electronics

MDPI

Article

Dynamic, Energy-Aware Routing in NoC with Hardware Support [†]

Lluís Ribas-Xirgo [1,*,‡,‖] and Antoni Portero [2,*,§,‖]

[1] Department of Microelectronics and Electronic Systems, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
[2] Barcelona Supercomputing Center, 08034 Barcelona, Spain
[*] Correspondence: lluis.ribas@uab.cat (L.R.-X.); antoni.portero@bsc.es (A.P.)
[†] This article contains adapted texts and figures from a paper entitled "How to implement the Hungarian algorithm into hardware", which was presented at the 39th Conference on Design of Circuits and Integrated Circuits (DCIS), Catania, Italy, 13–15 November 2024.
[‡] Current address: School of Engineering, Campus UAB, 08193 Bellaterra, Spain.
[§] Current address: Plaça d'Eusebi Güell, 1-3, Les Corts, 08034 Barcelona, Spain.
[‖] These authors contributed equally to this work.

Abstract

The Network-on-Chip applications' performance and efficiency depend on task allocation and message routing, which are complex problems. The existing solutions assign priorities to messages in order to regulate their transmission. Unfortunately, this message classification can lead to routings that block the best global solution. In this work, we propose to use the Hungarian algorithm to dynamically route messages with the minimal cost, i.e., minimizing the communication times while consuming the least energy possible. To meet the real-time constraints coming from requiring results at each flit transmission, we also suggest a hardware version of it, which reduces the processing time by an average of 42.5% with respect to its software implementation.

Keywords: dynamic NoC routing; hardware Hungarian algorithm; optimal message routing

check for updates

Academic Editor: Costas Psychalinos

Received: 2 June 2025
Revised: 15 July 2025
Accepted: 15 July 2025
Published: 17 July 2025

Citation: Ribas-Xirgo, L.; Portero, A. Dynamic, Energy-Aware Routing in NoC with Hardware Support. *Electronics* **2025**, *14*, 2860. https://doi.org/10.3390/electronics14142860

1. Introduction

A Network-on-Chip (NoC) application is a collection of tasks that run in different processing elements (PEs) and communicate with each other as defined by a directed acyclic graph (DAG).

As PEs increase in computational power, so does the ability of NoCs to execute complex applications, but assigning PEs to application tasks and routing messages across communication networks efficiently has also become complex. There are works [1–4] that propose different solutions for dynamically mapping tasks to PEs and assigning them priority in a way that reduces energy consumption and contention in communications. These proposals route message transmissions according to previously assigned priorities. However, these route assignments may not be optimal because they are made following the priority order of the tasks and miss the opportunity to find better assignments with some permutations of tasks in the order. In this work, we propose a solution to this problem and several strategies to minimize the time to generate message routes in cases where these routes are controlled by a single network manager.

Once PEs are allocated to tasks, messages over the network must be routed so that they can communicate with each other. The transmission of data requires an amount of energy proportional to the distance of the PEs where the sending and receiving tasks are.

The network can handle several message transmissions at the same time if they use different nodes, and the transmissions can also go over different routes. The network manager should look for the best routes for the required transmissions at a given time.

In this network, a (communication) *task* is transmitting a message from its source node to its destination, and a *resource*is the route followed by the message packets to reach its destination.

Choosing the best routes to consume the least power possible requires assigning the pending communication tasks at a given instant to the resources that minimize the overall distance of the transmission routes. In other words, minimizing the energy consumption of communications in a NoC requires solving an *assignment problem* each time a new set of tasks arises. This problem consists in assigning communication tasks to resources so that the cost of the assignment C is minimal:

$$C = \underset{b_{i,j} \in \{0,1\}}{\operatorname{argmin}} \sum_{i=1}^{n} \sum_{j=1}^{m} a_{i,j} \cdot b_{i,j} \tag{1}$$

where $a_{i,j}$ is the cost of assigning task $i \in [1, n]$ to resource $j \in [1, m]$; $b_{i,j}$ is 1 if task i is assigned to resource j and 0 otherwise; and a task is a transmission from a source to a destination, according to the DAG of the application.

The assignments must be unique, i.e.,

$$\begin{array}{l} \sum_{i=1}^{n} b_{i,j} = 1 \forall j 1 \leq j \leq m \\ \sum_{j=1}^{m} b_{i,j} = 1 \forall i 1 \leq i \leq n \end{array} \tag{2}$$

In this case, the number of resources m is greater than or equal to the number of tasks n since there is at least one route for each task. The right values for $B = \{b_{i,j}\}$ can be found iteratively by looking for the minimum value in each row of $A = \{a_{i,j}\}$. This greedy approach works fairly well for $m \gg n$, but the results strongly depend on row ordering. However, the Hungarian algorithm (HA) [5] has no dependencies on the order of the rows at the price of solving the same problem in $O(n^3)$.

Like the greedy approach, it looks for the minimum value in each row, which is a $O(n^2)$ procedure, but it keeps track of the assignments in case some selection conflicts with a previous assignment. In this situation, the algorithm traces back the previous assignments to make a minimum-cost increase change that resolves the detected conflict. Fortunately, with $m \gg n$, the probability of the occurrence of these situations diminishes, and the complexity becomes closer to that of the greedy approach.

For example, an application with the DAG shown in Figure 1 (left) has nine edges, and thus nine possible communication tasks. The application tasks can be placed at a 3×3 NoC as shown in Figure 1 (right), which allows them to communicate in a hop. In this example, we shall assume that the NoC uses a wormhole type of communication, that the PE network interfaces have dual ports to connect to the horizontal and the vertical lines that are close by, and that the routers configure the corresponding crossbars in accordance with the communication tasks' paths. Note that the entire horizontal or vertical bus lines are occupied for the duration of the transmissions, regardless of which sections are actually used.

For this example, there are 28 different paths that can be grouped into 13 resources, each being a set of paths that represent the possible routes that use some of the lines that also use the longest in the set.

At some given moment, the application requires communication tasks $T_1 \to T_4$, $T_3 \to T_5$, $T_4 \to T_7$, $T_5 \to T_8$ and $T_6 \to T_8$; thus, the network manager must map them to appropriate routes to minimize bus usage.

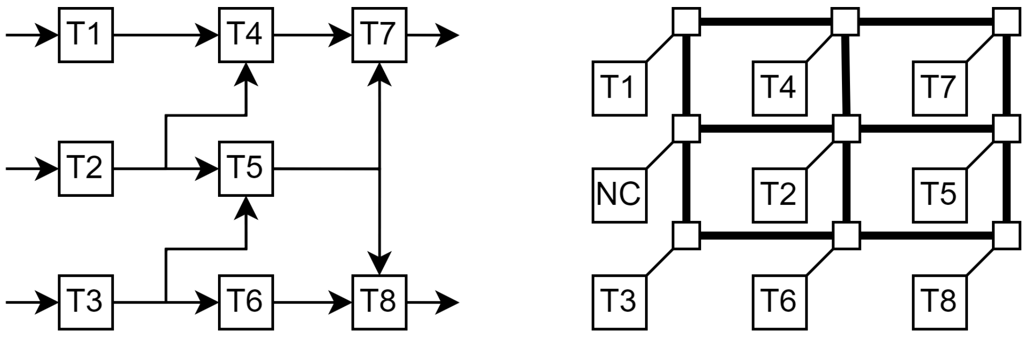

Figure 1. A DAG of an application and its mapping into a 3 × 3 NoC.

In a conventional approach, a minimum cost path (set of lines) is found for the highest-priority task, and the process is repeated until all tasks are assigned a path or a waiting state. In this work, we propose using a task-resource assignment that avoids the sequential calculation of minimum individual-cost routes to obtain a minimum global cost.

Table 1 shows the number of lines, that is, the cost, that are used to perform each of these tasks with the available resources (only the significant ones are shown). If a given resource does not contain a communication path for a given task, the cost is set to the maximum value (the number of lines) plus one (seven, in this case). The assignment problem is thus to bind these communication tasks to resources such that the sum of the cost is minimal. The bindings with cost equal to 7 imply that the corresponding task will remain pending for a further assignment.

Table 1. Task-resource table.

	R_1	R_2	R_3	R_4	R_5	⋯	R_9
$T_1 \to T_4$	3	3	7	7	$1^{\dagger,\ddagger}$	⋯	7
$T_3 \to T_5$	$2^{\dagger,\ddagger}$	7	7	2	2	⋯	2
$T_4 \to T_7$	7	7	$7^{\ddagger}$	$3^{\dagger}$	7	⋯	7
$T_5 \to T_8$	7	$7^{\dagger}$	7	$1^{\ddagger}$	7	⋯	7
$T_6 \to T_8$	7	$1^{\ddagger}$	3	1	7	⋯	$1^{\dagger}$

† The greedy approach takes the first available minimum value. ‡ The Hungarian algorithm proceeds the same way but looks for alternatives to minimize the sum of costs.

The greedy approach and the HA assign tasks $T_1 \to T_4$ and $T_3 \to T_5$ to the same resources, but they differ in the rest of the assignments. While the greedy algorithm assigns R_4 to $T_4 \to T_7$, the HA puts the task on hold by making an assignment that costs seven to R_3. In doing so, the HA can assign $T_5 \to T_8$ to R_4, which saves two lines for all communications. In effect, the transmission power required for these communications is proportional to $1 + 2 + 3 + 7 + 1 = 14$ in the case of using the greedy algorithm and to $1 + 2 + 7 + 1 + 1 = 12$ when using the HA.

Obviously, running the greedy procedure on all row permutations would lead to finding the minimum at the cost of exponential time complexity, which makes the HA better.

In this work, we shall explore how the HA can be used by the network manager to dynamically route communication tasks so that they are made as efficient as possible. In this case, efficiency is measured in terms of the number of lines used per communication frame and the number of waits.

Because pending communication tasks vary over time, the matrix A varies and so does the assignment matrix $B = \{b_{i,j}\}$. However, the calculation of A for all possible tasks can be performed offline (Section 2), and only the assignment problem (submatrix of A with the rows corresponding to the pending tasks) must be solved in real time.

Calculating B might require parallel implementations or hardware accelerators. In the last case, a customized processor can run a program or specific hardware can be used.

In our case, we shall build specific hardware on FPGA from a state machine version of the HA (Section 3). Indeed, modeling the HA with state machines enables the early verification of the system and makes the generation of hardware description straightforward.

The resulting model can be synthesized on a Field-Programmable System-on-Chip (FPSoC) and further refined to cut constants in execution time so as to suit stringent real-time requirements.

The paper is organized as follows. Section 2 discusses the dynamic routing problem and how the process can be separated into two phases so that the first is executed only once to compute the resource sets and the matrix A, and the second is executed online with the application without having to compute either paths or costs. Section 3 presents how to generate a hardware version of the HA so that it can be used in conjunction with the network manager to route messages. Since HA is essentially sequential, the hardware version does not improve the software in terms of power consumption, but we use the flexibility of the hardware to organize the memory to obtain a significant reduction in execution time. Finally, the concluding section summarizes the work performed and how it can complement other solutions to reduce execution time and energy consumption, too.

2. Dynamic NoC Routing

As communications between tasks in an application depend on data, it is not possible to efficiently schedule data transmissions offline; thus, some network manager must run online to assign routes to data packets traveling the communication infrastructure.

After a short review of the literature, we shall explain our two-step approach with an offline PE placement and computation of the resource sets and cost matrix A, and an online message routing process built on top of the HA.

2.1. State of the Art

Networks-on-Chip (NoCs) have emerged as a scalable and flexible solution to overcome the limitations of bus-based communication in System-on-Chip (SoC) designs [6]. Benini and Micheli introduced probabilistic and structured design principles that applied the foundation for NoC development.

Subsequent research on domain-specific NoC optimizations led to influential architectural advancements. A comprehensive survey from Bjerregaard et al. [7] exists, categorizing NoC routing algorithms, performance evaluation methods, and topological types. Meanwhile, Sahu and Chattopadhyay [8] proposed application-aware mapping techniques, emphasizing the impact of task placement on design latency and energy efficiency.

Routing strategies have received much interest and can typically be separated into the following categories:

(1) Deterministic and Adaptive Routing: Conventional methods such as XY and west-first routings offer low-complexity, deadlock-free solutions. Hybrid adaptive schemes extend these methods to manage congestion while preserving correctness guarantees [9].
(2) AI-Assisted Routing and Machine Learning:Reinforcement learning (RL)-based approaches [10] dynamically adapt routing decisions to handle faults and workload variations, especially in 3D NoCs. However, these methods incur additional hardware complexity and inference latency.
(3) Partitioned and Optical NoCs: Partition-aware routing and hierarchical methods, including Dijkstra-based algorithms [11,12], improve energy efficiency and congestion handling in optical and large-scale NoCs [13].

(4) Software-defined NoCs (SDNoCs): SDNoCs introduce a centralized control plane for dynamic reconfiguration [14,15], improving flexibility in heterogeneous systems but often at the cost of additional control plane latency.

The network controller of our approach would use a similar scheme, as tasks must send the requests for transmissions and wait for the communications' configuration.

In contrast to existing routing strategies, our work explores how a well-known assignment problem solver, such as the HA, enables dynamic, energy-aware routing in NoCs. Although HA is traditionally used to solve assignment problems in other areas, its potential for real-time routing in NoC systems has not been explored. Unlike RL-based solutions, our method provides guaranteed optimal communication cost assignmentswith a predictable execution time, making it suitable for FPGA-based NoCs with stringent real-time constraints.

This work introduces the first complete integration of the HA into a hardware-accelerated NoC environment, where it operates in conjunction with a PE and dynamic routing infrastructure.

(a) HA is embedded within a hardware-supported framework and directly interfaces with the task assignment and path allocation subsystems via the NoC.
(b) The algorithm is implemented in VHDL (VHSIC hardware description language), synthesized as a comprehensive extended finite state machine (EFSM), and evaluated in multiple memory organization models.
(c) We prove the impact on task-wait ratios and path contention by comparing HA-based routing to greedy heuristics under diverse workloads.
(d) FPGA-based emulation results significantly improve the execution time and communication efficiency over a CPU-based software variant.

2.2. Offline Processes

Each node of the application's DAG to be implemented in a NoC must be mapped onto a PE such that data transmission is performed with the least energy consumption and data contention possible. For this, the edges of the DAG must correspond to the shortest communication paths possible. Therefore, neighbor nodes should be placed in neighbor PEs.

In this work, placement is performed either manually for the example case or automatically by traversing the graph from input nodes to output nodes, level by level, in a zigzag way. Each level is defined by the edge distance to the primary inputs. (It is possible to obtain optimal placements for DAG whose properties have been profiled with a quadratic assignment [8] of DAG nodes to NoC PEs, but even in this case, optimality depends on the variability of communications over time.)

For a given placement, each edge in G can be covered by a set of paths in the NoC. The paths from node s to node d start at a vertical or horizontal bus port from the PE position $(x,y)_s$ and end at $(x,y)_d$.

The generation of these paths is performed by Algorithm 1, which explores all the paths from $(x,y)_s$ to $(x,y)_d$ for every edge (s,d) in G. To do so, it explores in a depth-first search manner the tree of subsequent neighbor positions. If the last position of the current path corresponds to $(x,y)_d$, the path is eventually stored in the path list P. Otherwise, the current path is augmented with the positions of neighbors that do not cause cycles. In this case, a path contains a cycle if the line where the new segment would sit intersects any previous position in the path.

The new paths are stored only if they are not included in the other paths. In case some of the other paths include the new one, they are removed from the list P, and the new one is appended.

Algorithm 1 Path generation.

```
Input: $G, M$                    ▷ $G$ is the DAG and $M$ the mapping of $G$ into the NoC structure
Output: $P$                      ▷ All paths in NoC mapping $M$ for all edges in $G$
  $P \leftarrow \emptyset$
  for $i, j \mid M_{i,j} > 0$ do           ▷ For all PE positions with some task node assigned
    $s \leftarrow M_{i,j}$                                         ▷ Source node $s$
    for $d \in G[s]$ do                                         ▷ For all edges $(s, d)$
      $Q \leftarrow \{\{(i,j)\}\}$          ▷ Stack of temporal paths with initial path starting at $(i, j)$
      $count \leftarrow 0$                              ▷ Counter of paths from $(i, j)$ to node $d$
      while $Q \neq \emptyset$ do
        $path \leftarrow Q$:pop()
        $(x,y) \leftarrow path$:last()                     ▷ $(x, y)$ gets the last point of $path$
        if $M_{x,y} = d$ then                       ▷ If $(x, y)$ is the destination point then...
          $b \leftarrow P - count + 1$                    ▷ ... the new path from node $s$ to $d$ ...
          $included \leftarrow$ false
          while $b \leq P$ and not $included$ do
            if $P_b$:includes($path$) then
              $included \leftarrow$ true
            else if $path$:includes($P_b$) then
              $P$:remove($b$)
              $count \leftarrow count - 1$
            else
              $b \leftarrow b + 1$
            end if
          end while
          if not $included$ then                    ▷ ... is inserted into $P$ if it was not there
            $P$:append($path$)
            $count \leftarrow count + 1$
          end if
        else                                   ▷ Otherwise, the path must be augmented...
          for $(x', y') \in M$:neighbors$(x, y)$ do                ▷ ... with neighbors of $(x, y)$
            if not $path$:cycle($(x', y')$) then
              $Q$:push($path + \{(x', y')\}$)
            end if
          end for
        end if
      end while
    end for
  end for
```

The generation of paths from $(x,y)_s$ to $(x,y)_d$ can be improved by dynamic programming, as the set of paths between two NoC nodes $(x,y)_{s'}$ and $(x,y)_{d'}$ is the same as that of $(x,y)_s$ to $(x,y)_d$ if $(x_d - x_s, y_d - y_s) = (x_{d'} - x_{s'}, y_{d'} - y_{s'})$. The relative positions of the two points must be taken into account when translating one path from one set to another, and all the paths that would use segments outside the NoC must be removed from the final sets.

However, the generated paths have different lengths, and some of them can include sequences of nodes that match other shorter paths for different source and destination points. In these cases, these paths are packed into a single resource. They are incompatible among them, so they can share the same column.

After the generation of the minimum paths, Algorithm 2 packs them into groups that form the resources. This algorithm starts by sorting the paths according to the number of lines they use and, in descending length order, proceeds to create a set S with the current path and any other path included in it. In the end, the new set is appended to the resource list R.

The lists C (for counter) and B (for boundary) store the number of resources to which paths belong and the maximum number of appearances, respectively. Any time a path is inserted into a resource, the corresponding counter in C increases. If any two paths within the same resource are compatible, i.e., they share no communications lines, they must appear in another set; thus, their occurrence maximums in B increase.

Algorithm 2 Resource generation (path packing).

```
Input: $P, M$                                  ▷ $P$ is the list of paths on node mapping $M$
Output: $R$                                    ▷ $R$ is the list of resources, i.e., sets of paths
  $R \leftarrow \emptyset$
  $C \leftarrow \{0, \ldots, 0\}$                ▷ $C$ counts appearances of paths in $R$
  $B \leftarrow \{1, \ldots, 1\}$                ▷ $B$ is the boundary of appearances of paths in $R$
  $P \leftarrow P$:mergeSort()                   ▷ Sort paths in $P$ in descending-length order
  for $i \in [1, P]$ do
    if $C_i < B_i$ then
      $S \leftarrow \{i\}$                       ▷ Path $i$ added to new resource
      $C_i \leftarrow C_i + 1$
      for $j \in [i+1, P]$ do
        if $P_i$:includes($P_j$) then            ▷ $P_i$ is longer than $P_j$
          $S \leftarrow S \cup \{j\}$
          $C_j \leftarrow C_j + 1$
        end if
      end for
      for $a \in [2, S-1]$ do                    ▷ Check for compatibilities among included paths
        $c \leftarrow 0$
        for $b \in [a+1, S]$ do
          if cmpPaths($P_{S_a}, P_{S_b}$) $= 0$ then   ▷ No lines in common
            $c \leftarrow c + 1$
          end if
          if $B_{S_a} \leq c$ then
            $B_{S_a} \leftarrow c + 1$
          end if
        end for
      end for
      $R$:append($S$)
    end if
  end for
```

The calculation of the cost matrix A is quite straightforward. Each position (i, j) is set to the number of lines used by a path serving the connection of the edge i in R_j or to the maximum number of lines plus one if there is no path in R_j, with the origin and end points corresponding to E_i.

2.3. Online Process

The assignment of resources to tasks must be performed when new pending communications tasks require it. We assume that computing tasks send requests to the network manager in specific time frames or *cycles.*

In each cycle, some communication tasks, or edges of G, must be assigned to resources from R_i such that the sum of costs is minimal.

A straightforward approach is to pair each edge E_i with the resource that serves it at the minimum cost. Once paired, the resources are blocked for further assignments.

This behavior is represented in Algorithm 3, with a nested *for* to look for the assignment to E_i (agent i in the algorithm) with the minimum cost μ among the resources that have not previously been assigned, that is, $P_j = 0$, as vector P contains the row to which a resource is assigned or 0 if not.

This greedy way of proceeding works fine when the number of resources exceeds, by a large amount, that of the edges to be assigned. Unfortunately, blocking some assignments might lead to suboptimal solutions and, thus, it is much better to do so with the HA.

Algorithm 3 The greedy algorithm.

```
Input: $A$                                  ▷ $A_{i,j} \mid i \in [1,n], j \in [1,m], n \leq m$
Output: $B, c$      ▷ $B$ binds action $B_i$ to resource $j \mid_{0 \leq j \leq m}$ and $c$ is the total cost
  $B \leftarrow \{0, \overset{n+1}{\ldots}, 0\}$                              ▷ $B_i = 0$ means unassigned
  $c \leftarrow 0$
  $P \leftarrow \{0, \overset{m+1}{\ldots}, 0\}$
  for $i \in [1,n]$ do                                  ▷ for each row (agent) do...
    $\mu, k \leftarrow \infty, 0$
    for $j \in [0,m]$ do                             ▷ for each column (resource) do...
      if $P_j = 0$ and $A_{i,j} < \mu$ then $\mu, k \leftarrow A_{i,j}, j$ end if
    end for
    if $k > 0$ then $P_k \leftarrow i$ end if
    $c \leftarrow c + \mu$
  end for
  for $j \in [1,m]$ do
    if $P_j \neq 0$ then $B_{P_j} \leftarrow j$ end if
  end for
```

In fact, both algorithms perform equally well if the sequential selection of minimum task-resource values leads to final solutions. In the HA (see Algorithm 4) there is a *for* loop for the edges to be assigned and another for the resources. The latter is inside a *repeat* loop that searches for the best possible assignment.

This version of the algorithm was extracted from a program by Andrey Lopatin's [16], which is one of the most compact implementations of HA in the literature and was later implemented in Lua [17].

In contrast with other HA versions, this one does not modify the cost matrix A and uses auxiliary vectors to account for row and column offsets (U and V, respectively), the indices of rows with which columns (P) are paired, the preceding elements in the current decision-taking step (W), the minimum costs per column (L), and a Boolean value to know whether a column is already paired (T). In this case, the zero positions of some vectors are used as control values for the program, and B to store the indices of the columns paired to each row.

Pairing an edge E_i of G with a resource R_j implies that no other path within can be assigned to another edge, i.e., there can be only one 1 per column in the pairing matrix B. Unfortunately, there is no guarantee that the corresponding path is compatible with another assigned path in a different column and row. Therefore, the solutions of greedy and Hungarian algorithms must be checked for compatibility. In case some assignments are incompatible because the corresponding paths in the affected resources share lines, the associated cost is set to the maximum for one of the task-resource bounds, and the assignment procedure is repeated. The loop continues until the result contains no incompatible assignments.

The complexity of the assignment problem for different NoC sizes is illustrated in Table 2, which shows the averages of several runs. In all of the cases, all the PEs of the NoC are assigned to some node of a random graph with average node fanout of 2. Each row of the table corresponds to averages of at least 25 assignment cycles in 10 or more different random DAGs (i.e., averages of 250 runs or more). The number of paths and resources (column "no. of rsrcs.") increases with the size of the NoC, although this growth is affected by the fact that local connections also have global effects, as they use all communication bus lines.

Algorithm 4 The Hungarian algorithm.

```
Input: $A$                                          ▷ $A_{i,j} \mid i \in [1,n], j \in [1,m], n \leq m$
Output: $B, -V_0$        ▷ $B$ binds action $B_i$ to resource $j \mid_{0 \leq j \leq m}$ and $-V_0$ is the total cost
  $U, B, V, P, W \leftarrow \{0, \overset{n+1}{\dots}, 0\}, \{0, \overset{n+1}{\dots}, 0\}\{0, \overset{m+1}{\dots}, 0\}, \{0, \overset{m+1}{\dots}, 0\}, \{0, \overset{m+1}{\dots}, 0\}$
  for $i \in [1,n]$ do                              ▷ for each row (agent) do...
    $P_0, j0 \leftarrow i, 0$
    $T, L \leftarrow \{$false$, \overset{n+1}{\dots},$ false$\}, \{\infty, \overset{n+1}{\dots}, \infty\}$
    repeat                                          ▷ repeat until task assigned
      $T_{j0}, i0, \delta, j1 \leftarrow$ true$, P_{j0}, \infty, 0$
      for $j \in [1,m]$ do                          ▷ for each column (resource) do...
        if not $T_j$ then                           ▷ if not assigned then...
          $c \leftarrow A_{i0,j} - U_{i0} - V_j$
          if $c < L_j$ then $L_j, W_j \leftarrow c, j0$ end if
          if $L_j < \delta$ then $\delta, j1 \leftarrow L_j, j$ end if
        end if
      end for
      for $j \in [0,m]$ do
        if $T_j$ then
          $U_{P_j}, V_j \leftarrow U_{P_j} + \delta, V_j - \delta$
        else
          $L_j \leftarrow L_j - \delta$
        end if
      end for
      $j0 \leftarrow j1$
    until $P_{j1} = 0$
    repeat
      $j1, P_{j0} \leftarrow W_{j0}, P_{W_{j0}}$
      $j0 \leftarrow j1$
    until $j1 = 0$
  end for
  for $j \in [1,m]$ do
    if $P_j \neq 0$ then $B_{P_j} \leftarrow j$ end if
  end for
```

The probability of a communication task being requested at any time is set to $\frac{1}{16}$; thus, the average number of requests per assignment cycle (column "no. of reqs.") is relatively low, though it grows with the size of the NoC.

As expected, the greedy approach and the Hungarian algorithm perform equally well with a slight advantage for the latter. It is worth noting that these are the results after solving conflicts among task-resource assignments. Because this conflict-solving procedure has exponential complexity, simpler strategies must be adopted.

We tried several options to see which one let the assignment procedures reach the best values, including selecting the first one or the one with the least cost, but none is as effective as choosing the one with the highest conflict count. This eventually generates suboptimal solutions. For the example in the table, the percentages of cases where the assignment procedure is repeated (column "%iter.") grow with the size of the NoC and so does the number of iterations (column "#iter.") to reach a fully compatible assignment of resources, typically with some waits.

Table 2. Simulation of dynamic message routing.

NoC Size	No. Paths	No. Rsrcs.	No. Reqs.	Greedy Algorithm [†] Cost [‡]	Waits	%Iter.	#Iter.	HA [†] Cost [‡]	Waits	%Iter.	#Iter.
4 × 4	109.0	41.4	1.9	3.28	0.08	**11.6**	16.41	**3.14**	**0.06**	**11.6**	**14.21**
5 × 5	217.5	80.4	2.7	5.72	0.20	**21.6**	24.98	**5.52**	**0.18**	**21.6**	**24.15**
6 × 6	362.8	126.3	3.3	8.76	0.33	**32.8**	33.09	**8.23**	**0.29**	33.6	**29.08**
7 × 7	573.8	174.9	4.6	14.52	0.57	**50.0**	42.90	**14.09**	**0.54**	**50.0**	**40.30**
8 × 8	888.8	245.5	6.0	22.74	**0.88**	**61.6**	69.16	**22.04**	**0.88**	62.0	**67.50**

[†] The numbers in bold are the smallest between the equivalent columns of the two algorithms. [‡] Cost includes #waits × (size + size) + 1.

Note that there is a cumulative effect of waits (i.e., unassigned tasks that remain for the next cycle), which are not considered here. For the cases in the table, the percentages of unattended tasks go from 3.4% to 18.7%, again with the HA option being the best. However, in a real case, the sparsity of communication needs will probably give enough free cycles to absorb the pending transmissions of a set of requests.

The same cases are simulated with different probabilities of occurrence of the communication tasks requests. As expected, the differences among the two methods reduce for probabilities lower than $\frac{1}{16}$ and increase for the higher ones. In fact, for $\frac{1}{8}$, the HA outperforms the greedy algorithm by percentages ranging from 5% to 25% in all cases and factors.

To see how much it can improve communication performance in real cases, the two algorithms are compared using the realistic traffic benchmark suite called MCSL [18]. This benchmark contains the traces of the simulated execution of eight real cases on NoC with various dimensions and topologies, namely, fat tree, torus, and mesh, which is the one used to simulate the assignments.

The recorded traffic pattern files contain data on the execution of tasks (location in the NoC, sequence number in the execution schedule, and execution time in the number of cycles) and on the communications (source and destination tasks, memory addresses, and data size). From this information, it is extracted which data transmissions must be made in each execution of the network manager. Unlike the random application graphs, which occupy all the available PEs of the NoC, the mapping that real applications have concentrates the tasks in a part of the PEs, which leaves more degrees of freedom for message routing, particularly in the bigger NoC.

Table 3 shows the average assignment costs for each application and NoC size. In this case, we choose not to discard the initial part, which is where the highest costs are, nor the final part of the executions, which is where the costs are usually lower. As expected, the smaller the NoC, the more difficult it is to find routes for all simultaneous requests, and the Hungarian algorithm performs better than the greedy one. It can be seen that, as the NoC dimension increases, there are more degrees of freedom, and the difference between the algorithms disappears. In fact, the number of zeros in the "% gain" columns increases as the NoC dimension increases. This percentage is calculated taking the average cost of the greedy algorithm as 100% and the HA in relation to it; thus, the more negative, the better for the HA. The best case shown in the table is for the "RS-32_enc" and the 2 × 2 NoC, where the average cost of the HA is only 76.81% of that of the greedy algorithm. (Note that costs increase with size because the cost of waits depends on 2 × size + 1, and actual gains on transmissions are slightly better than shown.)

In summary, in cases where there are many degrees of freedom because, for example, not all the PE of a NoC are used, the greedy algorithm works just as well as the Hungarian one and, in fact, would coincide with what a conventional network manager would do. In these cases, it is necessary to assess whether it is viable to run the offline processes since they involve calculating all possible paths to cover all the arcs of the application graph. If it is, the proposal made in this work greatly reduces the complexity of resource allocation to tasks during the execution of applications, since the minimum paths are calculated beforehand.

Table 3. Average cost [†] comparison between the greedy and Hungarian algorithms.

NoC Size	**2 × 2**			**3 × 3**			**4 × 4**			**5 × 5**		
Application	**Gry.**	**HA**	**%Gain**	**Gry.**	**HA**	**%Gain**	**Gry.**	**HA**	**%Gain**	**Gry.**	**HA**	**%Gain**
Robot	7.67	7.52	−1.96	22.42	21.92	−2.23	36.82	36.66	−0.43	46.10	46.05	−0.11
Sparse	3.00	2.87	−4.33	13.82	13.82	0.00	19.77	19.11	−3.34	33.25	33.25	0.00
RS-32_dec	6.10	5.62	−7.87	44.04	44.04	0.00	77.81	77.81	0.00	175.17	175.17	0.00
RS-32_enc	3.32	2.55	−23.19	10.23	10.23	0.00	12.58	12.31	−2.15	26.83	26.83	0.00
Fpppp	4.24	3.53	−16.75	86.56	86.56	0.00	229.27	229.27	0.00	398.67	398.67	0.00
H264-720p	2.30	2.28	−0.87	20.60	20.56	−0.19	59.00	58.98	−0.03	148.44	148.44	0.00
H264-1080p	3.29	3.27	−0.61	22.23	22.04	−0.85	60.91	60.90	−0.02	149.07	148.69	−0.25
FFT-1024	2.45	2.45	0.00	32.70	32.64	−0.18	53.60	53.58	−0.04	204.76	204.75	0.00

[†] Cost includes #waits × (size + size) + 1.

In cases with few degrees of freedom, the network manager should use the iterative conflict-solving approach with HA to maximize the use of communication resources and minimize energy consumption and waits.

3. Hardware Synthesis of the Hungarian Algorithm

As the Hungarian algorithm is the key procedure for obtaining optimal assignments, it must be run as fast as possible to enable the real-time operation of the network manager when assigning routes to messages.

To speed up HA execution, there are parallel implementations in CUDA [19] that take advantage of concurrently augmenting several paths (the equivalent to solving conflicts in the greedy algorithm) but none for FPGA, where it can benefit from executing one instruction per cycle.

We use automatic hardware synthesis tools to obtain a hardware implementation of the HA from a state-based version of it. The state machine model is easily obtained from the program, and the corresponding description can be synthesized to hardware straightforwardly. For this, we use our own state machine coding pattern and simulators to verify and then synthesize the HA.

3.1. Extended Finite State Machines

An EFSM is a finite-state machine whose state is extended with other variables that remember data from one instant to the next. For example, a timer that emits a signal e every time a certain period P elapses since it receives a start signal b (see Figure 2, left) would have a computational model represented by a state machine (Figure 2, right) with two main states (WAIT and COUNT) and an additional variable C that stores the number of pending cycles in the COUNT state and which constitutes the extension to its main state. The input signal b and the output e are events that are only present or active (i.e., *true* or 1) for a specific instant, while the input signal P contains a value that is continuously present, i.e., at each instant, although its value is only considered at the same instant when $b = true$.

It must be kept in mind that, from a functional point of view, the duration of these instants is not defined, and, in fact, it can be 0, fixed (unit delay) or variable. In the case of the example, it must be known, constant, and linked to a specific period which, in a hardware implementation, would be the period of the cycles of the base clock (or some exact multiple).

To avoid the problem of circular dependencies in the same instant, only Moore-type machines are allowed. In a Mealy machine, a circular dependence occurs when an output signal is connected in an immediate manner to an input signal. Typically, this can occur when multiple state machines whose outputs do not depend exclusively on their extended state are combined.

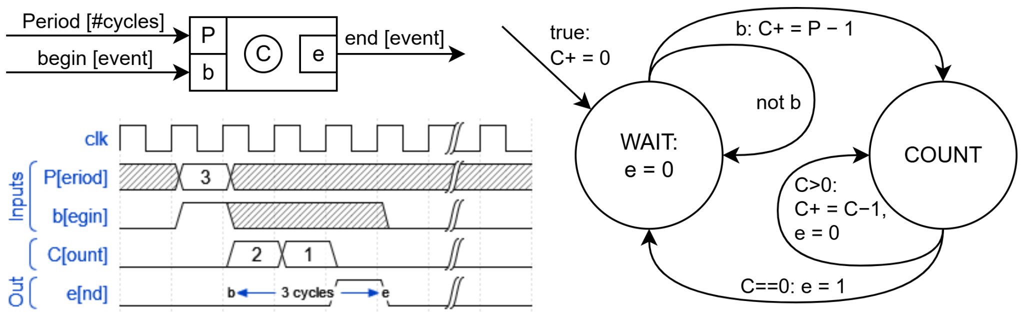

Figure 2. Computational model of a timer controlled by a signal b that indicates the instant, or clock cycle, from which it begins to count P transitions and, with the last one, emits a positive pulse through the output signal e.

Despite the underlying model being Moore, it is possible to benefit from the advantages that Mealy machines provide by linking actions to transitions. However, as can be seen in the transitions from the WAIT state, the immediate assignment (that is, made at the beginning of the same state) to the output signal e depends exclusively on the extended state, that is, the main state and C. Therefore, it is a Moore machine (the outputs depend only on the state), but with the advantage of being able to express changes in the outputs depending on conditions of the state variables as if it were a Mealy machine. In the example, the output e does not depend on any input signal, only on the state variable C.

EFSMs can include decision trees to represent complex transitions from one state (the root of the tree) to the next (the leaves of the tree), can be combined in parallel, and can share state variables, that is, memory.

Hierarchy is implemented by combining several state machines in parallel that communicate through a protocol in which a master and a slave are defined.

The verification of this type of machine must include not only confirming that they are Moore but that there are no unwanted final states, that all states can be reached from the initial one, and that the conditions of the output arcs of the nodes that represent the states cover all cases and do not overlap with each other.

To illustrate how easy it can be for failures to go unnoticed, an error of this nature is left in the diagram in Figure 2: The state machine is correct, with P and C being integer values and $P > 0$. With values of $P \leq 0$, the conditions of the COUNT output arcs do not cover the $C < 0$ case.

Although these basic properties of the model can be checked for each EFSM, maybe after homomorphic transformation of some expressions and variable values limited to low-cardinality sets, they must be simulated to verify that they behave as expected, at least for the tested cases.

3.1.1. Software Simulation of EFSM

EFSM networks can be simulated in SW. In this case, EFSM diagrams are translated into program objects whose behavior is simulated by a simulation engine.

In our case, we use C/C++, Python or Lua, and translation is performed by matching the EFSM elements with programming pattern elements [20]. (For this work, only Lua was used.) With this programming paradigm, the execution of the program is regulated like that of a state machine, and, to do so, it must include state variables whose value is used to determine what code must be executed in each cycle [21] and a main loop or superloop [22] that is responsible for repeating cycles and acts, in fact, as its execution engine.

Each state machine is constituted as an object of a class that has its space for data and methods for its initialization, the reading of input data, the calculation of values of the next state, the update of the current state values, and the writing of output data, among others.

The network of EFSM to be executed in parallel is represented by a *system* object, whose class methods call the components' homonymous ones. This object is responsible for the communication among the system's components and holds the common shared data. As for the timer example and the HA case, the system would be made of a single EFSM.

3.1.2. Hardware Description Synthesis of EFSM Networks

Like the software synthesis, the hardware synthesis starts from the behavior graphs of the corresponding EFSM and can be carried out in a few steps in a simple and systematic way.

The hardware description in VHDL that is used separates the computation of the next state and the actions corresponding to the current EFSM state in different concurrent processes [23]. In fact, the code obtained is very similar to that of the software, which also separates the computation of the next extended state from that of the immediate output signals.

This type of organization makes possible optimizations by downstream hardware synthesizers difficult, for which it may be better to integrate these processes into a single algorithm.

In the first case, each process deals with a specific aspect of the state machine. One of them describes the sequential part of the machine and is responsible for generating the necessary registers for states and variables, and for establishing the event that determines the loading of the registers that store the extended state. This process must include an asynchronous reset signal for the state machine.

A second process handles transitions between states, i.e., it computes the value of the next extended state [24]. Since it is a combinational process, it must contain assignments for all possible values of the state and variables. In fact, there should be an initialization block for all of them prior to specific value assignments. This prevents synthesizers from generating memory elements (latches) to hold previous values, which would typically be performed, particularly when using data types suitable for synthesis such as *std_logic*.

The third and last process is responsible for setting the value of the outputs based solely on the extended state (Moore). It could also be performed based on the state and value of the inputs (Mealy), but this might lead to possible combinational feedback when combining different state machines, and therefore we do not use this possibility.

In this case, a VHDL code style is promoted that is easy to generate but can be refined to achieve some optimization that may be necessary according to the requirements of each specific module.

Although it is possible to refine the VHDL code to obtain more compact, faster, or lower consumption versions than the original, we look for a coding style that enables a rapid first implementation.

3.2. State-Based Version of the Hungarian Algorithm

The Algorithm 4 is transformed into an EFSM to make its software and hardware synthesis feasible. Communication between the network manager and the module is memory-mapped through a handshake protocol with signals b (begin) and e (end of HA operation).

The memory contains the input cost matrix A and control data that include the list of edges to be assigned (not shown in Figure 3). The output data are the array P (pairing of a given column) and the first value of the array V (the negative value of the total cost). The memory system also stores auxiliary arrays L, V, W, U, and T.

The main states of the HA EFSM are shown in the state graph on the right of Figure 3, but the extended state variables are not shown for simplicity except for the outputs MA, WE and D, which are used for communication with the memory. In the initial state (IDLE), the EFSM waits for b to be *true* and begins execution by changing to the state NEW[0].

The states NEW[0], NEW[1], and NEW[2] initialize the auxiliary arrays in memory (the first two *for*of function *HA* in Algorithm 4). Once finished, the state machine moves to the REP0[0] and REP[1] states that correspond to the first *repeat* instruction and the previous setup of the extended state variables. This loop includes several *for*instructions and ends in state REP0END. In this state, the EFSM decides to return to REP0[0] or switch to REP1[0].

The REP1[0] is the first state corresponding to the last *repeat* instruction, and REP1[2] is the last state, where it decides whether to continue in the *repeat* loop, begin a new cycle for a new edge (row of A), or end.

State names include an index between square brackets when they have sequences of actions that cannot be parallelized. Typically, memory accesses require two states in sequence: in the first one, the address is supplied to the memory unit, and, in the second one, the read datum is used in calculations and decision-making.

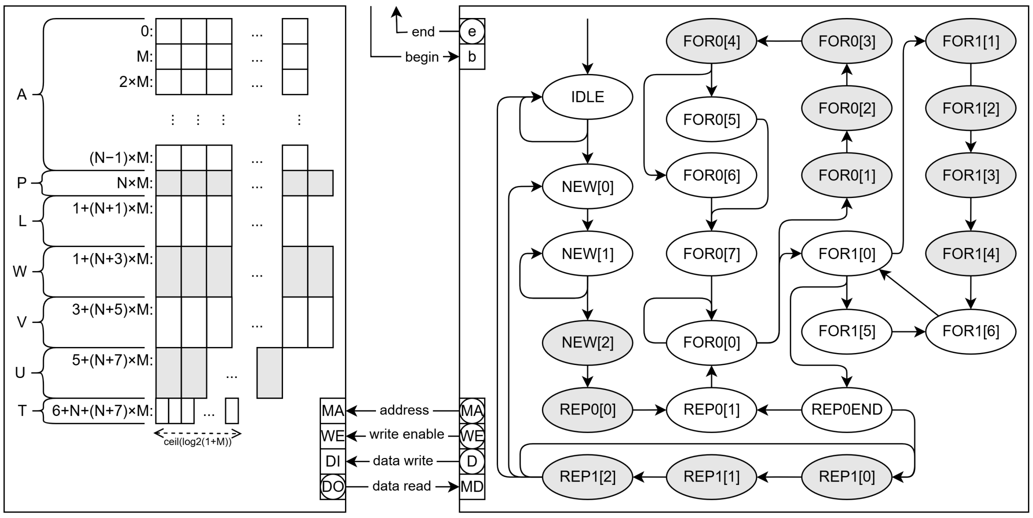

Figure 3. Schematic of the HA EFSM (**right**) and of the memory layout (**left**).

3.2.1. Single-Memory Architecture

A software version of this system, with two EFSM objects (the memory controller and the HA executor), is programmed in Lua for its early verification, and coded into VHDL for its final implementation on an FPGA for its evaluation.

To compare the software version given in Algorithm 4 with the hardware implementation on an FPGA, we tested them under the worst conditions for the HA, which are very dense square A arrays. This way, the differences in processing times reach the order of seconds, and are measurable for the software version. Note that, under actual NoC conditions, A has more resources than tasks to be assigned ($m > n$) and has relatively low density because only a fraction of the resources can carry out a given task (i.e., most of the values in A are the maximum cost plus one). As an example, an average random graph of an 8×8 NoC has 128 edges, 8 tasks per cycle, and 256 resources (values from Table 2 adjusted to powers of 2), with a cost matrix A sized 128×256 and 8×256 for assignments.

Figure 4 shows the averages of 10 executions of the HA program on an Intel 12th gen i5-1235SU 1.3 GHz processor, 16 Gb RAM machine running MS-Windows 11, and on an Altera Cyclone 10 LP FPGA. The hardware block works at 114 MHz and takes 1133 logic elements and 277 registers.

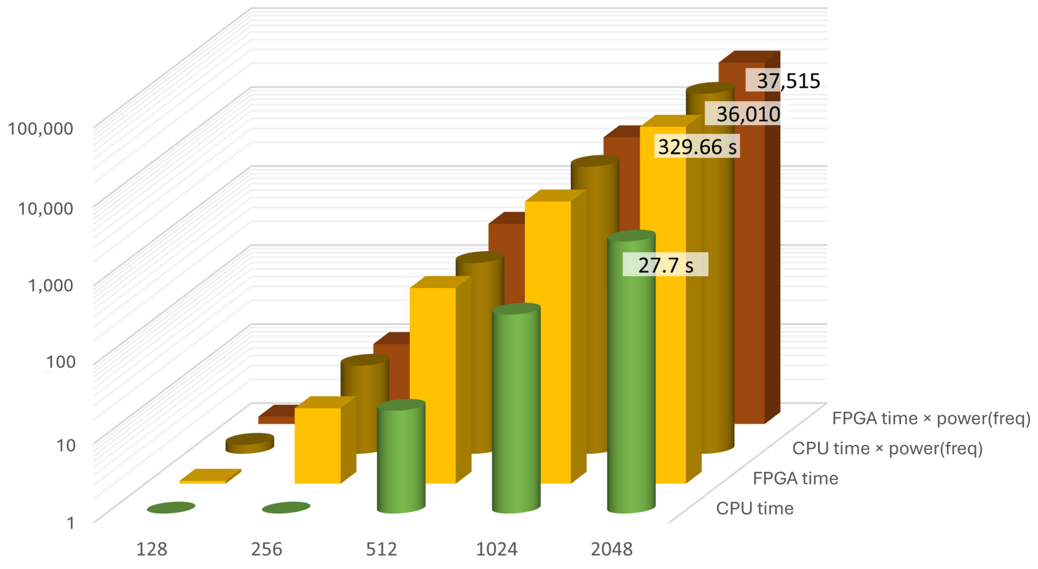

Figure 4. Comparison between software and hardware version of the HA with respect to the problem size (square A length). The z-axis is logarithmic with units adjusted to show differences: times are shown as 1/100 s.

The software version is roughly ten times faster than the FPGA (27.7 s vs. 329.66 s for the 2048 $\times$ 2048 matrix) due to the difference in the clock frequency (1.3 GHz vs. 114 MHz). In Figure 4, the vertical axis represents time units in hundredths of a second so that they have the same scale as the power units. In this case, the power consumption has not been measured experimentally but is a factor calculated theoretically from the fact that it is proportional to the elapsed time, the clock frequency, and the square of the supply voltage. The power consumption factor represented in the figure is the product of the time and clock frequency. As might be expected from two different implementations of the same algorithm, which therefore have the same computational effort, they present a similar value in all the cases tested.

It is interesting to note that the same software run on an Intel processor at 3.2 GHz with 32 Gb RAM takes roughly the same time to execute the testbench, indicating that the bottleneck of execution is communication with the memory. The advantage of using the hardware block is that the memory system can be adapted to minimize this bottleneck.

3.2.2. Other Memory Architectures

The presented hardware version (SPMEM) can be improved by using a double-port memory block and by splitting the single memory block into several blocks.

The double port memory enables two reads in a clock cycle and, thus, requires fewer states to achieve the same functionality (Figure 5). In this case, the memory layout is kept the same, but there are some operations that require more than two data from memory.

As the number of states is reduced (19 with respect to 25 for the single-port RAM version), so it is with the circuitry, which takes up only 783 logic elements and 160 registers on a Cyclone 10 LP. This reduction also enables the double port version (DPMEM) to go 11% faster, operating at a clock frequency of 126.5 MHz (worst case).

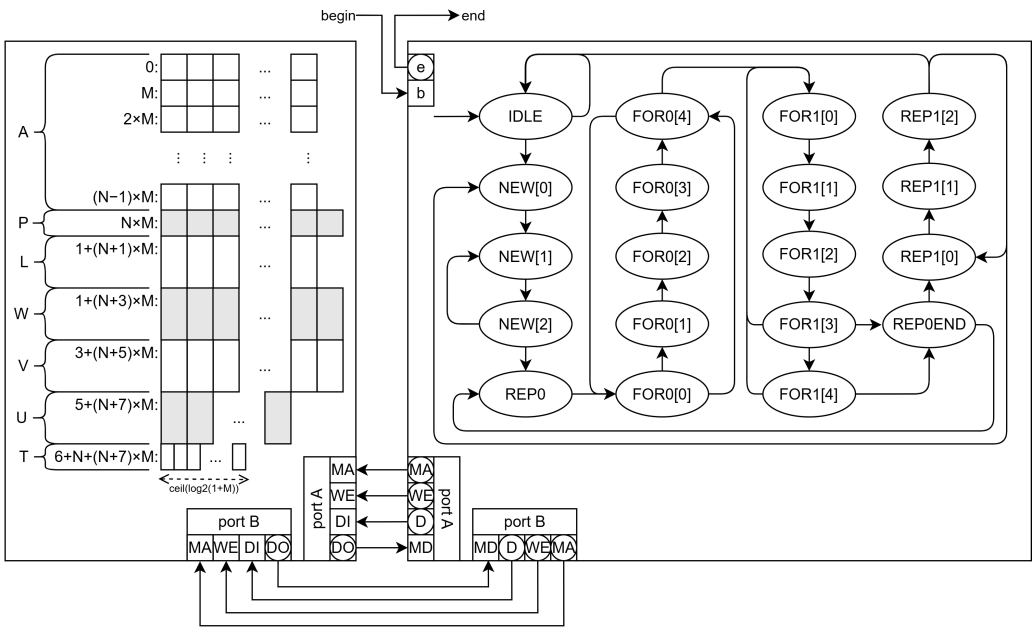

Figure 5. Schematic of the HA EFSM (**right**) with a true double port memory (**left**).

To be able to make an instruction per clock cycle, more memory reads are needed. We built a version with three memory blocks (TMEM, shown in Figure 6 that reduces the number of states (15), as compared to the previous EFSM, and closes the gap between cycles and operations. In this case, the number of logic elements is further reduced to 568 but the number of registers increases up to 180 with respect to the double port version. The circuit is smaller and the worst-case clock frequency (134.9 MHz) is higher than those of the previous versions.

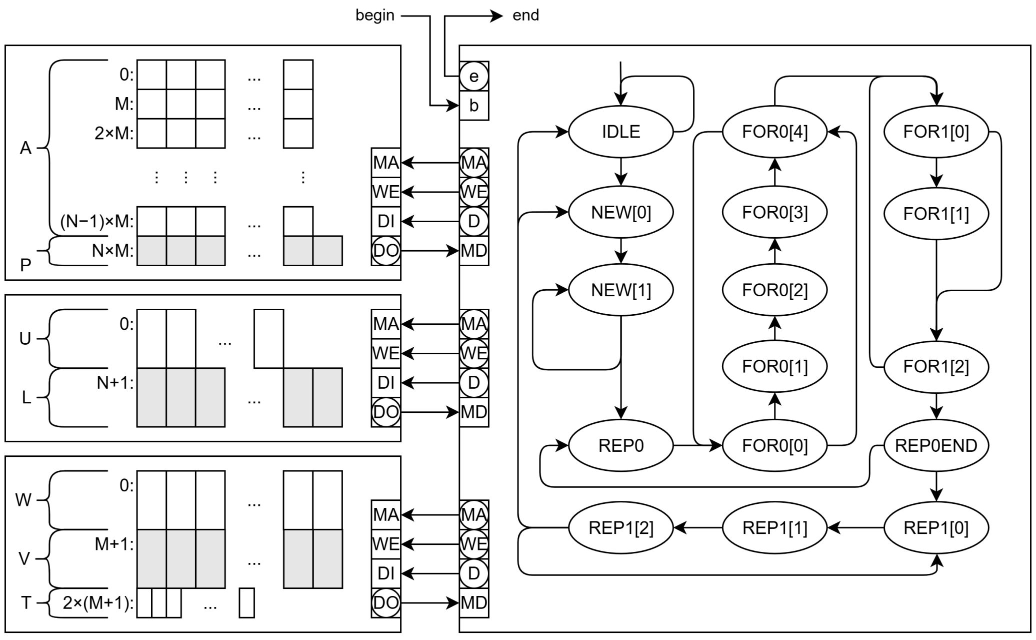

Figure 6. Schematic of the HA EFSM (**right**) with three memory blocks (**left**).

Table 4 shows the averages of the measures of ten executions for ten random DAGs for each NoC size. Each DAG takes up all the PEs available in the corresponding NoC. As expected, the number of clock cycles decreases with the number of parallel reads. Because the circuits are smaller in the DPMEM and TMEM versions, the critical paths are also shorter, and thus the maximum clock frequencies increase.

Table 4. Time complexity of the hardware versions of the HA depending on the size of the NoC.

NoC Size	SPMEM 113.8MHz Clock Cycles	DPMEM 126.5MHz Clock Cycles	DPMEM vs. SPMEM	TMEM 134.9 MHz Clock Cycles	TMEM vs. DPMEM	TMEM vs. SPMEM
8 × 8	1866	1349	−34.96%	1233	−14.30%	−44.26%
16 × 16	8051	5860	−34.52%	5430	−13.11%	−43.10%
32 × 32	43,129	31,620	−34.04%	29,597	−12.23%	−42.11%
64 × 64	229,326	167,246	−34.39%	158,406	−11.19%	−41.73%
128 × 128	1,238,500	902,290	−34.45%	860,260	−10.60%	−41.40%

To some extent, this frequency increase keeps the power consumption the same but at significant time savings.

Note that, regardless of the version, the number of cycles grows proportionally to the cube of the dimension of the NoC side since it corresponds to the time complexity of the HA. However, there is a multiplicative factor that is different for each case (2.21 for SPMEM, 1.62 for DPMEM and 1.53 for TMEM). Thus, the DPMEM version is 34.5% faster than the SPMEM, and the TMEM version reaches 42.5% savings in time. These savings reach an average of 26.7% (DPMEM) and 30.9% (TMEM) for the same clock frequency, which would also imply that these versions operate at a lower power consumption.

Therefore, compared to SPMEM or the software version, the TMEM module uses hardware parallelism to achieve an increase in speed up to ×1.75.

These results show that the NoC implementations in FPSoC may use hardware HA modules to minimize the power consumption and time to route internal messages with respect to performing the same computations in an embedded processor.

4. Conclusions

NoC and artificial intelligence (AI) can benefit each other: AI processes depend on complex systems with intercommunicating tasks that can be implemented in NoC efficiently, while NoC mapping and routing require AI to be efficient, too.

In this work, we have approached the problem of routing messages from a conventional algorithmic perspective. Although the problem belongs to the nondeterministic polynomial-time complexity class (NP), part of the complexity is transferred to *pre-runtime* processes, namely the computation of all paths in the application's DAG and their packing into the so-called resource groups. The computation of all paths is computationally viable because the target NoCs are the wormhole type, which reduces the exploration space significantly.

We have presented a strategy to leave the heavy computation processes (path generation and packing) away from the ones that have to be frequently run (binding communication tasks and resource packs).

Its main advantage is that it is not necessary to calculate the minimum-cost paths during application execution so that optimal assignments can be computed in real time but in exchange for having a prior process that cannot always be carried out.

As the resources are not totally independent from each other, the assignment algorithm is executed iteratively to cope with incompatible pairings. The probability of these pairings to occur increases with the size of the NoC, as well as the average number of iterations per assignment.

The dynamic nature of communications can be handled with frequent updates in the assignment of communication tasks and resources. We have shown that for the best case scenarios, the HA behaves like a greedy algorithm, and that for the worst cases, it outperforms the greedy version by up to 25%.

Therefore, it is clear that the HA is a good alternative to the greedy option for the NoC network managers to assign routes that minimize the overall time and power consumption.

In case the NoCs are implemented in FPSoC, the assignment can be performed by a specific hardware module in order to speed up this process.

We have implemented a compacted software version of the HA and the simulators of the equivalent EFSM versions, which had been verified against the original program and served to verify the functionality of the VHDL versions.

We used the experience of an early hardware implementation of a non-optimized HA [23] to produce an improved version based on the compact software version. To our knowledge, these are the first hardware versions of HA to be released.

Due to the sequential nature of the HA, direct translation of the program to an EFSM with a single memory results in a hardware module that executes the algorithm at the same cost as its software version.

We have developed two versions with different memory architectures to show that the hardware options achieve better performances than the software. In fact, the version with three memory blocks can run at least 1.75 times faster.

In a nutshell, this work emphasizes that although IA methods can undoubtedly be used for mapping and routing NoC applications, they can successfully be complemented by algorithmic methods.

Author Contributions: Conceptualization, L.R.-X. and A.P.; methodology, L.R.-X.; software, L.R.-X.; validation, L.R.-X. and A.P.; formal analysis, L.R.-X. and A.P.; investigation, L.R.-X. and A.P.; resources, L.R.-X. and A.P.; data curation, L.R.-X.; writing—original draft preparation, L.R.-X. and A.P.; writing—review and editing, L.R.-X. and A.P.; visualization, A.P.; supervision, A.P.; project administration, A.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article has been generated by software that will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CUDA	Compute Unified Device Architecture
DAG	Directed acyclic graph
EFSM	Extended finite-state machine
FPGA	Field-programmable gate array
FPSoC	Field-Programmable System-on-Chip
HA	Hungarian algorithm
NoC	Network-on-Chip
NP	Nondeterministic polynomial time complexity class
PE	Processing element
SDNoC	Software-defined NoC
SoC	System-on-Chip
VHDL	VHSIC hardware description language

References

1. Wang, C.; Zhu, Y.; Jiang, J.; Qiu, M.; Wang, X. Dynamic application allocation with resource balancing on NoC based many-core embedded systems. *J. Syst. Archit.* **2017**, *79*, 59–72. [CrossRef]
2. Khare, A.; Nallamalli, M.; Patil, C.; Chattopadhayay, S. Mapping and Priority Assignment for Real-Time Network-on-chip with Static and Dynamic Applications. In Proceedings of the 4th International Conference for Convergence in Technology (I2CT), Mangalore, India, 27–28 October 2018; pp. 1–6. [CrossRef]
3. Paul, S.; Chatterjee, N.; Ghosal, P. Dynamic task allocation and scheduling with contention-awareness for Network-on-Chip based multicore systems. *J. Syst. Archit.* **2021**, *115*, 102020. [CrossRef]
4. Paul, S. An Adaptive Strategy for Dynamic Resource Allocation and Scheduling for Multitasking NoC based Multicore Systems. In Proceedings of the 28th International Symposium on VLSI Design and Test (VDAT), Vellore, India, 1–3 September 2024; pp. 1–6. [CrossRef]
5. Kuhn, H.W. The Hungarian method for the assignment problem. *Nav. Res. Logist. Q.* **1955**, *2*, 83–97. [CrossRef]
6. Benini, L.; De Micheli, G. Networks on Chips: A new SoC paradigm. *Computer* **2002**, *35*, 70–78. [CrossRef]
7. Bjerregaard, T.; Mahadevan, S. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.* **2006**, *38*. [CrossRef]
8. Sahu, P.K.; Chattopadhyay, S. A survey on application mapping strategies for Network-on-Chip design. *J. Syst. Archit.* **2013**, *59*, 60–76. [CrossRef]
9. Arm Limited. ARM AMBA CHI 700 CoreLink Technical Reference Manual. 2023. Available online: https://developer.arm.com/documentation/101569/0300/?lang=en (accessed on 29 March 2025).
10. Jiao, J.; Shen, R.; Chen, L.; Liu, J.; Han, D. RLARA: A TSV-Aware Reinforcement Learning Assisted Fault-Tolerant Routing Algorithm for 3D Network-on-Chip. *Electronics* **2023**, *12*, 4867. [CrossRef]
11. Zheng, Y.L.; Song, T.T.; Chai, J.X.; Yang, X.P.; Yu, M.M.; Zhu, Y.C.; Liu, Y.; Xie, Y.Y. Exploring a New Adaptive Routing Based on the Dijkstra Algorithm in Optical Networks-on-Chip. *Micromachines* **2021**, *12*, 54. [CrossRef] [PubMed]
12. Yang, X.P.; Song, T.-T.; Ye, Y.-C.; Liu, B.-C.; Yan, H.; Zhu, Y.-C.; Zheng, Y.-L.; Liu, Y.; Xie, Y.-Y. A Novel Algorithm for Routing Paths Selection in Mesh-Based Optical Networks-on-Chips. *Micromachines* **2020**, *11*, 996. [CrossRef] [PubMed]
13. Fang, J.; Zhang, D.; Li, X. ParRouting: An Efficient Area Partition-Based Congestion-Aware Routing Algorithm for NoCs,. *Micromachines* **2020**, *11*, 1034. [CrossRef] [PubMed]
14. Scionti, A.; Mazumdar, S.; Portero, A. Towards a Scalable Software Defined Network-on-Chip for Next Generation Cloud. *Sensors* **2018**, *18*, 2330. [CrossRef] [PubMed]
15. Gómez-Rodríguez, J.R.; Sandoval-Arechiga, R.; Ibarra-Delgado, S.; Rodriguez-Abdala, V.I.; Vazquez-Avila, J.L.; Parra-Michel, R. A Survey of Software-Defined Networks-on-Chip: Motivations, Challenges and Opportunities. *Micromachines* **2021**, *12*, 183. [CrossRef] [PubMed]
16. Minisini, A.; Kulkov, O. Hungarian Algorithm for Solving the Assignment Problem. 13 December 2023. Available online: https://cp-algorithms.com/graph/hungarian-algorithm.html (accessed on 12 May 2025).
17. Ribas-Xirgo, L. TAS: Task Assignment Solver. SourceForge. December 2024. Available online: https://sourceforge.net/projects/tas/ (accessed on 12 May 2025).
18. Liu, W.; Xu, J.; Wu, X.; Ye, Y.; Wang, X.; Zhang, W.; Nikdast, M.; Wang, Z. A NoC Traffic Suite Based on Real Applications. In Proceedings of the 2011 IEEE Computer Society Annual Symposium on VLSI, Chennai, India, 4–6 July 2011. Available online: https://personal.hkust-gz.edu.cn/jiangxu/traffic.html (accessed on 9 July 2025).
19. Yadav, S.S.; Lopes, P.A.C.; Ilic, A.; Patra, S.K. Hungarian algorithm for subcarrier assignment problem using GPU and CUDA. *Int. J. Commun. Syst.* **2019**, *32*, e3884. [CrossRef]
20. Ribas-Xirgo, L. State-oriented programming-based embedded systems' course. In Proceedings of the XXIV Jornadas sobre la Enseñanza Universitaria de la Informática, Barcelona, Spain, 4–6 July 2018; pp. 79–86. (In Spanish)
21. Ribas-Xirgo, L. *How to Code Finite State Machines (FSMs) in C. A Systematic Approach*; Technical report TR01.102791 Embedded systems; Universitat Autònoma de Barcelona: Barcelona, Spain, 2014. [CrossRef]
22. Pont, M.J. *Embedded C*; Pearson Education Ltd.: Essex, UK, 2005.
23. Ribas-Xirgo, L. How to implement the Hungarian algorithm into hardware. In Proceedings of the 39th Conference on Design of Circuits and Integrated Circuits (DCIS), Catania, Italy, 13–15 November 2024.
24. Skahill, K. *VHDL for Programmable Logic*; Cypress Semiconductor: San Jose, CA, USA, 1995.