Modélisation de systèmes sur puce

Samy-William AYYADA
Antoine BERTRAND
Groupe TD n°3

Table des matières

1 Description générale

1.1 Introduction

Les *Systems on Chip* (SoC), également appelés "systèmes sur puce" en français, sont des éléments électroniques qui gagnent en popularité dans le secteur des semi-conducteurs. Cette tendance s'explique en partie par la loi de Moore, selon laquelle le nombre de transistors sur une puce double tous les deux ans. Ainsi, les SoC offrent des avantages considérables en termes de réduction de taille, d'accélération du calcul et de minimisation des coûts.

Un SoC est en essence un microprocesseur qui intègre non seulement des unités de calcul et de la mémoire vive, mais aussi des capteurs et des dispositifs de communication tels que les modules Wifi, Bluetooth et GPS. Ces éléments sont reliés les uns aux autres par un système de réseau dénommé *Network on Chip* (NoC).

Le rôle d'un SoC est de réaliser une variété de tâches en se basant sur une liste de requêtes qui permettent de transmettre des données entre les différents composants du système à travers le NoC. L'objectif de ce projet informatique est de modéliser le fonctionnement d'un SoC en utilisant le langage de programmation Python.

La première partie de ce projet consiste à modéliser l'architecture et le comportement d'un SoC et de réaliser une batterie de test pour vérifier le bon fonctionnement du programme. De plus, des politiques de sécurité seront introduites aux requêtes pour assurer la protection des données sur des plages d'adresses. Un système de priorité sur les requêtes sera également mis en place.

Dans la deuxième partie du projet, il est question de développer une interface homme machine en utilisant le module `PyQt5` de Python.

1.2 Hypothèses choisies

Il est difficile de prendre en compte tous les paramètres entrant en jeu dans un système aussi complexe que les SoC C'est la raison pour laquelle nous allons adopter les hypothèses et les choix de représentation suivants :

- Les uniques composants du SoC seront les nœuds de calcul et les nœuds mémoire. Les autres composants comme les périphériques ne seront pas pris en compte par soucis de simplification.
- Les nœuds de calculs sont définis par un identifiant unique, un temps de calcul (`computation_time`) et la taille du cache (`cache_size`) qui est la mémoire vive du nœud de calcul.

- Les nœuds de mémoire sont définis par un identifiant unique, la taille de leur mémoire (`capacity`) et une plage d'adresse protégée (`protected_range`) qui est intialisée à `None` et qui prendra pour valeur `(0, fin_de_plage)`.
- Chaque nœud possède un identifiant unique (`id`), un type (calcul : `'C'` ou mémoire : `'M'`) et un dictionnaire où sont recensés les nœuds avec lesquels il est connecté (`connections`). On notera qu'un nœud de mémoire et un nœud de calcul ne pourront avoir le même identifiant au sein d'un même NoC.
- Pour mesurer la distance entre deux nœuds connectés, on utilisera exclusivement la latence (`latency`), *i.e.* la durée de transmission d'une donnée entre les deux nœuds.
- Enfin, pour créer une tâche, on devra définir un nœud de départ, un nœud d'arrivée (`start_node_id` et `end_node_id`), et une donnée (`data`). La modélisation devra s'occuper de trouver le chemin valide le plus court (*i.e.* celui qui minimise la latence totale et le temps de calcul total) et fournir une liste de requêtes à effectuer pour réaliser la tâche.
  Par exemple, pour envoyer une donnée du nœud 1 au nœud 3, le chemin le plus court est de passer par le nœud 2 (1 et 3 ne sont pas directement connectés). Dans ce cas, pour réaliser la tâche, il faut effectuer deux requêtes : envoyer la donnée de 1 à 2 puis de 2 à 3.

1.3 Implémentation en Python

Le projet est composé de quatre fichiers Python.

- `architecture.py` : Contient les définitions des classes `SoC`, `NoC`, et des nœuds (`Node`, `MemoryNode` et `CalculNode`).
- `comportement.py` : Contient les définitions des classes tâche (`Task`) et requête (`Request`).
- `test.py` : Contient la batterie de tests sur les classes définies précédemment.
- `interface.py` : Contient les classes pour afficher l'interface utilisateur. **C'est ce fichier qu'il faut exéctuer pour lancer l'interface utilisateur**.

Sont également fournis deux fichiers `pickle` :

- `test_noc.pkl` : Exemple de NoC qui peut être importé et utilisé par l'utilisateur dans l'interface.
- `test_taches.pkl` : Exemple de liste de tâches qui peut être importé et utilisé par l'utilisateur dans l'interface.

Les méthodes sont accompagnées de commentaires pour faciliter leur compréhension. Par ailleurs, certaines méthodes importantes sont expliquées dans le présent rapport.

2 Présentation du volet architectural

2.1 La classe `SoC`

La classe SoC est un dictionnaire de NoC ("Network on Chip"). Elle sert de conteneur pour les divers NoC qui forment un système sur puce dans cette modélisation. Quand une instance de SoC est créée à l'aide du constructeur, un dictionnaire vide est initialisé. Ce dictionnaire, attribut nommé `NoC`, est destiné à contenir tous les NoC qui composent ce SoC. De plus, un attribut nommé size est également initialisé à zéro lors de la création d'une instance de SoC. Cet attribut sert à conserver le nombre de NoC actuellement stockés dans le SoC. Il est particulièrement utile pour suivre l'évolution de la taille du SoC au fur et à mesure de l'ajout de nouveaux NoC.

Il convient de souligner que notre modélisation s'est principalement portée sur les NoC. C'est pour cette raison que la classe SoC ne possède pas davantage de méthodes.

2.2 La classe `NoC`

FIGURE 1 – Diagramme de la classe `NoC`

Chaque instance de la classe `NoC` contient deux dictionnaires, `memory_nodes` et `calcul_nodes`, qui stockent respectivement les nœuds de mémoire et les nœuds de calcul. On rappelle que chaque nœud a un identifiant unique et peut

contenir des attributs supplémentaires, tels que la capacité de la mémoire ou la taille du cache pour les nœuds de calcul.

La classe offre une variété de méthodes :

- `to_networkx_graph` : Permet de transformer l'instance NoC en un graphe NetworkX, ce qui facilite la représentation du NoC.
- `add_node` : Permet d'ajouter un nœud au NoC. Le nœud peut être de type "M" (mémoire) ou "C" (calcul).
- `get_node` : Renvoie l'objet MemoryNode ou CalculNode correspondant à l'identifiant donné dans le NoC considéré.
- `remove_node` : Permet de supprimer un nœud du NoC.
- `shortest_path` : Trouve le chemin le plus court entre deux nœuds en utilisant l'algorithme de Dijkstra. La méthode `shortest_path` est expliquée en détail dans la partie 2.2.1.
- `are_nodes_connected` : Vérifie si deux nœuds sont directement connectés dans le NoC.
- `add_connection` : Ajoute une connexion entre deux nœuds dans le NoC considéré.
- `remove_connection` : Supprime une connexion entre deux nœuds dans le NoC considéré.
- `clear` : Supprime tous les nœuds et les connexions du NoC.
- `clean` : Supprime toutes les données stockées dans les mémoires des nœuds du NoC.
- `random` et `new_random` : Ces deux méthodes génèrent aléatoirement des nœuds et des connexions dans le NoC. La méthode `new_random` est expliquée en détail dans la partie 2.2.2.

2.2.1 La méthode `shortest_path`

La méthode `shortest_path` est une implémentation de l'algorithme de Dijkstra pour trouver le chemin le plus court entre deux nœuds dans un graphe pondéré. Cette fonction prend en entrée deux identifiants de nœuds (`start_node_id` et `end_node_id`), et retourne le chemin le plus court entre ces deux nœuds ainsi que son poids total (en termes de latence et de temps de calcul).

La fonction utilise la récursion pour explorer les voisins de chaque nœud visité et construire les chemins possibles. Elle maintient une liste de nœuds

visités (`visited`) pour éviter les boucles infinies, ainsi qu'un chemin en cours (`current_path`) et son poids total (`current_weight`).

La fonction parcourt les voisins du nœud de départ, appelle récursivement la fonction `shortest_path` pour chacun des voisins non visités, et met à jour le chemin le plus court (`shortest`) et son poids minimal (`min_weight`) si un chemin plus court est trouvé.

En cas d'erreur, la fonction lève une exception `ValueError` si les nœuds de départ ou d'arrivée n'existent pas dans le graphe.

Il est important de souligner que l'algorithme de Dijkstra aurait pu être mis en œuvre de façon itérative, ce qui aurait probablement réduit le temps de calcul. Cependant, conformément à notre objectif d'implémenter une fonction récursive (figure imposée) et compte tenu du fait que notre implémentation actuelle maintient un temps de calcul raisonnable (sans latence perceptible dans un usage standard), nous avons opté pour une approche récursive. Ce choix, bien que potentiellement moins optimal en termes de performance, nous permet d'explorer et de mettre en pratique les principes de la récursivité, un aspect fondamental de la programmation.

2.2.2 La méthode `new_random`

La méthode `new_random` est utilisée pour générer un nouvel état de *Network on Chip* (NoC) avec des attributs aléatoires pour les différents nœuds du réseau.

Cette fonction a les paramètres suivants :

— `nb_cac` : le nombre de nœuds de calcul dans le réseau.

— `nb_mem` : le nombre de nœuds mémoire dans le réseau.

— `rnd` : un paramètre qui détermine la probabilité de connexion entre deux nœuds.

— `**kwargs` : cela permet de spécifier des plages de valeurs pour divers attributs de nœuds, comme la latence, la taille de la cache, la taille de la mémoire et le temps de calcul.

La méthode commence par définir des valeurs par défaut pour les plages d'attributs de nœud (latence, taille de cache, taille de mémoire, temps de calcul). Si des plages alternatives sont fournies via `kwargs`, celles-ci remplaceront les valeurs par défaut.

La méthode crée ensuite un certain nombre de nœuds de calcul (`nb_cac`) avec des valeurs aléatoires pour la taille de la cache et le temps de calcul. Elle crée également un certain nombre de nœuds mémoire (`nb_mem`) avec des valeurs aléatoires pour la taille de la mémoire.

Enfin, la méthode établit des connexions entre les nœuds. La probabilité qu'une connexion soit établie est basée sur la valeur de `rnd`. Une latence aléatoire est également attribuée à chaque connexion. Une connexion ne sera pas créée si les deux nœuds sont déjà connectés.

Il convient de noter que si `rnd` est défini comme 0, aucune connexion ne sera établie, car la probabilité de connexion est $1/\texttt{rnd}$.

2.3 La classe `Node`

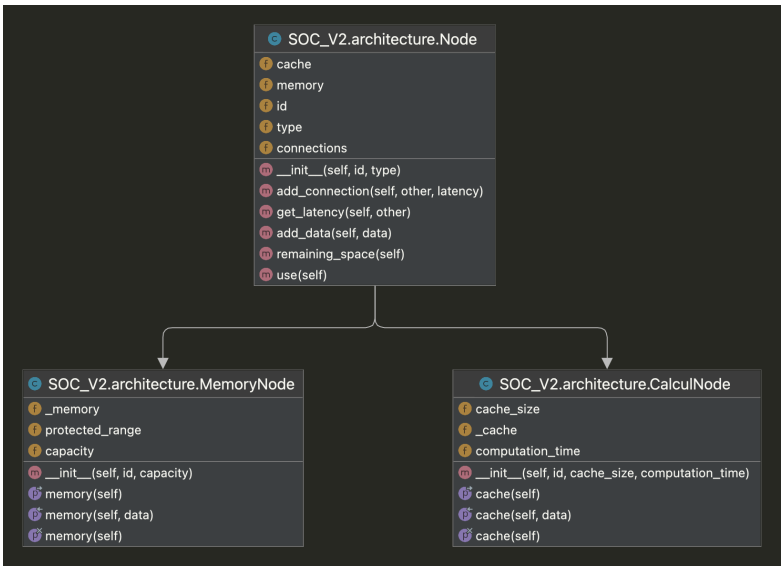

FIGURE 2 – Diagramme des classes `Node`, `MemoryNode` et `CalculNode`

La classe `Node` est la super-classe de `CalculNode` (classe des nœuds de calcul) et `MemoryNode` (classe des nœuds de mémoire). Elle représente un de manière générale un nœud dans NoC. Chaque instance de Node a un identifiant unique (`id`) et un dictionnaire `connections` qui contient des informations sur les autres nœuds auxquels il est connecté. La clé dans ce dictionnaire est l'identifiant du nœud connecté et la valeur est la latence de la connexion. De plus, chaque nœud a un type qui est défini à la création de l'objet : la variable `type`, de type `char`, vaut 'M' pour un nœud de mémoire et 'C' pour un nœud de calcul.

Elle possède plusieurs méthodes dont héritent ses sous-classes :

- `add_connection` : Permet d'ajouter une connexion à un autre nœud. Elle vérifie si l'autre nœud est une instance de la classe `Node` et si une connexion existe déjà. Elle ajoute ensuite la connexion des deux côtés avec la latence spécifiée.

- `get_latency` : Permet de récupérer la latence de la connexion à un autre nœud. Si les nœuds ne sont pas connectés, elle génère une erreur.
- `add_data` : Permet d'ajouter des données à un nœud. L'emplacement des données dépend du type de nœud : pour un nœud de calcul, les données sont stockées dans le cache, pour un nœud de mémoire, elles sont stockées dans la mémoire.
- `remaining_space` : Renvoie l'espace disponible dans le nœud. Si le nœud est un nœud de calcul, il renvoie la taille du cache (on peut toujours écrire dans le cache, il n'y a pas d'adresses protégées), sinon, il renvoie la capacité de mémoire disponible (c'est-à-dire la taille de la plage d'adresses non-protégées).
- `use` : Permet de transformer l'instance NoC en un graphe `NetworkX`, ce qui facilite la représentation du NoC.

2.4 La classe `CalculNode`

La sous-classe `CalculNode` permet de modéliser un nœud de calcul. Elle hérite de la classe `Node` et ajoute des attributs et des méthodes supplémentaires spécifiques aux nœuds de calcul. Les attributs supplémentaires sont la taille du cache (`cache_size`), le temps de calcul (`computation_time`), et le cache lui-même (`_cache`), variable privée qui est initialisé à une liste de `None` de la taille du cache. Les méthodes ajoutées comprennent des décorateurs pour le cache. Le setter du cache (`cache.setter`) est conçu pour gérer efficacement l'ajout de données dans le cache en tenant compte de l'espace disponible et en écrasant les données existantes si nécessaire.

2.5 La classe `MemoryNode`

La classe `MemoryNode` permet de modéliser un nœud de mémoire. Elle hérite également de la classe Node et ajoute des attributs et des méthodes spécifiques aux nœuds de mémoire. Les attributs supplémentaires sont la capacité de la mémoire (`capacity`), la mémoire elle-même (`memory`), qui est une variable protégée initialisée à une liste de `None` de la taille de la capacité, et une plage protégée (`protected_ranges`), qui est initialisée à `None` lorsque qu'aucune adresse n'est protégée. Lorsque des adresses sont protégées, `protected_ranges` est un `tuple` de la forme `(0, fin_plage_protegee)`. Les méthodes ajoutées comprennent des décorateurs pour la mémoire : getter, setter, et deleter. Le setter de la mémoire (`memory.setter`) est conçu pour ajouter des données dans la mémoire dans la plage d'adresses non protégées

tout en vérifiant que l'espace disponible est suffisant. La plage protégée est mise à jour chaque fois que des données sont ajoutées à la mémoire.

3 Présentation du volet comportemental

3.1 La classe `Request`

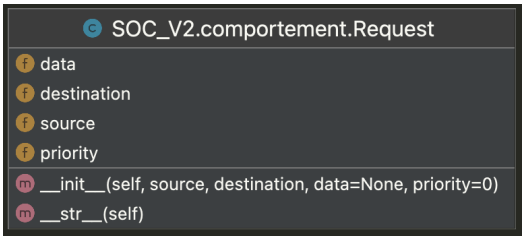

FIGURE 3 – Diagramme de la classe `Request`

La classe `Request` représente une requête de transfert de données dans un NoC. **Ce transfert de données se fait entre deux nœuds en liaison directe**. Chaque requête est caractérisée par :

- Un nœud `source` qui est le nœud d'origine de la donnée à transférer.
- Un nœud `destination` qui est le récepteur de la donnée.
- Les données `data` qui doivent être transférées.
- Une priorité `priority` qui permet de déterminer l'ordre dans lequel les requêtes doivent être exécutées. Un nombre plus petit indique une priorité plus élevée, avec 0 étant la plus haute priorité.

La méthode `__str__` est utilisée pour obtenir une représentation lisible de la requête, qui peut être utile pour le débogage.

3.2 La classe `Task`

Une tâche de la classe `Task` représente un transfert de données entre deux nœuds qui ne sont pas en liaison directe. Ce transfert est alors découpé en plusieurs requêtes (transfert entre deux nœuds en liaison directe).

Ainsi, la classe `Task` est une collection de requêtes organisées par priorité. Cette classe hérite du dictionnaire Python, où les clés sont les niveaux de priorité et les valeurs sont les requêtes correspondantes.

`Task` comprend plusieurs méthodes :

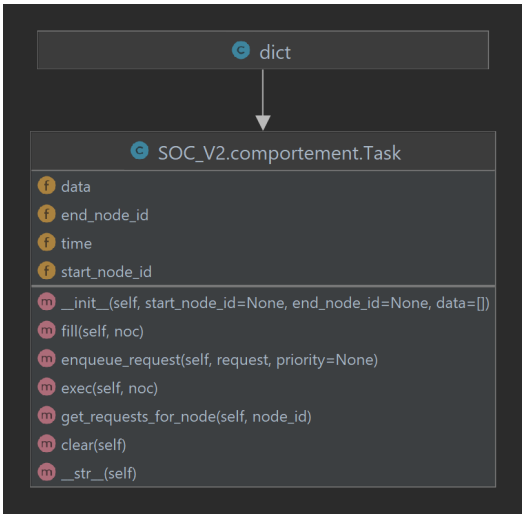

FIGURE 4 – Diagramme de la classe `Task`

- `__init__` : initialisation d'une tâche avec un nœud de départ (`start_node_id`), un nœud d'arrivée (`end_node_id`) et une liste de données (`data`) à transférer.
- `fill` : cette méthode remplit la tâche avec des requêtes. Elle est détaillée dans la `partie 3.2.2`.
- `enqueue_request` : ajoute une nouvelle requête à la tâche, en lui attribuant une priorité spécifique. Elle est détaillée dans la `partie 3.2.1`.
- `exec` : exécute la tâche, c'est-à-dire transfère les données de la tâche de son point de départ à son point d'arrivée via les nœuds du réseau NoC, en suivant la séquence de requêtes ajoutées à la tâche par la méthode `fill`. Cette méthode est détaillée dans la `partie 3.2.3`.
- `get_requests_for_node` : cette méthode renvoie toutes les requêtes qui concernent un nœud spécifique (soit en tant que nœud source, soit en tant que nœud de destination).
- `clear` : cette méthode supprime toutes les requêtes de la tâche, réinitialisant l'objet Task.
- `__str__` : renvoie une représentation lisible de la tâche.

3.2.1 La méthode `enqueue_request`

La méthode `enqueue_request` de la classe `Task` est utilisée pour ajouter une nouvelle requête à la tâche. Cette méthode prend deux arguments : `request`, qui est la requête à ajouter, et `priority`, qui est la priorité à

laquelle la requête doit être ajoutée. Si aucune priorité n'est spécifiée, la requête est ajoutée à la fin de la tâche.

Si une priorité est spécifiée et qu'une requête existe déjà à cette priorité dans la tâche, toutes les requêtes de cette priorité et des priorités supérieures sont décalées d'un rang vers des priorités plus faibles (c'est-à-dire que leur priorité est augmentée de 1). Par construction d'une tâche, les requêtes ont des priorités qui se suivent numériquement (une priorité de n est forcément suivie d'une priorité de $n+1$)

Cette méthode permet de maintenir l'ordre de priorité des requêtes dans la tâche tout en permettant d'ajouter de nouvelles requêtes à des priorités spécifiques.

3.2.2 La méthode `fill`

La méthode `fill` de la classe `Task` est utilisée pour remplir une tâche avec des requêtes. Elle permet de déterminer le chemin le plus court pour transférer les données d'un nœud de départ à un nœud d'arrivée dans un réseau NoC.

La fonction commence par créer une copie du NoC afin de ne pas modifier le réseau original avant de parcourir tous les nœuds du réseau pour vérifier s'ils ont suffisamment d'espace libre pour stocker les données à transférer. Si un nœud n'a pas suffisamment d'espace, il est retiré du NoC temporaire, sauf s'il s'agit du nœud de départ ou d'arrivée.

La méthode utilise ensuite la méthode `shortest_path` de la classe `NoC` pour trouver le chemin le plus court du nœud de départ au nœud d'arrivée dans le réseau temporaire. Si un tel chemin existe, la méthode enregistre le temps nécessaire pour effectuer le transfert et crée une série de requêtes correspondant à ce chemin. Chaque requête correspond à un saut d'un nœud à son voisin dans le chemin. Ces requêtes sont ensuite ajoutées à la tâche en utilisant la méthode `enqueue_request`.

Si aucun chemin n'existe (c'est-à-dire si le nœud de départ et le nœud d'arrivée ne sont pas connectés dans le réseau temporaire), la méthode fill lève une exception, indiquant qu'il n'est pas possible de réaliser la tâche.

3.2.3 La méthode `exec`

La méthode `exec` de la classe `Task` est utilisée pour exécuter une tâche, c'est-à-dire pour effectuer le transfert de données d'un point de départ à un point d'arrivée via les nœuds d'un réseau NoC.

Au début de l'exécution, la méthode `exec` ajoute les données à transférer au nœud de départ. Ensuite, elle parcourt toutes les requêtes contenues dans

l'objet `Task`, dans l'ordre de leur priorité. Pour chaque requête, la méthode `exec` transfère les données de la source à la destination de la requête. Le transfert de données est effectué en appelant la méthode `add_data` du nœud de destination. Cette méthode ajoute de nouvelles données dans la mémoire ou le cache du nœud en fonction de son type. La méthode `exec` ne renvoie aucune valeur. Une fois qu'elle a terminé de parcourir toutes les requêtes, le transfert de données est considéré comme terminé.

4 Interface homme-machine

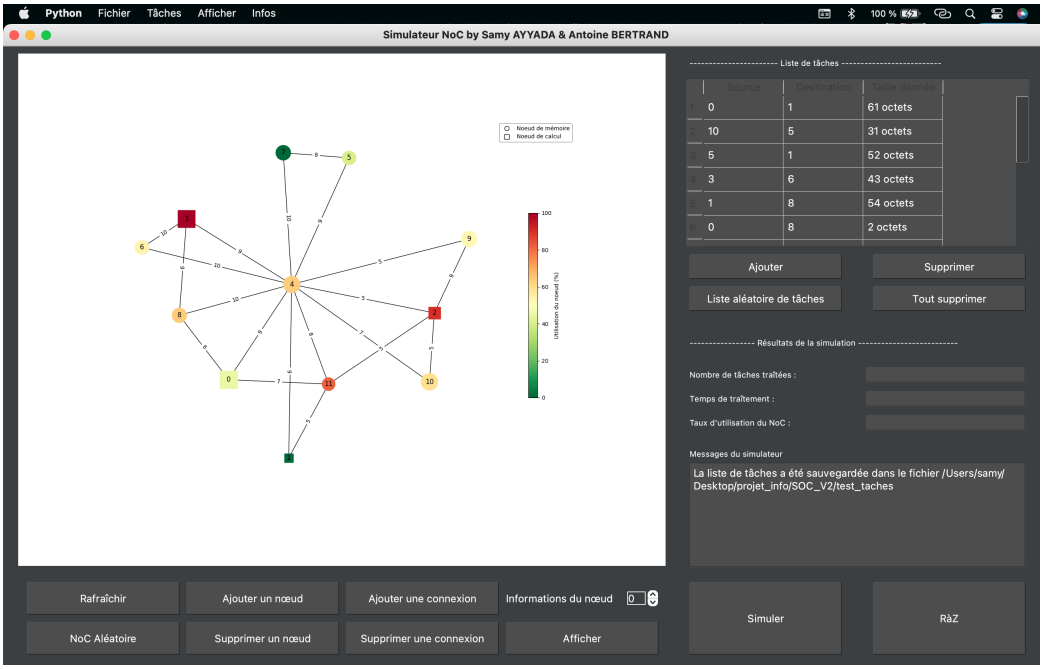

FIGURE 5 – Interface homme-machine

L'interface utilisateur, conçue à l'aide de `PyQt5`, est accessible en exécutant le fichier `interface.py`. Cette interface interactive offre à l'utilisateur la capacité de simuler une variété de scénarios dans le cadre d'un NoC.

Cette interface graphique donne une représentation visuelle du NoC, illustrant ses différents nœuds et les connexions existantes entre eux. Elle offre une grande flexibilité à l'utilisateur, lui permettant de personnaliser le NoC en ajoutant ou en supprimant des nœuds et des connexions, en fonction des exigences de la simulation.

De plus, l'interface permet à l'utilisateur de créer des tâches, représentant des transferts de données à travers le NoC. Pour faciliter la mise en place

de simulations, l'utilisateur peut opter pour la génération automatique d'un NoC et d'une liste de tâches.

Dans cette interface utilisateur, l'option de "Rafraîchir" manuellement la représentation graphique du NoC a été intégrée pour offrir à l'utilisateur un contrôle maximal. Cela permet d'éviter des mises à jour automatiques constantes, donnant à l'utilisateur la liberté de visualiser les changements qu'il effectue sur le NoC à son propre rythme.

Les nœuds sont symbolisés sous forme de rond pour les nœuds de mémoire et de carré pour les nœuds de calcul. Leur couleur varie, du vert au rouge, en fonction de leur niveau d'utilisation.

Des messages d'erreur ou d'avertissement ont également été rajoutés afin d'améliorer l'ergonomie de l'interface pour l'utilisateur.

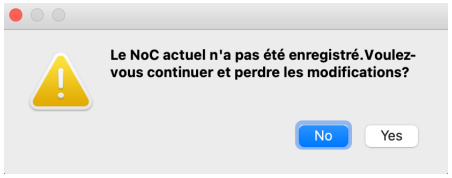

FIGURE 6 – Message d'avertissement déclenché lorsque l'utilisateur s'apprête à écraser le NoC actuel sans l'avoir enregistré.

FIGURE 7 – Message d'erreur déclenché lorsque l'utilisateur tente de supprimer un nœud qui n'existe pas.

L'interface offre également la possibilité d'enregistrer le NoC et la liste des tâches dans un fichier au format `pickle`, permettant ainsi à l'utilisateur de sauvegarder et de recharger ses configurations.

Une fois une simulation lancée, en cliquant sur le bouton "Simuler", les tâches sont exécutées par le NoC. En mettant à jour la représentation graphique (bouton "Rafraîchir"), il est possible de visualiser les noeuds les plus sollicités. Par ailleurs, un rapport de simulation est généré et stocké dans le fichier nommé `simulation_report.txt`. Ce rapport contient des détails précis sur le déroulement de la simulation, fournissant à l'utilisateur un aperçu exhaustif des opérations effectuées durant la simulation et un résumé des performances du NoC.

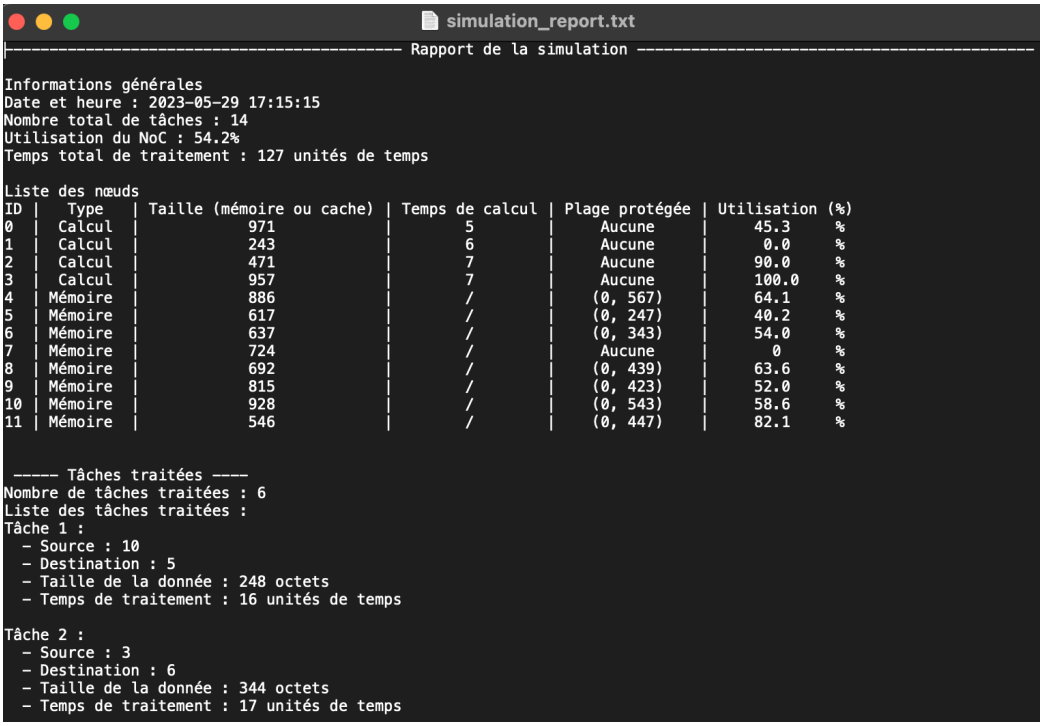

FIGURE 8 – Exemple de rapport de simulation

5 Conclusion

5.1 Figures imposées

No	Figures imposées	Code réalisé
1	Factorisation du code	Modules `architecture.py`, `comportement.py` et `interface.py`
2	Création d'au moins trois classes	5 classes : `NoC`, `Node`, `MemoryNode`, `CalculNode`, `Task` et `Request`
3	Héritage entre trois types créés	Les classes `MemoryNode` et `CalculNode` héritent de `Node`
4	Héritage depuis un type intégré	La classe `Task` hérite du type `dict` déja intégré dans Python
5	Documentation	Chaque méthode, classe est documentée
6	Tests unitaires	Tous les tests unitaires sont dans le fichier `test.py`
7	Fonction récursive	La méthode `shortest_path()` de la classe `NoC` utilise la récursivité
8	Théorie des graphes	Utilisation de l'algorithme de Dijkstra dans la méthode `shortest_path()`
9	Lecture/écriture de fichiers	Lecture/écriture des fichiers `.pkl` pour le NoC et les listes de tâches. Ecriture d'un fichier `.txt` pour le rapport de simulation.
10	Représentation graphique	Module `interface.py` permettant d'afficher une interface graphique avec PyQt5

TABLE 1 – Listes des figures imposées

5.2 Tests réalisés

Des tests unitaires ont été écrit dans un fichier dédié nommé `test.py`. Ce choix a permis de centraliser et d'organiser de manière claire et structurée l'ensemble des tests relatifs à notre projet. Ils ont été réalisés à l'aide de la bibliothèque `unittest` de Python. Les classes testées sont `Node`, `CalculNode`, `MemoryNode`, `NoC`, `Task`, et `Request`.

Pour la classe `Node` :

- Test d'ajout d'une connexion entre deux nœuds.

- Test d'ajout d'une connexion avec le même nœud.
- Test d'ajout d'une connexion avec un objet qui n'est pas un nœud.
- Test d'ajout d'une connexion déjà existante.
- Test de récupération de la latence d'une connexion.
- Test de récupération de la latence d'une connexion inexistante.

Pour la classe `CalculNode` :

- Test de création d'un nœud de calcul avec un cache.
- Test d'ajout d'une valeur dans le cache d'un nœud de calcul.
- Test d'ajout d'une valeur dans un cache déjà plein.
- Test de suppression du cache d'un nœud de calcul.

Pour la classe `MemoryNode` :

- Test de création d'un nœud de mémoire et vérification de ses propriétés.
- Test d'ajout de valeurs dans la mémoire.
- Test d'ajout d'une liste de valeurs trop grande pour la capacité du nœud.
- Test de suppression de la mémoire.

Pour la classe `NoC` :

- Test d'ajout et de suppression de nœuds de calcul et de mémoire.
- Test de récupération du type et de l'objet d'un nœud.
- Test du calcul du chemin le plus court entre deux nœuds.
- Test de la méthode `are_nodes_connected` qui vérifie si deux nœuds sont voisins directs.

Pour les classes `Task` et `Request` :

- Test de création d'une requête et vérification de ses attributs.
- Test de création d'une tâche et vérification si elle est vide.
- Test d'ajout de requêtes dans une tâche.
- Test de la méthode `clear` qui permet la suppression de toutes les requêtes d'une tâche.
- Test de la représentation d'une tâche par la méthode `__str__`.
- Test de l'exécution d'une tâche.

Ces tests ont permis d'identifier et de corriger des erreurs dans le code.

Dans la version finale, tous les tests cités ont été exécutés avec succès : les différentes classes et méthodes implémentées fonctionnent comme prévu.

5.3 Apport personnel : L'algorithme de Dijkstra

Le sujet ne spécifiant pas comment les connexions s'organisaient entre les nœuds, nous avons choisi de partir du principe que les connexions pouvaient se faire de n'importe quelle manière avec n'importe quelle latence. Ainsi, le but de la méthode `shortest_path` est de trouver le chemin le plus court à travers toutes les connexions entre les nœuds. Ceci est particulièrement important dans le contexte d'un NoC, où la minimisation de la latence peut contribuer à améliorer les performances globales du système.

Pour résoudre se problème des Graphes, nous avons choisi d'utiliser l'algorithme de Dijkstra. En effet, dans un NoC, chaque nœud peut être considéré comme un sommet du graphe, chaque connexion entre les nœuds peut être considérée comme un arc du graphe, et la latence de chaque connexion peut être considérée comme le poids de l'arc correspondant. Il s'agit alors d'un graphe pondéré qui ne présente pas de poids négatifs (la latence est toujours supérieure à zéro) : les hypothèses d'utilisation de cet algorithme sont donc bien vérifiées.

5.4 Limitations et pistes d'amélioration

Le focus de la modélisation actuelle est principalement sur les réseaux sur puce (NoC) individuels. Ceci pourrait limiter la capacité du modèle à simuler des systèmes complexes où plusieurs NoC interagissent. La modélisation actuelle n'inclut pas la notion d'interconnexion entre plusieurs NoC, qui est un élément clé dans un système sur puce (SoC) complet. En réalité, plusieurs NoC peuvent coexister et interagir dans un SoC. Cela nécessite de modéliser également les liens entre différents NoC.

De plus, les tâches sont simplement modélisées comme des transferts de données entre les nœuds. Cela peut ne pas suffire à représenter le comportement de certaines tâches réelles qui peuvent inclure du calcul complexe ou être soumises à des contraintes de temps réel. La notion d'horloge est totalement absente de notre modélisation mais est une notion importante dans les systèmes réels. Une approche possible pour améliorer la modélisation serait d'intégrer une représentation de l'horloge du système et d'introduire des contraintes de temps réel pour les tâches. De plus, il pourrait être intéressant de modéliser non seulement le transfert de données, mais aussi les opérations de calcul que les nœuds doivent effectuer.

Enfin, la sécurité est un élément crucial dans la conception et le fonctionnement des systèmes sur puce (SoC) et des réseaux sur puce (NoC). Cependant, la modélisation actuelle ne prend pas en compte cet aspect. Il serait intéressant de pouvoir garantir que seuls les nœuds autorisés puissent

accéder aux données qui circulent dans le réseau. Un modèle de sécurité adéquat doit être mis en place pour protéger contre les accès non autorisés. Il est également intéressant de pouvoir garantir l'intégrité des données qui circulent dans le réseau. Des mécanismes de sécurité doivent être mis en place pour détecter et prévenir la corruption des données. Le système doit aussi pouvoir garantir l'authenticité des nœuds pour prévenir les attaques de type "homme du milieu" où un attaquant pourrait se faire passer pour un nœud légitime.