HAL
open science

Exploitation optimale des circuits reconfigurables FPGA pour la mise en oeuvre d'un moteur de recherche de motifs

Yapi Georges Adouko, François Charot, Christophe Wolinski

To cite this version:

Yapi Georges Adouko, François Charot, Christophe Wolinski. Exploitation optimale des circuits reconfigurables FPGA pour la mise en oeuvre d'un moteur de recherche de motifs. 2008. hal-00202772v2

HAL Id: hal-00202772

https://hal.science/hal-00202772v2

Preprint submitted on 8 Jan 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Exploitation optimale des circuits reconfigurables FPGA pour la mise en œuvre d'un moteur de recherche de motifs *

Georges Adouko, François Charot, Christophe Wolinski

Irisa
Campus de Beaulieu
35042 Rennes Cedex, France
{adouko,charot,wolinski}@irisa.fr

Résumé
Nous proposons une implémentation matérielle d'un moteur de recherche de motifs inspirée de l'algorithme de Aho-Corasick, conçue pour exploiter de manière optimale les circuits reprogrammables de type FPGA et adaptée aux grandes bases de motifs (plusieurs dizaines de milliers de motifs) comme celles d'un système de détection d'intrusion tel que *Snort*. Nous présentons les idées qui ont guidé nos travaux et décrivons la méthodologie de mise en œuvre. Nous montrons comment, en utilisant la parallélisation de trafic et le *retiming* de circuits, des débits de près de 40 Gbits/s peuvent être atteints.

Mots-clés : recherche de motifs, FPGA, détection d'intrusions

---

1. Introduction

A l'heure où les réseaux locaux communiquent à des débits de plus de 1 Gbits/s par seconde, les systèmes de détection d'intrusion (IDS) logiciels tels *Snort* [11] ne peuvent traiter que des débits de quelques dizaines de Mbits/s. Un IDS saturé n'a pas d'autre solution que d'ignorer une proportion plus ou moins importante du trafic, ce qui peut entraîner la non détection d'un certain nombre d'attaques. Le déploiement progressif des liaisons à 10 Gbits/s, voire même à 40 Gbits/s, oblige à repenser les IDS actuels.

Le principal goulot d'étranglement des IDS actuels reste la recherche de signatures d'attaques ou de virus dans les flux. Des techniques logicielles performantes telles que celles dues à Aho-Corasick[2] ou Wu-Manber[17] ont été proposées et mises en œuvre dans *Snort*. Les performances insuffisantes de ces approches logicielles en regard des exigences pour des débits de plus en plus élevés poussent au développement de solutions matérielles d'analyse de trafic à base de composants spécialisés comme les processeurs de réseaux ou les circuits reprogrammables de type FPGA. Les travaux portant sur l'utilisation de FPGA pour la recherche de motifs sont nombreux et variés : filtre de Bloom [8], utilisation de mémoires associatives de type CAM (*Content Addressable Memory*) [14], automates finis non déterministes NFA [12, 7], etc. Une synthèse de ces techniques matérielles de recherche de motifs est proposée dans [1]. Avec ces techniques matérielles, des débits de quelques Gbits/s peuvent être atteints. Le traitement de trafic à plus de 10 Gbits/s pose encore aujourd'hui d'énormes contraintes, car il induit des coûts matériels prohibitifs. Les travaux les plus récents [15, 4, 5] contribuent en partie à proposer de nouvelles approches dont nous discutons les limites en section 2.

Nous proposons dans ce papier une technique de recherche de motifs inspirée d'Aho-Corasick, adaptée aux circuits reprogrammables de type FPGA et capable de traiter de grandes bases de motifs (plusieurs dizaines de milliers de motifs). Cette solution permet d'atteindre des débits entre 10 et 40 Gbits/s.

Pour répondre aux exigences d'un moteur de recherche de motifs supportant le très haut débit, nous proposons une architecture basée sur le fonctionnement parallèle de plusieurs mini automates, chaque automate traite une partie de l'ensemble des motifs. Ces automates sont capables de traiter plusieurs

* Ces travaux sont soutenus par la région Bretagne dans le cadre du projet PRIR FASTNET.

caractères par cycle. Ils sont peu coûteux en ressources logiques et surtout en ressources mémoire. L'accent est mis sur l'utilisation combinée et équilibrée des ressources logiques et mémoires des FPGA dans le but d'utiliser de façon optimale les ressources disponibles dans les composants actuels.

Le papier est structuré de la façon suivante. La section 2 présente la problématique de la recherche de motifs ainsi que l'état de l'art sur le sujet. La section 3 détaille la méthodologie de création d'automates multi-caractères qui sont la base de notre approche. La section 4 présente une implémentation FPGA d'automates à base de RAM. L'optimisation de cette architecture est détaillée en section 5. Enfin, en section 7, nous comparons cette solution avec les travaux existants après avoir préalablement expliqué en section 6 comment de très hautes performances peuvent être obtenues.

2. Mise en œuvre de la recherche de motifs sur FPGA : problématique et état de l'art

La recherche de motifs dans du trafic à très haut débit passe par l'utilisation de dispositifs dont les capacités de calcul relèvent de l'utilisation de circuits matériels spécialisés. Il est par ailleurs intéressant que ces dispositifs disposent de mécanismes de reconfiguration pour supporter les mises à jour des signatures d'attaques. Les circuits FPGA représentent à l'heure actuelle les meilleures cibles de par leur haut degré de parallélisme et leur possibilité de reconfiguration. Ces circuits sont constitués d'une structure régulière de petits blocs d'éléments logiques programmables, de bancs de mémoires distribués de taille réduite, et parfois d'opérateurs spécifiques ; l'ensemble est interconnecté par des bus configurables. Les architectures des FPGA d'Altera ont servi de support à nos études. Les mémoires de petite taille (quelques kilobits) fonctionnent à plus de 500 MHz. Elles peuvent être aussi interconnectées pour obtenir des mémoires de plus grande capacité, au dépend d'une baisse notable de la fréquence de fonctionnement. Il en est de même pour les blocs logiques programmables.

Dimensionnement des architectures implantées sur FPGA

Pour réaliser la recherche de motifs sur FPGA, l'approche classique consiste généralement à envisager une architecture qui ne s'adapte pas bien à la taille de l'ensemble des motifs à rechercher. Lorsque le nombre de motifs à traiter est très élevé (quelques milliers comme dans le cas de *Snort*), la complexité matérielle est très grande et de fait, la fréquence de fonctionnement s'écroule, et les performances aussi. Cette approche ne permet pas le passage à l'échelle. *L'approche retenue dans nos travaux consiste à partitionner la base de motifs en petits sous-ensembles, qui peuvent être implémentés en très grand nombre sans que les propriétés de l'implémentation globale soient remises en cause*.

Ainsi, Sourdis et al. [14] utilisent le partitionnement de l'ensemble des motifs pour augmenter les performances et réduire le coût en surface des CAM décodées implémentées sur FPGA (DCAM). Si le gain en surface reste limité, leurs performances en fréquence restent très bonnes, et ce quel que soit le nombre de motifs recherché. Dans leurs travaux sur l'implémentation de CAM sur FPGA, Laughlin et al. [10] exploitent un partitionnement judicieux de l'ensemble de motifs pour tirer partie de la présence de mémoires distribuées de type RAM dans le FPGA. Les CAM obtenues sont de grande taille (plusieurs Mbits) mais leur utilisation (remplissage du contenu des CAM) se révèle complexe. Le partitionnement permet d'adapter les implémentations à la structure du FPGA.

Augmentation du débit par le traitement multi-caractère

Notre objectif est la mise en œuvre d'architectures matérielles fonctionnant à des fréquences supérieures à 250 MHz pour atteindre plus facilement le très haut débit et limiter le besoin de parallélisation (réduction du volume de ressources matérielles). *Aussi, la mise en œuvre d'une technique traitant à la base plusieurs caractères par cycle permet d'augmenter les performances en débit à fréquence de fonctionnement fixe*.

Des travaux sur les automates multi-caractères ou non (Aho-Corasick par exemple) fournissent pour l'essentiel des méthodes de construction de graphes d'automates et des implémentations suivant le modèle des automates de Moore. La nouveauté réside dans l'implémentation de la fonction de transition (*FT*) non plus avec de la logique combinatoire, mais avec des tables de transitions stockées dans des mémoires. Il s'agit de tirer profit des hautes fréquences de fonctionnement des composants mémoires du FPGA. Alicherry et al. [4] utilisent des mémoires TCAM (ternary CAM) couplées à des mémoires RAM pour calculer les transitions de l'automate. Leurs implémentations d'automates sur FPGA sont du coup peu aisées car les FPGA actuels ne disposent pas de ressources en TCAM. [15, 3] utilisent exclusivement

des mémoires SRAM pour stocker leurs tables de transition. [3] privilégie l'implémentation d'un seul automate, il en résulte une taille de RAM de plus de 10 Mbits pour environ 1000 motifs. De tels volumes de mémoire ne sont pas disponibles sur FPGA. L'architecture éclatée proposée dans [15] s'adapte bien à la structure mémoire des FPGA, mais son coût reste encore élevé avec un minimum de 16 octets par caractère.
En résumé, les mises en œuvre d'automates sont soit trop coûteuses en ressources mémoire, soit inadaptées à la structure interne du FPGA. Le codage exploité par Van Lunteren[16], qui consiste à instaurer des priorités entre les transitions de l'automate, peut résoudre les problèmes de coût mémoire car il réduit considérablement la quantité de ressources en RAM nécessaire pour le codage des transitions d'automates.

Utilisation optimale des ressources mémoires et logiques du FPGA

La plupart des implantations de techniques de recherche de motifs s'appuie presque exclusivement sur l'utilisation d'un même type de ressources du FPGA. Soit ce sont des ressources logiques (LE) qui sont intensément utilisées au détriment des ressources mémoires, soit c'est l'inverse. On aboutit ainsi à une sous-utilisation globale des ressources du FPGA.
Notre approche met d'une part l'accent sur l'utilisation combinée et équilibrée des ressources logiques et mémoires des FPGA dans le but d'utiliser de façon optimale les ressources disponibles.

3. Principe de fonctionnement de la technique Aho-Corasick multi-caractère répartie sur FPGA

Une version de l'algorithme d'Aho-Corasick permettant de créer des automates gérant plusieurs caractères (nombre de caractères par cycle fixe) a été développée. Cette approche est combinée à des notions de partitionnement de la base de motifs, elle tient compte de l'incidence de ce partitionnement sur l'implémentation.

3.1. Fonctionnement de l'automate Aho-Corasick multi-caractère

Aho-Corasick permet de traiter efficacement sous forme d'automates des motifs fixes (c'est le cas de nos motifs). Ces automates sont simples à construire et l'algorithme garantit des performances constantes (pas de pire cas). Le procédé de construction de l'automate multi-caractère est similaire à celui d'Aho-Corasick de base, sauf que les transitions comportent plusieurs caractères. On crée le graphe des motifs réduit suivant les préfixes communs (transitions *goto* d'Aho-Corasick), puis on y rajoute les transitions à suivre dans le cas où la progression dans le graphe échoue (transitions *fail* d'Aho-Corasick). La figure 1-a illustre la construction du graphe traitant deux caractères pour les motifs *abcdab* et *abef*.

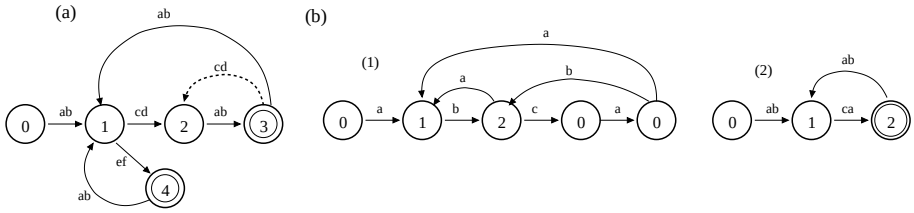


FIG. 1 – Graphe Aho-Corasick : (a) traitant deux caractères par cycle pour $abcdab$ et $abef$, (b) traitant un et deux caractères par cycle pour le motif $abca$

Chaque transition du graphe comporte deux caractères. Les deux motifs ont un préfixe commun *ab*, ils partagent donc en commun l'état 1. Les états 3 et 4 sont des états finaux ou états d'acceptation du graphe. Les états 2 et 3 sont reliés par une transition de type *fail* (arc en pointillé).

N	5x10 = 50	10x10=100	20x20=400
$P = 1$	22/47	26/91	39/327
$P = 2$	23/26	42/50	124/170

TAB. 1 – Nombre de symboles distincts/nombre d'états du graphe pour différentes tailles d'ensemble de motifs

Propriétés de l'automate multi-caractère et mode d'utilisation

La figure 1-b montre les différences de construction entre les graphes traitant un et deux caractères par transition pour le même motif *abca*. (2) a presque deux fois moins d'états et de transitions que (1). Mais il faut deux instances de (2) pour détecter correctement le motif *abca*. En effet, le graphe (2) détecte bien le motif *abca* dans le flux *bx***abca***er* (séquencé *bx ab ca er*) mais pas dans le flux *x***abca***x* (séquencé *xa bc ax*).
Pour un ensemble de motifs de taille N ($N < 500$) où N est la somme des longueurs des motifs et P le nombre de caractères traités par cycle ($P > 1$), le nombre d'états E et de symboles différents S du graphe peut être approximé par $S = E = N/P$. Les tests réalisés sur des échantillons de motifs et résumés dans le tableau 1 étayent cette approximation, surtout dans le cas de deux caractères par cycle. Cette approximation est fondamentale pour le dimensionnement d'architecture.
Pour la construction d'un automate multi-caractère à P caractères par cycle, il faut que les motifs soient de taille multiple de P. Pour les motifs non multiples de P, il faut mettre en œuvre une solution qui n'augmente pas la complexité des traitements comme c'est le cas avec les automates proposés par Alicherry et al. dans [4].

Détection d'un motif de longueur quelconque

Pour traiter des motifs de longueur quelconque, une troncature des motifs doit être réalisée, soit par leur début (préfixe), soit par leur fin (suffixe). Une fois les motifs tronqués, les données du graphe (transitions et symboles de transition) sont plus simples. Pour P caractères par cycle, la détection complète d'un motif se décompose alors en deux étapes :

1. détection du motif tronqué de taille multiple de P dans le trafic ;
2. comparaison ponctuelle du trafic avec les caractères restants du motif.

La troncature de plusieurs motifs distincts peut aboutir à un même motif tronqué commun. La détection de ce motif tronqué commun se terminera alors par plusieurs opérations de comparaison ponctuelle des restes de motifs au trafic. Le choix du suffixe ou du préfixe pour la troncature des motifs est fait de sorte à minimiser le nombre de motifs tronqués communs.

3.2. Implémentation sur FPGA

L'automate multi-caractère a été implémenté sur FPGA en s'inspirant du modèle de Moore (une fonction de transition *FT* suivie d'un registre d'état courant). Notre méthodologie d'implémentation d'automates est globale. Elle est modélisée par la figure 2.

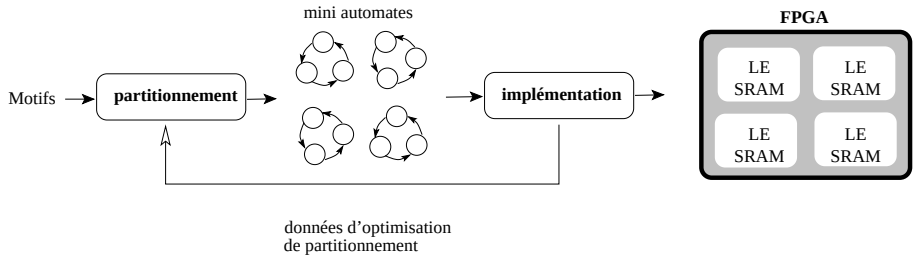


FIG. 2 – Processus de mise en œuvre sur FPGA

A partir d'un grand ensemble de motifs, il s'agit de créer de petits groupes de motifs dont les automates sont implémentés sur FPGA. Le trafic à analyser est diffusé en parallèle à tous les automates grâce à un arbre de distribution de trafic. Pour la décomposition de l'ensemble des motifs en petits groupes, il faut déterminer le comportement de la mise en œuvre matérielle sur FPGA des automates en fonction des caractéristiques des motifs à rechercher. Les propriétés de la mise en œuvre matérielle donnent les contraintes nécessaires pour effectuer un partitionnement de l'ensemble des motifs qui optimise les performances et maximise l'occupation des ressources du FPGA. Nous proposons dans les sections suivantes 4 et 5 deux modèles d'implémentation dont les propriétés sont utilisées pour trouver le partitionnement adéquat de la base de motifs.

4. Implémentations d'automates haute performance à base de RAM et à coût mémoire réduit

Bases de l'implémentation

Le modèle d'implémentation des automates est illustré à la figure 3. Il s'agit de fournir, pour chaque état courant de l'automate (valeur du registre d'état) et pour chaque symbole (bloc de P caractères) lu à l'entrée de l'automate, la valeur de l'état futur. Pour réduire le coût mémoire, l'automate fournit plutôt pour chaque état courant et chaque symbole *distinct* du graphe, la valeur de l'état futur.

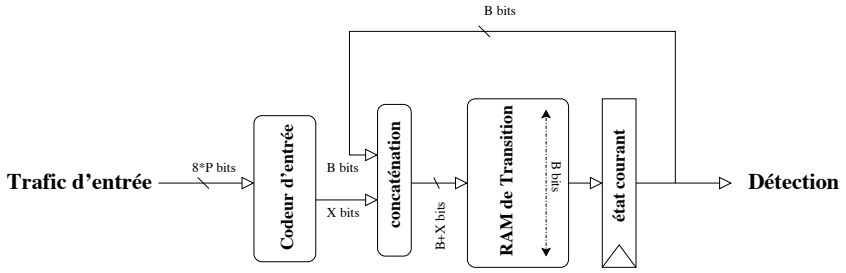


FIG. 3 – Modèle d'implémentation de base de l'automate

A chaque cycle de traitement, P caractères sont lus en entrée et identifiés par le *codeur*. Cet identifiant (X bits) est concaténé avec la valeur de l'état courant (B bits) pour indexer la RAM de transition (une SRAM). La valeur de l'état futur lue en RAM est alors enregistrée dans le registre d'état au prochain coup d'horloge. Les propriétés de ce circuit dépendent des caractéristiques du graphe qu'il implémente, et par conséquent des longueurs de motifs et du nombre de caractères traité par cycle.

Propriétés et dimensionnement des composants

Comme indiqué au paragraphe 3.1, les nombres d'états S et de symboles distincts E des automates peuvent être approximés par $S = E = N/P$ où N est le nombre de caractères des motifs et P le nombre de caractères traité par cycle. Le coût en mémoire C_M d'un automate unitaire peut alors être exprimé par l'équation suivante :

$$C_M \approx 2^{\lceil \log_2(N/P) \rceil + 1} . 2^{\lceil \log_2(N/P) \rceil + 1} . (\lceil \log_2(N/P) \rceil + 1)$$

où $\lceil x \rceil$ désigne la valeur entière de x. C'est le produit du nombre d'emplacements mémoires alloués pour indexer la valeur de l'état futur par la largeur en bits de la valeur de l'état futur. La progression mémoire est théoriquement en $O(N^2.\log(N))$. En pratique, les mémoires physiques disponibles imposent des contraintes car elles ont un dimension figée et restrictive.

Nos travaux montrent que pour le traitement de P caractères par cycle, il existe deux tailles de groupes N pour lesquelles le coût en mémoire de type M4k (bloc mémoire du FPGA d'Altera) est minimal. Cela maximise l'utilisation du FPGA relativement aux mémoires M4k. Pour $P = 4$, il faut choisir N=64 ou N=128 et pour $P = 8$, N vaut 128 ou 256.

Évaluation des coûts et des performances sur FPGA

Le composant FPGA Altera Stratix EP2S130F1020C3 (plus de 100 000 éléments logiques et quelques 700 blocs mémoires distribués M4k de 4096 bits) a été utilisé pour réaliser tous les tests et garantir ainsi l'équivalence et la comparabilité entre les résultats obtenus.
En pratique, la fréquence de fonctionnement maximale de notre modèle est de 335 MHz, limitation due à la boucle de retour d'une mémoire M4k sur elle-même. Les automates multi-caractères de P=2, 4, 8 caractères par cycle traitent respectivement des débits (Deb) maximum de 5, 10 et 20 Gbits/s de trafic ($Deb = F.8.nbcar$, où F est la fréquence d'horloge et $nbcar$ le nombre de caractères par cycle).
Le coût en cellules logiques C_{LE} a été expérimentalement établi. Il est donné par : $C_{LE} = \alpha.N.P$ avec $\alpha \approx 1$. C'est le coût de P instances de *codeurs* de N caractères.
Le traitement de 4 caractères par cycle (correspondant à un débit de 10 Gbits/s) avec des groupes de 64 caractères maximise le nombre total de caractères des motifs à rechercher par le FPGA. Le FPGA cible considéré a une capacité de 21 000 caractères. L'occupation maximale est de 92 000 LE, soit 92% du FPGA. Le coût en surface est de de 4,38 LE/caractère pour un coût mémoire de 16,6 octets/caractère.

Mesures de performance

Deux types de mesures de performance sont définies. La mesure de performance en ressources logiques symbolise le niveau de débit par unité d'éléments logiques. La mesure en mémoire représente le débit par unité de mémoire. Soient $Perf_{LE}$ et $Perf_M$ les performances en ressources logiques et en ressources mémoire :

$$Perf_{LE} = \frac{Deb}{C_{LE}} \quad Perf_M = \frac{Deb}{C_M}$$

où C_{LE} est le coût en éléments logiques par unité de caractère (LE/car) et C_M est le coût en mémoire par unité de caractère (octets/car).
Le modèle d'architecture détaillé dans cette section a la performance suivante : $Perf_{LE} = 10000/4,38 = 2200$, $Perf_M = 10000/16 = 625$.
Avec cette implémentation d'automates multi-caractères, seule une faible partie des ressources mémoire contient des informations réellement utiles. Le modèle peut donc être transformé dans le but de donner un rôle plus prépondérant à la mémoire. Cette transformation fait l'objet de la section suivante.

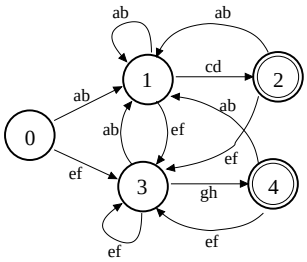
5. Optimisation de l'usage de la mémoire RAM

5.1. Motivations

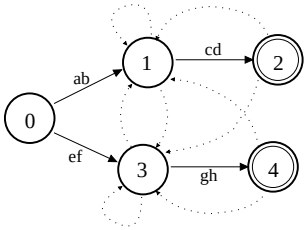
Dans le modèle proposé au paragraphe 4, appelé par la suite modèle de base, la mémoire RAM stocke la valeur de l'état futur de l'automate pour toute valeur d'état courant et pour tout identifiant du symbole d'entrée. Il s'agit d'un *mapping* direct et simple, mais inefficace du point de vue de l'occupation mémoire. En fait, l'empreinte mémoire réelle du graphe, c'est-à-dire les informations réellement nécessaires pour la caractérisation de l'automate dans la RAM, est très faible proportionnellement à l'occupation mémoire. Notre objectif est donc de mettre en œuvre des techniques de réduction de l'occupation mémoire tout en assurant le calcul d'une transition de l'automate par cycle d'horloge.

Prise en compte de la topologie du graphe

Le graphe multi-caractère généré à partir des motifs présente une structure particulière. Sur le graphe complet traitant deux caractères par cycle pour les motifs *abcd* et *efgh* de la figure 4-a, on remarque que les transitions sur les symboles *ab* et *ef* sont très nombreuses, et dirigées vers les états 1 et 3. En supposant que les symboles *ab* et *ef* conduisent par défaut vers les états 1 et 3 respectivement, il ne reste que deux transitions à coder dans le graphe. Le graphe peut donc être compressé et simplifié (figure 4-b). [16] exploite cette compression pour une utilisation logicielle à empreinte mémoire réduite tandis que [4] réduit la taille des TCAM. Cette méthode est employée dans notre approche pour diminuer la consommation mémoire de nos automates.



(a). Graphe complet non compressé



(b). Graphe simplifié et compressé

FIG. 4 – Processus de compression du graphe

Meilleure répartition des coûts entre ressources logiques et ressources mémoire

Avec un graphe d'automate compressé, l'occupation mémoire est plus faible. On peut ainsi stocker des informations supplémentaires de l'automate dans l'espace mémoire libéré. L'idée est de stocker dans la mémoire RAM, non seulement les relations entre les états de l'automate, mais aussi les valeurs des symboles de transition qui sont les conditions de transition. Ce transfert d'informations vers les RAM vise à soulager le composant *codeur* du modèle de base. Cela vise à diminuer le coût en ressources logiques de celui-ci.
Cette approche réduit la pression sur les ressources logiques (représentées par le *codeur*) pour en ajouter sur les ressources mémoire dans le but d'avoir une utilisation plus équilibrée de l'ensemble des ressources.

5.2. Définition d'un modèle optimisé

Soit G le graphe d'un ensemble de motifs traitant quatre caractères par cycle (figure 5).

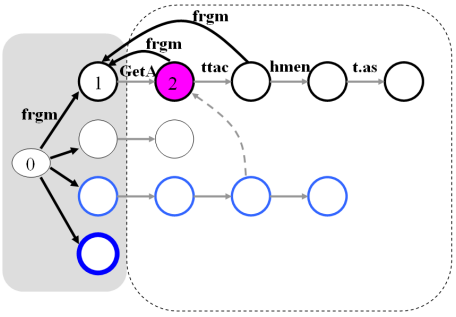


FIG. 5 – Différentes catégories de nœuds et de transitions dans un graphe

On définit le niveau d'un état comme étant la profondeur de l'état dans le graphe, l'état initial étant de niveau 0. Les états premiers (nœuds dans la zone grisée à gauche) sont les états de niveau 1 et l'état initial. D'après la topologie du graphe (figure 5), ce sont les états les plus pointés par les transitions. Ces états constituent les états **par défaut** pour notre nouveau codage d'automates. Les états secondaires (nœuds dans la zone encadrée en pointillé de la figure) sont les états de l'automate autres que les états premiers.
On appelle transition primaire toute transition du graphe qui aboutit aux états premiers (transitions en trait plus épais sur la figure 5). Ce sont les transitions par défaut de notre nouveau codage. Les transitions secondaires (en gris et trait plus fin sur la figure) dénotent toutes les transitions vers les états secon-

daires. Les transitions secondaires *de retour* (en pointillé gris) désignent les transitions secondaires allant d'un état secondaire vers un autre état secondaire de niveau inférieur (niveau 3 vers 2 par exemple). Ces transitions secondaires *de retour* sont peu nombreuses par rapport au nombre total de transitions secondaires ($\approx 10\%$ des transitions secondaires pour un graphe à 4 caractères par transition).

Contrainte CT : ***Pour une grande base de motifs, nous postulons qu'il est possible de créer des groupes de motifs dont les états des automates associés admettent au plus une seule et unique transition secondaire de sortie***. Nous proposons pour l'implémentation de ces automates, l'architecture décrite à la figure 6.

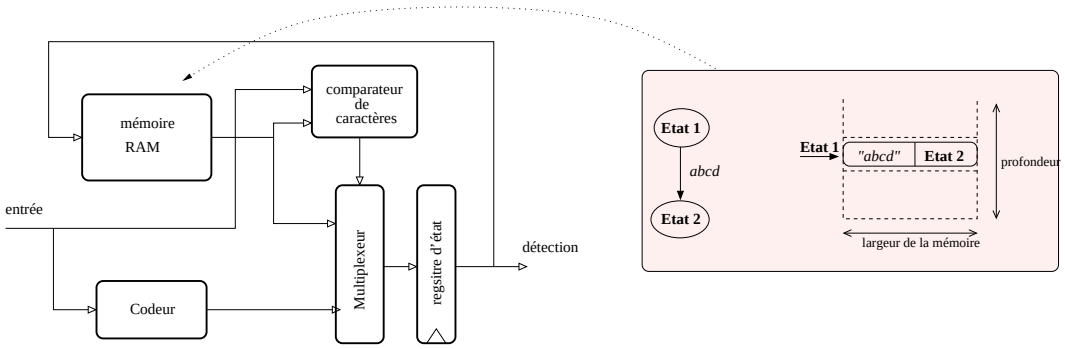


FIG. 6 – Nouveau modèle d'implémentation et contenu de la mémoire

1. Le *codeur* implémente un automate primaire (zone grisée du graphe de la figure 5) qui regroupe les états premiers et les transitions primaires. Ce *codeur* fournit l'état par défaut de l'automate global. L'automate primaire se présente sous la forme d'un graphe simple d'un nœud initial vers n nœuds suivant n symboles. Sa structure s'apparente clairement à un codeur de n symboles plutôt qu'à un automate réel.
2. La *mémoire* RAM stocke les informations correspondant aux transitions secondaires. Les transitions stockées dans l'automate sont représentées comme le montre la figure 6 où l'état courant (état 1) de l'automate adresse la mémoire. Le contenu de la mémoire est composé de la valeur du symbole de transition (*abcd*) et de la valeur de l'état futur (état 2).
3. *Le comparateur* compare la valeur du symbole d'entrée (trafic) avec la valeur du symbole acceptable pour la transition (stocké en mémoire).
4. *Le multiplexeur* choisit comme état réel futur de l'automate la valeur fournie par la mémoire si la transition fournie par la mémoire (automate secondaire) est validée. Dans ce cas, le symbole d'entrée est identique au symbole de transition stocké en mémoire. Sinon c'est l'état par défaut fournit par le *codeur* qui est choisi.

5.3. Propriétés

La figure 7 illustre les gains en ressources logiques réalisés avec ce modèle optimisé en mémoire par rapport au modèle de base de la section 4. Pour un même automate, on observe à travers les codes VHDL des différents *codeurs* des modèles d'implémentation que notre modèle optimisé en mémoire occupe beaucoup moins de ressources logiques car son *codeur* n'exploite que les transitions primaires (cadre gris figure 7). La proportion de diminution des coûts à nombre de caractères P par cycle fixe dépend du ratio T/P où T est la longueur moyenne d'un motif. Avec cette nouvelle implémentation, seules les transitions primaires sont codées. Ainsi, les variations du nombre de motifs M (égal au nombre de transitions primaires), de la taille moyenne des motifs T et du nombre de caractères traités par cycle P influencent directement les formes des graphes d'automate, et donc les coûts en cellules logiques.

Pour M motifs dont le nombre total de caractères vaut N, et P caractères traités par cycle, il y a $N/P + 1$ états dans l'automate, $M + 1$ états premiers (conséquence de la contrainte **CT**) et $N/P - M$ états secondaires. Il y a exactement M symboles de P caractères implémentés avec le *codeur* et il faut P instances

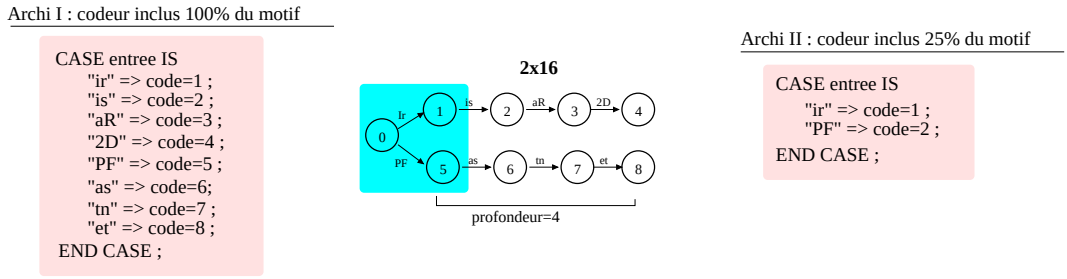


FIG. 7 – comparaison des *codeurs* de nos deux modèles et transfert des coûts

de *codeur* (pour un automate complet). En référence à la section 4, le coût en cellules logiques peut être estimé à : $C_{LE} \approx \alpha.M.P^2$.
Le coût en ressources logiques est linéaire en nombre de motifs par automate et quadratique en nombre de caractères traités par cycle. Il est indépendant de la taille des motifs ce qui signifie que l'architecture optimisée convient très bien aux motifs de grande taille.
Il y a en moyenne N/P transitions secondaires, donc autant d'emplacements de mémoire de taille $8.P + \lceil \log_2(N/P) \rceil$ bits. La taille mémoire totale est estimée à $C_M = N/P.(8.P + \lceil \log_2(N/P) \rceil)$.
En supposant que $\lceil \log_2(N/P) \rceil$ varie lentement, le coût mémoire augmente presque en $O(N)$. Ce coût est nettement meilleur que dans le cas du modèle de base où la variation est en $O(N^2.\log N)$.

5.4. Évaluation des coûts et performances

A P fixé, les caractéristiques de cette architecture dépendent de deux paramètres : le nombre de motifs M et le nombre total de caractères N. Le nombre de motifs fixe le coût du *codeur* alors que le nombre de caractères fixe le coût de la mémoire RAM. En confrontant les résultats des expérimentations avec l'architecture et les différents types de ressources du FPGA, nous avons déduit une configuration utilisant toutes les ressources de ce dernier de manière quasi optimale. L'implémentation détaillée par la figure 8 est dimensionnée pour un ensemble de motifs de 512 caractères. Le goulot d'étranglement principal (ou chemin long) est au niveau du rebouclage de la mémoire sur elle-même à travers le *comparateur* et le *multiplexeur*. En pratique, la fréquence maximale de fonctionnement de cette architecture est de 209 MHz. Ce qui conduit à un débit de 6,7 Gbits/s pour quatre caractères traités par cycle. Le coût en ressources logiques a été estimé à partir de l'implémentation de l'automate complet d'un ensemble de 32 motifs de 16 caractères. Le coût en surface s'élève à 700 éléments logiques.
L'automate de recherche d'un ensemble de motifs de 512 caractères au rythme de 4 caractères par cycle est constitué de 4 instances d'automates unitaires d'un coût mémoire total de 4 blocs mémoire M4k, 4 blocs M512, et de 700 LE. De fait, 150 automates de ce type peuvent être implémentés sur le FPGA considéré (Altera EP2S130F1020C3). Par extrapolation linéaire, ce FPGA a une capacité de 76 000 caractères pour une occupation maximale du FPGA estimée à 100% (théoriquement, on dispose jusqu'à 105 000 LE). Pour des motifs de taille moyenne de 16 caractères donc, le coût en cellules logiques est de 1,38 LE/car. Ce coût varie inversement avec la taille moyenne des motifs (section 5.3).

Application réelle aux motifs de *Snort*

Les 1073 motifs extraits des règles *http* de *Snort* ont été implémentés expérimentalement sur un moteur de recherche de motifs traitant 4 caractères par cycle. Nous avons développé en langage JAVA un outil qui compile les motifs fournis suivant la contrainte CT avec une limite de la taille maximale des automates (512 caractères). L'outil génère automatiquement le code VHDL (pour simulation ou synthèse sur FPGA) des *codeurs* et contenus *mémoire* des différents automates. Les 14 700 caractères des motifs ont été implémentés avec un peu moins de 35 automates, correspondant à 140 blocs mémoire M4k et à une taille mémoire cumulée de 540 Kbits (70 Ko). Le coût en cellules logiques est de 23 000 LE. Le coût total en mémoire est donc de 4 octets/car et le coût en cellules logiques s'établit à 1,56 LE/car. Ce coût s'explique car la taille moyenne des motifs de *Snort* est de 13 (< 16).

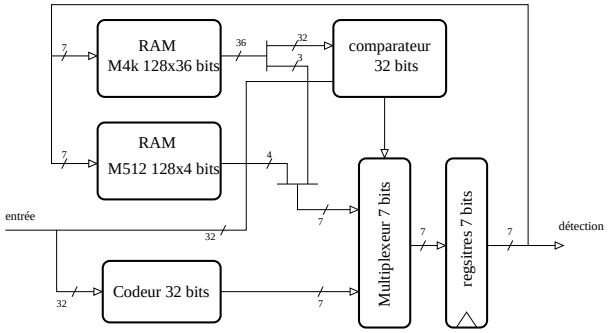


FIG. 8 – Modèle d'implémentation qui exploite au mieux des ressources logiques et mémoires du FPGA

Performances

Le modèle optimisé (figure 8) a un coût en éléments logiques de 1,38 LE/car et un coût en mémoire de 4 octets/car pour 4 caractères par cycle et des motifs de 16 caractères. Il en résulte les mesures de performance suivantes :
$\mathtt{Perf}_{\mathtt{LE}} = 6700/1,38 = 4885$ et $\mathtt{Perf}_{\mathtt{M}} = 6700/4 = 1675$
Pour $P < 4$ caractères par cycle, le coût en cellules logiques diminue de manière quadratique alors que la baisse de débit est linéaire. En conséquence, la performance en ressources logiques est plus grande. Ainsi pour $P = 3$ (respectivement 2), on a $\mathtt{Perf}_{\mathtt{LE}} = 7000$ (respectivement 9000). $\mathtt{Perf}_{\mathtt{M}}$ reste constant.

6. Augmentation des performances

Utilisation du *restreaming*

L'adéquation des architectures proposées avec celles des FPGA nous donne la possibilité de répliquer facilement une implémentation et de profiter du parallélisme offert par les FPGA. Si notre premier modèle d'architecture peut traiter à 10 Gbits/s un ensemble de motifs de 21 000 caractères, il est tout aussi capable de traiter à 20 Gbits/s une base de motifs de 10 000 caractères. Pour ce faire, il suffit d'implanter deux moteurs de recherche de motifs, identiques et fonctionnant en parallèle et de créer des paquets de données parallèles à partir du flux séquentiel de paquets d'entrée. Un répartiteur parallèle de trafic réalise le *restreaming* de paquets réseau.
Cette technique permet au modèle d'implémentation optimisé en mémoire de traiter sur le FPGA Altera 76 000 caractères à 6,7 Gbits/s, 38 000 caractères à 13,4 Gbits/s, 25 000 caractères à 20 Gbits/s voire même 12600 caractères à 40 Gbits/s. Les 1073 motifs de *Snort* peuvent être traités à plus de 33,5 Gbits/s. On transforme la haute performance (débit Gbit/s et très bas coût) en haut débit (débit en dizaines de Gbit/s et coût raisonnable). La performance reste constante et le débit augmente.

Utilisation du *retiming*

Le *retiming* introduit par [9] consiste à redéployer les registres d'un circuit synchrone à travers les parties combinatoires pour diminuer les chemins longs. Cette procédure permet d'augmenter la fréquence de fonctionnement d'un circuit. Le *retiming* peut être combiné avec le pipeline. Plus on pipeline, plus les chemins longs du circuit initial sont subdivisés par les registres redéployés par *retiming*, et plus la fréquence de fonctionnement augmente.
Dans le modèle d'implémentation optimisé en mémoire, le chemin critique se trouve au niveau de la boucle de la mémoire sur elle-même à travers le *comparateur* et le *multiplexeur*. Après l'intégration de x étages de pipeline (plus le *retiming*), la fréquence F du circuit augmente d'un facteur $\alpha > 1$ (αF au final). Il faut $(x + 1)$ cycles pour exécuter une transition de l'automate avec x cycles à vide. On peut de ce fait utiliser un seul automate pour en exécuter x en réutilisant les cycles à vide car tous les automates unitaires d'un même groupe de motifs sont identiques mais traitent le trafic sur des offsets consécutifs. Les

coûts en logique C_{LE} et en mémoire C_M sont alors divisés par x. En revanche, le nombre de caractères traité par cycle diminue et passe de P à $P/(x+1)$. La performance en éléments logiques $Perf_{LE}$ passe alors de $8.F.P/C_{LE}$ à $8\alpha.F.P/C_{LE}$. La performance augmente d'un facteur α.
Appliqué au modèle de base, la fréquence de fonctionnement passe de 335 MHz à plus de 500 MHz (limitation en fréquence du FPGA) avec un seul niveau de *retiming*. Ses performances sont ainsi multipliées par 1,5. Le débit de traitement de trafic passe de 10 Gbits/s à 8 Gbits/s alors que les coûts en logique et en mémoire sont divisés par 2. Quant au modèle optimisé, sa fréquence passe de 206 MHz à 423 MHz pour 3 niveaux de *retiming*. Le débit passe de 6,7 Gbits/s à 3,38 Gbits/s alors que les coûts sont divisés par 4. Les performances passent de 4 885 à 10 000. Avec ce niveau de performance, le moteur de recherche de 1073 motifs extraits de *Snort* pourrait fonctionner à plus de 67 Gbits/s si le *restreaming* est employé.

7. Conclusions

Les performances obtenues sont comparées avec celles des meilleures techniques de recherche de motifs répertoriées. Cette comparaison est résumée dans les tableaux 2 et 3. *Archi I* et *Archi II* représentent les modèles décrits dans ce papier (modèle de base et modèle optimisé). *3CC* et *4CC* désignent respectivement les traitements de 3 et 4 caractères par cycle.

Implémentation	Composant	Gbits/s	Nbre de car.	LE/car	Perf.
Adouko(archi. II,3CC+retiming)	stratix II E2PS130	3,38	+300 000	0,25	**13 200**
Adouko(archi. II,4CC+retiming)	stratix II E2PS130	3,38	+300 000	0,34	**10 000**
Adouko(archi. II,4CC)	stratix II E2PS130	6,7	76 000	1,38	**4 885**
Adouko(archi. I,4CC+retiming)	stratix II E2PS130	8	42 000	2,19	**3 652**
Adouko(archi. I, 4CC)	stratix II E2PS130	10	21 000	4,38	**2 283**
Baker-Prasana [5]	virtex2VP100-7	10	19 584	2,0	5150
Singaraju[13]	virtex2VP30-7	2,36	16 384	0,7	3424
Cho-Mangione-Smith[6]	spartan3-200	3,2	6 805	0,9	3556
Sourdis (D.NFA)[14]	virtex2-6000	9,7	18 000	3,6	2694

TAB. 2 – Comparaison de nos performances en ressources logiques à celles d'autres techniques

Nous obtenons des performances en ressources au moins deux fois supérieures à celles des solutions citées. Au niveau logique, bien que nous utilisons de la mémoire dans notre architecture, l'implémentation de Baker-Prasanna [5] est aussi performante que la nôtre. Cela est dû à nos *codeurs* implémentés sans aucune optimisation. Le coût élevé de ces derniers est amorti par l'utilisation de ressources mémoire. Des gains non négligeables peuvent être obtenus en implémentant les *codeurs* de façon plus intelligente.

Implémentation	Coût mémoire (octets/car)	Débit (Gbits/s)	Perf.
Adouko(archi. II+retiming)	1	3,38	3380
Adouko(archi. II)	4	6,7	1675
Adouko(archi. I+retiming)	8,3	8	963
Adouko(archi. I)	16,6	10	625
Alicherry (CAM+RAM)[4]	+3	2,0	666
Tan Sherwood (BitSplit)[15]	31,25	4,0	128
Aldwiri (RAM)[3]	126	2,0	15,87

TAB. 3 – Comparatif de nos performances en ressources mémoire à celles d'autres techniques

Par rapport aux techniques de recherche de motifs existantes, nous avons développé une nouvelle méthodologie plus adaptée aux architectures des FPGA et capable de produire des moteurs de recherche

de motifs à très bas coût en ressources et à performance très élevée. La flexibilité de cette méthodologie permet de créer des moteurs de recherche de motifs convenant à des débits de l'ordre du Gbits/s pour de très grandes bases de motifs (plus de 20000 motifs) ou à des contextes de très haut débit (plus de 40 Gbits/s) pour quelques milliers de motifs. Avec un seul FPGA, plus d'un millier de motifs peuvent être implémentés et recherchés à des débits de plus de 40 Gbits/s. Ces travaux ouvrent des perspectives pour de très hautes performances. Le modèle d'implémentation optimisé en mémoire doit pouvoir évoluer pour avoir moins de contraintes (ou des contraintes plus souples) dans la répartition des motifs sur les automates à implémenter. Nous nous proposons par ailleurs de travailler à l'insertion de nos moteurs de recherche de motifs au sein d'un système intégré complet d'IDS.

Bibliographie

1. G. ADOUKO, F. CHAROT, S. GOMBAULT, T. RAMARD et C. WOLINSKI : Panorama des algorithmes efficaces et architectures matérielles pour le filtrage réseau haut débit et la détection d'intrusions. *In MAJECSTIC 2006, Quatrième édition de la MAnifestation des JEunes Chercheurs STIC*, nov. 2006.
2. A. V. AHO et M. J. CORASICK : Efficient String Matching : an Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975.
3. M. ALDWAIRI, T. CONTE et P. FRANZON : Configurable String Matching Hardware for Speeding up Intrusion Detection. *SIGARCH Comput. Archit. News*, 33(1):99–107, 2005.
4. M. ALICHERRY, M. MUTHUPRASANNA et V. KUMAR : High Speed Pattern Matching for Network IDS/IPS. *In Network Protocols, 2006. ICNP '06. Proceedings of the 2006 14th IEEE International Conference on*, p. 187–196, nov. 2006.
5. Z. K. BAKER et V. K. PRASANNA : High-Throughput Linked-Pattern Matching for Intrusion Detection Systems. *In ANCS '05 : Proceedings of the 2005 symposium on Architecture for networking and communications systems*, p. 193–202, New York, NY, USA, 2005. ACM Press.
6. Y. H. CHO et W. H. MANGIONE-SMITH : A Pattern Matching Coprocessor for Network Security. *In DAC'05 : Proceedings of the 42nd annual conference on Design automation*, p. 234–239, New York, NY, USA, 2005. ACM Press.
7. C. R. CLARK et D. E. SCHIMMEL : Scalable Pattern Matching for High Speed Networks. *In FCCM '04 : Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 249–257, Washington, DC, USA, 2004. IEEE Computer Society.
8. S. DHARMAPURIKAR, P. KRISHNAMURTHY, T. S. SPROULL et J. W. LOCKWOOD : Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, jan./fév. 2004.
9. C. E. LEISERSON et J. B. SAXE : Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
10. K. MCLAUGHLIN, N. O'CONNOR et S. SEZER : Exploring CAM Design For Network Processing Using FPGA Technology. *In AICT-ICIW '06 : Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, p. 84, Washington, DC, USA, 2006. IEEE Computer Society.
11. M. ROESCH : Snort : Lightweight Intrusion Detection for Networks. *In 13th Systems Administration Conference (LISA'99)*, p. 229–238. USENIX Associations, 1999.
12. R. SIDHU et V. K. PRASANNA : Fast Regular Expression Matching using FPGAs. *In IEEE Symposium on Field-Programmable Custom Computing Machines*, avr. 2001.
13. J. SINGARAJU, L. BU et J. A. CHANDY : A signature match processor architecture for network intrusion detection. *In Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, p. 235–242, avr. 2005.
14. I. SOURDIS et D. PNEVMATIKATOS : Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching. *In FCCM '04 : Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, p. 258–267, Washington, DC, USA, 2004. IEEE Computer Society.
15. L. TAN et T. SHERWOOD : A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *In ISCA '05 : Proceedings of the 32nd Annual International Symposium on Computer Architecture*, p. 112–122, Washington, DC, USA, 2005. IEEE Computer Society.
16. J. van LUNTEREN, T. ENGBERSEN, J. BOSTIAN, B. CAREY et C. LARSSON : XML Accelerator Engine. *In First International Workshop on High Performance XML Processing*, mai 2004.
17. S. WU et U. MANBER : A fast Algorithm for Multi-Pattern Searching. Rap. tech. TR-94-17, Department of Computer Science, University of Arizona, 1994.