Algorithm 6.4.2 : Group migration

\[ P = P_{\text{in}} \]
loop

/** Initialize */
prev\_P = P
prev\_cost = Objfct(P)
bestpart\_cost = \infty
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 = \infty
  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 */
\[ G = \emptyset \]
for \( i \) in 1 to \( \text{gen\_size} \) loop
  \[ G = G \cup \text{CreateRandomPart}(O) \]
end loop
\[ P\_best = \text{BestPart}(G) \]

/* Evolve generation */
while not Terminate loop
  \[ G = \text{Select}(G, \text{num\_sel}) \cup \text{Cross}(G, \text{num\_cross}) \]
  \[ \text{Mutate}(G, \text{num\_mutate}) \]
  if \( \text{Objfct}(\text{BestPart}(G)) < \text{Objfct}(P\_best) \) then
    \[ P\_best = \text{BestPart}(G) \]
  end if
end loop

/* Return best partition in final generation */
return \( P\_best \)
Yorktown Silicon Compiler

Hierarchical Clustering for Hardware Objects

\[ Closeness(p_i, p_j) = \left( \frac{Conn_{i,j}}{\text{MaxConn}(P)} \right)^{k_2} \times \left( \frac{\text{size}_\text{max}}{\text{Min}(\text{size}_i, \text{size}_j)} \right)^{k_3} \times \left( \frac{\text{size}_\text{max}}{\text{size}_i + \text{size}_j} \right) \quad (6.5) \]

- \( Conn_{i,j} = k_1 \times \text{inputs}_{i,j} + \text{wires}_{i,j} \),
- \( \text{inputs}_{i,j} \) equals the number of common inputs shared by groups \( p_i \) and \( p_j \),
- \( \text{wires}_{i,j} \) equals the number of output to input and input to output connections between \( p_i \) and \( p_j \),
- \( \text{MaxConn}(P) \) equals the maximum \( Conn \) over all pairs of groups \( p_x, p_y \) in partition \( P \),
- \( \text{size}_i \) equals the estimated size of group \( p_i \) in transistors,
- \( \text{size}_\text{max} \) 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.

\[
Closeness(\pm,\pm) = \frac{8 + 0}{8} \times \frac{300}{120} \times \frac{300}{120 + 140} = 2.9
\]
\[
Closeness(\pm,\lt) = \frac{0 + 4}{8} \times \frac{300}{160} \times \frac{300}{160 + 180} = 0.8
\]

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

\[ Closeness(o_i, o_j) = \left( \frac{FU_{\text{cost}}(o_i) + FU_{\text{cost}}(o_j) - FU_{\text{cost}}(o_i, o_j)}{FU_{\text{cost}}(o_i, o_j)} \right) + \left( \frac{Conn(o_i, o_j)}{Total_{\text{conn}}(o_i, o_j)} \right) - N \times (Par(o_i, o_j)) \]  

(6.7)

- \( o_i \) is the \( i \)'th operation,
- \( FU_{\text{cost}}(o_i, o_j) \) 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 \( o_j \),
- \( Total_{\text{conn}}(o_i, o_j) \) 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
    $P_{\text{orig}} = P$
    for $i$ in 1 to $n$ loop
        if $\text{Objfct}(\text{Move}(P, o_i)) < \text{Objfct}(P)$ then
            $P = \text{Move}(P, o_i)$
        end if
    end loop
until $P = P_{\text{orig}}$

VULCAN II Partitioning

Algorithm 6.6.2 Vulcan II algorithm

$P = \{O, \emptyset\} /\ast$ all-hardware initial partition $\ast$/
repeat
    $P_{\text{orig}} = P$
    for each $o_i \in$ hardware loop
        AttemptMove($P, o_i$)
    end loop
until $P = P_{\text{orig}}$

procedure AttemptMove($P, o_i$)
    if SatisfiesPerformance($\text{Move}(P, o_i)$)
        and $\text{Objfct}(\text{Move}(P, o_i)) < \text{Objfct}(P)$ then
            $P = \text{Move}(P, o_i)$
        for each $o_j \in$ Successors($o_i$) loop
            AttemptMove($P, o_j$)
        end loop
    end if
end procedure
Performance Weighted Hill Climbing

\[ \text{Object}(F) = k_{\text{perf}} \times \sum_{i=1}^{m} \text{Violation}(C_i) + k_{\text{area}} \times \text{Size(hardware)} \quad (6.13) \]

\[ \text{Violation}(C_i) = \text{Performance}(G_i) - V_i \text{ if the difference is greater than 0; otherwise, } \text{Violation}(C_i) = 0. \] Also, \( k_{\text{perf}} \gg k_{\text{area}} \), but \( k_{\text{perf}} \) 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 Cost\((P', C, mid) = 0\) then
        high = mid - 1
        \(P_{\text{zero}} = P'\)
    else
        low = mid
    end if
end loop
return \(P_{\text{zero}}\)