Algorithm 6.4.2 : Group migration

```
P = P_in
loop
   /* Initialize */
   prev_P = P
   prev_cost = Objfct(P)
   bestpart_cost = ∞
   for each o_i loop
      o_i.moved = false
   end loop

   /* Create a sequence of n moves */
   for i in 1 to n loop
      bestmove_cost = ∞
      for each o_i, not o_i.moved loop
         cost = Objfct(Move(P, o_i))
         if cost < bestmove_cost then
            bestmove_cost = cost
            bestmove_obj = o_i
         end if
      end loop
      P = Move(P, bestmove_obj)
      bestmove_obj.moved = true
      /* Save the best partition during the sequence */
      if bestmove_cost < bestpart_cost then
         bestpart_P = P
         bestpart_cost = bestmove_cost
      end if
   end loop

   /* Update P if a better cost was found, else exit */
   if bestpart_cost < prev_cost then
      P = bestpart_P
   else return prev_P
   end if
end loop
```

Algorithm 6.4.4 : Genetic evolution

```
/* Create first generation with gen_size random partitions */
G = φ
for i in 1 to gen_size loop
   G = G ∪ CreateRandomPart(O)
end loop
P_best = BestPart(G)

/* Evolve generation */
while not Terminate loop
   G = Select(G, num_sel) ∪ Cross(G, num_cross)
   Mutate(G, num_mutate)
   if Objfct(BestPart(G)) < Objfct(P_best) then
      P_best = BestPart(G)
   end if
end loop

/* Return best partition in final generation */
return P_best
```

Yorktown Silicon Compiler

Hierarchical Clustering for Hardware Objects

$$Closeness(p_i, p_j) = \left(\frac{Conn_{i,j}}{MaxConn(P)}\right)^{k_2} \times \left(\frac{size\_max}{Min(size_i, size_j)}\right)^{k_3} \times \left(\frac{size\_max}{size_i + size_j}\right) \quad (6.5)$$

$Conn_{i,j}$	$= k_1 \times inputs_{i,j} + wires_{i,j}$,
$inputs_{i,j}$	equals the number of common inputs shared by groups p_i and p_j,
$wires_{i,j}$	equals the number of output to input and input to output connections between p_i and p_j,
$MaxConn(P)$	equals the maximum $Conn$ over all pairs of groups p_x, p_y in partition P,
$size_i$	equals the estimated size of group p_i in transistors,
$size\_max$	equals the maximum group size allowed, and
k_1, k_2, k_3	are constants.

YSC Partitioning Example

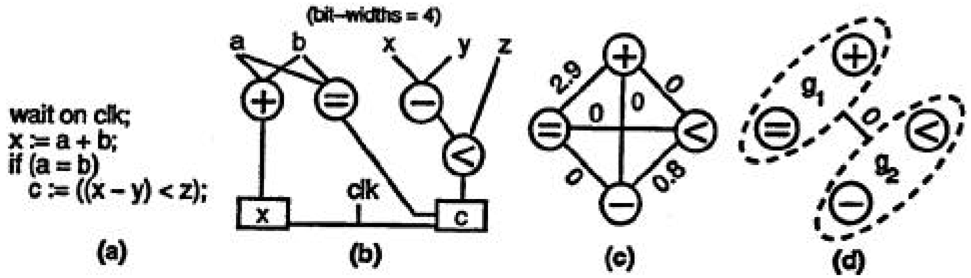

Figure 6.6: YSC partitioning example: (a) input, (b) operations, (c) operation closeness values, (d) clusters formed with 0.5 threshold.

$$Closeness(+,=) = \frac{8+0}{8} \times \frac{300}{120} \times \frac{300}{120+140} = 2.9$$

$$Closeness(-,<) = \frac{0+4}{8} \times \frac{300}{160} \times \frac{300}{160+180} = 0.8$$

All other operation pairs have a closeness value of 0. The closeness values between all operations are shown in Figure 6.6(c).

Figure 6.6(d) shows the results of hierarchical clustering with a closeness threshold of 0.5. The + and = operations form one cluster, and the < and − operations form a second cluster.

YSC Partitioning with similarities

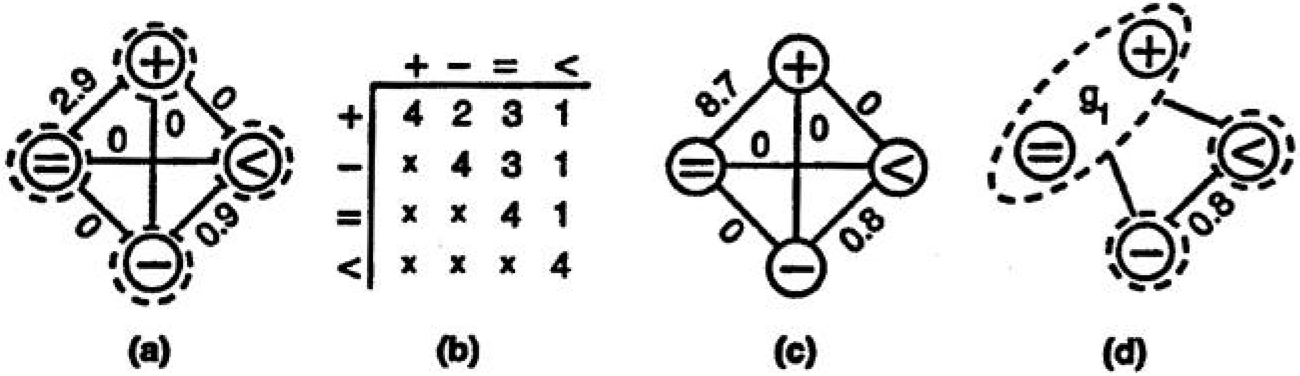

Figure 6.7: YSC partitioning with similarities: (a) clusters formed with 3.0 closeness threshold, (b) operation similarity table, (c) closeness values with similarities, (d) clusters formed.

BUD Closeness Function

$$
\begin{aligned}
Closeness(o_i, o_j) &= \left(\frac{FU\_cost(o_i) + FU\_cost(o_j) - FU\_cost(o_i, o_j)}{FU\_cost(o_i, o_j)}\right) \\
&+ \left(\frac{Conn(o_i, o_j)}{Total\_conn(o_i, o_j)}\right) \\
&- N \times (Par(o_i, o_j)) \qquad (6.7)
\end{aligned}
$$

o_i	is the i'th operation,
$FU\_cost(o_i, o_j)$	is the cost, based on delay and area, of the minimal number of functional units needed to perform the given operations,
$Conn(o_i, o_j)$	is the number of wires shared by o_i and o_j,
$Total\_conn(o_i, o_j)$	is the total number of wires to either o_i or o_j, and
$Par(o_i, o_j)$	is 1 if o_i and o_j can be executed in parallel, 0 otherwise.

Greedy Partitioning

Algorithm 6.6.1 Greedy move

```
repeat
   P_orig = P
   for i in 1 to n loop
      if Objfct(Move(P, o_i) < Objfct(P) then
         P = Move(P, o_i)
      end if
   end loop
until P = P_orig
```

VULCAN II Partitioning

Algorithm 6.6.2 Vulcan II algorithm

```
P = {O, φ} /* all-hardware initial partition */
repeat
   P_orig = P
   for each o_i ∈ hardware loop
      AttemptMove(P, o_i)
   end loop
until P = P_orig

procedure AttemptMove(P, o_i)
   if SatisfiesPerformance(Move(P, o_i))
         and Objfct(Move(P, o_i)) < Objfct(P) then
      P = Move(P, o_i)
      for each o_j ∈ Successors(o_i) loop
         AttemptMove(P, o_j)
      end loop
   end if
```

Performance Weighted Hill Climbing

$$Objfct(P) = k_{perf} \times \sum_{i=1}^{m} Violation(C_i) + k_{area} \times Size(hardware) \quad (6.13)$$

$Violation(C_i) = Performance(G_i) - V_i$ if the difference is greater than 0; otherwise, $Violation(C_i) = 0$. Also, $k_{perf} >> k_{area}$, but k_{perf} should *not* be infinity, since then the algorithm could not distinguish a partition that almost meets constraints from one that greatly violates those constraints.

Binary Constraint Search Copartitioning

Algorithm 6.6.3 Binary constraint-search (BCS) hw/sw partitioning

$low = 0, high = AllHwSize$
while $low < high$ **loop**
  $mid = \frac{low+high+1}{2}$
  $P' = \text{PartAlg}(P, C, mid, \text{Cost}())$
  if $\text{Cost}(P', C, mid) = 0$ **then**
    $high = mid - 1$
    $P\_zero = P'$
  else
    $low = mid$
  end if
end loop
return P_zero