The List Scheduling Approach

One of the most used approaches used to solve the temporal partitioning problem is the list scheduling (LS) method. List scheduling was first used as method for microcode compaction in order to generate efficient microcode from high-level languages. The idea behind list scheduling in architectural synthesis is to first sort the nodes in topological order and then assign some priority to the node of a dataflow graph. There are several methods to assign a priority to a node. A common strategy is to use the latency weighted depth of the node as its priority. The ***depth*** of a node *v* is defined as the length (number of nodes) of the longest path from an input to *v*. The ***latency weighted depth*** is the same as the latency depth, however the path from the input to *v* are weighted using the latency of the operation to be executed by the nodes on the path. Also the ***mobility*** of a given node, i.e. the difference between its ALAP-value and its ASAP-value (discussed in the previous lecture), can be used as its priority.

At any time step *t*, the so called *ready set*, that is the set of operations ready to be scheduled, is constructed. The ready set contains operations whose predecessors have already been scheduled early enough to complete their execution at time *t*. The algorithm checks if there are enough resources of a given type *k* to implement all the operations of type *k*. If so, the operations are assigned the resources, otherwise, higher priority tasks are assigned the available resources and the rest of the operations will be scheduled later, when some resources will be available. If the mobility of a node is used as priority criteria, it is possible that all operators in the ready list are on a *critical paths*, which means that their mobility is zero. As consequence, the complete depth of each operators is increased by one, thus increasing the latency of the graph's execution.

Example: Consider the graph shown in Figure 1 on the left side, in which each node is labelled with its priority. The nodes must be scheduled on a resource set made upon an adder and a multiplier. With the priority of a node defined as its depth, the list scheduling is shown on the right side.

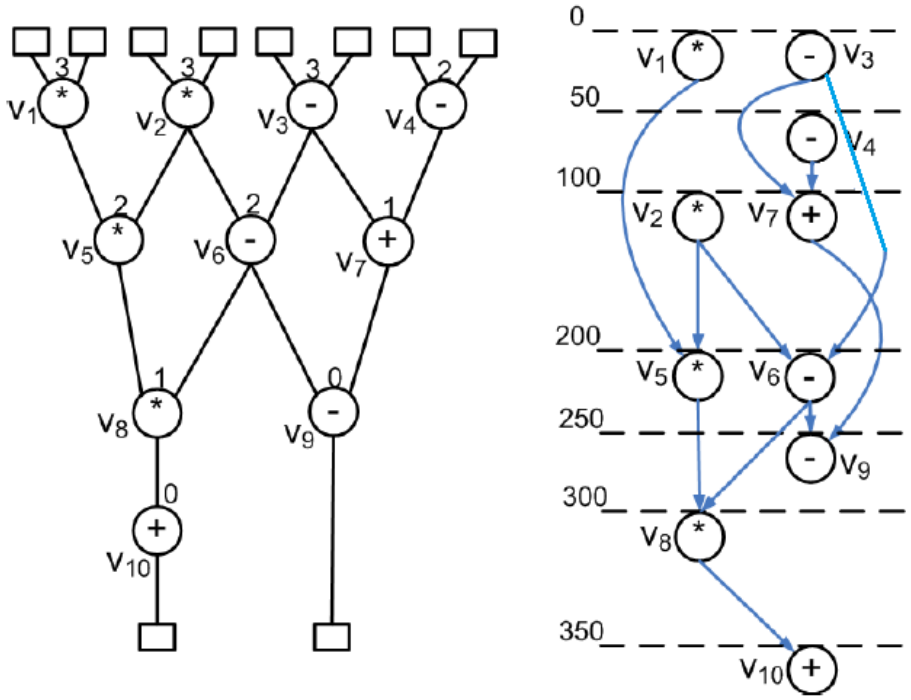

Figure 1: An example of list scheduling using the depth of a node as priority

Forced-Directed List Scheduling: The *Force-Directed List Scheduling (FDLS)* is an extension of the list-scheduling algorithm. Instead of computing the priority for each node at the beginning, priorities are dynamically computed at each step. Force-directed list scheduling is a resource constrained scheduling method aimed at finding a minimum latency schedule for a given resource set. The concept of *force* is used to select the operation to be assigned a given resource. First the possible *time frame*, i.e. the mobility interval of each node is computed using the ALAP and ASAP scheduling. Next, the probability *p(v, t)* of a node *v* to be scheduled at step *t* is computed for each node *v* of the graph. The probability *p(v, t)* is zero outside the mobility interval of *v* and equal the reciprocal of the mobility within the mobility interval. Formally, we have:

$$f(n) = \begin{cases} \frac{1}{mob(v)+1} \; if \; t \in [ASAP(v), ALAP(v)] \\ 0 \qquad\qquad\qquad\qquad\qquad Otherwise \end{cases} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Where $mob(v) = ALAP(v) - ASAP(v)$

A distribution function *d(t, k)* is calculated as the sum of probabilities of all the operations with a resource type *k*. Formally, we have:

$$d(t,k) = \sum_{\{v \in V, \alpha(v)=k\}} p(v,t) \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

This can be plotted into a graph called the distribution graph that indicates the concurrency of similar operations over the schedule steps. Whenever many operators compete for a fewer amount of resources, the operations that produce a global increase of concurrency in the graph are selected and assigned the resources.

List Scheduling for Reconfigurable Devices: In high-level synthesis for reconfigurable devices, the resource type does not play a big role. Only the total amount of basic resources is important. We don't have operators competing for a resource type but for a given surface on the device. Furthermore, we are interested in building complete partitions describing a set of computational steps that will be performed at the same time. Despite the fact that any component has a different starting time and a different end time, only the starting time and the end time of the complete partition is usually considered.

The list-scheduling algorithm in reconfigurable devices works in the same way as the common list-scheduling algorithm, by building and updating a list of ready operators. Components are then removed from the ready list and assigned to partitions. The only assignment criterion is that there should be enough places left on the device to accommodate the new component. If this is the case, then the component is placed on the partition currently being built. Otherwise, a new partition is built and the process is repeated until all the nodes of the graph are placed in partitions. The pseudo code for this approach is provided in algorithm 1. Because the method is an iterative heuristic, several improvements can be done during the construction in order to generate optimal solutions.

Algorithm 1 List-scheduling scheduling algorithm for reconfigurable devices

sort the nodes of v according to their priorities
$P_0 := \emptyset$
while $V \neq \emptyset$ **do**
  select a vertex $v \in V$ with highest priority and whose predecessors are all placed
  if (a partition P_i exists with $s(P_i) + s(v) \leq s(H)$) **then**
    $P_i = P_i \cup \{v\}$
  else
    create a new partition P_{i+1} and set $P_{i+1} = \{v\}$
  end if
end while

Optimization usually considers the minimization of the overall computation latency of the dataflow graph. The two main factors that influence the overall latency are the latency of each segment and the configuration overhead. If *k* segments are generated by an algorithm, then the overall computation time can be formally written by following equation 3.

$$t_{DFG} = k \times C_H + \sum_{i=1}^{n}(t_{pi}) \quad \text{.....(3)}$$

where t_{DFG} is the overall computation latency for a dataflow graph DFG, C_H is the reconfiguration time of the device H and t_{Pi} is the computation time of partition *i*.

The minimization of the overall computation time of the given function can be done by tuning with the different parameters in equation 3. If the reconfiguration time of the device is too big, then the optimization will tend to minimize the number of partitions in order to avoid a lot for reconfigurations. However, if the reconfiguration time can be neglected, then only the delay in partitions will be of great interest and a good algorithm will tend to avoid long paths in partitions. The main advantage of the list scheduling method is it's linear run-time in the number of nodes in the graph. Furthermore the method allows for local optimizations while selecting the nodes to be placed in partitions.

List scheduling has a major drawback called the *levelization*. Levelization means that the partitions are built on the basis of the level number of the components in the dataflow graph. Modules are assigned to partitions based more on their level number, rather than their interconnectivity with other components already placed in the partition. This may have a bad effect on the quality of the partitioning as result of a bad connectivity inside the partitions and between the partitions. With this, the goal of minimizing the number of nets connecting two different partitions, i.e. the minimization of the data exchange among partitions becomes difficult to reach.

Consider the circuit of Figure 2(a) partitioned by a list-scheduling algorithm that produces three partitions. While the components are less connected inside the partitions, the partitions have more edges among each other, thus leading to a poor quality. A better partitioning is shown in Figure 2(b) for the same graph. The connectivity is better preserved inside and outside the partitions.

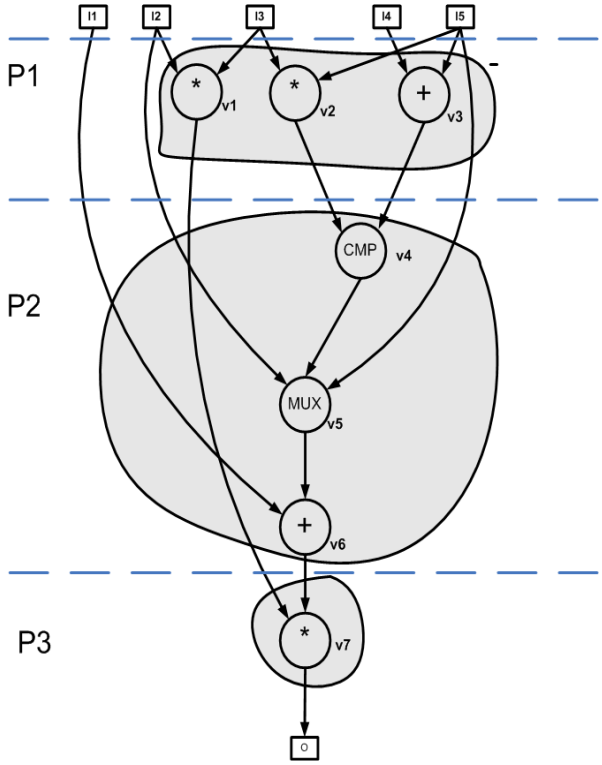

Figure 2(a): Levelizing effect on the list-scheduling on a dataflow graph

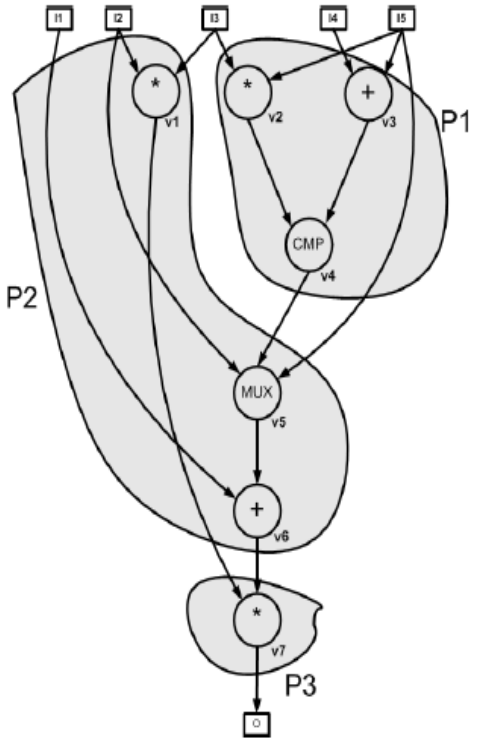

Figure 2(b): Partitioning with a better quality than the list-scheduling

Despite its drawback in connectivity, the list scheduling algorithm remains, thanks its linear computation time, a good temporal partitioning candidate, in particular for low-connected graphs. Also, it can be used to construct initial solutions that can be improved by more complex iterative improvement optimization heuristics.

Integer Linear Programming

The approach consists of formulating the temporal partitioning problem as an integer linear programming problem instance and then to use well known tools to compute the solution to the equation. All problems related constraints are defined as a set of equations involving only integer numbers. The equations must be solved subject to a given goal to be minimized.

The most important constraints that need to be captured in an ILP problem instance are the *unique assignment constraint*, the *precedence constraint* and the *resource constraint*. The integer requirement in the equation vector entries is due to the fact that we deal with indivisible variables and therefore no real number must be involved.

Before we present the formal description of the constraints, we would like to provide the definitions of the variables that will be used in the equations.

Definition (0-1variable) *Given a dataflow graph* G = (V,E) *and a partitioning* P = P1, ..., Pn *of* G *sorted in order of precedence. We defined 0-1 variable* y *and* w *as follow:*

- $y_{vi} = 1 \Leftrightarrow v \in P_i$
- $w_{vv} = 1 \Leftrightarrow \big((u,v) \in E\big) \wedge (u \in P_i) \wedge \big(v \in P_j\big) \wedge (P_i \neq P_i)$

The role of the *y* variables is to define the memberships of a given node to a partition. A value of one means that the node is in the corresponding partition, while a value of zero means that the node is not in the partition. In the same way a 0-1 variable for an edge determines if that edge connects two different partitions. The 0-1 variables provide the basis for defining the set of ILP equations, that we next formulate.

Unique assignment constraint: This constraint is used to control the assignment of a node to a single partition. Each node must be assigned exactly to one partition. The unique assignment constraint is formally defined by equation 4.

$\forall v \in V, \sum_{i=1}^{n} Y_{vi} = 1$ …….(4)

The sum of the assignment index of a given variable over all partitions is exactly one, which means that there is exactly one value *i* for which the *y* variable is one. This is the index *i* of the partition in which *v* is placed.

Precedence constraint: This constraint is used to control the assignment of data dependent nodes of the dataflow graph to partitions. A node *v* that is data dependent from a node *u* must be placed in a partition with index bigger or equal than that of the partition into which the node *u* is placed. This constraint is captured by the equation 5:

$\forall (u,v) \in E, \sum_{i=1}^{n} i \times y_{ui} \leq \sum_{i=1}^{n} y_{vi}$ …… (5)

The two sums define the index of the partition in which the components are placed.

Resource constraint: The resource constraint defines the constraint on the architecture used. This can be limited to the reconfigurable device or to a complete system into which the reconfigurable device is integrated. For a reconfigurable device, the area constraint as well as the constraint on the number of terminals must be specified. The area constraint states that the total amount of resources assigned to a given partition must not exceed the amount of available resource on the chip. Recall that the computation resources in a reconfigurable device are defined in term of area occupy on the device. The constraint can therefore be expressed in term of device area: the total area assigned to a given partition must not exceed the device area. This is formally defined by equation 6.

$\forall P_i \in P, \sum_{v \in P_i} a(v) \leq a(H)$ …… (6)

The terminal constraint defined in equation 7 states that the total number of input and output in a partition must not exceed the total number of pins on the device.

$\forall P_i \in P, \sum_{(u \in P_i) \wedge (v \notin P_i)} w_{vv} + \sum_{(u \notin P_i) \wedge (v \in P_i)} w_{vv} \leq p(H)$ …….... (7)

where p(H) is the number of terminals (pins) of the device H.

Communication memory constraint: The total amount of data to be temporally saved must not exceed the size of the communication memory used. This constraint is captured by the following equation 8:

$$\sum_{(u,v)\in E \ and(u\in P_i)\wedge(v\in P_{j(j>i)})} w_{vv} \leq M_s \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots(8)$$

Where w_{uv} is the width of the edge (u,v) and M_s is the size of the communication memory.

Having formulated the constraint as ILP equations, commercial solvers can be invoked to compute the solutions of the equations.

Temporal Placement

Till now, we presented the high-level synthesis problem for reconfigurable devices and some solution approaches. The result is a set of partitions that are used to reconfigure the complete device. While the implementation of single partitions is easy, the amount of waste resources in partitions can be very high. Recall that the waste resource of a component is the amount resources occupied by that component multiplied by the time where the component is idle on the device.

Wasting resources on the chip can be avoided if any single component is placed on the chip only when its computation is required and remains on the device only for time it is active. With this, idle components can be replaced by new ones, ready to run at a given point of time. Exchanging a single component on the chip means reconfiguring the chip only on the location previously occupied by that component. This process is called *partial reconfiguration* in contrast to *full reconfiguration* where the full device must be reconfigured, even for the replacement a single component. In order to be exploited, the partial reconfiguration must be technically supported by the device, which is not the case for all available devices. While most of the existing devices support full reconfiguration, only few are able to be partially reconfigured.

For a given set of operations to be executed, the resource allocation on the device is a time dependant process in which not only the placement of the components on the device is defined, but also the time slot in which the execution of the task must be performed. The time dependant placement of task on the device is called *temporal placement*.

Temporal placement can be graphically illustrated through an arrangement of rectangular boxes in a 3-dimensional container whose base is defined by the surface of the device and the high is the time axis. Each box represents components with a given surface placed at a given location on the device at a time defined on the time axis.

In figure 3 a temporal placement of a set of components {v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8} is given. Component v_1 for example occupies a surface on the defined by its length and height for a time slot 0 to 55, which corresponds to its computation latency. Component v5 starts its execution at time 90 and occupies among others, part of the device that was previously occupied by component v4. Component v8 occupies a part of the device for the whole computation time.

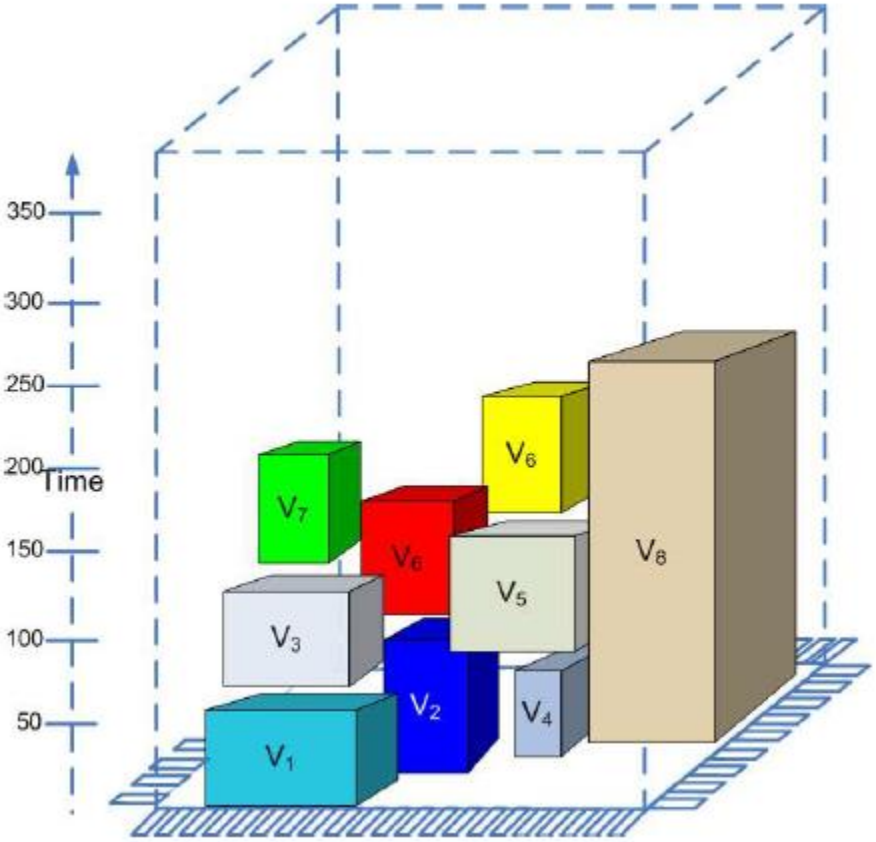

Fig 3: Temporal placement as 3-D placement of blocks

The computation of a temporal placement for a given set of components representing some tasks to be implemented can be done off-line, at compile-time or on-line, while the device is executing. In the first case, for an application specified as dataflow graph that must be computed on the device, the computation sequence is defined at compile- time and remains fixed for the given function. In on-line temporal placement or simple said, on-line placement, the computation sequence is not known in advance. The creation of task that must be placed on the device is done dynamically at run-time. It is therefore not possible to capture the specification of an algorithm for on-line placement through a given model at compile-time.

Temporal placement has the advantage of being highly flexible and efficient in term of device utilization, however it also has a major drawback. Efficient temporal placement algorithms are difficult and cost intensive. On-line placement requires to solve some computational intensive problems at run-time in a fraction of millisecond. Those are: the efficient management of the free space, the selection of the best site for a new component and the management of communication. For a new module to be placed at run-time, it is not sufficient to only compute the best placement site. The communication between the modules running on the chip must also be considered. The communication between modules running on the chip and the external world must be taken in to account as well. A reconfigurable device must provide viable on-line communication mechanisms to help establish the communication among modules on the chip at run-time. This problem is not easy to solve and requires most of the time, some prerequisites on the device architecture.

Although communication aspects like the distance among components are considered in some of the methods provided in this chapter, we do not deal with the technical realization of

communication here. We assume that communication among modules placed on the device is somehow possible.

Off-Line Temporal Placement

We first provide a formal definition of the temporal placement problem before investigating existing solution approaches.

Definition 5.2 ((Off-line) Temporal placement) Given a dataflow graph G = (V,E) and a reconfigurable device H with length ($H_x$and width H_y), a temporal placement is a 3-dimensional vector function:$p = (p_x, p_y, p_t): V \rightarrow^3$, where p_t defines a feasible schedule.

For a given node vi, the values $p_x(v_i), p_y(v_i)$ and $p_t(v_i)$ denote the coordinates of the node v_i in a 3 dimensional vector space. $p_x(v_i)$ and $p_y(v_i)$ are the coordinates of v_i on the device H, while $p_t(v_i)$ defines the starting time of v_i on H.

We next present some solution approaches for the off-line temporal placement problem. We first introduce a simple incremental method based on the first-fit/best-fit concept.

First-Fit and Best-Fit Placement

Given a dataflow graph for which a temporal placement must be computed, a simple approach will consist of continuously keeping track of free locations on the chip. The nodes to be placed will then be selected according to their readiness. A node is said to be ready to be placed, if all it predecessors are already placed. For each selected node, one of the free locations on the chip is selected and the node is placed at that location, provides that the space is large enough to accommodate the selected node. If the selection procedure chooses the first free place on the chip to place the new component, then we call it a *firs-fit*.

The *firt-fit* procedure is fast and run in linear time in the number of free locations. However, if we assume that each selected location can accommodate only one component, the amount of unused resource created by the component can be very high, if a location is selected whose surface is bigger than that of the component. To void this situation, we may choose a best fit strategy in which the component is placed on the location whose surface is the closest to that of the component. We call this approach a *best-fit*. *Best fit* is much time consuming than first-fit, because the complete list of free locations must be searched for the best location for the component to be placed.

The definition of a location has a great importance in the firs-fit and best-fit strategy. What is a location? A simple rectangle? or is it a shape with an arbitrary contour? For sake of simplicity, we assume that a location is a rectangle, in which case the test of inclusion of a given component in a free location is easy. What happens with the new free locations that results from the placement of a component on a selected free area of the device? Just ignoring those free fragmented locations can be inefficient. In many case the total amount of fragmented area can be enough to hold some components. If we don't keep track of the fragmented area and if those fragmented area are not consecutive, then many components cannot be placed on the device, although enough free space exist to accommodate those components. Managing the fragmented locations in turn requires a very large book keeping

efforts. Later we will present some algorithms that deal with the fragmentation issues in the context of on-line temporal placement.

A possibility to keep the first-fit and the best-fit approaches simple is to segment the device in sectors, each of which is defined as a location. Such a location is able to hold a given amount of components, whose total area cannot exceed the size of that sector. This approach is attractive for many devices in which the partial reconfiguration is done with some restrictions, as it is the case for example with the Xilinx Virtex and Virtex II FPGA, where the reconfiguration can be done only column wise. Whenever a component has to be exchanged, all the components sharing some columns with that component are affected. The situation is better in the Virtex 4 and Virtex 5 families, where the reconfiguration does not affect the complete device column, but only the complete height of a rectangular block within the chip. By defining the location to span the complete device column in the case of the Virtex and Virtex II and the block height for the Virtex 4 and Virtex 5, we avoid to disturb the components already running on the device during the reconfiguration process.

Given a dataflow graph and partitioned locations in a device, we would then like to place several components at the same time at a given location on the device. The dataflow graph must then be partitioned in sets of components such that each set can fit into a location. Several partitioning strategies exist in the literature. However, most of them do not take the precedence constraints into account. Partitioning strategies are not presented in this section. We refer to the temporal partitioning algorithms presented in the previous chapter that can also be used here. In this case, the size constraint must se to be the size of a location. In order to limit the amount of wasted resources on the chip, a clustering algorithm will be used to place components in partitions according to their finishing time. This has the advantage that all the components finish at the same time, thus minimizing the wasted amount of resources.

Let now assume that for a given a dataflow graph G = (V,E) a partitioning or better say a clustering C = {C1, ...,Cn} is computed such that all the clusters fulfil the size constraints. Algorithm 2 computes a temporal placement of C in a first-fit manner. For a new cluster Cact to be placed on the device, the algorithm checks for the cluster Ctop with the minimum run-time among the clusters allocated to the device for which the precedence constraint Ctop ≤ Cact between Ctop and Cact holds. The cluster Cact is placed on top of Ctop. The placement of cluster Cact on top Ctop simply means that the resources allocated to Ctop will be allocated to Cact after Ctop has completed he's execution. The algorithm stops when all the clusters have been placed.

Algorithm 2 First-fit temporal placement of clusters

while while all the clusters are not placed do **do**
  Select one cluster C_{act} from the list of *ready to run* clusters
  From the clusters already placed, select the cluster C_{top}
  with the smallest finishing-time such that $C_{top} \leq C_{act}$
  place the cluster C_{act} on top of C_{top}
end while

Figure 4 illustrates the firs-fit temporal cluster placement on a set of ten clusters. We assume that the precedence constraints are satisfied during the placement. At the beginning, the

clusters C0, C1, C2, C3 *to* C4 *are completely placed on the device. In order to place cluster* C5, *cluster* C0 *with the earliest finish time is selected and cluster* C5 *is placed on top of* C0. *Cluster* C6 *occupies the space freed by cluster* C2 *and cluster* C7 *is placed on top of* C3. C8 *is then placed on top of* C4, C9 *on top of* C1 *and* C10 *on top of* C5. *The configuration sequence produced by this example is* ({0, 1, 2, 3, 4} → {5, 1, 2, 3, 4} → {5, 1, 6, 7, 4} → {5, 1, 6, 7, 8} →{5, 9, 6, 7, 8} → {10, 9, 6, 7, 8}).

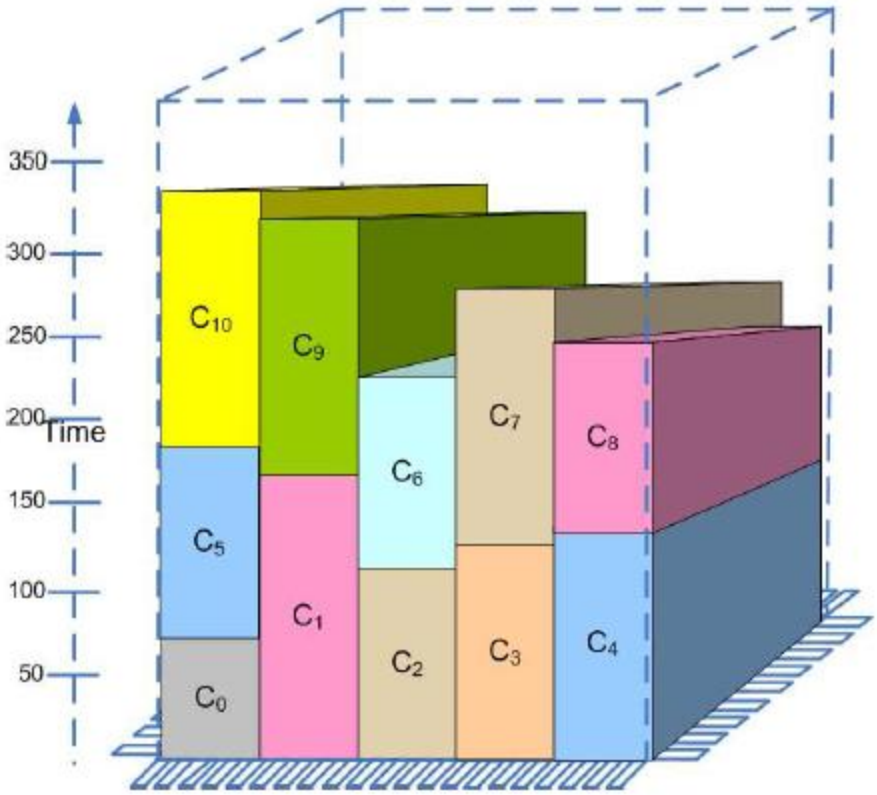

Fig 4: First-fit temporal placement of a set of clusters

The first-fit and best-fit approaches provide fast method to compute a temporal placement solution. However, no effort is spent on the efficiency of the space management. Integer linear programming can be used as exact method to solve the temporal placement problem. In this case, a set of constraint equations that must be fulfilled by each solution must be formulated. Solving the equations will then provide a solution to the temporal placement problem. Because the computation of a ILP-solution is computational intensive, integer linear programming is usually suitable only for small size problems.