An Integer Linear Program for Generating Optimal Schedules with Resource Constraints.

vamsi.k.kundeti@intel.com

Introduction

We consider the problem of generating an optimal operation schedule which meets various resource constraints imposed by the hardware (e.g. CMX bound or Cluster count etc.). In this document we formulate an *Integer Linear Program (ILP)* which can generate schedules to maximize the throughput (or minimize the *makespan*).

Preliminaries

Following are some preliminary definitions to aid the problem definition:

- *Operation Precedence Graph* (OPG)*:* is a directed acyclic graph $G = (V, E)$, $V$is the set of *operations* and Ecorresponds to *partial order* (i.e. $v_i < v_j$) between them.
- *Delay:* $d : V \rightarrow N$ is a function which associates a delay to each operation. If an operation $v_i \in V$ starts at time t then its completion time time is $\geq t + d(v_j)$
- *Schedule:* is a sequence of operations ordered according to their *start times*.
- *Resource Utility Function:* $r : V \rightarrow \{1,2,3,...k\}$, is a function which maps every operation to a natural number $\leq k$. If $r(v_i) = q$ means that operation v_i needs q units of a *resource* (e.g. memory). Here k is the upper bound on the available resources.
- *MakeSpan Upper Bound:* integer $m > 0$ is the upper bound on the makespan obtained by a heuristic schedule which meets resource constraints. Note it may not be the optimal schedule. See section on *Feasible Schedule Generator*
- *Processing Units:* we assume that a *processing unit* can execute any operation in V. The integer $n \geq 1$ indicates the maximum number of concurrent operations which can be executed.
- *Packing:* is an arrangement of a set of rectangles in a *region* such that no two rectangles *overlap* (touching along edges is allowed e.g. **Fig-4**).

Commented [1]: Several of the equations below assume that the completion time is exactly equal to t+d(vj); it's probably best to define it that way unless there's some reason for making it a bound.

Commented [2]: Yes, that's true. We could make this an equality.

Commented [3]: Feasible Schedule Generator (FSG) is used to compute this value 'm' (makespan upperbound)

Integer Linear Program

We now define an ILP variables follows:

- $x_{i,t}$ ≡ is 1 if the operation $v_i \in V$ starts at time t otherwise its 0.
- $a_{i,t}$ ≡ is 1 if the operation $v_i \in V$is active in time step t otherwise its 0.
- $u_{i,t}$ ≡is $r(v_i)$ if the operation $v_i \in V$ is active at time step t otherwise its 0. This corresponds to the *resource utilization* of the operation v_i at time step t.

Notice that $1 \leq t \leq m$ since we have an upper bound on the time any operation starts. We now express the linear constraints

Active Operation and Active Resource Utilization

An operation v_i is active (executing) at time step l iff the following constraint holds:

$$a_{i,l} = \sum_{t=max\{0,l-d(v_i)+1\}}^{l} x_{i,t} = 1 \qquad \textbf{(1)}$$

Hence the *resource utilization* of an active operation v_i at time step l is the following:

$$u_{i,l} = r(v_i) \times \left(\sum_{t=max\{0,l-d(v_i)+1\}}^{l} x_{i,t}\right) \qquad \textbf{(2)}$$

Resource Constraint

At any time step in the schedule we don't want the resource utilization of all active operations exceed the resource usage bound k. We can now express the *resource constraint* as a linear inequality as follows :

$$\sum_{v_i \in V} u_{i,t} \leq k \ , \ \forall\, t \in [1,m] \qquad \textbf{(3)}$$

The above inequality ensures that the sum of resources used by the operations during the schedule does not exceed the upper bound on the resources preceding section.

Concurrency Constraint

Since we allow at most n operations to execute concurrently we capture this by the following constraint which says that at any time of the schedule execution there are at most n active processing units.

$$\sum_{v_i \in V} a_{i,t} \leq n \ , \ \forall\, t \in [1,m] \qquad \textbf{(4)}$$

Start Time Constraint

Now within this bounded *makespan* every operation starts *exactly once*, this is captured by the following constraint:

$$\sum_{t=1}^{m} x_{i,t} = 1 \qquad \textbf{(5)}$$

As a corollary the start time of an operation v_i is precisely $\sum_{t=1}^{m} t \times x_{i,t}$

Operation Dependency Constraint

We need to generate schedules which meet *precedence* enforced by the DAG $G = (V, E)$. This means that for an edge $(v_i, v_j) \in E$, the start time of v_j should be at least the start time of $v_i$plus its delay.

$$\left(\sum_{t=1}^{m} t \times x_{i,t}\right) + d(v_i) \leq \sum_{t=1}^{m} t \times x_{j,t} \quad , \forall\, (v_i, v_j) \in E \qquad \textbf{(6)}$$

Commented [4]: It'd be good to update the numbering for these equations. (I wanted to note this equation as one of the ones that's assuming the completion time is == d(vj), but which equation does (4) reference?)

Minimum Makespan Objective

Our goal is to find a schedule with minimum *completion time* (makespan) of all the operations in the *precedence* graph. This is can be expressed by adding a *no-op sink node* $v_{sink} \in V$ and we want to minimize the time t when the sink node starts in the schedule:

$$min \sum_{t=1}^{m} x_{sink,t} \qquad \textbf{(7)}$$

The goal of the ILP is to find a *vector* $[x_{1,1}, x_{1,2} \ldots x_{sink,m}] \in \{0,1\}^{m \times |V|}$ which minimizes the above objective function subject to *resource, concurrency, start time* and *operation dependency* constraints defined above.

Modeling Contiguous Resource Constraints:

When dealing with *memory* resources applications typically require all the memory locations allocated for an operation to be *contiguous* since the operations are *non-preemptive.* Once the operation is scheduled to start its assigned memory resources does not change. The *resource constraint* (see **(3)**) ensures that the total amount of the resources used by *active* operations does not exceed the specified bound. However it does not capture the contiguous resource requirement. Following is a general method on how you can model such requirements into the former linear program:

- $x_{i,t,p}$ ≡ is 1 iff the operation $v_i \in V$ starts at time t at resource base location $p \in [1, k]$. This means that the operation needs locations $[p, p+1, p+2 \ \ldots p + r(v_i) - 1]$.
- $b_{i,t,p}$ ≡ is 1 iff the operation uses a resource location $p \in [1, k]$

Contiguous Resource Constraint:

Given an operation $v_i \in V$ executing at time t with a resource requirement of $r(v_i)$ we define $b_{i,t,p}$ as follows:

$$b_{i,t,p} = \sum_{q=\min\{0,p-r(v_i)\}}^{p} x_{i,t,q} \qquad \textbf{(3.1)}$$

Notice that $b_{i,t,p}$ is 1 for resource locations $[p, p+1, p+2 \ldots p+r(v_i)-1]$ (see **Fig-4**). We now need to encode the requirement what no other operations can have resources $[p, p+1 \ldots p+r(v_i)-1]$, this can easily be done using the indicator variable $b_{i,t,p}$ as follows:

$$\sum_{v_i \in V} b_{i,t,p} \leq 1, \forall t \in [1, m] \wedge \forall p \in [1, k] \qquad \textbf{(3.2)}$$

The above constraint states that at all times during schedule execution at most ***one*** operation can use the resource location p. **Fig-5** illustrates that any schedule where there is an overlap in the resource locations leads to violation of constraint **(3.2)**. Thus the constraints **(3.1)** and **(3.2)** model the requirement of contiguous resource assignment for resources like memory etc. Notice that if you decided to enforce contiguous memory constraint then we need to replace all $x_{i,t}$ with $x_{i,t,p}$ and add the new constraints **(3.1)** and **(3.2)** for the former linear program.

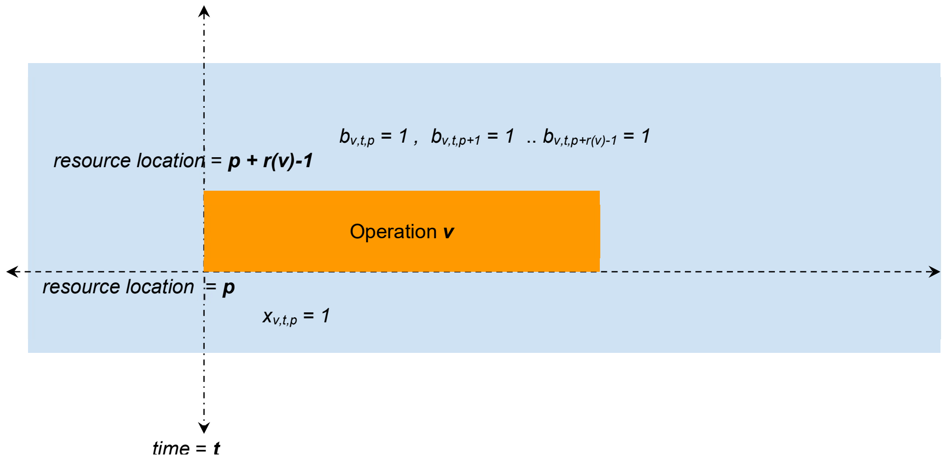
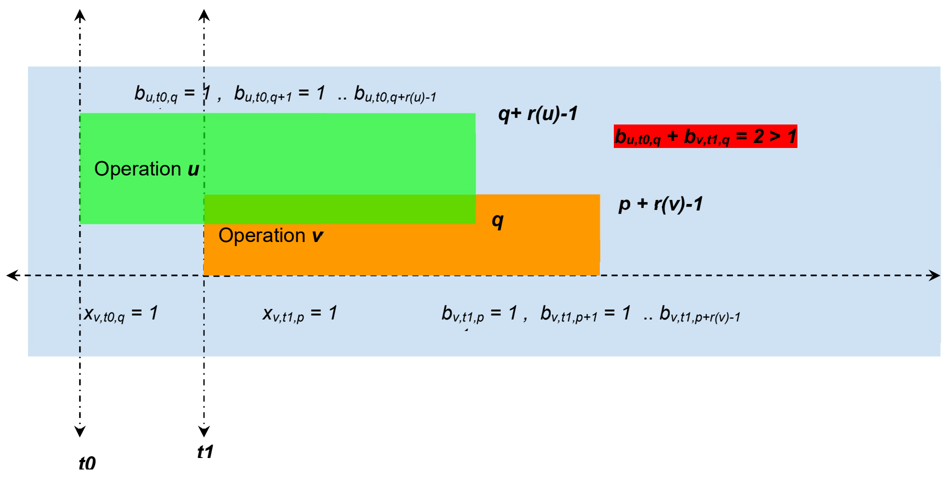

b_{v,t,p} = 1 , b_{v,t,p+1} = 1 .. b_{v,t,p+r(v)-1} = 1
resource location = p + r(v)-1
Operation v
resource location = p
x_{v,t,p} = 1
time = t

Fig-4: Variables corresponding to scheduling an operation $v \in V$ at time t and at resource location p



$b_{u,t0,q} = 1$, $b_{u,t0,q+1} = 1$ .. $b_{u,t0,q+r(u)-1}$
q+ r(u)-1
$b_{u,t0,q} + b_{v,t1,q} = 2 > 1$
Operation u
p + r(v)-1
q
Operation v
$x_{v,t0,q} = 1$
$x_{v,t1,p} = 1$
$b_{v,t1,p} = 1$, $b_{v,t1,p+1} = 1$ .. $b_{v,t1,p+r(v)-1}$
t0
t1

Fig-5: Illustration of schedule violating **(3.2)** since $b_{u,t_0,q} + b_{v,t_1,q} = 1 + 1 > 1$

Feasible Schedule Generating Algorithm:

Following is an algorithm which can provide an upper bound (tighter than simply adding delays of all the tasks). This algorithm generates a feasible schedule but may not be optimal but can be used as an input to the ILP as a makespan upper bound.

```
//INPUT: $G(V \cup \{v_{sink}\}, E)$  , $d: V \rightarrow N^+$, $r : V \rightarrow \{1,2 \dots k\}$, $R_{boun}$  .
//OUTPUT: schedule: $x_v$ ≡ start time of $v \in V$  meeting resource bounds//

GenerateFeasibleSchedule($G$, $d($ $)$, $r($ $)$, $R_{bound}$)

$U$ ≡ list of unscheduled ops topologically sorted from $G(V,E)$
$H$ ≡ min-heap to store end-times of active operations $(t,v)$
$R_{curr}$ ≡ current resource usage initialized to 0
$T$ ≡ current schedule time initialized to 0
$v_{sink}$ ≡ without the loss of generality assume exactly one sink node.
$d(v_{sink})$ ≡ 0 delay of the sink is zero.

While ($U$ != $\Phi$ )
  $v_{curr}$ = FindOpMeetingResourceBounds($U$, $R_{curr}$)
  If ($v_{curr} \in V$) // Found an operation to schedule
    $H.push$ (($T + d(v_{curr}), v_{curr}$) // Add the completion time to min-heap.
    $U = U - \{v_{curr}\}$ // Remove the operation from unscheduled list.
```

Commented [5]: This call is leaving out a lot of detail, like how we're tracking scheduled operations...

In practice, I've found it useful to maintain U as an actual graph, and to maintain a separate "Ready Set" R, defined as all V s.t. V has no incoming edges in U and V has not been scheduled. Scheduling an operation removes it from R; when an operation completes, it's removed from U, and all operations that now have no incoming edges become part of R.

Commented [6]: Yes, the details are left out purposefully. Because the core scheduling mechanism (outlined here) is seperated from the underlying policy of maintaining the resource state and selecting the next operation to schedule. See https://github.com/movidius/mcmCompiler/blob/master/src/scheduler/feasible_scheduler.hpp#L611

```
    $R_{curr}$ += $r(v_{curr})$  // Update the resource bound.
    $x_{v_{curr}}$ = $T$       // Schedule time of $v_{curr}$
  Else
    $(t, v_{curr}) = H.\min(\ )$ // Remove the smallest element for the heap.
    $T = t$          // Update the schedule time.
    $R_{curr}$ −= $r(v_{curr})$  // Update resource bound.

// Makespan bound is $T$ //
```

The algorithm starts with a topological order of the operations which are unscheduled at start. The algorithm maintains a state variable R_{curr} to keep track of the amount of resource usage by active tasks. In each iteration an operation is either scheduled or an operation which smallest completion time is found and removed (thus releasing some resources). The function **`FindOpMeetingResourceBounds`** looks at the current resources and identifies an unscheduled operation such that all its predecessors are completed. The algorithm terminates since the size of U monotonically decreases in one or more (depends on the maximum number of elements in the heap) iterations. An implementation of this algorithm can be found at **[2]**.

Post Process Feasible Schedule With Barrier Insertion:

In the previous section we have explained the details on how to generate *feasible* schedules. Consider a feasible schedule of the DAG in **Fig-1** which meeting the resource constraints (CMX). One additional requirement is that the CMX memory used by each operation should be contiguous. Consider the operation *e* (blue) in **Fig-2** as you can see that the schedule time of *e* (blue) is after the completion of operation *c* (red) even though there is no dependency between *e* and *c* (in the graph). However to ensure that the CMX memory of *e* is contiguous we add a *barrier* between *c* and *a* to ensure that all CMX virtual addresses of *e* are contiguous before start of *e*. This can be easily done as a post process of the schedule you scan the schedule from left to right and maintain a disjoint interval tree corresponding to the end-times of the operations just finished. At the start of each operation identify all overlapping operations w.r.t their virtual address ranges and put a barrier between them. Note that the *max* number of barriers you use is bounded by the maximum number of parallel operations which gets scheduled. This is illustrated in **Fig-3.**

Commented [7]: The previous algorithm only considered total resource usage, not spacial contiguity; virtual addresses have not been assigned. Note that unless virtual address assignment is taken into account during scheduling, adding barriers may substantially inflate the makespan.

Commented [8]: As I mentioned earlier the resource policy is separated from scheduling mechanism: see this for finer details https://github.com/movidius/mcmCompiler/blob/master/src/scheduler/feasible_scheduler_test.cpp#L616

Commented [9]: Assigning the virtual addresses is non-trivial.

Commented [10]: Hello Robert, we do the virtual address assignment for each of the tasks. We call them CMX addresses -- we use a disjoint interval set to keep track of the resource state. You can see the full scheduler code at : https://github.com/movidius/mcmCompiler/blob/master/src/scheduler/feasible_scheduler.hpp#L958

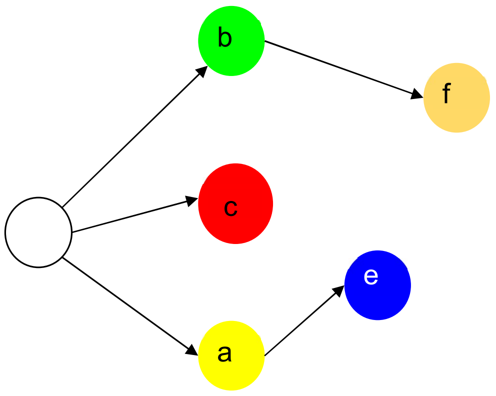


Fig-1: Input dependency DAG for the feasibility schedule generator

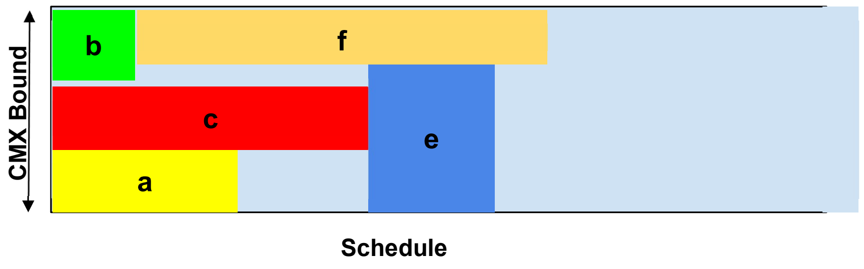


Fig-2: One possible feasible schedule

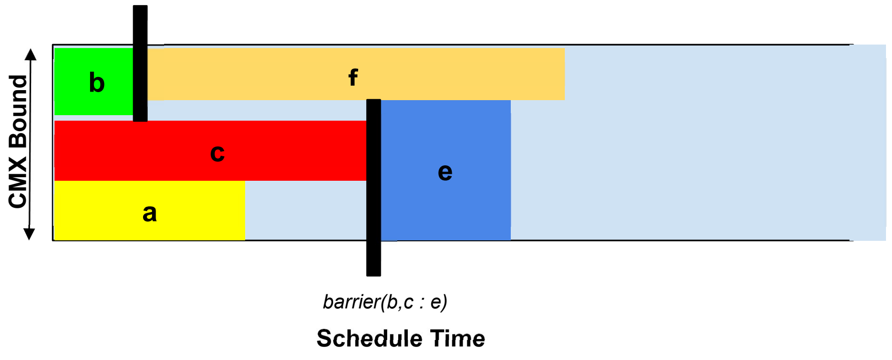


Fig-3: Post process schedule with barrier insertion to enforce *contiguous CMX* addresses.

An Algorithm for Control Edge Generation:

Given a feasible schedule this section describes an algorithm to output *control edges* -- from which *barriers* can be determined. As we have seen earlier for every feasible schedule ∃ a *packing* of the operations in an *open strip* of fixed height (resource upper bound e.g. CMX). We now focus on the following problem:

Let $P = \{r_1, r_2 \ldots r_n\}$ a packing of rectangles and $P(r_i) = (x_i, y_i)$ denotes the *lower left corner* (placement) of a rectangle $r_i \in P$. A *control edge* $e = (r_i, r_j)$ exists between a pair of rectangles $r_i$and $r_j$in the following *control edge property* ($CtrlProperty(r_i, r_j)$):

- $r_i$and $r_j$touch along a *vertical* edge (or)
- ∃ a horizontal *line* segment (not overlapping any rectangle in P) connecting a vertical edge of $r_i$with r_j (e.g. (A, B) in **Fig-4** notice A and B don't overlap vertically)

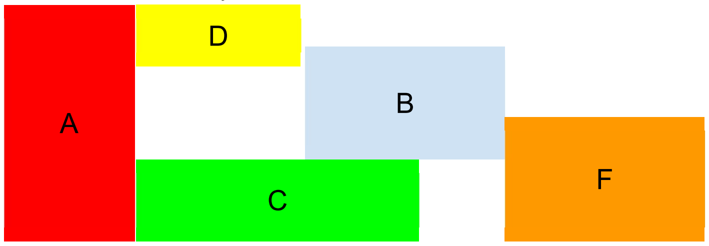


Fig-4: Input to the *control edge* generation algorithm.

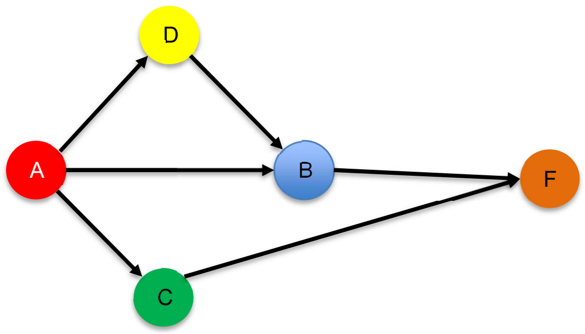


Fig-5: Control edges derived from the packing in **Fig-4**

We are now ready to describe our control edge generation algorithm:

```
//INPUT: packing $P = \{r_1, r_2 \ldots r_n\}$
//OUTPUT: control edge set $C = \{ (r_i, r_j) \mid CtrlProperty(r_i, r_j) = true \}$
GenerateControlEdges($P$)
$P_{sorted}$ ≡ lexicographically sorted rectangle set w.r.t lower left corner
$D$ ≡ disjoint interval set vertical intervals  and $D(i) = r_j \in P$

$D = \phi$ // Invariant: $D$ is collection of disjoint intervals //
Foreach $r_{curr} \in P_{sorted}$
  $i_{curr} = (y_{beg}, y_{end}) = LeftEdge(r_{curr})$
   $OverlapSet = i = (y_a, y_b) | (y_a, y_b) \in D \wedge i overlaps i_{curr}$
   If $OverlapSet.empty(\ )$ Then $D.insert(i_{curr}, r_{curr})$
   Foreach $i = (y_p, y_q) \in OverlapSet$
        $r_i = D(i)$ // output control edge ($r_i, r_{curr}$)
        $D.erase(i)$ // invariant holds
        $i_{union} = i \cup i_{curr}$
        $I_{xor} = i_{union} - (i \cap i_{curr}) = \{i_1, i_2\}$ // set of <= 2 disjoint intervals
        For $i_k \in I_{xor} \wedge NonEmpty(i_k)$
           If $i_k \subset i_{curr}$
              $D.insert(i_k, r_{curr})$ // invariant holds
           Else
              $D.insert(i_k, r_i)$ // invariant holds
         $D.insert((i_{curr} \cap i), r_{curr})$
```

The idea of the algorithm is to sweep from left to right over the *left* edges over the edges of rectangles in the packing and maintain a *disjoint interval set* (see **[2]**) over these edges. In each iteration we query the data structure to find the overlapping intervals, each of these overlapping interval either *touches* the current rectangle edge (vertically) or it satisfies the *visibility* property (see $CtrlProperty(r_i, r_j)$) which yields a control edge between the corresponding rectangles. Additionally we update the *disjoint interval set* as follows (**Fig-6**) to retain the invariant. The total work done by the algorithm is $O(n\log(m))$ where m is the maximum number of disjoint intervals in the data structure and $n = |P|$. An implementation of a complete algorithm (with some changes) is found at **[4]**.

Commented [11]: I don't think this produces the correct invariant if i_k also overlaps some other entry in the overlap set -- consider the case where you're processing an r_curr where the top and bottom both overlap separate existing intervals (ideally with a gap between them, since that makes the problem even more obvious -- you have to update i_curr somewhere to communicate what you've done between loop iterations). In practice, I don't think it matters; the interval set just won't be disjoint.

On the other hand -- I think you could make this code a bit simpler and more obviously correct (catching cases like this): just scan through the entries in the D, replacing each with i - i_curr (removing entirely if the result is empty, duplicating if i - i_curr creates two separate intervals); after that, it's always safe to add i_curr to D.

Commented [12]: If i_k is in the XOR set of the current interval and the intervals in the set D -- XOR means that that part of the interval does not overlap with any one. So I'm not sure what you are saying is correct. The complete control edge generation algorithm is here : https://github.com/movidius/mcmCompiler/blob/master/src/pass/lp_scheduler/control_edge_generator.hpp . **Please add a unit-test to create an example which you are trying to communicate.**

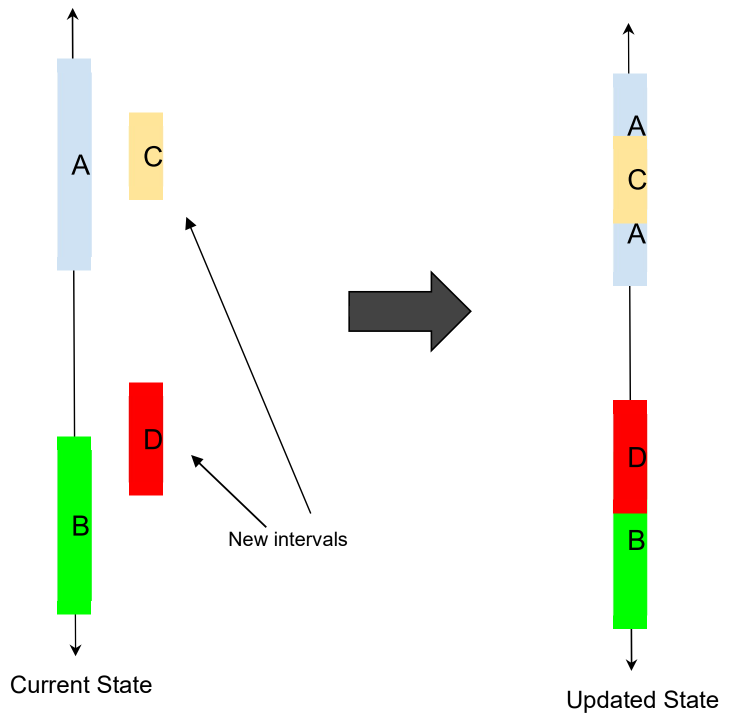


Fig-6: Invariant maintenance at each iteration.

Generating a Feasible Schedule Under Producer/Consumer Resource Constraints with Dynamic Spill Operations:

In the previous section we have seen an algorithm to generate a feasible schedule with a simple *resource constraint* model (S-R). We now show how to generate a feasible schedule under a slightly general model called *producer consumer resource constraint* model (PC-R). Consider the following dependency among operations in **Fig-7**. In this new model resources of the producers (blue) are engaged until the consumer operation (red) is completed.

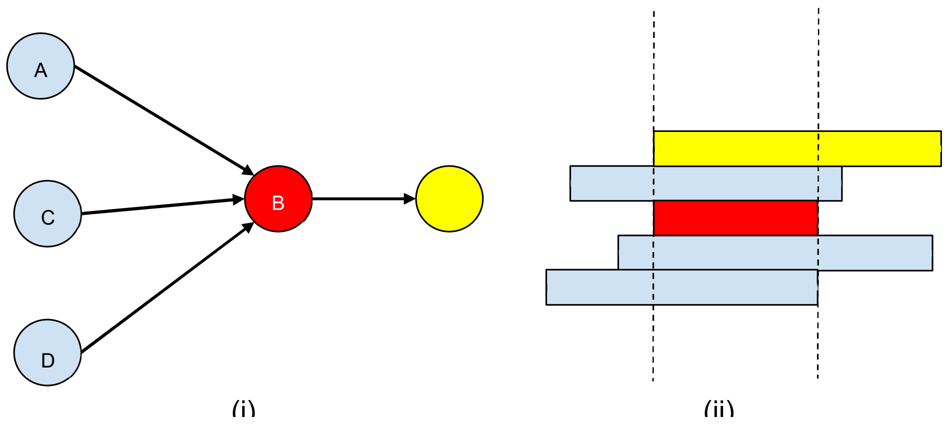


Fig-7: (i) producer operations = {A,C,D} , consumer operation = {B} (ii) resource engagement of the input (blue) and output(yellow) until completion of operation.

The key idea behind the *list scheduler* (in the previous section) is that every time an operation *completes* its resources are *released*. This guarantees that every operation is schedulable after waiting enough time. Unfortunately, this is not true in the PCR model

Under this new model the list scheduler in the previous section may *not make progress* since it may not be possible to find a *simultaneous resource engagement* for all the producer and consumer tasks. Hence for generating a feasible schedule under this model we assume the following precondition:

$$\forall v \in V \quad \sum_{(u,v)\in E} r(u) + r(v) \leq k$$

The above condition states that for every operation in the input the sum of resources used by all its inputs and itself cannot exceed the resource upper bound k.

Even with the above *precondition* there are situations where it's impossible (see **Fig-8**) to generate a schedule in this model without some additional operations which can *clear* some resources.

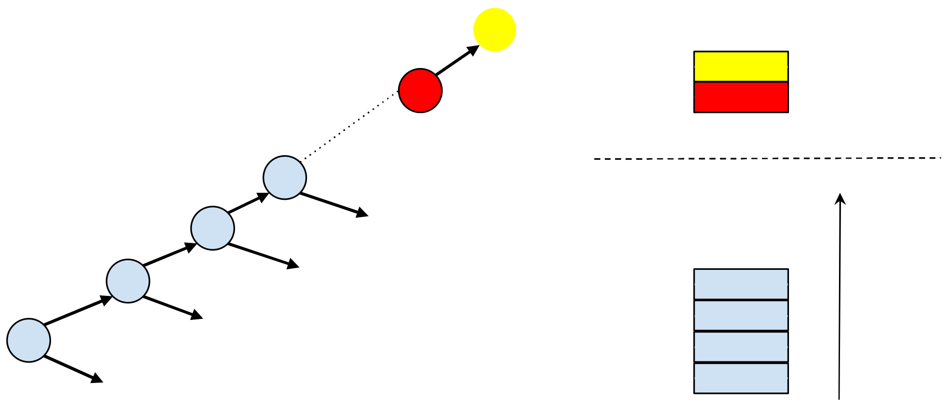


Fig-8: Illustration of a network where to schedule a single operation you need to store all the intermediate operations which can grow unbounded.

To encounter situations like these we provide the scheduler with what is called a *spill operation* which lets the scheduler to clear out some resources by temporarily moving them to a buffer resource bank (e.g. DDR). This additional operation along with the precondition lets us generate a feasible schedule for any network which meets the precondition.

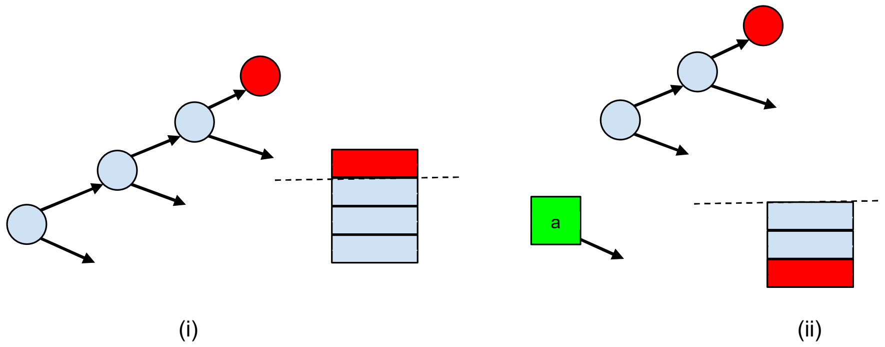


Fig-9: (i) The operation *d* cannot be scheduled since it exceeds the upper bound on resource usage. (ii) scheduling operation *d* by spilling out the output of operation *a* to a buffer.

We now describe an algorithm which can dynamically add these spill operations do generate a schedule which meets the resource constraints. Full implementation of this algorithm is available at [3].

```
//INPUT: $G(V,E)$ , $d:V \to N^+$, $r :V \to \{1,2 \ldots k\}, R_{bound}$
//OUTPUT: schedule: $x_v$ ≡ start time of $v \in V$ meeting resource bounds
and producer consumer resource constraints. //
```

```
**GenerateFeasibleSchedulePCR**($G$, $d(\ )$, $r(\ )$, $R_{bound}$)

$L_{active}$ ≡ list of all ready operations with at least one of its producers
(inputs) actively using resources.  (**initialized to $\Phi$**)
$L_{no\text{-}active}$ ≡ list of all ready operation with none of its producers
actively using resources (**intialized to set** ).
$R_{state}$ ≡ current resource usage state
$T$ ≡ current schedule time initialized to 0
$H$ ≡ min-heap to store end-times of active operations $(t, v)$

While ( $!L_{active}.empty(\ )$ and $!L_{no-active}.empty(\ )$ ){

    // STEP-0: remove all scheduled ops ending at time T //
    do{
     $(T, v_{curr}) = H.popmin(\ )$ // Remove the smallest element for the heap.
      ClearResourcesAndUpdate($v_{curr}$, $R_{state}$, $L_{active}$, $L_{no-active}$ )
    } While $(!H.empty() \wedge (T_{top}, v_{top}) = H.\min() \wedge T_{top} == T)$

   //STEP-1: try to schedule ops from active producers //
   $V_{schedule} = \Phi$
   While ($!L_{active}.empty(\ )$ and $V_{scheduled} == \phi$) {
      $V_{scheduled}$ = FindAllOpsMeetingResourceBounds($L_{cmx}$ ,$R_{state}$)
      If ($V_{scheduled} = \phi$)
         $v_{LRU}$ = EvictLeastRecentlyUsedResouce($R_{state}$)
         AddSpillOperation($v_{LRU}$) // update current time T //
         $L_{update} = \{ w \mid (u, w) \in E \wedge u \notin R_{state} \}$
         $L_{active} = L_{cmx} - L_{update}$
         $L_{no-active} = L_{no-cmx} \cup L_{update}$
      Else
         $L_{active} = L_{active} - V_{scheduled}$
         UpdateResourceState($R_{state}$, $V_{scheduled}$)
   } //end while part-1 //

   // STEP-2: schedule op from non-active ready list //
   If ($V_{scheduled} == \phi$) {
     assert( $R_{state} = \phi$)  // no active producers engaging resources//
     $V_{fresh}$ = FindAllOpsWhichCanBeScheduledNoActiveResourceUse($L_{no-cmx}$)
     UpdateResourceState($R_{state}$, $V_{fresh}$)
     $L_{no}$     = $L_{no-cmx} - V_{fres}$
  }// end if part-2 //
  // Output the scheduled ops $V_{schedule}$ at time $T$
}
// Makespan bound is $T$ //
```

Example Network to Illustrate Dynamic Spill Function:

We now illustrate the function of *dynamic spill operations* on a simple synthetic network in **Figure-10**.

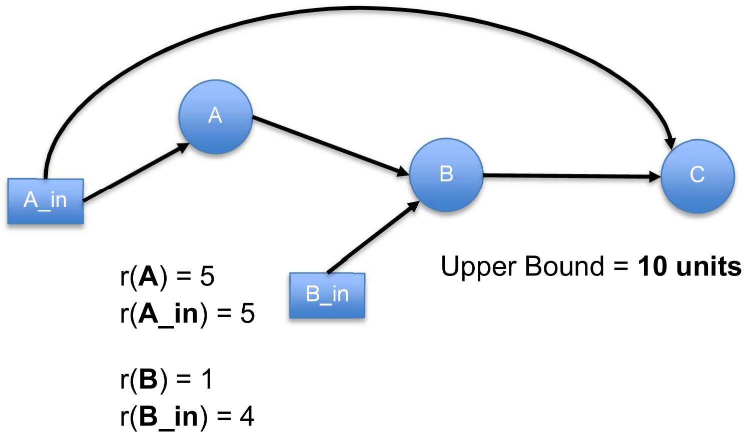


Figure-10: Simple network to illustrate schedule generation with dynamic spill operations.

In this example we use a resource upper bound of *10* units. The input *A_in* is consumed both by operation *A* and operation *C*. Hence the *A_in* need to be active till *C* finishes and it consumes *5 units* , however the operation *B* needs to use *10* units of resource and its not possible to execute operation *B* and hold *A_in* as it exceeds the upper bound of *10 units.* Also to make progress we need to execute operation *B*. This is where the dynamic spill operations help the algorithm described in the previous section automatically detects that it cannot make progress and will add an implicit *write (spilled write)* operations to clear the resources to make progress and will add corresponding *implicit read* operations for any future operations which needs to the spilled output.

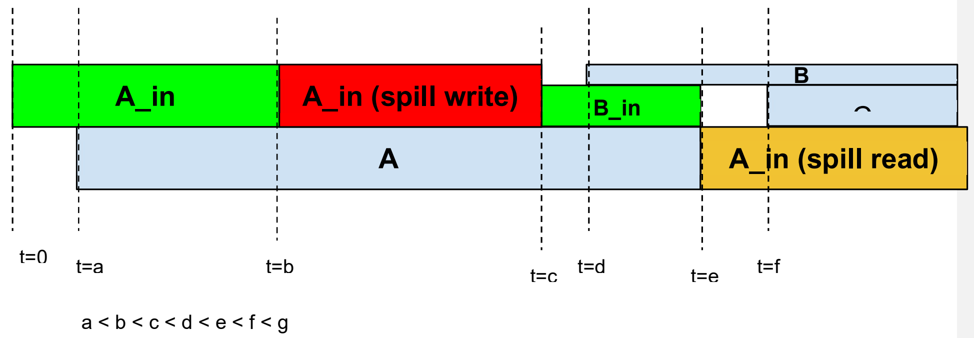


Figure-11: Feasible packing within resource constraints with spill write and read operations

```
op = A_in         type = ORIGINAL          time = 1   resource=[6 10]
op = A            type = ORIGINAL          time = 2   resource=[1 5]
op = A_in         type = SPILLED_WRITE     time = 3   resource=<none>
op = B_in         type = ORIGINAL          time = 4   resource=[6 9]
op = B            type = ORIGINAL          time = 5   resource=[10 10]
op = A_in         type = SPILLED_READ      time = 6   resource=[1 5]
op = C            type = ORIGINAL          time = 7   resource=[6 9]
op = C            type = SPILLED_WRITE     time = 8   resource=<none>
```

Figure-12: Text schedule generated by the program.

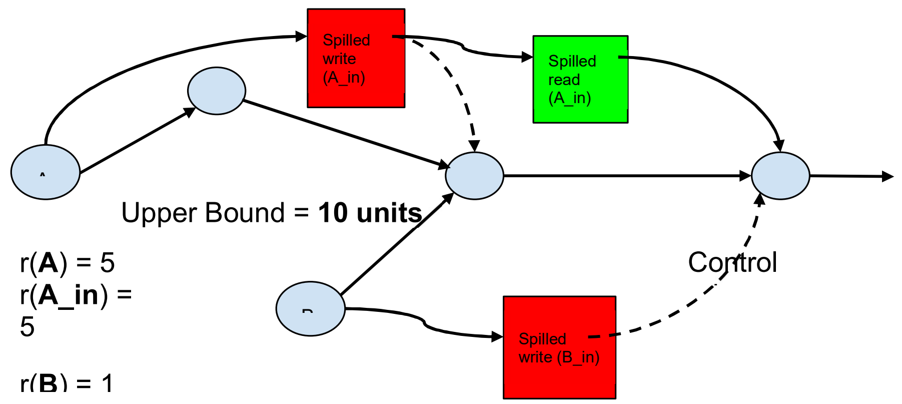


Figure-11: Updated OpControl model with new spill write/read ops and control edges.

Scheduling with Barrier Resource Constraints:

In this section we address the problem of generating a *feasible schedule* with a specified number of *Barriers*. Recall that barriers are physical synchronization primitives in hardware.

Scheduler has access to only a finite number of them ($b \leq 8$). The barrier scheduling problem is a special case of *concurrency control problem* on a operation precedence graph $G_c = (V, E_c)$ (see constraint **(4)** in the section on ILP formulation). Barrier scheduler need to ensure that in *all possible schedules* of $G_c$the number of concurrent operations does not exceed the specified number b. To achieve this the scheduler may need to impose some additional *pseudo dependencies* (control edges) among the operations in G_c. We need to clarify the following terminology before we describe the algorithm and its correctness.

Preliminaries

Relation: Given a set $S = \{u_1, u_2, \dots u_m\}$ a *relation*R on S is a set of ordered pairs $R = (x, y) | x, y \in S$. Relation is sometime expressed succinctly using some symbol $<$. E.g. $u_i < u_j$ iff $(u_i, u_j) \in R$. Given a DAG $G = (V, E)$ the set of edges E can be associated with a relation $<$. If $u < w$ then there is an edge $(u, w) \in E$.

Partial Orderings on a DAG: Given a DAG $G = (V, E), |V| = n$ a *partial order* p is a bijection $p: V \rightarrow [1,2,3 \dots n]$ such that $u < w \rightarrow p(u) < p(w)$. This partial order is expressed as a sequence $\{p^{-1}(1), p^{-1}(2) \cdots p^{-1}(n)\}$. Notice that there are $n!$ bijections however not all of them are *partial orders*.

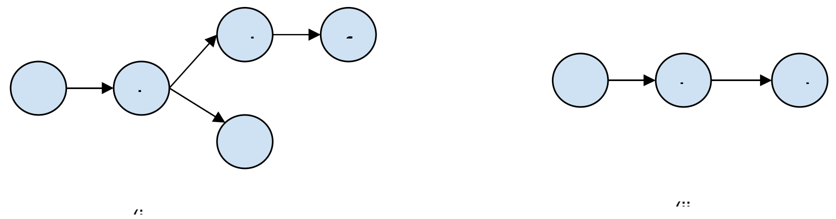


Figure-12: DAG in (i) has *three* partial ordering *{a,b,d,f,c}*, *{a,b,c,d,f}* and *{a,b,d,c,f}* out of 120=5! bijections. DAG (ii) has only one partial ordering out of 6=3!

Directed Closure: Given a DAG $G = (V, E)$the *directed closure* $G^* = (V, E^*)$ of G is defined as follows: $E^* = (u, w) | u, w \in V \wedge path(u, w) \in G$ -- we have an edge between u and w iff there is a directed path between those nodes in G.

Maximal Directed Independent Set: Given a DAG $G = (V, E)$ then a subset $M \subset V$ is an *independent set* iff $\forall u, w \in M \nexists path(u, w) \in G$ -- for any pair of nodes is Mthere is no connecting path between them. This subset M is *maximal* iff you cannot add any more nodes to M and still keeping it as an *independent set*.

Maximum Directed Independent Set: Given a DAG $G = (V, E)$, let Ube a set of all *maximal directed independent* sets of G. Then a *maximum directed independent set* $M^*$is defined as follows: $M^* = argmax_{M \in U} \{ |M| \}$ -- largest cardinality *maximal* directed independent set. Notice that a *maximum* directed independent set *is a maximal* directed independent set. However a *maximal* directed independent set may not be a *maximum* directed independent set.

Maximum Directed Closure Independent Set: Given a DAG $G = (V, E)$ the a *directed maximum closure independent set* in G is a maximum directed independent set in directed closure of G.

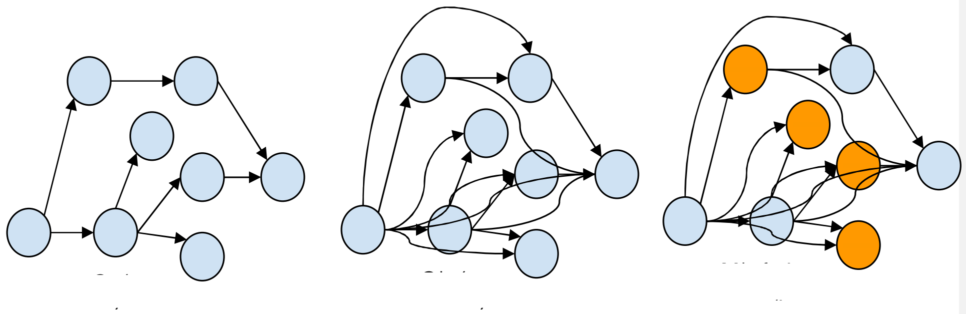


Figure-12(a): DAG in (ii) is a *directed closure* of DAG in (i) and the *maximum directed closure independent set* $M^*$is highlighted.

Characterizing Concurrency in an Operation Precedence Graph:

Let $G = (V, E)$be an operation precedence graph and $G^* = (V, E^*)$be its directed closure. A key sub-problem for barrier scheduling is to determine the *maximum* number of tasks which can run in parallel among all possible schedules -- *partial orderings.* We now show that this problem is equivalent to determining the cardinality of *maximum directed closure independent set* of G.

Lemma-1: Two operations $u, v \in G$ can be simultaneously active in any schedule of G, if and only if there is no edge between u and $v$in the directed closure of G.

Proof: (*forward proposition*) *If* u, v are simultaneously active *then* $(u, v) \notin E^*$. Consider the contrapositive, which means $(u, v) \in E^*$this means there is a path connecting u and v in G which means in any valid schedule v can only start when $u$finishes. **QED**

(*reverse proposition*) *If* $(u, v) \notin E^*$ *then* u, v are simultaneously active. Since $(u, v) \notin E^*$there is no path between u and v in G. We now show how to construct a partial order such that u and v can be active simultaneously. Let $w$be a *lowest common ancestor* of u, v. If w does not exist then just start the partial order with u, v ... which will u, w active simultaneously. On the other hand if $w$exists let $\{w, u_1, u_2 \cdots u_p, u\}$, $\{w, v_1, v_2 \ldots v_q, v\}$ be the paths from w to u, v respectively. Without the loss of generality let $p \leq q$ then we construct the following partial order $\ldots w, u_1, u_2 \ldots u_p, v_1, v_2 \ldots v_q, v, u \cdots$ where u, v are adjacent and since there is no edge between them they can be active simultaneously. **QED**

Theorem-1: In any schedule of G the largest number of simultaneously active operations is bounded (less than or equal) by the size of maximum directed closure independent set of G.
Proof: By Lemma-1 two operations can be simultaneously active iff there is no edge between them in G^*. Let M^*be the maximum directed closure independent set of G, hence by definition for any pair of nodes in M^*there is no edge between them in G^*. Thus $|M^*|$ is the upper on the number of simultaneously active tasks in G. **QED**

Problem Statement

Based on *Theorem-1* we can reduce the problem of *barrier scheduling* can be reformulated as follows.

Input: $G_c = (V, E_c)$ and a natural number $b \geq 2$
Output: $G_{c^*} = (V, E_c \cup E_{c^*})$ such that the cardinality of *maximum directed closure independent set* in G_{c*} is $\leq b$

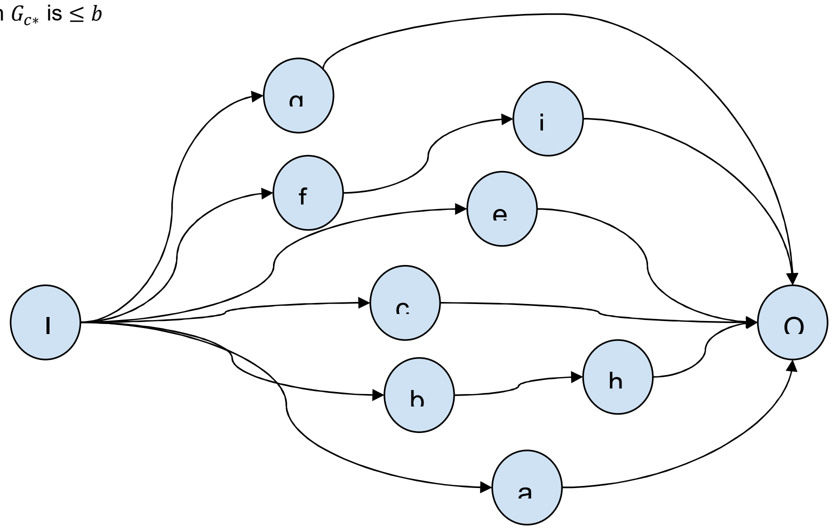


Figure-12: Example input $G_c = (V, E_c)$ to the algorithm with barrier bound $b = 4$

An Algorithm to Solve Concurrency Control Problem:

As in the previous sections we use a variant of the *list scheduler* to generate a schedule. However we need to precisely define the resource and delay models. Since barriers are asynchronous we use an *unit-delay* model. We now describe the resource model

Barrier Resource Model

First we need to ask the following basic question:
Why does an operation/task need a barrier resource ? The answer is to *indicate* its completion to its adjacent tasks. Thus every scheduled operation with at least one outgoing edge needs an *update barrier* (see **Figure-12(a)**). Without the loss of generality we assume that a scheduled operation *exclusively* owns the update barrier and all barriers are *update barriers* -- and the scheduler has access to b of them. We will show later how these constraints can be relaxed (see section on *wait barriers* and *barrier slots*).

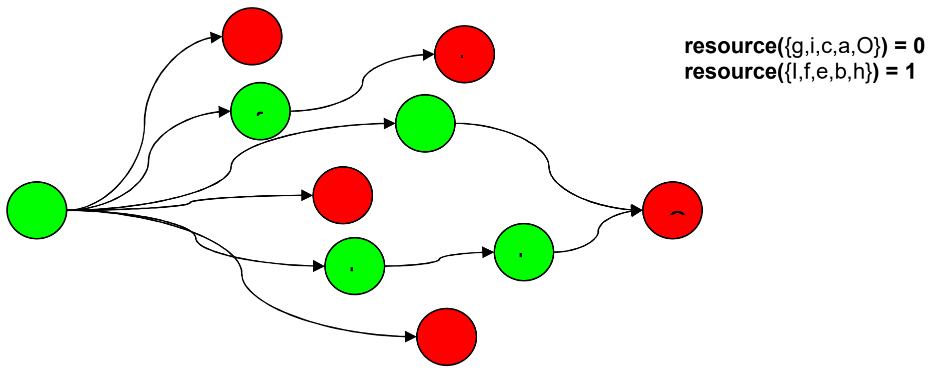

resource({g,i,c,a,O}**) = 0**
resource({l,f,e,b,h}**) = 1**

Figure-12(a): Update barrier resource demand depends on the out-degree of an operation/task.

Schedule Generation

We now just call the feasible schedule generator with a resource upper bound of b (see Feasible Schedule Generation section) . The result is a feasible schedule with schedule time associated with each operation. For the input in **Figure-12** and $b = 4$. The output of the feasible scheduler looks something like **Figure-13 .** Now the goal is to take the schedule and add control edges such that *maximum directed closure independent set* size does not exceed the upper bound b.

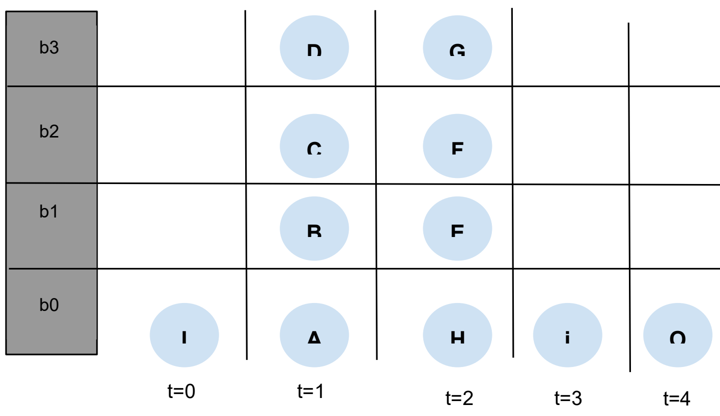


Figure-13: Output of the list scheduler on the DAG in *Figure-12* with four barrier resources.

Constructing a b-Maximum Closure Independent Set DAG:

As shown in *Theorem-1* if the size of maximum closure independent set is bounded by b then we cannot have more than b simultaneously active tasks. Since every active task exclusively holds a update barrier this means that if are able to construct such a DAG then the number of active update barriers are bounded by b. The output of the feasible schedule from the previous section (see Figure-13) is a triplet (b_i, t, v) corresponding to update barrier id $(1 \leq b_i \leq b)$, schedule time and operation $v \in G_c$. Following is a simple algorithm takes the schedule and produces edges to construct a *b-maximum closure independent set* DAG:

```
//INPUT: schedule $S = (v, t, b_i) | v \in G_c, 1 \leq b_i \leq b, G_c = (V, E_c)$
//OUTPUT: edge set $E^*$ such that the maximum closure independent set in
$G^*{}_c = (V, E_{c^*})$ is bounded by $b$
GeneratebMaximumIndependentSetEdges ($S$)
  $lastOp[1 \dots b] = \{\phi\}$
  $E_{c^*} = \phi$
  Foreach $(v, t, b_i) \in S$ //schedule already sorted on t
      If $lastOp[b_i] \neq \phi$
            $E_{c^*} = E_{c^*} \cup (lastOp[b_i], v)$
      $lastOp[b_i] = v$
```

For the schedule in **Figure-13** the new edges generated by the algorithm are shown in **Figure-14(a)** and its topologically sorted in **Figure-14(b).**

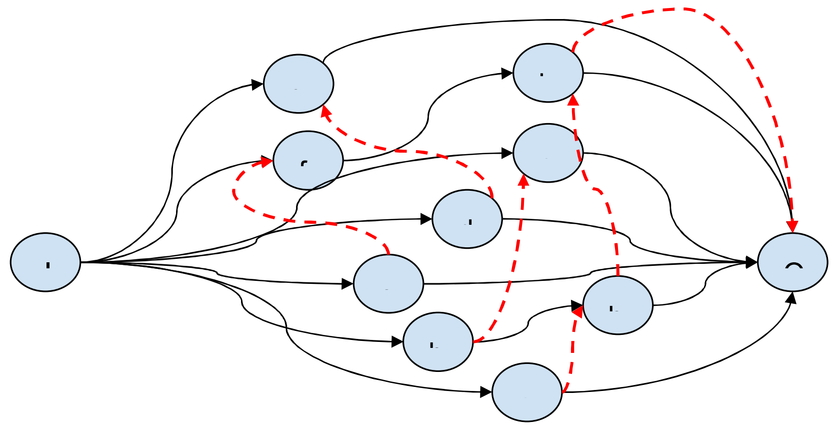


Figure-14(a): New control edges (red) E_{c^*}generated from the output of the list scheduler in Figure-13

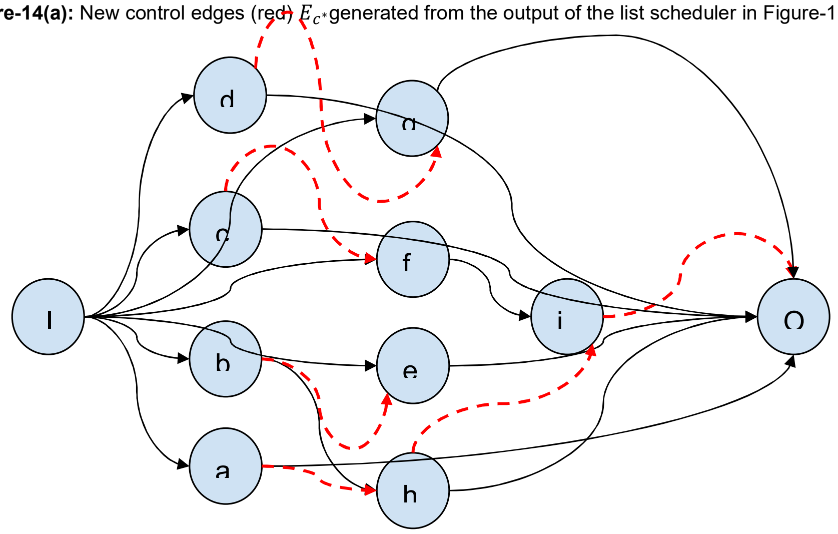


Figure-14(b): Topologically sorted final output $G_{c^*} = (V, E_c \cup E_{c^*})$

Correctness Proof of the Construction:

The only thing remaining, is to show the resulting DAG $G_{c^*} = (V, E_{c^*})$ *cannot* have a maximum closure independent set M^* such that $|M^*| > b$. We can trivially prove this by contradiction. Let's assume that $M^* = \{u_1, u_2 \dots u_q\}, q > b$, since the scheduler assigns the barrier indexes to each of the operations let $B = \{b_1, b_2 \dots b_q\}$ be the set of barrier indexes notice that $|B| \leq b$ this means that $\exists u_i, u_j \in M^*$ such that $b_i = b_j$ (by pigeon hole principle). However if the $u_i$and u_j then the algorithm **`GenerateMaximumIndependentSetEdges`** adds edges such that there should be a path between $u_i$and u_j-- this means $M^*$is not an independent set. **QED**.

Barrier Slots and Wait Lists

In the previous section we have seen a resource model where an operation exclusively owns an update barrier. This means that the update barrier has a *unit producer count,* $p = 1$. This is too restrictive resource model, in fact the hardware allows a finite producer count (e.g. $p \leq 256$) for each of the update barriers -- this means multiple tasks can update the same active barrier. This can be easily incorporated into the resource model with *barrier slots* , in addition to the upper bound b on the barriers we can assume that each of these barriers has p slots. The update barrier demand is now expressed as the number of slots required. In the context of KMB micro-architecture the number of slots for a DPU task is nothing but its *workloads* and for a DMA tasks it's simply 1.

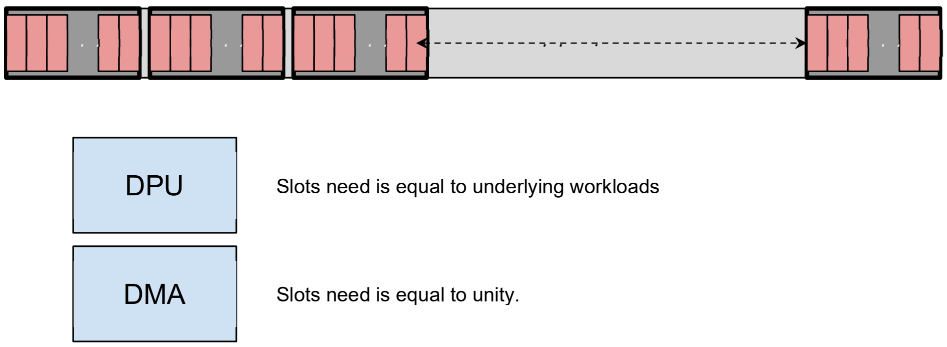

DPU
Slots need is equal to underlying workloads
DMA
Slots need is equal to unity.

Figure-15: Resource model to support finite producer count in update barriers.

Let $U(v)$be an update barrier associated with an operation $v \in G_{c^*}$. The wait barrier list $W(v)$can be easily computed as follows: $W(v) = \{ U(w) \,|(w, v) \in G_{c^*}$

Transform 1-0 LP to Integer LP by Reducing to 2D Strip Packing.

The ILP formulation in the previous section involving Boolean variables can be *re-written* into a *Two-Dimensional Strip Packing Problem* (2D-SP) with a smaller number of variables and additional constraints. The input to the 2D-SP problem is a set $R = \{r_1, r_2 \dots r_n\}$ of rectangles and open ended vertical strip of width W, let (w_i, h_i) denote the width and height of a rectangle r_i and let (x_i, y_i) be its placement inside the strip. The goal of 2D-SP is to pack the rectangles into the strip such that no two rectangles overlap and height H of the strip is *minimized.* Notice that the upper bound on $H \le H_{fsg}$ is the result obtained from running the list scheduler from the previous section.

How to Express Non-Overlap of Rectangles as Linear Constraints?

Given two rectangles r_i, r_j the sufficient condition for their *non-overlap* is the following:

$$(x_i + w_i \le x_j)\ or\ (x_j + w_j \le x_i)\ or\ (y_i + h_i \le y_j)\ or\ (y_j + h_j \le y_i)$$

Notice the above is **union (OR)** of linear inequalities which is not a convex region and cannot be an input to a linear program -- the feasible region of a linear program is the **intersection (AND)** of linear inequalities which are convex sets and intersection of convex sets is always convex.

So how to express the above condition as a set of linear inequalities? Following is a well-known technique to achieve this. First, we introduce *four* Boolean variables t_L, t_R, t_B, t_T corresponding to *left, right, bottom* and *top*. Now keeping one of the rectangles r_i fixed and the non-overlapping rectangle should lie in one of the *four* regions, see **Figure-16**.

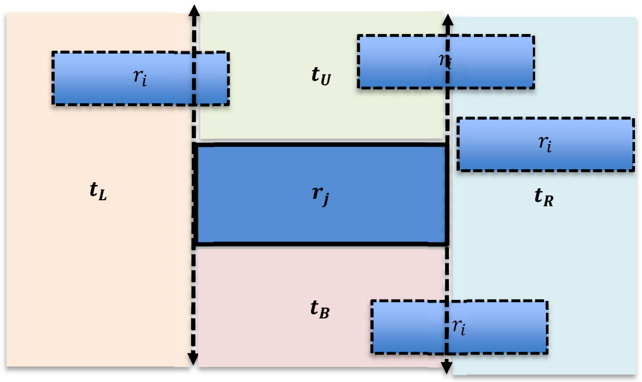


Figure-15: Partitioning the space around a rectangle to express non-overlap as linear constraint.

Now consider the following for inequalities $L_{\widehat{overlap}} \equiv le_1, le_2, le_3, le_4, le_5$:

$$\begin{aligned}
x_i + w_i &\le x_j + 2W \times t_R && (le_1)\\
x_j + w_j &\le x_i + 2W \times t_L && (le_2)\\
y_j + h_j &\le y_i + 2H_{fsg} \times t_U && (le_3)\\
y_i + h_i &\le y_j + 2H_{fsg} \times t_B && (le_4)
\end{aligned}$$

$$\sum_{k \in L,U,R,B} t_k \le 3, t_k \in 0{,}1 (le_5)$$

Theorem (*Non-Overlap*): Two rectangles $r_i = (x_i, y_i, w_i, h_i)$ and $r_j = (x_j, y_j, w_j, h_j)$ are non-overlapping *if* and *only if* all the linear constraints in $L_{\widehat{overlap}}$ are satisfied.
Proof: We have seen that the sufficient condition for non-overlap is the predicate $P(r_i, r_j)$:

$$P(r_i, r_j) = \ (x_i + w_i \le x_j\ )\ or\ (x_j + w_j \le x_i)\ or\ (y_i + h_i \le y_j)\ or\ (y_j + h_j \le y_i)$$

We will now show that for a given pair $r_i = (x_i, y_i, w_i, h_i)$ and $r_j = (x_j, y_j, w_j, h_j)$ for which $P(r_i, r_j) = true,\ \exists\ \ (t_R, t_L, t_U, t_B) \in \{0,1\}^4$. Note there are *three* different inequality combinations` which will make this predicate true, we will demonstrate only one case and rest of them are similar:

CASE-I: $(x_i + w_i > x_j) or (x_j + w_j > x_i) or (y_i + h_i > y_j) or (y_j + h_j \le y_i)$ (only one condition true and rest of them are false).

$$0 \le x_i, x_j, w_i, w_j \le W\ \ (Given)\quad 0 \le y_i, y_j, h_i, h_j \le H_{fsg}\ \ \ (Given)$$
$$x_i + w_i \le W + W = 2W \Rightarrow x_i + w_i \le (2W + x_j) = (x_j + 2W \times 1)$$
$$x_j + w_j \le W + W = 2W \Rightarrow x_j + w_j \le (2W + x_i) = (x_i + 2W \times 1)$$

$$y_j + h_j \le H_{fsg} + H_{fsg} = 2H_{fsg} \Rightarrow y_j + h_j \le (2H_{fsg} + y_i) = (y_i + 2H_{fsg} \times 1)$$
$$y_i + h_i \le y_j \Rightarrow y_i + h_i \le (y_j + 2H_{fsg} \times 0)$$

Based on the above inequalities we assign: $t_R = 1$, $t_L = 1$, $t_U = 1$, $t_B = 0$, $(t_R + t_L + t_U + t_B) = 3$. Same reasoning can be applied for other remaining cases.

Finally consider the case where $P(r_i, r_j)$, following the above approach will lead to $t_R = 1, t_L = 1, t_U = 1, t_B = 1$, which makes $(t_R + t_L + t_U + t_B) = 4$ (violating the constraint $(t_R + t_L + t_U + t_B) \le 3$. **QED.**

Standard 2D-SP ILP formulation:

$$\min\ \{\ z_H\ \},$$
$$x_i + w_i \le W$$
$$y_i + h_i \le z_H \le H_{fsg}$$
$$\forall\ r_i, r_j \in R,\qquad i < j$$
$$x_i + w_i \le x_j + 2W \times t^{i,j}{}_R$$
$$x_j + w_j \le x_i + 2W \times t^{i,j}{}_L$$
$$y_j + h_j \le y_i + 2H_{fsg} \times t^{i,j}{}_U$$
$$y_i + h_i \le y_j + 2H_{fsg} \times t^{i,j}{}_B$$
$$\sum_{k \in L,U,R,B} t^{i,j}{}_k\ \le 3, t^{i,j}{}_k \in 0{,}1$$

If $|R| = n$, the above LP has $2n + 5 \times \frac{n\times(n+1)}{2} + 1 = \Theta(n^2)$ constraints and $4n + 4 \times \frac{n\times(n+1)}{2} + 1 = \Theta(n^2)$ variables. Please see [7] for a complete GLPK encoding of the 2D-SP problem.

A Note on Modeling Indicator Variables and Absolute Values in ILP:

Commented [KK13]: 03/20/2021 - theorems basis for squishing infrastructure: https://github.com/movidius/mcmCompiler/blob/2dStripPackingSolver/src/pass/lp_scheduler/2d-strip-packing/2d_strip_packing_lp.hpp#L1290

Theorem (*Indicator Invariant*): Let $\boldsymbol{t_i} \in [\mathbf{0},\mathbf{1}]$ an *indicator variable* and $x \in [0, M)$ be an *integral variable* and Q: $t_i = 0 \Leftrightarrow x \geq b, \mathrm{b} \in [0, \mathrm{M})$ is an invariant. Then in *every* feasible ILP solution with following constraints the invariant Q holds:

$$x \geq \mathrm{b} - t \times M \qquad (1)$$
$$x \leq (b-1) + (1-t) \times M \qquad (2)$$

Proof: (forward direction): given a feasible solution prove that Q holds.
Notice that the solution space is partitioned into *four* regions:

$$(x \geq b, t = 1), (x \geq b, t = 0), (x < b, t = 1), (x < b, t = 0)$$

We now show that two of following possibilities are infeasible:

- $(x \geq b, t = 1)$: if $t = 1 \rightarrow x \geq b - M(true), x \leq (b-1)(false)$. Hence this variable combination cannot be feasible.
- $(x < \mathrm{b}, t = 0)$: If $t = 0 \rightarrow x \geq b(false), x \leq (b-1) + M(true)$. Hence this variable combination cannot be feasible.
- $(x \geq b, t = 0)$: if $t = 0 \rightarrow x \geq b(true), x \leq (b-1) + M\ (true)$. Since $x < M, (b-1) \geq 0 \rightarrow x < M + (b-1)$. Hence this variable combination is feasible and satisfies the invariant Q
- $(x < b, t = 1)$: if $t = 1 \rightarrow x \geq b - M\ (true), x \leq (b-1)\ (true)$. Since $0 \leq x < M$, $b < M \rightarrow b - M < 0 \leq x$. Hence this variable combination is feasible and satisfies the invariant Q

(reverse direction) trivial. **QED.**

Theorem (*Absolute Value Constraints*): Let $z, y \in [-M, M], t \in 0{,}1$ be integer variables and $Q{:}\, y = |z|$ be an invariant. Then in every feasible ILP solution with following constraints the invariant Q holds:

$$-z \leq y \leq z \qquad \textbf{(1)}$$
$$z \geq \mathrm{y} - \mathrm{t} \times 2\mathrm{M} \qquad \textbf{(2)}$$
$$-z \geq \mathrm{y} - (1-\mathrm{t}) \times 2\mathrm{M} \qquad \textbf{(3)}$$

Proof: $\boldsymbol{y} = |\boldsymbol{z}| \rightarrow \boldsymbol{y} \leq |\boldsymbol{z}| \wedge \boldsymbol{y} \geq |\boldsymbol{z}|$

- $|\boldsymbol{z}| \leq \boldsymbol{y} \rightarrow -\boldsymbol{y} \leq \boldsymbol{z} \leq \boldsymbol{y}$ -- this derives **(1)** (easy case of continuous range)
- $|z| \geq y \rightarrow -z \geq y$, $(z < 0)$ or $z \geq y$, $(z > 0)$

- If $z \geq y$ then $t = 0$ satisfies **(2)** and **(3)** – also $t = 1$ violates **(3)** hence $t = 0$ indicates $z \geq y$
  - **(3)** : $y - 2M \leq M - 2M = -M \leq -z$
- If $-z \geq y \equiv z \leq -y$ then $t = 1$ satisfies **(2)** and **(3)** -- also $t = 0$ violates **(2)** hence $t = 1$ indicates $-z \geq y$
  - **(2)** : $y - 2M \leq M - 2M = -M \leq z$

QED

Relaxing Packing with Possibly Coinciding Rectangles:

We have seen in the previous section on how to model non-overlapping of rectangle packings are linear constraints. We now slightly relax the packing with possibly *coinciding rectangles* in the packing. Recall that in any valid packing $\forall r_A, r_B \in R$ we should have $r_A \cap r_B = \Phi$. We now allow the possibility of *certain pair of rectangles* to *coincide*.

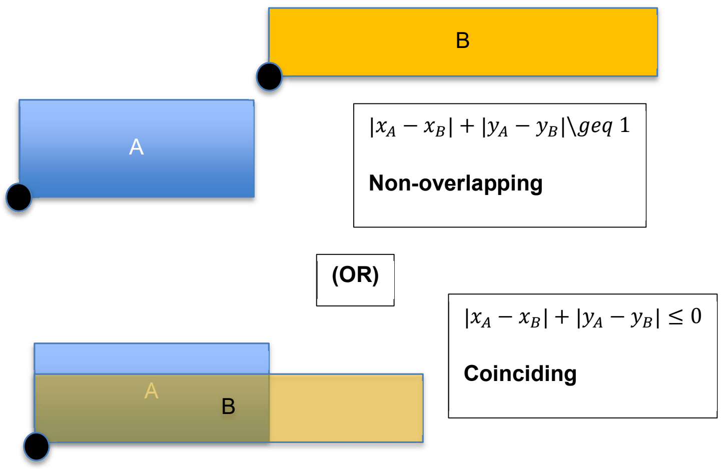
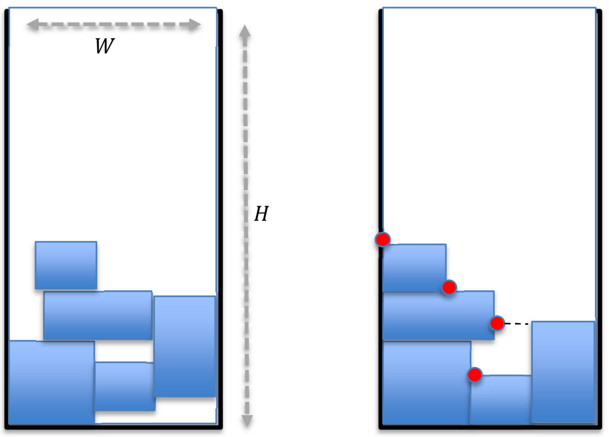

B
A
$|x_A - x_B| + |y_A - y_B|\geq 1$
Non-overlapping
(OR)
$|x_A - x_B| + |y_A - y_B| \leq 0$
Coinciding
A
B

Figure-16: Partitioning the space around a rectangle to express non-overlap as linear constraint.

$$x_i + w_i \leq x_j + 2W \times t_R \qquad (le_1)$$
$$x_j + w_j \leq x_i + 2W \times t_L \qquad (le_2)$$
$$y_j + h_j \leq y_i + 2H_{fsg} \times t_U \qquad (le_3)$$
$$y_i + h_i \leq y_j + 2H_{fsg} \times t_B \qquad (le_4)$$
$$z = |x_i - x_j| + |y_i - y_j| \qquad (le_5)$$
$$z \geq 1 - \mathbf{t_{coincide}} \times 2M \qquad (le_6)$$
$$z \leq (1 - \mathbf{t_{coincide}}) \times 2M \qquad (le_7)$$

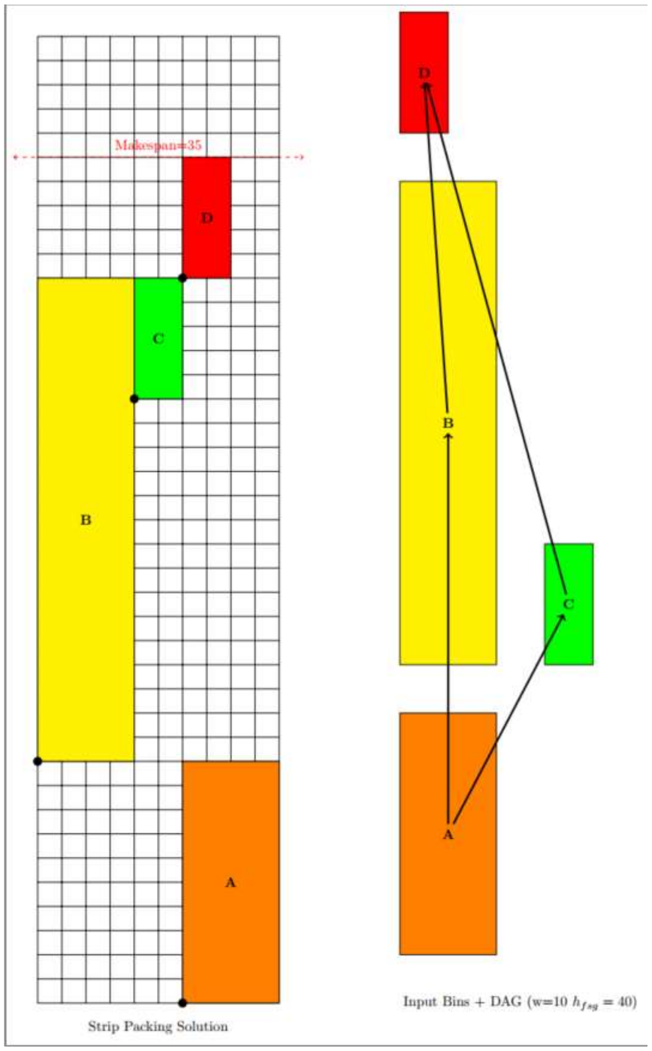
$$\sum_{k \in L,U,R,B} t_k - \boldsymbol{t_{coincide}} \leq 3,\ t_k, t_{coincide} \in 0,1 \qquad (le_8)$$

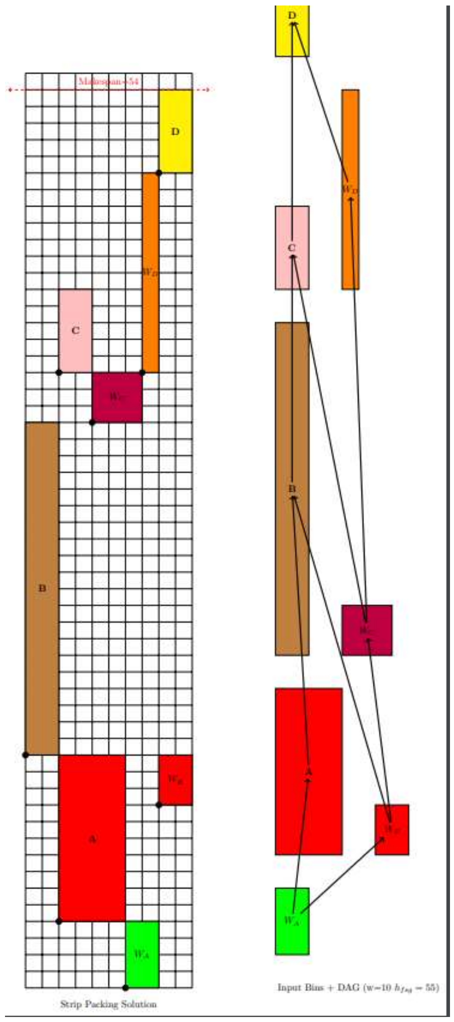
Additional Constraints to 2D-SP (WIP)

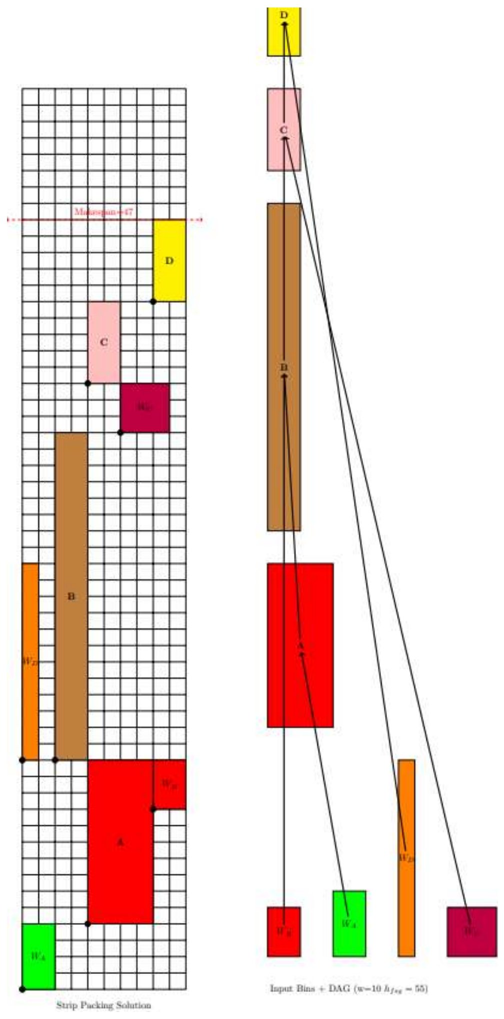


(a) (b)

Figure-16: (a) BL-Unstable packing **(b)** BL-Stable packing with corner points (red)


Makespan=35
D
C
B
A
Strip Packing Solution
D
B
C
A
Input Bins + DAG (w=10 h_{fsg} = 40)


Makespan=54
D
W_D
C
W_C
B
W_B
A
W_A
Strip Packing Solution
D
W_D
C
B
W_C
A
W_B
W_A
Input Bins + DAG (w=10 h_{fsg} = 55)



References:

[1] *Giovanni De Micheli,* Synthesis and Optimization of Digital Circuits, McGraw-Hill.
[2] Feasible Schedule Generation Under Resource Constraints: https://github.com/movidius/mcmCompiler/blob/feasible-scheduler-algorithm/src/scheduler/feasible_scheduler.hpp
[3] Disjoint Interval Set Data Structure: https://github.com/movidius/mcmCompiler/blob/feasible-scheduler-algorithm/src/scheduler/disjoint_interval_set.hpp

[4] Control Edge Generation Algorithm: https://github.com/movidius/mcmCompiler/blob/feasible-scheduler-algorithm/src/pass/lp_scheduler/control_edge_generator.hpp
[5] Dynamic Spill Scheduling Algorithm (Feasible_Memory_Schedule_Generator): https://github.com/movidius/mcmCompiler/blob/feasible-scheduler-algorithm/src/scheduler/feasible_scheduler.hpp
[6] Unit Test Corresponding to Figure-10 :https://github.com/movidius/mcmCompiler/blob/feasible-scheduler-algorithm/src/scheduler/feasible_scheduler_test.cpp
[7] Linear Programming Encoding of 2D-SP problem: https://github.com/movidius/mcmCompiler/blob/2dStripPackingSolver/src/pass/lp_scheduler/2d-strip-packing/2d_strip_packing_lp.hpp