Chapitre 5

Le routage distribué

On a tort d'apprendre aux enfants que tous les problèmes n'ont qu'une et une seule solution...

Alice FERNEY , *Extrait de Libération - 20 janvier 2001*

L'ENSEMBLE des solutions de routage matériel présentées dans le chapitre précédent possèdent une même caractéristique, à savoir qu'un contrôleur global est toujours présent. Dans l'optique de réaliser des systèmes cellulaires autonomes, nous nous proposons d'introduire des algorithmes distribués autonomes, ne nécessitant aucun contrôle global.

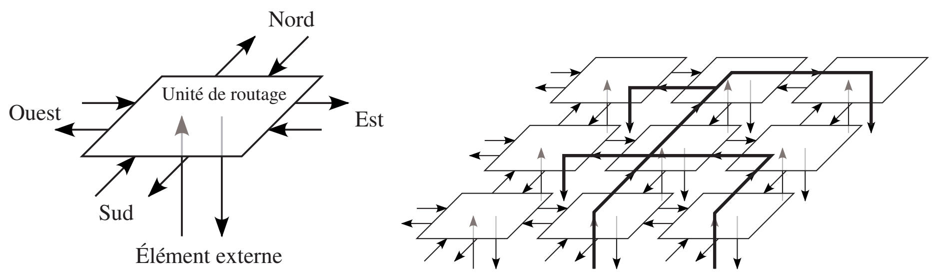
Dans ce chapitre nous présentons nos solutions au problème du routage appliqué aux circuits électroniques reconfigurables. Nous commençons par poser le concept général de notre système, les principes directeurs qui nous ont dirigés durant la mise au point de nos algorithmes, puis nous introduisons des hypothèses structurelles quant à son architecture. Nous proposons quelques solutions préliminaires, qui nous amènent à la première version de nos algorithmes, HIDRA, que nous présentons alors en détail. Suivent trois autres versions, HIDRA-RC, HIDRA-RT et HIDRA-RTC, ayant pour caractéristique respective de réduire le risque de congestion, le temps d'exécution ou une combinaison des deux. HIDRA-L, un algorithme fonctionnant uniquement avec des communications locales est alors présenté, comme alternative scalable aux quatre premiers algorithmes. Alors que ces implémentations sont basées sur des unités de routage reliées à leurs 4 voisines, nous proposons ensuite d'explorer des voisinages de 3, 6 et 8, et de les comparer sur le plan de l'efficacité en terme de congestion et de nombre de transistors requis. Finalement les expériences réalisées sur les différents algorithmes et voisinages sont présentées et les résultats analysés en détail.

5.1 Concept

Notre but est d'inclure un mécanisme de routage dynamiquement configurable dans un circuit reconfigurable. Dans les FPGAs commerciaux, le routage est calculé par un logiciel dédié, puis chargé dans le circuit lors de la configuration. Il n'est ensuite plus possible de le modifier, si ce n'est par une intervention externe sous la forme d'une reconfiguration. Les systèmes de processeurs parallèles utilisent fréquemment un mécanisme appelé “wormhole” [177], qui consiste à envoyer des paquets d'information au travers du réseau de processeurs, en connaissant la coordonnée exacte du destinataire. Chaque processeur possède donc un système de gestion du routage, qui fait transiter des parties de paquets dans la bonne direction, et évite tant que faire se peut les problèmes de congestion. L'implémentation de tels mécanismes nécessite cependant des piles capables de stocker momentanément des parties de paquets lorsque le réseau se congestionne, et donc leur réalisation purement matérielle peut être d'une certaine importance en terme de nombre de transistors (une implémentation en a été faite dans [127], qui comporte 15K transistors pour une unité de routage avec un voisinage de 4). De même, les systèmes de *paquet switching*, qui permettent d'adresser des unités de routage par le biais d'identifiants, nécessitent des unités de routage possédant des tables de routage. Or, dans le cas d'une implémentation matérielle, ces tables impliquent la réquisition d'une grande quantité de ressource.

Alors que le routage des FPGAs standards est statique, et que le type de routage wormhole ou packet switching est totalement dynamique, dans le sens où l'information est transmise dans une direction choisie dynamiquement par une unité de routage, nous proposons une solution intermédiaire, pseudo-dynamique. Des unités de routage y sont responsables de transmettre de l'information entre différents points du circuit. Ces unités peuvent être configurées grâce à un algorithme de routage, qui prend en compte les désirs des éléments connectés aux unités de routage, et lorsque le routage est effectué, les unités servent de transmetteurs statiques. Nous offrons donc la possibilité de modifier dynamiquement la structure du réseau, et la transmission d'information se fait ensuite toujours de la même manière, jusqu'à une modification ultérieure.

La figure 5.1(a) montre une unité de routage, qui dans notre cas est reliée à ses quatre voisines, ainsi qu'à un élément externe, qui n'est autre qu'une partie du circuit. Dans le cas de la réalisation du circuit POEtic, cet élément, relativement semblables aux éléments de base des FPGAs standards, est notamment composé d'une look-up table de 16 bits et d'une bascule. La figure 5.1(b) illustre le fait que la grille des unités de routage est utilisée pour faire transiter de l'information entre différents endroits du circuit.



(a) Unité de routage

(b) Les unités de routage transmettent de l'information au travers du circuit

Figure 5.1 : *Une unité de routage et un réseau de celles-ci.*

Durant tout ce chapitre nous utiliserons le terme “source” pour désigner une unité de routage chargée d'envoyer une information de la part d'un élément externe, au travers de la grille des unités de routage. Par analogie, une "destination" signifiera une unité de routage recevant cette information, et la transmettant à un élément externe. Il est également important de noter que les liaisons que nous désirons créer ne sont pas binaires. Une source peut y être connectée à plusieurs destinations, comme c'est le cas dans les réseaux de neurones, où la sortie d'une cellule est récupérée par un grand nombre d'autres cellules. De plus, un nouveau routage peut être lancé par une source ou une destination, cette dernière désignant la source à laquelle elle désire se connecter. En comparaison des techniques de wormhole, notre approche offre un temps de transfert d'information nettement plus faible, puisqu'elle se fait de façon purement combinatoire. De plus, le mécanisme à base d'identifiant que nous utilisons est nettement plus flexible que celui basé sur des adresses physiques des unités communiquantes.

5.2 Principes

Trois principes de base nous ont guidés durant la mise au point de nos algorithmes, de manière à les rendre les plus efficaces possibles dans le cas d'une implémentation matérielle.

- **Parallélisme.** Premièrement, nous nous intéressons aux systèmes parallèles, ou distribués. L'idée sous-jacente est ici de disposer d'un grand nombre d'éléments identiques capables de mener à bien la tâche de routage. Le projet encadrant cette thèse visant la réalisation d'un circuit de type FPGA, il est également clair qu'une structure régulière d'éléments identiques correspond entièrement au paradigme FPGA. Nous verrons plus loin que l'architecture retenue met en jeu une structure de type FPGA dont les éléments de base sont reliés à un deuxième niveau de modules responsables du routage.
- **Autonomie.** Deuxièmement, contrairement aux systèmes existants, nous désirons rendre le routage autonome. Aucun contrôleur global ne doit être présent, les unités de routage distribuées devant être seules maîtres de leur fonctionnement. Cette approche a été retenue dans l'optique de réaliser un système autoréparable. Un élément de contrôle global y serait un maillon faible qui, en cas de défaillance, pourrait compromettre l'ensemble du fonctionnement du système. Nous verrons que l'autoréparation n'a toutefois pas été investiguée dans cette thèse, mais que de futurs travaux pourraient être entrepris dans ce sens. Finalement, un des points forts d'un système autonome concerne les applications où des parties de circuit décident elles-mêmes de créer des connexions, tels des réseaux de neurones à topologie variable, où des neurones pourraient dynamiquement créer de nouvelles liaisons en fonction de leur taux d'activation. Dans de tels cas un contrôle global est superflu, et n'est pas en accord avec le concept de système cellulaire pour lequel un maximum d'autonomie est souhaitable.
- **Simplicité.** Troisièmement, notre optique est de travailler sur des systèmes physiquement réalisables sur du silicium et potentiellement en nanotechnologies dans le futur. Le nombre de transistors disponibles dans les circuits intégrés n'étant pas infini, une attention particulière doit être donnée à la minimisation de la taille de nos éléments de routage. Il est hors de question d'implémenter un protocole du type TCP/IP, c'est pourquoi le développement de nos algorithmes s'est effectué dans le respect du principe de simplicité. Chaque élément de routage est peu complexe, mais permet au système de fonctionner parfaitement.

Nous pouvons également noter que l'un des aboutissements de ce travail fut la réalisation physique d'un nouveau FPGA comportant un système de routage dynamique. Le peu de silicium à disposition nous a donc fortement poussé à une optimisation maximale de la taille de nos composants.

5.3 Hypothèses

Les trois principes directeurs énumérés ci-dessus ont fixé un cadre large concernant la conception matérielle d'un système de routage distribué autonome. Les hypothèses présentées dans cette section visent à donner un cadre plus étroit dans lequel tous les algorithmes développés doivent tenir.

1. Un système de routage est composé d'unités de routage identiques connectées à leurs plus proches voisines (principe de parallélisme). Nous présenterons les solutions concernant un voisinage de 4 avant de proposer d'autres alternatives.
2. Aucun contrôleur global ne doit être présent, l'entièreté de la gestion du routage étant laissée aux unités de routage (principe d'autonomie).
3. Nous tolérons des liaisons combinatoires de longue distance traversant les unités de routage présentes dans le système.
4. Une unité de routage est entre autre composée de multiplexeurs permettant de sélectionner le signal à transmettre à chacune des voisines. Le but de l'algorithme de routage est la configuration correcte de ces multiplexeurs.
5. Le routage terminé, les multiplexeurs transmettent de manière combinatoire les valeurs de proche en proche.
6. Chaque unité de routage est connectée à un élément externe, qui peut typiquement être un élément logique d'un FPGA. Cet élément externe reçoit et envoie des valeurs à l'unité de routage afin de mener à bien tout processus de routage ainsi que pour transmettre des informations à travers le circuit.
7. L'implémentation matérielle des algorithmes étant le but de notre étude, le design des unités de routage se doit de nécessiter un nombre de transistors le plus faible possible (principe de simplicité).
8. De même, le nombre de connexions entre les unités de routage doit être le plus petit possible.

La figure 5.2 illustre la grille à deux dimensions composée d'unités de routage interconnectées. Nous y observons également la présence des éléments externes reliés à ces unités. Dans ce chapitre nous ne définirons pas la structure de ces éléments, mais seulement le type d'interactions qu'ils doivent avoir avec leur unité de routage. Ce n'est que dans le chapitre 6, consacré à l'implémentation du circuit POEtic, que nous en montrerons une réalisation.

5.4 Premières solutions

Se basant sur les hypothèse énoncées, plusieurs solutions peuvent être réalisées. Nous en proposons trois dans cette section, par ordre croissant de complexité. Ces options n'ont pas été retenues pour une implémentation matérielle, de par leur manque de flexibilité et d'adaptabilité, caractéristiques indispensables quant à la réalisation de

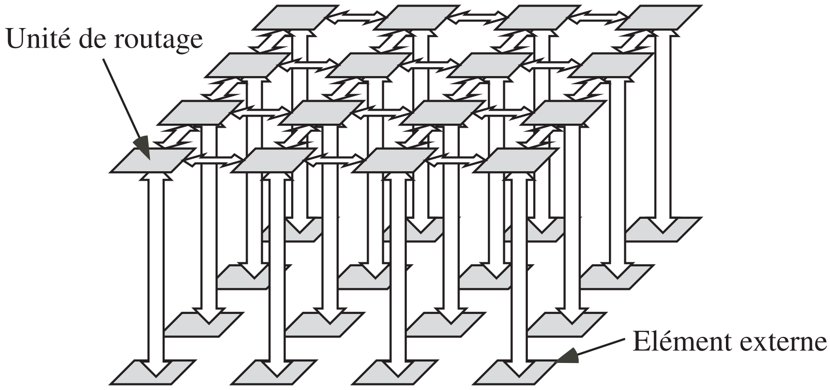


Figure 5.2 : *Le tableau d'unités de routage, connecté à un tableau d'éléments quelconques.*

systèmes bio-inspirés. Nous les présentons toutefois de manière à être le plus complet possible sur les potentielles solutions de routage distribué autonome. Ce cheminement vers une complexité croissante nous mènera tout naturellement vers les algorithmes HIDRA de la section suivante.

5.4.1 Algorithme direct

Le système de routage le plus simple consiste à envoyer une valeur dans une direction, la première destination rencontrée la récupérant (Figure 5.3). La réalisation matérielle d'une unité de routage, telle qu'explicitée à la figure 5.4 ne fait intervenir aucun contrôleur, les seuls bits envoyés par l'élément externe configurant directement les multiplexeurs.

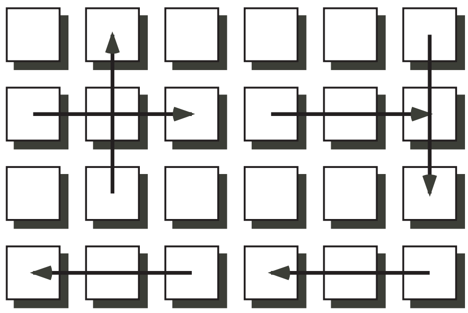


Figure 5.3 : *Dans le cas le plus simple, une connexion est une ligne droite entre la source et la destination.*

Une unité de routage transmet donc par défaut les signaux reçus dans la direction opposée, sauf si elle est une source. Dans ce cas, le signal provenant de son élément externe est transmis dans une direction particulière. Dans le cas d'une destination, la valeur reçue d'une certaine direction est transmise à l'élément externe (Figure 5.4). Sur le plan de l'implémentation matérielle, quatre multiplexeurs à deux entrées sélectionnent, pour chacune des directions, la valeur à envoyer. Par défaut, il s'agit de la valeur reçue de la voisine de direction opposée. Grâce à un démultiplexeur contrôlé par deux bits de direction, et sensible au fait que l'unité de routage est une source, au maximum un des multiplexeurs est configuré pour sélectionner la valeur envoyée par l'élément externe. Si, en revanche, l'unité de routage est une destination, les deux bits de direction permettent de récupérer la valeur venant d'une des quatre directions, via un multiplexeur à quatre entrées, et de la transmettre à l'élément externe.

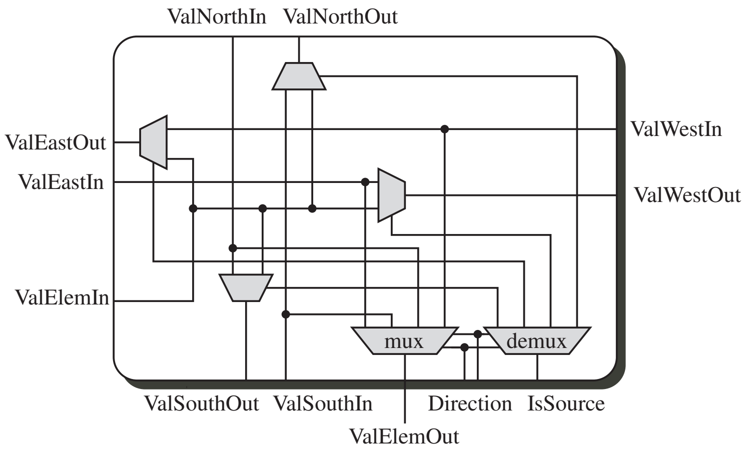


Figure 5.4 : *Une unité de routage ne permettant que de connecter des points sur la même ligne.*

Cette approche a l'avantage de ne nécessiter qu'un nombre très réduit de matériel par unité de routage. Il est également intéressant de noter qu'aucun processus de routage n'est nécessaire, puisqu'ici les signaux de configuration soumis par l'élément externe suffisent à configurer correctement les multiplexeurs. La réalisation d'un tel système serait adaptée à des applications de type automates cellulaires, où une cellule est toujours connectée à ses plus proches voisines. Toutefois, les connexions ne peuvent s'effectuer que sur des lignes verticales ou horizontales, sans qu'un seul virage ne soit possible. Une source émettant dans une direction X ne peut en effet toucher que des destinations présentes sur la même ligne, dans ladite direction. Son manque de flexibilité est donc flagrant, et elle n'a par conséquent pas été retenue pour une implémentation physique.

5.4.2 Algorithme à adressage relatif direct

Afin de pallier au faible potentiel de routage de la solution précédente, nous proposons d'introduire un système d'adressage relatif. Une source y contient deux valeurs, représentant respectivement un décalage en X et un en Y, utilisés pour joindre une destination. Nous avons ici l'introduction d'un processus de routage durant lequel une source cherche à joindre sa destination correspondante, en envoyant la valeur relative (x, y) à sa voisine présente dans la direction de la destination.

Pour l'implémentation d'un tel système, chaque unité de routage doit être composée de quatre multiplexeurs à 4 entrées pour les signaux à propager vers ses voisines, et un supplémentaire pour le signal à transmettre à l'élément externe. De plus, un contrôleur doit gérer le processus de routage afin de configurer de manière correcte les multiplexeurs. L'algorithme réalisé est relativement simple.

Premièrement, un mécanisme permettant d'établir une priorité entre les unités de routage est nécessaire. Une fois un maître élu, il envoie de manière sérielle les coordonnées (x, y) à sa voisine correspondant à la direction requise, telle décrite au tableau

5.1.

Règle	**Direction de propagation**
Si $x > 0$	Est
Sinon si $x < 0$	Ouest
Sinon si $y > 0$	Nord
Sinon si $y < 0$	Sud
Sinon	Élément externe

Tableau 5.1 : *Direction de propagation du routage, en fonction des coordonnées.*

L'unité de routage recevant ces coordonnées effectue alors une opération sur les coordonnées reçues. Cette incrémentation ou décrémentation dépend de l'origine de leur réception, et est décrite dans le tableau 5.2. Une seule opération est réalisée, et son résultat est directement utilisé afin de déterminer la direction de propagation décrite par le tableau 5.1.

Origine	**Opération**		
Nord	$x = x$	;	$y = y + 1$
Est	$x = x + 1$	;	$y = y$
Sud	$x = x$	;	$y = y - 1$
Ouest	$x = x - 1$	;	$y = y$

Tableau 5.2 : *Opération réalisée sur les coordonnées, en fonction de l'origine de celles-ci.*

Lorsque l'unité de routage obtient une coordonnée $(0, 0)$, c'est qu'elle est la destination du processus, et ce dernier se termine. La figure 5.5 présente trois chemins créés, chaque source contenant les coordonnées relatives de sa destination. Nous pouvons noter que l'algorithme à adressage relatif est plus souple que le direct, étant donné qu'il est possible d'y relier n'importe quelle unité de routage avec n'importe quelle autre. Toutefois, les chemins sont toujours construits de la même manière, en se positionnant d'abord sur l'axe X, puis en retrouvant la destination sur l'axe Y. De ce fait, une unité de routage se trouvant sur un chemin existant allant dans la même direction que la propagation ne peut être connectée. C'est pourquoi une variante à adressage relatif indirect se propose de pallier à ce manque de flexibilité.

5.4.3 Algorithme à adressage relatif indirect

L'idée de cette troisième solution est, comme pour la précédente, de connecter une source et une destination en fonction de leur adressage relatif. Le manque de flexibilité de l'adressage relatif direct vient du fait qu'un chemin possède au plus un virage. Nous proposons donc de permettre la création de chemins sinueux garantissant ainsi la réalisation du chemin s'il en existe un. Pour ce faire, deux modifications sont nécessaires. Tout d'abord, après avoir choisi un maître, celui-ci transmettra les coordonnées dans les quatre directions, et non dans une seule. De la même manière que précédemment, chaque unité de routage recevant ces coordonnées stocke l'origine de ces informations

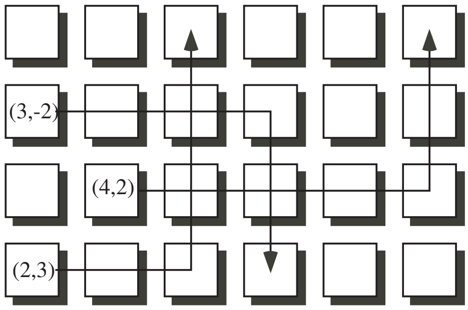


Figure 5.5 : *Trois chemins créés avec un adressage relatif direct.*

et effectue l'opération du tableau 5.2 puis compare le résultat à $(0,0)$. Si elle n'est pas la destination, elle propage les nouvelles coordonnées dans les quatre directions. Cette phase d'expansion se termine lorsque la destination est atteinte, et une phase de rétro-propagation partant de la destination permet de configurer les multiplexeurs de manière à créer le chemin désiré.

Nous n'entrons pas en détail dans la description de la phase d'expansion et de rétro-propagation, étant donné qu'elles seront décrites dans le cadre de la solution retenue. La figure 5.6 montre la création d'un chemin supplémentaire en comparaison de la figure 5.5. Ce chemin ne pouvait pas être créé avec l'algorithme relatif direct, alors que la variante indirecte le rend possible, au prix de deux virages.

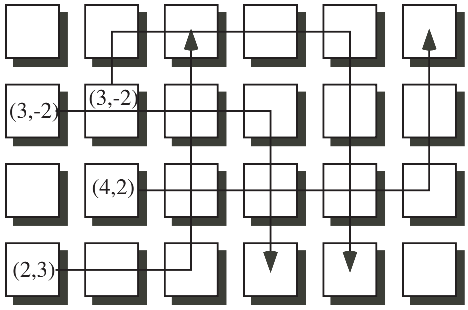


Figure 5.6 : *Quatre chemins créés avec un adressage relatif indirect.*

5.5 HIDRA

Nous venons de présenter trois solutions au problème du routage matériel distribué autonome, dont la troisième offre une bonne flexibilité dans le sens où un adressage relatif permet de relier deux unités de routage sans que le chemin créé ne doive être composé seulement de deux lignes droites. Toutefois, cette flexibilité manque encore d'adaptabilité, une destination devant connaître par avance la position exacte (même si relative), de sa source. Afin d'améliorer encore les performances de notre routage matériel, nous allons maintenant baser nos algorithmes sur un adressage par identifiants. Une source y possède un identifiant, et une destination y connaît l'identifiant de sa source. De cette manière, la position des deux unités de routage à relier n'importe pas, et n'a donc pas besoin d'être connue a priori.

Cet avantage est décisif dans l'optique d'implémenter des systèmes cellulaires adaptatifs bio-inspirés du type réseaux de neurones à topologie variable, ou du type système multicellulaire capable de croissance et d'autoréparation. En effet, si un nouveau neurone est ajouté à un réseau, il doit avoir la possibilité de se connecter à ceux déjà présents, et ce sans avoir de connaissance préalable sur leur position exacte. De plus, le projet POEtic, comme nous le verrons au chapitre 6, a vu la mise au point d'un nouveau circuit FPGA adaptatif contenant l'algorithme de routage que nous allons présenter, dont le fonctionnement a été décrit dans [234]. Dans le cas d'applications cellulaires, le placement des cellules sur le substrat électronique qu'est la partie reconfigurable du FPGA n'est donc plus important, la connexion de cellules par le réseau de routage n'étant pas dépendante de la position de celles-ci.

Le système à base d'identifiants est donc un excellent candidat pour l'implémentation distribuée autonome, chaque unité de routage étant consciente de son identifiant ou de l'identifiant de sa source. Aucun contrôle global n'est dès lors nécessaire pour déterminer deux unités de routage à connecter. Alors que dans les solutions précédemment évoquées (sections 5.4.2 et 5.4.3) les adresses correspondaient aux coordonnées relatives de la destination par rapport à la source, nous utiliserons à présent le terme "adresse" comme synonyme d'identifiant.

Une des particularités de HIDRA est que la création d'un chemin, c'est-à-dire le lancement d'un processus de routage, peut être effectué à n'importe quel instant, par une source ou par une destination. En effet, certaines applications, tels les processus ontogénétiques, nécessitent qu'une cellule cherche à se connecter à une cellule en attente, et donc qu'une source tente de trouver une destination libre. De même, pour un réseau de neurones à topologie variable, un nouveau neurone doit pouvoir choisir la provenance de ses entrées. Une destination doit donc être en mesure de partir à la recherche de sa source correspondante.

Avant de présenter l'algorithme HIDRA, nous allons brièvement expliquer son nom, composé de cinq lettres liées à des mots anglais, publication scientifique oblige :

- **H (Hardware) :** Nous nous intéressons à la réalisation matérielle (hardware) de circuits électroniques.
- **I (Incremental) :** Les algorithmes développées se doivent d'être incrémentaux, c'est-à-dire que la création de chemins de données doit pouvoir se faire à tout moment, sans qu'une connaissance de tous les chemins à créer ne soit nécessaire. De plus, la création d'un nouveau chemin ne doit en aucun cas en détruire un préexistant.
- **D (Distributed) :** Nos algorithmes sont distribués, c'est-à-dire qu'un ensemble d'unités de routage identiques est responsable de la création des chemins, sans qu'un contrôle global ne soit nécessaire.
- **R (Routing) :** Il s'agit bien évidemment de routage, où l'on désire configurer les multiplexeurs d'un switchbox qui sont ensuite utilisés pour la transmission d'information entre différents points du circuit.
- **A (Algorithm) :** Finalement, nous parlons d'un algorithme, servant à créer des chemins de données entre différents points d'un circuit électronique.

Le nom de notre algorithme étant maintenant explicité, nous le présentons dans la section suivante, avant de détailler l'architecture de l'unité de routage.

5.5.1 Algorithme

Dans le chapitre précédent, nous avons étudié l'implémentation matérielle distribuée de l'algorithme de Lee proposée par Moreno. Dans notre algorithme, la création d'un chemin d'une source à sa destination suit le même principe de fonctionnement correspondant à l'expansion d'une vague. Nous allons toutefois y ajouter la capacité des unités de routage de gérer elles-mêmes les processus de routage. HIDRA est principalement composé de cinq phases exécutées l'une à la suite de l'autre, que nous allons illustrer par l'exemple de la figure 5.7. Dans cette figure, ainsi que dans toutes les suivantes, la lettre "S" désigne une source, et la lettre "T" (Target) une destination.

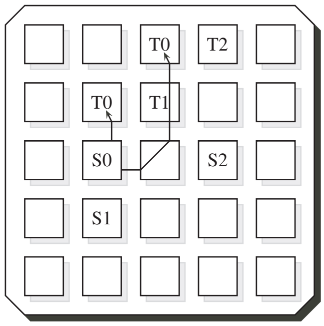


Figure 5.7 : *Exemple de routage à effectuer, avec des chemins déjà créés.*

Travaillant avec un voisinage de 4, nous utiliserons les points cardinaux Nord, Est, Sud, et Ouest pour désigner les positionnements relatifs des unités de routage. A titre d'exemple, dans la figure 5.7, la source S0 est au Nord de S1, et T2 est à l'Est de T0, et le numéro des unités de routage correspond à leur identifiant.

Phase 1 : Définition d'un maître

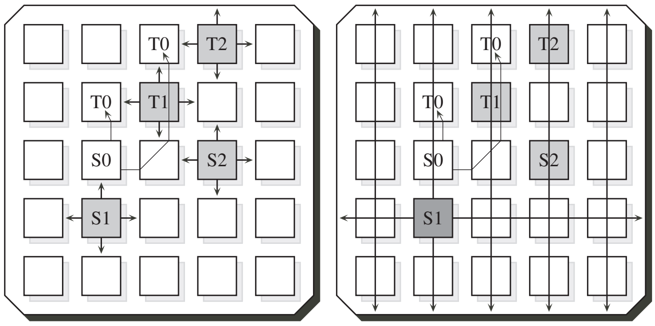
Tout d'abord, plusieurs unités de routage pouvant désirer se connecter à d'autres en même temps, il faut pouvoir régler ce litige. Pour ce faire nous utilisons une priorité donnée à l'unité de routage la plus au Sud-Ouest. Un dispositif simple a été développé, laissant plusieurs unités de routage placer un signal à '1', et permettant à chacune de savoir si elle a la priorité ou non. La priorité est gagnée par l'unité la plus au Sud n'ayant pas d'unité active à l'Ouest. Nous définissons la ligne implémentant ce système comme étant la ligne de propagation. Un seul coup d'horloge est nécessaire à cette première phase, la priorité étant définie de manière combinatoire.

La figure 5.8 montre la tentative de propagation de signal des quatre unités de routage à connecter, avec la victoire de celle placée le plus au Sud-Ouest.

A ce stade, nous venons d'introduire la ligne de propagation, qui permet aux unités de routage de transmettre un signal à toutes les autres, ainsi qu'à définir une priorité. Les unités de routage possèdent au total deux sorties dans chacune des directions, dont la première est cette ligne de propagation, et dont la deuxième sert à la phase d'expansion et de création du chemin.

Phase 2 : Envoi de l'identifiant

L'unité de routage ayant gagné la priorité envoie son identifiant à toutes les autres unités de routage en broadcast combinatoire, via la ligne de propagation. L'identifiant



(a) Tentative de devenir maître

(b) Une unité de routage devient maître

Figure 5.8 : *1re phase de HIDRA : Définition d'un maître.*

est envoyée de manière sérielle, en n coups d'horloge pour une adresse de n bits. Chaque unité de routage étant une source ou une destination la compare au sien, grâce à un comparateur sériel. Un trigger est chargé d'envoyer un signal à chaque unité de routage après les n coups d'horloge, période après laquelle chaque unité sait si elle possède ou non le même identifiant que le maître.

Phase 3 : Eliminations des concurrents

Après la comparaison des identifiants, plusieurs destinations ou sources peuvent se considérer comme participant au processus de routage courant, si elles possèdent le même identifiant que le maître. Deux cas peuvent se présenter :

- Si le maître est une source, il envoie un signal sur la ligne de propagation, et toutes les sources participantes sont désactivées. De cette manière, même lorsque plusieurs sources ont le même identifiant, seule celle ayant initié le processus de routage participe effectivement à celui-ci. Plusieurs destinations de même identifiant peuvent prendre part au processus de routage, et seule la première atteinte par la propagation sera effectivement connectée à la source.
  Dans le cas d'un système multicellulaire capable de croissance, par exemple, cette caractéristique est importante, et permet de disposer de plusieurs cellules totipotentes qui possèdent chacune une entrée (destination) en attente de connexion. Ayant toutes le même identifiant, il faut garantir qu'une cellule qui tente de se connecter à l'une d'elle pour y introduire un génome ne se connecte qu'à une et une seule de ces cellules.
- Si le maître est une destination, aucun signal n'est transmis via la ligne de propagation, et toutes les destinations participantes sont désactivées. Seule la destination ayant initié le routage sera effectivement connectée, permettant à d'autres destinations de même identifiant de se connecter ultérieurement. Plusieurs sources possédant le même identifiant peuvent participer au processus de routage, et seule la première à atteindre la destination lui sera connectée.

La figure 5.9 montre la propagation du signal à '1' de la part de la source.

Cette phase est cruciale afin d'assurer que seule l'unité de routage ayant lancé le processus et ayant gagné la priorité sera connectée. Plusieurs applications nécessitent

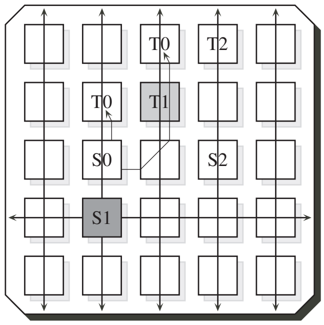


Figure 5.9 : *3^{e}phase de HIDRA : Désactivation des concurrents.*

effectivement cette caractéristique, dont notamment les processus ontogénétiques, que nous décrirons en page 224. Les sources y lancent des routages, alors que plusieurs sources ont le même identifiant. Il faut donc pouvoir garantir que seule l'unité maître ne se connecte.

Dans le cas de neurones, par exemple, l'ajout d'une nouvelle cellule implique qu'elle connecte ses entrées à la sorties des neurones auxquels elle doit être reliée. Pour ce faire, elle cherche, pour chaque entrée, la source la plus proche qui possède le même identifiant que son entrée, de manière à récupérer la sortie d'un autre neurone.

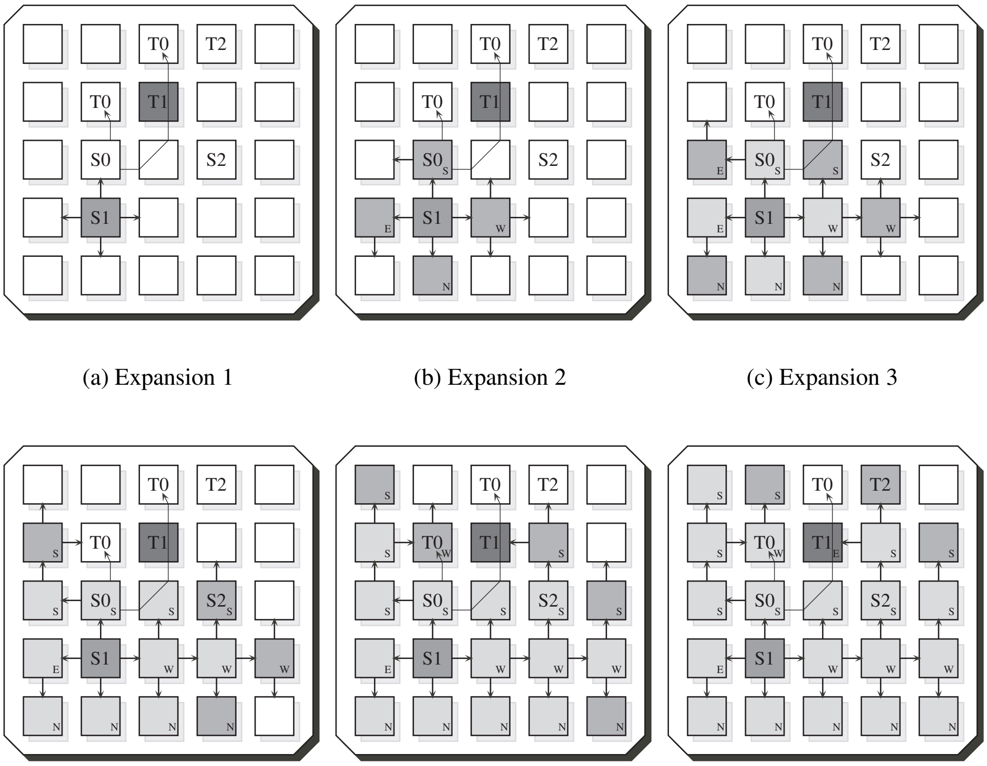
Phase 4 : Expansion

Après que les unités de routage concurrentes du maître aient été inhibées, les sources participantes lancent l'expansion. Cette phase est semblable à l'algorithme de Lee-Moore, et à l'implémentation qu'en a fait Moreno. Le front d'onde est propagé, en partant des sources actives, à chaque coup d'horloge, jusqu'à atteindre une destination. Chaque source participante envoie un signal à chacune de ses voisines, pour autant que le multiplexeur correspondant ne soit pas configuré de manière à sélectionner un signal autre que celui venant de l'élément externe. De par ce fait, un chemin existant ne peut être détruit, et la possibilité de réutiliser un chemin déjà créé est conservée.

Chaque autre unité de routage attend qu'une de ses entrées soit activée, et lorsqu'une ou plusieurs entrées sont activées, une priorité est donnée au Nord, à l'Est, au Sud, puis finalement à l'Ouest, et l'origine du signal est stockée dans deux bascules. Une fois l'origine stockée, c'est-à-dire après un coup d'horloge, l'unité de routage propage l'expansion en activant les sorties vers ses voisines, pour autant que le multiplexeur correspondant ne soit pas configuré en sélectionnant une autre entrée que l'origine de l'expansion.

Cette phase nécessite un nombre de coups d'horloge égal à la plus courte distance entre la source et la destination prenant en compte les chemins déjà créés.

La figure 5.10 montre l'entièreté de la phase d'expansion permettant de relier la source 1 à la destination 1. Les nuances de gris permettent d'identifier les différentes unités de routage. La plus foncée est la destination à atteindre, la nuance suivante indique les unités de routage sur le front d'onde, c'est-à-dire qui vont étendre la vague au coup d'horloge suivant, et les plus claires sont les unités qui ont déjà mémorisé leur origine et propagé la vague. La lettre présente dans le coin inférieur droit des unités atteintes par la vague indique l'origine de l'expansion avec les premières lettres anglaises des quatre points cardinaux.



(a) Expansion 1 (b) Expansion 2 (c) Expansion 3

(d) Expansion 4 (e) Expansion 5 (f) Expansion 6

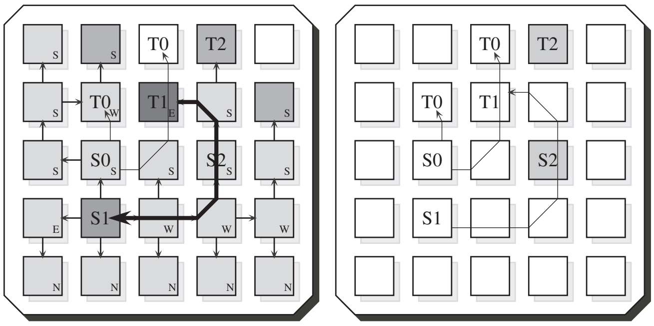
Figure 5.10 : *4[e] phase de HIDRA : Propagation du front d'onde.*

Phase 5 : Création du chemin

Lorsque la destination est atteinte par l'expansion, elle active la ligne de propagation au coup d'horloge suivant, signalant ainsi la fin du processus de routage. Si plusieurs destinations sont atteintes au même instant, la ligne de propagation permet d'établir une priorité, et seule la destination la plus au Sud-Ouest crée effectivement un chemin. La destination sélectionnée envoie alors un signal actif en direction de l'origine de l'expansion, qu'elle a précédemment stockée. L'unité de routage touchée par ce signal le propage également en direction de son origine, et cette rétro-propagation s'effectue jusqu'à arriver à la source. En parallèle à cette rétro-propagation, les unités de routage touchées configurent le multiplexeur correspondant au sens de leur expansion de manière à créer le chemin entre la source et la destination. Une fois le chemin créé, soit après un coup d'horloge, toutes les unités de routage se retrouvent dans leur état initial, prêtes à recommencer un nouveau processus de routage, ou à simplement œuvrer à la transmission d'information.

La figure 5.11 présente la création du chemin, où en un coup d'horloge un signal est propagé de la destination à la source, fixant ainsi les multiplexeurs présents sur le chemin.

L'algorithme 5.1 illustre les cinq phases de HIDRA, d'un point de vue global. Nous allons maintenant nous intéresser aux unités de routage, qui sont des composantes locales, sur le plan de leur fonctionnement et de leur implémentation.



(a) Rétropropagation (b) Exploitation du routage effectué.

Figure 5.11 : *5^e phase de HIDRA : Création du chemin.*

Algorithme 5.1 Algorithme HIDRA

Tant que vrai **Faire**
- **Si** au moins une unité de routage (RU) veut se connecter **alors**
  - Le maître est la RU la plus au Sud n'ayant pas de RU désirant se connecter à l'Ouest
  - **Pour** i=1 à taille de l'identifiant **Faire**
    - Le maître envoie le i$^{\text{ème}}$ bit de son identifiant à tous les RUs
    - Les RUs le comparent avec le i$^{\text{ème}}$ bit de leur propre identifiant
  - **Fin faire**
  - Les RUs de même identifiant participent au routage
  - **Si** Le maître est une source **alors**
    - Aucune autre source ne participe
  - **Sinon**
    - Aucune autre destination ne participe
  - **Fin si**
  - Les sources participantes sont sur le front d'onde
  - **Tant que** Une destination participante n'est pas atteinte **Faire**
    - Les RUs sur le front d'onde s'étendent
    - Les RUs atteintes par le front d'onde stockent son origine
    - Les RUs atteintes par le front d'onde deviennent le front d'onde
    - **Si** Le front d'onde est vide **alors**
      - Problème de congestion, le chemin ne peut être trouvé
    - **Fin si**
  - **Fin tant que**
  - Parcourir les RUs de la destination à la source, en configurant les multiplexeurs
- **Fin si**

Fin tant que

5.5.2 Unité de routage

L'algorithme HIDRA ayant été explicité, nous pouvons nous intéresser à la structure d'une unité de routage permettant son implémentation distribuée. Après avoir présenté sa structure globale, nous allons détailler ses composants, et plus particulièrement son contrôleur.

Structure Globale

Une unité de routage est principalement composée d'un switchbox, d'un contrôleur, d'un comparateur d'adresse, et d'une unité de propagation, comme le montre la figure 5.12. Nous pouvons y observer la présence d'un multiplexeur sélectionnant la valeur à transmettre dans chaque direction, permettant au contrôleur d'envoyer un signal venant de lui durant les processus de routage, ou d'être utilisé de manière à laisser le switchbox sélectionner la valeur à transmettre lors de la phase d'exécution normale. Ceci implique que le niveau de routage distribué ne peut être utilisé durant un processus de routage, et fonctionne donc sur deux modes : en cours de routage, ou en transmission d'information. Il serait possible aux deux modes d'être actifs simultanément à condition d'ajouter une sortie dans chacune des directions, et de supprimer ces quatre multiplexeurs. Étant donné que la minimisation du nombre de liaisons entre unités de routage fut de la plus grande importance, nous avons préféré la solution multiplexée, mais rien n'empêcherait de modifier cette implémentation pour faire fonctionner ces deux modes simultanément. Dans le cas où le fonctionnement des éléments externes doit être inhibé pendant les processus de routage, le signal `IsRouting`, en sortie de l'unité de routage, indique si un routage est en cours ou non.

Intéressons-nous tout d'abord aux entrées de l'unité de routage :

- `Clk` : Une horloge, identique pour toutes les unités.
- `Rst` : Un reset, également identique pour toutes.
- `AddrIn` : Un bit de l'identifiant envoyé par l'élément externe.
- `IsSource` : Ce signal, envoyé par l'élément externe, indique si l'unité de routage est une source ou non.
- `IsTarget` : Ce signal, envoyé par l'élément externe, indique si l'unité de routage est une destination ou non.
- `StartRouting` : Ce signal, envoyé par l'élément externe, indique s'il désire se connecter ou non. Dans le cas d'une activation, un routage sera lancé par l'unité de routage tant qu'elle n'est pas connectée.
- `TriggerIn` : Ce signal est utilisé pour détecter la fin de comparaison d'un identifiant. Il envoie un '1' tous les n coups d'horloge pour lesquelles la sortie ReadAddr est active, et ce pour un identifiant de n bits.
- `ValInExt` : Valeur envoyée par l'élément externe lorsque l'unité de routage est une source.
- `ValInN, ValInE, ValInS, ValInW` : Un signal venant de chaque voisine sert à transmettre une valeur durant le fonctionnement normal du système, ou à gérer le déroulement de l'algorithme lors d'un processus de routage.
- `PropInN, PropInE, PropInS, PropInW` : Un signal venant de chaque voisine permet de transmettre un signal à toutes les unités de routage du circuit ainsi qu'à introduire une priorité entre les unités de routage (ligne de propagation).

Et à ses sorties :

- `ReadAddr` : Indique à l'élément externe qu'il doit fournir un nouveau bit d'identifiant et active également le trigger.

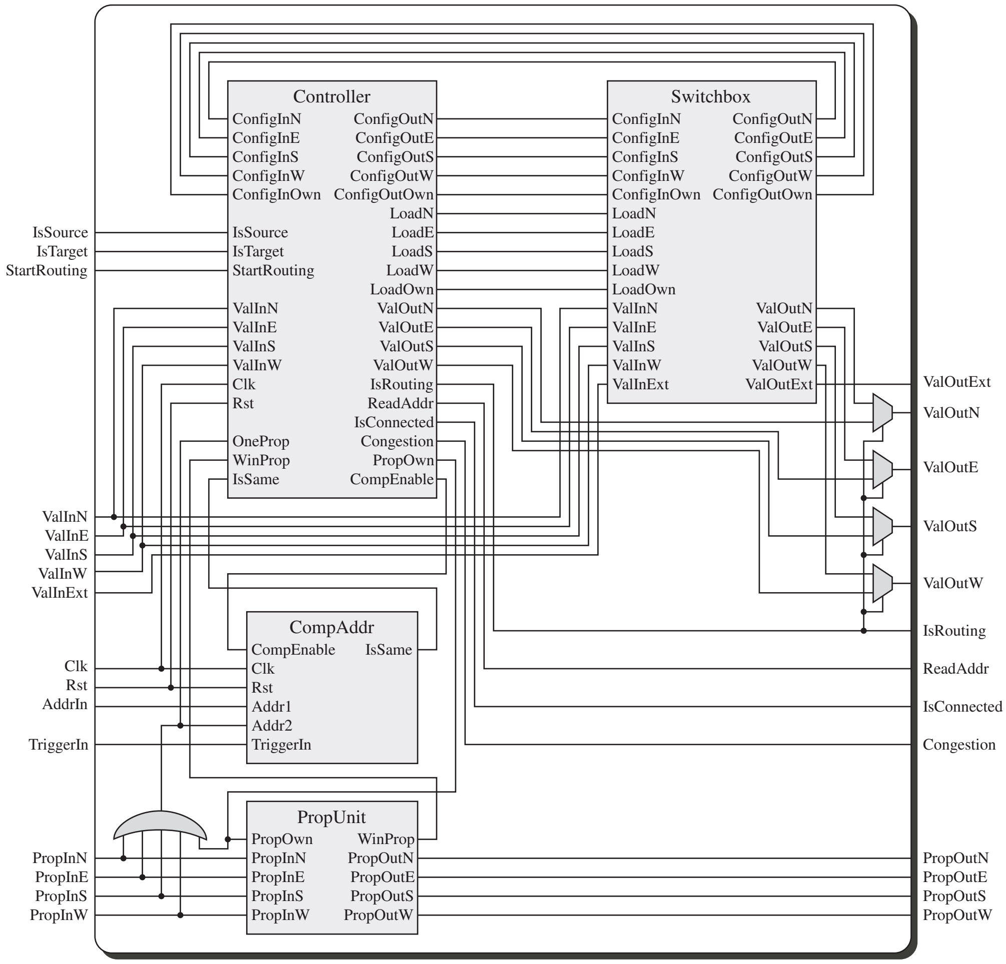


Figure 5.12 : *Schéma général d'une unité de routage, composée de quatre parties : un switchbox, un contrôleur, un comparateur d'adresse, et une unité de propagation.*

- `Congestion` : Indique une potentielle congestion. Au niveau du circuit, une combinaison de ces signaux reçus par toutes les unités de routage permet de savoir si l'algorithme est bloqué ou non.
- `IsRouting` : Indique si un processus de routage est en cours ou non.
- `IsConnected` : Indique à l'élément externe si l'unité de routage est connectée à son correspondant.
- `ValOutExt` : Valeur envoyée à l'élément externe lorsque l'unité de routage est une destination et qu'elle a été connectée.
- `ValOutN, ValOutE, ValOutS, ValOutW` : Valeurs envoyées à chacune des voisines, servant à transmettre une valeur durant le fonctionnement normal du système, ou à gérer le déroulement de l'algorithme lors d'un processus de routage.
- `PropOutN, PropOutE, PropOutS, PropOutW` : Signal envoyé à chaque voisine permettant de transmettre un signal à toutes les unités de routage du circuit ainsi qu'à introduire une priorité entre les unités de routage (ligne de propagation).

Switchbox

Commençons notre description des composants par celui permettant la transmission des signaux dans la direction adéquate. Le switchbox contient cinq multiplexeurs à quatre entrées : un dans chacune des directions, ainsi qu'un dernier dirigé vers l'élément externe. Chaque multiplexeur est contrôlé par deux bits de configuration, qui sont modifiés par le contrôleur de l'unité de routage. Un bit supplémentaire est également ajouté aux quatre premiers multiplexeurs, indiquant si le multiplexeur a été configuré ou non. Il permet au contrôleur de vérifier l'existence d'un chemin précédemment créé, de manière à ne pas le détruire.

La figure 5.13 montre la manière dont les multiplexeurs sont reliés aux entrées et sorties. Pour chacune des quatre directions, la sélection se fait parmi les trois autres voisines et l'élément externe, alors que la sortie vers l'élément externe est choisie entre les quatre voisines.

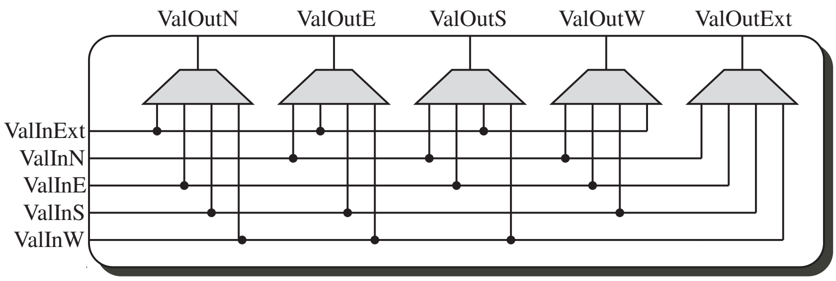


Figure 5.13 : *Schéma d'un switchbox, ne représentant que la connectique des multiplexeurs.*

La figure 5.14 donne le schéma d'un des quatre premiers multiplexeurs, sur lequel nous pouvons distinguer les bascules nécessaires au stockage de la configuration d'un multiplexeur. Cette illustration représente le multiplexeur responsable de sélectionner la valeur à envoyer au Sud. Il peut choisir parmi les valeurs reçues par les trois voisines Nord, Est, et Ouest, ou la valeur envoyée par l'élément externe. Les signaux `LoadS` et `ConfigInS` sont envoyés par le contrôleur de l'unité de routage, qui récupère `ConfigOutS`, dont il a besoin pour la bonne marche de la phase d'expansion de l'algorithme. Nous pouvons d'ores et déjà noter que les éléments nécessaires à la réalisation d'un switchbox sont 5 multiplexeurs à 4 entrées et 14 bascules [1].

Unité de propagation

L'unité de propagation, présente dans chaque unité de routage, sert à transmettre une valeur en broadcast à toutes les autres unités de routage, et à définir une priorité lors de la phase 1 de l'algorithme. L'idée est ici de pouvoir transmettre un '1' à travers tout le circuit tout en étant capable de discerner si une unité de routage se trouve la plus au Sud-Ouest.

Pour ce faire, nous disposons d'une entrée et d'une sortie par côté. Il ne reste qu'à définir la manière dont une entrée active est traitée, c'est-à-dire à quelle voisine elle doit être transmise. La figure 5.15 montre le sens de transmission des signaux dans le

[1] Le bon fonctionnement du contrôleur ne nécessite pas de savoir si le multiplexeur de l'élément externe est configuré ou non.

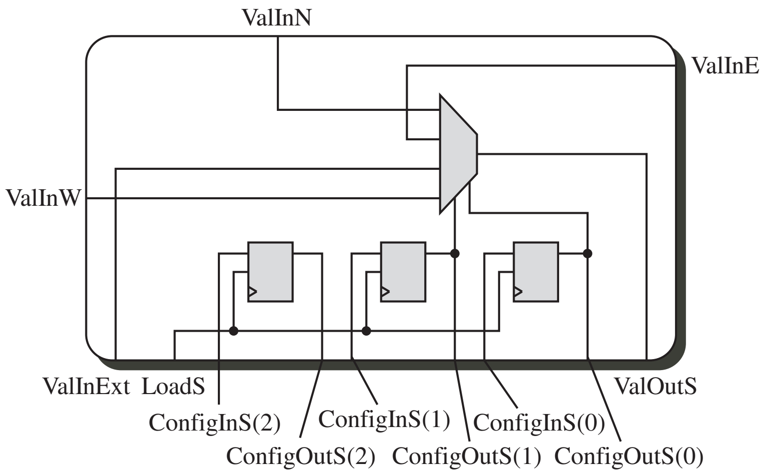


Figure 5.14 : *Schéma représentant un multiplexeur et ses trois bits de configuration.*

cas où plusieurs unités de routages placent un '1' sur la ligne de propagation. Nous pouvons observer que l'élément le plus au Sud-Ouest est le seul n'ayant aucune entrée à '1', car il a pu gagner la priorité sur ceux placés plus au Nord ou plus à l'Est de lui.

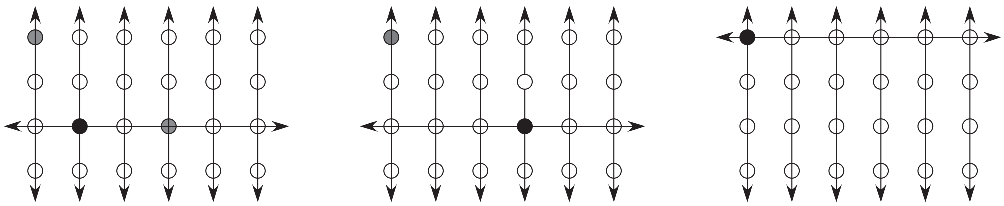


Figure 5.15 : *Ces trois figures illustrent la direction de propagation d'un signal sur la ligne de propagation. L'unité noire constate qu'elle est le maître alors que les grisées, qui tentent également de l'être, ne peuvent que constater leur échec.*

Le tableau 5.3 définit la transmission du signal de propagation permettant d'obtenir ce comportement. Le signal PropOwn est géré par le contrôleur de l'unité de routage, qui le place à '1' pour propager, tandis que WinProp indique si le contrôleur a gagné la priorité, c'est-à-dire est le plus au Sud-Ouest.

La figure 5.16 illustre deux schémas correspondant au comportement du tableau 5.3. Celui du haut n'est pas spécialement optimisé, mais présente bien la priorité donnée aux signaux provenant du Sud, de l'Ouest, du contrôleur, de l'Est, puis du Nord. Celui du bas, en revanche, offre un délai moins long au travers des portes logiques, et sera donc un meilleur candidat pour une réalisation matérielle, où la longueur des chemins combinatoires doit être réduite au maximum.

Comparateur d'adresse

Dans la phase 2 de l'algorithme HIDRA, le contrôleur vérifie si l'identifiant reçu correspond à sa propre adresse. Pour ce faire, un comparateur sériel a été implémenté. Outre une horloge et un reset, il possède quatre entrées : deux bits d'adresse à comparer, un *enable* autorisant la comparaison, et un trigger signifiant la fin de la comparai-

Règle	**Sorties actives**
Si `PropInS`='1'	Nord
Sinon si `PropInW`='1'	Nord, Sud, Est
Sinon si `PropOwn`='1'	Nord, Est, Sud, Ouest, WinProp
Sinon si `PropInE`='1'	Nord, Sud, Ouest
Sinon si `PropInN`='1'	Sud

Tableau 5.3 : *Direction de transmission du signal de propagation, en fonction de son origine.*

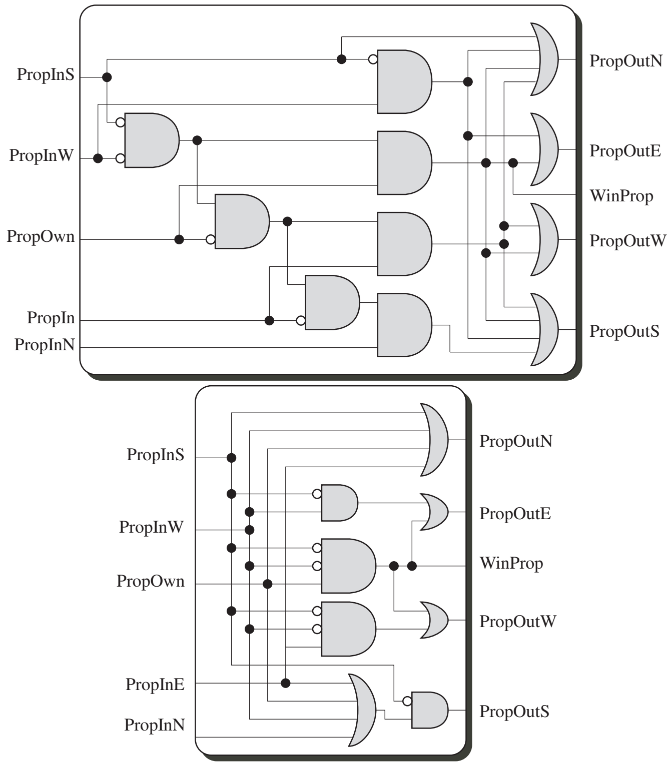


Figure 5.16 : *Deux schémas possibles d'une unité de propagation.*

son. Une seule sortie sert à indiquer si la comparaison a présenté une égalité des deux adresses ou non.

Sur le plan de l'implémentation, une seule bascule est nécessaire, de manière à mémoriser si au moins un bit comparé a été différent. Sa valeur vaut '1' après un reset, ou après la fin d'une comparaison d'adresse, et passe à '0' lorsque les deux bits d'adresse passés en entrée ne sont pas identiques, lorsque l'*enable* de la bascule est à '1'. La sortie du comparateur est à '1' si la bascule est à '1', et que les deux bits d'adresse courants sont identiques. La figure 5.17 montre son schéma.

Contrôleur

Nous en arrivons à la partie centrale de l'unité de routage, à savoir le contrôleur. Le choix a été fait de l'implémenter à l'aide d'une machine d'états finis, de manière

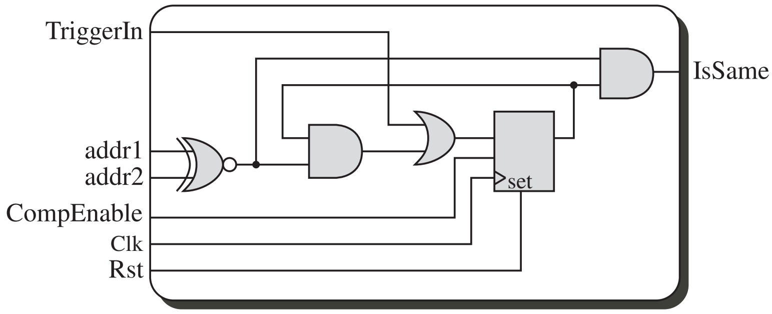


Figure 5.17 : *Schéma d'un comparateur sériel.*

à en rendre le développement et la compréhension les plus faciles possibles. Il s'agit d'une machine de Mealy (Figure 5.18), où les sorties dépendent de l'état du contrôleur et de ses entrées [2].

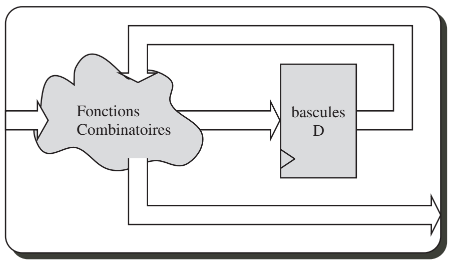


Figure 5.18 : *Le contrôleur est implémenté grâce à une machine de Mealy.*

Concernant les entrées/sorties du contrôleur, outre celles de l'unité de routage (page 111), les suivantes lui permettent de contrôler les multiplexeurs du switchbox.

En sortie :

- `LoadN`, `LoadE`, `LoadS`, `LoadW`, `LoadOwn` : forcent le chargement des bascules de contrôle de chacun des multiplexeurs.
- `ConfigOutN`, `ConfigOutE`, `ConfigOutS`, `ConfigOutW`, `ConfigOutOwn` : pour chacun des multiplexeurs, les trois bits de configuration à charger, sauf `ConfigOutOwn` qui n'en compte que deux.

Et en entrée :

- `ConfigInN`, `ConfigInE`, `ConfigInS`, `ConfigInW`, `ConfigInOwn` : pour chacun des multiplexeurs, les trois bits de configuration (sauf `ConfigOutOwn` qui n'en compte que deux), qui sont utilisés lors de la phase d'expansion de l'algorithme.

Six états, que nous appelerons `sInit`, `sAddress`, `sChooseSource`, `sWaitExp`, `sFrontExp`, et `sHasExp`, sont nécessaires à la bonne marche de l'algorithme HIDRA décrit précédemment. Le graphe de transitions de la machine d'états est illustré à la figure 5.19, et nous allons passer en revue ses six états en présentant dans les encarts le pseudo-code correspondant. Une description "françisée" est également fournie, afin de le rendre le plus clair possible.

Mais avant de commencer, précisons qu'en plus des bascules nécessaires à la réa-

[2]Le fait qu'il s'agisse d'une machine de Mealy implique des chemins combinatoires qui traversent potentiellement tout le circuit.

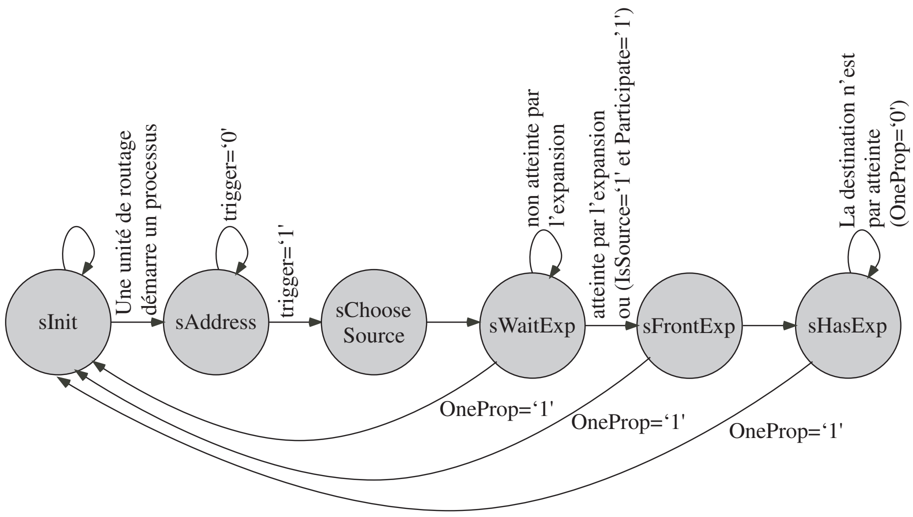


Figure 5.19 : *Machine d’états du contrôleur.*

lisation de la machine d’états, cinq autres sont indispensables à la bonne marche de l’algorithme :

- `Participate` : indique si l’unité de routage doit participer au processus de routage courant, c’est-à-dire est la source ou la destination courante.
- `IsMaster` : indique si l’unité de routage est celle qui lance le processus. Elle passe à ’1’ lors de la première phase de l’algorithme, lorsqu’un maître est choisi en fonction de la priorité Sud-Ouest.
- `IsConnected` : lorsque l’unité de routage est une source ou une destination, indique si elle est connectée à sa correspondante.
- `Origin` : sur 2 bits, permet de stocker l’origine de la phase d’expansion, de manière à pouvoir effectuer la phase de rétro-propagation.

De même, ces quelques signaux sont utiles à la compréhension des sections suivantes :

- `PropOwn` : signal géré par le contrôleur, relié à la ligne de propagation, et qui lui permet de l’activer.
- `WinProp` : indique si le contrôleur est prioritaire sur la ligne de propagation.
- `OneProp` : indique si au moins une des entrées de propagation ou PropOwn est active.
- `ValIn` : indique si au moins une des valeurs `ValIn`*X* est à ’1’.
- `CompEnable` : Autorise la comparaison d’adresses.
- `IsSame` : indique si les deux adresses comparées sont identiques ou non.
- `WantConnect` : indique si l’unité de routage doit encore tenter de se connecter à sa correspondante.

Concernant les signaux séquentiels, nous placerons le préfixe `Next` devant le nom du signal pour indiquer la valeur qu’il prendra au coup d’horloge suivant (Exemple : `NextState`, `NextIsMaster`). Nous partons du principe que si aucune affectation n’est effectuée, la valeur de la bascule reste inchangée.

sInit Après un Reset, la machine d’états est placée dans son état initial. Les actions qui y sont prises sont décrites par la partie d’algorithme 5.2. Le contrôleur reste dans cet état tant qu'aucune unité de routage ne désire lancer un processus de routage. Le lancement est signalé par un passage à '1' de la ligne de propagation, et dans ce cas, tous les contrôleurs passent dans l'état sAddress, et la bascule `IsMaster` passe à '1' pour le contrôleur ayant gagné la priorité.

Algorithme 5.2 HIDRA : Contrôleur de l'unité de routage, état sInit

```
1: Si state = sInit alors
2:    PropOwn ⇐ WantConnect AND StartRouting ;
3:    NextIsMaster ⇐ WinProp ;
4:    Si OneProp alors
5:       NextState ⇐ sAddress ;
6:       IsRouting ⇐ '1' ;
7:    Fin si
8:    NextParticipate ⇐ '0' ;
9: Fin si
```

sAddress Durant la deuxième phase de l'algorithme, tous les contrôleurs se trouvent dans l'état sAddress (partie d'algorithme 5.3). Ils y restent jusqu'à ce que le signal `TriggerIn` passe à '1'. Le maître envoie sériellement son identifiant (lignes 12-14), et les autres la comparent au leur (ligne 15). Lorsque `TriggerIn` s'active, tous les contrôleurs passent dans l'état sChooseSource, et ceux ayant le même identifiant que le maître font passer la bascule `Participate` à '1' (lignes 19-21).

Algorithme 5.3 HIDRA : Contrôleur de l'unité de routage, état sAddress

```
10: Si state = sAddress alors
11:    IsRouting ⇐ '1' ;
12:    Si IsMaster alors
13:       PropOwn ⇐ AddrIn ;
14:    Fin si
15:    CompEnable ⇐ '1' ;
16:    ReadAddr ⇐ '1' ;
17:    Si TriggerIn alors
18:       NextState ⇐ sChooseSource ;
19:       Si IsSame AND (WantConnect OR IsSource) alors
20:          NextParticipate ⇐ '1' ;
21:       Fin si
22:    Fin si
23: Fin si
```

sChooseSource Le but de l'état sChooseSource (partie d'algorithme 5.4), correspondant à la troisième phase de l'algorithme, est de désactiver la participation de certaines unités de routage. Si le maître est une source, il place un '1' sur la ligne de propagation (lignes 26-28), et les autres unités de routage ayant `Participate` à '1', c'est-à-dire ayant le même identifiant, et étant également des sources, désactivent leur participation (lignes 29-30). De même, si le maître est une destination, la ligne de propagation reste à '0', et les autres destinations placent `Participate` à '0'. Les contrôleurs ne restent qu'un coup d'horloge dans cet état, et passent directement à l'état sWaitExp.

sWaitExp La phase d'expansion de l'algorithme débute en voyant tous les contrôleurs passer dans l'état sWaitExp (partie d'algorithme 5.5). Les sources participantes n'y effectuent aucun traitement, mais passent directement à l'état sFrontExp (lignes 39-40). Tous les autres contrôleurs attendent d'avoir une de leurs entrées `ValIn`*X* à

Algorithme 5.4 HIDRA : Contrôleur de l'unité de routage, état sChooseSource

```
24: Si state = sChooseSource alors
25:    IsRouting ⇐ '1' ;
26:    Si IsMaster AND IsSource alors
27:       PropOwn ⇐ '1' ;
28:    Fin si
29:    Si OneProp AND not(IsMaster) AND IsSource alors
30:       NextParticipate ⇐ '0' ;
31:    Fin si
32:    Si not(OneProp) AND not(IsMaster) AND IsTarget alors
33:       NextParticipate ⇐ '0' ;
34:    Fin si
35:    NextState ⇐ sWaitExp ;
36: Fin si
```

'1'. Lorsque c'est le cas, une priorité est donnée dans l'ordre Nord, Est, Sud, Ouest pour le calcul de l'origine de l'expansion. Cette origine est stockée, et le contrôleur passe dans l'état sFrontExp.

Algorithme 5.5 HIDRA : Contrôleur de l'unité de routage, état sWaitExp

```
37: Si state = sWaitExp alors
38:    IsRouting ⇐ '1' ;
39:    Si IsSource and Participate alors
40:       NextState ⇐ sFrontExp ;
41:    Sinon
42:       Si ValInN alors
43:          NextOrigin ⇐ "00" ;
44:          NextState ⇐ sFrontExp ;
45:       Sinon si ValInE alors
46:          NextOrigin ⇐ "01" ;
47:          NextState ⇐ sFrontExp ;
48:       Sinon si ValInS alors
49:          NextOrigin ⇐ "10" ;
50:          NextState ⇐ sFrontExp ;
51:       Sinon si ValInW alors
52:          NextOrigin ⇐ "11" ;
53:          NextState ⇐ sFrontExp ;
54:       Sinon
55:          Congestion ⇐ '1' ;
56:       Fin si
57:       Si OneProp alors
58:          NextState ⇐ sInit ;
59:       Fin si
60:    Fin si
61: Fin si
```

sFrontExp Une unité de routage se retrouve dans l'état sFrontExp (partie d'algorithme 5.6) après avoir été atteinte par l'expansion, ou s'il s'agit d'une source participante. Elle n'y reste qu'un seul coup d'horloge, et passe directement à l'état sHasExp. S'il s'agit d'une source participante, le contrôleur place un '1' en valeur de sortie `ValOut`*X*, si l'une de ces deux conditions est remplie : le multiplexeur correspondant à la sortie n'est pas configuré, ou il l'est en sélectionnant la valeur de l'élément externe. De ce fait, aucun chemin existant ne peut être effaçé, et il est possible que le nouveau chemin en réutilise un déjà créé et partant de la même source (lignes 71-98). Si l'unité de routage n'est pas une source participante, elle active ses sorties `ValOut`*X* si l'une des deux conditions suivantes est remplie : le multiplexeur correspondant à la sortie n'est pas configuré, ou il l'est en sélectionnant la valeur venant de l'origine de l'expansion. Là encore, un chemin existant ne peut être effacé, mais il est possible de suivre un chemin connecté à la source participante.

Algorithme 5.6 HIDRA : Contrôleur de l'unité de routage, état sFrontExp

```
Si state = sFrontExp alors
  IsRouting ⇐ '1' ;
  Si OneProp alors
    NextState ⇐ sInit ;
    Congestion ⇐ '1' ;
  Sinon
    Congestion ⇐ '0' ;
    NextState ⇐ sHasExp ;
    Si IsSource AND Participate alors
      Si ConfigInN(2) alors
        Si ConfigInN(1..0)="00" alors
          ValOutN ⇐ '1' ;
        Fin si
      Sinon
        ValOutN ⇐ '1' ;
      Fin si
      Si ConfigInE(2) alors
        Si ConfigInE(1..0)="01" alors
          ValOutE ⇐ '1' ;
        Fin si
      Sinon
        ValOutE ⇐ '1' ;
      Fin si
      Si ConfigInS(2) alors
        Si ConfigInS(1..0)="10" alors
          ValOutS ⇐ '1' ;
        Fin si
      Sinon
        ValOutS ⇐ '1' ;
      Fin si
      Si ConfigInW(2) alors
        Si ConfigInW(1..0)="11" alors
          ValOutW ⇐ '1' ;
        Fin si
      Sinon
        ValOutW ⇐ '1' ;
      Fin si
    Sinon
      ValOutN ⇐ (ConfigInN(2)='0') OR (Origin=ConfigInN(1..0)) ;
      ValOutE ⇐ (ConfigInN(2)='0') OR (Origin=ConfigInE(1..0)) ;
      ValOutS ⇐ (ConfigInN(2)='0') OR (Origin=ConfigInS(1..0)) ;
      ValOutW ⇐ (ConfigInN(2)='0') OR (Origin=ConfigInW(1..0)) ;
    Fin si
  Fin si
Fin si
```

sHasExp Après avoir passé dans l'état sFrontExp, un contrôleur se retrouve dans l'état sHasExp (partie d'algorithme 5.7), en attendant la création du chemin par la destination atteinte. Si l'unité de routage est une destination participante, elle place un '1' sur la ligne de propagation (lignes 109-111). Étant donné qu'il est possible que plusieurs destinations soient atteintes au même instant, la priorité de la ligne de propagation permet de ne créer un chemin que pour la vainqueur. Le contrôleur prioritaire place alors un '1' à la sortie `ValOutX` correspondant à l'origine qu'il a stockée (lignes 118-124), et chaque contrôleur recevant une entrée `ValInX` active fait de même. De ce fait, en un coup d'horloge toutes les unités de routage sur le chemin entre la destination et la source sont touchées par ce signal de rétro-propagation. Le contrôleur touché configure alors le multiplexeur sélectionnant la valeur en direction de la destination, en fonction de l'origine qu'il a précédemment stockée (lignes 126-145). Au coup d'horloge suivant la phase de rétro-propagation, tous les contrôleurs se retrouvent dans l'état sInit, prêts à relancer un nouveau processus de routage.

Les deux bits de poids faibles chargés dans la configuration des multiplexeurs dépendent du fait que l'unité de routage est une source participante ou non. Si tel est le cas, le multiplexeur est configuré de façon à sélectionner la valeur reçue de l'élément externe, alors que dans le cas contraire, ce sont les deux bits `Origin` que le sont.

Algorithme 5.7 HIDRA : Contrôleur de l'unité de routage, état sHasExp

```
107: Si state = sHasExp alors
108:   IsRouting ⇐ '1' ;
109:   Si Participate AND IsTarget alors
110:     PropOwn ⇐ '1' ;
111:   Fin si
112:   Si WinProp alors
113:     NextIsConnected ⇐ '1' ;
114:   Fin si
115:   Si OneProp alors
116:     Si IsSource and Participate alors
117:       NextIsConnected ⇐ '1' ;
118:     Sinon si ValIn OR WinProp alors
119:       switch (origin)
120:         "00" ⇒ ValOutN ⇐ '1' ;
121:         "01" ⇒ ValOutE ⇐ '1' ;
122:         "10" ⇒ ValOutS ⇐ '1' ;
123:         "11" ⇒ ValOutW ⇐ '1' ;
124:       end switch
125:     Fin si
126:     Si ValInN alors
127:       Load0 ⇐ '1' ;
128:       ConfigOut0(2) ⇐ '1' ;
129:     Fin si
130:     Si ValInE alors
131:       Load1 ⇐ '1' ;
132:       ConfigOut1(2) ⇐ '1' ;
133:     Fin si
134:     Si ValInS alors
135:       Load2 ⇐ '1' ;
136:       ConfigOut2(2) ⇐ '1' ;
137:     Fin si
138:     Si ValInW alors
139:       Load3 ⇐ '1' ;
140:       ConfigOut3(2) ⇐ '1' ;
141:     Fin si
142:     Si WinProp alors
143:       LoadOwn ⇐ '1' ;
144:       ConfigOutOwn(2) ⇐ '1' ;
145:     Fin si
146:     NextState ⇐ sInit ;
147:   Sinon
148:     Congestion ⇐ '1' ;
149:   Fin si
150: Fin si
```

Notons, avant de poursuivre, que la congestion du réseau est détectée grâce à une combinaison des signaux `Congestion`, qui sont en sortie des unités de routage. Dans l'état sInit, ce signal est à '0', et durant la phase d'expansion et de création de chemins, il est à '1' si le contrôleur ne change pas d'état. En combinant les sorties de toutes les unités dans une grande porte ET, il est alors possible de détecter, lors de l'expansion et de la création de chemin, si au moins un contrôleur change d'état. Si tous les contrôleurs restent dans le même état, nous sommes donc en présence d'un problème de congestion, et par ce moyen il est facile de le détecter.

Les encarts 5.1 à 5.2 donnent les équations des différents signaux et bascules du contrôleur de l'unité de routage de HIDRA. Ces équations peuvent directement servir à une implémentation matérielle sur la base de portes logiques, et vont nous servir à comparer les différents algorithmes que nous allons présenter. Bien que leurs comportements ne soient absolument pas identiques, seules des modifications mineures dans quatre de ces équations permettent de passer d'un algorithme à l'autre. Il s'agit des valeurs transmises aux unités de routage voisines, de l'encart 5.2.

États du contrôleur de HIDRA

$$\begin{aligned}
sInit^{+} &= (sInit \wedge \neg OneProp) \vee (sWaitExp \wedge OneProp) \\
&\quad \vee (sFrontExp \wedge OneProp) \\
&\quad \vee (sHasExp \wedge OneProp) \\
sAddress^{+} &= (sInit \wedge OneProp) \vee (sAddress \wedge \neg TriggerIn) \\
sChooseSource^{+} &= sAddress \wedge TriggerIn \\
sWaitExp^{+} &= sChooseSource \vee (sWaitExp \wedge \neg OneProp \wedge \\
&\quad \neg(IsSource \wedge Participate \vee ValInN \\
&\quad \vee ValInE \vee ValInS \vee ValInW)) \\
sFrontExp^{+} &= \neg OneProp \wedge (sWaitExp \wedge (IsSource \wedge \\
&\quad Participate \vee ValInN \vee ValInE \vee \\
&\quad ValInS \vee ValInW)) \\
sHasExp^{+} &= \neg OneProp \wedge (sFrontExp \vee sHasExp)
\end{aligned}$$

Bascules du contrôleur de HIDRA

$$\begin{aligned}
IsConnected^{+} &= IsConnected \vee (sHasExp \wedge WinProp) \vee \\
&\quad (sHasExp \wedge OneProp \wedge IsSource \wedge Participate) \\
Origin(0)^{+} &= ((sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (\neg ValInN \wedge (ValInE \vee \neg ValInS \wedge ValInW))) \vee \\
&\quad Origin(0) \wedge \neg(sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (ValInN \vee (\neg ValInE) \wedge ValInS))) \\
Origin(1)^{+} &= ((sWaitEsp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (\neg ValInN \wedge \neg ValInE \wedge (ValInS \vee ValInW))) \vee \\
&\quad Origin(1) \wedge \neg(sWaitExp \wedge \neg(IsSource \wedge Participate) \wedge \\
&\quad (ValInN \vee ValInE))) \\
IsMaster^{+} &= (sInit \wedge WinProp) \vee (\neg(sInit \wedge IsMaster)) \\
Participate^{+} &= (Participate \wedge \neg(sInit \vee (sChooseSource \wedge \\
&\quad ((OneProp \wedge \neg IsMaster \wedge IsSource) \vee (\neg OpeProp \wedge \\
&\quad \neg IsMaster \wedge IsTarget))))) \vee (sAddress \wedge \\
&\quad TriggerIn \wedge IsSame \wedge (WantConnect \vee IsSource))
\end{aligned}$$

Equations 5.1: Signaux séquentiels du contrôleur de HIDRA

Implémentation matérielle Le but de nos algorithmes étant d'être implémentés en matériel, nous avons évalué le nombre de transistors nécessaires à leur implémentation. Pour ce faire, nous avons utilisé le synthétiseur Leonardo Spectrum, pour lequel nous avons écrit une librairie de composants simples. Pour chacun de ces composants, décrits par le tableau 5.4, nous avons défini son comportement ainsi que le nombre de

Divers signaux du contrôleur de HIDRA

$$
\begin{aligned}
OneProp &= PropInN \vee PropInE \vee PropInS \vee PropInW \vee PropOwn \\
WantConnect &= \neg IsConnected \wedge (IsSource \vee IsTarget) \\
PropOwn &= (sInit \wedge WantConnect \wedge StartRouting) \vee \\
& (sAddress \wedge IsMaster) \vee \\
& (sChooseSource \wedge IsSource \wedge IsMaster) \vee \\
& (sHasExp \wedge IsTarget \wedge Participate) \\
ValIn &= ValInN \vee ValInE \vee ValInS \vee ValInW \\
ReadAddr &= sAddress \\
CompEnable &= sAddress \\
IsRouting &= \neg sInit \vee OneProp \\
Congestion &= (sWaitExp \wedge \neg(IsSource \wedge Participate \vee ValIn)) \vee \\
& (sFrontExp \wedge OneProp) \vee (sHasExp \wedge \neg OneProp)
\end{aligned}
$$

Valeurs de contrôle des multiplexeurs

$$
\begin{aligned}
ConfigOutN(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
ConfigOutE(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
& \vee(IsSource \wedge Participate \wedge \text{"01"}) \\
ConfigOutS(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
& \vee(IsSource \wedge Participate \wedge \text{"10"}) \\
ConfigOutW(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
& \vee(IsSource \wedge Participate \wedge \text{"11"}) \\
ConfigOutOwn(1..0) &= \neg(IsSource \wedge Participate) \wedge Origin \\
LoadN &= ConfigOutN(2) = sHasExp \wedge OneProp \wedge ValInN \\
LoadE &= ConfigOutE(2) = sHasExp \wedge OneProp \wedge ValInE \\
LoadS &= ConfigOutS(2) = sHasExp \wedge OneProp \wedge ValInS \\
LoadW &= ConfigOutW(2) = sHasExp \wedge OneProp \wedge ValInW
\end{aligned}
$$

Valeurs transmises aux voisines par le contrôleur de HIDRA

$$
\begin{aligned}
ValOutN &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInN(2) \wedge ConfigInN = \text{"00"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
& Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"00"}) \\
ValOutE &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInE(2) \wedge ConfigInE = \text{"01"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
& Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"01"}) \\
ValOutS &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInS(2) \wedge ConfigInS = \text{"10"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
& Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"10"}) \\
ValOutW &= (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInW(2) \wedge ConfigInW = \text{"11"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
& Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"11"})
\end{aligned}
$$

Equations 5.2: Signaux combinatoires du contrôleur de HIDRA

transistors qu'il contient. La synthèse du code VHDL décrivant nos unités de routage s'est ensuite exécutée en demandant à Leonardo d'optimiser le nombre de transistors du design.

Composant	**Entrées**	**Fonction**	**Transistors**
Mux	sel, a, b	$sel * a + \overline{sel} * b$	10
MuxInv	$sel0, sel1, a, b$	$sel0 * a + sel1 * b$	8
Inv	a	$\overline{a}$	2
Nor2	a, b	$\overline{a + b}$	4
Nor3	a, b, c	$\overline{a + b + c}$	6
Nor4	a, b, c, d	$\overline{a + b + c + d}$	8
Nor5	a, b, c, d, e	$\overline{a + b + c + d + e}$	10
Nor6	a, b, c, d, e, f	$\overline{a + b + c + d + e + f}$	12
Nand2	a, b	$\bar{a} + \bar{b}$	4
Nand3	a, b, c	$\bar{a} + \bar{b} + \bar{c}$	6
Nand4	a, b, c, d	$\bar{a} + \bar{b} + \bar{c} + \bar{d}$	8
Nand5	a, b, c, d, e	$\bar{a} + \bar{b} + \bar{c} + \bar{d} + \bar{e}$	10
Nand6	a, b, c, d, e, f	$\bar{a} + \bar{b} + \bar{c} + \bar{d} + \bar{e} + \bar{f}$	12
And2	a, b	$a * b$	6
DF1	clk, d	latch : $Q, \bar{Q}$	18
DFC1	clk, clr, d	Bascule D : $Q, \bar{Q}$	20
DFC2	clk, set, d	Bascule D : $Q, \bar{Q}$	20

Tableau 5.4 : *Composants de la librairie de synthèse.*

Il est bien clair que la valeur fournie par le synthétiseur ne correspond pas forcément au résultat d'un layout dessiné par des ingénieurs, qui ont d'autres possibilités d'optimisation (nous n'avons notamment créé qu'un nombre limité de composants pour notre librairie). Cependant, il permet de se faire une idée sur un ordre de grandeur de la taille d'une unité de routage, et offre un moyen de comparer les unités de routage des différents algorithmes.

Le tableau 5.5[3] résume le nombre de transistors et le nombre de bascules nécessaires à l'implémentation d'une unité de routage, de son contrôleur et de son switchbox. Le nombre de transistors n'y tient pas compte des transistors des bascules, leur réalisation étant extrêmement dépendante de la technologie utilisée.

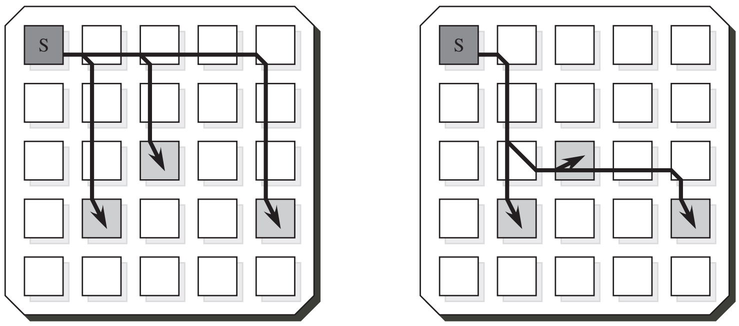
Composant	**Transistors**	**Bascules**
Contrôleur	646	12
Switchbox	292	14
Unité de routage	884	26

Tableau 5.5 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA.*

[3]Le lecteur attentif aura noté que le cumul des valeurs du contrôleur et du switchbox est supérieur au nombre total de transistors. Ce phénomène est dû aux qualités d'optimisation de Leonardo.

5.6 HIDRA-RC

L'algorithme HIDRA procède en lançant la phase d'expansion depuis la source uniquement. La création d'un chemin peut y exploiter un chemin préexistant partant depuis la même source, mais il n'y a pas de priorité donnée aux unités de routage déjà connectées à la source. De ce fait, le chemin trouvé est toujours le plus court, mais le nombre de multiplexeurs réquisitionnés n'est pas forcément optimal, comme le montre la figure 5.20(a). Une priorité étant donnée aux signaux arrivant du Nord lors de la phase d'expansion, les trois destinations de cet exemple se connectent grâce à des chemins allant à l'Ouest, puis au Sud.



(a) Trois destinations reliées à la source, avec HIDRA

(b) Trois destinations reliées à la source, avec réduction des ressources nécessaires

Figure 5.20 : *Comparaison de trois chemins créés avec HIDRA, et la version améliorée HIDRA-RC.*

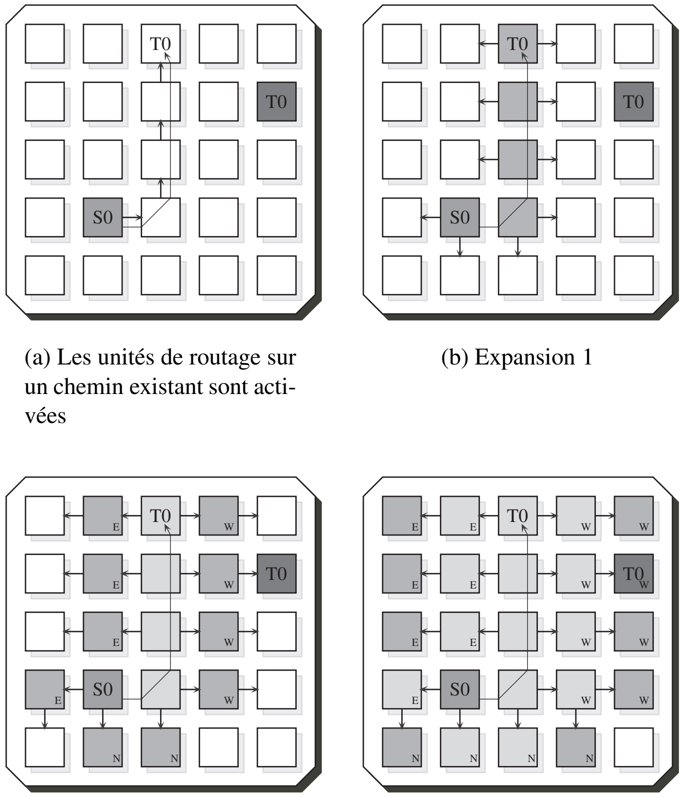
La congestion du réseau de routage, qui voit l'impossibilité de créer certaines nouvelles connexions, de par l'occupation des multiplexeurs, est un problème à éviter absolument. Il sera étudié en détail dans la section 5.11.5, où nous montrerons qu'il est possible de prévoir le nombre de chemins implémentables dans un tableau d'unités de routage de taille définie. Un algorithme capable de minimiser le nombre de multiplexeurs configurés serait donc bienvenu, comme le montre la figure 5.20(b), qui donne une façon de connecter la source à ses trois destinations en minimisant le nombre de multiplexeurs nécessaires, qui passe de 12 à 8.

L'algorithme HIDRA-RC, pour Reduced Congestion, se propose donc d'améliorer les performances de HIDRA en terme de probabilité de congestion. Plusieurs auteurs ont suggéré de lancer la phase d'expansion de l'algorithme de Lee par toutes les cellules présentes sur les chemins reliés à la source à connecter. Nous reprenons cette idée dans HIDRA-RC, en accédant en un coup d'horloge à toutes les unités de routage déjà reliées aux sources participantes. Pour ce faire, nous modifions l'état sWaitExp de la manière suivante :

> Si le contrôleur est une source active, il active `ValOut`*X* si le multiplexeur correspondant est configuré ET qu'il sélectionne la valeur de l'élément externe. S'il n'est pas une source active, et si le contrôleur reçoit une entrée `ValIn`*Y* active, il transmet de manière combinatoire le signal dans

la direction *X* si le multiplexeur correspondant à la direction X est configuré ET s'il est configuré de façon à sélectionner la valeur venant de *Y*.

Cette simple modification implique un changement du comportement des unités de routage, et une propagation combinatoire d'un signal le long de tous les chemins partant de la source active. Au coup d'horloge suivant, la source et toutes les unités de routage atteintes se trouvent dans l'état sFrontExp, et sont dès lors prêtes à lancer l'expansion. La figure 5.21 montre cette phase pour un exemple simple.



(a) Les unités de routage sur un chemin existant sont activées

(b) Expansion 1

(c) Expansion 2

(d) Expansion 3

Figure 5.21 : *Phase d'expansion de l'algorithme HIDRA-RC.*

Les seules modifications à apporter au contrôleur de HIDRA, pour obtenir l'algorithme HIDRA-RC, consistent en les termes en gras de l'encart 5.3. Il s'agit uniquement d'ajouter deux cas où la sortie `ValOut`*X* passe à '1', comme nous venons de le décrire. Concernant le nombre de transistors nécessaires, présentés au tableau 5.6, il est légèrement plus élevé que pour l'algorithme HIDRA, les équations modifiées étant plus importantes que les originales.

Comme nous l'avons déjà mentionné, nos unités de routage utilisent des liaisons combinatoires pouvant traverser le circuit entier. Cette caractéristique est essentielle à l'implémentation de HIDRA-RC, car toutes les unités de routage présentes sur un chemin relié à la source participante doivent être atteintes en un seul coup d'horloge. En effet, dans un système ne fonctionnant qu'avec des interactions locales, pour obtenir le même résultat, il faudrait un moyen de synchroniser toutes les unités reliées à la source, afin de garantir que toutes les unités lancent simultanément une expansion,

Composant	**Transistors**	**Bascules**
Contrôleur	826	12
Switchbox	292	14
Unité de routage	1078	26

Tableau 5.6 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RC.*

$$
\begin{aligned}
ValOutN \;=\; & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInN(2) \wedge ConfigInN = \text{"00"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
& Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"00"}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInN} = \text{"}\mathbf{100}\text{"}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInN}(\mathbf{2}) \wedge (\mathbf{ConfingInN}(\mathbf{1..0}) = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \text{"}\mathbf{00}\text{"})))) \\
ValOutE \;=\; & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInE(2) \wedge ConfigInE = \text{"01"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
& Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"01"}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInE} = \text{"}\mathbf{101}\text{"}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInE}(\mathbf{2}) \wedge (\mathbf{ConfingInE}(\mathbf{1..0}) = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \text{"}\mathbf{01}\text{"})))) \\
ValOutS \;=\; & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInS(2) \wedge ConfigInS = \text{"10"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
& Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"10"}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInS} = \text{"}\mathbf{110}\text{"}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInS}(\mathbf{2}) \wedge (\mathbf{ConfingInS}(\mathbf{1..0}) = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \text{"}\mathbf{10}\text{"})))) \\
ValOutW \;=\; & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInW(2) \wedge ConfigInW = \text{"11"}) \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
& Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = \text{"11"}) \\
& \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInW} = \text{"}\mathbf{111}\text{"}) \\
& \vee((\neg(\mathbf{IsSource} \wedge \mathbf{Participate})) \wedge \mathbf{ValIn} \wedge \\
& \mathbf{ConfigInW}(\mathbf{2}) \wedge (\mathbf{ConfingInW}(\mathbf{1..0}) = \mathbf{Origin}^{+}) \wedge (\mathbf{Origin}^{+} \neq \text{"}\mathbf{11}\text{"}))))
\end{aligned}
$$

Equations 5.3: Valeurs transmises aux voisines par le contrôleur de HIDRA-RC

après un nombre indéfini de coups d'horloge. Ce problème, connu sous le nom de peloton d'exécution (firing squad), a largement été traité par les mathématiciens et les informaticiens dans les cas d'automates cellulaires à une ou deux dimensions, dans le plan continu ou discret [19, 149, 160, 166, 249]. La synchronisation d'un arbre n'a toutefois pas de solution à base d'automate cellulaire à états finis, et donc HIDRA-RC ne pourrait être réalisé à l'aide d'unités de routage à connexions locales.

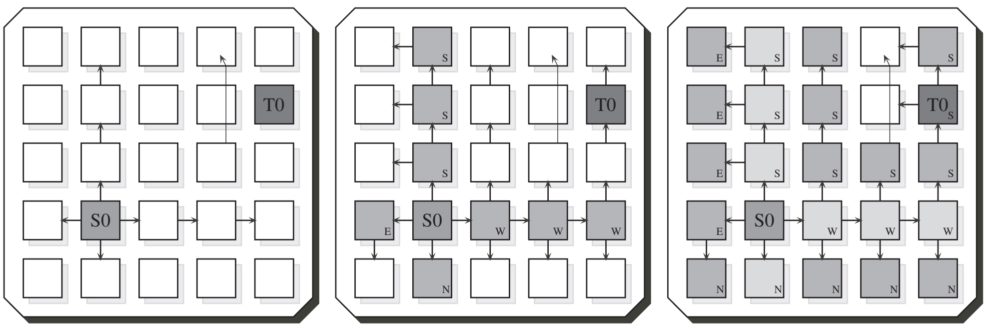
Finalement, notons que HIDRA-RC se comporte exactement de la même manière que HIDRA dans le cas où une source est connectée à au plus une destination. La première connexion à une source est effectivement identique pour les deux algorithmes, HIDRA-RC n'exploitant de nouvelles possibilités que dans le cas de connexions plurielles. Les applications de type réseaux de neurones pourraient en tirer parti, de par le fait que la sortie d'un neurone est, dans la grande majorité des cas, utilisée par plusieurs autres neurones.

5.7 HIDRA-RT

Alors que HIDRA-RC se propose d'améliorer le risque de congestion de HIDRA, une seconde variante permet de réduire le temps d'exécution de l'algorithme. HIDRA-RT, pour Reduced Time, se base sur une recherche de type line-search, inspirée par Mikami et Tabuchi [156]. L'idée est ici de minimiser le temps requis par la phase d'expansion de l'algorithme, qui, dans le cas de HIDRA, nécessite un nombre de coups d'horloge égal à la distance minimal entre la source et la destination.

Alors que dans HIDRA, l'expansion s'exécute pas à pas, où une unité de routage est atteinte après un temps égal à sa distance à la source, dans HIDRA-RT, l'expansion s'effectue par lignes entières. Un contrôleur dans l'état sFrontExp, c'est-à-dire sur le front d'onde, se comporte exactement de la même manière que précédemment, en activant la sortie `ValOut`*X* si cela est possible. Cependant, dans l'état sWaitExp, un contrôleur recevant une entrée `ValIn`*Y* active, transmet immédiatement le signal dans la direction opposée à Y, de façon combinatoire. De ce fait, toutes les unités de routage atteignables par une ligne droite depuis une des unités du front d'onde passent dans l'état sFrontExp au coup d'horloge suivant.

Dans le cas où aucun chemin précédemment routé ne bloque l'expansion en direction de la destination, cette dernière est atteinte en un maximum de deux coups d'horloge, comme illustré à la figure 5.22.



(a) Expansion 1 (b) Expansion 2 (c) Expansion 3

Figure 5.22 : *Phase d'expansion de l'algorithme HIDRA-RT.*

L'avantage de cette approche est donc le temps d'exécution, qui peut être grandement réduit. Toutefois, cette rapidité a un prix, sous la forme de la longueur des chemins. Cet algorithme, de par le fait qu'une ligne est étendue au maximum, tend à minimiser le nombre de virages entre une source et une destination, mais pas forcément la longueur du chemin. Dans le cas où des chemins préexistants bloquent l'expansion courante, il se peut très bien que le chemin trouvé ne soit pas optimal en terme de nombre de multiplexeurs traversés, comme dans l'exemple de la figure 5.23.

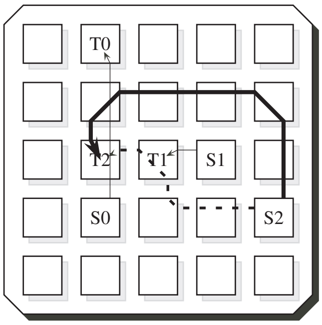


Figure 5.23 : *En gras, le chemin créé par HIDRA-RT (6 muxs), et en traitillé, la solution trouvée par HIDRA (4 muxs), qui minimise la taille du chemin, dans le cas où les sources et destinations 0 et 1 sont déjà connectées.*

Là encore, de même que pour HIDRA-RC, seul le comportement des contrôleurs dans l'état sWaitExp est modifié par rapport à HIDRA, comme ceci :

> Outre sous les conditions présentées dans HIDRA, la valeur `ValOut`*X* est activée si les conditions suivantes sont réunies :
> - l'unité de routage n'est pas une source participante
> - la valeur `ValIn`*Y* est active, avec Y étant la direction opposée à X
> - soit le multiplexeur X n'est pas configuré, soit il l'est en sélectionnant la valeur venant de Y.

L'encart 5.4 illustre les modifications apportées aux équations des signaux `ValOut`*X*, écrites en gras. Là encore, nous pouvons noter qu'une faible modification des unités de routage permet de grandement influencer le comportement de l'algorithme de routage. Le nombre de transistors nécessaires est alors légèrement supérieur à l'implémentation de HIDRA, mais moins importante que pour HIDRA-RC, comme le montre le tableau 5.7.

Composant	**Transistors**	**Bascules**
Contrôleur	754	12
Switchbox	292	14
Unité de routage	958	26

Tableau 5.7 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RT.*

Finalement, à l'instar de HIDRA-RC, HIDRA-RT ne peut fonctionner qu'avec des liaisons combinatoires pouvant traverser les unités de routage. Dans un système n'acceptant que des connexions locales entre les unités de routage, il est clair que le nombre de coups d'horloge indispensables pour atteindre la destination depuis la source est

$$
\begin{aligned}
ValOutN \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInN(2) \wedge ConfigInN = "00") \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee \\
& Origin = ConfigInN(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "00") \\
& \vee(\mathbf{sWaitExp} \wedge (\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \\
& \mathbf{ValInS} \wedge (\neg\mathbf{ConfigInN(2)} \vee \mathbf{ConfigInN(1..0)} = \mathbf{"10"})) \\
ValOutE \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInE(2) \wedge ConfigInE = "01") \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee \\
& Origin = ConfigInE(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "01") \\
& \vee(\mathbf{sWaitExp} \wedge (\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \wedge \neg\mathbf{ValInS} \\
& \mathbf{ValInW} \wedge (\neg\mathbf{ConfigInE(2)} \vee \mathbf{ConfigInE(1..0)} = \mathbf{"11"})) \\
ValOutS \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInS(2) \wedge ConfigInS = "10") \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee \\
& Origin = ConfigInS(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "10") \\
& \vee(\mathbf{sWaitExp} \wedge (\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \\
& \mathbf{ValInN} \wedge (\neg\mathbf{ConfigInS(2)} \vee \mathbf{ConfigInS(1..0)} = \mathbf{"00"})) \\
ValOutW \quad = \quad & (sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge \\
& \neg ConfigInW(2) \wedge ConfigInW = "11") \vee \\
& (\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee \\
& Origin = ConfigInW(1..0))))) \vee (sHasExp \wedge OneProp \wedge \\
& \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "11") \\
& \vee(\mathbf{sWaitExp} \wedge (\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \neg\mathbf{ValInN} \wedge \\
& \mathbf{ValInE} \wedge (\neg\mathbf{ConfigInW(2)} \vee \mathbf{ConfigInW(1..0)} = \mathbf{"01"}))
\end{aligned}
$$

Equations 5.4: Valeurs transmises aux voisines par le contrôleur de HIDRA-RT.

au moins aussi grand que la distance minimale les séparant. HIDRA y est donc la meilleure alternative, le principe line-search n'étant efficace que pour des solutions non parallèles ou acceptant des liaisons combinatoires longue distance.

5.8 HIDRA-RTC

Dernière variante proposée, HIDRA-RTC se propose de combiner les approches de HIDRA-RC et HIDRA-RTC afin de réduire le temps d'exécution, et potentiellement le nombre de multiplexeurs réquisitionnés. L'idée y est de faire partir la phase d'expansion de toutes les unités de routage déjà reliées à la source, comme dans HIDRA-RC, et d'effectuer ensuite l'expansion par lignes entières, à la manière de HIDRA-RT.

Les modifications des équations, par rapport à HIDRA, sont un peu plus importantes que pour les deux variantes précédentes, mais ne touchent toujours que les signaux `ValOut`X, comme le montre l'encart 5.5.

La taille de l'implémentation matérielle se situe entre HIDRA-RT et HIDRA-RC, avec un total de 1022 transistors et 26 bascules D, comme illustré au tableau 5.8.

Sur le plan de la congestion, nous verrons, lors de l'analyse ultérieure, que HIDRA-RTC est légèrement plus efficace que HIDRA-RT, mais inférieur à HIDRA-RC.

Composant	Transistors	Bascules
Contrôleur	828	12
Switchbox	292	14
Unité de routage	1048	26

Tableau 5.8 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-RTC.*

5.9 HIDRA-L

Les quatre algorithmes que nous venons de décrire, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, bien qu'efficacement implémentés en terme de nombre de transistors et nombre de pins, possèdent une faiblesse : la scalabilité. En effet, la grille d'unités de routage peut être facilement étendue, en connectant simplement les unités de routage à leurs voisines, mais les liaisons combinatoires, notamment celles gérées par les unités de propagation, impliquent une réduction de la fréquence d'horloge si la taille de la grille prend des proportions trop importantes. Dans l'optique de pallier à ce problème de scalabilité, nous avons développé un algorithme totalement local, où aucune liaison combinatoire ne parcourt plus d'une unité de routage.

Nous allons décrire en détail cet algorithme, et constater que, bien que plus élégant en terme de scalabilité, il ne sera pas retenu pour la réalisation du circuit POEtic, et ce pour deux raisons. Premièrement, son implémentation nécessite la présence d'un nombre nettement plus important de logique que les quatre versions précédentes, faisant doubler la taille d'une unité de routage. Le circuit POEtic étant limité en terme de surface de silicium, nous avons dû faire le choix de sacrifier un peu de sa scalabilité, pour pouvoir y implémenter un nombre raisonnable d'éléments reconfigurables. Deuxièmement, le nombre de connexions entre deux unités de routage voisines est 2.5 fois plus important, et, dans le cas où plusieurs circuits devaient être reliés entre eux, il n'est pas possible de réduire le nombre de pins nécessaires, contrairement à HIDRA (nous le découvrirons en page 200, dans le cadre de la réalisation du circuit POEtic). Pour un tableau de taille $x \times y$, ce nombre vaut $20x + 20y$, ce qui n'est pas négligeable. Le nombre de pins à disposition pour la réalisation physique du circuit POEtic ayant également été limité, nous ne pouvions décemment garder cet algorithme comme bon candidat.

5.9.1 Algorithme

De manière globale, HIDRA-L (pour Local) fonctionne grâce à cinq mécanismes, à savoir (1) la création d'un espace de routage, (2) la propagation de l'identifiant, (3) la désactivation des concurrents, (4) la création du chemin, qui est décomposée en une phase d'expansion et une phase de rétro-propagation, et (5) la destruction d'un chemin. Ils sont gérés par trois processus concurrents, dont l'un est responsable des mécanismes 2-4, qui sont identiques à ceux décrits dans le cadre de HIDRA, si ce n'est que la propagation des signaux est séquentielle plutôt que combinatoire.

- La création d'un chemin est semblable à celle observée dans HIDRA, et pour qu'elle puisse être menée à bien, il faut que, dans un processus de routage, seules les sources et les destinations répondant à un identifiant unique soient activées.

$$
\begin{aligned}
\mathbf{Follow} \quad &= \quad \mathbf{ValIn} \wedge ((\mathbf{ConfigInN(2)} \wedge \mathbf{ConfigInN(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"00"}) \vee \\
& \quad (\mathbf{ConfigInE(2)} \wedge \mathbf{ConfigInE(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"01"}) \vee \\
& \quad (\mathbf{ConfigInS(2)} \wedge \mathbf{ConfigInS(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"10"}) \vee \\
& \quad (\mathbf{ConfigInW(2)} \wedge \mathbf{ConfigInW(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"11"}) \vee \\
& \quad (\mathbf{IsConnected} \wedge \mathbf{IsTarget} \wedge \mathbf{ConfigInOwn(1..0)} = \mathbf{Origin^{+}})) \\
ValOutN \quad &= \quad \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge} \\
& \quad \cancel{\neg ConfigInN(2) \wedge ConfigInN = "00") \vee} \\
& \quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInN(2) \vee} \\
& \quad \cancel{Origin = ConfigInN(1..0))))) \vee} (sHasExp \wedge OneProp \wedge \\
& \quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "00") \\
& \quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInN(2)}) \\
& \quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInN} = \mathbf{"100"}) \\
& \quad \vee(\neg(\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInN(2)} \wedge \\
& \quad \mathbf{ConfigInN(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"00"}) \vee \\
& \quad (\neg\mathbf{Follow} \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \wedge \mathbf{ValInS} \wedge \neg\mathbf{ConfigInN(2)}))))) \\
ValOutE \quad &= \quad \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge} \\
& \quad \cancel{\neg ConfigInE(2) \wedge ConfigInE = "01") \vee} \\
& \quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInE(2) \vee} \\
& \quad \cancel{Origin = ConfigInE(1..0))))) \vee} (sHasExp \wedge OneProp \wedge \\
& \quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "01") \\
& \quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInE(2)}) \\
& \quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInE} = \mathbf{"101"}) \\
& \quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInE(2)} \wedge \\
& \quad \mathbf{ConfigInE(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"01"}) \vee \\
& \quad (\neg\mathbf{Follow} \wedge \neg\mathbf{ValInN} \wedge \neg\mathbf{ValInE} \wedge \neg\mathbf{ValInS} \wedge \mathbf{ValInW} \wedge \neg\mathbf{ConfigInE(2)}))))) \\
ValOutS \quad &= \quad \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge} \\
& \quad \cancel{\neg ConfigInS(2) \wedge ConfigInS = "10") \vee} \\
& \quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInS(2) \vee} \\
& \quad \cancel{Origin = ConfigInS(1..0))))) \vee} (sHasExp \wedge OneProp \wedge \\
& \quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "10") \\
& \quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInS(2)}) \\
& \quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInS} = \mathbf{"110"}) \\
& \quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInS(2)} \wedge \\
& \quad \mathbf{ConfigInS(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"10"}) \vee \\
& \quad (\neg\mathbf{Follow} \wedge \mathbf{ValInN} \wedge \neg\mathbf{ConfigInS(2)}))))) \\
ValOutW \quad &= \quad \cancel{(sFrontExp \wedge \neg OneProp \wedge ((IsSource \wedge Participate \wedge} \\
& \quad \cancel{\neg ConfigInW(2) \wedge ConfigInW = "11") \vee} \\
& \quad \cancel{(\neg(IsSource \wedge Participate) \wedge (\neg ConfigInW(2) \vee} \\
& \quad \cancel{Origin = ConfigInW(1..0))))) \vee} (sHasExp \wedge OneProp \wedge \\
& \quad \neg(IsSource \wedge Participate) \wedge ValIn \wedge WinProp \wedge Origin = "11") \\
& \quad \vee(\mathbf{sFrontExp} \wedge \neg\mathbf{OneProp} \wedge \neg\mathbf{ConfigInW(2)}) \\
& \quad \vee(\mathbf{sWaitExp} \wedge ((\mathbf{IsSource} \wedge \mathbf{Participate} \wedge \mathbf{ConfigInW} = \mathbf{"111"}) \\
& \quad \vee((\neg\mathbf{IsSource} \wedge \mathbf{Participate}) \wedge \mathbf{ValIn} \wedge ((\mathbf{ConfigInW(2)} \wedge \\
& \quad \mathbf{ConfigInW(1..0)} = \mathbf{Origin^{+}} \wedge \mathbf{Origin^{+}} \neq \mathbf{"11"}) \vee \\
& \quad (\neg\mathbf{Follow} \neg\mathbf{ValInN} \wedge \mathbf{ValInE} \wedge \neg\mathbf{ConfigInW(2)})))))
\end{aligned}
$$

Equations 5.5: Valeurs transmises aux voisines par le contrôleur de HIDRA-RTC.

Nous créons donc des espaces de routage, à l'intérieur desquels seules les unités de routage répondant à un identifiant participent au processus de création de chemin.

- Dans un espace de routage, le maître désigné envoie son identifiant, puis élimine ses concurrents, de la même façon que dans HIDRA. La phase d'expansion est également identique : seule la configuration des multiplexeurs se fait de manière séquentielle, et non en un seul coup d'horloge.
- Finalement, étant donné que plusieurs unités de routage peuvent désirer se connecter au même instant, les espaces de routage sont concurrents, et un système de priorité permet à un des espaces de l'emporter sur les autres. Lorsqu'un espace est petit à petit détruit, et que la phase de configuration des multiplexeurs n'est pas achevée, il faut déconfigurer les multiplexeurs qui ne sont pas reliés à leur source.

Nous allons passer en revue le détail de fonctionnement de ces trois processus, afin de mieux cerner les subtilités de l'algorithme HIDRA-L.

Création d'un espace de routage

Un système de propagation permet à une unité de routage désirant se connecter d'émettre un signal vers ses quatre voisines, qui au coup d'horloge suivant le transmettront plus loin en suivant les règles du tableau 5.9. L'ensemble des unités atteintes par cette vague constituent ce que nous appelons un espace de routage, et participent ensuite au processus de routage de l'unité centrale. Si plusieurs unités tentent de prendre possession de l'espace de cette façon, une priorité est donnée à celle la plus au Sud-Ouest, son expansion écrasant l'espace de routage réservé par les autres, comme illustré à la figure 5.24.

Règle	**Sorties actives**
Si Sud actif	Nord
Sinon si Ouest actif	Nord, Sud, Est
Sinon si Contrôleur actif	Nord, Sud, Est, Ouest
Sinon si Est actif	Nord, Sud, Ouest
Sinon si Ouest actif	Sud

Tableau 5.9 : *Direction de transmission du signal de propagation, en fonction de son origine.*

Ce mécanisme est prioritaire sur le processus de création de chemins. Lorsqu'un espace de routage $E1$ en écrase un autre $E2$, le processus de routage de $E2$ est détruit au fur et à mesure de l'avancée de $E1$.

Lorsque la création du chemin impliquant l'unité de routage maître de l'espace de routage est terminé, la source de ce chemin relâche l'espace de routage, en propageant un signal actif de manière identique à la création d'un espace de routage. Les unités touchées transmettent ce signal, et se retrouvent à nouveau libres. Dès lors, les unités désirant se connecter peuvent recréer un nouvel espace, pour lancer un nouveau processus de routage, comme illustré à la figure 5.25.

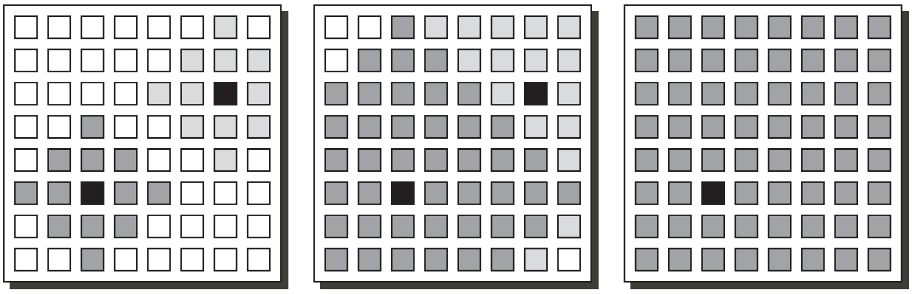


Figure 5.24 : *Trois étapes de la création des espaces de routage, aux temps t_0+2, t_0+5 et t_0+10. Les deux unités noirs désirant initier une connexion. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

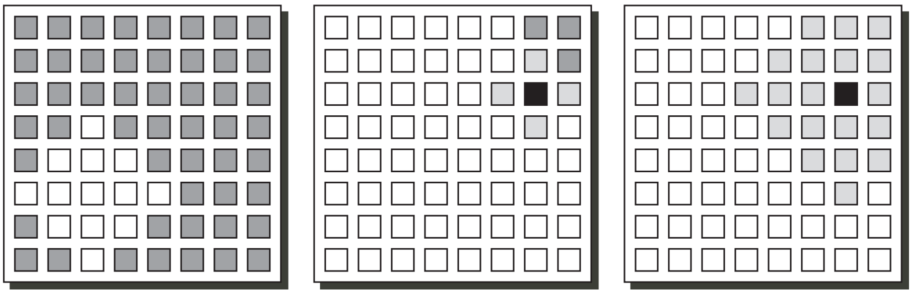


Figure 5.25 : *Destruction d'un espace de routage, aux temps t_1+2, t_1+8 et t_1+10. L'unité noir désire initier une connexion, et crée un nouvel espace de routage. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

Propagation d'identifiant

Immédiatement après avoir émis son bit de propagation, l'unité de routage désirant se connecter envoie son identifiant de manière sérielle à ses voisines, qui le transmettent ensuite à leur voisinage suivant les règles du tableau 5.9. Pour ce faire, un compteur est inclus dans le contrôleur afin de sélectionner un bit de l'identifiant stocké dans l'élément externe. Typiquement, pour un identifiant sur 16 bits un compteur de 4 bits permet d'effectuer cette tâche. Chaque unité de routage recevant les bits de l'identifiant les compare avec son propre identifiant, afin de savoir si elle doit ou non participer au processus courant. Ici encore, le compteur est utilisé afin d'accéder au bon bit de l'identifiant de l'élément externe. A ce stade nous pouvons noter qu'un accès sériel aurait nécessité un nombre moindre de liaisons entre l'unité de routage et l'élément externe, mais qu'il était rendu impossible par l'écrasement des espaces de routage. Le coup d'horloge suivant un écrasement, le premier bit de l'identifiant est reçu et doit immédiatement être comparé avec le bit local. Il faudrait donc pouvoir replacer le registre à décalage de l'élément externe dans son état initial en un seul coup d'horloge, ce qui nécessiterait une grande quantité de logique.

Lorsque l'adresse a été envoyée, l'unité de routage désirant se connecter indique son type (source ou destination) en envoyant un signal dans la même direction que l'identifiant. Il est également transmis par les autres unités et sert à désactiver les unités de routage possédant le même identifiant et le même type. De cette manière nous garantissons qu'un seul chemin est créé à la fois.

Création du chemin

Après que l'adresse ait été envoyée, la phase d'expansion est similaire à celle de HIDRA, où un front d'onde s'étend, à partir de la source, à chaque coup d'horloge jusqu'à atteindre la destination. Lorsque la destination est atteinte, elle lance la phase de rétro-propagation, en envoyant un signal actif dans la direction de l'origine qu'elle a stockée. La voisine transmet ensuite le signal toujours en direction de son origine, tout en configurant le multiplexeur permettant de créer le chemin de données. Ce processus est séquentiel, et nécessite donc un nombre de coups d'horloge identique à la phase d'expansion, qui correspond à la distance entre la source et la destination. Lorsque la source est atteinte, à la fin de cette phase, toutes les unités de routage sur le chemin le plus court entre la source et la destination sont configurées. A ce moment-là elle utilise le même principe que lors de la création de l'espace de routage pour le détruire, laissant ainsi la place à de futurs processus de routage.

Destruction d'un chemin

Finalement, un mécanisme d'effacement de chemin est nécessaire dans le cas où un processus de routage n'a pas le temps de se terminer dans un espace de routage avant que cet espace ne soit supprimé par un autre. Le chemin partiellement créé doit alors être détruit, et il sera reconstruit lors d'un processus ultérieur. Ce cas arrive lorsqu'un chemin est en train d'être construit lors de la deuxième phase de l'algorithme de Lee. Si une unité de routage est atteinte par la rétro-propagation en même temps que par une destruction d'espace de routage, elle envoie un signal dans le sens de sa destination et ne configure pas son multiplexeur. L'unité suivante déconfigure alors son multiplexeur et le signal est transmis jusqu'à ce que la destination soit atteinte. Nous pouvons noter que ce processus est prioritaire sur tous les autres, car un chemin qui a commencé à être détruit doit impérativement l'être jusqu'au bout afin de ne pas réquisitionner inutilement des ressources.

La figure 5.26 montre un espace de routage qui en détruit un dans lequel un chemin est en train d'être créé en phase de rétro-propagation. Les multiplexeurs sont déconfigurés, et un nouveau processus de routage sera nécessaire à la réalisation de ce chemin.

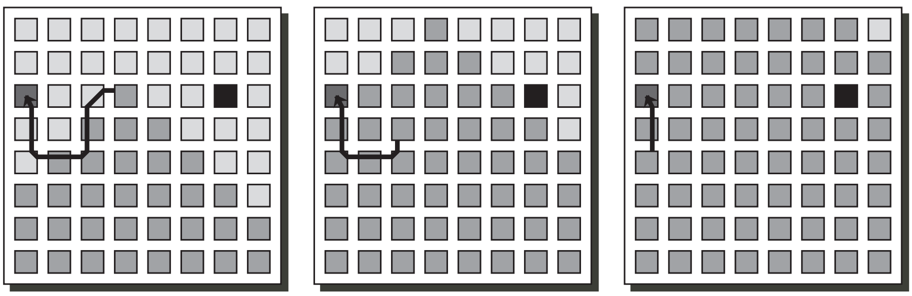


Figure 5.26 : *Destruction d'un espace de routage par un autre, aux temps t_2, t_2+2 et t_2+5. Le chemin qui était en train de se créer est détruit petit à petit. Les deux couleurs grises indiquent à quel espace de routage une unité appartient.*

Un deuxième cas de figure voit la destruction d'un chemin partiellement créé. Lorsque deux chemins, dans un même espace de routage, sont en phase de rétropropagation, il faut garantir qu'un seul d'entre eux sera effectivement réalisé, de manière à ce qu'un processus de routage, comme dans HIDRA, ne crée qu'une seule connexion à la fois. Pour ce faire, une unité de routage qui est touchée par la rétropropagation donne une priorité à un des chemins qui tentent de passer par elle, et détruit les autres.

5.9.2 Implémentation

L'algorithme global explicité, nous pouvons maintenant entrer dans les détails de l'unité de routage. La figure 5.27 nous montre la décomposition de l'unité en un switchbox et un contrôleur. Le switchbox est réalisé à l'aide de cinq multiplexeurs, un pour chacune des directions et un pour le signal à envoyer à l'élément externe, si l'unité est une destination. Chacun des multiplexeurs est contrôlé par deux bits de sélection, stockés dans deux bascules, et un bit supplémentaire sert à indiquer au contrôleur si le multiplexeur est déjà configuré, c'est-à-dire appartient à un chemin de données. Ce switchbox se distingue de ceux précédemment utilisés par la présence de cinq bascules supplémentaires, qui imposent une transmission locale des données, afin de garantir qu'aucun chemin combinatoire ne traverse plusieurs unités de routage.

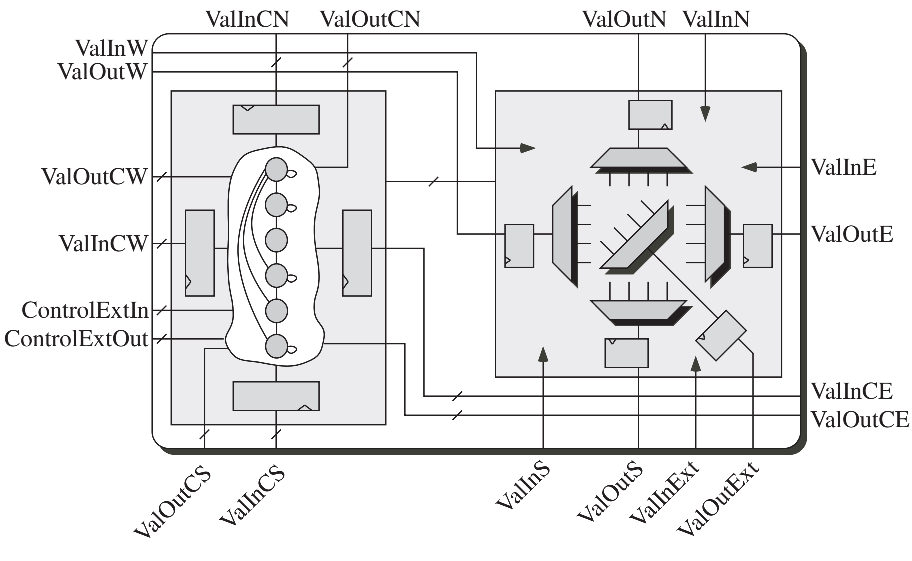


Figure 5.27 : *Une unité de routage de type HIDRA-L.*

Nous pouvons également constater que le contrôleur possède des registres, en entrée, qui en font une machine de Moore. Le reste du contrôleur est réalisé à l'aide d'une machine d'états codée en 1 parmi M. Cette implémentation n'est pas optimale en terme de ressources, mais a servi à démontrer le fonctionnement correct de l'algorithme. Dans le cas où un circuit embarquant cet algorithme devait voir le jour, nous devrions pouvoir la réduire de quelques bascules. Nous n'allons pas donner ici le code correspondant, mais un pseudo-code décrivant les trois processus concurrents intervenant dans le contrôleur. Le premier, la destruction de chemins (Processus 5.10), est prioritaire sur les deux autres. Le deuxième, gérant les espaces de routage (Processus

5.8), est quant à lui prioritaire sur le dernier, qui implémente le mécanisme d'envoi d'adresse et de construction de chemin (Processus 5.9).

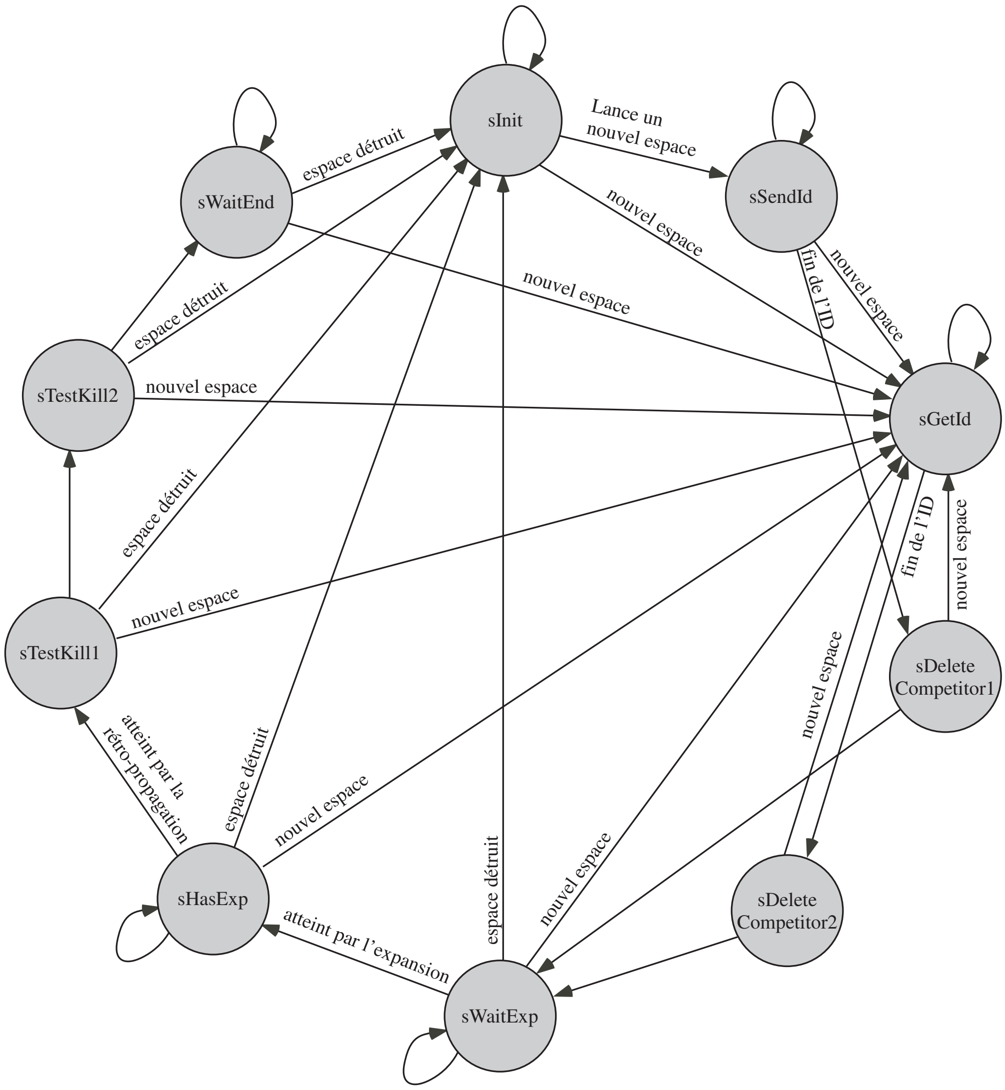


Figure 5.28 : *Machine d'états du contrôleur de HIDRA-L.*

Sur le plan de l'implémentation matérielle, la machine d'état nécessite l'utilisation de 10 bascules en codage 1 parmi M, et plusieurs bascules supplémentaires sont présentes :

- `Participate` : Indique si l'unité de routage participe au routage courant (possède la même adresse que le maître de l'espace de routage et n'est pas du même type que lui).
- `Origin` : Sur 2 bits, mémorise l'origine de la phase d'expansion.
- `PropOrigin` : Sur 2 bits, mémorise l'origine de la phase de création de l'espace de routage.
- `IsConnected` : Indique si l'unité de routage est connectée à son correspondant.
- `IsSame` : Un bit, qui indique si les deux identifiants comparés sont identiques ou non.

- `LastConf` : Sur 3 bits, mémorise quel est le dernier multiplexeur configuré.
- `Counter` : Un compteur, typiquement sur 4 bits, permettant l'accès aux bits d'adresse.

Enfin, des bascules sont ajoutées à chacune des entrées afin d'assurer qu'aucun chemin combinatoire ne traverse le circuit. Dans l'implémentation actuelle nous avons quatre entrées depuis chacune des directions, pour un total de 16 bascules, ce qui implique qu'un côntrôleur comporte 40 bascules.

Sur le plan des liaisons entre unités de routage, un signal par direction sort du switchbox, et est transmis à la voisine. Les contrôleurs nécessitent quant à eux quatre signaux dans chaque direction, qui sont définis de la manière suivante :

- `SpaceIn`*X*, `SpaceOut`*X* : Sur deux bits, est utilisé pour créer et détruire les espaces de routage, envoyer l'identifiant, désactiver les concurrents, et pour l'expansion du front d'onde :

"11" :	Création d'un espace de routage
"01" :	Destruction d'un espace de routage
"10" :	Expansion, bit d'adresse à 1, ou désactivation des sources
"00" :	bit d'adresse à 0, ou désactivation des destinations

- `KillPathIn`*X*, `KillPathOut`*X* : Ce signal s'active pour détruire un chemin qui n'a pas terminé de se former avant d'être avalé par un nouvel espace de routage.
- `RetroPropIn`*X*, `RetroPropOut`*X* : Utilisé lors de la phase de rétropropagation, qui est lancée par la destination lorsqu'elle est atteinte par l'expansion.

Nous allons à présent détailler le comportement des contrôleurs, en commençant par la création et la destruction des espaces de routage. Nous enchaînerons sur leur comportement dans chacun des états de la machine d'états, avant de terminer par la destruction de chemins.

Création/destruction des espaces de routage

Lorsqu'une unité de routage se trouve dans l'état sInit et que son élément externe lui demande de se connecter, le contrôleur lance la création d'un espace de routage, en plaçant `SpaceOut`*X* à la valeur "11" dans chacune des quatre directions, pour autant que `SpaceInS(0)`$= 0$ et `SpaceInW(0)`$= 0$. De ce fait, une priorité est donnée aux espaces étendus depuis les points situés au Sud-Ouest.

Quand une source se trouve connectée grâce au signal de rétropropagation lancé par la destination correspondante, elle place `SpaceOut`*X* à la valeur "01" dans chacune des directions, pour détruire l'espace de routage dans lequel elle se trouve.

Le processus de création d'espace est, comme explicité par la partie d'algorithme 5.8, prioritaire sur celui de destruction d'espace, et sur le plan des directions, la priorité est donnée dans l'ordre suivant : Sud, Ouest, élément externe, Est, et Nord. Les deux bascules de `PropIn` mémorisent l'origine de la précédente vague de création d'espace de routage, de façon à ce que la priorité soit toujours correctement respectée.

Machine d'état

Nous allons maintenant décrire les actions exécutées dans chacun des états du contrôleur, qui sont résumées par la partie d'algorithme 5.9. Nous partons du principe que ces actions sont prises pour autant qu'une création ou une destruction d'espace de

Processus 5.8 HIDRA-L : Contrôleur de l'unité de routage, processus des espaces de routage

```
Si SpaceInS="11" OU SpaceInW="11" alors
  Passer dans l'état sGetId
  Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si SpaceInS="01" OU SpaceInW="01" alors
  Passer dans l'état sInit
  Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si l'unité veut se connecter et la machine d'état est dans l'état sInit alors
  Passer dans l'état sSendId
  SpaceOutX="11"
Sinon si l'unité est la source et vient d'être connectée (état sHasExp) alors
  Passer dans l'état sInit
  SpaceOutX="01"
Sinon si SpaceInE="11" OU SpaceInN="11" alors
  Passer dans l'état sGetId
  Activer SpaceOutY comme décrit dans le tableau 5.9
Sinon si SpaceInE="01" OU SpaceInN="01" alors
  Passer dans l'état sInit
  Activer SpaceOutY comme décrit dans le tableau 5.9
Fin si
```

routage n'a pas cours. Si tel n'est pas le cas, le comportement de la machine d'états suit les indications de la partie d'algorithme 5.8.

sInit Le contrôleur reste par défaut dans son état initial. S'il désire initier une connexion, il passe dans l'état sSendId, et s'il n'est pas touché par un nouvel espace de routage, il passe dans l'état sGetId.

sSendId Dans cet état, le contrôleur est maître de l'espace de routage, et envoie son identifiant de manière sérielle, en un nombre de coups d'horloge correspondant à la valeur de son compteur. Il l'incrémente, de façon à accéder le bon bit d'identifiant de l'élément externe, et détecter la fin de l'envoi de l'identifiant, qui le fait passer dans l'état sDeleteCompetitor1. Avant de changer d'état, il fait passer `Participate` à '1'.

sGetId De même que dans l'état sSendId, le compteur est activé, mais ici l'identifiant reçu est comparé avec celui de l'élément externe. Lorsque le compteur a terminé son comptage, l'état suivant est sDeleteCompetitor2, et si les identifiants sont égaux, `Participate` passe à '1'.

sDeleteCompetitor1 Si l'unité de routage est une source, elle place `SpaceOut`*X* à "10" dans toutes les directions. L'état suivant est, pour autant qu'il n'y ait pas création d'un nouvel espace de routage, sWaitExp.

sDeleteCompetitor2 Si l'unité de routage est une source, et que `SpaceIn`*Y* vaut "10", alors elle désactive sa participation en plaçant '0' dans `Participate`. Si elle est une destination et que `SpaceIn`*Y* vaut "00", elle désactive également sa participation. L'état suivant est sWaitExp.

Processus 5.9 HIDRA-L : Contrôleur de l'unité de routage, processus d'envoi d'adresse et de création de chemin

```
Si (state)
  Egale sInit ⇒
    Tous les signaux sont désactivés
  Egale sSendId ⇒
    Envoie un bit d'adresse à la fois aux 4 voisines
    Incrémente le compteur
    Si compteur termine alors
      Aller à sDeleteCompetitor1
    Fin si
  Egale sGetId ⇒
    Compare le bit reçu avec son adresse et le transmet plus loin
    Incrémente le compteur
    Si compteur termine alors
      Si IsSame=='1' alors
        Participate ⇐ '1'
      Fin si
      Aller à sDeleteCompetitor2
    Fin si
  Egale sDeleteCompetitor1 ⇒
    Si l'unité est une source alors
      SpaceOut à "10" pour les 4 voisines
    Fin si
    Aller à sWaitExp
  Egale sDeleteCompetitor2 ⇒
    Si Participate=='1' alors
      Si (Un SpaceIn est à "10" ET l'unité est une source) OU
      (Pas de SpaceIn à "10" reçu ET l'unité est une destination) alors
        Participate ⇐ '0'
      Fin si
    Fin si
    Aller à sWaitExp
  Egale sWaitExp ⇒
    Si Participate=='1' et l'unité est une source alors
      Activer le signal d'Expansion pour toutes les voisines
      Aller à sHasExp
    Sinon si Expansion en entrée est actif alors
      Stocker l'origine du signal
      Transmettre le signal aux voisines
      Aller à sHasExp
    Fin si
  Egale sHasExp ⇒
    Si (Participate='1' ET l'unité est une destination) OU un RetroPropIn est actif alors
      Configurer le multiplexeur correspondant
      Si Participate='0' alors
        Transmettre la rétropropagation dans la direction de l'origine
      Fin si
      Aller à sTestKill1
    Fin si
  Egale sTestKill1 ⇒
    Active KillPathOut dans la direction de LastConf
  Egale sTestKill2 ⇒
    Si KillPathIn ne s'active pas alors
      Déconfigure le multiplexeur pointé par LastConf
      Activer KillPathOut dans la direction pointée par LastConf
    Fin si
  Egale sWaitEnd ⇒
    Tous les signaux sont désactivés
Fin si égal
```

sWaitExp Si l'unité est une source et que `Participate` est à '1', alors le contrôleur passe dans l'état sHasExp et active le signal d'expansion dans chacune des directions dont le multiplexeur n'est pas configuré de façon à sélectionner une autre valeur que celle de l'élément externe. Dans tous les autres cas, le contrôleur attend d'être atteint par l'expansion. Il stocke alors son origine, transmet le signal dans les directions autorisées, et passe dans l'état sHasExp.

sHasExp Si l'unité est une destination et que `Participate` est à '1', elle active `RetroPropOut`*Y* dans la direction pointée par `Origin`, et configure son multiplexeur sélectionnant la valeur à envoyer à l'unité externe. Toute autre unité de routage attend que `RetroPropIn`*X* soit activé, et si tel est le cas, elle configure le multiplexeur de la direction *X*, et active `RetroPropOut`*Y* dans la direction correspondant à l'origine de l'expansion. Avant de passer dans l'état sTestKill1, le contrôleur stocke, dans `LastConf`, quel est le multiplexeur qui vient d'être configuré. Il est important de noter qu'il est possible, si plusieurs destinations ont été atteintes par la phase d'expansion, que plusieurs chemins tentent de se créer au même instant.

sTestKill1 Le contrôleur active le signal `KillPathOut`*X*, où X correspond à la direction pointée par `LastConf`, et passe dans l'état sTestKill2.

sTestKill2 S'il ne reçoit pas de signal `KillPathIn`*X*, où X correspond à la direction pointée par `Origin`, il déconfigure le multiplexeur pointé par `LastConf`, et lance la destruction du chemin.

Les deux états sTestKill1 et sTestKill2, qui peuvent sembler surprenants, servent à ne créer qu'un seul chemin par processus de routage. En effet, si plusieurs destinations sont atteintes par la phase d'expansion, elles vont toutes commencer à créer un chemin en direction de la source, en y configurant les multiplexeurs. Il faut absolument éviter qu'ils ne soient tous créés, et donc si nous avons trois unités de routage, U1, U2, et U3, où U2 et U3 sont voisines de U1, que U1 a été atteint par l'expansion avant U2 et U3, et que U2 et U3 sont atteintes en même temps par la rétropropagation (Figure 5.29), il faut que l'une des deux soit déconfigurée. Le comportement illustré par le tableau 5.10 sera alors observé, à chaque nouveau coup d'horloge.

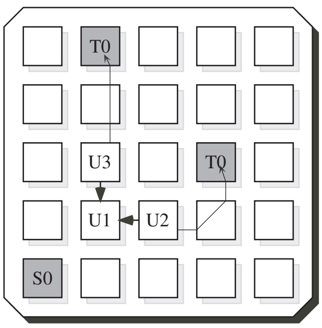


Figure 5.29 : *Exemple de rétropropagation où deux chemins entrent en concurrence.*

sWaitEnd Cet état est atteint par une unité de routage qui est présente sur un chemin qui vient de se créer. Elle y attend que l'espace de routage soit détruit, ou qu'un nouvel espace l'écrase.

U1	**U2**	**U3**	action
sHasExp	sHasExp	sHasExp	U2 et U3 activent le signal de rétropropagation en direction de U1, configurent un multiplexeur, et passent dans l'état sTestKill1.
sHasExp	sTestKill1	sTestKill1	U1 continue la rétropropagation et configure les deux multiplexeurs des directions de U2 et U3, et les trois contrôleurs changent d'état.
sTestKill1	sTestKill2	sTestKill2	U1 active `KillPathOut` dans la direction de U2. U3 ne reçoit pas de signal, déconfigure le multiplexeur qu'il vient de configurer, et active `KillPathOut` dans la direction de la destination courante.

Tableau 5.10 : *Comportement de trois unités de routage voisines, lors de la rétro-propagation.*

Destruction de chemin

La destruction d'un chemin partiellement construit s'exécute avec l'aide d'un des quatre signaux de connexion inter-unité de routage, `KillPath`. L'activation de ce signal s'exécute dans trois cas, résumés par la partie d'algorithme 5.10 :

- Si le contrôleur est dans l'état `sTestKill1`, et qu'il ne change pas d'espace de routage, il active `KillPathOut`*X* dans la direction pointée par `LastConf`.
- Si le contrôleur est dans l'état `sTestKill2` et qu'il ne reçoit pas de `KillPathIn`*Y* actif, il active `KillPathOut`*X* dans la direction pointée par `LastConf`.
- Si un signal `KillPathIn`*Y* est actif en entrée et que l'état n'est pas `sTestKill2`, les multiplexeurs sélectionnant cette direction sont déconfigurés, et les `KillPathOut`*X* correspondant à ces multiplexeurs sont activés.

Processus 5.10 HIDRA-L : Contrôleur de l'unité de routage, processus de destruction de chemins

Si la machine d'état est dans l'état sTestKill1 **alors**
    Activer KillPathOut dans la direction pointée par LastConf
Sinon si KillPathIn n'est pas actif et l'état courant est sTestKill2 **alors**
    Déconfigurer le multiplexeur pointé par LastConf
    Activer KillPathOut dans la direction pointée par LastConf
Sinon si KillPathIn est actif en entrée et l'état n'est pas sTestKill2 **alors**
    Déconfigurer le multiplexeur sélectionnant cette entrée
    Activer KillPathOut dans la direction du multiplexeur
Fin si

Réalisation matérielle

Concernant la réalisation matérielle des unités de routage, elles ont, comme les précédentes, été décrites en VHDL synthétisable. Le tableau 5.11 résume la quantité de logique nécessaire à l'implémentation d'une unité de routage, et nous pouvons constater qu'elle double par rapport aux quatre algorithmes précédents. Bien que plus scalable, cette solution a donc le désavantage de la taille et du nombre de pins nécessaires à la potentielle mise en réseau de plusieurs circuits.

Composant	**Transistors**	**Bascules**
Contrôleur	1828	40
Switchbox	292	19
Unité de routage	2078	59

Tableau 5.11 : *Ressources nécessaires à l'implémentation matérielle d'une unité de routage de type HIDRA-L.*

Dans la suite de ce chapitre, consacré tout d'abord aux voisinages, puis à l'analyse des algorithmes, nous nous concentrons sur les quatre premiers, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC. Concernant la phase d'analyse, les performances de HIDRA-L sont en effet comparables à celles de HIDRA, tant au niveau des longueurs de chemins et des multiplexeurs réquisitionnés, qu'au niveau des problèmes de congestion. Seul le nombre de coups d'horloge nécessaires à l'exécution de tous les processus de routage diffère, étant donné que la rétro-propagation n'est plus combinatoire, mais séquentielle.

5.10 Voisinages

Les études concernant le routage matériel présentées au chapitre 4 se sont toutes penchées sur le même type de voisinage, à savoir des systèmes à quatre voisines. De même, jusqu'à présent nous nous sommes uniquement concentrés sur des algorithmes faisant intervenir un voisinage de Von Neumann. Toutefois, rien ne prouve qu'il n'existe pas de voisinages plus performants, considérant la probabilité de congestion comparée au nombre de transistors requis pour une implémentation matérielle. Nous allons donc étudier l'impact de différents voisinages sur les performances des quatre algorithmes HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC.

Travaillant sur des systèmes matériels à deux dimensions, et désirant le faire avec des structures régulières, nous allons nous occuper des pavages réguliers du plan. Les trois pavages basés sur des polygones réguliers sont réalisés à l'aide de triangles, carrés et hexagones, correspondant respectivement à des voisinages de 3, 4 et 6. Nous ajoutons à ces possibilités le voisinage de Moore, qui peut aisément être créé avec une structure régulière d'octogones (Figure 5.30).

Il est également possible d'exploiter les unités de routage du voisinage de 8 pour créer un voisinage de 4 où deux liaisons sont établies entre chaque paire de voisines. L'implémentation de base des unités de routage reste identique, seule la connectique entre voisines étant modifiée, comme le montre la figure 5.31. Nous sommes donc en possession d'un voisinage de 4 à double liaison, et ce sans devoir modifier la description des unités. Par la suite, nous désignerons ce voisinage par le sigle 4-2.

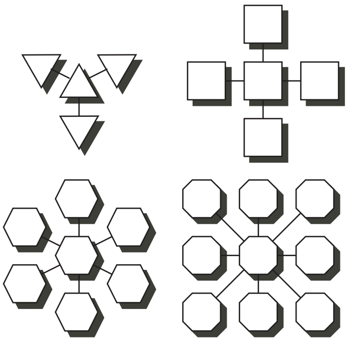


Figure 5.30 : *Les différents types de voisinages utilisés dans cette étude.*

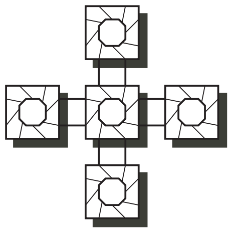


Figure 5.31 : *Un voisinage de 4 avec deux liaisons par couple d'unité de routage.*

5.10.1 Switchboxs

Un des éléments déterminants dans la taille occupée par une unité de routage est le switchbox [4]. Il doit contenir $n+1$ multiplexeurs pour un voisinage de n : un dans chaque direction, plus un additionnel vers l'élément externe lui étant connecté. De plus, la taille de ces multiplexeurs, et donc le nombre de bascules nécessaires au stockage des configurations, dépend également du voisinage. Le nombre de bits nécessaires à la définition du comportement d'un multiplexeur est égal à $\lceil log_2(n) \rceil + 1$, le "$+1$" correspondant au bit indiquant si le multiplexeur est déjà configuré ou non. Le tableau 5.12 résume ces caractéristiques dans le cas des voisinages traités dans cette thèse. Le nombre total de bascule y est calculé comme $NombreDeMux \times BitsDeConfiguration - 1$, où le "$-1$" correspond au multiplexeur dirigé vers l'élément externe, qui ne nécessite pas d'indiquer s'il est configuré ou non. L'équation en fonction de n est donc : $(n+1)(\lceil log_2(n) \rceil + 1) - 1$.

Voisinage	Multiplexeurs	Bits de sélection (par mux)	Bits de configuration (par mux)	Transistors (total)	Bascules (total)
3	4	2	3	192	11
4	5	2	3	292	14
6	7	3	4	650	27
8	9	3	4	982	35

Tableau 5.12 : *Composition d'un switchbox en fonction du type de voisinage.*

[4] Une description de la fontionnalité du switchbox a été faite en page 113.

5.10.2 Nombre total de transistors

Le risque de congestion, que nous allons traiter un peu plus loin, est d'autant plus faible que le nombre de voisines d'une unité de routage est élevé, ce qui semble relativement intuitif. Cependant, avant de déclarer qu'un voisinage de 8 est plus efficace qu'un de 3, ou que tel algorithme est meilleur qu'un autre, il faudra prendre en compte le nombre de transistors nécessaires à leurs implémentations respectives. En effet, si une unité à huit voisines est nettement plus efficace qu'une à trois, mais qu'elle nécessite 100 fois plus de matériel pour sa réalisation physique, nous ne pourrons pas arguer qu'elle lui est préférable.

Pour les besoins de la comparaison, nous avons implémenté en VHDL les quatre algorithmes HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, pour chacun des voisinages. Nous ne décrirons pas ici les équations des signaux des contrôleurs des unités de routage, étant donné que leur fonctionnement est semblable à celui du voisinage de 4. Le tableau 5.13 résume les caractéristiques de chacune des 16 implémentations que nous avons réalisées.

	3	4	6	8
HIDRA	736/23	884/26	1476/40	1952/48
HIDRA-RC	862/23	1078/26	1826/40	2422/48
HIDRA-RT	772/23	958/26	1592/40	2070/48
HIDRA-RTC	834/23	1048/26	1748/40	2240/48

Tableau 5.13 : *Nombre de transistors et de bascules d'une unité de routage, pour chaque algorithme, en fonction du voisinage.*

Afin d'être complète, la comparaison doit également tenir compte du nombre de pins nécessaires à l'implémentation d'une grille d'unités de routage de taille $X \times Y$ dans un circuit physique. Dans le voisinage de quatre, nous disposions de deux sorties dans chacune des directions, `ValOut`X et `PropOut`X. Alors que la première doit être présente dans chacune des directions, pour n'importe quel voisinage, la deuxième ne nécessite que d'être transmise dans les quatre points cardinaux. De plus, dans une de ces quatre directions, il est possible de regrouper les signaux `PropOut`X grâce à une porte OU, et donc de minimiser le nombre de pins nécessaires [5]. La ligne de propagation n'occupe que 8 pins : une entrée et une sortie par point cardinal. Le reste des pins ne sert qu'à transmettre les signaux `ValOut`X et `ValIn`X. Le tableau 5.14 résume le nombre de pins nécessaires, en fonction de la taille de la grille et du voisinage choisi.

5.10.3 4 versus 8

Avant d'observer les résultats des expériences qui ont été menées, il est intéressant d'analyser le potentiel des voisinages 4 et 4-2. Pour ce faire, nous pouvons grouper quatre unités de routage à 4 voisines à la manière du schéma de gauche de la figure 5.32. Ce groupement correspondant à une super-unité de routage qui possède huit connexions, de la même manière qu'une unité de routage 4-2. Le schéma de droite

[5] Cette particularité est explicitée en page 200, dans le chapitre consacré au circuit POEtic.

Voisinage	Pins
3	$2X + 4Y + 8$
4	$4X + 4Y + 8$
6	$8X + 8Y + 8$
4-2	$8X + 8Y + 8$
8	$12X + 12Y$

Tableau 5.14 : *Nombre de pins nécessaires à l'implantation d'une grille de taille $X \times Y$.*

de la figure montre une manière d'organiser des unités à 8 connexions à la façon 4-2 que nous avons définie, en ayant deux liens avec chacune des quatre voisines.

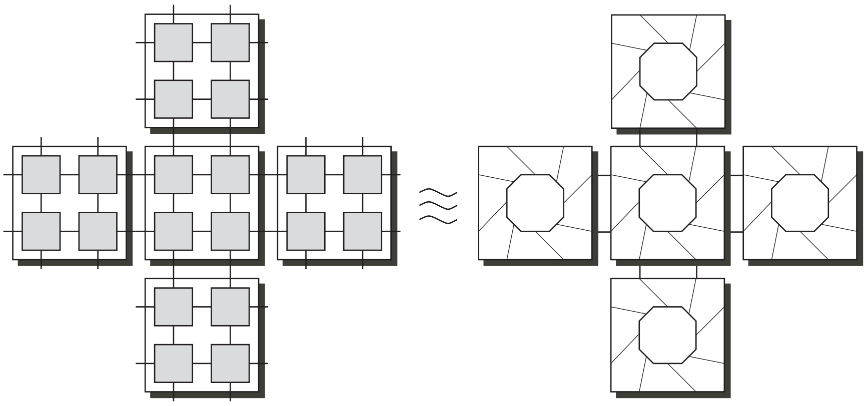


Figure 5.32 : *Regroupement par quatre d'unités de routage à quatre voisines.*

Il est alors aisé de noter qu'une unité 4-2 est plus efficace qu'un groupement de quatre unités à 4 voisines. En effet, la première permet n'importe quelle connectique, chaque sortie pouvant transmettre le signal de n'importe quelle autre entrée, sans aucune contrainte. La deuxième option, en revanche, n'offre pas la même souplesse, car certaines configurations sont impossibles, tel que le montre la figure 5.33. Sur cette figure, en nous conformant au voisinage défini à la figure 5.32, les liaisons seraient équivalentes. Toutefois, la liaison en pointillé ne peut être réalisée avec quatre unités de routage à 4 voisines groupées, alors qu'elle peut l'être avec une unité 4-2.

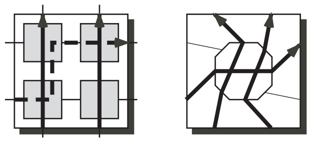


Figure 5.33 : *En suivant les connexions de la figure 5.32, les deux schémas seraient équivalents.*

Avant que les résultats des expériences ne corroborent nos dires, nous pouvons calculer le nombre de connexions impossibles à réaliser par un groupement de quatre unités à 4 voisines. Il est en effet possible, pour un nombre fixé de connexions à réaliser à l'aide des quatre unités, d'en calculer exactement le nombre réalisable. Pour ce faire, nous disposons de huit sorties, chacune d'elles pouvant ou non sélectionner une parmi les sept entrées correspondant autres directions. Une sortie est donc active si elle sélectionne une entrée, et inactive sinon. Il ne reste qu'à compter, pour chaque combinaison de sorties actives, le nombre de sélections d'entrées qui sont réalisables. Le tableau 5.15 expose ces chiffres, pour un nombre de sorties actives compris entre 1 et 8. La première colonne y indique le nombre de sorties actives, la deuxième le nombre de combinaisons réalisables, la troisième le nombre total de combinaisons possibles, et la dernière le pourcentage de succès. Le nombre total de combinaisons est calculé par la fonction suivante, qui correspond au nombre de combinaisons de sorties actives, multiplié par le nombre de combinaisons d'entrées sélectionnées : $\frac{8!}{n!(8-n)!}7^n$.

Nombre de connexions	**Succès**	**Total**	**Succès/Total (%)**
1	56	56	100.0
2	1372	1372	100.0
3	18568	19208	96.6681
4	145830	168070	86.7674
5	673750	941192	71.5848
6	1798392	3294172	54.5931
7	2561950	6588344	38.8861
8	1506243	5764801	26.1283

Tableau 5.15 : *Nombre de connexions possibles.*

Nous pouvons observer, dans ce tableau, que l'efficacité du groupement de quatre unités de routage, en comparaison d'une unité 4-2 baisse grandement avec le nombre de sorties actives. Pour une faible connectivité, les deux approches sont donc équivalentes, et les unités de type 4-2 montrent toute leur efficacité dans le cas d'une forte demande en connexions. Elles sont donc moins sujettes à des problèmes de congestion, comme nous le verrons empiriquement.

Finalement, le voisinage de 8 standard est encore plus performant que le 4-2, de par la possibilité de mettre à profit les diagonales. Ces dernières permettent de réduire la longueur du chemin entre deux points du circuit, et donc de diminuer d'autant le nombre de multiplexeurs réquisitionnés. Alors que dans un voisinage de 4 ou de 4-2, la distance minimale entre deux points $(x1, y1)$ et $(x2, y2)$ correspond à la distance de Manhattan $|x2 - x1| + |y2 - y1|$, le voisinage de 8 offre une distance minimale de $\max(|x2 - x1|, |y2 - y1|)$. Dès lors, comme nous le verrons de manière empirique, le voisinage de 8 est un meilleur candidat que celui de 4-2, puisque pour un même nombre de transistors, il offre un risque plus faible de congestion.

5.11 Analyse

Ayant présenté les quatre algorithmes à comparer, ainsi que les voisinages possibles, nous sommes parés pour entrer dans le vif de l'expérimentation. Nous allons tout d'abord décrire le type d'expériences que nous avons menées, puis nous observerons les différences majeures entre toutes les implémentations, dans le but de définir un optimum en terme de congestion par nombre de transistors. Nous proposerons ensuite un modèle explicatif, capable d'approximer la probabilité de congestion en fonction de divers paramètres, avant de suggérer une analyse grâce à un modèle de percolation, qui nécessiterait une thèse entière à lui seul.

5.11.1 Expérience

Les expériences que nous avons menées se sont effectuées sur la base d'un logiciel, écrit par nos soins, capable de simuler le comportement d'un tableau d'unités de routage. Une interface graphique y offre une visualisation des unités de routage, c'est-à-dire de leur état, et de la configuration de leur switchbox. Deux types de simulations peuvent y être exécutées : purement logicielle, ou basée sur une simulation matérielle des fichiers VHDL.

Dans le premier cas, le logiciel, écrit en C++, simule le comportement exact des unités de routage matériel, en étant capable de calculer le nombre de coups d'horloge nécessaires à l'exécution d'un processus de routage. Dans le deuxième cas, les unités de routage ayant été décrites en VHDL, nous avons utilisé la libraire FLI (Foreign Language Interface), de Modelsim, qui permet d'interfacer une simulation VHDL avec du code C. Une entité peut y être décrite en C plutôt qu'en langage de description matériel, et depuis ce code, il est possible d'accéder à tous les signaux de la simulation, autant en lecture qu'en écriture. Ce code est compilé sous la forme d'une librairie dynamique (DLL), et nous avons utilisé un pipe pour transmettre différentes informations entre la simulation de Modelsim et notre interface graphique, comme nous le montre la figure 5.34.

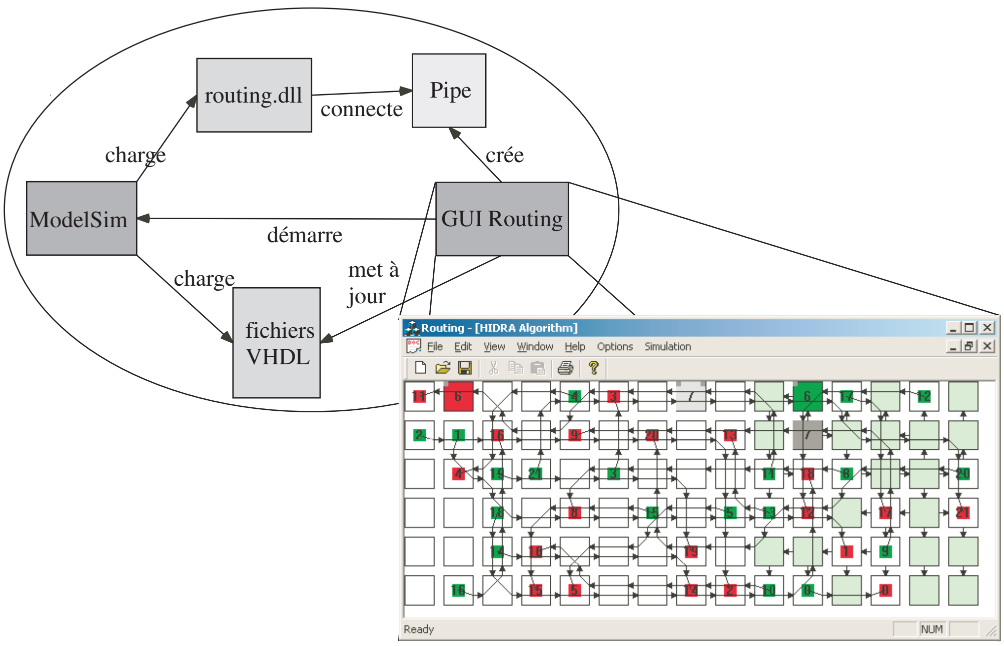


Figure 5.34 : *Lors d'une simulation, l'interface graphique communique avec Modelsim au travers d'un pipe.*

Ce logiciel nous a tout d'abord servit de debugger, lors de la mise au point de nos différents algorithmes. Il s'est ensuite transformé en outil d'expérimentation capable de lancer un grand nombre d'expériences, dans le but de comparer nos différents algorithmes.

Chacune de nos expériences est basée sur le même principe de base. Pour chaque nombre de destinations $ndest$, nous lançons 100 runs, pour chacun desquels nous plaçons à une position aléatoire des sources et des destinations, en fonction de paramètres de l'expérience. Nous effectuons ces 100 runs pour une destination, puis deux, et ainsi de suite, jusqu'à ce que, pour 10 $ndest$ consécutifs, nous obtenions 100% de congestion. Nous partons du principe que si 10 runs consécutifs congestionnent le système, les suivants n'auront pas plus de chance de succès [6]. Les paramètres d'une expérience sont les suivants :

- **Algorithme** : définit le type d'algorithme à utiliser, et peut être HIDRA, HIDRA-RC, HIDRA-RT, ou HIDRA-RTC.
- **Voisinage** : le type de voisinage, qui peut être de 3, 4, 6, 8 ou 4-2.
- **Taille de la grille** : définit la taille du tableau d'unités de routage ($n \times m$).
- **Destinations par source** : définit le nombre de destinations reliées à la même source.
- **Liaisons** : définit le type de liaisons entre sources et destinations. Elles peuvent être libres, auquel cas les sources et destinations sont placées totalement aléatoirement dans le tableau, ou de longueur fixe, la distance minimale entre une source et une destination étant toujours égale à une valeur fixée.

Chacune de nos expériences, pour chaque run, nous fournit les données suivantes :

- **Congestion** : indique si au moins un des chemins n'a pas pu être créé.
- **NbRouted** : Nombre de chemins routés.
- **Clocks** : Nombre de coups d'horloge nécessaire à la terminaison de tous les processus de routage.
- **NbMux** : Nombre de multiplexeurs réquisitionnés par l'ensemble des chemins créés.
- **MaxLength** : Longueur du plus long chemin créé.
- **Longueurs** : Nombre de chemins de chacune des longueurs de 1 à MaxLength.

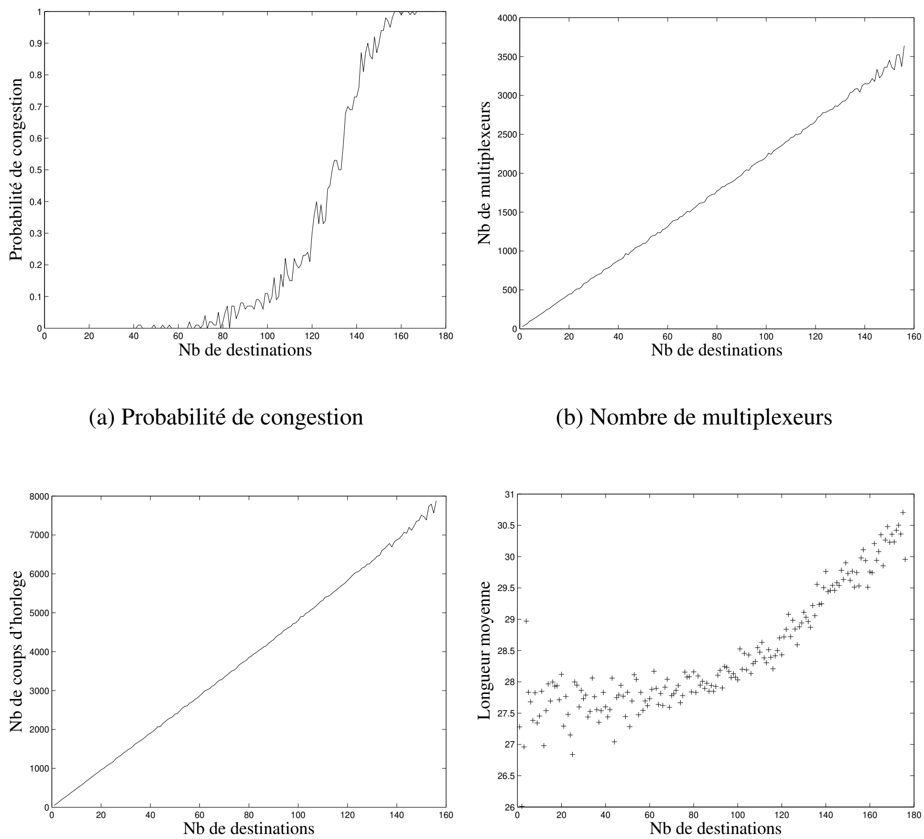
Il est bien clair que le nombre de paramètres de nos expérience offre un espace de recherche des plus considérables. Nous allons donc nous contenter de comparer les algorithmes entre eux, de même que les différents voisinages. Chaque expérience nous donne, pour chaque nombre de destinations, 100 de ces données, que nous pouvons imprimer en fonction de ce nombre de destination, à la manière des figure 5.35(a) à 5.35(d).

5.11.2 Temps d'exécution

Deux de nos algorithmes, HIDRA-RT et HIDRA-RTC, ont pour but de minimiser le nombre de coups d'horloge nécessaires à l'exécution des processus de routage. Les expériences menées prouvent leur efficacité, mais avant de l'observer empiriquement, passons sur quelques constats théoriques. Le nombre de coups d'horloge pris par un processus de routage, T, peut être décomposé en deux parties : la latence fixe Tf, et la phase d'expansion Te.

Pour ce qui est de la première, un processus de routage, c'est-à-dire la génération d'une connexion entre une source et une destination voit toujours les contrôleurs des unités de routage passer de l'état sInit à l'état sAddress, dans lequel elle reste un

[6] Avant de décider de bloquer les expérimentation après 10 $ndest$ congestionnés, nous avons mené plusieurs expériences afin de vérifier que ce nombre était suffisant.



(a) Probabilité de congestion

(b) Nombre de multiplexeurs

(c) Nombre de coups d'horloge

(d) Longueur moyenne

Figure 5.35 : *Données récupérées par une simulation de l'algorithme HIDRA, pour une taille de* 40×40*, trois destinations par source, et un placement aléatoire.*

nombre de coups d'horloge correspondant à la taille de l'adresse, puis à l'état sChooseSource, avant que ne débute la phase d'expansion. Après la terminaison de celle-ci, un coup d'horloge supplémentaire voit la configuration des multiplexeurs lors de la phase de rétropropagation. Chaque destination à connecter nécessite donc un nombre de pas de temps fixe minimum, qui est de $taille(adresse) + 5$. Pour un nombre $ndest$ de destinations, le temps fixe Tf est donc de $ndest \times (taille(adresse) + 5)$. Ce nombre est indépendant de l'algorithme considéré, seule la phase d'expansion étant modifiée entre les quatre variantes étudiées.

Le temps d'exécution de la phase d'expansion, par contre, est influencé par l'algorithme considéré, ainsi que par le nombre de destinations reliées à une même source, dans le cas de HIDRA-RC et HIDRA-RTC.

- Dans le cas de HIDRA, ce temps d'exécution est égal à la distance minimale locale [7] entre la source et la destination. Le temps d'exécution de la phase d'ex-

[7] Le chemin trouvé entre la source et la destination ne correspond pas forcément au chemin minimal, étant donné que les multiplexeurs déjà configurés peuvent bloquer certains passages. Il s'agit donc d'un minimum local, et non global.

pansion, Te_{hidra}, peut alors être directement calculé à partir du nombre de destinations à connecter et de la longueur des chemins créés, comme ceci : $Te_{hidra} = \sum_{i=1}^{ndest} length_i$.

- L'algorithme HIDRA-RC propose un temps d'exécution plus faible ou égal à celui de HIDRA, pour la phase d'expansion. Si une source est reliée à au plus une destination, le temps d'exécution est identique, puisque le comportement des deux algorithmes y est similaire. En revanche, si plusieurs destinations sont reliées à une même source, le temps d'exécution peut être réduit de par le fait que toutes les unités de routage déjà reliées à la source sont atteintes en un seul coup d'horloge. Le temps total est donc borné supérieurement par Te_{hidra}, et peut être calculé en fonction du nombre de multiplexeurs utilisés par l'ensemble des chemins. En effet, lors de la première connexion d'une source, le temps de la phase d'expansion correspond à la longueur du chemin créé, et donc au nombre de multiplexeurs réquisitionnés. Les connexions successives de la même source seront exécutées en un temps égal à la longueur du chemin séparant la destination de la plus proche des unités de routage déjà reliées à la source. Et cette longueur équivaut au nombre de multiplexeurs utilisés pour la création du nouveau chemin. Nous avons donc que $Te_{hidra-rc} = \sum mux_{conf}$.
- Concernant HIDRA-RT, le temps d'exécution de la phase d'expansion est nettement inférieur aux deux premières approches. Dans le cas idéal où aucun multiplexeur ne bloque le passage entre la source et la destination, un maximum de 2 coups d'horloge sont nécessaires à la source pour atteindre n'importe quel endroit du tableau d'unités de routage. Si la destination ne peut être atteinte en 2 coups d'horloge, la seule affirmation que nous pouvons tenir est que le nombre de coups d'horloge correspond au nombre de segments droits reliant la source à la destination.
- Enfin, pour HIDRA-RTC, de même que pour HIDRA-RT, dans le cas idéal, un maximum de deux coups d'horloge sont nécessaires à la phase d'expansion. Toutefois, de manière générale, ce nombre ne peut être couplé à aucune autre mesure.

Empiriquement, nous pouvons comparer les temps d'exécution des quatre algorithmes, avec et sans Tf, et ce pour plusieurs paramètres d'expérimentation. Dans le tableau 5.16, nous observons, pour plusieurs tailles de tableau et nombres de destinations connectées à une source (DpS), le temps moyen d'exécution T_m ainsi que le temps d'expansion Te_m d'un processus de routage. Les colonnes $\%T_m$ et $\%Te_m$ comparent les algorithmes à la solution HIDRA, en terme de temps total, pour un identifiant de 16 bits, et en terme de temps d'expansion. Les sources et destinations sont ici placées au hasard dans la grille d'unités de routage.

5.11.3 Longueur de chemins

A l'instar de la rapidité d'exécution, la longueur des chemins générés par les processus de routage est fortement liée à l'algorithme exploité. Seul HIDRA garantit de trouver le chemin de taille minimale, car l'expansion est lancée par la source uniquement, et, à chaque pas de temps, le front d'onde atteint une nouvelle couche d'unités de routage. Les unités postées à une distance d de la source sont atteintes après d coups d'horloge, ce qui garantit l'optimalité de la solution. Les autres solutions ont pour but l'optimisation d'autres mesures : le nombre de multiplexeurs utilisés pour

Taille	DpS	**HIDRA**		**HIDRA-RC**				**HIDRA-RT**				**HIDRA-RTC**			
		T_m	Te_m	T_m	Te_m	$\%T_m$	$\%Te_m$	T_m	Te_m	$\%T_m$	$\%Te_m$	T_m	Te_m	$\%T_m$	$\%Te_m$
20×20	1	34.67	13.67	34.55	13.55	99.6	99.1	23.26	2.26	67.1	16.5	23.24	2.24	67.0	16.4
20×20	3	34.75	13.74	30.41	9.41	87.5	68.5	23.29	2.29	67.0	16.7	22.84	1.84	65.7	13.4
20×20	5	34.78	13.78	28.79	7.79	83.8	56.6	23.31	2.31	67.0	16.8	22.69	1.69	65.2	12.3
40×40	1	47.99	26.99	47.92	26.93	99.9	99.8	23.3	2.30	48.6	8.5	23.31	2.31	48.6	8.6
40×40	3	48.16	27.16	39.50	18.50	82.0	68.1	23.41	2.41	48.6	8.9	22.96	1.96	47.7	7.2
40×40	5	48.26	27.26	36.21	15.21	75.0	55.8	23.42	2.42	48.5	8.9	22.81	1.81	47.3	6.7
60×60	1	61.06	40.06	61.16	40.16	100.2	100.3	23.29	2.29	38.1	5.7	23.30	2.3	38.2	5.7
60×60	3	61.51	40.51	48.50	27.50	78.8	67.9	23.43	2.43	38.1	6.0	23.00	2.0	37.4	4.9
60×60	5	61.73	40.73	43.55	22.55	70.6	55.4	23.46	2.46	38.0	6.0	22.86	1.86	37.0	4.6
80×80	1	74.44	53.44	74.45	53.45	100.0	100.0	23.27	2.27	31.3	4.3	23.28	2.28	31.3	4.3
80×80	3	74.81	53.81	57.44	36.44	76.8	67.7	23.45	2.45	31.4	4.6	23.01	2.01	30.8	3.7
80×80	5	75.03	54.03	50.87	29.87	67.8	55.3	23.47	2.47	31.3	4.6	22.90	1.90	30.5	3.5

Tableau 5.16 : *Comparaison des algorithmes, en terme de coups d'horloge.*

HIDRA-RC, le temps d'exécution pour HIDRA-RT, et une conjugaison des deux pour HIDRA-RTC. La longueur moyenne des chemins peut alors y être sous-optimale.

Dans le cas de HIDRA-RC, la différence apparaît lorsque plus d'une destination sont connectées à une source. A partir de la deuxième, la longueur du chemin créé n'est pas égale à la distance à la source, mais bien à la distance minimale de la destination à l'une des unités de routage déjà reliées à la source. Ce nombre possède une borne minimale étant la distance entre la source et la destination, mais aucune borne maximale.

Aucune propriété théorique ne peut être énoncée pour les deux options HIDRA-RT et HIDRA-RTC, si ce n'est que la longueur d'un chemin est forcément aussi grande que celle d'un chemin créé avec HIDRA.

Empiriquement parlant, le tableau 5.17 présente les longueurs moyennes des chemins créés, pour les mêmes données que lors de la comparaison des temps d'exécution. Nous donnons ici la longueur moyenne, ainsi que la comparaison avec la solution de base HIDRA.

Taille	DpS	**HIDRA**	**HIDRA-RC**		**HIDRA-RT**		**HIDRA-RTC**	
		L_m	L_m	$\%L_m$	L_m	$\%L_m$	L_m	$\%L_m$
20 × 20	1	15.53	15.51	99.9	15.63	100.7	15.70	101.1
20 × 20	3	15.07	15.75	104.5	15.45	102.5	15.99	106.1
20 × 20	5	15.11	16.62	110.1	15.41	102.0	16.66	110.3
40 × 40	1	29.07	29.35	100.9	29.57	101.7	29.56	101.7
40 × 40	3	28.43	30.10	105.9	29.00	102.0	30.55	107.5
40 × 40	5	28.60	31.90	111.5	29.17	102.0	31.97	111.8
60 × 60	1	42.91	43.03	100.3	43.42	101.2	43.49	101.4
60 × 60	3	41.74	44.39	106.3	42.67	102.2	44.97	107.7
60 × 60	5	42.12	47.14	111.9	42.89	101.8	47.13	111.9
80 × 80	1	57.16	57.01	99.7	57.21	100.1	57.41	100.4
80 × 80	3	55.05	58.74	106.7	56.13	102.0	59.32	107.8
80 × 80	5	55.47	62.41	112.5	56.44	101.7	62.26	112.2

Tableau 5.17 : *Comparaison des algorithmes, en terme de longueur de chemins.*

Ce tableau présente la moyenne générale, sans qu'aucune distinction ne soit faite entre les différents nombres de destinations à connecter. La figure 5.36 montre l'évolution des longueurs, pour un tableau de taille 80×80, en fonction de l'algorithme. Nous pouvons y observer qu'à faible densité de destinations, les variantes HIDRA et HIDRA-RT sont plus efficaces que les deux autres, mais que l'écart se réduit lorsque cette densité augmente. Ceci s'explique aisément par l'arrivée des solutions congestionnées. En effet, nous ne prenons en compte que les routages n'ayant pas subi de congestion, afin de ne pas biaiser les données. La courbe traitillée de la figure 5.36 montre la probabilité de congestion de l'algorithme HIDRA, qui passe de 0 pour les petites densités, à 1 pour les grandes. Plus la densité augmente, plus les solutions HIDRA-RC et HIDRA-RTC peuvent tirer profit de la réutilisation des multiplexeurs, et donc ne pas augmenter la longueur moyenne des chemins. Les deux autres, en revanche, doivent trouver des chemins de plus en plus longs, afin d'éviter les multiplexeurs déjà configurés.

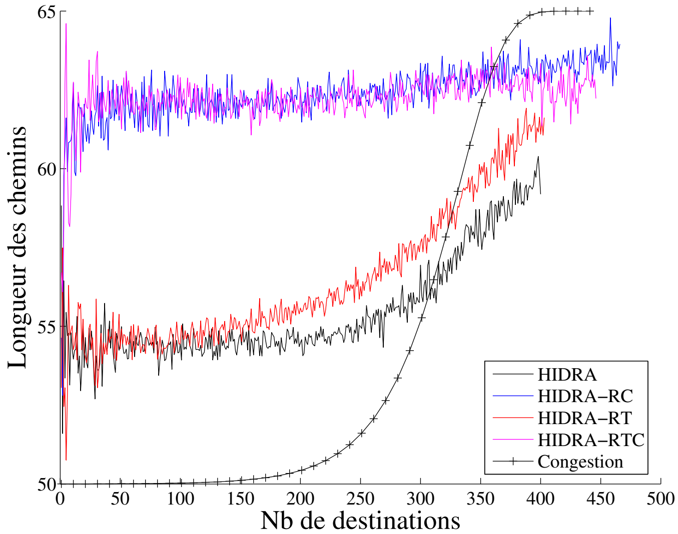


Figure 5.36 : *Comparaison de la longueur des chemins générés par les quatre algorithmes, pour un voisinage de 4, et 5 destinations par source, disposées aléatoirement. La congestion indiquée est celle de HIDRA.*

5.11.4 Nombre de multiplexeurs

Fortement lié à la longueur des chemins, le nombre de multiplexeurs réquisitionnés pour la connexion d'un ensemble de sources à un ensemble de destinations va nous mener tout naturellement vers l'analyse de la congestion. En effet, le nombre de multiplexeurs occupés influe directement sur la probabilité qu'un nouveau chemin ne trouve pas de solution.

Nous pouvons nous attendre à ce que HIDRA-RC soit le plus efficace en regard du nombre de multiplexeurs réquisitionnés, car il réutilise au maximum les chemins déjà créés à partir d'une source. En comparaison de HIDRA-RTC, qui lance également l'expansion depuis toutes les unités de routage déjà reliées à la source, il est au moins aussi efficace, puisque sa phase d'expansion trouve le chemin de longueur minimum entre la destination et l'unité la plus proche parmi toutes les unités déjà reliées à cette source. Il semble également instinctif de prédire que HIDRA-RT occupe le plus grand nombre de transistors, puisqu'il ne garantit pas le chemin minimal, ni ne réutilise les chemins déjà créés.

Le tableau 5.18 compare nos quatre algorithmes, en indiquant le nombre moyen de multiplexeurs configurés par destination, ainsi que le pourcentage en comparaison de HIDRA. Il est intéressant de noter que HIDRA-RT se comporte de façon semblable à HIDRA, probablement de par le fait que dans la plupart des processus de routage, la destination peut être trouvée en deux pas pour HIDRA-RT, et que si tel est le cas, alors la solution est optimale. Si nous revenons au tableau 5.16, nous pouvons en effet constater qu'en moyenne la phase d'expansion s'exécute en 2.5 coups d'horloge, et que donc la solution trouvée est presque toujours très proche de l'optimale.

Concernant les variantes HIDRA-RC et HIDRA-RTC, nos prédictions sont confirmées, et leur nombre de multiplexeurs configurés est nettement inférieur à celui de HIDRA, lorsque le nombre de destinations par source augmente, économisant jusqu'à 25% de multiplexeurs pour un tableau de taille 60×60 et 5 destinations par source.

Taille	DpS	HIDRA	HIDRA-RC		HIDRA-RT		HIDRA-RTC	
		L_m	L_m	$\%L_m$	L_m	$\%L_m$	L_m	$\%L_m$
20 × 20	1	13.67	13.55	99.1	13.88	101.5	13.84	101.3
20 × 20	3	11.02	9.41	85.4	11.15	101.2	10.11	91.7
20 × 20	5	9.74	7.79	80.0	9.74	100.0	8.63	88.6
40 × 40	1	26.99	26.93	99.8	27.33	101.3	27.39	101.5
40 × 40	3	22.19	18.49	83.4	22.28	100.4	20.10	90.6
40 × 40	5	19.69	15.21	77.3	19.66	99.9	17.17	87.2
60 × 60	1	40.06	40.16	100.3	40.63	101.4	40.52	101.1
60 × 60	3	33.27	27.50	82.7	33.38	100.3	29.98	90.1
60 × 60	5	29.71	22.55	75.9	29.57	99.5	25.68	86.4
80 × 80	1	53.44	53.45	100.0	53.84	100.7	53.84	100.8
80 × 80	3	44.34	36.44	82.2	44.41	100.2	39.81	89.8
80 × 80	5	39.7	29.87	75.2	39.44	99.4	34.21	86.2

Tableau 5.18 : *Comparaison des algorithmes, en terme de nombre moyen de multiplexeurs configurés.*

La figure 5.37 présente l'évolution du nombre moyen de multiplexeurs configurés par destination, en fonction du nombre de ces destinations, et ce pour 5 destinations reliées à chaque source, dans un tableau de taille 80 × 80. Nous pouvons y observer que les valeurs pour un faible nombre de destinations sont nettement au-dessus du niveau moyen des valeurs à partir de 50 destinations. Ceci est dû au fait qu'un seul chemin à créer ne peut réutiliser aucun multiplexeur. Il est également intéressant, que contrairement aux longueurs de chemins, qui ont tendance à s'allonger avec le nombre de destinations à connecter, le nombre de multiplexeurs est stable, ce qui montre bien que les algorithmes peuvent exploiter les unités de routage déjà reliées à la source du processus courant.

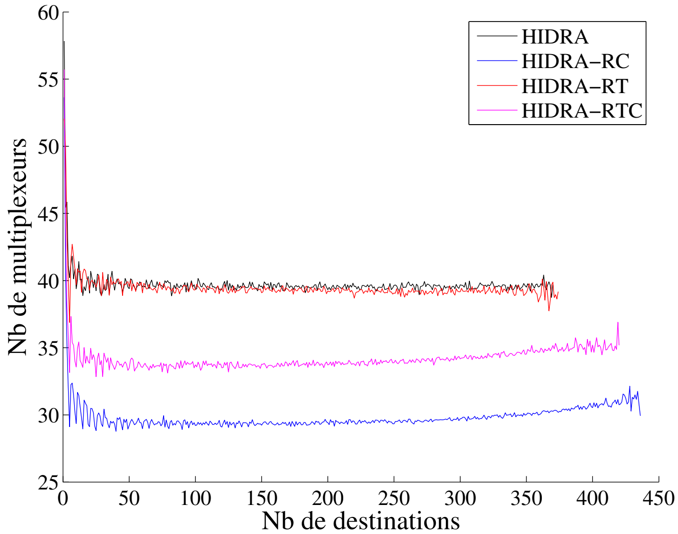


Figure 5.37 : *Comparaison des quatre algorithmes, sur le plan du nombre de multiplexeurs configurés, divisé par le nombre de destinations connectée, pour un voisinage de 4, et 5 destinations par source, disposées aléatoirement.*

Finalement, conformément à nos attentes, HIDRA-RC reste la meilleure solution en terme de nombre de multiplexeurs, et nous amène tout naturellement à analyser la congestion de nos algorithmes, où nous verrons que cet algorithme y est le plus efficace.

5.11.5 Congestion

Alors que les données que nous venons de traiter ne sont pas cruciales, la congestion constitue un point critique de nos algorithmes. Ce problème intervient lorsqu'un nouveau chemin à créer ne trouve aucune solution, c'est-à-dire qu'il n'existe pas de combinaison de multiplexeurs inoccupés, ou reliés à la source, permettant de relier la source à la destination courante. La figure 5.38 illustre un de ces cas, où la source numéro 4 n'a aucune possibilité de joindre la destination de même numéro.

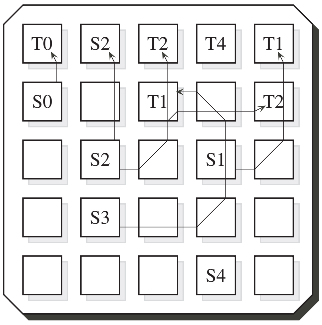


Figure 5.38 : *Exemple de congestion : la source 4 ne peut se connecter à la destination 4.*

Chaque expérience menée nous a fourni, pour chaque nombre de destinations $ndest$, 100 échantillons, caractérisés par un problème de congestion ou non. Il s'agit d'un résultat, qui vaut 1 si l'échantillon est congestionné, et 0 s'il est entièrement routé.

Nous nous intéressons ici à la probabilité de congestion en fonction de $ndest$, de manière à pouvoir prédire si une application particulière, dont on connaît la connectivité, a une chance d'être implémentée sur une grille de taille fixe. Pour un nombre de chemins à router donné, nous avons une probabilité p_c qu'un problème de congestion survienne, et une probabilité $p_{\overline{c}} = 1 - p_c$ que le routage s'effectue normalement. Nous pouvons considérer la variable aléatoire de Bernoulli définie par :

$$C = \begin{cases} 1 & \text{si l'échantillon est congestionné} \\ 0 & \text{si l'échantillon est entièrement routé} \end{cases}$$

L'espérance de cette variable aléatoire est alors :

$$E(C) = \sum_i p_i c_i = 0 p_{\overline{c}} + 1 p_c = p_c$$

La variance de cette variable aléatoire est ensuite définie comme suit :

$$\begin{aligned} V(C) &= \sum_i p_i (c_i - E(C))^2 = \sum_i p_i c_i^2 - (\sum_i p_i c_i)^2 && (5.1) \\ &= 0^2 p_{\overline{c}} + 1^2 p_c - (0 p_{\overline{c}} + 1 p_c)^2 = p_c - p_c^2 && (5.2) \end{aligned}$$

Réciproquement, nous avons :

$$p_c = \frac{1 \pm \sqrt{1 - 4V(C)}}{2} \tag{5.3}$$

Pour un échantillon de taille n suffisamment grande, la statistique de C, $\overline{C}$, qui est un estimateur sans biais de p_c, obéit approximativement à une loi du type $N(p_c, p_c(1-p_c)/n)$. La variance inconnue de C peut alors être remplacée par son estimation $\bar{c}(1-\bar{c})$, où $\bar{c}$ est une estimation ponctuelle de $p_c = E(C)$.

Nous pouvons donc estimer la probabilité de congestion p_c par la moyenne des résultats de l'échantillon, pour un $ndest$ donné. Désirant trouver une courbe approximant la probabilité de congestion, en fonction de $ndest$, nous avons testé plusieurs modèles explicatifs.

Modèles explicatifs

La courbe de congestion, telle celle illustrée à la figure 5.39, possède deux caractéristiques particulières. Premièrement, il s'agit d'une fonction de transition, semblable à une sigmoïde, ou à une tangente hyperbolique, dont la valeur est incluse dans $[0, 1]$. Deuxièmement, la courbure est plus faible lorsque la probabilité de congestion est inférieure à 0.5 que lorsqu'elle en est supérieure. Une simple sigmoïde ne se prête donc pas parfaitement à un ajustement sur les données, puisque cette fonction est totalement symétrique.

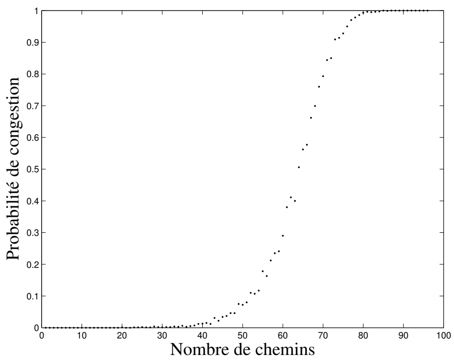


Figure 5.39 : *Probabilité de congestion en fonction du nombre de chemins.*

Dans notre recherche d'une bonne candidate, nous avons tout d'abord tenté des fonctions de type sigmoïde :

$$P_c(x|a,b) = \frac{1 + \tanh(a + bx)}{2} \tag{5.4}$$

$$P_c(x|a,b,c,d) = \frac{1 + \tanh(a + bx))}{2} + (1 - \frac{1 + \tanh(a + bx))}{2})\frac{1 + \tanh(c + dx)}{2} \tag{5.5}$$

La première est une fonction sigmoïde, et donc symétrique. Elle offre une bonne approximation pour les tableaux de petite taille, mais peine à être efficace lorsque le nombre d'unités de routage devient plus important et que la courbe devient de plus en plus asymétrique. La deuxième combine deux sigmoïdes, afin de pallier au manque de symétrie de la première. Elle permet de nettement mieux approcher les valeurs expérimentales, mais a le désavantage de nécessiter quatre paramètres au lieu de deux. La figure 5.40 montre les deux courbes sur une expérience de taille 80×80, avec voisinage de 4, et 3 destinations par source. La courbe traitillée correspond à l'équation 5.4, tandis que la courbe pleine, qui approxime mieux les données, correspond à l'équation 5.5.

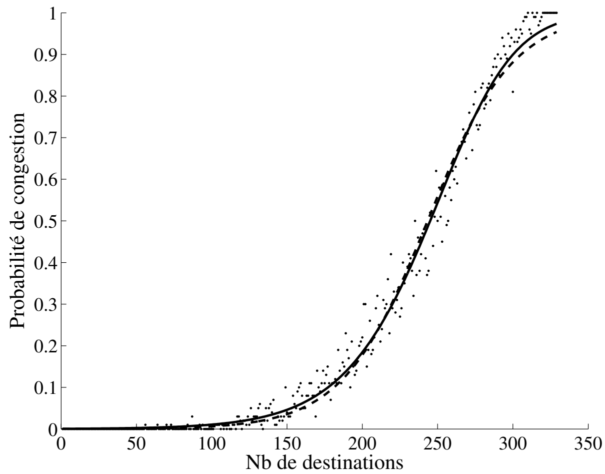


Figure 5.40 : *Comparaison des estimateurs 1 et 2, sur une taille de* 80×80*, 4 voisins, 3 destinations par source.*

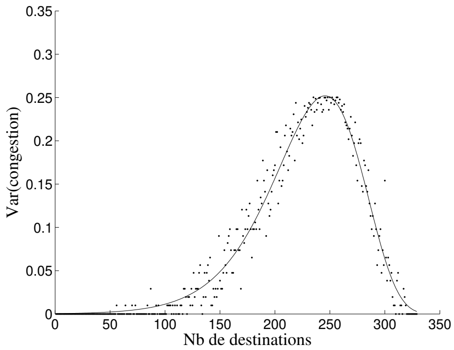
Ces deux courbes ont toutefois l'inconvénient de sous-estimer la probabilité de congestion lorsque celle-ci approche 1. Nous nous sommes alors penchés sur l'observation de la variance des échantillons, en observant qu'avec une symétrie verticale, elle ressemblait fort à une loi de distribution Lognormale. Nous avons donc tenté d'estimer la variance par l'équation 5.6, qui correspond à une loi Lognormale légèrement modifiée, et de calculer ensuite la probabilité de congestion grâce à l'équation 5.7, dérivée de l'équation 5.3.

$$V(x|a,b,\sigma,\mu) = \frac{a}{(b-x)\sigma\sqrt{2\Pi}} e^{\frac{-(\ln(b-x)-\mu)^2}{2\Pi^2}} \tag{5.6}$$

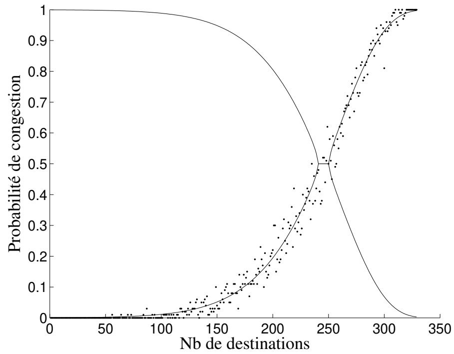
$$P_c(x|a,b,\sigma,\mu) = \frac{1 \pm \sqrt{1 - 4\frac{a}{(b-x)\sigma\sqrt{2\Pi}} e^{\frac{-(\ln(b-x)-\mu)^2}{2\Pi^2}}}}{2} \tag{5.7}$$

La figure 5.41(a) illustre la variance des échantillons, et la courbe calculée grâce à l'équation 5.6. En la transformant, nous obtenons le résultat pour la probabilité à la figure 5.41(b), qui contient deux courbes, correspondant aux deux possibilités de l'équation 5.7. L'estimation est ici très bonne aux probabilités extrêmes, mais manque de justesse lorsque la probabilité approche 0.5, et n'offre pas forcément la possibilité d'en construire une fonction continue.

Dans notre recherche d'une bonne candidate, nous avons finalement posé nos yeux



(a) Variance des échantillons, et courbe approchée



(b) Probabilité de congestion et courbe approchée

Figure 5.41 : *Approximation de la variance par une loi de distribution Lognormale.*

sur la formule de la fonction de distribution cumulative [8] de Weibull :

$$F(x|gamma) = 1 - e^{-(x^{\gamma})}, \text{ pour } x \geq 0; \gamma > 0$$

La courbe de Weibull respecte parfaitement les deux caractéristiques de celle de congestion, et nous l'avons donc choisie comme moyen d'approximer les données reçues des expériences. En la modifiant très légèrement, nous obtenons la fonction suivante :

$$P_c(x|a, b, \gamma) = 1 - e^{-(|ax+b|^{\gamma})} \tag{5.8}$$

La figure 5.42 illustre son adaptation parfaite aux données, la courbe étant continue et offrant la même efficacité que la précédente au niveau des probabilités proches de 0 ou de 1.

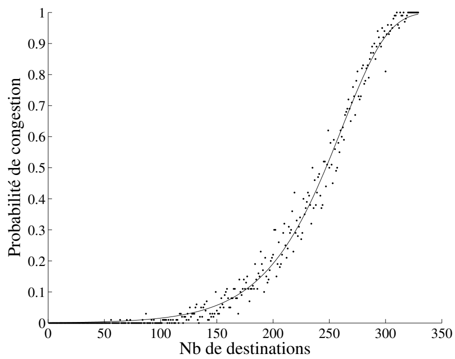


Figure 5.42 : *Probabilité de congestion, et approximation par une fonction de distribution cumulative de Weibull.*

[8] Une fonction de distribution cumulative est la probabilité que la variable x (en abscisse) prenne une valeur inférieure ou égale à x.

Analyse

Grâce à l'ensemble des expériences effectuées, nous pouvons à présent déterminer la probabilité de congestion d'un système plaçant des sources et $ndest$ destinations de manière aléatoire dans une grille de taille $n \times n$, pour un nombre de destinations par source de 1, 3, ou 5, en fonction de l'algorithme choisi. Les expériences ont été menées avec des tableaux de taille allant de 5×5 à 80×80. Matlab a été utilisé comme outil de régression non-linéaire capable de nous fournir les paramètres a, b, et γ, en fonction des échantillons récoltés, grâce à la méthode des moindres carrés. Concernant le nombre d'échantillons que nous avons choisi, nous avons mené quelques expériences avec 100 et 1000 échantillons, et avons comparé les différences. Pour un nombre quelconque de destinations, la différence de probabilité de congestion est de moins de 2%, ce qui est tout à fait raisonnable. Nous avons donc décidé de travailler avec 100 échantillons, de manière à accélérer les expériences. De plus, Matlab nous fournit l'intervalle de confiance de la courbe. Pour un niveau de confiance de 95%, et avec 100 échantillons, nous obtenons un intervalle de confiance inférieur à 3%, pour la grande majorité des expériences. Nous pouvons donc dire que la probabilité de congestion se trouve, avec 95% de certitude, sur un point situé à $\pm 3\%$ de la courbe calculée.

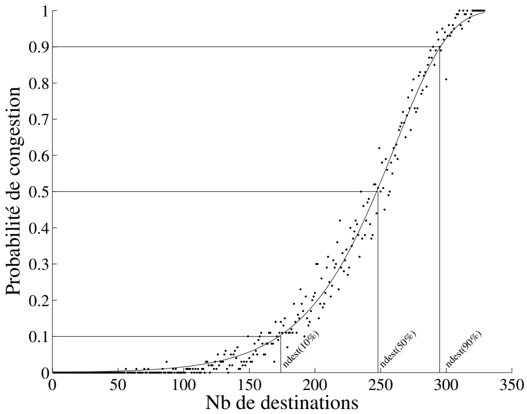
Avec nos expériences, nous disposons de toutes les courbes d'estimation de nos échantillons, et pouvons les exploiter de deux manières. Premièrement, pour une application particulière implémentée dans une grille de taille définie, nous pouvons estimer la probabilité de congestion en fonction du nombre de sources et de destinations présentes. Deuxièmement, nous pouvons établir le nombre de destinations qu'il est possible de connecter en fonction d'une probabilité donnée, grâce à la formule inverse de 5.8 :

$$x(P_c|a, b, \gamma) = \frac{\sqrt[\gamma]{\log(1 - P_c)} - b}{a} \tag{5.9}$$

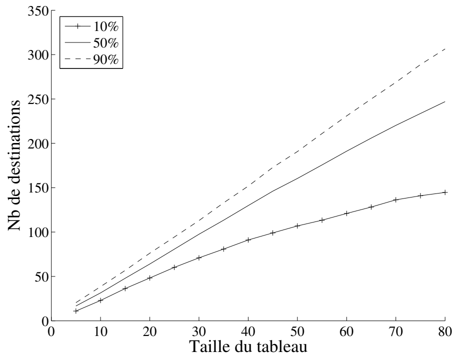
Nous pouvons, grâce à cette formule, calculer à partir de quel nombre de destinations le risque de congestion devient trop élevé, et utiliser cette valeur pour comparer nos algorithmes, ainsi que les différents voisinages. La figure 5.43 illustre le nombre de destinations pour lequel la probabilité de congestion est de 10, 50 ou 90%, pour l'algorithme HIDRA, 3 destinations par source, et des tableaux de taille $5n \times 5n$, pour $n \in [1, 16]$.

Notre intérêt est plus de comparer les algorithmes et les voisinages que de réellement pouvoir évaluer la probabilité de congestion en fonction des multiples facteurs – algorithme, voisinage, taille en x, taille en y, nombre de destinations par source. Nous disposons en tout de 16 options, les quatre algorithmes combinés aux quatre voisinages, et nous pouvons effectuer des comparaisons deux à deux de la manière suivante : pour les probabilités de congestion de $]0, 1[$, nous calculons le nombre de destinations $ndest1(p_c)$ et $ndest2(p_c)$ correspondantes pour les deux options, puis traçons un graphe de $ndest1(p_c)/ndest2(p_c)$ ou de $ndest1(p_c), ndest2(p_c)$. A titre d'exemple, nous obtenons, pour le cas d'un voisinage de 4, une taille de 40×40, et 3 destinations par source, la comparaison de HIDRA et HIDRA-RC présentée à la figure 5.44. Nous pouvons y observer que HIDRA-RC est plus efficace que HIDRA, ce qui correspond à nos estimations intuitives.

Pour une comparaison générale, nous prenons HIDRA avec un voisinage de 4



(a) Trois points de congestion et leur nombre de destinations correspondant



(b) Nombre de destinations acceptables donnant un taux de congestion donné, en fonction de la taille du tableau d'unités de routage

Figure 5.43 : *Estimation du nombre de destinations acceptables pour un taux de congestion donné.*

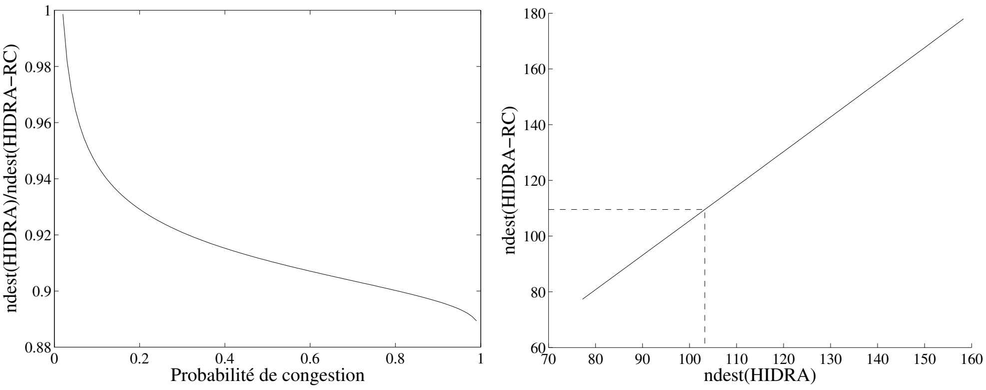


Figure 5.44 : *Comparaison de HIDRA et HIDRA-RC en terme de congestion.*

comme référence, et évaluons les points de congestion 10, 50, et 90% pour toutes les combinaisons $Comb(algorithme, voisinage)$, et transcrivons dans le tableau 5.19 la valeur $Comb(algorithme, voisinage)/Comb(HIDRA, 4)$, et ce pour 1, 3, et 5 destinations par source. Les nombres inscrits dans ce tableau représentent donc l'efficacité d'une implémentation, relativement à (HIDRA,4). Un nombre x signifie que le risque de congestion de $y\%$ est atteint pour un nombre de destinations x fois supérieur à celui voyant le même risque pour (HIDRA,4).

Ce tableau nous permet de constater l'efficacité de l'algorithme HIDRA-RC, qui, comme nous l'avions prévu, possède un risque de congestion plus faible que les autres, étant donné qu'il tire un maximum parti des multiplexeurs précédemment configurés. Cette différence est d'autant plus marquée que le nombre de destinations par source est élevé, atteignant un facteur 4.3 pour un risque de 10% et un voisinage de 8. Les trois autres algorithmes ont en revanche des performances équivalentes en terme de

V-D	HIDRA			HIDRA-RC			HIDRA-RT			HIDRA-RTC		
	10	50	90	10	50	90	10	50	90	10	50	90
3-1	0.37	0.41	0.44	0.37	0.41	0.44	0.34	0.39	0.41	0.37	0.39	0.42
3-3	0.38	0.43	0.46	0.41	0.46	0.49	0.30	0.37	0.42	0.33	0.39	0.45
3-5	0.38	0.44	0.47	0.44	0.50	0.53	0.30	0.37	0.43	0.33	0.40	0.47
4-1	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.01	1.04	0.99	1.02	1.04
4-3	1.00	1.00	1.00	1.06	1.10	1.12	0.94	0.98	0.99	1.02	1.06	1.07
4-5	1.00	1.00	1.00	1.16	1.16	1.16	0.95	0.98	1.00	1.10	1.11	1.11
6-1	2.28	2.05	1.94	2.30	2.04	1.92	2.16	1.95	1.84	2.13	1.93	1.85
6-3	2.29	2.00	1.89	2.51	2.20	2.07	1.99	1.85	1.75	2.17	1.96	1.87
6-5	2.15	1.93	1.83	2.49	2.21	2.09	1.97	1.85	1.80	2.19	2.00	1.91
8-1	4.01	3.46	3.20	3.97	3.45	3.21	3.45	3.02	2.77	3.47	3.02	2.78
8-3	3.95	3.39	3.15	4.36	3.71	3.43	3.50	3.09	2.91	3.67	3.23	3.03
8-5	3.71	3.24	3.04	4.27	3.71	3.46	3.48	3.10	2.93	3.73	3.28	3.08

Tableau 5.19 : *Comparaison des algorithmes et des voisinages. La première colonne indique le voisinage et le nombre de sources reliées à une même destination.*

congestion, ce qui va dans le sens des résultats que nous avons obtenus en analysant le nombre de multiplexeurs configurés, en page 154.

La grande importance de ces résultats tient plutôt dans la comparaison des voisinages que dans les différences entre algorithmes. Pour ce faire, et en nous référant au nombre de transistors nécessaires à l'implémentation des différentes unités de routage (Tableau 5.13, page 145), nous pouvons mettre en relation l'efficacité en terme de congestion, et le nombre de transistors nécessaires. Trois constatations vont nous donner ces résultats :

- Pour un même algorithme et une même taille de tableau, nous pouvons observer que :
  - Le voisinage de 3 est 2 fois moins performant que le voisinage de 4.
  - Celui de 6 est 2 fois plus performant que celui de 4.
  - Celui de 8 est 3.5-4 fois plus performant que celui de 4.
- Si nous reprenons la figure 5.43(b), nous notons que, pour un risque de congestion donné, le nombre de destinations correspondant est égal ou inférieur à une valeur proportionnelle à n, pour un tableau de taille $n \times n$. En effet, sur cette figure, pour les probabilités de congestion de 50 et de 90%, le nombre de destinations est directement proportionnel à n, et pour 10% de probabilité, la valeur suit une droite légèrement infléchie. Cette affirmation a été vérifiée sur les autres expériences, et nous pouvons donc affirmer, que si nous connaissons $ndest(p_c, n)$, nous pouvons calculer $ndest(p_c, 2n)$ comme étant $2ndest(p_c, n)$.
- Enfin, concernant le nombre de transistors, nous pouvons, sur la base du tableau de la page 145, calculer les valeurs relatives entre les différents voisinages. Le tableau 5.20 nous donne ces valeurs pour l'algorithme HIDRA-RC, où la proportion calculée correspond au nombre de transistors (respectivement bascules) du voisinage de la colonne divisé par celui de la ligne. Un des résultats intéressant y est que nous avons environ un facteur 2 entre les voisinages 3 et 6, et le même facteur entre les voisinages 4 et 8.

A la vue des ces trois points nous pouvons affirmer qu'un tableau de taille $2n \times 2n$

	3	4	6	8
3	1.00/1.00	1.25/1.13	2.12/1.73	2.81/2.08
4	0.80/0.88	1.00/1.00	1.69/1.53	2.25/1.85
6	0.47/0.58	0.59/0.65	1.00/1.00	1.33/1.20
8	0.36/0.50	0.45/0.54	0.75/0.83	1.00/1.00

Tableau 5.20 : *Pour HIDRA-RC, rapport entre les nombres de transistors et les nombres de bascule, en fonction du voisinage.*

avec un voisinage de 8 voit des problèmes de congestion arriver lorsque le nombre de destinations est deux fois plus grand que pour un tableau de $n \times n$. La comparaison des voisinages de 4 et 8 nous donne ceci :

$$\begin{aligned} ndest(p_c, n, voisinage = 8) &= ndest(p_c, 2n, voisinage = 8)/2 \\ &\simeq 2ndest(p_c, 2n, voisinage = 4) \end{aligned}$$

L'implication de ce résultat est grande. Il signifie qu'il est préférable de disposer d'un tableau de taille $\frac{n}{2} \times \frac{n}{2}$ avec un voisinage de 8 que d'un tableau de taille $n \times n$ avec un voisinage de 4, le premier étant 2 fois plus efficace en terme de congestion que le deuxième. De plus, sur le plan de l'implémentation physique, si le nombre de transistors requis par le deuxième est nbt, alors ce nombre n'est que de $nbt \times 2/4 = nbt/2$ pour le premier, sachant que nous avons un facteur 2 entre la taille des unités de routage, et un nombre 4 fois moindre de ces unités présentes dans le tableau. Avec une quantité de logique deux fois moindre, il est donc possible de router deux fois plus de destinations sur un tableau de voisinage 8.

Finalement, notons que nous n'avons pas intégré le voisinage 4-2 dans nos comparaisons, pour la simple raison qu'il est moins efficace que le voisinage de 8, en terme de congestion, alors qu'il nécessite la même quantité de transistors. La figure 5.45 montre la supériorité du voisinage de 8, pour un tableau de taille 40×40, l'algorithme HIDRA, et 3 destinations par source.

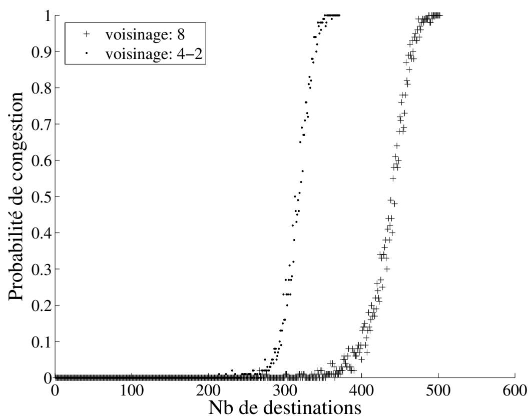


Figure 5.45 : *Comparaison entre un voisinage de 8 et de 4-2.*

5.11.6 Théorie de la percolation

La forme des courbes de congestion montre une caractéristique particulière. Lorsque le nombre de chemins à créer est faible, il est toujours possible de les trouver ; et lorsque ce nombre est très élevé, en proportion du nombre d'unités de routage présentes, aucune solution ne peut être trouvée. Entre ces deux phases, nous observons donc un passage de "tout est routable" à "rien n'est routable", qui n'est autre qu'une transition de phase. Les transitions de phase peuvent être observées dans nombre de phénomènes physiques, tel le passage de l'eau en glace.

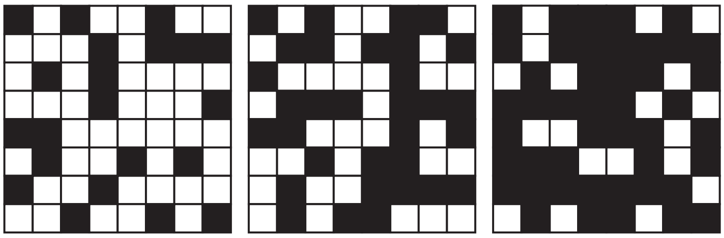
La théorie de la percolation, dont les premiers travaux sont dus à Broadbent et Hammesley [31], sert à analyser des phénomènes macroscopiques liés à des actions microscopiques. Son nom, introduit par Hammersley, vient du percolateur de la machine à café. Le percolateur est chargé de compresser le café moulu, au travers duquel l'eau chaude passe et se charge en caféine. Plus la pression est forte, plus l'eau est en contact avec un grand nombre de particules de café, et plus le café est fort. A partir d'un certain point, appelé point de percolation, l'eau ne peut plus circuler, l'espace entre les particules de café étant trop faible. Au delà de ce point critique, l'eau ne peut donc jamais passer, et en deça elle le peut toujours. De nombreux autres phénomènes peuvent être analysés selon ce point de vue de la percolation. A titre d'exemple, citons les feux de forêts. Les arbres peuvent être modélisés comme des points reliés à leurs proches voisins. Chaque ligne entre deux arbres possède une probabilité de propager le feux, si un des arbres est touché par les flammes. Il est alors possible de prévoir, en fonction de la probabilité de propagation, si toute la forêt risque de brûler ou non, si un incendie se déclare. Au delà d'un point critique, elle brûlera entièrement, et en deça, seule une partie de la forêt sera détruite.

Deux modèles théoriques ont notamment émergé de cette théorie, la percolation de sites, et la percolation de liens [9]. Dans le premier, nous disposons d'une grille d'éléments pouvant être noirs ou blancs avec une probabilité p. La grille, de taille $n \times n$ est remplie d'éléments, chacun d'eux ayant la probabilité p d'être noir (Figure 5.46). Nous nous intéressons à la propriété global consistant à pouvoir relier le haut de la grille au bas en ne passant que sur des cases noires. Si la probabilité p est proche de 0, jamais un cluster noir ne sera assez grand pour relier les deux bords de la grille, et si elle est proche de 1, il existera toujours un cluster assez grand pour le faire. Entre les deux probabilités se situe une transition de phase, passant par le point critique. La caractéristique intéressante d'un tel problème apparaît lorsque n tend vers l'infini. La transition de phase devient de plus en plus petite, et pour $n = \infty$, un point de probabilité critique p_c définit la frontière entre les réseaux ne pouvant jamais relier les deux bords, et ceux qui le peuvent toujours. Un tel problème, pour une grille de voisinage 4, possède un point critique $p_c = 0.6$.

La percolation de liens est relativement semblable. Des points d'une grille carrée sont reliés par des liens, qui peuvent être actifs avec une probabilité p, et inactifs avec une probabilité $1 - p$. Le modèle percole, comme dans le cas de la figure 5.47, si le haut de la grille peut être relié au bas par une suite de liens actifs. Ici, le point critique, lorsque la taille de la grille tend vers l'infini, est de $p_c = 0.5$.

Le modèle de percolation de liens peut être étendu à d'autres voisinages plus complexes, voir même à des graphes quelconques dans le cas des applications de réseaux de télécommunications. Le parallèle avec notre problème de congestion semble évident.

[9] Le lecteur intéressé trouvera une excellente introduction à la théorie de la percolation dans [182].



(a) $p(noir) = 0.3$ (b) $p(noir) = 0.5$ (c) $p(noir) = 0.7$

Figure 5.46 : *Modèle de percolation de sites, avec différentes probabilités de coloriage. Les deux réseaux de droite percolent.*

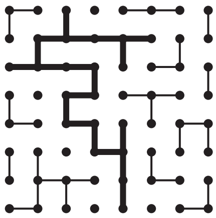


Figure 5.47 : *Modèle de percolation de liens, où nous pouvons observer un réseau percolant.*

Nous nous intéressons effectivement à savoir, si pour un nombre de liaisons source-destination donné, un chemin supplémentaire peut être créé ou non. Nous pourrions dire que le réseau percole lorsqu'une nouvelle connexion ne peut être établie.

Alors que pour le modèle de percolation de liens, une analyse mathématique rigoureuse permet de calculer exactement le point critique, la tâche semble plus ardue dans notre cas. En effet, le réseau entre unités de routage est en fait un réseau entre les multiplexeurs présents dans les switchboxs. Ce réseau directionnel, illustré à la figure 5.48, bien que régulier, montre une complexité nettement supérieure à celle d'une grille de points reliés à leurs quatre voisins. De plus, alors que dans le modèle standard, la probabilité d'activité d'un lien est indépendante des autres liens, dans notre problème de routage, les chemins relient les sources aux destinations, et donc l'occupation des liens entrant et sortant d'un multiplexeurs ne sont pas des variables indépendantes.

Nous suggérons donc une analyse plus poussée du problème de congestion de nos algorithmes, sous l'œil bienveillant de la théorie de la percolation, qui pourrait aisément conduire à une nouvelle thèse, tant les paramètres sont vastes : taille de la grille, type d'algorithme, nombre de destinations par source, longueur moyenne des chemins, ...

Nous avons, dans les pages précédentes, mené une analyse dans ce sens, en tentant de caractériser nos algorithmes en fonction d'une probabilité qu'une unité de routage soit une destination (p_{dest}), et en fonction du nombre de destinations reliées à la même source. Le nombre total de destinations, pour un tableau de taille $n \times m$ est défini par $ndest = p_{dest} \times n \times m$. Il serait alors intéressant de voir émerger une propriété de congestion indépendante de la taille de la grille, uniquement en fonction de la densité de destinations.

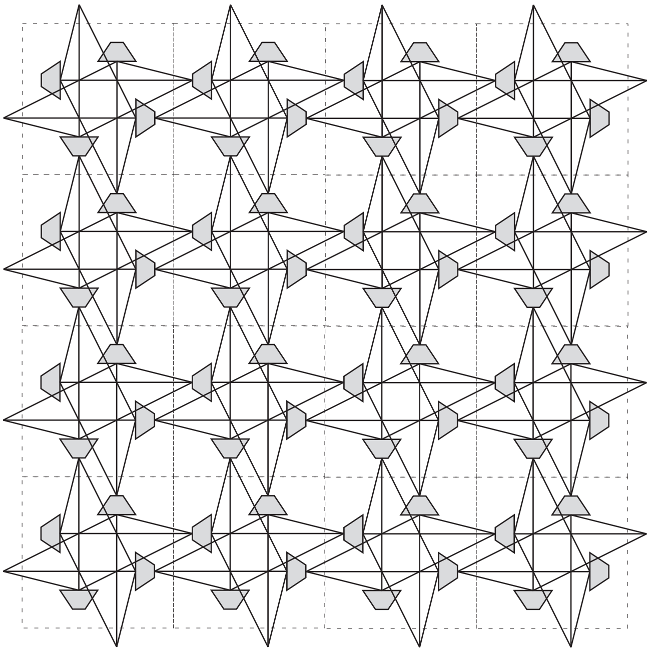


Figure 5.48 : *Réseau directionnel d'unités de routage (carrés traitillés) avec un voisinage de 4.*

Notons que le type d'expériences à mener, pour observer un phénomène de percolation mettant en jeu un point critique indépendant de la taille du réseau, devrait se faire en définissant une longueur moyenne de chemin. En effet, la pose aléatoire de sources et destinations dans le plan de taille $n \times n$ a pour effet de créer des chemins dont la longueur moyenne est proportionnelle à n, ce qui est le cas dans toutes nos expériences précédentes.

Dans le cas d'une longueur moyenne l fixe, avec $ndest$ destinations reliées à différentes sources, la probabilité qu'un lien entre deux multiplexeurs soit occupé est proportionnelle à $\frac{ndest \times l}{n^2} = \frac{p_{dest} \times n^2 \times l}{n^2} = p_{dest} \times l$. Dans le cas d'une pose aléatoire, il est en revanche proportionnel à $\frac{ndest \times n}{n^2} = \frac{p_{dest} \times n^3}{n^2} = p_{dest} \times n$, donc dépendant de la taille du tableau. Plus cette taille augmente, plus le risque de congestion est élevé.

Lors des comparaisons des probabilités de congestion entre différentes tailles de tableau, il faudrait donc prendre garde à bien traiter la probabilité qu'un lien soit occupé, de manière à pouvoir faire émerger une caractéristique indépendante de la taille du tableau d'unités de routage. Comme piste de recherche, nous pouvons nous intéresser à nos résultats précédents. La figure 5.43(b), en page 161, montre que l'apparition du problème de congestion apparaît à un nombre de destinations proportionnel à n. Ceci correspond exactement au fait que la longueur des chemins générés par nos expériences est justement proportionnelle à n. En divisant le nombre de destinations, correspondant à un niveau de congestion donné, par n, nous obtenons des droites de pente 0, qui, pour les faibles niveaux de congestion, ont tendance à baisser lorsque la taille du tableau augmente.

La figure 5.49 est identique à la figure 5.43(b), si ce n'est que le nombre de destinations a été divisé par la taille n du tableau. Pour les grandes valeurs de probabilité de congestion, il s'agit d'une droite de pente nulle, ce qui tend à prouver que nous sommes

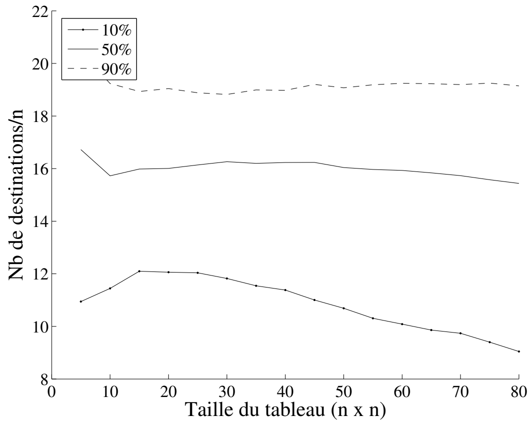


Figure 5.49 : *Nombre de destination correspondant à une probabilité de congestion donnée, divisé par la taille n du tableau, en fonction de la taille $n \times n$ du tableau (HIDRA, 3 destinations par source).*

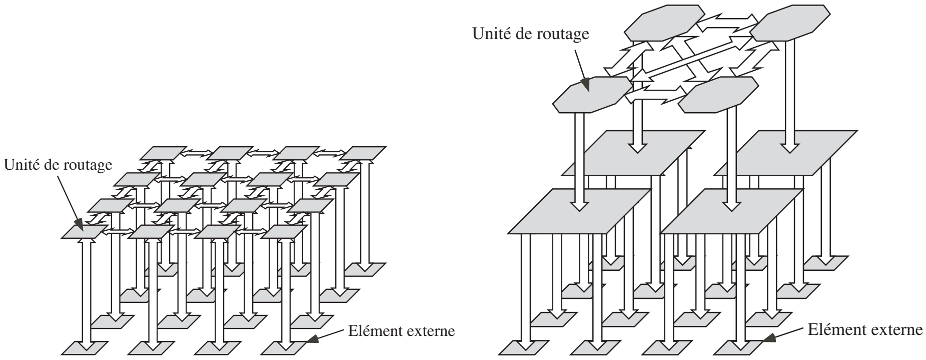
en présence d'un phénomène indépendant de la taille du tableau, pour une longueur de chemins moyenne fixe. Cependant, un nombre considérable d'expériences supplémentaires, et de recherches théoriques, seraient nécessaires pour confirmer ces dires, et pour caractériser le problème de congestion comme étant un modèle de percolation.

5.12 Conclusion

Nous venons de présenter cinq algorithmes de routage distribué, dont trois d'entre eux sont des variations sur le premier, HIDRA. Leurs caractéristiques ont été étudiées, et plus particulièrement le risque de congestion. En effet, notre mécanisme de routage est dynamique en comparaison du routage calculé par un logiciel spécialisé, qui est ensuite chargé de manière à configurer un circuit programmable. Il est toutefois statique en comparaison des méthodes de type *wormhole* [177] ou *packet switching*, où des paquets d'information transitent au travers d'un réseau de façon dynamique, leur acheminement pouvant prendre des chemins différents d'une fois à l'autre. Nos algorithmes se chargent de configurer des multiplexeurs, qui sont ensuite utilisés pour transmettre de l'information par des chemins fixes. Dès lors, il est possible, lorsqu'un nombre trop élevé de multiplexeurs sont configurés, que la création d'un nouveau chemin se trouve bloquée sans trouver de solution.

Le circuit POEtic, que nous décrivons dans le prochain chapitre, embarque un algorithme de routage, qui se trouve être HIDRA, et ce avec un voisinage de 4. Nous venons cependant de prouver qu'il aurait été préférable de baser notre réalisation sur un voisinage de 8. En effet, en connectant quatre éléments externes à une unité de routage de voisinage 8 au lieu d'un élément par une de voisinage 4, pour un nombre total de transistors deux fois moins élevé, cette option offre un nombre de connexions potentiellement routables deux fois plus élevé (Figure 5.50(b)). Cette analyse n'a malheureusement été effectuée qu'après le lancement de la fabrication du circuit, et nous ne pouvons que suggérer l'emploi de l'option à 8 voisines dans le cas où un deuxième circuit de ce type devait voir le jour.

Toujours sur la question du voisinage, il serait intéressant d'approfondir celui de



(a) Voisinage de 4

(b) Voisinage de 8, où quatre éléments sont reliés à une unité de routage.

Figure 5.50 : *Tableaux à voisinage de 4 et 8, dont celui de 8 est deux fois plus efficace en terme de congestion.*

6, en exploitant les propriétés des membranes d'eau savonneuse, décrites en page 69. Trois parois d'eau savonneuse se croisent toujours en des angles de 120 degrés, et une telle membrane reliant plusieurs points s'organise de manière à créer un arbre de Steiner minimal (ou qui correspond à un minimum local). La figure 5.51 illustre la manière dont plusieurs sources reliées à une destination pourraient l'être grâce à un chemin total minimum, en le modifiant itérativement. Le schéma de gauche correspond à la solution trouvée par HIDRA, et les deux autres sont des modifications du chemin menant au chemin de taille totale minimum, qui utilise deux multiplexeurs de moins que la solution initiale. Un tel mécanisme permettrait d'économiser le nombre de multiplexeurs réquisitionnés lorsque plusieurs destinations sont reliées à une même source, et donc de réduire substantiellement le risque de congestion.

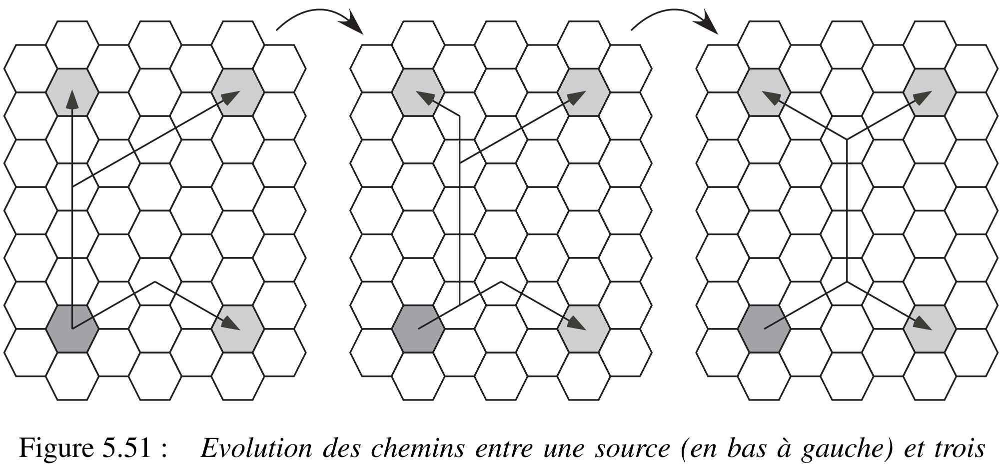


Figure 5.51 : *Evolution des chemins entre une source (en bas à gauche) et trois destinations, de manière à construire un arbre de Steiner de poids minimum.*

Concernant la scalabilité des quatre premiers algorithmes, HIDRA, HIDRA-RC, HIDRA-RT, et HIDRA-RTC, nous pouvons noter que les chemins combinatoires traversant les unités de routage obligent à baisser la fréquence d'horloge du système pour que son fonctionnement ne se trouve pas altéré. L'algorithme HIDRA-L a apporté une solution synchrone à ce problème, en transformant les unités de routage en machines de Moore. Cette implémentation possède toutefois le désavantage de nécessiter un nombre de transistors qui, pour un voisinage de 4, double par rapport à l'algorithme de base HIDRA.

Enfin, il serait possible de concevoir un algorithme semblable à HIDRA-L, basé sur des mécanismes asynchrones. Il serait alors possible de disposer d'un système totalement scalable, sans soucis de synchroniser une horloge. Bien que potentiellement élégante, cette solution se distinguerait toutefois par un nombre de transistors sans doute encore plus important que pour HIDRA-L.