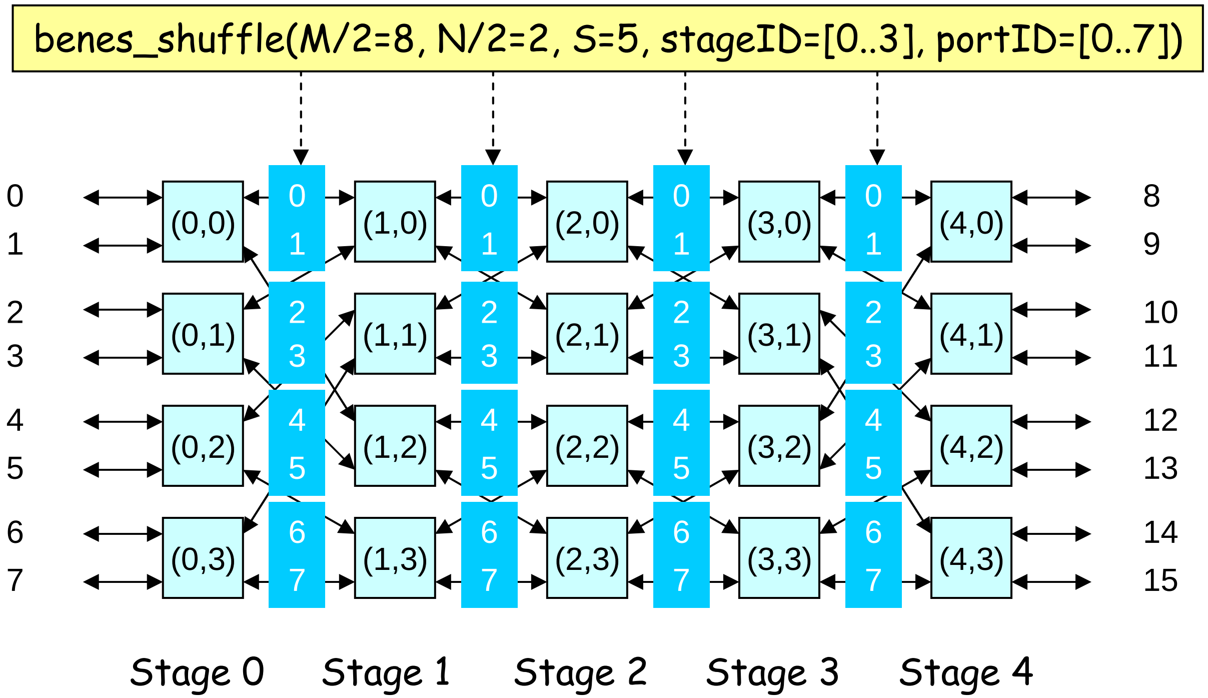
Left nodes are numbered 0 through M/2-1

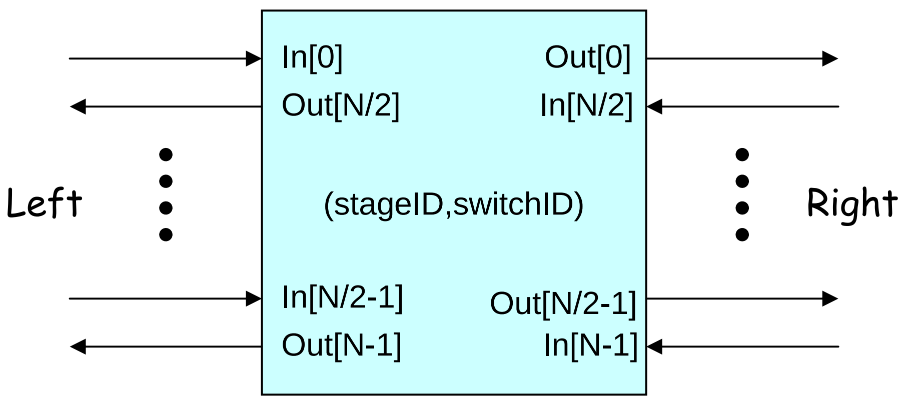
Right nodes are numbered M/2 to M-1



Unfolded to Benes

Shuffle & port numbering





Apply appropriate (see next foil) shuffle pattern between every pair of stages

Note: Shuffle and routing depend on the switch and port labels.

Shuffle parameters

1. Half the number of end nodes: M/2
2. Half the switch radix: N/2
3. Stage ID, ranging from 0 to S-2
4. Port ID, ranging from 0 to N/2-1, numbered top to bottom across all switches in a stage

Do the shuffle: Fat tree construction

```
int perfect_shuffle(int X, int Y, int i) {
  return (Y*i+i/(X/Y)) % X;
}

int inverse_perfect_shuffle(int X, int Y, int i) {
  return (i/Y+i*(X/Y)) % X;
}

int benes_shuffle(int X, int Y, int S, int s, int p) {
  int i, k, K, r;

  if (s < (S-1)/2)
    K = X/(int(pow(Y, s)));
  else
    K = X/(int(pow(Y, (S-1)-s-1)));

  i = p/K;
  k = p%K;
  if (s < (S-1)/2)
    r = i*K+inverse_perfect_shuffle(K, Y, k);
  else
    r = i*K+perfect_shuffle(K, Y, k);

  return r;
}
```

Fat vs. spanning trees

- Spanning tree protocol limits bisection bandwidth to capacity of root bridge: N*C
  - C = link capacity
- Main advantage of fat tree: Full bisection bandwidth: M*C
  - Fundamental mismatch!

Deterministic routing algorithm

- Shortest path
  - Up: from source to first common ancestor (multiple paths possible)
  - Down: to destination (single path)
  - Guaranteed loop-free

- Destination-based
  - Routing decision depends only on current position (switch) and destination, not on source

- Fat tree = constant bisectional bandwidth
  - Multiple paths up to spine
  - All nodes are reachable from any middle stage switch (spine)
  - Number of alternative (upwards only!) paths = $(N/2)^{L-1}$
  - Static path assignment
    - Depending on the destination distribution, static routing may lead to oversubscription of fabric-internal ports even under admissible, non-congestive traffic
    - Load balancing (LB) or adaptive routing (AR) may alleviate congestion, but that's a different kettle of fish (spatial instead of temporal response)
    - See "Deterministic versus adaptive routing in fat-trees" when available at CAC '07
- For simulation purposes, we propose a static, destination-based, shortest-path routing algorithm

Routing decision code

```
int routing_lookup(int s, int i, int m) {
    // s = stageID, i = switchID, m = destination node ID
  int l, x, range, start, dir, d;
  d = pow(N/2, L-1);                        // number of switches per stage
  l = s > S/2 ? S-1-s : s;                  // level ID
  range = pow(N/2, l+1);                    // number of nodes reachable
  x = (M/N) / pow(N/2, l);
  start = (m < M/2 ? 0 : M/2) + (i % x) * range;
  if (((s ≤ S/2 && m < M/2) || (S ≥ S/2 && m ≥ M/2))
    && (start ≤ m && m < start + range)) { // destination reachable on down path
    dir = m < M/2 ? N/2 : 0;                // direction (left/right)
    return dir + ((m – start) / pow(N/2, l)) % (N/2) + (m % d)*(N/(2*d));
  } else { // dst not reachable on down path, route in opposite direction
    dir = s < S/2 ? 0 : N/2;                // direction (left/right)
    return dir + ((m – (m < M/2 ? 0 : M/2)) / pow(N/2, l)
                 + (last_hop(s,i,m) ? 0 : i)) % (N/2);
  }
}

bool last_hop(int s, int i, int m) {
    //s = stageID, i = switchID, m = destination node ID
  if (m < M/2 && s == 0)                    // dst on the left and in leftmost stage
    return m/(N/2) == i;                    // matching switch ID?
  else if (m ≥ M/2 && s == S-1)             // dst on the right and in rightmost stage
    return (m-M/2)/(N/2) == i;              // matching switch ID?
  else
    return false;
}
```