Approche générative conjointe logicielle-matérielle au développement du support protocolaire d'applications réseau.

Jigar Solanki

N° d'ordre: 0301

université de BORDEAUX

THÈSE

présentée à

L'UNIVERSITÉ DE BORDEAUX
École Doctorale de Mathématiques et Informatique

par **Jigar SOLANKI**

pour obtenir le grade de :

DOCTEUR
Spécialité : INFORMATIQUE

—

Approche générative conjointe logicielle-matérielle au développement du support protocolaire d'applications réseau.

—

Laboratoire d'acceuil : *LaBRI*
Équipe d'acceuil : *PROGRESS - Programmation Réseaux et Systèmes*

Directeur de thèse : Laurent RÉVEILLÈRE
Co-encadrants : Yérom-David BROMBERG - Bertrand LE GAL

Soutenue le 27 novembre 2014,
Devant la commission d'examen formée de :

MM. :	Gilles MULLER	Professeur au LIP6, Paris	Rapporteurs
	Stéphane FRÉNOT	Professeur à l'INSA de Lyon	
MM. :	Xavier BLANC	Professeur à l'Université de Bordeaux	Président
	Laurent RÉVEILLÈRE	Maitre de conférences à l'ENSEIRB	Examinateurs
MM. :	Yérom-David BROMBERG	Maitre de conférences à l'Université de Bordeaux	
	Bertrand LE GAL	Maitre de conférences à l'ENSEIRB	

Université de Bordeaux
LaBRI - Laboratoire Bordelais de Recherche en Informatique
UMR 5800
351 Cours de la Libération
F-33405 Talence Cedex
—

Remerciements

UN immense Merci ! à tous ceux qui, de près ou de loin, ont contribué à l'aboutissement de ce travail. Il me sera difficile de remercier tout un chacun, tant vous êtes nombreux.

Je tiens tout d'abord à adresser mes plus sincères remercients à Laurent Réveillère pour avoir accepté de diriger mes travaux de thèse. L'autonomie et la liberté d'action qu'il m'a donné à chaque étape m'ont permi de réaliser ces travaux dans les meilleures conditions.

Mes remerciements s'adressent ensuite à David Bromberg et Bertrand Le Gal. Coencadrants, ils ont été présent tout au long de ce travail, attentifs et disponibles malgré leurs nombreuses charges. Leurs compétence, rigueur scientifique et clairvoyance m'ont beaucoup appris.

Je remercie également l'ensemble des membres de mon jury : Monsieur Gilles Muller ainsi que Monsieur Stéphane Frénot pour avoir accepté la fonction de rapporteur, Monsieur Xavier Blanc pour m'avoir fait l'honneur de présider ce jury.

Un merci tout particulier à Serge Chaumette et à Christian Toinard pour m'avoir mis le pied à l'étrier et m'avoir donné l'envie de toujours creuser plus. Qu'ils reçoivent ici ma plus profonde gratitude.

Mention spéciale pour tous les membres et ex-membres de la CVT que j'ai pu cotoyé de près ou de loin, les nombreuses discussions que j'ai pu avoir m'ont beaucoup apporté. Merci donc à Bissyandé, Mickey, Pluto, Renaud, Hugo, Telesphore, Daouda, Jonathan, Dub, Vincent, Yassine, Sebastien, Matthieu, Martin. Le climat de travail sympathique qui règne dans ces quelques mètres carré est propice à l'épanouissement scientifique.

J'exprime ma gratitude à tous mes amis qui m'ont soutenu tout au long de ces années qui ont vu l'aboutissement de ce travail. Leurs multiples encouragements répétés ont été d'un réel réconfort au quotidien.

Mes plus chaleureuses pensées à mes parents et ma famille et belle-famille qui m'ont toujours supporté et m'ont aidé à en arriver là où je suis. Leur tendresse et leur affection me portent tous les jours, merci à eux !

Enfin, je ne saurais trouver de mots pour rendre pleine justice à Laurie, qui s'est trouvé là, les bons comme (surtout) les mauvais jours, pendant que je m'échinais sur ces pages ! Merci à elle ainsi qu'à mes deux petits lutins pour le bonheur qu'ils me procurent au jour le jour.

वक्र तुण्ड महाकाय, सूर्य कोटि सम प्रभः।
निर्विघ्नं कुरू मे देव, शुभ कार्येषु सर्वदा।

सरस्वति नमस्तुभ्यं वरदे कामरूपिणि
विद्यारम्भं करिष्यामि सिद्धिर्भवतु मे सदा

à Kryshant, Leena, Laurie,

à la famille que nous dessinons,

Résumé

Les communications entre les applications réseaux sont régies par un ensemble de règles regroupées sous forme de *protocoles*. Les messages protocolaires sont gérés par une couche de l'application réseau connue comme étant la couche de support protocolaire. Cette couche peut être de nature logicielle, matérielle ou conjointe. Cette couche se trouve à la frontière entre le coeur de l'application et le monde extérieur. A ce titre, elle représente un composant névralgique de l'application. Les performances globales de l'application sont ainsi directement liées aux performances de la couche de support protocolaire associée.

Le processus de développement de ces couches consiste à traduire une spécification du protocole, écrite dans un langage de haut niveau tel que ABNF dans un langage bas niveau, logiciel ou matériel. Avec l'avènement des systèmes embarqués, de plus en plus de systèmes sur puce proposent l'utilisation de ressources matérielles afin d'accroître les performances des applicatifs. Néanmoins, peu de processus de développement de couches de support protocolaire tirent parti de ces ressources, en raison notamment de l'expertise nécessaire dans ce domaine.

Cette thèse propose une approche générative conjointe logicielle-matérielle au développement du support protocolaire d'applications réseaux, pour améliorer leur performance tout en restant ergonomique pour le développeur de l'application. Notre approche est basée sur l'exploitation d'un langage dédié, appellé Zebra pour générer les différents composants logiciels et matériels formant la couche de support. L'expertise nécessaire est déportée dans l'utilisation du langage Zebra et les composants matériels générés permettent d'accroître les performances de l'application.

Les contributions de cette thèse sont les suivantes :

- Nous avons effectué une analyse des protocoles et applications raiseaux. Cette analyse nous a permi d'identifier les composants pour lesquels il est possible d'obtenir des gains de performances.
- Nous avons conçu et exploité un langage dédié, Zebra, permettant de décrire les différentes entités de la couche de support protocolaire et générant les éléments logiciels et matériels la composant.
- Nous avons construit un système sur puce exécutant un système d'exploitation Linux afin d'étayer notre approche. Nous avons conçu des accélérateurs matériels déployables pour différents protocoles réseaux sur ce système et pilotables par les applicatifs.
- Afin de rendre l'accès aux accélérateurs matériels transparent pour les applications réseaux, nous avons développé un intergiciel gérant l'ensemble de ces accès. Cet intergiciel permet à plusieurs applications et/ou à plusieurs clients

d'une même application d'utiliser les accélérateurs pour le traitement des messages protocolaires.
- Nous avons évalué les performances de notre approche dans des conditions réelles. Nous avons comparé ces performances à celles de couches de supports faisant référence dans le domaine. Nous avons constaté un gain de performance conséquent pour l'approche que nous proposons.

Mots clés :
Système sur puce, approche conjointe, protocoles textuels, applications réseaux.

Abstract

Communications between network applications is achieved by using rulesets known as *protocols*. Protocol messages are managed by the application layer known as the protocol parsing layer or protocol handling layer. Protocol parsers are coded in software, in hardware or based on a co-design approach. They represent the interface between the application logic and the outside world. Thus, they are critical components of network applications. Global performances of network applications are directly linked to the performances of their protocol parser layers.

Developping protocol parsers consists of translating protocol specifications, written in a high level language such as ABNF towards low level software or hardware code. As the use of embedded systems is growing, hardware ressources become more and more available to applications on systems on chip (SoC). Nonetheless, developping a network application that uses hardware ressources is challenging, requiring not only expertise in hardware design, but also a knowledge of the protocols involved and an understanding of low-level network programming.

This thesis proposes a generative hardware-software co-design based approach to the developpement of network protocol message parsers, to improve their performances without increasing the expertise the developper may need. Our approach is based on a dedicated language, called Zebra, that generates both hardware and software elements that compose protocol parsers. The necessary expertise is deported in the use of the Zebra language and the generated hardware components permit to improve global performances.

The contributions of this thesis are as follows :

- We provide an analysis of network protocols and applications. This analysis allows us to detect the elements which performances can be improved using hardware ressources.
- We present the domain specific language Zebra to describe protocol handling layers. Software and hardware components are then generated according to Zebra specifications.
- We have built a SoC running a Linux operating system to assess our approach. We have designed hardware accelerators for different network protocols that are deployed and driven by applications.
- To increase sharing of parsing units between several tasks, we have developped a middleware that seamlessly manages all the accesses to the hardware components. The Zebra middleware allows several clients to access the ressources of a hardware accelerator.
- We have conducted several set of experiments in real conditions. We have com-

pared the performances of our approach with the performances of well-known protocol handling layers. We observe that protocol handling layers baded on our approach are more efficient that existing approaches.

Keywords :
System on chip (SoC), co-design based approach, textual protocols, network applications.

Table des matières

Table des figures

1

Introduction

Sommaire

1.1 Contexte

La dernière décennie a vu de gigantesques bouleversements dans les méthodes de consommation de l'information par le grand public. L'avènement de l'Internet conjugé aux progrès technologiques fulgurants en matière d'équipements de communications a métamorphosé l'accès quotidien à cette information. La domotique, les loisirs numériques, les équipements informatiques personnels, l'automobile, les villes intelligentes sont quelques uns des innombrables domaines où les équipements communicants prolifèrent de manière croissante et où les utilisateurs souhaitent accéder aux fonctionalités les plus adéquates en fonction de leur environnement et des caractéristiques de leurs équipements.

Ces fonctionalités sont fournies par des applications interagissant avec le réseau ou plus simplement des applications réseaux. Les communications entre ces applications sont régies par un ensemble consensuels de principes figurant dans un formalisme appellé *protocole*. Historiquement, les protocoles réseaux étaient destinés à la communication entre machines où l'utilateur final n'intervenait qu'à moindre échelle. Depuis, ils ont évolué pour répondre aux besoins croissants des utilisateurs et sont devenus des protocoles applicatifs permettant aux utilisateurs d'interagir davantage directement avec leurs équipements. C'est le cas du protocole HTTP, construit à l'origine pour le transfert de pages Internet et qui permet, aujourd'hui, de répondre aux besoins spécifiques d'une application puisqu'il a été, à l'origine, concu de manière extensible. Ainsi

a emmergé toute une famille de protocoles dit *à la* HTTP, basés sur celui-ci. C'est, par exemple, le cas de protocoles dédiés aux services de communications comme le protocole SIP, de protocoles dédiés à la découverte de service comme le protocole UPnP, ou encore de protocoles destinés à la lecture en ligne de contenus multimédia comme le protocole RTSP. Ces protocoles sont fondés sur les caractéristiques principales de HTTP : un échange de requêtes/réponses, un encodage textuel (par opposition à un encodage binaire), la structure des messages échangés ou encore l'extensibilité du protocole.

Les applications réseaux sont divisées en deux couches : une couche de *logique applicative* et une couche de *support protocolaire*. La logique applicative fournit la fonctionalité finale à l'utilisateur. La couche de support protocolaire est en charge de la manipulation des messages qui transitent sur le réseau. Elle se charge d'analyser les messages qui proviennent du réseau ou de construire les messages à transmettre. Cette couche représente ainsi l'interface entre la logique applicative, ou l'application (par abus de langage), et le monde extérieur. A ce titre, l'efficacité de l'application est intimement liée aux performances de la couche de support protocolaire. Afin de répondre à un volume de communications de plus en plus important, cette couche se doit d'être réactive et assurer un débit important de messages traités.

L'augmentation constante des débits des réseaux couplé à l'apparition de nouveaux dispositifs informatiques toujours plus petits et ne disposant pas toujours de grandes quantités de ressources a conduit à l'émergence de nouveaux domaines applicatifs comme les réseaux de capteurs ou les drônes grand public. Ces nouveaux domaines applicatifs ont vu l'émergence de nouveaux types d'application reposant sur de nouveaux protocoles applicatifs. Malgré la standardisation de ces nouveaux protocoleset leur utilisation croissante dans les applications réseaux aux besoins diverses, peu de progrès ont été réalisés dans le processus du développement de leur support protocolaire. De plus, historiquement, la couche de support protocolaire est développée en grande partie de manière logicielle. Néanmoins, avec l'apparition de nouveaux dispositifs communicants intégrant des ressources qu'il est possible de configurer à la volée pour des besoins spécifiques d'une application (ressources reconfigurables), le besoin d'une nouvelle approche tirant profit de ces ressources matérielles se fait ressentir.

En effet, les difficultés principales dans le processus de développement d'un support protocolaire, logiciel ou matériel, sont de plusieurs nature. Il faut tout d'abord connaître la logique applicative afin de fournir un service pertinent à l'utilisateur. Ensuite, il faut avoir une connaissance approfondie du protocole utilisé et donc étudier sa spécification. Enfin, il faut disposer d'une double expertise logicielle-matérielle : expertise logicielle pour la programmation système et réseau et expertise dans le domaine de l'électronique et des langages de description matérielle pour pouvoir utiliser les ressources matérielles disponibles. Cepandant, très peu de personnes peuvent prétendre disposer de l'ensemble de ces compétences. Développer une couche de support protocolaire à la fois efficace et accessible aux programmeurs non chevronnés est une tâche fastidieuse.

1.2 Thèse

La thèse présentée dans ce manuscrit propose une approche d'aide au développement du support protocolaire d'applications réseaux tirant profit des possibilités d'accélération matérielle offerte par les ressources matérielles qu'embarquent les dispositifs communicants.

Cette approche est basée sur un langage dédié permettant de générer automatiquement une couche de support protocolaire tirant profit des ressources reconfigurables. L'intérêt dans l'utilisation de ces ressources réside dans leur rapidité d'exécution : ces ressources sont programmées pour l'exécution d'une tâche spécifique, par conséquent, celle-ci s'effectue de manière très efficace. Un langage dédié ou métier est un langage de programmation spécialisé pour un domaine métier particulier. Un tel langage est déclaratif et permet de masquer les détails de l'implémentation au programmeur. La restriction à un domaine particulier engendre une expressivité du langage amoindrie ce qui a pour effet de faciliter l'analyse des programmes écrits dans ce langage.

Notre approche se fonde sur l'introduction d'un langage, nommé Zebra, dédié au développement du support protocolaire conjoint logiciel-matériel pour les applications réseaux. Ce langage est basé sur un formalisme largement répandu dans les standards de protocoles réseaux. Ce langage permet de générer une couche de support protocolaire intégrant des accélérateurs matériels correspondants aux protocoles utilisés ainsi qu'une couche logicielle permettant de les manipuler. Cette couche de support protocolaire est de plus adaptée aux besoins spécifiques d'une application, ce qui permet d'en accroître l'efficacité. Seules les parties du message utilies à l'application sont analysées. Une spécification Zebra est traitée et analysée par un compilateur avant que ces accélérateurs matériels soient générés de pair avec la couche logocielle permettant de les piloter. Cette couche de support peut alors être couplée à la logique applicative correspondante.

1.3 Contributions

Cette thèse présente deux contributions principales. Tout d'abord, nous montrons la faisabilité d'une approche langage dans le processus de développement du support protocolaire d'applications réseaux dans un environnement contraint en ressources et disposant de ressources matérielles configurables. Nous présentons les différentes étapes de conception de notre langage dédié. Ensuite, nous évaluons les performances de notre approche par une série d'expérimentations réalisées tout au long de nos travaux.

Zebra. Nous introduisons un langage dédié, nommé Zebra, pour la spécification du support protocolaire. Ce langage est basé sur un formalisme largement utilisé dans les spécifications de protocoles réseaux, permettant ainsi de réduire l'effort de développement nécessaire. Nous présentons les différents composants générés et la manière dont ils s'intègrent dans une infrastructure existante.

Plateforme Nous avons mis au point une plateforme logicielle et matérielle dédiée regroupant l'ensemble des éléments nécessaires pour l'exécution d'un support protoco-

laire logiciel-matériel ainsi que pour l'exécution des applications réseaux qui l'utilisent.

Performances. Nous présentons une évaluation de notre approche en montrant l'impact de l'utilisation de Zebra sur les performances du support protocolaire d'une application réseau. Nous montrons que la conception conjointe logiciel-matériel représente le meilleur compromis entre efficacité et ergonomie de développement dans notre contexte.

1.4 Organisation du manuscrit

Cette thèse s'articule autour de trois parties principales : la présentation du contexte et des problématiques associées, la description de l'approche langage que nous proposons et enfin l'évaluation des performances de notre solution.

Contexte. La première partie porte sur l'étude du contexte scientifique dans lequel nous nous plaçons. Le chapitre 2 présente les communications et services réseaux dans un environnement ubiquitaire. Ce chapitre détaille la notion de couche applicative dans laquelle se place nos travaux. Le chapitre 3 présente les langages dédiés et décrit différents travaux significatifs dans notre domaine. Il présente ensuite l'intérêt de l'utilisation de l'accélération matérielle avant de décrire d'autres travaux intégrant l'utilisation de celle ci. Le chapitre 4 se propose de définir les différentes étapes que nous avons suivie dans le processus de conception de notre approche, allant de l'identification de la problématique à l'évaluation de l'approche proposée.

Approche proposée Dans la seconde partie, nous présentons les étapes du développement de notre approche. Le chapitre 5 décrit la plateforme d'expérimentations que nous avons mise au point afin d'étayer notre approche. Le chapitre 6 présente l'architecture logicielle de notre approche. Elle inclut les éléments logiciels nécessaires à l'application cliente pour pouvoir s'exécuter sur la plateforme, ainsi qu'un intergiciel rendant l'accès aux ressources transparents pour les applicatifs. Enfin, le chapitre 7 mesure les performances de notre approche en terme d'efficacité et rapidité d'exécution et de consommation mémoire.

2

Services réseaux en environnement ubiquitaire

LES RESEAUX INFORMATIQUES permettent de relier différents équipements informatiques entre eux afin qu'ils puissent s'échanger des données.Ces éléments, qui se limitaient à des ordinateurs personnels ou des téléphones portables, il y a une quinzaine d'années, se sont largement diversifiés depuis. Malgré diverses contraintes telles que la consommation énergétique ou la capacité de calcul disponible, l'utilisateur final a accès, *via* ces réseaux, à une multitude d'applications de plus en plus complexes allant du domaine de la domotique à celui de la santé en passant par les transports.

Ce chapitre s'articule autour de ces services et applications réseaux dans un milieu ubiquitaire et se concentre plus particulièrement sur la manière dont ces appications manipulent les messages. Suite à une brève présentation des communications réseaux, nous mettons l'accent sur l'informatique ubiquitaire et l'hétérogéneité des composants qui la caractérisent. Nous présentons ensuite les modèles de réseaux en couche pour finalement nous focaliser sur la couche applicative et le traitement des messages associés.

Sommaire

2.1 Services ubiquitaires

2.1.1 L'informatique ubiquitaire

L'informatique *ubiquitaire* ou *diffuse* ou *pervasive* est un concept introduit pour la première fois au début des années 1990, par Mark Weiser [Wei91], concept qu'il a raffiné par la suite [Wei93, Wei94, Wei99]. Dans ses articles, il propose une conception révolutionnaire dans les technologies de l'information afin de rendre les ordinateurs complètements *invisibles* et *transparents*. Ainsi, en s'appuyant sur la miniaturisation des composants, la transmission des données sans-fil ou encore la baisse des coûts de production, il prône le dépassement du modèle de l'ordinateur de bureau traditionnel au profit d'un modèle où le traitement et l'accès à l'information serait complètement intégré dans les objets et les environnements de notre activité quotidienne.

L'idée sous-jacente de l'informatique ubiquitaire est de pouvoir accéder à n'importe quelle information, de n'importe où et à n'importe quel moment, grâce à des dispositifs informatiques transparents, complètement intégrés à notre environnement quotidien afin d'améliorer l'expérience utilisateur. Le Larousse définit l'*ubiquité* comme la faculté d'être présent en plusieurs lieux à la fois.

L'une des analogies les plus célèbre de Weiser pour illustrer ce concept est celle de l'écriture, adoptée par les civilisations depuis des millénaires. Elle est totalement intégrée dans notre environnement quotidien, puique les livres ne sont pas les seuls éléments contenant du texte écrit. Elle est omniprésente et universelle, nous la remarquons à peine.

Il devrait en être de même pour les « ordinateurs » : ils doivent se s'intégrer à notre cadre de vie, devenir transparents afin que les utilisateurs ne soient plus conscients de leur présence en les intégrant dans des objets et endroits que nous utilisons quotidiennement comme des vêtements ou des meubles [CBM04]. Les nouvelles technologies de l'information et de la communications doivent être mises au service de l'utilisateur, et non l'inverse comme l'affirme Weiser :

« *Les technologies les plus profondes sont celles qui disparaissent. Elles se mêlent dans le tissu de la vie de tous les jours jusqu'au point où elles en deviennent indiscernables.* »

Cette vision basée sur la notion de *transparence* est à l'origine de plusieurs interprétations qui sont aujourd'hui regroupées en autant d'axes de recherche en informatique. Satyanarayanan dresse un panorama des tendances offertes par l'informatique ubiquitaire [Sat01]. Basée en partie sur les systèmes distribués et l'informatique nomade, elle doit également prendre en compte de nouveaux environnements comme les espaces intelligents ou l'hétérogénéité des dispositifs.

Cette vision futuriste, à l'époque, a emmergé grâce à plusieurs facteurs (Fig. 2.1) :

— **L'émergence et la prolifération des dispositifs.** Depuis une cinquantaine d'années, la réduction du coût des dispositifs informatiques ainsi que la diminution de la taille des composants électroniques sont à l'origine de cette prolifération. En effet, cette démocratisation a permi passer de quelques « ordinateurs » par

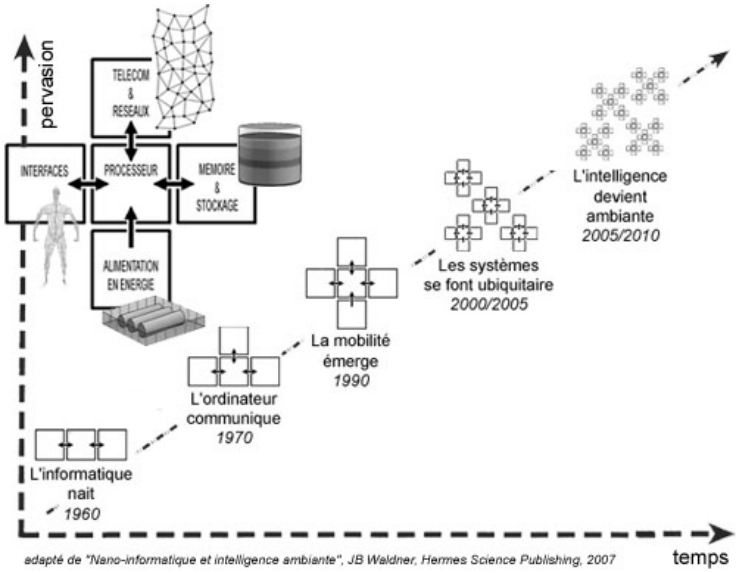


FIGURE 2.1 – Evolution de l'informatique. *Source :* "Nano-informatique et intelligence ambiante", *JB Waldner, Hermes Science Publishing, 2007*

universités à un ordinateur par famille, puis par personne. Aujourd'hui, chaque personne utilise ou possède plusieurs dispositifs tels que des tablettes, des téléphones portables, des ordinateurs portables ou encore des liseuses numériques ;

— **Les communications *sans fil***. Les dispositifs cités précédemment sont, pour la plupart d'entre eux, connectés les uns avec les autres grâce à des technologies de communications sans fil (e.g Bluetooth, IEEE 802.11). Cette connectivité permet de s'affranchir des contraintes physiques liées à l'utilisation de supports physiques de communications (tels que des fils) afin d'accroître la mobilité et l'expérience utilisateur.

— **Un environnement en perpetuel évolution.** L'hétérogénéité des dispositifs ainsi que des services proposés, la distribution des données à travers le réseau ou encore la capacité de l'environnement à s'auto-configurer sont autant de facteurs qui, avec la mobilité, rendent ce milieu extrêmement mouvant et dynamique où les composants interagissent de manière spontanée [KF02].

L'utilisateur peut ainsi obtenir l'information la plus pertinente suivant l'endroit où il se trouve, la connectivité des équipements dont il dispose, le type d'information que ces dispositifs peuvent traiter, etc. En effet, les fonctionalités mise à disposition des utilisateurs par ces équipements informatiques peuvent être assimilés à des **services**. [PG03]. Parmi de multiples exemples, on peut citer Thiliez *et al.* [TD04] qui définissent des requêtes évaluées suivant la localisation physique de l'utilisateur ; ou encore l'office de tourisme de Dublin qui s'est doté d'un système, nommé *Ad-Me*, fournissant des services de publicité basés sur la localisation des utilisateurs [HO04]. Nous illustrons cette prolifération de services disponibles à travers un scénario typique montrant les différents aspects de l'informatique ubiquitaire.

2.1.2 Services et applications

Les scénarii permettant d'illustrer les objectifs de l'informatique ubiquitaire sont nombreux, nous en présentons un reflétant la transparence des dispositifs mobiles que présentait Weiser.

> Un étudiant, appelons le François, termine ses planches de présentation d'un article à son hôtel. Sa présentation est proche, et François ne les a toujours par terminés. Il sauvegarde alors son travail sur son ordinateur. Au moment de quitter son hôtel, il saisit son assistant numérique personnel afin de pouvoir continuer à les préparer. Sa présentation est automatiquement transférée de son ordinateur portable vers son assistant numérique. Il se sert ensuite de son téléphone portable et de la connectivité Wi-Fi offerte par la ville pour localiser le lieu de la conférence et l'indiquer au taxi. Le système GPS du taxi calcule automatiquement le coût de la course. A destination, François se sert de la technologie de paiment sans contact intégré à son téléphone portable afin de régler le taxi. En arrivant dans la salle, un service permet de transférer sa présentation sur l'ordinateur de l'amphithéatre. En outre, un service de visioconférence permet à des intervenants extérieurs, qui n'ont pas pu se déplacer à la conférence, de suivre et commenter la présentation de François en direct.

Ce scénario illustre l'utilisation de plusieurs applicatifs et services depuis différents dispositifs communiquants pour un accès universel et omniprésent à l'information à travers diverses technologies issues de différents domaines à l'origine de nombreux travaux de recherche [Sat01].

Bien qu'elles nous soient aujourd'hui familières, l'ensemble de ces services (géolocalisation, visioconférence, technologie sans contact) requiert un traitement de l'information efficace, prenant en compte les diverses caractéristiques de chaque dispositif communicant (mémoire disponible, batterie disponible) mais aussi les caractéristiques de l'environnement dans lequel ils évoluent (localisation physique du dispositif, connectivité disponible). Ces services sont implémentés sur ces dispositifs communiquants sous la forme d'applications réseaux. Ces applications interagissent avec l'utilisateur d'une part, en lui fournissant l'information utile et d'autre part, elles interagissent avec leur environnement en émettant et recevant des messages afin de traiter et faire transiter les données utilisateurs.

Le traitement efficace de ces messages qui s'échangent perpétuellement sur les réseaux est un des défi que tente de relever cette thèse, dans un contexte où la réactivité est primordiale alors que les dispositifs informatiques ne fournissent pas toujours une puissance de calcul adéquate pour la fournir.

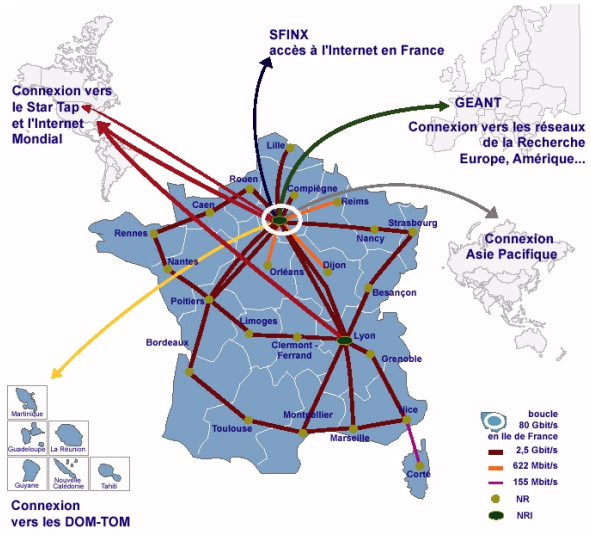


FIGURE 2.2 – Réseau RENATER

2.2 Réseaux informatiques

2.2.1 Communications réseaux

2.2.1.1 Présentation

De manière générique, un « réseau » est défini par un ensemble d'entités (personnes, programmes, objets) interconnectées les unes avec les autres. Il permet de faire circuler des éléments entre ces entités selon des règles définies au préalable. Les *réseaux informatiques* sont ainsi définis par un ensemble d'équipements informatiques reliés entre eux et échangeant des données les uns avec les autres. On parle alors plus généralement de *noeuds réseaux* ou plus simplement de *noeuds*. Connecter des éléments informatiques entre eux offre de multiples avantages parmi lesquels on peut citer le partage de ressources (fichiers, imprimantes, etc), la communication entre personnes ou programmes, un accès universel à l'information (base de données en réseau) ou encore la réduction des coûts. A titre d'exemple, la figure 2.2 présente une cartographie du réseau RENATER (**RE**seau **N**ational de télécommunications pour la **T**echnologie, l'**E**enseignement et la **R**echerche) permet de fédérer l'ensemble des universités et différents centres de recherche en France depuis le début des années 1990.

Ces données circulent sur le support physique de communication, appellé aussi *medium* de communication, sous la forme d'une série continue de `0` et de `1`, appellés des *bits*.

Un message utilisateur, ou applicatif, ne peut être envoyé tel quel sur le médium. Il doit etre découpé en plusieurs parties, encapsulé dans des structures permettant de ne pas perdre le fil du message et envoyé morceau par morceau à travers le réseau sous la forme d'un flot de bits. Au fur et à mesure de la réception du flot de bits, le message est reassemblé et le contenu qui sert effectivement à l'utilisateur final est délivré. On parle alors de *données utiles*.

2.2.1.2 Taille et hétérogeneité des réseaux

Suivant la distance géographique séparant les différents équipements, le débit disponible ou le nombre d'entités qui s'y connectent, les réseaux informatiques sont classés en différentes catégories. Les réseaux personnel et local permettent d'interconnecter les équipements d'un utilisateur ou d'une entreprise sur quelques dizaines de mètres. A l'opposé, les réseaux métropolitain et étendu servent à relier plusieurs villes ou continents géographiquement éloignés et ainsi, transporter l'information sur de grandes distances.

Un point important à souligner à travers cette classification concerne la diversité des dispositifs utilisés pour relier ces différents réseaux, tandis que l'utilisateur attend un service transparent quelque soit le dispositif qu'il utilise. Ainsi, les applications en charge de ces services doivent s'adapter au matériel. Cette caractéristique rajoute de la complexité au développeur d'applications réseaux, comme nous le verrons à la section 2.3, puisque celles ci doivent prendre en compte un panorama toujours plus importants de dispositifs sur lesquelles elles peuvent s'exécuter. De plus, une même application peut rendre le même service à l'utilisateur en utilisant plusieurs types de connectivité différentes. Ainsi, un smartphone peut utiliser une connexion sans fil de type Wi-Fi [1], une connexion pour la téléphonie mobile de type 4G [2] ou encore une connexion satellitaire. Bien que cela soit complètement transparent pour l'utilisateur, l'application rendant le service doit gérer les différents types de réseaux et par conséquent, traiter un ou plusieurs types de messages différents pour chacun de ces réseaux, ce qui alourdit davantage la tâche du développeur de ces applications.

Les applications peuvent rendre un service à l'utilisateur, mais également à d'autres applications distantes. On distingue alors plusieurs modes de communications entre ces applications.

2.2.2 Protocoles réseaux

Les entités communicantes doivent s'accorder sur un ensemble de règles afin de rendre l'information échangée exploitable. Ce cadre formé de règles et conventions de communication est appellé *protocole de communication* ou plus simplement *protocole*.Un protocole définit d'une part, la manière dont les messages sont structurés et d'autre part, la manière dont ils sont échangés entre les entités communicantes.

Le but étant de transmettre l'information, les entités communicantes doivent s'accorder sur différents éléments comme le média de communication à utiliser, le type d'information à transmettre, etc. Ainsi, une communication réseau met en oeuvre plusieurs protocoles structurés sous formes de *couches réseaux*. Chaque couche effectue une

1. Wi-Fi est un ensemble de protocoles de communication sans fil régis par les normes du groupe IEEE 802.11 (ISO/CEI 8802-11).
2. 4G réprésente la 4ème génération de standards pour la téléphonie mobile, successeur de la 2G et de la 3G.

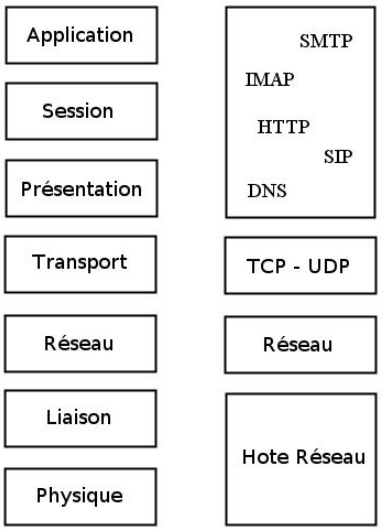


FIGURE 2.3 – Modèles OSI et TCP/IP

tâche en particulier au profit de la couche qui lui est immédiatement supérieure. Cette structuration facilite le transfert de l'information utile.

Cette structuration hiérarchique de couches et de protocoles forment une **une architecture de réseau**. Les architectures de réseau les plus répandues de nos jours sont *le modèle OSI* et le modèle TCP/IP.

2.2.2.1 Modèles de référence

Les modèles OSI et TCP/IP (Fig. 2.3) et leurs protocoles associés sont deux modèles de référence réseau les plus répandus. Alors que les protocoles définis par le modèle OSI sont rarement utilisés en pratique, les concepts décrits par le modèle abstrait restent largement robustes et approuvés, à l'opposé du modèle TCP/IP dont les protocoles sont déployés à grande échelle.

Les deux modèles proposent une architecture orientée *communication réseaux* à leur base et une architecture orientée *applications utilisateurs* à leur sommet. Les couches composant l'architecture orientée communication réseaux se chargent de fournir un système de communication stable, robuste et performant aux couches supérieures. Ceci inclut le contrôle d'erreur, la congestion, la disponibilité des liens, les chemins d'acheminement des messages, etc. A l'opposé, les couches qui forment l'architecture orientée applications utilisateurs traitent des aspects liés aux fonctionnalité des programmes. Nous nous interessons, dans le cadre de nos travaux, à cette couche orientée *applications et services utilisateurs*, ou couche applicative.

2.3 Couche applicative

La couche *Application* traite de l'ensemble des applications et services utilisateurs. Ces applications sont basées sur des protocoles afin d'homogéneiser leur communication, tout comme les protocoles de couches plus basses que nous avons rencontré à la section 2.2.2.1. Le plus souvent, ces applications communiquent via le mode *client/serveur*.

2.3.1 Structure d'une application réseau

A l'instar des architectures de réseaux sous forme de couches, les applications réseaux suivant également ce modèle de couches superposées.

Une couche basse englobe les paramètres que l'utilisateur ou l'environnement passe à l'application dans des messages conformes aux spécifications du protocole. Cette couche logicielle de traitement de messages est nommée *couche de traitement de messages protocolaires* ou plus simplement *support protocolaire*, détaillée dans la section 2.3.4. Cette couche se charge d'extraire l'information utile du message afin de la transmettre à la couche qui lui est supérieure. Le support protocolaire représente l'interface entre le réseau et les messages qui y transitent et le traitement de l'information en elle même.

Cette couche supérieure est appellée *logique applicative* et représente le coeur de l'application proprement dite. Elle accomplit le traitement à proprement parler de l'information et rend le service à l'utilisateur. Cette logique applicative est présente sous diverses formes et fortement liées à la couche de support protocolaire. Il peut s'agir de *clients*, comme des navigateurs web utilisés quotidiennement pour naviguer sur l'Internet ou des clients de messagerie électronique, respectivement basés sur la couche de support protocolaire du protocole HTTP [FGM+99] et du protocole SMTP [Kle01] et IMAP [Cri94]. A l'opposé se trouvent des serveurs fournissant les pages web et permettant d'envoyer les courriels, basés sur la couche de support des même protocoles. Un autre type de logique applicative de plus en plus répandu dans un contexte d'hétérogeneité permanent est celui des passerelles réseaux. Une passerelle permet de traduire un ensemble de messages d'un protocole dans un ensemble de messages correspondant à un autre protocole. Ces passerelles permettant de relier différents réseaux hétérogènes, supportant des protocoles différents, sont de plus en plus présentes au coeur des réseaux et sont même générés automatiquement à partir de spécifications des protocoles supportés [BRLM09].

La communication entre ces applications se fait par la transmission de messages conformes à des protocoles. Néanmoins, ces messages doivent adopter un alphabet commun et les applications concernées doivent s'accorder sur un format de message spécifique permettant de décrire les messages.

2.3.2 Format des messages

Le format des messages a évolué parrallèlement avec la diversification et la performance des réseaux sur lesquels ils sont échangés. A l'origine, les messages étaient codés

```
date            =       day month year
year            =       4*DIGIT / obs-year
month           =       (FWS month-name FWS) / obs-month
month-name      =       "Jan" / "Feb" / "Mar" / "Apr" /
                       "May" / "Jun" / "Jul" / "Aug" /
                       "Sep" / "Oct" / "Nov" / "Dec"
day             =       ([FWS] 1*2DIGIT) / obs-day
```

FIGURE 2.4 – Format de date défini dans la RFC 2822

sous forme binaire. Elle présente l'avantage d'un traitement rapide lorsque la communication n'implique que des machines.

Lorsque cette communication implique qu'un humain accède et interprète lui même l'information transmise, cet encodage représente plus un handicap. Outre le fait que l'utilisateur humain comprend difficilement le langage binaire, la puissance de calcul et les performances des machines et des réseaux rendent possible le choix d'une autre méthode d'encoder les messages. De manière naturelle, nous pouvons penser à un encodage textuel. Les messages sont envoyés, non plus sous forme de bits non interprétés, mais directement sous forme de *texte brut*, plus facilement compréhensible par l'utilisateur. Les protocoles définissant des messages sous forme de texte sont appelés **protocoles textuels**.

Néanmoins, bien que l'utilisateur humain participe à la communication, celle-ci est toujours réalisée par une machine. A ce titre, elle ne peut interpréter le langage textuel à moins qu'on lui montre la manière de l'effectuer, tout comme nous apprenons une langue étrangère. Cet apprentissage se fait avant tout par un formalisme qui décrit la syntaxe des mots et la grammaire du langage. Il en va de même pour les protocoles textuels.
La *métasyntaxe* Augmented Backus-Naur Form [CO97](ABNF) permet de formaliser la grammaire qui décrit la structure des messages conformes au protocole. A l'origine, les langages de programmation avaient également besoin de formalisme pour que l'ensemble des développeurs puissent s'accorder sur la syntaxe du langage qu'ils utilisaient. Ce formalisme a été standardisé à travers la *métasyntaxe* Backus-Naur Form [Knu64] dont l'ABNF dérive.

A titre d'exemple, la figure 2.4 présente le formalisme permettant de définir une date [Res01].

Une autre possibilité de définir la syntaxe des messages des protocoles textuels est de s'appuyer sur le langage XML [BPSM+08]. La syntaxe du XML extensible du langage XML permet de définir différents espaces de noms et ainsi forme une cible de choix pour la spécification du format de messages protocolaires. Les outils associés permettent une manipulation rapide et aisée des messages spécifié sous forme de schéma XML.

```
<?xml version="1.0" encoding="utf-8" ?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <soap:Body>
   <GetCustomerInfo xmlns="http://tempUri.org/">
<CustomerID>1</CustomerID>
<OutputParam />
  </GetCustomerInfo>
</soap:Body>
</soap:Envelope>
```

FIGURE 2.5 – Message de requête SOAP

2.3.3 Echantillon de protocoles

Comme nous l'avons vu auparavant, l'informatique ambiante utilise une multitude de protocoles divers et variés. Les messages protocolaires peuvent être encodés de plusieurs façons.

Les protocoles binaires sont des protocoles où le message est directement encodé sous formes de bits. La particularité de ces protocoles est d'avoir un traitement rapide lorsqu'il s'agit de faire communiquer deux entités informatiques. Les protocoles historiques de la couche réseau et de la couche transport que sont les protocoles TCP [Pos81c] et IP [Pos81b] utilisent cet encodage. Des protocoles applicatifs peuvent également être basés sur cet encodage comme le protocole NTP [Mil96] permettant de synchroniser l'horloge d'un ordinateur avec celle d'un serveur de référence. Toutefois, l'augmentation massive de dispositifs informatiques hétérogènes tant sur le plan matériel que logiciel, ces dernières années, rend l'utilisation de protocoles binaires spécifiques à un type de matériel et d'applications précis caduque. L'utilisation de protocoles textuels standards palie ce manque de flexibilité inhérente aux systèmes embarqués.

Les protocoles XML [WM98] sont des protocoles basés sur un langage permettant d'agencer toute sorte de données allant de feuilles de calcul à des paramètres de configuration en passant par des transactions financières. Le développement du langage XML a commencé en 1996. Il regroupe un ensemble de conventions pour la conception de format texte favorisant la structuration de données. De plus en plus de protocoles applicatifs s'appuient sur ce langage. C'est le cas du protocole XHTML [Sta02], successeur du protocole HTML [BLC95] ; ou encore du protocole SOAP [ML07] - Simple Object Access Protocol - permettant la transmission de messages entre entités distantes illustré par un message à la figure 2.5.

XML est modulaire et permet de définir de nouveaux format de document à partir

de formats préexistants. Il a été initiallement conçu pour le traitement de données hors ligne. A ce titre, l'utilisation de protocoles basés sur XML dans le contexte de systèmes ubiquitaires pose des problèmes relatifs aux contraintes d'efficacité dûes aux ressources potentiellement limitées des dispositifs.

Enfin, les protocoles textuels, beaucoup plus largement répandus, permettent de pallier ces limitations. La figure 2.6 représente des messages des protocoles HyperText Transfert Protocol - HTTP [BLFF96] et Simple Mail Transfer Protocol - SMTP [Pos81a]. Le premier a été historiquement crée pour le transport de données textuelles, notamment des pages de sites Internet. Son utilisation s'est alors diversifiée au point que l'on retrouve des protocoles basés sur HTTP dans le domaines du multimédia comme c'est le cas pour le protocole Real Time Streaming Protocol - RTSP [SRL98]. Le deuxième est un protocole servant pour le transport, l'envoi et la réception de courrier électronique.

```
220 smtp.labri.fr SMTP Ready
HELO client
250-smtp.labri.fr
250-PIPELINING
250 8BITMIME
MAIL FROM: <pierre@labri.fr>
250 Sender ok
RCPT TO: <paul@labri.fr>
250 Recipient ok.
DATA
354 Enter mail, end with "."
Subject: Test

Corps du texte
.
250 Ok
QUIT
221 Closing connection
```

```
GET /docs/index.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
(blank line)
```

FIGURE 2.6 – Exemples de messages SMTP et HTTP

2.3.4 Support protocolaire

L'utilisation de protocoles textuels permet de rendre la communication entre entités hétérogènes plus flexible. Le protocole Session Initiation Protocol - SIP [RSC+02], par exemple, est déployé dans les réseaux de capteurs [Kri06] ou encore dans les réseaux mobiles *ad-hoc* [LMHR05, SBRA07]. Elle présente également beaucoup d'avantages, notamment parce qu'il est plus aisé d'ajouter de nouvelles fonctions sans perturber les fonctionalités déjà opérationnelles. Le format des messages des protocoles textuels est

décrit à l'aide de la métasyntaxe ABNF (cf. Section 2.3.2). Lorsque de nouvelles fonctionalités sont rajoutées à un protocole, sa description en est enrichie. L'inconvénient de cette méthode réside dans l'accumulation de ses différentes versions et, en particulier, dans le développement du support protocolaire de celles-ci. De plus, ce support logiciel doit être écrit à la main ce qui peut représenter une tâche fastidieuse et difficile à maintenir.

En effet, comme nous l'avons vu à la section 2.3.1, la couche de support protocolaire représente la liaison entre la couche réseau et la logique applicative. A la réception de messages, elle est en charge de l'extraction de l'information utile passée à l'application. De manière analogue, elle se charge de l'encapsulation de l'information utile avant l'envoi des messages sur le réseau. Elle offre ainsi une interface complète de manipulation des messages à la logique applicative. Néanmoins, la perpétuelle évolution des méthodes de communication requiert une adaptation rapide de leur couche de support protocolaire. De plus, l'augmentation du débit des réseaux informatiques impose à cette couche une efficacité croissante dans le traitement des données. L'un des défi que tente de relever les travaux de cette thèse concerne l'efficacité de ce support protocolaire.

2.3.4.1 Efficacité

La couche de support protocolaire forme un composant crucial d'une application puisqu'elle est en charge de l'acheminement des messages entre les couches plus basses et le coeur de l'application. A ce titre, l'efficacité de cette couche de support protocolaire est un élément primordial lors processus de développement d'une application réseau. En effet, le support doit etre efficace en temps. Les services fournis par les applications reposant sur ces supports doivent être fournis de manière rapide et fluide. Le support doit également être efficace en mémoire. Les systèmes contraints n'embarquent pas toujours une puissance de calcul et une quantité de mémoire confortables pour l'exécution du support.

En effet, S. Wanke *et. al* [WSKW07] mesurent les performances de SIP Express Router - SER [POJK03], un serveur mandataire SIP considéré comme la référence en terme d'efficacité de sa couche de support protocolaire. Les expérimentations ont souligné que le temps d'exécution du support protocolaire représente près de 25% du temps de traitement global. D'autres travaux mettent également en avant cette quête d'efficacité, comme ceux de Cortes *et. al* [CE02, CEE04] dont les tests montrent un temps d'exécution variant de 33% à près de 88% du temps de traitement global. Les performances de la couche de support protocolaire conditionnent ainsi les performances de l'application réseau reposant dessus et influencent directement la qualité du service rendu.

Toutefois, du point de vue du développeur d'applications réseaux, le foisonnement des protocoles de communication rend la mise au point de leur couche de support contraignante.

2.3.4.2 Ergonomie de développement

Un protocole est généralement décrit à l'aide de la métasyntaxe ABNF et permet d'en définir le format des messages. Le développeur doit avoir, non seulement, une

```
[...]
switch (len)
{
 case 3:
   switch (method[0])
   {
     case 'P':
       return (method[1] == 'U'
          && method[2] == 'T'
          ? M_PUT : UNKNOWN_METHOD);
      case 'G':
        return (method[1] == 'E'
          && method[2] == 'T'
          ? M_GET : UNKNOWN_METHOD);
      default:
        return UNKNOWN_METHOD;
   }

 case 4:
   switch (method[0])
   {
     case 'H':
       return (method[1] == 'E'
         && method[2] == 'A'
         && method[3] == 'D'
         ? M_GET : UNKNOWN_METHOD);
     case 'P':
       return (method[1] == 'O'
         && method[2] == 'S'
         && method[3] == 'T'
         ? M_POST : UNKNOWN_METHOD);
     [...]
 }
```

```
[...]
register_one_method(p,"GET",M_GET);
register_one_method(p,"PUT",M_PUT);
register_one_method(p,"POST",M_POST);
[...]
```

FIGURE 2.7 – Extrait du code analysant les méthodes HTTP dans Apache

connaissance approfondie de des spécifications de la syntaxe des protocoles utilisés en ABNF mais également des compétences spécifiques en programmation réseau et génie logiciel.

Ce fossé entre les différents niveaux d'abstraction est souligné dans les figures 2.7 et 2.8. La figure 2.7 représente l'extrait du code permettant l'analyse des méthodes HTTP du support protocolaire dans le serveur Apache, référence en matière de serveur web. La figure 2.8, quant à elle, représente un extrait de l'ABNF de HTTP définissant la syntaxe des méthodes.

Le développeur doit alors traduire la description du protocole et garantir la conformité du code développé par rapport à la spécification. Une machine à états permet l'analyse des méthodes de HTTP, programmée à l'aide d'instruction d'aiguillage (*switch*). D'une part, une simple erreur peut alors rendre le support protocolaire défaillant. D'autre part, l'utilisation massive de ces instructions d'aiguillage ralenti l'exécution du support

```
Request=Request-Line/*(general-header/request-header/entity-header)
CRLF [message-body]
Request-Line=Method SP Request-URI SP HTTP-Version CRLF
Method = "OPTIONS"/"GET"/"HEAD"/"POST"/"PUT"/"DELETE"/"TRACE"/
extension-method
extension-method=token
```

FIGURE 2.8 – Extrait de l'ABNF du protocole HTTP

protocolaire puisque l'ensemble des cas possibles sont testés séquentiellement.

De plus, les protocoles sont sujets à des modifications et mises à jour tout au long de leur existence. Elles doivent alors être prise en compte dans le support protocolaire développé. Un tel code devient alors difficilement maintenable lorsque le protocole fournit de nombreuses extentions possible comme c'est le cas de l'en-tête `Via` du protocole SIP dont le support protocolaire dans SER comporte pas moins de 2000 lignes de code, près de 60 instructions d'aiguillage et plus de 500 cas possibles.

2.3.4.3 Discussion

La couche de support protocolaire représente une section critique d'une application réseau et consomme près du quart du temps total de traitement des messages [WSKW07]. Cette couche doit être la plus efficace possible et la logique applicative ne doit pas être handicapée par un support protocolaire bancal.

Ceci est d'autant plus vrai lorsque l'environnement dans lequel s'exécute ce support présente une quantité de ressources en mémoire et une puissance de calcul limitées. La fluidité des services rendus par l'application étant un autre critère décisif, le développeur doit en garantir les performances d'autant plus.

D'une part, la lourdeur des spécifications des protocoles ajoutées à la profonde connaissance des langages de programmation rendent le développement de la couche de support protocolaire fastidieux. Paradoxalement, l'application utilisant ce support protocolaire peut ne pas avoir besoin de l'ensemble des méthodes disponibles ou de la totalité des champs que propose le protocole, mais seulement d'une partie [BRLM11]. Un support protocolaire dédié à aux besoins spécifiques d'une application permettrait de garder uniquement les informations nécessaire à l'application, ce qui augmenterait les performances globales puisque l'ensemble des cas possibles ne seraient plus analysés. Néanmoins, un tel support ne serait de *prime abord* pas conforme au protocole. Bien qu'efficace, il serait également très peu évolutif puisqu'il serait couplé à une application spécifique. Par conséquent, le programmeur devrait mettre en oeuvre un support protocolaire pour chaque nouvelle application, pour un même protocole. Le gain de performances se ferait alors du détriment de l'ergonomie de développement.

D'autre part, historiquement, les approches aux problèmes de supports protoco-

laires et d'extraction de données en règle générale ont été mise en oeuvre de manière logicielle [KMKC09]. L'intégration croissante de dispositifs électroniques au sein d'un même appareil concerne l'ensemble des domaines courants, allant de la téléphonie à l'avionique en passant par le médical. Ainsi, les systèmes intégrés complexes sont constitués d'une partie matérielle mais également de logiciels. Au vu de l'émergence de nouvelles technologies ajoutées à l'augmentation du débit des réseaux ainsi que leur omniprésence dans notre quotidien, il apparaît clairement qu'il devient difficile de mettre en oeuvre le support pour l'ensemble des protocoles impliqués à la main et uniquement de manière logicielle. Les tâches répétitives pourraient être reléguées sur le plan matériel afin d'accroître l'efficacité globale du système. Il s'agit de **l'accélération matérielle**. Elle consiste à confier une fonction spécifique effectuée, traditionnellement, par un programme exécuté sur un processeur, à un circuit intégré dédié qui effectuera cette fonction de manière performante [SGL+11]. En effet, ces circuits spécifiques à une tâche sont plus efficaces qu'un processeur généraliste qui doit supporter l'ensemble des instructions d'une architecture donnée. On parle d'accélération matérielle car elle est obtenue par un câblage matériel des actions à effectuer. Toutefois, la mise en oeuvre de support protocolaire comprenant une partie matérielle rajouterait un domaine de compétences dont le développeur devrait disposer.

Nous nous efforçons, tout au long de nos travaux, de réduire ce fossé entre la taille et la compléxité croissante des spécifications des protocoles, les difficultés liées à l'utilisation des langages de programmation et la nécessité d'acquérir des compétences électroniques spécifiques dans le but de fournir un support protocolaire efficace, lié aux besoins spécifiques de l'application, tout en déchargeant le programmeur de l'ensemble des contraintes évoquées ci-dessus.

2.4 Bilan

Au cours de ce chapitre, nous avons présenté la manière dont évolue notre environnement quotidien à l'égard des systèmes informatiques. L'accès à l'information devient universel, quelque soit le dispositif informatique utilisé. Les systèmes embarqués sont omniprésents dans des domaines variés, allant de la domotique au médical en passant par l'automobile ou encore la logistique.

Les réseaux informatiques permettent d'interconnecter l'ensemble de ces entités. Ces entités dialoguent entre elles en s'aidant de protocoles réseaux dont les spécifications deviennent de plus en plus complexes. Qu'il s'agisse de protocoles de bas niveau, plutôt orientés réseaux ou de protocoles de haut niveau plutôt orientés applications, développer la couche de support pour ces protocoles est une tâche fastidieuse. Combler le fossé entre les spécifications d'un protocole, et la couche du support protocolaire correspondante s'avère être complexe et délicat.

De plus, l'efficacité du support protocolaire est cruciale pour garantir une efficacité optimale de l'application. Nous avons émis l'hypothèse que mettre en oeuvre un support protocolaire dédié aux exigences d'une application spécifique serait plus performant dans le contexte de systèmes embarqués ne fournissant pas une quantité confortable de ressources. Traditionnellement considéré comme un problème logiciel, la mise

en oeuvre de cette couche de support protocolaire tend à glisser vers un support matériel permettant un gain de performance adéquat grâce à l'accélération matérielle.

3

Traitement de messages protocolaires : État de l'art

LA COUCHE DE SUPPORT PROTOCOLAIRE est un point névralgique de la structure d'une application réseau. Il représente l'interface entre la logique applicative et le monde extérieur en lui fournissant une couche de manipulation des messages protocolaires. La programmation du support protocolaire de la couche applicative est à la charge du developpeur. Comme nous l'avons vu au chapitre précédent, le support doit etre à la fois efficace dans le contexte des systèmes ubiquitaires où les ressources sont limitées ; et maintenable, eu égard à l'évolution perpétuelle des protocoles réseaux. Le développeur doit ainsi concilier efficacité et ergonomie de développement.

Ce chapitre présente divers travaux relatifs à cette couche de support protocolaire. L'un des domaines candidats à cette quête d'ergonomie sont les langages dédiés. Après avoir décrit les avantages que représente leurs utilisations, nous mettons l'accent sur des solutions existantes dans ce domaine. D'autre part, les dispositifs mobiles embarquent de plus en plus fréquemment, en leur sein, des circuits programmables dont certaines solutions tirent parti, dans la quête d'efficacité du support protocolaire. La deuxième partie du chapitre leur est consacré.

Sommaire

3.1 Introduction

Le développement d'applications réseaux implique le développement de deux couches : la logique applicative fournissant les fonctionalités finales à l'utilisateur et le support protocolaire offrant une interface à la logique applicative pour la manipulation de messages protocolaires.

Une étape clef du développement de la couche de support protocolaire est l'extraction d'information suivant les spécifications des protocoles utilisés. Ce processus permet de transformer un flux d'information brut en données structurées et manipulables par la logique applicative. L'analyse de traffic Web [FKM+04], par exemple, implique l'extraction des entêtes du protocol HTTP afin d'obtenir des informations sur la taille des messages, le type de contenu qu'il véhicule ou encore des informations d'horodatage. Plus généralement, le traffic Email [JS04], les applications *peer-to-peer* [SGD+02] ou les communications basées sur le protocole SIP [PCRL07] sont autant de domaines où la nécessité d'avoir un support protocolaire efficace se fait ressentir. D'ailleurs, la couche de support protocolaire représente un élément prépondérant de nombreux outils de supervision du réseau comme *tcpdump* ou *Ethereal* [FK05], de détection et de prévention d'intrusions tels que *Snort* [Roe99] ou *Bro* [Som03] mais également de serveurs mandataires comme *Squid* [Sai11].

Étant donné une spécification d'un protocole, développer cette *couche de support* implique la traduction de cette spécification en un langage cible. En pratique, il est nécessaire de réduire le fossé entre la complexité des spécifications des protocoles et les langages utilisés pour le développement du support correspondant. Lorsque l'on considère, par exemple, le protocole HTTP [BLFF96], très largement répandu, le développeur doit prendre en compte la combinaison des requêtes HTTP (*pipelining*). Le protocole NetWare Core [CC97], utilisé dans le cadre de systèmes de fichiers distants, fournit près de 400 types de requêtes différentes, chacune ayant une syntaxe particulière. Cette lourdeur des spécifications rendent la tâche du développeur épineuse.

De plus, les raisons historiques ont favorisé la programmation en langages bas niveau tels que le langage C pour une garantie de performances. Comme nous l'avons vu à la section 2.3.4 du chapitre précédent, l'écart d'abstraction que doit combler le développeur entre la spécification des protocoles et le code à développer est conséquent, comme l'illustrent les figures 2.7 et 2.8.

La section suivante présente les langages dédiés, une alternative permettant de combler ce fossé en générant le support protocolaire à partir de spécifications de haut niveau. Cela permet d'avoir un support protocolaire conforme aux spécifications tout en déchargeant le développeur d'applications réseaux de la programmation manuelle de cette couche. Certaines approches existentes au développement du support protocolaire d'applications réseaux basées sur les langagées dédiées sont présentées à la section 3.3. La deuxième partie du chapitre est consacrée aux approches de développement du support protocolaire basées sur l'accélération matérielle. La section 3.4 définit les principes de l'accélération matérielle et enfin, la section 3.5 présente des approches de conception conjointe logiciel-matériel.

3.2 Les langages dédiés

3.2.1 Domaines d'applications

Décrire un système informatique est un processus complexe. Cette description est, le plus souvent, effectuée à l'aide de langages de programmation généralistes (ci-après nommés les ***G**eneral **P**urpose **L**anguages* ou GPL) couvrant un large spectre de problèmes. Ces GPLs ne coincident pas toujours avec les besoins des dévellopeurs dont les problématiques sont souvent cantonnées à un domaine en particulier [CE00]. Ceci sucite un intérêt croissant pour les langages dédiés ou langages métier (***D**omain **S**pecifi **L**anguages* ou DSL). En effet, ces DSLs ont pour caractéristique de mettre en exergue les concepts et idées sous-jacentes liés à un domaine en particulier [MHS05].

L'utilisation des DSLs est large et variée. Parmi quelques exemples, nous pouvons citer le langage de manipulation de données SQL (*Structured Query Language*) [CB74], le générateur d'analyseurs Lex et Yacc [LS90, Joh79], le métalangage BNF (*Backus-Naur-Form*) [CO97] que nous avons introduit à la section 2.3.2. Ils sont également répandus dans l'étude des systèmes d'exploitations : Bossa intègre un DSL permettant l'implémentation de politiques d'ordonnancements, Devil est un langage dédié à la spécification de programmation de pilotes de périphériques [Law02, MRC+00]. Toutefois, l'utilisation de langages dédiés ne se limite pas au domaine informatique. Nous les retrouvons en finance [vDF97], en biologie [HFS+03] ou encore en musique [LL90] et en conception matérielle [Ash08].

L'ensemble de ces langages ont la particularité d'être restreint à un ensemble de problématiques spécifiques à leur domaine respectif. Ils permettent, par opposition aux langages plus généralistes (GPL - ***G**eneral **P**urpose **L**anguages*) fournissant un large spectre

de méthodes générales, d'exprimer plus directement les besoins d'un domaine spécifique en permettant faire correspondre les concepts émanants du domaine avec ceux qui sont fournit par le langage. En plus de cette correspondance bénéfique pour l'expressivité du langage, les DSL présentent d'intéressantes perspectives en termes de maintenabilité, de configurabilité et de réusabilité [KT08].

3.2.2 Qu'est ce qu'un langage dédié ?

Comme nous l'avons vu précédemment, les DSL sont des langages cantonnés à leur domaine respectif et destinés à des développeurs qui n'ont pas toujours une expertise en termes de langages de programmation. En ce sens, le langage se doit d'utiliser un vocabulaire et une syntaxe qui se rapproche du domaine considéré. Comme le souligne Martin [Mar67] :

> *« We must develop languages that the scientist, the architect, the teacher, and the layman can use without being computer experts. The language for each user must be as natural as possible to her/him. The statistician must talk to his terminal in the language of statistics. The civil engineer must use the language of civil engineering. When a man learns his profession he must learn the problem-oriented languages to go with that profession. »*

Ainsi, un langage dédié doit être accessible aux experts du domaine même s'ils ne sont pas informaticiens [PADF+07]. D'après Réveillère [Ré01], un langage dédié est un langage de programmation restreint à un domaine particulier. Il fournit des abstractions appropriées à ce domaine. Suivant les domaines qui sont traités ou les utilisateurs qui sont visés, la littérature foisonne de définitions d'un langage dédié [Fow10, MHS05, LBCO04, vDKV00]. On les appelle tantôt *« Little Languages »* ou *« Micro Languages »*, tantôt *« Domain Specific Modeling Languages »* ou *« Domain Specific Visual Languages »*.

Un langagé dédié est caractérisé par un domaine d'application, une syntaxe et une sémantique particulière liée au domaine ainsi qu'un vocabulaire familier aux experts du domaine.

Un langage dédié est souvent de petite taille puisque seules les méthodes relatives aux problématiques du domaines y sont définies [Wil04]. En outre, un langage dédié offre un fort niveau d'abstraction. Les détails d'implémentation sont masqués à l'utilisateur ce qui permet d'améliorer la sûreté de développement. Ces caractéristiques représentent à la fois les forces et les faiblesses des langages dédiés.

3.2.3 Bilan

Les langages dédiés sont des langages de programmation restreint à un domaine métier spécifique. Ils offrent des solutions spécialement adaptées aux nécessités particulières du domaine métier considéré. Ils sont répandus dans de nombreux domaines comme la chimie, les produits financiers ou la conception de circuits éléctroniques. En proposant un niveau d'abstraction sur le domaine, ils sont une alternative prometteuse aux approches logicielles généralistes.

Ce niveau d'abstraction permet d'en faciliter la programmation à travers des notations se rapprochant à celles du domaine. Le domaine du support protocolaire des applications réseaux ne fait pas exception. De nombreuses approches basées sur les langages dédiés existent et permettent de réduire le fossé entre la spécification de la syntaxe du protocole, souvent exprimée en langage de haut niveau comme BNF [**?**] et la couche de support protocolaire souvent programmée en utilisant des langages de bas niveau tel que le langage C.

Ainsi, une approche basée sur les langages dédiés permet aux développeurs non-experts de la programmation réseau ou des langages de programmation bas niveau d'obtenir une couche de support protocolaire pour leurs applications en adéquation avec les spécifications des protocoles qu'ils utilisent.

3.3 Approches langage logicielle au développement de protocoles réseaux

3.3.1 Approches langage à la description de données

De manière idéale, toutes les données que l'on serait emmené à analyser serait présentée dans des formats standardisés pour lesquels de nombreux outils seraient disponibles. C'est, par exemple, le cas du format XML [BPSM+08] pour lequel on dispose de librairies d'analyse, programmes de visualisation, d'analyseurs syntaxiques, etc. En réalité, les données *ad hoc* proviennent de domaines très diverses et sous des formats tout aussi variés, comme l'illustre la figure 3.1.

Format : *Utilisation*	Représentation
Logs de serveur Web (CLF) : *Charge des serveurs web*	Fixed-column ASCII records
NetFlow : *Mesure de performances réseau*	Data-dependant number of fixed-width binary records
Gene Ontology *Corrélations entre gênes*	Variable-width ASCII records
CoMon data : *PlanetLab Machines*	ASCII records

FIGURE 3.1 – Quelques formats de données extraits de *PADS/ML : A Functional Data Description Language*

Il existe de multiples formats de données binaires ou textuels, semi-structurés, pour lesquels les outils d'analyse ne sont pas toujours disponibles. Le langage dédié PADS [FG05](*Processi*
Ad hoc Data Sources) ainsi que son évolution PADS/ML [MFW+07] sont des solutions génériques pour le traitement de ces données.

PADS est un langage déclaratif : une spécification PADS est composée d'une suite de déclarations de types décrivant les données ainsi que la manière de les analyser. D'une

part, les spécifications PADS sont assez explicites pour pouvoir servir de documentation au format de donnée traité, et d'autre part, assez flexible pour exprimer une multitude de format de donneés tels que l'ASCII, les données binaires ou des programmes Cobol. Etant donné une spécification PADS, le compilateur génère des librairies et outils paramétrables en langage C. Ces outils permettent de transformer, manipuler les données à traiter et incluent des fonctions de lecture, de transformations au format XML ou encore d'analyse statistiques.

PADS/ML est plus flexible en permettant de mettre en place des extensions au langage pour le traitement personnalisé des données. PADS/ML permet de définir des types récursifs et intègre du polymorphisme. Les utilisateurs peuvent ainsi définir des types de données avancés en combinant des types de base. Enfin, les outils d'analyse sont générés en langage O'Caml [oca] et non plus en C.

DataScript [Bac02] est un langage de description et de manipulation de données binaires. Il se compose d'un langage dédié déclaratif définissant des types pour la description du format physique des données et d'une interface de programmation manipulant les données. DataScript est non seulement indépendant vis à vis de toute architecture, mais il définit également des contraintes sémantiques sur les données et permet d'en vérifier la cohérence. Il est utilisé dans le cadre du chargement de classes Java dans une Machine Virtuelle Java [LY99].

Toutefois, ces approches ne se révèlent pas adaptées au le développement du support protocolaire d'applications réseaux.

D'une part, l'ergonomie n'est pas au rendez-vous. En effet, les protocoles sont normalisés suivant une syntaxe appellée ABNF. Cette syntaxe est largement répandue dans les spécifications de protocoles. Les langages décrits ci-dessus en sont très éloignés. Par exemple, dans le cas de PADS, il faut traduite une description en ABNF à l'origine en une description en langage PADS avant la génération du support protocolaire. D'autre part, ces approches restent peu efficaces. En effet, elles ont été conçues pour le traitement de données et non pas pour le traitement massif de messages protocolaires, plus nombreux et de plus petite taille.

On peut alors s'orienter vers des approches prenant en compte ces considérations, spécifiquement adaptées aux protocoles réseaux et aux contraintes qu'ils véhiculent.

3.3.2 Approche langage au développement de supports protocolaire

Chandra *et al.* proposent PacketTypes [CM, MC00], destiné au traitement de messages binaires bruts. Ils partent du postulat que la disposition des champs définissant un packet ainsi que les contraintes qui leur sont associées peuvent être considérés comme un *type*. Il s'agit d'un langage dédié à la spécification de format de paquets. Une description PacketTypes est une suite de structures *champ-valeur* ressemblant à des structures en C. Un compilateur génère une interface de programmation établissant la concordance entre les packets reçus et leurs types, évitant par conséquent l'écriture manuelle de ce code bas niveau. Toutefois, le code généré est de l'ordre de 40% moins performant que celui développé manuellement rendant cette approche caduque.

Lambright *et al.* [LDD+96] sont à l'origine d'un langage similaire, appellé APF. A l'instar de PacketTypes, APF définit des contraintes liées aux types définis. Bien que la cible originelle d'APF réside dans la spécification de format de packets binaires, le langage s'avère suffisamment expressif pour la spécification de protocoles textuels.

Un des domaines gourmands en analyse protocolaire est celui de la sécurité des systèmes d'information et en particulier, la sécurité des réseaux. Un système de détection d'instrusion ou *IDS* est un mécanisme destiné à repérer un ensemble d'activités suspectes sur un réseau ou un hôte et de remonter à l'administrateur les tentatives d'instrusions en comparant ces activités présumées malicieuses à des signatures prédéfinies. Le traitement des données qui circulent à travers le réseau devient un critère primordial pour garantir la détection la plus efficace possible car la couche de support protocolaire est en première ligne. Snort [Roe99] et Hogwash [Hai02] sont de tels systèmes de détection d'intrusions, ce dernier utilisant le moteur de détection du premier. Afin d'acccroître la diversité des protocoles supportés, ils utilisent une architecture modulaire à base de *plugins*, écrits dans un langage généraliste bas niveau tel que le langage C. Bien que la vitesse d'exécution ou la compatibilité avec des systèmes existants représentent autant d'avantage d'utilisations de ces langages, développer et maintenir manuellement ces composants est coûteux et sujet à des erreurs de programmation récurentes pouvant entrainer des failles de sécurité sur le système [QPP02]. Fort de ce constat, des approches génératives ont émergé afin de rendre le développement du support protocolaire plus sûr.

Pang *et al.* [PPSP06] suggèrent Binpac, un langage dédié au développement d'analyseurs syntaxiques de protocoles réseaux. Proche de Yacc [Joh79], il génère un support pour les protocoles applicatifs textuels comme binaires en C++. De plus, il fournit une couche de gestion des connexions, les séquences de messages incomplets et la détection d'erreurs. Comme PacketTypes, une spécification Binpac est une suite de description de types. Par conséquent, il se heurte aux même problèmes que ceux soulevés par l'utilisation de PacketTypes. Le premier d'entre eux concerne le jonglage avec le format des spécifications. L'information contenue dans les spécifications originelles des protocoles doit être traduite en langage Binpac pour être traitée. De plus, l'utilisation de types est limitée pour certaines caractéristiques des protocoles qui doivent alors être exprimées par l'écriture d'expression régulières. Le deuxième manque provient de la réutilisabilité des analyseurs syntaxiques. En effet, l'implémentation de Binpac est fortement ancré à celle du système de détection d'instrusions Bro [Som03] rendant l'intégration des analyseurs syntaxiques dans d'autres applications réseaux plus complexe.

Borisov *et al.* [BBW07] de *Microsoft Research* créent une implémentation spécifiquement ciblé aux systèmes de détection d'intrusions. Elle comprend un support d'exécution, nommé *Generic Application-level Protocol Analyser* ou GAPA ainsi qu'un langage dédié, *Generic Application-level Protocol Analyser Language* ou GAPAL. A l'instar de Binpac, GAPAL est destiné à la génération d'analyseurs syntaxiques applicatifs. Toutefois, les auteurs de GAPAL mettent en avant l'expressivité de GAPAL, permettant de construire une pile entière de protocoles, incluant IP, TCP ainsi que des protocoles applicatifs. De plus, l'utilisation de la *métasyntaxe* ABNF dans les spécifications GAPAL permet un développement rapide d'analyseurs syntaxiques. Neanmoins, l'inconvénient majeur de

l'utilisation de GAPAL à large échelle provient de l'infrastructure globale que forme le système GAPA + GAPAL. En effet, celui-ci est un langage interprété et nécessite donc celui-là comme support d'exécution. Cette forte association ne s'avère pas propice à l'utilisation de GAPAL dans un cadre différent, comme le notre.

Madhavapeddy *et al.* [MHD+07] proposent de créer un « Internet fonctionnel »à travers le projet Melange, basé sur le langage OCaml. Ils partent du postulat que les protocoles critiques de l'Internet sont écrits dans des langages de typage faible comme C ou C++ qui sont régulièrement la cible de problèmes de vulnérabilités et de fiabilité. Les langages à fort typage sont certes plus sûrs mais souffrent de pertes de performances. Les auteurs proposent une approche basée sur du typage statique fort afin d'éliminer les vérifications de types au moment de l'exécution et de la méta-programmation générative pour abstraire le code bas niveau nécessaire pour le traitement des messages. A cette fin, les auteurs introduisent le langage dédié Meta-Programming Language (MPL) destiné à la spécification de protocoles réseaux de bas niveau comme IPv4 ou TCP, mais également des protocoles applicatifs plus complexes comme SSH ou DNS. Le compilateur produit du code OCaml performant, plus flexible et sans erreurs de typage. Les auteurs évaluent leur approche en mesurant et comparant l'efficacité de serveurs SSH et DNS basé sur Mélange avec leur équivalents en C. Leur approche révèle un débit plus performant ainsi qu'une latence moindre.

Burgy *et al.* [BRLM07] sont à l'origine de Zebu, une approche langage pour l'amélioration de la robustesse des implémentations de protocoles réseaux. Elle comprend Zebu, un langage débié pour la spécification de protocoles réseaux mettant en avant une logique applicative particulière, respectant les nécessités de support d'une application spécifique. Il repose sur la *métasyntaxe* ABNF nécessitant un effort de programmation moindre tout en fournissant des abstractions permettant de vérifier la cohérence des protocoles. Cette approche comprend également un compilateur, ZebuYacc, générant une couche de support protocolaire en C ainsi qu'une chaine de compilation μ-Zebu fournissant un support pour le développement d'applications réseaux complètes basées sur Zebu. Intégrant des optimisations de consommations mémoire, cette approche s'avère adaptée à la cible des systèmes embarqués. Dans le cas de SER, un serveur SIP largement répandu, le support protocolaire pour SIP généré via Zebu détecte 100% des messages erronés tandis que l'implémentation du support dans SER n'en détecte que 25%.

Adrian Thurston introduit l'outil Ragel [Thu06] permettant de compiler des machines à états finis. Il permet de décrire des expressions régulières mais également d'implémenter des protocoles robustes et d'analyser des formats de données. Le langage Ragel est proche de l'ABNF et, à ce titre, se rapproche des nécessités des programmeurs d'applications devant implémenter des couches de supports protocolaires. Il permet de générer une implémentation de l'automate décrivant le protocole dans des langages tels que C, C++ ou encore Java. Toutefois, les actions utilisateurs doivent être codées par le développeur de l'application.

Dans cette section, nous avons présenté des travaux relatifs au développement de supports protocolaire. Qu'il s'agisse d'applications orientées *données* ou d'applications orientées *contrôl*, les langages dédiés sont au coeur de ces approches et permettent de

générer des couches de support protocolaire efficace tout en réduisant l'effort de programmation nécessaire en se reposant sur des spécifications de haut-niveau, loin des problématiques inhérentes aux langages de programmation bas niveau comme la programmation réseau. Le point commun de l'ensemble de ces approches réside dans la génération de code pûrement logiciel, qui, une fois compilé, est exécuté sur un microprocesseur généraliste. Toutefois, l'apparition et l'adoption des *System On Chip*(SoC), ou systèmes sur puce offrent de nouvelles perspectives d'approches en terme de support protocolaire, en exploitant les possibilités offertes par l'accélération matérielle, basée sur des circuits programmables dont sont de plus en plus dotés les SoC.

3.4 Architectures reconfigurables

3.4.1 Présentation

L'histoire des circuits programmables commence dans les années 1980, même si le principe remonte au début des années 1960. Il a été proposé par G. Estrin [Est02], les premiers prototypes n'apparaitront qu'une vingtaine d'années plus tard.

La première famille de circuits programmables était de type PAL (*Programmable Array Logic*) et se programmaient comme des mémoires de type ROM. Ils étaient utilisés pour implémenter des fonctions simples comme des contrôleur de bus.Les progrès dans le domaine micro-électronique ont vu l'apparition de nouvelles familles de circuits programmables comme les CPLD (*Complex Logic Programmable Device*) et les FPGA (*Field Programmable Gate Array*), commercialisés par la société Xilinx en 1985.

Les FPGA sont de plus en plus intégrés sur des Systèmes sur Puce. Ces systèmes miniaturisés disposent de microprocesseurs, de mémoire, de bus de communication permettant de relier les différents composants, des contrôleurs vidéos, des contrôleurs d'entrée/sortie, etc. Cette miniaturisation a permi l'avènement d'appareils portables de plus en plus ancrés dans notre quotidien, comme les appareils de téléphonie mobile.

Ces progrès technologiques peuvent être mesurés à travers les lois de Gordon Moore, l'un des trois fondateurs d'Intel, relative à la compexité du matériel informatique ; et de Shannon relative à la complexité algorithmique, commme l'illustre la figure 3.2.

L'architecture utilisée des microprocesseurs utilisée de nos jours n'a quasiment pas évolué depuis son apparition en 1945. Les microprocesseurs modernes, issu du modèle Von Neuman, que nous avons vu au chapitre 2.2, sont plus enclins à l'exécution séquentielle d'une tâche, même s'ils sont capables d'extraire le parallélisme au niveau des instructions à travers des pipelines et autres unités fonctionnelles. Pour améliorer les performances, la taille des transistors doit être diminuée de manière à pouvoir augmenter la fréquence de fonctionnement ce qui implique *de facto* une consommation d'énergie plus grande. Comme le montre la figure 3.2, c'est un inconvénient rédhibitoire pour l'adoption de processeurs à des fréquences de plus en plus hautes, l'évolution de la capacité des batteries se faisant netemment moins rapidement.

Les microprocesseurs ne sont pas pour autant en voie de disparition. Ils sont associés à d'autres composants effectuant des tâches spécifiques pour avoir un maximum de performances. Un microprocesseur peut par exemple être couplé à un coprocesseur

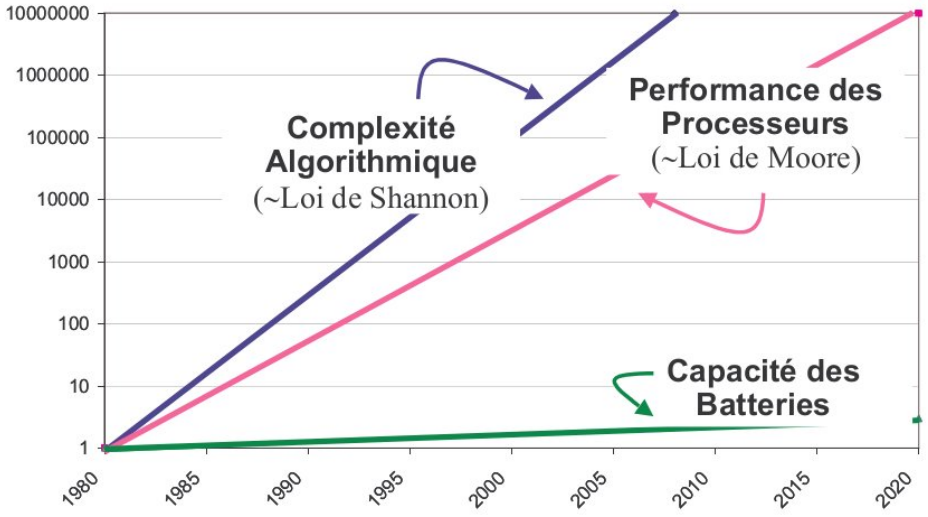


FIGURE 3.2 – Evolution des systèmes sur puces. Source : [Rab00]

cryptographique qui aura la charge du calcul éponyme. De nombreuses approches ont été mises au point pour l'étude du couplage de circuits reconfigurables avec un microprocesseur. Cette architecture vise principalement à décharger le microprocesseur pour des traitements qui ne lui sont pas adaptés et à les confier à la ressource reconfigurable. On peut citer l'équipe BRASS (*Berkeley Reconfigurable Architectures, Systems and Software*) et leur projet GARP [Hau00].

Lorsque les circuits sont figés et fondus dans le silicium, ils sont appellés cirtuis ASIC (*Application Specific Integrated Circuit*). Ils présentent un modèle d'exécution spatial répartissant le traitement sur une panoplie d'unités fonctionnelles spécifiques au traitement. L'avantage de ces circuits spécialisés est la performance qu'ils atteignent tout en ayant une consommation d'énergie moindre. Il alors possible de combiner un microprocesseur dont la puissance est limitée à un circuit ASIC dédié à un calcul en particulier (comme un ensemble de calcul cryptographique ou des calculs dédiés aux traitements vidéos). Néanmois, ces systèmes ne sont efficaces qu'au détriment de leur évolutivité : pour un traitement donné, l'architecture est figée, les données sont traitées suivant un schéma pré-établi et il n'est pas possible de le modifier une fois fondu. Ceci est très contraignant lorsqu'il s'agit de mettre en place des approches au support protocolaire, puisque la moindre évolution des protocoles (changement de versions, améliorations des standards, mise à jour) implique une refonte complète des circuits. Ce manque de flexibilité est un inconvénient majeur dans notre contexte.

Ainsi, les microprocesseurs généralistes sont très flexibles, mais offrent des performances limitées tandis qu'il s'agit de l'inverse dans le cas des ASIC, dédiés à une tâche spécifique. Comme l'illustre la figure 3.3, il s'agit de trouver le compromis correct dans un cas particulier entre l'efficacité du système et son évolutivité. Les architectures reconfigurables apportent ce compromis en alliant performances et flexibilité.

Les FPGA (*Field Programmable Gate Array*) sont de tels circuits. Ils offrent la possibilité de programmer (et reprogrammer) des tâches spécifiques. Ils sont pour cette raison très prisés pour le prototypage de circuits. La console de jeux vidéos Sony Playstation Por-

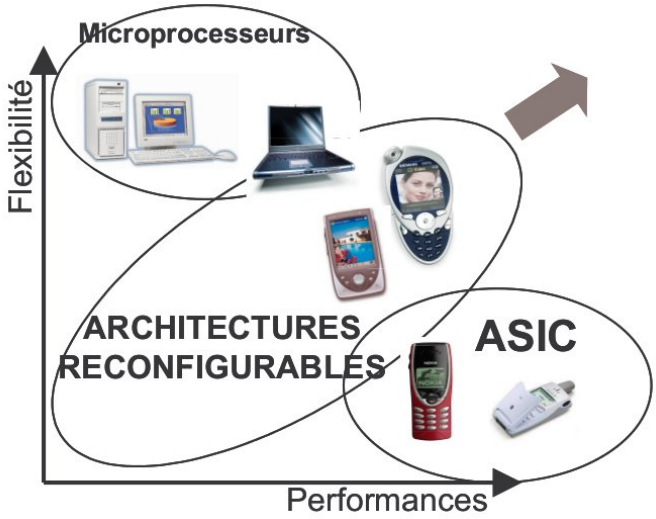


FIGURE 3.3 – Flexibilité et performance des circuits

table et sa machine virtuelle Virtual Mobile Engine [Cor] est un exemple de composant exploitant les fonctionalités des FPGA relatives l'accélération matérielle à la demande.

Les sections suivantes répondent aux interrogations suivantes relatives à l'utilisation des circuits programmables :

Qu'est ce qu'un FPGA ?

Comment programme t-on un FPGA ?

Quels sont les avantages et inconvénients de leur utilisation ?

3.4.2 Qu'est ce qu'un FPGA ?

L'acronyme FPGA désigne un circuit intégré composé d'un réseau de cellules logique programmables. Ce concept a toutefois évolué vers plus d'autonomie et on parle plus de *calcul reconfigurable* (ou *reconfigurable computing*) lorsque l'on fait référence à ces architectures. Ceux sont des composants reconfigurables qui permettent, après les avoir programmé, de réaliser des calculs et fonctions logiques. Leur structure régulière permet de réaliser de nombreaux traitements bas niveau réguliers. Il se compose essentiellement de trois types de ressources (cf. Fig 3.4) :

— Les ressources de traitement qui incluent les mémoires, les registres ou bascules, la logique.
— Les ressources d'interconnexions programmables permettant de relier les différents blocs logiques entre eux.
— Les ressources d'entrée/sortie permettant d'interfacer le FPGA avec le monde extérieur.

❶ Ressources de traitement

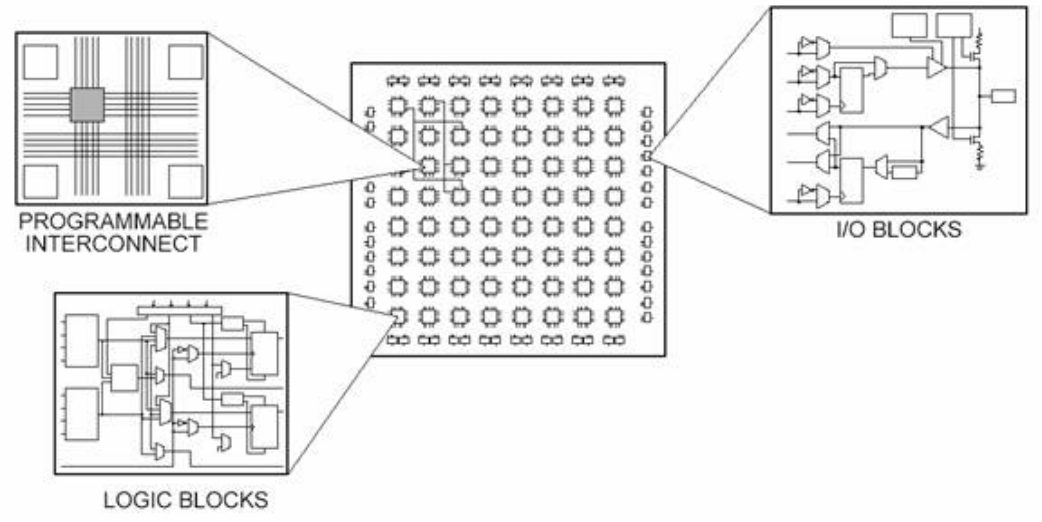


FIGURE 3.4 – Eléments composants un FPGA - *http ://www.ni.com*

Un FPGA se compose de blocs logiques configurables, les CLBs (Configurable Logic Block) qui en forment le coeur. Il s'agit de cellules d'éléments logiques programmables constitués de registres, des multiplexeurs, des portes logiques et des LUTs (Look-Up Table). Ces cellules peuvent être reliées par un bus d'interconnexion afin d'effectuer des fonctions logiques.

Les FPGA intègrent de plus en plus de cellules sur une même surface. Les cartes Virtex 7 de Xilinx [Inc] sont composées de près de deux millions de CLBs.

❷ Ressources d'interconnexions

Les CLBs réalisent des fonctions logiques simples. Pour pouvoir combiner plusieurs CLBs et réaliser des fonctions beaucoup plus complexe, il nest nécessaire de pouvoir relier ces différentes cellules. Les ressources d'interconnexion s'en chargent. Ce processus s'appelle le *routage* et il en existe plusieurs types. Le routage *classique* permet de relier les CLBs soit directement, soit par l'intermédiaire de matrices de connexion permettant d'aiguiller les signaux. Toutefois, le délai d'interconnexion évolue linéairement avec la distance qui sépare les CLBs, ce qui peut être pénalisant sur des FPGA très denses. Il est alors possible de s'orienter vers d'autres types de routage comme le routage *segmenté* [BR99].

❸ Ressources d'entrée/sortie

Les ressources d'entrée/sortie sont composées de blocs d'entrée/sortie, les IOBs (*Input Output Block*). Ces cellules permettent d'interfacer le FPGA avec le monde extérieur.

A ces ressources viennent s'ajouter d'autres composants tels que la mémoire permettant de fournir un espace de stockage sans avoir à recourir à des mémoires externes, les multiplieurs cablés permettant d'accroitre les performances, les coeurs de processeurs

programmables au sein des FPGA (Power PC pour la famille Xilinx Virtex 5 et ARM pour la famille Xilinx Virtex 7).

3.4.3 Comment programme t-on un FPGA ?

Programmer un FPGA peut se faire de différentes façons. A l'origine, les circuits implantés sur les FPGA étaient décrits transistor par transistor, rendant la conception très lente. Il n'est plus possible d'utiliser de telles méthodes pour la conception de circuits de plusieurs milliers de portes logiques.

Les outils récents de spécification de circuits sont basés sur des langages de description matérielle tel que VHDL ou VERILOG [Ash08]. Ces langages offrent des caractéristiques proches des langages de programmation comme C permettant une description plus naturel de l'architecture. Cette description s'opère au niveau du transfert entre les registres (RTL) des FPGA et décrit les transferts de signaux entre ces registres, indépendamment d'une technologie particulière. Un outil de design électronique (EDA) se charge de transformer la description abstraite du circuit en un ensemble de primitives et une description de portes logiques pouvant être implantés sur le FPGA. Cette description se nomme une *netlist*. Toutefois, les netlists ne font que décrire les instances des différents composants et non pas une topologie concrète. Cette tâche est effectuée par un outil de *place-and-route* qui crée une instance du circuit déployable sur le FPGA.

Les données servant à configurer le FPGA sont associées dans un *bitstream*. Celui-ci est chargé sur le FPGA via les ressources d'entrée/sortie disponibles que nous avons décrites auparavant. Ce processus a pour effet de peupler les blocs logiques et d'effectuer le routage entre les différents éléments. Le temps de reconfiguration est proportionnel à la taille du *bitstream* et, dans le cas d'une reconfiguration totale de l'ensemble du FPGA, l'ensemble des calculs et processus s'exécutant sur celui-ci sont interrompus. Afin de permettre une reconfiguration dynamique, donc sans interrompre le FPGA, certaines cartes disposent de région reconfigurables à la volée par un *bitstream* partiel. Ces régions reconfirables dynamiquement sont appellées des *PRRs*.

Dans les systèmes exploitant les circuits reconfigurables, la ressource reconfigurable est souvent un composant parmi d'autres. Les architectures couplent en général les ressources reconfigurables à un microprocesseur généraliste. Le recours à un processeur permet de lui céder les tâches généralistes tout en gardant les tâches spécifiques aux ressources reconfigurables. Les composants critiques de l'application, ou ceux nécessitant le plus de performances sont exécutés sur le support reconfigurable alors que le processeur généraliste se charge de l'exécution des composants non critiques, développés sous forme logicielle exclusivement. Il s'agit de conception conjointe matériel-logiciel ou *codesign*.

3.4.4 Avantages et inconvénients

Les circuits programmables représentent un compromis entre des approches exclusivement logicielles, flexibles, mais peu performantes et des solutions à base de circuits dédié et figé une fois fondu, très performant mais non évolutive. Nous répondons au

« Pourquoi » à travers une comparaison des circuits programmables avec les microprocesseur généralistes et les circuits ASIC sur les critères courants : le temps de développement et les performances.

❷ Temps de développement

Une logique reprogrammable possède un avantage indéniable en efforts de développement par rapport à un circuit fondu. Une fois la spécification écrite, le déploiement sur une ressource programmable ainsi que la validation se font relativement rapidement. Ce n'est pas du tout le cas pour un circuit fondu dédié où les étapes de vérification et de conception peuvent prendre plusieurs semaines à cause de la fabrication des composants. Ainsi, le temps de développement sur un circuit configurable est très inférieur à celui d'un ASIC.

Un microprocesseur est doté d'une capacité de reconfiguration limitée à une liste d'opérations arithmétiques et logiques et par un choix également limité de transferts de données entre unités fonctionnelles. Une fois la configuration définie, elle peut être mise en place à chaque cycle. Typiquement, il s'agit du jeux d'instructions du processeur. A l'opposé, les circuits programmables ne sont pas restreints à ces opérations. Toutefois, la liberté qu'offrent ces circuits dans le choix de l'ensemble des éléments (opérateurs, interconnexions, transferts de données, etc) à un prix, celui de la description de cette structure puisqu'il faut tout spécifier. Le temps de développement est ainsi beaucoup plus avantageux en faveur des microprocesseur généraliste. En effet, une solution logicielle a un cycle de conception relativement court, tandis qu'il faut beaucoup plus d'efforts pour la conception d'une même application sur un circuit programmable. Néanmoins, des solutions ont été mises en oeuvre pour réduire cet écart. Liu *et al.* [LCMC08] proposent une approche de compilation de langages se rapprochant du langage C en circuits matériels déployable sur des ressources reconfigurables.

❸ Performances

La fréquence de fonctionnement d'un circuit ASIC est généralement plus élevée que celle d'un circuit programmable. La raison en est la nature configurable et programmable de ces circuits [KR06]. Ceci a un impact évident sur les performances des deux types de circuits : l'ASIC se montrera plus efficace que le FPGA pour un traitement équivalent. Cet écart de performances est néanmoins fluctuant suivant le type de traitement et l'architecture des circuits considérés et peu être réduit à travers le choix d'architectures reconfigurables améliorées (multiplieurs, types de mémoires). Certaines solutions reconfigurables optimisées parviennent même à être plus performantes comme c'est le cas du chiffrement DES sur FPGA proposé par Patterson [Cam01].

Dans le cas des microprocesseurs généralistes, comme nous l'avons vu, les solutions logicielles ne sont pas comparables à des solutions matérielles pour un même traitement. Les fréquences de fonctionnement dopent encore une fois les performances, celles des microprocesseurs généralistes étant beaucoup plus élevée que celles des circuits programmables (plusieurs giga-octets par seconde pour un processeur moderne contre quelques centaines de méga-octets par seconde pour les cartes FPGA). Toute-

fois, l'exploitation du parallélisme qu'offrent ces circuits permet de contre balancer les fréquences de fonctionnement moindres mais également d'en réduire la consommation d'énergie.

La figure 3.3 met en exergue le positionnement intermédiaire de la logique programmable. Entre solutions strictement matérielles dédiées et solutions exclusivement logicielles, les circuits programmables allient flexibilité, évolutivité par rapports aux ASIC et performances par rapport aux processeurs.

Coupler un processeur à une ressource configurable devient de plus en plus fréquent sur les systèmes sur puce modernes, léguant les tâches spécifiques aux composants programmables, tout en laissant les tâches plus élémentaires au processeur. Fort de ce constat, de nouvelles approches de traitement de messages protocolaires ont émmergé, alliant des composants logiciels à des éléments matériel configurables.

3.5 Conception conjointe logiciel-matériel au développement de supports protocolaires

La couche de support protocolaire pour les application réseaux est traditionnellement développée manuellement. Afin d'éviter la panoplie de problèmes posés par une telle conception (maintenance, bugs, fossé entre code et spécification), de nouvelles approches basées sur les langages dédiés ont émergées comme le décrit la section 3.3. Ces approches rendent le développement de support protocolaire plus sûr, naturel et efficace. Toutefois, ces solutions s'avèrent innadaptées au contexte des systèmes embarqués fortement contraints qui ne disposent pas toujours des ressources nécessaires (mémoire disponible, capacité de stockage, énergie) à leur exécution. Néanmoins, ces systèmes embarquent, en plus des composants classiques (processeur, mémoire, interfaces d'entrée/sortie), des circuits configurables qu'il est possible de programmer à la volée afin de leur atitrer une tâche spécifique. Cette section dresse un état de l'art des approches hybrides combinant solutions logicielles et composant matériels pour le développement de supports protocolaires.

3.5.1 Accélération matérielle au traitement d'expressions régulières

La traitement d'expressions régulières est un domaine de recherche très actif ces dernières années.

Sidhu et Prasanna décrivent une approche à la recherche de motifs basée sur des FPGA [SP01]. Etant donné un texte d'entrée, il s'agit de trouver l'ensemble des chaines de caractères correspondantes à une expression donnée. Cette approche est fondée sur la construction d'automates finis non-deterministes. Un automate fini non-deterministe ou NFA (Non deterministic Finite Automaton) est un automate tel que, dans un état donné, il peut y avoir plusieurs transitions possibles avec la même entrée. Cette solution n'est pas envisageable sans la construction d'un automate fini déterministe ou DFA (Deterministic Finite Automaton) pour une exécution sur un processeur qui traite les instructions de manière séquentielle. Toutefois, cette solution s'avère adaptée pour les

circuits prograblammables en exploitant le parallélisme qu'ils offrent. La construction du NFA se fait à l'exécution, lorsque l'utilisateur fournit l'expression régulière. Le temps de construction de l'automate, linéaire en la taille de l'expression régulière, doit donc être minimisé. Les auteurs valident leur approche en comparant son efficacité avec celle de la commande Unix `grep`.

Hutchings *et.al* [HFC02] utilisent cette approche dans la détection de motifs dans les signatures du système de détection d'intrusions Snort.

Dans le sillage des approches basées sur les automates finis, Moscola *et al.* présentent une solution de filtrage d'un pare-feu basée sur des FPGA [MLLP03]. Ils proposent de traiter les expressions régulières en construisant cette fois des automates finis déterministes. L'approche de Sidhu est fondée sur des NFAs à cause de l'espace faible et du temps plus court nécessaire à la construction des automates. Moscola s'intéresse plutôt à la taille finale de l'automate, une fois construit. De plus, les DFAs sont plus adaptés à une recherche de motifs basées sur des contextes multiples provenant de flux d'entrée différents. Dans un tel environnement, il est plus avantageux de n'avoir qu'un seul état actif à la fois en cas de contexte à sauvegarder ou restaurer.

Toujours dans le contexte des systèmes de détection d'intrusions, Clark *et al.* mettent en oeuvre des optimisations sur la construction de NFAs en vue d'améliorer les performances des systèmes, une fois programmés sur les FPGA. L'idée directrice est de réduire le nombre de comparaisons pour la recherche de motifs. Cette réduction se fait par l'intermédiaire de diverses optimisations matérielles relatives à l'implémentation des NFAs sur des FPGA. Une autre amélioration vient de la casse des caractères. Dans le cas d'un motif non sensible à la casse, deux comparaisons doivent être effectuées, l'une pour les minuscules, l'autre pour les majuscules, ce qui n'est pas optimal. A l'inverse, un gain de performance est obtenu en utilisant le fait que le code ASCII d'un caractère ne diffère que par un *bit* entre minuscule et majuscule. L'ensemble de ces optimisations permettent à cette approche de rechercher des motifs relatifs à l'ensemble des règles définies par un système de détection d'instructions comme Snort.

Bien que l'ensemble de ces solutions soient efficaces pour l'analyse d'expressions régulières, elles présentent deux inconvénients majeurs. Tout d'abord, ces solutions n'offrent qu'une approche d'analyse lexicale. En effet, l'extraction d'informations et leur traitement sémantique n'est pas abordé. Ensuite, ces approches sont basées sur une conception exclusivement matérielle. Cette méthode de conception implique d'avoir des connaissances avancées dans un domaine d'expertise qu'est la micro-électronique. De plus, l'exploitation des données par une application logicielle implique un interfaçage avec un processeur généraliste, le développement de ces interfaces représentant autant d'obstacles.

3.5.2 Accélération matérielle au développement d'analyseurs syntaxiques

Mitra *et al.* développent une architecture de filtrage de contenu XML basée sur des FPGA [MVB+09]. Le besoin de filtrage de contenu XML provient notamment de la taille sans cesse croissante des données échangées sur des systèmes de type *publish/-*

subscribe [CRW01]. Sur de tels systèmes, des informations sont *publiées*, typiquement sous forme de flux XML. Elles sont alors filtrées et des utilisateurs peuvent y *souscrire*. Ces systèmes sont caractérisés par leur débit d'entrée élevé ce qui nécessite des performances de filtrage élevées. Mitra propose de recourrir aux FPGA pour effectuer ce filtrage. Dans le cas d'une approche logicielle, le filtrage est effectué de manière séquentielle et les *tags* sont évalués les uns après les autres. Avec les capacités de parallélisme que l'on peut tirer par le cablage des circuits FPGA, ceux-ci s'avèrent plus adaptés à ce type de structure. Les informations contenues dans les requête XPATH sont converties en expressions régulières et, une fois programmés sur le FPGA, comparées aux tags XML des documents disponibles. Les débits pour certains types de donneés sont de l'ordre d'une centaine de fois supérieurs avec cette approche qu'avec une solution exclusivement logicielle telle que YFilter [DAF+03].

Moscola *et al.* [MLLP03] synthétisent un module de routage basé sur le contenu de messages. Cette approche fondée sur les circuits programmables comprend un analyseur syntaxique ainsi qu'un module de routage permettant de modifier les en-têtes IP des paquets suivant le résultat de l'analyse de leur contenu. Au lieu de router le paquet suivant des chaines de caractères apparaissant à l'intérieur, la totalité du paquet est analysé, une grammaire spécifie les paquets devant changer de destination. Un module de routage basé sur XML illustre l'approche. La grammaire XML est traduite dans une grammaire de type Lex/Yacc avant d'être transformée en code VHDL. La description comprend un analyseur lexical, un analyseur syntaxique et un module de routage de paquets. Les expérimentations révèlent un débit de près de 6Go/s avec une fréquence de fonctionnement de 200MHz sur une carte Xilinx Virtex XCV200E. Toutefois, ces débits ont été mesurés sans infrastructure de traitements protocolaires comme des accès réseaux, une pile de protocoles réseaux, etc.

ProGram est une approche générative de spécification et de validation de protocoles réseaux proposée par Öberg *et al.* [OKH00]. ProGram est un langage de spécification de protocoles, il utilise une syntaxe se rapprochant de BNF. Un compilateur traite une spécification ProGram et génère du code VHDL décrivant le protocole. Ce code est synthétisé et déployé sur un circuit programmable. Cette approche présente l'avantage d'abstraire les fonctionalités du protocole à spécifier.

Les analyseurs syntaxiques présentés dans ces approches présentent l'avantage d'être performants comparés à des approches logicielles équivalentes. Toutefois, la plupart du temps, ces analyseurs syntaxiques sont exclusivement matériels nécessitant, tout comme les approches de traitements d'expressions régulières, d'importantes compétences dans le domaine électronique. De plus, les expérimentations réalisées pour évaluer l'efficacité de ces analyseurs sont souvent atomiques : seules la performance de l'analyseur en lui même est mesurée, faute d'infrastructure suffisante pour les intégrer dans de réelles applications réseaux, notamment avec des logiques applicatives. Enfin, elles tendent à creuser le fossé entre les spécifications des protocoles réseaux et la synthèse matérielle de ceux ci en introduisant une surcouche de description matérielle, rendant le processus de développement des analyseurs syntaxiques encore plus complexe.

3.5.3 Outils de synthèse de haut-niveau au développement d'analyseurs syntaxiques

La synthèse de haut niveau ou HLS (*High Level Synthesis*) produit automatiquement une description d'une architecture réalisant le comportement d'un système numérique, spécifié dans un langage de plus haut niveau. Des algorithmes décrits dans des langages de haut niveau sont automatiquement transformés en design matériel, via un langage de description comme le langage VHDL.

LegUp [CCA+11] est un outil de synthèse de haut niveau destiné aux accélérateurs et processeurs matériels. Les auteurs partent du constat que l'écriture de code de description matériel est coûteux et sujet à de multiples erreurs puisqu'il faut gérer l'ensemble du processus de spécification matériel. A l'opposé, une approche logicielle est généralement plus naturelle et de nombreux outils existent pour la validation et la vérification d'une solution logicielle, au détriment des performances offertes par les circuits programmables. LegUp est un outil de synthèse de haut niveau qui prend en entrée un programme C standard et produit une architecture conjointe logiciel/matériel déployable sur un système sur puce disposant de ressources programmables. Les éléments logiciels générés sont exécutés sur un processeur généraliste de type MIPS tandis que d'autres composants sont directement synthétisés en circuits programmable appellés accélérateurs matériels. LegUp est *open source* et distribué avec un ensemble de programmes tests en C que l'utilisateur peut compiler en vue d'une exécution exclusivement logicielle, exclusivement matérielle ou une combinaison des deux [HTHT09]. Il serait donc possible de synthétiser une couche de support protocolaire générée à l'aide d'une approche logicielle basée sur les langages dédiées en accélérateur matériel. Toutefois, un tel procédé rajouterait une surcouche au processus de conception. De plus, l'ensemble d'un programme en C n'est pas toujours synthétisable en une implémentation matérielle. Les tâches séquentielles, comme le parcours d'une liste chaînée, sont plus adaptées à un processeur tandis que d'autres le sont plus sur circuits. Enfin, cette approche est généraliste, elle ne répond pas aux besoins spécifiques des applications réseaux en terme de support protocolaire.

Par ailleurs, LegUp effectue une analyse de profils afin de repérer les parties candidates à l'accélération matérielle. D'un protocole à un autre, le code exécuté sur circuits et celui exécuté sur le processeur peut ne pas être le même ce qui peut être contraignant dans le cas d'une même logique applicative.

Rinta-aho *et al.* [RAKD12] suggèrent Click2NetFPGA, un ensemble d'outils permettant, à partir d'un système logiciel complet d'en générer une implémentation matérielle. Les auteurs illustrent leur approche en implémentant une suite d'outils de développement de routeurs logiciels appellés *Click modular router* [KMC+00] sur une plateforme matérielle de développement de routeurs, la Stanford NetFPGA [LMW+07]. Comme son nom l'indique, il s'agit d'une suite d'outils déjà existante, transformant le code du routeur logiciel Click en implémentation matérielle qu'il est possible de déployer sur une plateforme matérielle dédiée aux développements de composants réseaux. Toutefois, cette approche est innadaptée au développement de support protocolaires pour

trois raisons. Tout d'abord, la chaine d'outils souffre de limitations car seul les systèmes logiciels d'un domaine en particulier, en locurence celui des routeurs logiciels, peuvent être traduits en implémentation matérielle, même si la faisabilité de l'approche a été démontrée. Ces limitations empêchent une généralisation rapide au développement de couches de supports protocolaires. Ensuite, même si ces limitations été levées, il s'agit d'une implémentation exclusivement matérielle et ne permet que très peu de flexibilité quant à son intégration dans une application réseau réelle. Enfin, dans la panoplie d'outils dont dispose le développeur pour mener à bien son processus de conception, l'utilisation de cette chaine d'outils supplémentaires serait coûteuse en terme de temps d'apprentissage et d'efforts déployés.

3.6 Synthèse

Cette section reprend l'ensemble des projets et solutions présentés dans ce chapitre. Les tableaux 3.5 et 3.6 présentent les solutions logicielles au développement du support protocolaire d'applications réseaux. Dans un premier temps sont résumées les principales approches logicielles pour le développement de couches de support protocolaires. Dans un second temps sont décrites les solutions matérielles et enfin, dans un troisième temps, nous évoquons quelques projets de synthèse de haut niveau.

Ces approches se révèlent non-adaptées au développement de supports protocolaires pour les applications réseaux. En effet, certaines sont orientées traitement de données, d'autres sont fortement couplées à une application ou une infrastructure en particulier, d'autres encore sont exclusivement matérielles et impliquent que le développeur monte en expertise sur la programmation électronique.

Approches logicielles	
Description de données	
PADS	- PADS/PADS-ML : Solution générique pour le traitement des données. - Langage déclaratif explicite proche du langage C. - Génération d'outils de traitement, manipulation et transformation des données en C (PADS) et OCaml (PADS-ML).
DataScript	- Langage de manipulation de données binaires. - Langage déclaratif définissant des types pour la description des données. - Dispose d'une interface de programmation pour la manipulation des données. - Définition de contraintes sémantiques sur les données. - Utilisé dans le cadre du chargement de classes Java.
Bilan	
	- Approches destinées au traitement de gros volume de donées. - Approches très éloignées de la syntaxe ABNF. - Non-adaptées au développement de supports protocolaires.
Support protocolaire	
PacketTypes	- Langage dédié à la spécification du format de paquets binaires. - Spécification sous forme de suite de structures *champ-valeur*. - Code généré plus lent de 40% qu'un code équivalent développé manuellement.
Binpac	- Langage dédié au développement d'analyseurs syntaxiques de protocoles réseaux. - Intégré dans le détecteur d'instrusions Bro. - Spécification sous forme de description de types. - Implémentation fortement couplée à celle de Bro.
GAPAL	- Langage dédié du projet GAPA pour la générateur d'analyseurs syntaxiques de protocoles applicatifs. - Permet de construire une pile de protocoles incluant IP, TCP et protocoles applicatifs. - Langage GAPAL fortement couplé aux outils fournis par l'infrastructure GAPA.
Melange	- Projet de création d'un Internet plus fonctionnel, basé sur l'utilisation de méthodes formelles. - MPL : langage pour la spécification de protocoles réseaux bas-niveau (IP, TCP) et applicatifs (DNS, SSH). - Efficacité meilleure que leurs équivalents en C.
Zebu	- Langage dédié à l'implémentation de support protocolaire basé sur les besoins spécifiques des applications réseaux. - Proche de la syntaxe ABNF. - Génération du support protocolaire ainsi que d'un support pour le développement des applications basées dessus. - Evaluation sur le serveur SIP SER.
Ragel	- Outil de compilation de machines à états finis. - Proche de la syntaxe ABNF. - Génération du support protocolaire dans des langages tels que C, C++ ou Java. - Fonctions de rappels devant être implémentées par le développeur de l'application.
Bilan	
	- Code généré plus lent qu'un code équivalement développé à la main (PacketTypes). - Langages fortement couplés à une application particulière (Binpac). - Nécessité d'une infrastructure lourde (GAPA). - Aucune approche n'exploite les avantages offerts par les ressources reconfigurables.

FIGURE 3.5 – Synthèse des approches logicielles au développement de supports protocolaires

Approches matérielles	
Traitement d'expressions régulières	
Sidhu *et. al*	- Recherche de motifs basés sur des FPGA. - Approche fondée sur la constructions d'automates finis non-déterministes (NFA). - Evaluation faite sur la comparaison avec la commande UNIX `grep`. - Approche utilisée pour la détection de motifs dans Snort.
Moscola *et. al*	- Solution de filtrage de pare-feu réseau basé sur des FPGA. - Approchée fondée sur la construction d'automates finis déterministes (DFA). - Un unique état actif à la fois, contrairement aux NFAs. - Approche adaptée à un environnement à contexte multiples.
Clarck *et. al*	- Amélioration de l'implémentation des NFAs sur FPGAS. - Réduction du nombre de comparaisons pour la recherche de motifs. - Evaluation basée sur la recherche de motifs dans Snort.
Bilan	
	- Approches visant une analyse lexicale uniquement, extraction d'informations et traitement sémantique non-abordé. - Approches exclusivement matérielles. - Expertise nécessaire en développement sur FPGA.
Développement d'analyseurs syntaxiques	
Mitra *et. al*	- Architecture de filtrage de contenu XML basée sur des FPGAs. - Destiné aux systèmes *pub/sub* à forts débits : Besoin de fortes performances. - Performance de l'approche de 100fois supérieure à une approche purement logicielle équivalente.
Moscola *et. al*	- Module de routage en fonction du contenu des messages basé sur des FPGAs. - Traitement des en-têtes IP des paquets en fonction de leur contenu. - Débit allant jusqu'à $6Go/s$ à une fréquence de $200MHz$ sur une carte Xilinx Virtex XCV200E.
ProGram	- Approche génération pour la spécification et la validation de protocoles réseaux. - Syntaxe se rapprochant de l'ABNF. - Génération de code en VHDL déployage sur des cartes FPGAs après synthétisation. - Abstraction des fonctionalités à spécifier.
Bilan	
	- Approches souvent exclusivement matérielles. - Nécessité d'importantes connaissances en développement matériel sur FPGAs. - Manque d'infrastructure et d'outils pour l'intégration avec de réélles applications réseaux.
Synthèse de haut niveau	
LegUP	- Outils de HLS destiné aux accélérateurs et processeurs matériels. - Prend en entrée un programme C et génère une architecture conjointe logiciel-matériel déployage sur FPGA. - Eléments logiciels exécutés sur un CPU généraliste, éléments matériels synthétisés pour l'exécution sur FPGA.
Click2Net	- Suite d'outils permettant de générer une implémentation matérielle d'un système logiciel complet. - Illustration sur une suite d'outils de développement de routeurs *Click Modular Router*. - Génération d'une implémentation matérielle des routeurs exécutables sur une plateforme matérielle comme la Standford NetFPGA.
Bilan	
	- Limitations des outils de HLS : ensemble d'un programme logiciel en C pas toujours synthétisable en description matérielle. - Approche généraliste ne tenant pas compte des besoins spécifiques des protocoles réseaux. - Implémentation exclusivement matérielle (Click2Net) offrant peu de flexibilité.

FIGURE 3.6 – Synthèse des approches matérielles au développement de supports protocolaires

3.7 Bilan

Dans la première section de ce chapitre, nous avons dressé un pannel de quelques solutions permettant de générer une couche de support protocolaires pour les applications réseaux. Ces approches sont basées sur des langages dédiés grâce auxquels il est possible de se restreindre à un domaine métier particulier en adoptant son vocabulaire et sa syntaxe. La syntaxe de ces langages dédiés se rapproche de celle des spécifications décrivant le comportement des protocoles. Ces approches se révèlent toutefois peu adaptées à des systèmes fortement contraints qui ne disposent pas toujours des ressources nécessaires à leur exécution.

Toutefois, ces systèmes possèdent en leur sein, en plus des composants classiques tels que le processeur ou la mémoire, des composants matériels dédiés effectuant des tâches spécifiques, tels que des contrôleurs vidéos pour le traitement de l'image, ou encore des modules cryptographiques pour le chiffrement de données. D'une part, ces éléments dédiés accomplissent leur tâche de manière plus efficace que le processeur et d'autre part, ils permettent de décharger celui-ci. Néanmoins, ces éléments sont le plus souvent gravés dans le sillicium et ne peuvent plus être modifiés, une fois fondu en usine. Afin de remédier à cette limitation, ces systèmes sur puce (SoC) embarquent également des circuits reconfigurables qu'il est possible de programmer et reprogrammer au besoin.

Des solutions combinant logiciel et matériel ont emmergés pour le développement de supports protocolaires. Toutefois, ces solutions sont généralistes, visent des applications de traitements de données massives plutôt que des applications réseaux où les messages à traiter sont de tailles beaucoup plus faibles à un débit beaucoup plus élevé. En outre, l'adoption de ces solutions implique le plus souvent l'acquisition de nouvelles compétences spécifiques, comme le design matériel. Enfin, ces solutions ne réduisent pas l'écart d'abstraction entre la spécification des protocoles réseaux et leur implémentation, certaines d'entre elles allant même jusqu'à creuser cet écart.

4

Démarche suivie

LE CHAPITRE 2 nous a montré les bouleversements auxquelles nous assistons depuis les dernières années en terme d'accès à l'information. Le modèle *client - serveur* historique tend à s'effacer au profit d'un modèle plus ubiquitaire où les utilisateurs accèdent à l'information désirée à n'importe quel moment et de n'importe où. Parrallèlement, les équipements informatiques évoluent et embarquent des ressources et une capacité de calcul limitée. Ils disposent également de ressources matérielles reconfigurables qu'il est possible de programmer à la volée pour une tâche dédiée, comme décrit à la section 3.4 du chapitre 3.

L'accès à l'information se fait par l'intérmédiaire d'applications réseaux qui communiquent entre elles suivant un ensemble consensuel de règles appellées *protocole*. La couche logicielle qui permet de supporter ces protocoles réseaux représente l'interface entre l'application réseau et le monde extérieur. Elle est en charge de l'analyse des messages entrants et de la construction des messages sortant. A ce titre, cette couche se doit d'être la plus efficace possible eu égard au peu de ressources disponibles.

De nombreuses approches au développement du support protocolaire d'applications réseaux existent et permettent de guider le développeur vers un support protocolaire performant, comme nous l'avons vu au chapitre 3. Ces approches sont basées sur des langages dédiés qui ont la particularité d'être restreint à un domaine métier particulier et permettent ainsi de se focaliser sur le développement de cette couche, comme l'a montré l'approche Zebu [BRLM11].

A travers un exemple concret, ce chapitre caractérise les problématiques auxquelles les développeurs font face lorsqu'il s'agit de mettre au point un support protocolaire dans un environnement contraint en ressources. L'ensemble de ces problématiques nous guident vers l'élaboration d'une approche langage accélérée matériellement dédiée au développement du support protocolaire d'applications réseaux.

L'approche que nous proposons suit le cheminement suivant :

Cas d'étude - Cette section décrit un exemple d'application simple nécessitant un support protocolaire. Ce cas d'étude met en exergue les difficultés intrinsèques au

```
LOGIN = TheGreatUser  PASSWORD = @$_my_password_$@ END
```

FIGURE 4.1 – Couple utilisateur et mot de passe

développement d'un tel support vis à vis des spécifications des protocoles, mais également des supports d'exécution cibles ainsi que les ressources qu'ils embarquent.

Problématiques - Cette section regroupe les principales problématiques auxquelles les développeurs d'applications réseaux font face lors de la programmation de couches de support protocolaire.

Démarche globale - Cette section présente l'approche que nous proposons. Dans un premier temps, nous effectuons une analyse de domaine afin de nous guider vers l'élaboration du langage dédié Zebra. Dans un second temps, nous décrivons la plateforme de prototypage que nous avons mise au point avant de présenter le langage dédié Zebra. Enfin, nous décrivons l'ensemble du flux d'exécution, de l'écriture de la spécification du protocole en langage Zebra au déploiement des éléments générés sur la plateforme d'exécution.

Sommaire

4.1 Cas d'étude : le couple nom d'utilisateur - mot de passe

Afin de mieux cerner l'impact du traitement des messages sur les performances globales de l'application réseau, cette section propose la description d'un cas d'étude : une application basée sur l'acquisition d'un nom d'utilisateur et du mot de passe associé.

Considérons le message textuel présenté à la figure 4.1. Il contient le nom d'utilisateur ainsi que le mot de passe associé. Lorsque l'application traitant l'authentification des utilisateurs reçoit ce message, il traverse la couche de support protocolaire afin d'y extraire les différentes informations utiles à l'application, à savoir le nom d'utilisateur `TheGreatUser` ainsi que le mot de passe `@$_my_password_$@`, comme le décrit la figure 4.2.

Cette extraction se déroule en plusieurs étapes, suivant la granularité de la couche de support du protocole correspondante. Elle est notamment en charge de vérifier la

```
struct authentication_block {
  char username[64];
  /* Contient 'TheGreatUser', après traitement */
  char password|64];
  /* Contient '@$_my_password_$@', après traitement */
}
```

FIGURE 4.2 – Structure de données présentée à la logique applicative

```
%%{
   machine foo parser;
   SP = ' ';
   main := 'USER' SP 'LOGIN' '=' alnum+ ^login SP 'PASS' '=' alnum+
      ^pass SP 'END' ^^;
}%%
```

FIGURE 4.3 – Spécification Zebu pour le protocole nom d'utilisateur/mot de passe

cohérence syntaxique et sémantique du message reçu. Par exemple, le nom d'utilisateur peut contenir des caractères minuscules ou majuscules mais il ne peut pas contenir d'espace ou de caractères spéciaux. Ensuite, le message en lui même doit commencer par la séquence `USER` et se terminer par la séquence `END`, le contenu utile étant localisé entre ces deux séquences. Enfin, l'ordre des informations est validé : le nom d'utilisateur doit être fourni avant le mot de passe.

Cet exemple basique nous montre la complexité d'écriture de la couche de support protocolaire correspondante. En effet, le développeur doit vérifier son code traite effectivement l'ensemble des étapes mentionnées ci-dessus. Lors de l'évolution du protocole, le développeur doit modifier son code en conséquence, par exemple lors de l'autorisation des espaces dans le nom d'utilisateur ou encore l'ajout de la date de naissance dans le processus d'authentification.

Les spécifications sont basées sur le langage Ragel [Thu06] afin de générer une couche de support pour le traitement du couple nom d'utilisateur/mot de passe. La figure 4.3 montre un exemple de spécification correspondant à notre protocole.

Cette spécification est proche de celle du protocole, elle reprend les mêmes notations et expressions et permet ainsi au développeur de s'affranchir de sa traduction dans un langage de programmation de plus bas niveau tel que le langage C. Ceci réduit

```
/* Création de la structure de données */
authentication_block dataSet;
/* Lecture des données sur l'interface réseau */
int nb data = read(socket, buffer, BUFFER SIZE);
/* Traitement du flux de données */
parser_stream(dataSet, buffer, nb data);
/* Affichage des informations */
printf("connection from %s [%s]\n",
                get_user(dataSet),
                get_passwd(dataSet));
```

FIGURE 4.4 – Application finale utilisant les fonctions générées par Zebu

les risques d'erreurs, diminue la complexité de la tâche et offre un gain de temps au développeur.

Ainsi, à partir de cette spécification, les outils fournis avec Zebu génèrent automatiquement les composants suivants :

- Une implémentation de l'automate de traitement protocolaire.
- La structure de données permettant de stocker l'information extraite.
- Un ensemble de fonctions permettant d'accéder et manipuler (remplissage, lecture) de cette structure de données.

Ces fonctions permettent au développeur de l'application d'intégrer, de manière transparente, la couche de support générée à leur logique applicative. Un exemple d'une telle intégration est présentée dans la figure 4.4.

Dans [BRLM07], ces couches sont implantées uniquement à base de logiciel. L'exécution des couches de support représentent près de 25% du temps global d'exécution de l'application réseau. Ainsi, afin d'accroitre la performance de ces couches de support - et les performances globales de l'application -, la plateforme Zebra utilisée dans ces travaux permet l'intégration d'accélérateurs matériels. Ce système conjoint repose sur un système sur micro-puce (SoC) complet intégré au sein d'un circuit de type FPGA. De manière plus globale,la section suivante s'attarde sur les problématiques rencontrées lors du développement de couches de support plus complexes.

4.2 Problématique

La figure 4.5 montre le code analysant les méthodes du protocole HTTP extrait du serveur web Apache. Le support protocolaire des applications réseaux est le plus souvent formé de ce type de code, verbeux, écrit en langage bas niveau tel que le langage C, sujet à erreurs. Ce code présente plusieurs types de contraintes :

- Ce code est difficilement lisible. En effet, il n'est pas possible, sans une lecture approndie, de connaître l'ensemble des méthodes supportées.
- Il est difficilement maintenable : lors d'une mise à jour du protocole HTTP, ce code devra être revu et remanié. La difficulté de lecture accroit ce caractère de non-maintenabilité.
- L'intégration de ce code dans une application réseau est délicat puisqu'elle peut ne pas avoir besoin de l'ensemble des méthodes citées. L'exécution de ce support consommera donc des ressources de manière inutile.
- Enfin, ce code est très éloigné de la spécification du protocole HTTP, tel que l'illustre la figure 4.6. Il incombe alors au développeur de combler ce fossé en étudiant méticuleusement les spécifications du protocole d'une part et d'autre part en ayant l'expertise nécessaire au développement du code correspondant dans un langage de bas niveau.

```
[...]
switch (len)
{
 case 3:
   switch (method[0])
   {
     case 'P':
       return (method[1] == 'U'
          && method[2] == 'T'
          ? M_PUT : UNKNOWN_METHOD);
      case 'G':
        return (method[1] == 'E'
          && method[2] == 'T'
          ? M_GET : UNKNOWN_METHOD);
      default:
        return UNKNOWN_METHOD;
   }

 case 4:
   switch (method[0])
   {
     case 'H':
       return (method[1] == 'E'
         && method[2] == 'A'
         && method[3] == 'D'
         ? M_GET : UNKNOWN_METHOD);
     case 'P':
       return (method[1] == 'O'
         && method[2] == 'S'
         && method[3] == 'T'
         ? M_POST : UNKNOWN_METHOD);
     [...]
 }
```

```
[...]
register_one_method(p,"GET",M_GET);
register_one_method(p,"PUT",M_PUT);
register_one_method(p,"POST",M_POST);
[...]
```

FIGURE 4.5 – Extrait du code analysant les méthodes HTTP dans Apache

```
Request=Request-Line/*(general-header/request-header/entity-header)
CRLF [message-body]
Request-Line=Method SP Request-URI SP HTTP-Version CRLF
Method = "OPTIONS"/"GET"/"HEAD"/"POST"/"PUT"/"DELETE"/"TRACE"/
extension-method
extension-method=token
```

FIGURE 4.6 – Extrait de l'ABNF du protocole HTTP

De plus, cette couche logicielle est le dernier rempart traversé par les messages protocolaires avant leur présentation à l'application réseau. De ce fait, elle forme l'interface entre l'application et le monde extérieur.

Afin de réduire ce fossé entre la spécification des protocoles réseaux en ABNF et le code de la couche de support protocolaire correspondante, de nombreuses approches basées sur les langages dédiées ont emmergées. Ces approches répondent à différentes contraintes soulevées par le développement du support protocolaire dans un environnement contraint où les équipements informatiques disposent de peu de ressources de calcul ou de mémoire, mais fournissent également des ressources matérielles reconfigurables permettant d'accélérer un traitement spécifique.

Plusieurs constatations motivent l'étude des problèmes posés par le développement conjoint logiciel-matériel du support protocolaire d'applications réseaux :

- L'efficacité du support protocolaire conditionne les performances de toute l'application réseau. Par exemple, une étude a décelé que la seule analyse syntaxique de messages du protocole SIP représente près de 25% du temps de traitement global d'un message dans le cas du serveur mandataire SER considéré comme la référence dans le domaine [CE02, CEE04]. Une autre étude a révélé que le volume de code d'un support protocolaire représente, dans les cas extrêmes, près d'un quart du volume de code global d'une application [BRLM07], ce qui en fait un composant fondamental de toute application réseau.
- Les protocoles réseaux sont complexes. Comme nous l'avons vu au chapitre 2, les spécifications des protocoles sont décrites dans un langage présentant un haut niveau d'abstraction, la *métasyntaxe* ABNF. De plus, l'ABNF n'est pas toujours claire quant à la structure des messages, certaines contraintes étant exprimées dans un langage *naturel*. La complexité des spécifications protocolaires conjuguées à la complexité du code à écrire pour le support de ces protocoles creuse d'autant plus le fossé existant.
- De nombreux travaux existent pour combler ce fossé, basés sur les langages dédiés. Toutefois, ces travaux se concentrent principalement à la génération d'un support protocolaire uniquement logiciel et ne tirent pas parti des possibilités de reconfigurations matérielles offertes au sein des systèmes embarqués. Ainsi, l'approche Zebu [BRLM11] semble être le plus avancé en terme de robustesse et de performances mais les composants qu'elle génère sont purement logiciels ce qui

rend son adoption dans des milieux où l'accélération matérielle est omniprésente délicate.

- D'autres approches tirant parti des possibilités d'accélération matérielles ont été présentés à la section 3.5 du chapitre 3. Ces travaux sont basés sur une couche de suport protocolaire entièrement matérielle ou sur une conception conjointe logiciel-matériel où les tâches critiques sont accélérées matériellements tandis que les traitements les moins critiques s'exécutent sur des processeurs généralistes. Malheureusement, ces approches ne réduisent en aucun cas le fossé existant entre les spécifications des protocoles et le code du support protocolaire. Bien au contraire, l'adoption de ces approches implique un savoir-faire et une expertise poussés dans le domaine de l'électronique et des langages de descriptions matériels, comme le langage VHDL [Ash08] ou Verilog, en plus des langages généralistes de bas niveau tel que la programmation réseau en langage C.
- Enfin, des travaux visant à générer automatiquement une description matérielle à partir de langage de programmation tel que le langage C peuvent être utilisés. C'est par exemple le cas de l'approche LegUp [CCA+11]. Toutefois, ces approches sont généralistes et ne permettent pas d'adresser les contraintes spécifiques des applications réseaux.

De part ces problématiques, il apparait qu'une approche capable de soulager le programmeur dans l'écriture d'un support protocolaire efficace est d'un réel interêt. Nous présentons maintenant la démarche que nous avons suivie pour y parvenir.

4.3 Démarche globale

L'approche que nous proposons découle des observations précédentes. Tout d'abord, l'exécution de la couche de support représente près de 25% du temps d'exécution global d'une application. L'efficacité de l'application dépend est directement liée à celle de la couche de support. Ensuite, pendant l'exécution de la couche de support, il apparait que près de 90% du temps d'exécution est consommé par le traitement des automates décrivant le protocole. Le temps d'exécution des fonctions de manipulations des messages (remplissage de vue après la détection des différents chammps) devient alors négligeable.

Notre approche propose également de centrer les efforts autour du développeur. En effet, nous avons mis en exergue la complexité du processus de programmation de couches de support. Conjugué à la diversité et la prolifération des protocoles réseaux, il apparait que l'ergonomie de développement de telles couches de support est tout aussi primordiale que l'efficacité de celles-ci.

Nous proposons de palier à la première contrainte en mettant l'accent sur l'utilisation de ressources reconfigurables dont disposent les supports d'exécution. En effet, le traitement des automates décrivant le protocole est un processus répétitif qu'il est possible de dédier à une unité d'exécution spécifique : un accélérateur matériel. Un processeur généraliste se charge de l'exécuction des fonctions plus généralistes comme la manipulation des messages ainsi que de la logique applicative.

Une solution à la seconde contrainte serait l'utilisation d'un langage dédié dont la

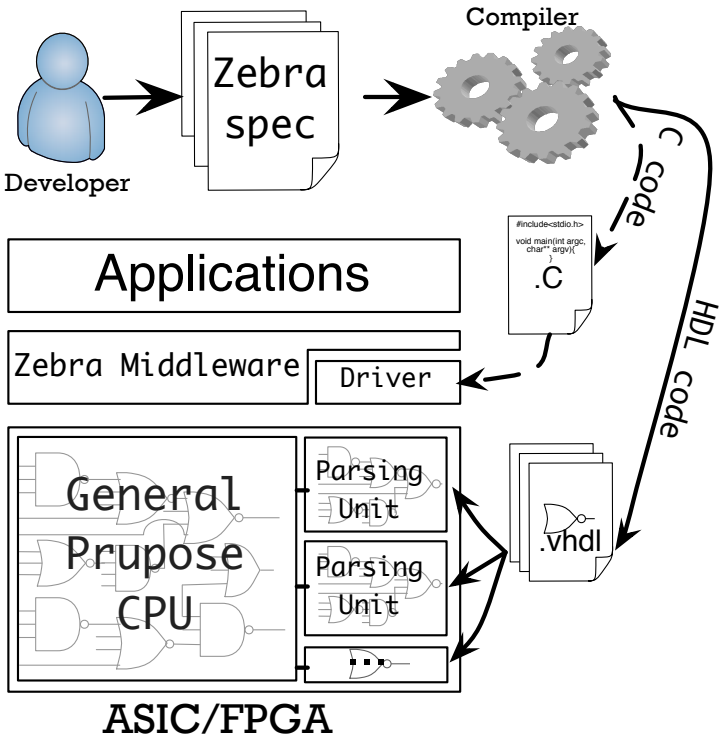


FIGURE 4.7 – Processus de développement du support protocolaire accéléré matériellement

syntaxe se rapprocherait le plus possible à la syntaxe des spécifications BNF permettant de décrire les protocoles réseaux. L'effort fourni par le développeur serait alors moindre, l'ensemble des composants nécéssaires étant automatiquement générés.

Afin de réunir un processeur généraliste ainsi que des ressources configurables sur un même support physique, nous avons choisi de bâtir une plateforme de prototypage sur une carte de type FPGA. De telles cartes regroupent des ressources qu'il est possible de reconfigurer suivant les besoins, des interfaces d'entrée/sortie et une interface réseau. Une partie de ces ressources sont réservées pour un processeur généraliste tandis qu'une autre partie sert à l'accélération du traitement des automates. Avoir l'ensemble des composants sur une même carte rend le prototypage moins complexe.

La figure 5.1 illustre le processus de développement que nous définissons. Dans un premier temps, une spécification du protocole à supporter est écrite par le développeur à partir de la spécification formelle du protocule. Cette spécification est ensuite analysée et traitée afin de détecter les erreurs de syntaxe. Le compilateur permet alors de générer les accélérateurs matériels correspondant aux protocoles ainsi que le code logiciel permettant de les piloter. L'approche fournit un intergiciel permettant de manipuler les différents accélérateurs matériels de manière transparente pour la logique applicative. Le programmeur d'une application réseau peut utiliser les fonctions fournies par l'interface de programmation de cet intergiciel pour accéder et manipuler les accélérateurs matériels *via* le code logiciel généré par la spécification.

Les étapes suivies pour définir ce processus de développement et que nous dé-

taillons par la suite dans cette thèse sont les suivantes.

La première étape consiste à concevoir un langage dédié à la spécification de la couche de support protocolaire intégrant des accélérateurs matériels pour une application réseau. Pour ce faire, nous avons identifié les notations et abstractions appropriées à ce domaine métier spécifique. Ces notations permettent à des programmeurs familiers du domaine de développement de supports protocolaires d'appréhender notre langage sans qu'ils aient besoin d'être des programmeurs chevronnés.

La deuxième étape concerne la phase de génération des accélérateurs matériels et le code logiciel nécessaire pour les piloter. Différents types d'accélérateurs matériels peuvent être générés suivant les performances et l'ergonomie de reconfigurabilité dynamique souhaitée. La quatrième étape consiste à développer un intergiciel permettant la manipulation et l'analyse des messages protocolaires de manière transparente pour le développeur d'applications réseaux.

Enfin, la dernière étape consiste à évaluer l'efficacité du support protocolaire généré. Nous nous intéressons à deux critères primordiaux : la consommation mémoire et les performances à l'exécution. Les expérimentations réalisées nous permettent de mesurer l'impact de l'approche sur le développement de support protocolaire conjoint logiciel-matériel pour les applications réseaux.

4.4 Approche proposée

La suite du manuscrit est articulée autour de l'approche Zebra que nous proposons.

Le chapitre 5 présente la plateforme de prototypage que nous avons conçue. Elle est constituée d'une carte de type FPGA muni de ressources reconfigurables ainsi que de périphériques d'entrées/sorties, de mémoire vive et d'une interface réseau. Elle inclut ainsi l'ensemble des composants nécessaires pour l'éxécution des applicatifs réseaux.

Le chapitre 6 détaille l'architecture logicielle Zebra. Il explique la manière dont s'articulent l'ensemble des composants impliqués dans l'approche Zebra, du développeur écrivant la spécification du protocole en langage Zebra à la génération des composants logiciels et matériels ainsi que leur déploiement sur la plateforme d'exécution.

Le chapitre 7 regroupe les expérimentations et évaluations que nous avons menées pour étayer notre approche.

5

Architecture matérielle Zebra

L'OBJECTIF de ce chapitre est de décrire l'architecture matérielle du système ciblé. Cette plateforme d'exécution et d'expérimentations a été développée pour valider expérimentalement l'ensemble des hypothèses que nous avons mentionnées précédemment.

Cette architecture hétérogène, composée d'un processeur et d'un ensemble d'accélérateurs, est intégrée au sein d'un circuit de type FPGA. Elle permet la conception de systèmes complexes. Ces systèmes, combinant éléments logiciels et matériels, sont aujourd'hui couramment employés dans les domaines des les systèmes embarqués à destination des applications hautes performances aussi bien que pour les applications destinées au grand public. Cependant ce type de systèmes combinant parties logicielles et matérielles nécessite, dans notre cas, le développement et le prototypage d'un support d'exécution de la couche de support protocolaire adapté. En effet, les tâches répétitives effectuées par la couche de support protocolaire sont déportées sur des coprocesseurs tandis que les tâches liées à l'exécution de l'application et à la gestion du réseau sont laissées au processeur généraliste.

Ce système conjoint intégré dans un FPGA concentre différents domaines d'expertise qui sont difficiles à maitriser pour un développeur d'applications réseaux. De cette contrainte d'ergonomie est né un besoin d'automatisation de certaines tâches de conception et d'utilisation de la plateforme. Cette nécessité est adressée par la génération automatique et la configuration des composants pour une exploitation efficace et naturelle de la plateforme.

Sommaire

5.1 Introduction

Comme nous l'avons évoqué précédemment, une application réseau est décomposée sous la forme de deux couches principales : une couche applicative et une couche de support protocolaire. La couche de support protocolaire offre une interface de manipulation des messages à la logique applicative en abstrayant la gestion du réseau et le protocole d'échange de l'information à la logique de haut niveau. En effet, cette dernière nécessite uniquement l'accès aux données pertinentes contenues dans les messages protocolaires, peut importe la manière dont les données sont véhiculées.

Compte tenu de la faible puissance de calcul disponible dans bon nombre de systèmes embarqués, nous présentons la manière dont un programmeur intègre la couche de support protocolaire au développement de son application réseau. La figure 5.1 présente ce processus. Dans un second temps, une application de *login* basique illustre les contraintes qui pèsent sur le développeur.

Dans le chapitre 3, nous avons présenté des méthodologies de développement de la couche de support protocolaire pour des applications réseaux. Ces méthodologies ciblent des implantations purement logicielles. Lors du développement de l'application, l'intégration de la couche de support protocolaire peut se faire à travers deux méthodes principales, comme l'illustrent la figure 5.1.

La première méthodologie, présentée en figure 5.1, consiste pour le concepteur à développer ou à réutiliser la couche de support protocolaire nécessaire à son application. Cette solution est la plus naturelle,même si de *prime abord*, écrire une couche de support protocolaire à la main peut paraître légitime, cette solution n'est pas satisfaisante à long terme. En effet, comme nous l'avons vu, les protocoles sont décrits dans des spécifications de haut niveau proches du langage humain. Le niveau d'abstraction entre la syntaxe de la spécification et la syntaxe des langages de bas niveau tel que le langage C rend le développement manuel d'une telle couche fastidieux. De plus, la solution ainsi obtenue est difficillement évolutive et/ou maintenable. Enfin, une telle couche est *de facto* proche des nécessités spécifiques de l'application et donc d'une réutilisabilité amoindrie. A terme, il faudrait développer une couche de support protocolaire pour chaque nouvelle application. Une solution alternative consiste alors à utiliser des couches de support déjà existantes et reconnues. Cette solution permet un gain de temps considérable. Le développeur a uniquement besoin d'intégrer et d'adapter une couche de support logiciel à son application. Toutefois, cette solution, bien que rapide à mettre en oeuvre, n'est guère adaptée à un déploiement sur des systèmes embarqués. En effet, dans la majorité des cas, ces couches de support protocolaire supportent l'intégralité du protocole tandis que les besoins réels de l'applicatif sont moins importants. Le support complet des protocoles mis en oeuvre se traduit par une diminution des performances ainsi que par une consommation mémoire importante vis à vis d'une solution dédiée.

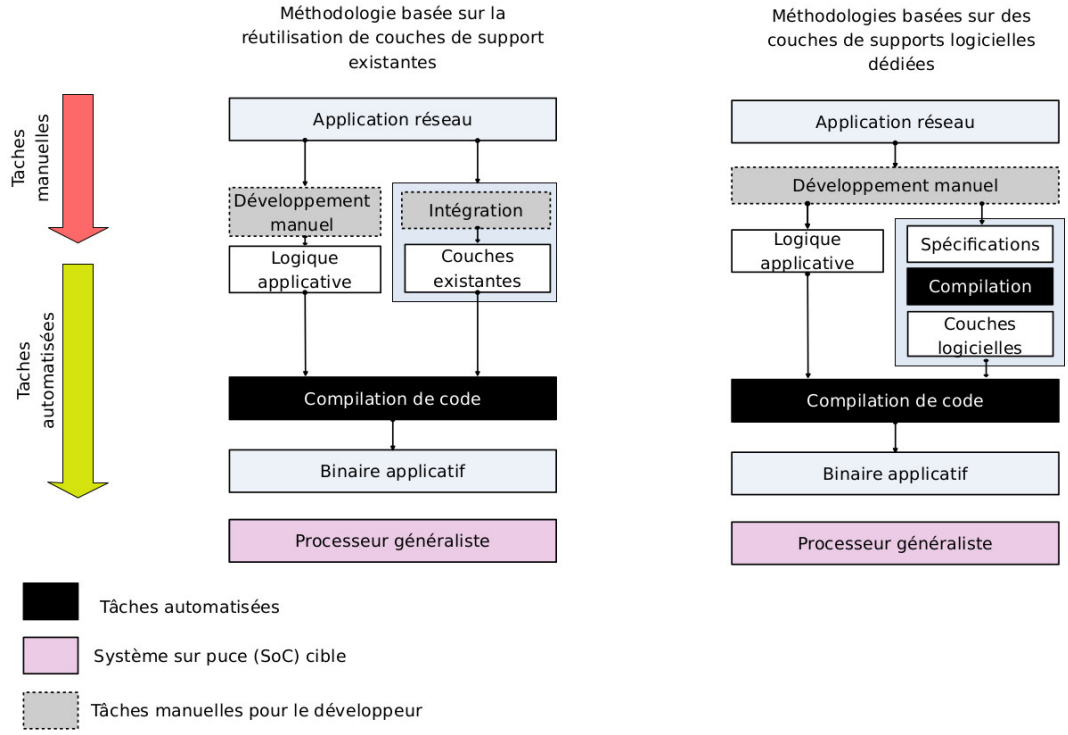


FIGURE 5.1 – Stratégies d'intégration d'analyseurs syntaxique dans une application réseau

Par exemple, dans le cas du protocole HTTP, une application n'ayant aucun accès privilégié au serveur n'a nullement besoin de traiter les méthodes `PUT` `PATCH` ou encore `DELETE`. L'efficacité globale de l'application dépend fortement de l'efficacité de son support protocolaire. Celui-ci doit donc se rapprocher au plus près des besoins spécifiques de l'application pour une efficacité optimale.

Pour atteindre cet objectif, diverses méthodologies automatisées permettant de concevoir une couche protocolaire spécifique ont été proposées. Ces approches génèrent les couches logicielles dédiées à partir de spécifications de haut niveau des protocoles comme cela est présenté à la figure 5.1. Ces approches présentent un double avantage : elles permettent une granularité fine de la couche de support en fournissant à l'application une couche adaptée à ses besoins réels et cela à partir de spécifications de plus haut niveau. Les spécifications sont plus proches de celles des protocoles considérés. L'effort d'abstraction est déporté au sein des outils de génération. Le développeur écrit d'une part sa logique applicative et d'autre part une spécification de plus haut niveau, proche de l'ABNF, permettant de générer la couche de support protocolaire. Ainsi, cette couche exprime les besoins spécifiques de l'application et est d'une meilleure efficacité que les couches généralistes présentées précédemment.

Dans le cadre de cette thèse, nous proposons une solution alternative à cette seconde approche. Cette solution est basée sur une plateforme d'exécution regroupant conjointement des parties logicielles et matérielles. Une vue simplifiée de la plateforme est présentée en figure 5.2. Ce système conjoint repose sur l'utilisation d'un processeur

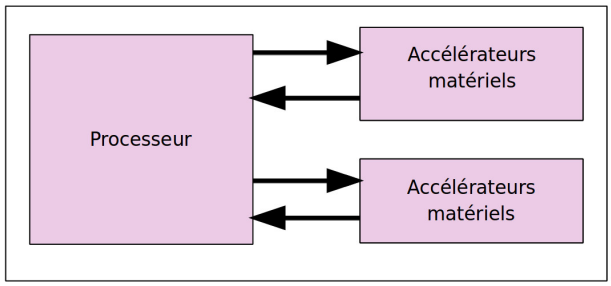


FIGURE 5.2 – Schéma simplifié de la plateforme d'exécution ciblée

dont le rôle est d'exécuter les parties logicielles de l'application réseau et d'accélérateurs matériels dédiés pour l'accélération de l'étape d'analyse protocolaire. Cette plateforme d'execution, nommée Zebra, allie performances et flexibilité afin de proposer une solution à la fois efficace et ergonomique dans le développement des couches de supports protocolaire. Cette dernière sera intégrée dans des composants de type FPGA qui permettent le prototypage et l'évaluation de tels systèmes.

Toutefois, compte tenu des nombreux domaines d'expertise nécessaire au développement et à l'utilisation d'une telle solution, notamment dans le domaine de la microélectronique, nous proposons une méthodologie générative permettant de créer automatiquement ces coprocesseurs ainsi que le code logiciel permettant de les piloter. La méthodologie de conception proposée est présentée dans la figure 5.3.

Dans la suite ce de chapitre, nous allons présenter de manière détaillée la plateforme matérielle Zebra ainsi que les couches logicielles nécessaires à son utilisation.

5.2 Présentation générale de la plateforme

La plateforme Zebra a été développée afin d'intégrer en son sein un processeur et des accélérateurs matériels. L'association de ces éléments doit permettre de fournir de hautes performances ainsi qu'une flexibilité élevée. Les couches de supports protocolaires que nous souhaitons déployer sur la plateforme sont formées, d'une part, de la partie de validation protocolaire à proprement parler, basée sur les spécifications du protocole et d'autre part, de la partie effectuant la gestion et la manipulation (remplissage, lecture, etc.) des structures de données contenant les informations utiles à la logique applicative.

Afin d'architecturer le système conjoint, nous avons débuté notre étude par une analyse du temps d'exécution des couches de supports protocolaires afin d'identifier les parties consommatrices de temps de calcul. Pour mener à bien cette analyse, nous avons profilé différents analyseurs syntaxiques logiciels généré à l'aide de Zébu[BRLM07]. Les analyseurs employés lors de cette évaluation implantaient différents protocoles usuels que nous détaillons dans la partie évaluation présentée au chapitre 7. Les résultats de l'analyse du temps d'exécution des différents analyseurs syntaxiques logiciels a démontré qu'environ 90% du temps d'exécution est consommé par l'étape de validation protocolaire des messages, c'est à dire par l'exécution de la couche logicielle modélisant le

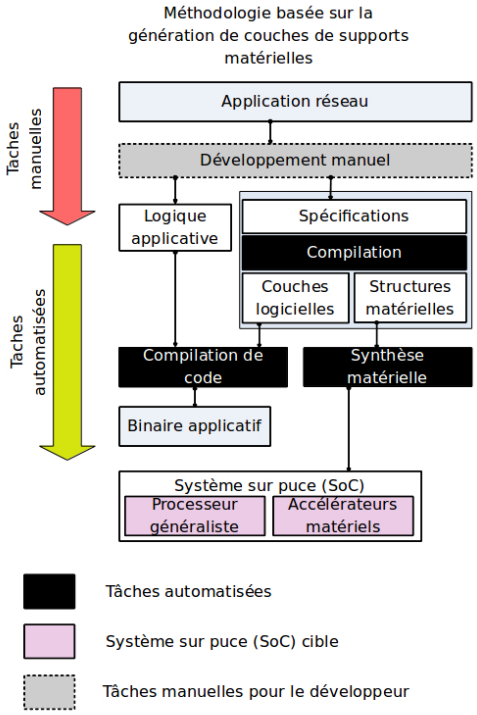


FIGURE 5.3 – Stratégies d'intégration d'analyseurs syntaxique dans une application réseau

protocole *via* des automates à états finis. Le temps d'exécution de la partie en charge du remplissage de la structure de données consomme les 10% du temps restant. Ces données factuelles nous ont guidé dans la conception du un système conjoint dont une vue simplifiée est fournie en figure 5.2.

Comme nous l'avons vu, les protocoles sont décrits et implantés à l'aide d'automates à états finis. L'exécution de ces derniers consomme environ 90% du temps de traitement. En conséquence, nous avons décidé de déporter l'implantation de l'automate décrivant le protocole.

La figure 5.4 présente l'architecture interne du système, implanté à des fin de prototypage dans un FPGA. Cette plateforme, nommée Zebra, est composée principalement des composants suivants :

- Un processeur généraliste chargé de l'exécution du système d'exploitation, de la logique applicative et/ou de la gestion des vues du message. Nous détaillons le rôle du processeur généraliste à la section suivante ;
- Une interface réseau permettant aux applications exécutées sur le processeur généraliste de communiquer ;
- D'accélérateurs matériels ou coprocesseurs chargés du traitement protocolaire. Chaque coprocesseur inclu dans le système est dédié à un protocole.

Considérons l'exemple décrit précédemment : une application réseau en charge de l'authentification des utilisateurs. Elle reçoit un flux de données correspondant à des sé-

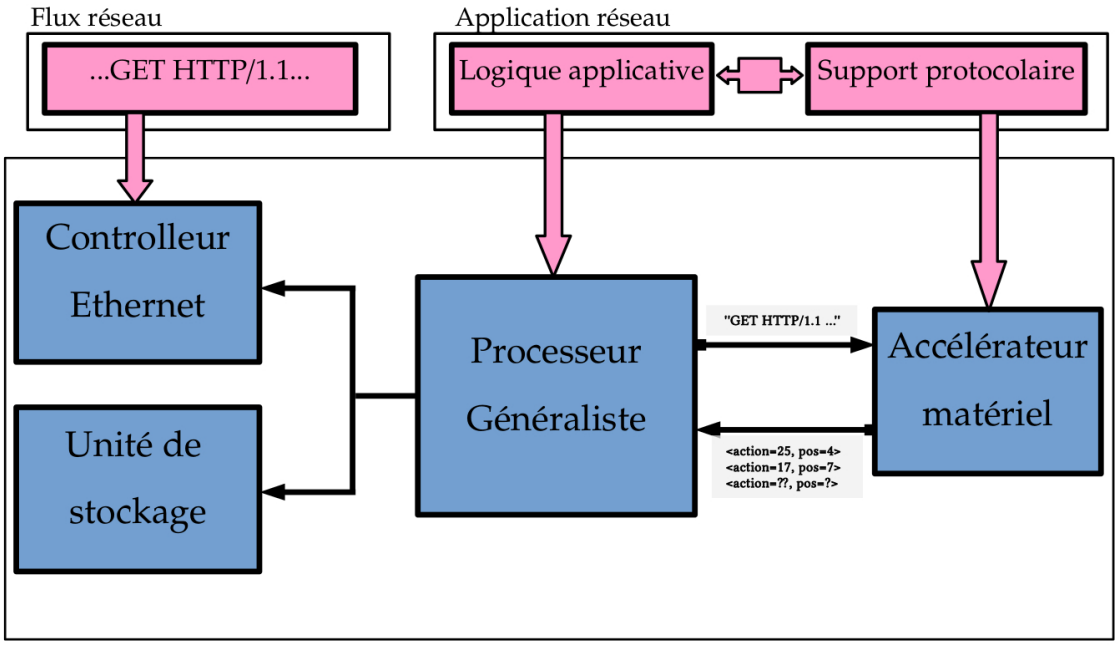


FIGURE 5.4 – Architecture de la plateforme Zebra, articulée autour du couple processeur généraliste - accélérateur matériel

quences de noms d'utilisateur (*logins*) et mots de passe (*password*) via l'interface réseau du système. Le processeur généraliste exécute l'application réseau, il est notamment en charge de la gestion complète du réseau à l'aide d'une pile logicielle, de la logique applicative et du remplissage des vues du message. Ainsi, à la réception du flux de message, celui-ci est directement transmis au coprocesseur réalisant l'analyse du protocole. Ce dernier valide la syntaxe ainsi que la sémantique du message et transmet les résultats de l'analyse au processeur généraliste. Ces résultats se présentent sous forme de couples *(action ; position)*.

L'action correspond à l'identifiant de la fonction de rappel qui doit être invoquée, la position correspond à la position du premier caractère associé à l'action au sein du flux réseau. Cette information permet d'identifier un champ donné. Le processeur généraliste exécute la fonction correspondante à l'action en lui fournissant la position du caractère en paramètre.

Dans notre exemple, lors du traitement de « *USER LOGIN=lambda PASSWORD=123pass456 END* », le premier couple *(action, position)* permet d'identifier le début du champ nom d'utilisateur, c'est à dire le caractère `l`, l'action correspondante permet de mémoriser cette position. Le second couple identifie la fin du champ, c'est à dire le caractère `a`. L'action associée effectue une copie de la suite de caractères `{l, a, m, b, d, a }` dans la structure de données qui sera fournie, par la suite, à la logique applicative.

En fonction du nombre de protocoles à supporter, le nombre de coprocesseurs varie. En effet, les automates permettant d'analyser les messages protocolaires possèdent des structures (nombre d'états, transitions, conditions différentes) en fonction du protocole. A cause des fortes disparités entre les automates, nous avons décidé que les coproces-

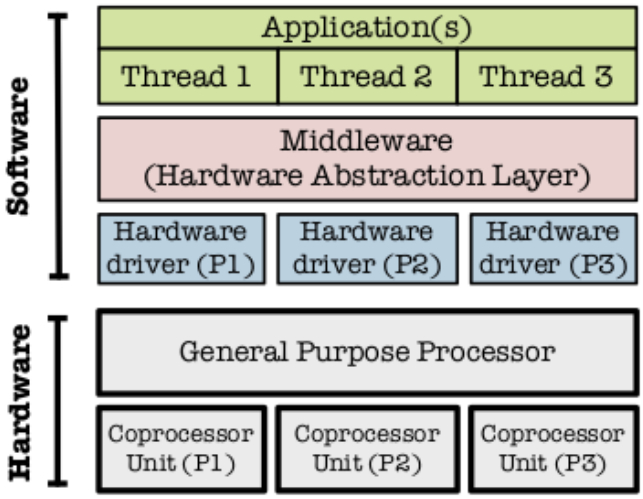


FIGURE 5.5 – Couches matérielles et logicielles mise en oeuvre pour l'exploitation de la plateforme

seurs seraient dédié à un seul protocole.

Afin de rendre l'utilisation de la plateforme matérielle exploitable depuis les couches applicatives, nous avons identifié les différentes couches logicielles à implanter afin de rendre transparent l'implantation des services rendus. La figure 5.5 décrit les différentes couches mises en jeu lors de ce processus.

Afin de pouvoir traiter simultanément les requêtes provenant de multiples clients, les applications sont, le plus souvent, *multithreadées*. Chaque *thread* d'exécution ou processus léger a à sa chage le traitement de requêtes provenant d'un client. Les processus légers s'exécutent simultanément sur le processeur généraliste.

Toutefois, le processeur généraliste n'a à sa disposition qu'un nombre limité d'accélérateurs. Par conséquent, il est nécessaire de mettre en oeuvre un intergiciel, nommé HAL (Hardware Abstraction Layer) permettant d'abstraire les fonctionalités offertes par ces derniers. Il sera chargé de l'affectation d'un ou plusieurs accélérateurs à une application. Elle sera également en charge de l'initialisation des accélérateurs et de la récupération des résultats du traitement. Ceci est effectué via les couches de pilotage des accélérateurs. Enfin, une application ne recevant que très rarement une requête dans sa totalité en une seule fois, il appartient à l'intergiciel de gérer les sauvegardes et les restaurations du contexte d'exécution lors du traitement de requêtes incomplètes.

5.3 Présentation détaillée de la plateforme

Comme nous l'avons précédemment mentionné, la plateforme est composée d'un processeur généraliste et d'un ensemble de coprocesseurs. Il existe principalement deux approches permettant d'intégrer un coprocesseur à un processeur généraliste.

- La première approche intègre directement le coprocesseur à l'intérieur de l'architecture du processeur. Ainsi le coprocesseur devient une unité fonctionnelle à

part entière au même titre que le multiplieur ou l' unité arithmétique et logique. L'avantage principal de cette approche provient du temps de latence faible des communications entre le processeur et le coprocesseur. Toutefois, cette approche s'avère complexe à mettre en œuvre et peu flexible. En effet, d'une part, il est nécessaire de modifier le jeu d'instructions du processeur généraliste afin d'y insérer un ensemble d'instructions permettant de manipuler les coprocesseurs ; et d'autre part, si le coprocesseur nécessite plusieurs cycle d'horloge pour effectuer un traitement, le pipeline d'exécution du processeur généraliste doit être gelé. Le gel du pipeline du processeur inhibe l'exécution parallèle d'autres tâches comme par exemple le traitement d'un autre thread ou la réception de données depuis l'interface réseau.

- La seconde approche, plus générique, vise à interconnecter le coprocesseur au processeur généraliste via un bus périphérique. Les coprocesseurs sont alors perçus comme des périphériques système (tel l'interface réseau). Cette approche présente l'avantage de ne pas nécessiter la modification du jeu d'instructions du processeur puisqu'il est définit en tant que périphérique externe. De plus, le processeur généraliste ne reste pas bloqué pendant le traitement effectué par le coprocesseur. Toutefois, l'échange d'information entre les deux composants est plus lent, notamment à cause du partage du BUS de communication. Afin d'accroitre les performances de l'approche, il est cependant possible d'utiliser des mécanismes de type DMA afin de décharger le processeur de cette tache et d'accélérer le transfert de volumes important d'information.

La seconde approche basée sur l'utilisation de DMA semble fournir des caractéristiques intéressantes. Toutefois, le cadre applicatif étudié ici réduit fortement ses avantages. En effet, l'utilisation de DMA est efficace lorsque ❶ les flux de données sont de tailles déterministes et ❷ lorsque le volume d'information a transporté est important. La première partie du traitement visant à fournir les données à traiter au coprocesseur répond à ces 2 exigences. Cependant, la lecture, en sortie du coprocesseur, des résultats de l'analyse, dont le nombre de couples *(action, postions)* est ❶ faible et ❷ indéterminé à priori ce qui limite l'intérêt de l'approche. Ce constat est renforcé si l'on considère les accès aux registres de contrôle que devront nécessairement posséder les coprocesseurs afin d'indiquer par exemple leur statuts, le nombre de données en attente, etc. De plus, le développement d'un composant matériel de type DMA est une tâche complexe, tout comme sa mise au point.

Aucune des deux approches ne répond pleinement aux besoins énoncés. Par conséquent, nous avons mis au point une approche alternative afin d'intégrer nos coprocesseurs dans l'architecture du système. Cette approche vise à répondre aux besoins intrinsèques du domaine liés au domaine applicatif :

- L'application logicielle ou le système d'exploitation déployé sur le processeur doit être en mesure de pouvoir exécuter diverses tâches lorsqu'un coprocesseur traite un flux ;
- Différents threads d'exécution d'une même application doivent pouvoir faire traiter des données en parallèle sur des coprocesseurs distincts ;
- Différents threads d'une même application doivent pouvoir faire traiter des don-

nées séquentiellement sur un même coprocesseur (Thread 1, puis Thread 2, etc.) ;
- Le temps d'accès aux entrées/sorties des coprocesseurs (registres de configuration, résultats, etc.) doit être suffisamment rapide pour ne pas compenser l'accélération obtenue grace à l'utilisation des coprocesseurs.

Ainsi, afin de respecter ces contraintes fonctionnelles, nous avons décidé d'implanter une version modifiée de la première approche. La principale modification vise à implanter un système de communication asynchrone entre le processeur et ses coprocesseurs. Par conséquent, lorsqu'un coprocesseur effectue des traitements, le pipeline du processeur n'est plus gelé. Il peut donc réaliser d'autres traitements en attendant la production des résultats. Cependant le fonctionnement asynchrone de ces 2 éléments rend la présence de signaux de contrôle et d'états nécessaire, par exemple pour savoir si le coprocesseur a terminé le traitement du flux ou s'il a encore des données en attente de traitement. La figure 5.6 présente une vue plus explicite de cette architecture matérielle.

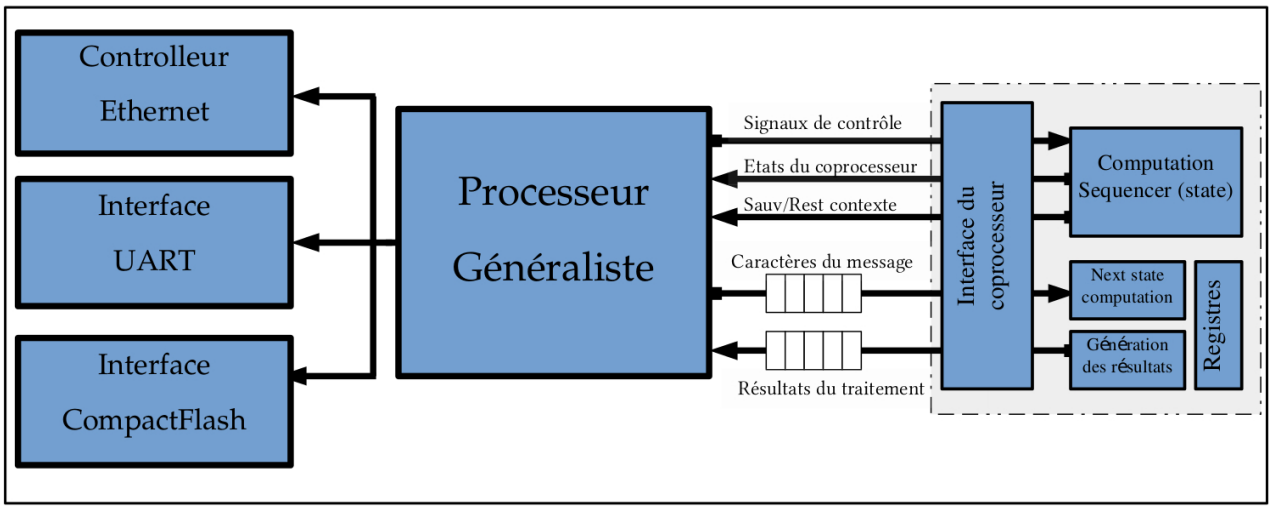


FIGURE 5.6 – Couplage du processeur et des coprocesseurs

Le découplage du processeur et des coprocesseurs est réalisé à l'aide de files d'attentes matérielles (FIFO). Ces FIFOs, dont le fonctionnement est similaire aux files d'attente logicielles, permet à chacun des acteurs de venir consommer et/ou produire des éléments a son rythme. La synchronisation des deux acteurs est réalisée à l'aide de signaux de type FIFO vide ou FIFO pleine. Ces derniers permettent, premièrement, de contraindre l'exécution de l'automate implanté dans les coprocesseurs à la présence de données à traiter. Deuxièmement, ils sont exploités par les pilotes logiciels afin de déterminer si d'éventuels résultats ont été produits par le traitement.

Afin de prendre en considération les besoins de flexibilité énoncés précédemment, une interface générique permettant d'associer au processeur jusqu'à 8 coprocesseurs matériels a été développée. Le routage des informations vers le coprocesseur adéquat est réalisé à l'aide d'un paramètre fourni à l'instruction assembleur correspondante au niveau du code logiciel.

Ainsi, l'architecture développée permet de répondre aux besoins fonctionnels énoncés précédemment. De plus, l'intégration des coprocesseurs au sein de l'architecture du

processeur généraliste permet une communication rapide entre ces deux entités. En effet, l'accès au coprocesseur en lecture ou en écriture ne nécessite qu'un unique cycle d'horloge. Ce délai est équivalent au temps nécessaire à la réalisation d'une opération arithmétique ou logique. Par conséquent, les temps de communications n'impactent pas significativement sur le temps de traitement.

5.4 Extension du jeu d'instructions du processeur

Afin de pouvoir piloter le coprocesseur à partir d'éléments logiciels, un ensemble d'instructions doivent être rajoutées au jeu d'instructions du processeur. Ces nouvelles instructions seront en charge de :

- l'échange de données entre le processeur et les coprocesseurs : envoi des caractères des messages à traiter, lecture des résultats de traitements ;
- le contrôle du fonctionnement des coprocesseurs : lecture de l'état du coprocesseur (actif, passif) ;
- la lecture de l'état des fifos : nombre de données en attente, état vide ou état plein ;
- d'effectuer une restauration/sauvegarde de contexte [1] ;
- la lecture de l'identifiant du coprocesseur : cette identifiant fournit le type/nom du protocole supporté par le coprocesseur.

Une synthèse des instructions rajoutées dans le cœur du processeur est fournie dans la figure 5.7. Les prototypes fournis correspondent aux prototypes de fonctions C qui encapsulent l'appel aux instructions assembleur.

L'ensemble de ces instructions a été rajouté au sein de l'architecture du processeur (dans son décodeur d'instructions). Les liens physiques entre le pipeline du processeur et les coprocesseurs ont été établis. Et finalement la chaine de compilation logicielle a été mise à jour pour permettre l'utilisation de ces instructions à partir de codes logiciels développés en langage C. Ainsi, un ensemble d'instructions dédiées à la manipulation du coprocesseur a été intégré dans le code source RTL du coeur de processeur généraliste. Une liste des fonctions faisant appel à ces instructions est fournie dans le tableau 5.7.

5.5 Architecture interne des coprocesseurs

5.5.1 Architecture

L'automate décrivant le protocole à analyser est implémenté dans le coprocesseur, celui-ci traite le message et fournit au processeur généraliste les informations nécessaires au remplissage de la vue finale. Afin d'obtenir des performances accrues, une implémentation intégralement matérielle du coprocesseur a été effectuée.

1. Dans le cadre d'applications *multithreadées*, il est nécessaire de sauvegarder le contexte d'exécution avant d'attribuer le coprocesseur à une autre tâche. Ainsi, la première tache peut reprendre son exécution une fois qu'elle aura reçu la prochaine trame Ethernet.

Instruction	Description
void px_reset(void)	Réinitialise le coprocesseur ainsi que le compteur interne de caractères.
void px_send(char)	Envoie un caractère au coprocesseur, pour traitement.
void px_send(char[4])	Envoi des caractères 4 par 4. Ceci permet l'exploitation complète du BUS 32*bits* pour accélérer le traitement.
bool px_isRunning(void)	Notifie si, oui ou non, le coprocesseur est actuellement en cours de traitement.
bool px_isIdle (void)	Notifie si, oui ou non, le coprocesseur a terminé le traitement des données.
bool px_hasResults (void)	Notifie s'il y a des résultats de traitement en attente dans la FIFO de sortie du coprocesseur.
int px_getResults (void)	Lit le résultat du traitement de la FIFO de sortie du coprocesseur, sous la forme d'un couple *(action,position)*.
int px_Identifier (void)	Renvoie l'identifiant du coprocesseur. Permet d'identifier le type de protocole supporté par le coprocesseur.
int px_saveContext (void)	Sauvegarde le contexte actuel afin d'affecter le coprocesseur à un autre traitement dans le cadre d'applications *multithreadées*.
void px_loadContext (int)	Restaure un contexte donné en vue de la reprise d'un traitement préalablement interrompu.

FIGURE 5.7 – Instructions rajoutées au coeur de processeur généraliste pour la gestion des coprocesseurs

Comme nous l'avons vu précédemment, l'implémentation de l'automate décrivant le protocole est effectuée à l'aide de machines à états finis, largement répandues dans le domaine des analyseurs syntaxiques. Le principal avantage d'une implémentation matérielle de machines à états finis comparé à leur homologue logiciel réside dans leur efficacité. Une implantation matérielle est spécifiquement étudiée afin d'effectuer plusieurs opérations parallèles en un seul cycle d'horloge. Elle permet une évaluation de plusieurs transitions en même temps. De plus, les branchements conditionnels, utilisés dans les implémentations logicielles de machines à états finis, sont évalués en un unique cycle d'horloge. Par exemple, si l'on considère l'automate présenté dans la première partie de la figure 5.9, l'ensemble des transitions d'un état donné peuvent être évaluées en un unique cycle d'horloge. Ainsi, le temps d'évaluation nécessaire pour 1 transition (Etat 0) est le même que le temps d'évaluation nécessaire pour 4 transitions (Etat 3). Il est difficile d'atteindre ce temps d'exécution constant sur un processeur généraliste où le nombre d'instructions assembleurs nécessaires (et leur temps d'exécution) dépend du nombre de transitions ainsi que de la complexité associée. Une implémentation matérielle peut garantir un temps de traitement constant et indépendamment de la complexité intrinsèque de l'automate associé en contrepartie d'un coût silicium.

Un modèle d'automate décrivant le protocole à analyser est généré sur la base de la spécification Zebra fournie par le développeur de l'application. Dans cette spécification,

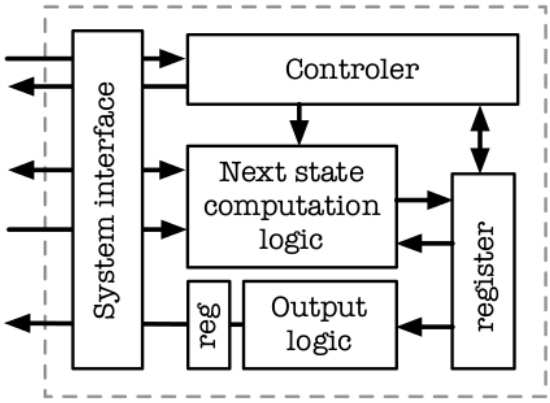


FIGURE 5.8 – Architecture d'un coprocesseur

seul le protocole est décrit. L'automate réalisant l'analyse du flux est généré à partir de cette spécification. Ensuite, des états spécifiques à la synchronisation des entrées/sorties ainsi que d'autres fonctionalités telle que la sauvegarde/restauration de contexte sont alors insérées dans l'automate. Enfin, l'architecture matérielle de niveau RTL est générée. Cette architecture matérielle est composée de quatre parties principales (Figure 5.8) :

- Un module dédié à la sauvegarde de l'état courant,
- Un module dédié au calcul des transitions,
- Un module dédié au calcul des sorties de l'accélérateur,
- Un module dédié à la restauration/sauvegarde de contexte.

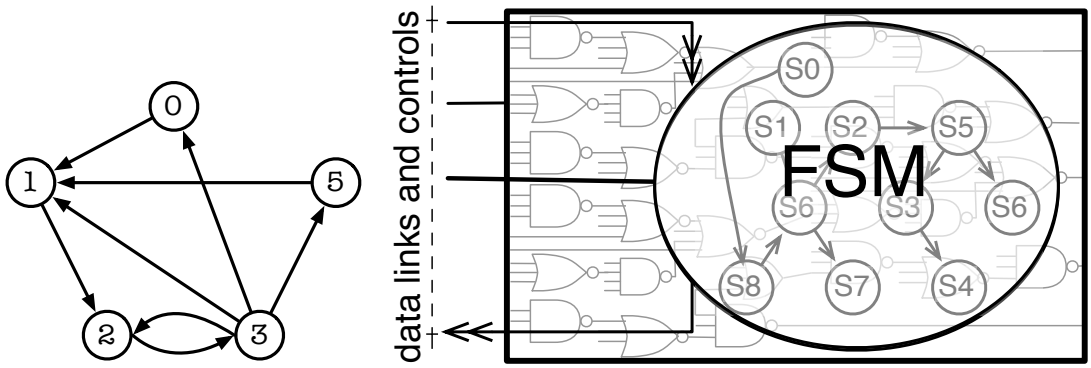
L'implémentation du calcul des transitions a été dimensionnée afin de permettre l'évaluation du calcul de toutes les transitions en un seul cycle d'horloge. Le temps de traitement d'un caractère nécessite donc un cycle d'horloge dans le cas où il n'y a pas d'actions levées et trois cycles dans le cas où il y en a une. Le temps de traitement d'un caractère est ainsi indépendant du nombre de transitions d'un état de l'automate et de leur complexité associée. Cette caractéristique requiert toutefois une augmentation de la complexité matérielle des accélérateurs.

Les ressources permettant la restauration/sauvegarde de contexte permettent de transformer le couple (état du parser, position du dernier caractère traité) dans le flux en un mot de 32*bits* pouvant être restauré/sauvé par le processeur généraliste en un unique cycle d'horloge. Le changement de contexte est ainsi très rapide, notamment dans un contexte multithreadé où un ensemble de plusieurs threads peuvent nécessiter l'accès au même coprocesseur.

5.5.2 Communications

Une vue détaillée des entrées/sorties des coprocesseurs est fournies dans la figure 5.10. L'interface d'Entrée/Sortie du coprocesseur est principalement composée de quatre sous-ensemble :

- Le premier est dédié à l'envoi des données au coprocesseur (*input data*) ;



(a) Automate à états finis (b) Automate implémenté dans le coprocesseur

FIGURE 5.9 – Implémentation de l'automate décrivant le protocole dans le coprocesseur

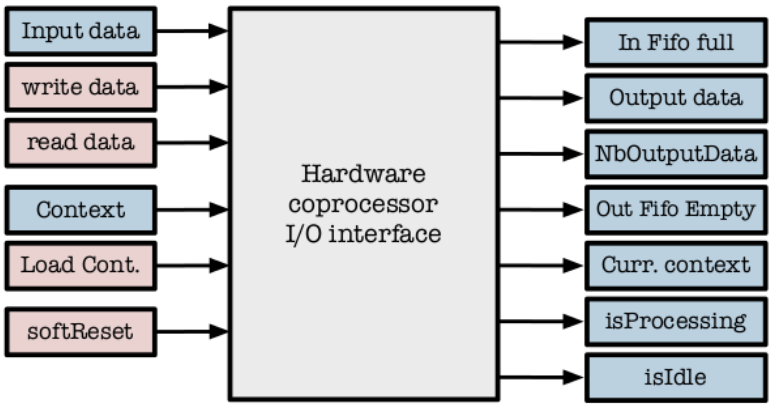


FIGURE 5.10 – Interface d'entrées/sorties du coprocesseur

- Le deuxième est dédié au transfert des résultats du traitement (couple numéro d'action-position dans le flux) ;
- Le troisième est dédié aux signaux de contrôl du coprocesseur (*idle*, *runing*, *etc*) ;
- Le quatrième est dédié à l'envoi du statut du coprocesseur.

L'envoi des données au coprocesseur se fait sur un champ d'une largeur de 32*bits*. Suivant la configuration, il est possible de transmettre un seul caractère par cycle d'horloge ou bien quatre caractères, cette dernière possibilité permet d'optimiser le temps de chargement des données dans le coprocesseur. En effet, cette approche permet de masquer les cycles d'horloge nécessaires au processeur généraliste pour accéder au flux réseau en mémoire et effectuer l'envoi au coprocesseur. Ainsi, une interface d'entrée de 32*bits* fournit au coprocesseur assez de données pour une durée de 4 à 8 cycles d'horloge. L'envoi des résultats du coprocesseur vers le processeur généraliste se fait également *via* une interface d'une largeur de 32*bits*. Le couple (action,position) est concaténé en un mot de 32*bits* pour un accès plus rapide. Enfin, l'interface effectuant la restauration et la sauvegarde de contexte est sur une largeur de 32*bits* également.

Du point de vue de l'application, l'accès à ces instructions ou l'accès au coprocesseur de manière plus générale ne se fait pas directement. Pour une meilleure transparence vis à vis du développeur de l'application, le pilotage du coprocesseur *via* ces ensembles d'instructions se fait à l'aide de fonctions de plus haut niveau faisant appel à ces instructions, telles qu'elles sont décrites dans le tableau 5.7. De cette manière, l'application dispose d'une interface de programmation complète permettant la gestion des coprocesseur. En effet, les applications peuvent accéder aux différents coprocesseurs de manière concurente dans un contexte multithreadé puisqu'elles gèrent un grand nombre de connexions simultanées. L'architecture sous-jacente doit alors supporter le traitement parallèle de ces clients. De plus, de manière générale, les trames sont rarement reçus dans l'ordre ou reçu entièrement. Les possibilités de sauvegarde/restauration de contexte permettent de libérer les coprocesseurs des aléats du réseau et de les rendre indépendant vis à vis des couches supérieures.

5.6 Bilan

Ce chapitre présente une description de la plateforme de prototype Zebra que nous ciblons. Après avoir mis en évidence le fait que la plus grande majorité du temps d'exécution d'une couche de support protocolaire est consacré à au traitement de l'automate décrivant le protocole, nous avons décidé d'orienter le prototype Zebra vers la migration de ce traitement effectué par le processeur vers une unité d'exécution spécialisée.

Dans un soucis d'efficacité, nous avons intégré sur une même plateforme, l'ensemble des composants nécessaires au déploiement d'un système : une interface réseau, un espace de stockage, un processeur généraliste, le ou les coprocesseurs, des unités d'entrées/sorties.

La gestion des logiques applicatives et du réseau ainsi que le remplissage des structures contenant l'information utile à l'application a été laissée à un processeur généraliste, cela afin de fournir la flexibilité nécessaire aux applicatifs réseaux. Afin de pouvoir piloter les coprocesseurs, nous avons étendu le jeu d'instructions du processeur en y ajoutant des instructions spécifiques. Les applications nécessitant un traitement protocolaire accèdent à ces coprocesseurs *via* des fonctions de haut niveau faisant appel à ces instructions, l'ensemble étant donc complètement transparent du point de vue de l'application réseau et donc du développeur.

Dans une démarche d'ergonomie de développement et de transparence vis à vis du matériel sous-jacent, nous avons développé le langage dédié Zebra. A partir de spécifications de haut niveau dont la syntaxe se rapproche à celle avec laquelle sont définis les protocoles réseaux, ce langage permet de générer l'automate implémenté dans le coprocesseur, les structures de données utiles à l'application ainsi que les fonctions de manipulation de ces structures.

6

Architecture logicielle Zebra

DANS les chapitres précédents, nous nous sommes attelés à décrire la plateforme de prototypage Zebra mise au point en vue de l'accélération du traitement des couches de support protocolaire. Le développeur a ainsi à sa disposition une plateforme d'exécution performante exploitant les possibilités offertes par celle-ci sans qu'il n'ait besoin d'élargir son domaine de compétences sur son fonctionnement interne.

Ce chapitre propose de décrire l'architecture logicielle mise en oeuvre autour de la plateforme Zebra : de l'écriture de la spécification Zebra du protocole jusqu'au traitement protocolaire effectué dans le coprocesseur et le remplissage et l'exploitation de la vue résultante.

Ainsi, à partir de la spécification Zebra, nous présentons le flux de génération des composants ainsi que la manière dont ils s'agglutinent pour former le prototype matériel-logiciel conjoint. Nous détaillons ensuite les étapes exécutées de la réception des données sur l'interface réseau du système jusqu'à leur traitement dans le processeur généraliste *via* l'intergiciel Zebra. Enfin, nous mettons l'accent sur le processus d'analyse protocolaire et l'échange des données et des résultats du traitement au niveau du processeur généraliste - coprocesseur.

Sommaire

6.1 Introduction

6.1.1 Processus général

Lors du chapitre 4, nous avons exploré la démarche que nous avons suivi au cours de ces travaux de thèse. Ces travaux s'articulent autour d'une conception conjointe logiciel-matériel. Elle présente les démarches qui permettent de réaliser la conversion d'une spécification de haut niveau en une architecture exécutable sur un ensemble de composants interconnectés (CPU et accélérateurs matériels). Ces différents composants peuvent être transposés sur des éléments logiciels ou matériels.

6.1.2 Composants générés

Les différents composants impliqués dans cette méthodologie sont les suivants :

Entrée - Les composants fournis pour l'exploitation de la plateforme Zebra :

- Une spécification en langage Zebra du protocole à analyser. L'utilisation du langage Zebra permet de se rapprocher de l'ABNF afin de diminuer le fossé entre la spécification du protocole et le code bas-niveau correspondant à l'analyseur syntaxique associé.
- Une description RTL du processeur généraliste nécéssaire à l'exécution des couches logicielles. Cette description est modifiée afin de pouvoir communiquer avec les coprocesseurs à travers les instructions de pilotage décrites lors de la présentation de la plateforme au chapitre 5.

Sortie - A partir de ces éléments, les outils fournis génèrent les composants suivants :

- Une description RTL des coprocesseurs décrivant les protocoles à traiter.
- Une description RTL modifiée du processeur généraliste afin d'instancier et établir la communication avec les coprocesseurs.
- Un ensemble d'instructions bas-niveau permmettant la gestion des différents coprocesseurs (configuration, contrôle, statuts, entrée/sortie, modification de la chaîne de compilation)
- Une configuration de la couche basse de l'intergiciel Zebra incluant notamment le nombre de coprocesseur disponibles et les protocoles associés.
- Un ensemble de fonctions logicielles, de plus haut niveau, lié aux besoins spécifiques de l'application cible. Ces fonctions fournissent une interface de programmation pour l'accès et la gestion logicielle des coprocesseurs et des vues des messages (remplissage, accès). Ces éléments logiciels sont générés à partir de la spécification du protocole fournie en entrée.

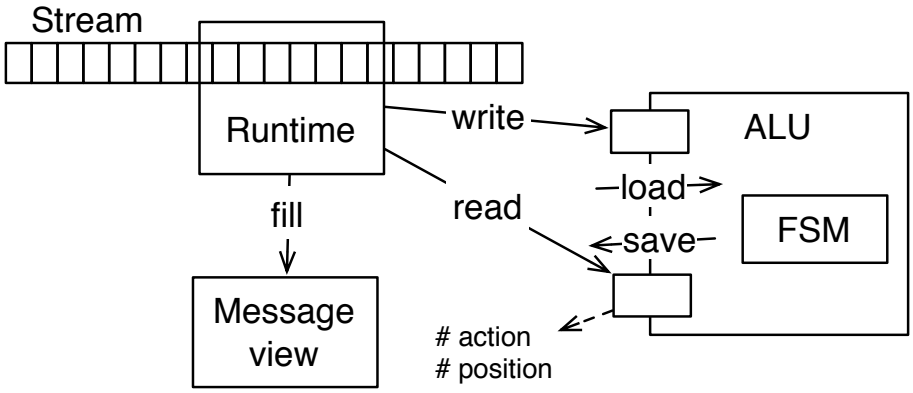


FIGURE 6.1 – Flux d'exécution associé à la plateforme Zebra

6.1.3 Flux d'exécution

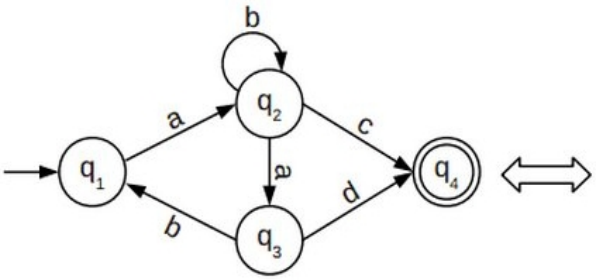
La figure 6.1 décrit le flux d'exécution associé. Il se divise ainsi en deux parties : la première partie du flux s'exécute sur le processeur généraliste. Elle comprend la réception des données sur l'interface réseau jusqu'à l'envoi de ces données au coprocesseur, après un changement de contexte de celui-ci. La deuxième partie est concentrée sur le coprocesseur : il effectue le traitement des résultats fournis par le coprocesseur. Ces résultats, sous forme de couples (action,position), sont utilisés pour effectuer le remplissage de la vue manipulée par la logique applicative. La réception du flux à travers le réseau ainsi que la vue disponible correspond au niveau *applicatif*, l'envoi, la réception aux coprocesseurs ainsi que le remplissage de la vue est gérée par le HAL et enfin, le coprocesseur en lui même intégrant l'automate décrivant le protocole correspond à la partie matérielle de la plateforme formée par le processeur généraliste et des coprocesseurs. travers les sections suivantes, nous proposons une description détaillée de ce flux sur la base d'un exemple concret de protocole : le protocole HTTP.

6.2 Génération des composants

6.2.1 Langage

6.2.1.1 Formalisme ABNF comme socle de Zebra

Nous l'avons vu au cours des chapitres précédents, le formalisme ABNF est utilisé dans les spécifications de protocoles réseaux pour la spécification de la syntaxe des protocoles. Ce formalisme permet d'exprimer, de manière simple et intuitive, les besoins des protocoles à travers des constructions qui lui sont propres. Cette simplicité qu'offre ce formalisme permet également de monter en niveau d'abstraction et de s'affranchir des contraintes qui seraient liées à l'écriture d'analyseurs syntaxiques. En effet, comme nous l'avons vu, il appartient au développeur de traduire cette spécification dans un langage qui lui convient. En plus de la difficulté d'une telle traduction, le développeur



		a	b	c	d
→	q_1	q_2	∅	∅	∅
	q_2	q_3	q_2	q_4	∅
	q_3	∅	q_1	∅	q_4
*	q_4	∅	∅	∅	∅

FIGURE 6.2 – Représentation d'une machine à états finis : graphe à gauche, table de transitions à droite.

fait face aux contraintes inhérentes de ces langages en termes d'expressivité et de performances. Enfin, le développeur doit décortiquer cette spécification afin d'y puiser uniquement les éléments qui lui sont nécessaires à l'application qu'il programme.

Zebra est basé sur le formalisme ABNF, d'une part afin de tirer parti des avantages qu'il offre en terme d'ergonomie et d'expressivité et d'autre part, afin de réduire considérablement la charge du développeur puisqu'il n'a plus cet effort de traduction vers un langage de plus bas niveau à effectuer. En effet, le programmeur a la possibilité de prendre certaines parties de la spécification telles quelles afin de commencer le développement d'un analyseur syntaxique. Ceux ci sont le plus souvent programmés à l'aide de *machines à états finis*.

6.2.1.2 Machines à états finis

Une *machine ou automate à états finis* est une structure abstraite ayant des entrées et des sorties discrètes et constituée d'états et de transitions. L'automate passe d'état en état, suivant les transitions, à la lecture de chaque caractère du mot en entrée. On peut citer quelques exemples élémentaires comme les ascenceurs, les portiques d'accès aux bâtiments ou encore les analyseurs lexicaux.

Les machines à états finis sont représentés par des graphes mettant en exergue des diagrammes de transition ou par une table de transition comme l'illustre la figure 6.2.

De premier abord, une machine à états finis peut ainsi être vue comme un graphe orienté avec des labels sur chaque arc et transition. Les analyseurs syntaxiques de protocoles réseaux textuels sont programmés, le plus souvent, à l'aide de machines à états finis. Cette machine *accepte* un message protocolaire s'il y a un chemin entre l'état initial et un état final lors de la lecture de chaque caractère du message.

Les performances d'un analyseur syntaxique sont alors directement liées à la manière de programmer les machines à états finis définissant le protocole à analyser. En effet, le passage d'une structure abstraite à une représentation concrète peut être délicat suivant les technologies employées et les contraintes de ressources qui y sont liées. Nous détaillons ce point dans le chapitre 7, consacré à l'évaluation de notre approche.

6.2.1.3 Description

Afin de pouvoir capitaliser sur l'existant, le langage Zebra est basé sur Ragel [Thu06], un outil de compilation de machines à états finis. Nous avons enrichi les fonctionalités fournies par Ragel suivant les nécessités de notre approche. Par exemple, notre approche permet de générer les fonctions de rappel dédiées à la gestion et au remplissage des vues des messages.

Zebra est un langage de compilation de machines à états finis. Il permet non seulement de reconnaitre des expressions régulières mais également d'exécuter à des points clés de reconnaisance de l'expression régulière. Lors de notre étude des normes des protocoles réseaux et de leurs spécifications, nous avons constaté l'omniprésence de la description des messages protocolaires en ABNF. Naturellement, la syntaxe du langage Zebra tend vers la syntaxe de l'ABNF afin de minimiser l'intervention du développeur.

Ainsi, en nous basant sur la syntaxe existante de Ragel, déjà proche de l'ABNF, nous avons rajouté différentes annotations afin que le développeur puisse décrire la structure de la vue du message qu'il souhaite. Le coeur du langage est composée d'opérateurs d'expression régulières standards (union, concaténation, Kleene Star) et d'un ensemble d'opérateurs permettant d'embarquer des actions à exécuter lors de la reconnaissance de motifs. Dans notre cas, les actions embarquées consistent à sauvegarder les caractères de début et de fin de motif. Une fois les caractères sauvegardés, lors de la reconnaissance de la fin du motif, une copie mémoire des caractères situés entre la position du début et la position de la fin est effectuée. Ainsi, nous avons conçu une approche permettant de générer ces actions afin que le développeur n'ait pas à se préoccuper de la gestion des champs traités, une fois que ceux-ci ont été identifiés dans le flux réseau. La figure 6.3 en fournit un exemple.

En effet, lors de l'analyse de la spécification Zebra, les champs nécessaires à l'application sont identifiés et la vue du message correspondante est générée. Par exemple, si l'on considère l'extrait de la spécification Zebra pour le protocole HTTP présentée en figure 6.4 extrayant notamment les champs `Content-Lenght` et `date`, nous retrouvons les fonctions de rappels correspondantes (respectivement *http_msg_view_get_clen*, *http_msg_view_fill_clen* et *http_msg_view_get_date*, *http_msg_view_fill_date*) au remplissage de ces champs une fois qu'ils seront identifiés.

Ainsi, l'ensemble des éléments nécessaires à la gestion mémoire et aux différents remplissage des vues des messages sont déportés au sein de l'intergiciel, *in fine* l'application fournissant uniquement en entrée une spécification Zebra, que nous présentons dans la section suivante.

6.2.2 Spécifications

L'application traite des requêtes basées sur le protocole HTTP. Elle doit être en mesure d'analyser les en-têtes du protocole. Toutefois, elle ne nécessite que certaines en-têtes en particulier. Ainsi, la couche de support n'a pas besoin de supporter et d'extraire l'ensemble des en-têtes décrits dans les spécifications du protocole HTTP [].

Ces en-têtes spécifiques sont les suivantes :

- `Content-Length` et `Content-Type` : longueur et type du corps de la requête.

```
typedef struct{
char* clen;
char* date;
char* host;
char* uri;
} http_msg_view;

http_msg_view* http_msg_view_new(){
http_msg_view *msg_view =
  (http_msg_view *)malloc(sizeof(*msg_view));
msg_view->clen = NULL;
msg_view->date = NULL;
msg_view->host = NULL;
msg_view->uri = NULL;
return msg_view;
}
char* http_msg_view_get_clen(http_msg_view *msg_view){
return msg_view->clen;
}
void http_msg_view_fill_clen(http_msg_view *msg_view, char* val, int len){
msg_view->clen = str_n_copy(val, len);
}
char* http_msg_view_get_date(http_msg_view *msg_view){
return msg_view->date;
}
void http_msg_view_fill_date(http_msg_view *msg_view, char* val, int len){
msg_view->date = str_n_copy(val, len);
}
[...]
```

FIGURE 6.3 – Vue du message et fonctions de rappels associées

- `User Agent` : Agent utilisateur permettant d'identifier le client.
- `Host` : Hôte permettant d'identifier le FQDN ou l'addresse IP cliente.
- `Date` : date de la requête.

A cette fin, le développeur fournit une spécification Zebra permettant de générer les composants logiciels et matériels. La figure 6.4 présente un extrait de la spécification Zebra correspondante au protocole considéré.

```
architecture core of parser is
{
        machine zebra_http_parser;

        #Actions à réaliser sur les champs identifiés
        action mark_start_tok {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(0,16);
                 data_out_valid <= '1';
        }
        action ua_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(1,16);
                 data_out_valid <= '1';
        }
        action ctype_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(2,16);
                 data_out_valid <= '1';
        }
        action host_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(3,16);
                 data_out_valid <= '1';
        }
        action port_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(4,16);
                 data_out_valid <= '1';
        }
        action clen_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(5,16);
                 data_out_valid <= '1';
        }
        action body_write {
               data_out <= conv_std_logic_vector(mark,16) &
              conv_std_logic_vector(6,16);
                 data_out_valid <= '1';
        }

        #Format des messages à analyser
        CRLF = "\r\n";
        ua_value = any*  >mark_start_tok %ua_write;
        ctype_value = any* >mark_start_tok %ctype_write;
        host_value = (alnum* | "." )* >mark_start_tok %host_write (":" (digit+)
             >mark_start_tok %port_write)?;
        clen_value = (digit+)>mark_start_tok %clen_write;
        body = any* >mark_start_tok %body_write ;
        UA = ( "USER-AGENT:" | "User-Agent:" ) . " "* . ua_value :> CRLF;
        CTYPE = ( "CONTENT-TYPE:" | "Content-Type:" ) . " "* . ctype_value :>
             CRLF;
        HOST_PORT = ( "HOST:" | "Host:" )    " "*    host_value :> CRLF;
        CLEN = ( "CONTENT-LENGTH:" | "Content-Length:" )  . " "*? .  clen_value
             > CRLF;
        PATH = any*;
        req_headers = ( HOST_PORT  . UA . CLEN  .  CTYPE );
        Req_Method = "POST";
        Req_Line = Req_Method " "* PATH " "* "HTTP/1.1" CRLF;
        Http_Req_Msg = Req_Line req_headers body CRLF;
        main := ( Http_Req_Msg )*;
}
write vhdl;
end architecture core;
```

FIGURE 6.4 – Extrait d'une spécification Zebra pour le protocole HTTP spécifiée en ABNF

Cette spécification est formée de deux parties : la première partie décrit le format des messages à analyser (ainsi que le format des différents champs) ; la seconde partie s'attache à décrire les actions qu'il faut réaliser lorsque ces champs ont été identifiés et traités.

On retrouve les actions correspondantes aux champs nécessaires à l'application : `ua_write` pour l'agent utilisateur, `clen_write` et `ctype_write` pour la longueur et le type du corps de la requête, `host_write` et `port_write` pour le FQDN ou l'adresse IP et le numéro de port du client. Enfin, `body_write` permet d'identifier et sauvegarder le corps du message.

Ces fonctions servent à l'appel des différentes fonctions de remplissage de la vue du message protocolaire qui est présenté à l'application, une fois que le traitement est effectué.

```
POST / HTTP/1.1

Host: localhost:8080
User-Agent: Firefox
Content-Length: 8
Content-Type: text\\html

RESEAU
```

FIGURE 6.5 – Message reçu sur l'interface réseau, à traiter par l'application

6.2.3 Compilation et synthèse

La figure 6.6 résume le processus de génération des composants et de compilation des éléments logiciels générés.

En sortie, le développeur dispose, d'une part, de son application intégrant la logique applicative et la gestion du réseau, et d'autre part de la spécification Zebra correspondant aux besoins protocolaires de celle-ci.

La spécification du protocole est traitée par le compilateur Zebra, générant les éléments logiciels permettant la gestion des messages protocolaires (vues, remplissage des vues) ainsi que le coprocesseur décrivant l'automate. Le coprocesseur est déployé sur la plateforme d'exécution et les interconnections avec le processeur généraliste sont crées.

Enfin, les éléments logiciels générés (fonctions de pilotage appellant les instructions permettant d'accéder au coprocesseur, les vues logicielles des messages au format HTTP, les fonctions de gestion de ces vues) sont compilés avec le code de l'application réseau afin de créer le binaire exécutable. Comme nous l'avons décrit précédemment, ce compilateur intègre, en son sein, les instructions nécessaires à la manipulation du coprocesseur.

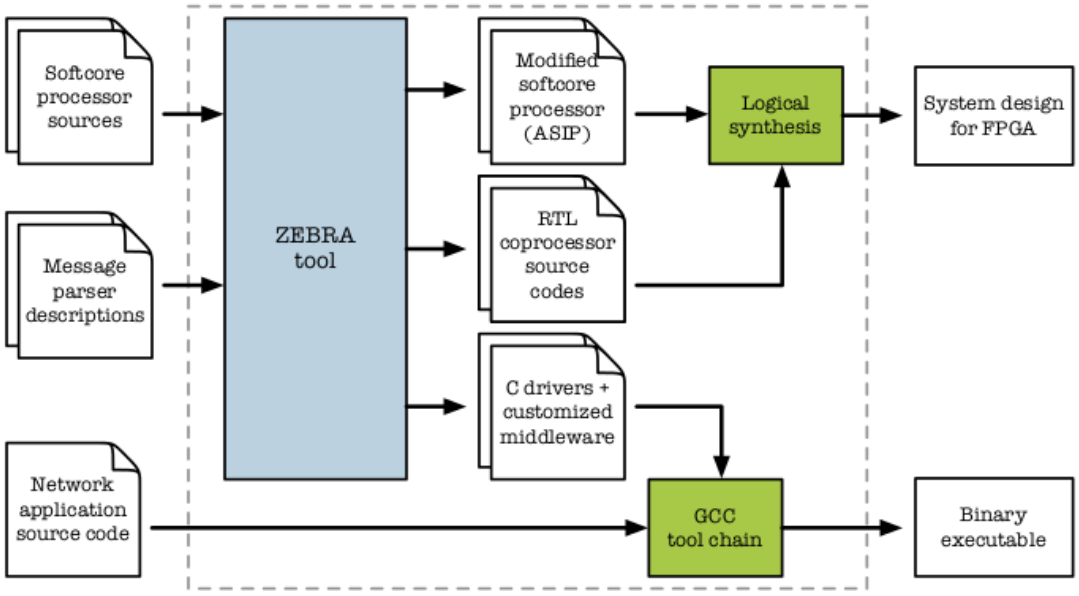


FIGURE 6.6 – Différentes couches conjointes composant l'approche Zebra

La description RTL modifiée du processeur généraliste ainsi que les descriptions RTL des coprocesseurs sont *synthétisées* à l'aide d'outils de synhtèse logique. Les outils de synthèse fournis par Xilin (ISE 4.2) sont exploités dans le cadre de cette étude.

Le développeur dispose ainsi du binaire de son application intégrant les possibilités de traitements protocolaires au niveau matériel ainsi que d'une plateforme d'exécution complète.

6.3 Intergiciel Zebra

6.3.1 Présentation

Les sections précédentes présentent une vision linéaire de l'application réseau accédant au coprocesseur de manière directe à travers les fonctions de pilotage de haut niveau.

Toutefois, les applications réseaux traitent de multiples clients simultanément et sont donc le plus souvent multithreadées. Ainsi, tirer parti du caractère asynchrone des communications entre le coprocesseur et le processeur généraliste devient indispensable, à travers une couche logicielle gérant les accès au coprocesseur. Cette couche présente un ordonnanceur permettant de sélectionner le client à traiter. De plus, les paquets sont rarement réceptionnés dans l'ordre de leur émission. En effet, ceux-ci peuvent arriver par fragments qu'il faut réassembler afin d'obtenir le message originel. Dans un tel contexte, il est difficile de bloquer un coprocesseur en attente de la réception de la suite d'un fragment tandis que d'autres clients peuvent être en attente de celui-ci. C'est la raison pour laquelle nous avons intégré les capacités de restauration et de sauvegarde de contexte dans les coprocesseurs. Ainsi, lorsqu'un message arrive par fragment et que l'analyse

n'est pas terminée, l'état de l'automate ainsi que la position dans le fragment est sauvegardé. Lors de l'arrivée du fragment suivant, cet état sera restauré et l'automate reprendra le traitement sur le caractère dont la position a été sauvegardée au préalable.

De plus, plusieurs protocoles peuvent être traités par plusieurs coprocesseurs différents, ou plusieurs coprocesseurs peuvent également être au même protocole.

Ainsi, l'ensemble de ces fonctionalités ont été déportées vers un intergiciel contrôlant l'ensemble des accès aux coprocesseurs. Les applications réseaux n'ont alors accès qu'à un certain nombre de fonctions de plus haut niveau leur permettant de requérir les ressources d'un coprocesseur à l'intergiciel qui s'occupe alors des accès directs.

L'intergiciel Zebra fournit trois principaux ensembles de fonctionalités :

- Il initialise l'ensemble des coprocesseurs au démarrage et identifie les protocoles supportés par chacun.
- Il gère l'ensemble des accès concurents aux coprocesseurs en ordonnant les requêtes. Il sauve et restaure le contexte de chaque requête suivant les besoins et les fragments reçus.
- Il propose une interface de programmation transparente et simple à l'application cliente, de la même manière qu'une couche de support protocolaire logicielle.

Développer une application réseau avec Zebra devient alors un processus moins complexe au regard des expertises nécessaires dans le domaine de la programmation bas niveau ou de la micro électronique.

6.3.2 Pilotage des coprocesseurs

Comme nous l'avons présenté au chapitre précédent, les coprocesseurs disposent d'instructions de pilotages permettant de gérer l'envoi et la réception des données au travers. Cependant, pour une application donnée, il devient difficile de piloter un coprocesseur directement à travers ces instructions. En effet, l'affectation d'un coprocesseur à une application implique la réservation entière de ces ressources à l'application, pénalisant ainsi les autres applications nécessitant éventuellement ces ressources. D'autre part, l'arbitrage entre les applications qui requiert les différentes ressources et la disponibilité de celles-ci devant se faire de manière transparente, l'intergiciel Zebra propose une interface de programmation rendant l'accès aux ressources et donc aux coprocesseurs transparent.

A cette fin, les instructions ont été encapsulées dans des fonctions standards en C pouvant être appellées par l'intergiciel. Le tableau 6.7 reprend ces fonctions.

Ces fonctions permettent de rendre transparent l'accès aux fonctionalités du coprocesseur. Par exemple, lorsque l'application cliente souhaite lire les résultats du traitement sur la sortie du coprocesseur, elle fait appel à la fonction `int parser_px_Read()}`. L'encapsulation évite à l'application l'utilisation des instructions assembleurs et permet de fournir une interface de programmation homogène aux applicatifs.

6.3.3 Multithreading

Afin d'exploiter au mieux les capacités de parallélisme offerts par la plateforme mise en place, l'intergiciel Zebra permet d'affecter chaque nouveau client de l'application à

```
static inline int parser_pxId(int a){
      int res;
      asm volatile ( "pxid %1, %0 \ n \ t ":"=r" (res) : "r" (a) );
      return res;
}

static inline  int parser_pxNbInData(int a){
     int res;
     asm volatile ( "pxnbindata %1, %0 \ n \ t": "=r" (res) : "r" (a) );
     return res;
}

static inline  int parser_pxNbOutData(int a){
     int res;
     asm volatile ( "pxnboudata %1, %0 \ n \ t": "=r" (res) : "r" (a) );
     return res;
}

static inline  int parser_pxSaveContext(int a){
     int res;
     asm volatile ( "pxsavecontext %1, %0 \ n \ t" : "=r" (res) : "r" (a) );
     return res;
}

static inline  int parser_pxEmpty(int a){
     int res;
     asm volatile ( "pxempty %1, %0 \ n \ t" : "=r" (res) : "r" (a) );
     return res;
}

static inline  void parser_pxsReset(int a){
     asm volatile ( "pxsreset %0 \ n \ t" : : "r" (a));
}

static inline  int parser_pxRead(int a){
     int res;
     asm volatile ( "pxread %1, %0 \ n \ t" : "=r" (res) : "r" (a) );
     return res;
}

static inline  void parser_pxLoadContext(int a, int b){
     asm volatile ( "pxloadcontext %0, %1 \ n \ t" : : "r" (a), "r" (b) );
}

static inline  int parser_pxFull(int a){
     int res;
     asm volatile ( "pxfull %1, %0 \ n \ t" : "=r" (res) : "r" (a) );
     return res;
}

static inline  void parser_pxLoadContext(int a, int b){
     asm volatile ( "pxloadcontext %0, %1 \ n \ t" : : "r" (a), "r" (b) );
}
```

FIGURE 6.7 – Encapsulation des fonctions de pilotages

un processus léger rattaché à un flux réseau ouvert par celui-ci.

Cependant, créer un processus léger pour chaque nouvelle tâche à traiter peut très vite devenir consommateur de ressources en temps et en mémoire et ainsi diminuer les performances globales. En effet, la construction et la destruction d'un processus léger représente un surcout non négociable, d'autant plus si la tâche à laquelle ils sont affectés est courte.

Une autre alternative réside dans l'utilisation d'un réservoir de processus légers (pool de threads). L'ensemble des processus légers constituant le réservoir sont crées à l'initialisation de l'intergiciel Zebra, une bonne fois pour toute. A chaque nouvelle tâche, un processus léger est affecté à son traitement, de manière dynamique, via une fonction d'exécution (cf. Figure 6.8). Ainsi, une fois que le réservoir est crée, les processus légers sont en attente d'affectation. Lors de l'arrivée d'une tâche à traiter, ceux-ci sont informés et la tâche est affectée à un processus léger disponible. Lorsque la tâche est effectuée, le processus léger est libéré et redevient disponible pour une autre tâche. Cette architecture permet de ne pas avoir à créer un processus léger à chaque nouvelle tâche à traiter ou à en détruire un à chaque fois qu'une tâche est effectuée.

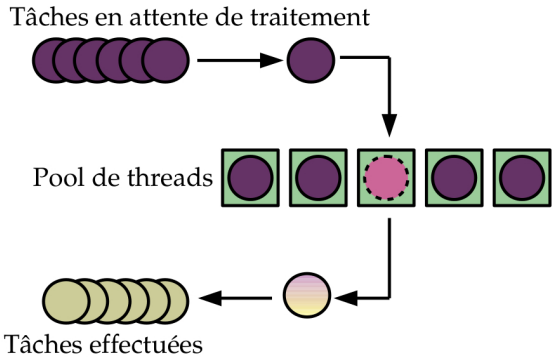


FIGURE 6.8 – Pool de threads

C'est ce modèle que nous avons choisi de mettre en oeuvre dans l'intergiciel Zebra. En effet, le création et la destruction des processus légers prendrait beaucoup trop de temps et de ressources par rapport au temps de traitement du flux qui est de l'ordre de quelques cycles d'horloges par caractère. Ainsi, un réservoir de processus légers, crée à l'initialisation de l'intergiciel Zebra est-il plus à même de répondre à ces problématiques.

Comme le décrit la figure 6.9, une liste des tâches à effectuer est maintenue à jour par le réservoir. Lorsqu'un client nécessite un traitement sur une socket, la tâche vient s'ajouer à la liste en cours et une notification est envoyée au réservoir afin qu'un processus léger soit affectéau traitement des paquets lus sur la socket réseau. Ce processus léger va ainsi s'occuper de l'envoi des caractères à traiter au coprocesseur, de la lecture des résultats, du remplissage de la vue et le cas échéant d'une éventuelle sauvegarde/-restauration de contexte.

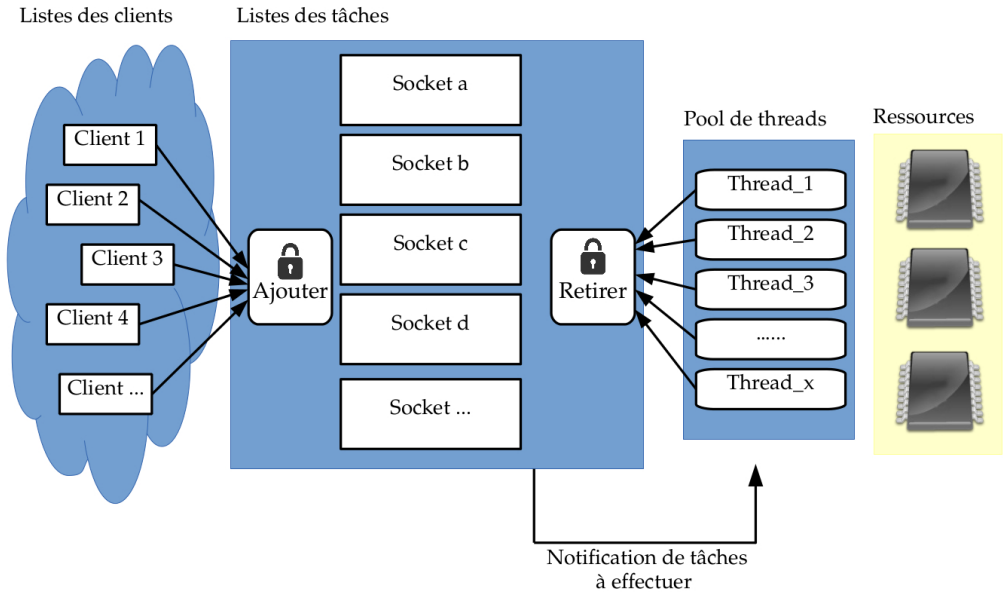


FIGURE 6.9 – Pool de threads pour la gestion des ressources matérielles

6.4 Flux d'éxecution

Une fois en cours d'exécution sur le processeur généraliste, le comportement de l'application est classique : elle se met en attente d'un flux d'informations provenant du réseau. Dans notre cas, nous présentons la suite de l'exécution sur la base du message protocolaire présenté à la figure 6.5.

Une fois ce message reçu par l'application, celle-ci utilise les fonctions fournies par l'intergiciel pour l'envoi de données au coprocesseur correspondant au protocole HTTP. Les données sont mises en attente dans la file d'attente d'entrée du coprocesseur, après une éventuelle restauration de contexte, dans le cas où le message ait été reçu par fragments précédemment (restauration conséquente à la sauvegarde préalable du contexte).

Le traitement se faisant de manière complètement asynchrone, l'application et donc le processeur généraliste, ne sont pas bloqués en attente des résultats du traitement des données par le coprocesseur.

Ce processus est décrit dans la figure 6.10 :

1. Lorsque l'application détecte un flux de données à traiter, elle l'envoie à l'intergiciel afin que celui-ci s'occupe des réservations de ressources adéquates.
2. L'intergiciel Zebra se charge, d'une part, d'affecter un processus léger au client et d'autre part de réserver un accélérateur matériel correspondant au protocole à traiter. Les données sont ensuite envoyées aux unités de pilotage.
3. Les unités de pilotage se chargent d'envoyer les caractères du flux à l'accélérateur matériel adéquat, qui effectue alors le traitement.
4. Au fur et à mesure du traitement, l'accélérateur matériel produit des couples de résultats (action,position) qui remontent jusqu'à l'intergiciel.

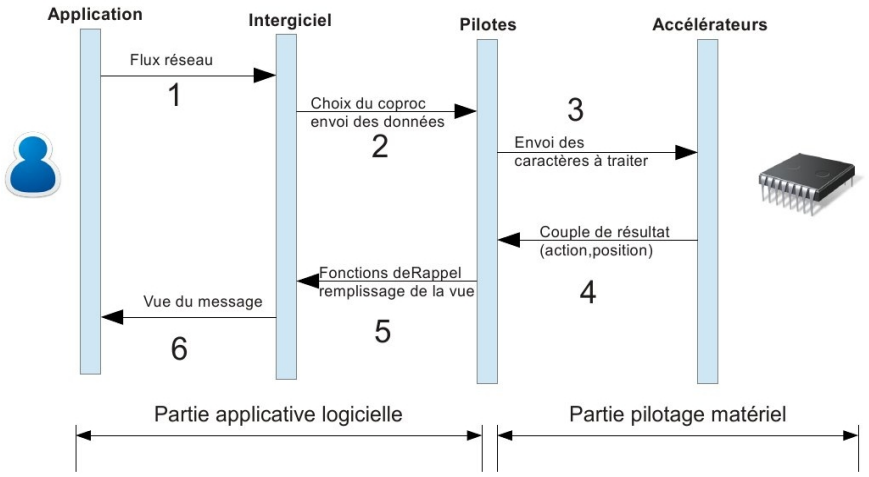


FIGURE 6.10 – Séquences de traitements

5. L'intergiciel se charge d'effectuer les copies mémoires nécessaire et remplit la vue du message au fur et à mesure.
6. Une fois que la vue du message est entièrement remplie, celle-ci est passée à l'application.

6.5 Bilan

Ce chapitre a décrit le flux d'exécution d'une application réseau basée les outils fournis par Zebra. Le développeur fournit l'application ainsi qu'une spécification en langage Zebra décrivant le protocole sur laquelle est basée l'application réseau. Cette spécification, une fois traitée par le compilateur Zebra, génère un coprocesseur matériel de traitement protocolaire qui est déployé sur la plateforme d'exécution. Elle génère également un ensemble d'éléments logiciels permettant à l'application de gérer les vues des messages.

Les données reçues par l'application à travers l'interface réseau sont envoyées au coprocesseur qui effectue le traitement protocolaire correspondant. Une fois le traitement effectué, le coprocesseur envoie les résultats au processeur généraliste. Sur la base de ces résultats, celui-ci remplit les champs correspondants dans la vue du message. L'ensemble de ce processus s'effectue de manière totalement asynchrone : les deux composants matériels sont totalement indépendants grâce aux files d'attente en entrée et en sortie.

Pour un traitement transparent et la possibilité de gérer un grand nombre de clients, les fonctionalités de gestion des coprocesseurs ont été déportées dans un intergiciel dédié. A cette fin, celui-ci intègre des capacités de sauvegarde et de restauration de contexte puisque les données reçues arrivent rarement dans l'ordre de leur émission. De plus, l'utilisation d'un intergiciel soulage le développeur des considérations bas niveau d'accès au matériel. Il a accès à une couche de support protocolaire de la même

manière qu'une couche de support logicielle, de manière ergonomique et transparente, avec des performances plus importantes en plus. Dans le prochain chapitre, nous proposons une évaluation de ces performances, en terme de consommation mémoire mais également en terme d'efficacité pure ou de taille de la couche de support.

7

Évaluation

LES TROIS DERNIERS CHAPITRES ont détaillé l'approche que nous proposons pour la conception conjointe logiciel-matériel de couches de supports protocolaire pour les applications réseaux. L'ensemble des messages manipulés transitent par ces couches qui représentent ainsi l'interface entre l'application réseau et le monde extérieur. Par conséquent, les performances globales de l'application réseau dépendent en grande partie des performances de la couche de ce support protocolaire.

Ce chapitre propose une évaluation de l'approche Zebra par rapport aux autres approches de développement de couches de support protocolaire.

Dans un premier temps, nous introduisons le contexte des expérimentations à travers une description de notre plateforme de prototypage. Dans un second temps, nous évaluons sur la même plateforme, plusieurs approches de développement de couches de support. Parmi celles-ci, nous retrouvons les approches développées manuellement, les approches génératives dédiées ainsi que les approches matérielles générées à partir d'outils de synthèse de haut niveau. Afin d'appuyer notre démarche, nous nous basons sur un pannel de protocoles largement déployés dans l'Internet afin de nous rapprocher le plus possible des conditions réelles de déploiement dans le contexte des systèmes embarqués.

Sommaire

7.1 Introduction

7.1.1 Procédure d'évaluation

Afin d'évaluer l'approche que nous avons proposé, nous nous sommes basés sur les besoins d'un développeur d'applications réseaux en termes de support protocolaire. Comme nous l'avons décrit dans les chapitres précédents, dans le contexte de systèmes embarqués disposant de ressources reconfigurables, le développeur dispose des approches de développement de supports protocolaires suivantes :

- Réutiliser des couches de support logicielles existantes et largement déployées.
- Se baser sur des approches logicielles dédiées afin d'être en phase avec les besoins spécifiques de son application.
- Dans certains cas, il lui est possible de s'aider d'outils de synthèse de haut niveau afin de transformer un algorithme décrit sous forme de code logiciel (en C, par exemple) en une description matérielle déployable sur les ressources matérielles reconfigurables, tel qu'une carte FPGA.

Par conséquent, nous avons suivi une démarche similaire dans le cadre de nos procédures d'évaluation. Pour une même requête, nous avons comparé les performances de traitement dans chacun des cas évoqués : l'application réseau utilise un support existant, écrit manuellement ; un support logiciel dédié généré et prenant en compte uniquement les besoins spécifiques de l'application et enfin, les composants de la couche de support générés par Zebra.

A cette fin, une application de *logging* a été développée. Elle récupère les vues des messages traités et, le cas échéant, permet d'afficher les champs dans un objectif de contrôle. Afin de comparer les résultats des traitements, cette application a été interfacée avec chacune des approches de développement de support protocolaire. Tout d'abord, nous l'avons couplé à une couche de support existante, développée manuellement et largement déployée sur l'Internet. Ensuite, nous avons généré une couche de support dédiée aux besoins spécifiques de l'application : la couche de support traite uniquement les champs qui lui sont nécessaires. Enfin, nous avons comparé les résultats obtenus avec la couche de support générée par l'approche Zebra que nous proposons.

7.1.2 Protocoles

Afin de mener nos expérimentations dans des conditions réelles de déploiement, nous avons fais le choix d'utiliser quatre protocoles répandus dans les applications réseaux contemporaines.

- Le protocole **HTTP** - *Hypertext Transfer Protocol* -, employé dans le domaine des serveurs web.
- Le protocole **SIP** - *Session Initiation Protocol* -, utilisé notamment dans les domaines de la communication audio et vidéo.
- Le protocole **SMTP** - *Simple Mail Transfer Protocol* -, utilisé dans le transfert de courrier éléctronique.
- Le protocole **RTSP**, - *Real Time Streaming Protocol* -, destiné aux systèmes l'accès aux contenus multimédias en ligne.

Ces protocoles sont des protocoles dit *à la HTTP* (*http-like*) car il s'agit de protocoles textuels basés sur HTTP. Les messages issus de ces protocoles ont en commun d'avoir une structure basée sur trois parties distinctes : une *commande*, un pannel d' *en-têtes* et enfin un *corps* de message.

— La commande permet d'identifier des éléments comme la version du protocole ou la méthode d'une requête.
— Une en-tête décrit un élément spécifique intrinsèque au protocole ainsi que la valeur associée. Par exemple, dans le cas de HTTP, la date, l'URI, le Content-Length sont des en-têtes. Elles peuvent également être composées de sous en-têtes.
— Enfin, le corps du message est formé par des éléments textuels utiles.

Ainsi, ces protocoles ont été utilisés dans leur domaine d'origine mais également dans des cas particuliers, en ne considérant que certains champs spécifiques. Ils ont été tronqués afin de montrer la faisabilité d'une approche qui ne prendrait en compte que certains éléments spécifiques du protocole pour se rapprocher le plus des besoins de l'application.

7.2 Plateforme d'expérimentations

7.2.1 Plateforme de développement matérielle

La plateforme de démonstration que nous avons concue est basée sur l'utilisation de la carte de développement *ML*605 de la société Xilinx^{TM}. Le circuit FPGA intégré sur cette carte est Xilinx Virtex-6 (modèle XC6VLX240T-1FFG1156). Comme le montre le schéma 7.1, cette carte intègre l'ensemble des composants nécessaires à la plateforme : une interface réseau permettant la communication avec le monde extérieur, un système de stockage afin de sauver le système d'exploitation ainsi que les binaires applicatifs, un module mémoire, d'un moniteur de debug et des interfaces d'entrée/sortie pour le contrôle de la carte.

Le coeur de processeur généraliste que nous avons sélectionné est le Leon-3 [Gai10]. Il s'agit d'un coeur de processeur 32*bits* développé par Jiri Gaisler. Il a été financé par l'Agence Spatiale Européenne, qui l'utilise dans ses satellites. Ce coeur de processeur a été déployé avec succès sur de nombreuses technologies FPGA et ASIC. Ce coeur de processeur est compatible avec les instructions de l'architecture Sparc V8 [SPA99]. Son architecture interne est de type RISC (*Reduced Instruction-set Computer*). Ce type d'architecture possède la caractéristique d'avoir un jeu d'instructions réduit et simple à déco-

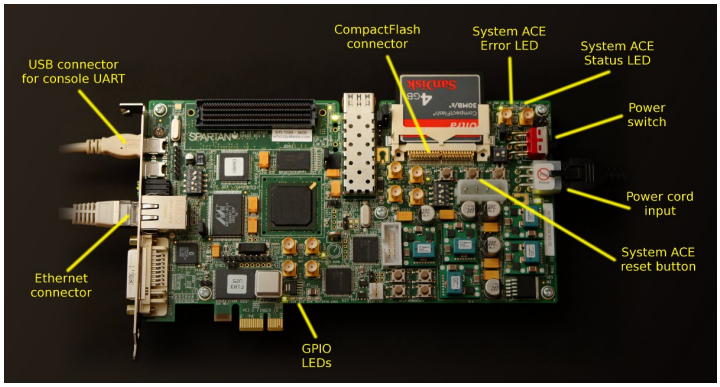


FIGURE 7.1 – Carte de développement Xilinx ML605 - *Source : Xillibus*

der, par opposition aux architectures de type CISC (*Complex Instruction-set Computer*). L'architecture interne du Léon utilise au total 7 niveau de *pipeline* lui permettant ainsi d'atteindre des fréquences de fonctionnement élevées (de $50MHz$ à $400MHz$ en fonction de la cible technologique). Ajouté à cela, il implémente également des techniques de prédictions de branchement lui offrant la possibilité de spéculer sur les résultats d'exécution des instructions. L'ensemble de ces éléments en font un coeur de processeur intéressant pour le bon déroulement de nos expérimentations.

De plus :

- le Léon-3 est un processeur open-source, il est donc possible d'en obtenir les codes sources afin de modifier son architecture interne ;
- le système fourni avec le processeur est compatible avec la carte de développement ML605 que nous avons à notre disposition. Ainsi, l'ensemble des composants éléctronqiues présents sur la carte de développement (mémoire DDR, interface Ethernet) sont supportés nativement.

Le jeu d'instructions du Leon-3 a été étendu afin de supporter l'intégration des coprocesseurs générés par Zebra. En effet, à l'origine, le *sparc-v8* possède 117 instructions, auxquelles ont été ajoutées 10 instructions dédiées pour le pilotage des coprocesseurs(5.7 du chapitre 5). Afin de pouvoir compiler des applications utilisant ce jeu d'instructions étendu, l'ensemble de la chaîne de compilation *GCC*4.4.1 (plus particulièrement *binutils*) a été mise à jour et recompilée en prenant en compte les nouvelles instructions intégrées au Leon-3.

La figure 7.2 schématise la plateforme concue. Elle inclut les composant nécessaires au bon fonctionnement du système, tel qu'un contrôlleur mémoire, une interface Ethernet et un deboggeur UART. L'ensemble du système sur puce a été cadencé, pour notre étude, à une fréquence de $60MHz$.

7.2.2 Composants logiciels déployés

Afin de mener à bien l'évaluation de l'approche proposée, un système d'exploitation basé sur un noyau Linux 2.6.32 a été déployé à l'aide de la bibliothèque GRLib [**?**] fournie par Gaisler. Elle regroupe l'ensemble des éléments nécessaires au déploiement d'applications sur des systèmes basés sur le Leon-3. Par exemple, elle fournit les pilotes

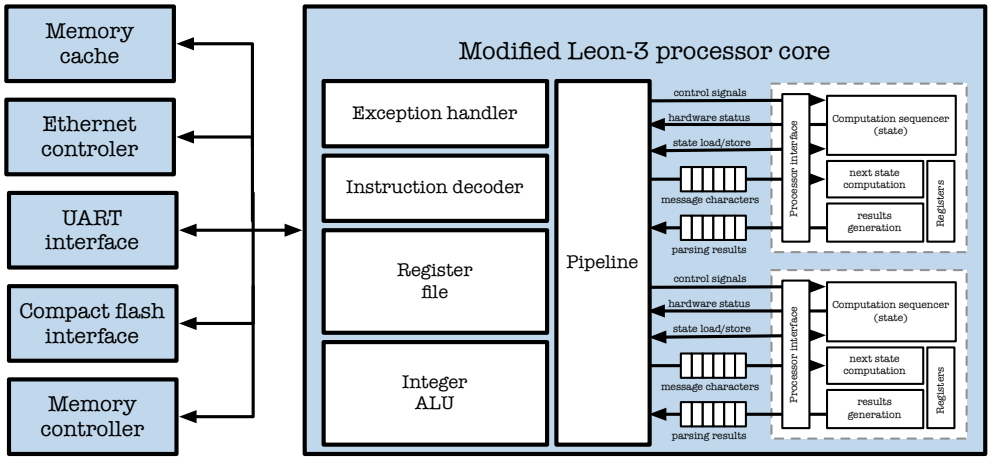


FIGURE 7.2 – Plateforme d'intégration du Leon-3 avec les coprocesseurs

de périphériques nécessaires au noyau Linux pour le pilotage de l'interface Ethernet.

Le système d'exploitation Linux a été cross-compilé à l'aide des outils fournis par Buildroot, ensemble de scripts et fichiers dédiés à la cross-compilation d'un système Linux complet pour des architectures embarquées.

L'application réseau de *logging* développée pour les expérimentations a été cross-compilée (avec GCC4.4.1) et exécutée sur ce système. Elle lit les trames réseaux, effectue un traitement syntaxique et affiche les résultats du traitement. Le temps de traitement des messages reçus sur l'interface Ethernet a été mesurée dans chacun des cas décrit précédemment.

7.3 Performances à l'exécution

7.3.1 Échantillons de messages

Dans le soucis de réaliser des expérimentations le plus proche possible des conditions réelles de traffic et ainsi mettre en exergue les avantages d'une solution basée sur une approche conjointe, nous avons effectué l'ensemble des tests en nous basant sur un pannel de messages réels. Il s'agit d'un ensemble de messages HTTP, SIP, SMTP et RTSP capturés sur les réseaux du laboratoire de recherche. La figure 7.3.1 résume les principales caractéristiques des messages en fonction du protocole.

La figure 7.3 montre un exemple de message HTTP réel sur lequel nous nous sommes basés pour notre évaluation. Il est composé d'une méthode POST et de quatre en-têtes que sont l'hôte, le type de contenu, la taille du contenu et l'action à réaliser et enfin du corps de la requête.

Par exemple, dans le cas du protocole HTTP, 515 messages ont été capturés. La taille de message minimum de l'échantillon est de 330 caractères, la taille maximum de 779 caractères. La moyenne est de 532 caractères et la médiane de 557. Ces caractéristiques nous permettent de dresser des résultats d'évaluation par caractère ou par message. En comparant les caractéristiques des messages, nous remarquons que la taille varie sui-

```
 POST /solanki/thesis.tex HTTP/1.1
Host: labri.fr
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.labri.fr/save"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
  <Enlighten xmlns="http://www.labri.fr/">
  <licenseID>string </licenseID>
  <content>string</content>
  <paramsXML>string </paramsXML>
  </Enlighten>
  </soap:Body>
</soap:Envelope>
```

FIGURE 7.3 – Exemple de requête HTTP

Protocole	#messages	Caractéristiques			
		Min	Max	Avg	Med
HTTP	515	330	779	532	557
RTSP	205	56	210	135	151
SIP	62	274	1357	627	582
SMTP	9378	6	1003	118	27

FIGURE 7.4 – Caractéristiques du pannel de messages

Specification	#d'états	Transitions (non factorisées)		
		#trans	Avg (trans/état)	Max (trans/état)
HTTP	66	3922	59.4	98
RTSP	56	2722	48.6	84
SIP	284	17451	61.4	119
SMTP	333	6974	20.9	86

FIGURE 7.5 – Caractéristiques des coprocesseurs HTTP - SIP - SMTP - RTSP

vant les protocoles utilisés. En effet, la taille des messages HTTP est sensiblement plus grande que celles des messages SMTP, RTSP et SIP. Par exemple, dans le cas de SMTP, les messages sont formés d'une commande et d'un corps. Ainsi, les messages SMTP sont courts, tandis que les messages HTTP, RTSP et SIP sont plus longs. Ces différences entre les protocoles impliquent également des caractéristiques différentes des coprocesseurs les décrivant et donc, une implantation différente.

7.3.2 Caractéristiques des coprocesseurs HTTP, SIP, SMTP et RTSP

Le compilateur Zebra a été utilisé pour générer les coprocesseurs correspondants aux quatres protocoles HTTP, SIP, SMTP et RTSP (ainsi que leurs fonctions de pilotages et les composants logiciels relatifs aux manipulations des vues de messages) à partir de spécifications Zebra. Les spécifications en langage Zebra font respectivement 97, 87, 128 et 80 lignes, pour les protocoles HTTP, RTSP, SIP et SMTP. La spécification Zebra pour le protocole HTTP est présentée dans la figure 6.4.

Afin de mettre en exergue la complexité des automates réalisant l'analyse protocolaire pour ces quatre protocoles, le tableau 7.3.2 fournit les principales caractéristiques des automates générés.

Pour le protocole SIP, l'automate ne compte pas moins de 284 états pour près de 17000 transitions au total. Les caractéristiques des coprocesseurs varient en fonction des protocoles (et de leurs spécifications). Les transitions ne sont pas factorisées, c'est à dire qu'il peut y avoir plusieurs transitions possible d'un état a vers un état b lorsque la condition $t_{(a,b)}$ est différente.

Afin d'affiner l'évaluation des performances, nous avons effectué deux types d'expérimentations : le premier pannel d'expérimentations est fait au niveau des coprocesseurs et le deuxième pannel est fait au niveau applicatif comme l'illustre les figure 5.5 et 6.10.

Au niveau des coprocesseurs, nous mesurons globalement le temps de traitement de l'automate. Ainsi, lorsqu'une requête est envoyé au coprocesseur, celui-ci effectue le traitement approprié et renvoie les couples *(action, position)* correspondants aux champs analysés. La frontière du traitement au niveau des coprocesseurs est représentée par la partie en bleue de la figure 5.5.

Au niveau applicatif(Fig. 6.10), le temps de traitement est mesuré au delà. Il inclut le remplissage de la vue, afin de fournir à l'application une structure sémantiquement utile afin que celle-ci puisse effectuer ses traitements. Ainsi, en plus du traitement de

l'automate au niveau du coprocesseur s'ajoute le temps de remplissage de la vue au niveau du processeur généraliste (entre les éléments rose et vert de la figure 5.5).

Les résultats de ces expérimentations sont présentées ci-après.

7.3.3 Performances de traitement au niveau des coprocesseurs

Dans cette section, nous mettons en exergue les performances obtenues par l'approche Zebra en les comparant aux autres approches de couches de traitement protocolaire. Ainsi, nous avons couplé l'application de référence à différentes couches de support et comparé les résultats obtenus au niveau des coprocesseurs pour le traitement des messages et au niveau applicatif après le remplissage de la vue du message, une fois le messages traité.

Dans un premier temps, nous nous sommes basés sur les couches de support protocolaire existantes, considérées comme des références dans leur domaine respectif. Nous avons utilisé pour HTTP, RTSP, SIP et SMTP les couches de support fournies respectivement par le serveur Apache [apa] et Cherokee [che], GSTreamer [rts], SER [Bon03] et Postfix [BdW00]. Les résultats ont été obtenu à partir de messages réels présentés dans le pannel de la figure 7.3.1.

Dans un second temps, nous avons généré une couche de support logicielle dédiée pour chacun de ces protocoles. A cette fin, nous nous sommes basés sur le projet Ragel [Thu06] afin de générer des spécifications protocolaires pour les besoins de notre application. Les couches générées par l'outil sont décrites en langage C. Nous avons utilisé le backend logiciel optimisant le temps de traitement.

Enfin, dans un troisième temps, nous avons comparé les résultats de ces deux approches avec les résultats obtenus avec Zebra.

Les figures 7.6 et 7.7 reprennent l'ensemble des résultats que nous avons obtenus pour chacun des protocoles. La taille des messages est donnée en nombre de caractères et les temps de traitement sont en nombre de cycles d'horloge. Pour mesurer le temps nécessaire à l'analyse des messages, nous avons exploité un registre spécial du processeur, ce registre compte le nombre de cycles d'horloges ecoulés depuis le démarrage du système.

		Legacy parser			Dedicated parser		
		Min	Max	Med	Min	Max	Med
HTTP	Size	428	437	557	428	437	557
	Time	183476	227678	231320	14050	14053	23274
	Avg		451.8			36.1	
	Factor	**9**	**14**	**13**	**1**	**1**	**1**
RTSP	Size	56	210	151	56	210	151
	Time	168802	132726	145920	3098	5456	6941
	Avg		1072.9			37.1	
	Factor	**34**	**39**	**25**	**1**	**1**	**1**
SIP	Size	274	1244	581	274	1244	581
	Time	1286710	2955595	2779204	9766	18637	16996
	Avg		3345.1			25.6	
	Factor	**149**	**185**	**179**	**1**	**1**	**1**
SMTP	Size	8	420	202	8	420	202
	Time	17162	650767	346034	2093	5482	3683
	Avg		1609.5			17.9	
	Factor	**15**	**206**	**123**	**1**	**1**	**1**

Taille en nombre de caractères ; mesures en cycles CPU.

FIGURE 7.6 – Comparaisons entre les couches existantes et les approches logicielles dédiées

Nous avons effectué la comparaison entre les couches existantes et les couches logicielles dédiées en prenant comme référence les couches logicielles dédiées. Par exemple, dans le cas du protocole RTSP, le message le plus court d'une longueur de 56 caractères est traité en 168802 cycles d'horloges, ce qui représente un facteur neuf fois plus lent que le même message traité par une couche logicielle dédiée, en 14050 cycles. Un message d'une taille médiane de 151 caractères est traité en 145920 cycles d'horloges ce qui est 25 fois plus lent que le même message traité par une couche dédiée en 6941 cycles. Cette analyse se généralise sur les quatres protocoles que nous avons testés.

Le premier constat que nous pouvons donner est celui de la rapidité d'exécution : dans l'ensemble des cas testés, sur l'ensemble des messages, les couches les plus lentes en temps d'exécution sont les couches de support existantes, au profit de l'approche logicielle dédiée. En effet, les couches existantes supportent l'ensemble du protocole considéré. Par exemple, la couche de support pour le protocole SIP, extraite de SER, supporte l'ensemble du protocole SIP. L'application utilisant cette couche ne nécessite parfois pas le support de tout le protocole. A l'inverse, une approche de support dédiée permet de générer une couche pour les besoins spécifiques de l'application : seul le support pour les champs nécessaires est généré. Par exemple, la spécification Zebra que nous avons développée pour le protocole SIP comporte 128 lignes de code pour le traitement de 6 en-têtes. A titre de comparaison, dans le routeur SER, le support

protocolaire pour SIP supporte l'intégralité du protocole et est composé de près de 6500 lignes de code.

		Dedicated parser			Zebra-based parser		
		Min	Max	Med	Min	Max	Med
HTTP	Size	428	437	557	428	437	557
	Time	14050	14053	23274	1291	1317	1975
	Avg		36.1			3.2	
	Factor	**10**	**12**	**11**	**1**	**1**	**1**
RTSP	Size	56	210	151	56	210	151
	Time	3098	5456	6941	537	741	848
	Avg		37.1			5.1	
	Factor	**5**	**8**	**7**	**1**	**1**	**1**
SIP	Size	274	1244	581	274	1244	581
	Time	9766	18637	16996	1001	1824	1548
	Avg		25.6			2.4	
	Factor	**9**	**11**	**10**	**1**	**1**	**1**
SMTP	Size	8	420	202	8	420	202
	Time	2093	5482	3683	1134	1920	1522
	Avg		17.9			9.8	
	Factor	**1**	**2**	**2**	**1**	**1**	**1**

Taille en nombre de caractères ; mesures en cycles CPU.

FIGURE 7.7 – Comparaisons entre les approches logicielles dédiées et l'approche Zebra

Nous observons ainsi un gain d'un facteur 10 dans le cas du protocole HTTP jusqu'à près de 200 dans le cas du protocole SMTP. Les différences que nous avons constatés sur la taille des messages capturés reflètent les résultats que nous obtenons : le traitement des protocoles SIP, SMTP et RTSP est au moins une dizaine de fois plus rapide que celui de HTTP. De plus dans le cas de SMTP, les messages courts favorisent un traitement rapide puisque qu'il est de 123 fois plus performant.

Dans un deuxième temps, la comparaison entre l'approche logicielle dédiée et l'approche conjointe logiciel-matériel dédiée fournie par Zebra a été réalisée. Cette fois ci, Zebra a été choisie comme référence. Cette comparaison met en évidence de meilleures performances pour l'approche proposée. Ainsi, dans le cas du protocole SIP, un message d'une taille médiane de 581 caractères est traité en 16996 cycles d'horloge ce qui représente un temps de traitement 10 fois plus lent que le même message traité par Zebra en 1548 cycles.

Dans le cas du protocole SMTP, l'usage de coprocesseur matériels permet de réduire le temps de traitement des messages d'un facteur 2 au minimum. En effet, le temps d'exécution dépend à la fois du nombre de champs traités et de la longueur des messages. Ainsi, le temps nécessaire à l'initialisation et la gestion des coprocesseurs devient

négligeable devant la taille des messages qu'il faut traiter. Ceci explique notamment les moins bonnes performances obtenues dans le cas du protocole SMTP qui traite un grand nombre de messages dont la taille est inférieure à une centaine de caractères.

Ainsi, De manière plus globale, nous observons de meilleures performances dans le cas de l'approche Zebra par rapport à une approche logicielle dédiée, et par conséquent, d'autant plus par rapport aux approches existantes développées manuellement. Ainsi, dans le cas des protocoles HTTP, RTSP et SIP, le gain obtenu avoisinne d'un facteur 100 en moyenne sur l'ensemble des messages testés par rapport aux approches existantes.

7.3.4 Performances de traitement au niveau applicatif

Au niveau applicatif, une fois le message traité et la récupération du couple de résultat *action-position* effectuée, la vue du message est remplie et passée à l'application. Cette vue est remplie au fur et à mesure, par copie mémoire des champs qui ont été traités dans le flux. Cette copie est assurée par les fonctions de rappel invoquées lors de la détection de la fin du champ à l'aide de l'identifiant renvoyé par le coprocesseur (numéro d'action). Nous mesurons, dans un second temps, les performances obtenues au niveau applicatif après le remplissage de cette vue. Le tableau 7.8 reprend les résultats obtenus.

		Analyseurs logiciels			Analyseurs conjoints		
		Min	Max	Med	Min	Max	Med
HTTP	Size	428	437	557	428	437	557
	Time	15345	28449	17043	3566	6946	3828
	Avg		39			4	
	Factor	**3.70**	**4.47**	**4.80**	**1**	**1**	**1**
RTSP	Size	56	210	151	56	210	151
	Time	4165	9580	8030	1512	2723	2477
	Avg		53.18			3.30	
	Factor	**2.49**	**3.85**	**3.30**	**1**	**1**	**1**
SIP	Size	274	1244	581	274	1244	581
	Time	14804	30779	23606	3870	8512	5699
	Avg		40.63			9.81	
	Factor	**3.13**	**4.46**	**4.13**	**1**	**1**	**1**
SMTP	Size	8	420	202	8	420	202
	Time	141	8422	5151	116	2090	1838
	Avg		25.5			9.1	
	Factor	**1.5**	**5.5**	**3.7**	**1**	**1**	**1**

Taille en nombre de caractères ; mesures en cycles CPU.

FIGURE 7.8 – Performances de traitements des messages au niveau application

A spécification identique (automates identiques), les gains obtenus sont inférieurs à

ceux obtenus précédemment au niveau *driver*. Cette perte tient du fait des copies mémoire effectuées lors du remplissage de la vue qui ne sont pas optimisées. En effet, indépedamment de la couche de support protocolaire utilisée, l'allocation de mémoire ainsi que la copie mémoire des champs traités consomme des ressources au niveau du processeur. Cette consommation est repercutée sur les performances globales de l'application. De plus, comme nous l'avons mentionné précédemment, la couche de support protocolaire consomme environ 90% du temps de traitement global. Or, l'optimisation de cette couche rend les 10% restant non-négligeables et force donc à optimiser ces copies mémoire. Toutefois, comme nous l'avons fait remarquer, cette copie mémoire se fait indépendamment de la couche de support utilisée. Ainsi, entre une approche logicielle dédiée et une approche conjointe logiciel-matériel dédiée comme Zebra, nous observons un facteur de gain, variant en fonction du protocole, de 3 à 4 au profit de Zebra.

Ces différentes évaluations montrent que les coprocesseurs et le code générés par l'approche Zebra est plus performante que d'autres approches logicielles dédiées ainsi que des couches de support existantes.En contre partie de ce gain de performances s'ajoute un coût en terme d'occuptation du FPGA.

7.4 Coût matériel de l'approche

L'approche que nous présentons permet de déporter l'exécution d'un analyseur syntaxique vers un coprocesseur. Ce déport implique l'ajout d'un coprocesseur dans le système et par conséquent, un coût matériel (silicium) non nul. Le tableau 7.9 met en exergue le coût relatif en ressources FPGA supplémentaires utilisée par les coprocesseurs que nous avons déployés. Ce coût relatif est proportionnel au coprocesseur.

HTTP	3%
SMTP	7%
SIP	12%
RTSP	4%
Processeur	3%

FIGURE 7.9 – Occupation du FPGA par le processeur et les coprocesseurs

Ainsi, le Leon-3 occupe près de 30% des ressources disponibles sur le FPGA. Dans le cas du protocole HTTP, le coprocesseur nécessite 3% de ressources supplémentaires. Dans le cas d'un traitement HTTP parallèle nécessaitant plusieurs coprocesseurs HTTP, il est donc possible de déployer 33 coprocesseurs HTTP sur les 66% de ressources FPGA disponibles.

En plus des performances à l'exécution, dans le cadre de notre évaluation, nous avons mesuré l'emprunte mémoire statique des analyseurs syntaxiques des différentes approches. La figure 7.10 présente les résultats obtenus.

	Support existant	Support dédié	Zebra
HTTP	188.2	13.0	1.8
SMTP	457.7	81.4	1.8
SIP	183.5	31.1	1.8
RTSP	241.8	12.6	1.8

FIGURE 7.10 – Taille des analyseurs compilés, en Ko

Nous avons mesuré la mémoire statique nécessaire (coût du programme et mémoire statique) à notre application de test, une fois compilée pour le processeur Leon-3. Ceci inclut les librairies du systèmes d'exploitation Linux, la couche de support protocolaire utilisée ainsi que la logique applicative.

La version de l'application qui nécessite le plus de ressources en mémoire statique est celle basée sur les couches de support protocolaire existantes. De part leur support total des protocoles, ces couches nécessitent plus de code logiciel que des approches dédiées qui génèrent un support spécifique pour les besoins de l'application considérée. Par exemple, dans le cas du protocole SMTP, la version de l'application basée sur la couche de support extraite du serveur Postfix nécessite $457Ko$ tandis que son homologue basée sur une approche logicielle dédiée n'en nécessite que $81Ko$.

La version de l'application basée sur Zebra ne nécessite que $1.8Ko$ pour l'ensemble des protocoles. En effet, la complexité de traitement de la couche de support est déportée sur les coprocesseurs matériels, la sous-couche logicielle ne représentant que les fonctions de pilotage nécessaires à la gestion de ces derniers. Cependant l'approche Zebra est consommatrice de ressources matérielles.

7.5 Zebra face aux outils de synthèse de haut niveau

La synthèse de haut niveau (HLS) (nommée aussi synthèse comportementale) et les outils associés permettent permet de générer un coprocesseur matériel à partir d'un algorithme écrit dans un langage séquentiel de plus haut niveau. Afin d'évaluer la pertinence de générer nos coprocesseurs à partir de spécifications Zebra plutot que d'utiliser ce type d'appraoches à partir de description C des analyseurs syntaxiques, nous avons étudié la qualité des coprocesseurs générés.

A cette fin, nous avons évalué cinq outils de synthèse de haut niveau : GAUT [CCB+08], LegUP [CCA+11], Xilinx Vivaldo [Xil12] fourni par Xilinx, C to Verilog [BAMR10] et eXcite [Y E10]. Ces évaluations ont été réalisées en utilisant le code de couche de support en C généré par l'outil Ragel [Thu06].

Les protocoles sont décrits à l'aide d'automates dans les couches de support. Il existe plusieurs moyens d'implémenter des automates en langage C. Par exemple, l'outil Ragel fournit cinq méthodes de génération d'automates. Toutefois, globalement, l'ensemble de ces méthodes utilisent des fonctionalités avancées de la programmation en C telles que l'arithmétique des pointeurs, les sauts conditionnels ou encore les appels de fonctions imbriqués. Du fait de l'utilisation de ces fonctionalités, l'ensemble des outils de HLS que nous avons évalués n'ont pas été en mesure de générer une description

matérielle du code source en C. La figure 7.11 regroupe les échecs et succès des générations.

Outil HLS	Compilation	Synthèse
GAUT	Échec	Échec
GraphLab	Échec	Échec
LegUP 2.0	Succès	Succès
Xilinx Vivado	Échec	Échec
ExCite 5.0c	Succès	Échec
C 2 Verilog	Succès	Échec

FIGURE 7.11 – Echecs et succès des évaluations des outils de HLS

Ainsi, par exemple, les outils GAUT et Vivaldo n'intègrent pas le support du mot-clé *goto*, largement utilisé dans l'implémentation d'automates en C. A l'inverse, les outils tels que LegUP, C 2 Verilog et ExCite supportent l'ensemble des fonctionalités nécessaires pour l'implémentation d'automates en langage C. Toutefois, l'outil C 2 Verilog n'a pas été en mesure de synthétiser les coprocesseurs et générer la description matérielle correspondante alors que la compilation du code C a bien réussi. De même, l'outil Excite accepte la compilation du code C mais ne parvient pas à synthétiser les coprocesseurs dû à un crash au milieu du processus de génération. Seul l'outil LegUP est parvenu à compiler, puis à synthétiser les coprocesseurs correspondants. De ce fait, nous présentons la comparaison entre Zebra et les coprocesseurs générés par l'outil LegUP.

La figure 7.5 regroupe les performances ainsi que l'utilisation des ressources des deux approches LegUP et Zebra. Il est à noter que les coûts matériels rapportés sont pour un FPGA de type Stratix 4 de chez ALTERA. En effet, les coprocesseurs générés par Leg-Up ne sont pas génériques et nous ont obligé, pour une comparaison équitable, à évaluer nos coprocesseurs sur une autre famille de FPGA à travers une chaine de synthèse différente (Altera Quartus II).

Les résultats fournis dans la figure 7.5, pour une même cible technologique, nous permettent de remarquer que les coprocesseurs générés en utilisant l'outil LegUP nécessitent 6 à 10 fois plus de ressources que ceux de l'approche Zebra. Cette consommation supplémentaire de ressources implique une fréquence de fonctionnement moindre des coprocesseurs générés par LegUP. Un autre point de comparaison entre les deux approches est celui du temps de traitement nécessaire. En effet, l'approche LegUP nécessite de 2 à 4 fois plus de cycles d'horloge pour le traitement d'un unique caractère, ce qui rend l'approche Zebra plus performante en terme de débit.

Protocol	Coprocesseurs générés par LegUP				Coprocesseurs générés par Zebra			
	ALUT	REGs	Freq. (MHz)	Avg (cycle/char)	ALUT	REGs	Freq. (MHz)	Avg (cycle/char)
HTTP	7771	1040	25	9.40	1021	466	240	2.20
RTSP	5806	925	35	9.00	894	504	225	2.11
SIP	36847	2930	9	8.34	3970	1330	174	2.11
SMTP	29894	2559	9	5.05	2827	1516	183	2.06

FIGURE 7.12 – Comparaison des approches Zebra et LegUP

Ces observations s'expliquent, en partie, par la nature intrinsèque des outils de synthèse de haut niveau. En effet, ces outils, tel que LegUP, ont été développés spécifiquement pour des applications de types traitement du signal et des images. Ces dernières possèdent peut de structures conditionnelles par rapport au nombres d'opérations arithmétiques à réaliser. Les applications de support protocolaire, tel que celle présentée dans notre approche, sont des applications de type contrôle dont les propriétés sont à l'opposé des précédentes.

Cette évaluation met en exergue les performances des coprocesseurs matériels générés par l'approche Zebra vis à vis des approches alternatives.

7.6 Bilan

Dans ce chapitre, nous avons proposé une évaluation de l'approche Zebra en la confrontant à des messages tirés d'échanges réels.

Dans un premier temps, nous avons comparé l'approche Zebra avec les approches logicielles généralistes et dédiées en termes de performances (temps de traitement et ressources nécessaires). Cette comparaison nous a permi de démontrer que l'approche conjointe logiciel-matériel développée est plus performante que ses homologues purement logiciels.

Dans un second temps, nous nous sommes attachés à démontrer les performances des coprocesseurs matériels générés. Pour cela nous avons comparé nos coprocesseurs à ceux obtenus à l'aide d'approches alternatives. Ces approches alternatives, basées sur l'utilisation d'outils de synthèse de haut niveau, ont, pour une grande partie d'entre elles, été incapables de générer les coprocesseurs matériels à partir des spécifications. Seul l'outil LegUp a été en mesure de produire des architectures matérielles exploitables. Toutefois, les performances de ces architectures matérielles sont très en retrait vis à vis de celles générées par Zebra.

L'ensemble de ces deux évaluations de l'approche Zebra nous a permi de démonter son efficacité pour le traitement des messages protocolaires dans les applications réseau embarquées.

8

Conclusion

LA MONTEE EN PUISSANCE du réseau Internet et des systèmes mobiles communicants conjugués à la miniaturisation des équipements de communications réseaux ont provoqué l'apparition de nouvelles méthodes de consommation de l'information. La consommation de l'information se fait actuellement à travers des services de plus en plus riches accessibles à tout moment. Ces services sont rendus en fonction de l'environnement dans lequel se trouvent les utilisateurs. Ces fonctionnalités sont fournies par des *applications réseaux.*

Les applications réseau se divise généralement en deux parties distinctes : une couche de *logique applicative* fournissant les services à l'utilisateur et une couche de *support protocolaire* chargée de la gestion et la manipulation des messages issus des communications avec le réseau. La couche de support protocolaire représente ainsi la frontière entre la logique applicative et le monde extérieur. A ce titre, elle représente un composant critique de l'application. Les performances globales de l'application sont alors directement liées aux performances de la couche de support protocolaire. Les systèmes sur puce actuels intègrent au sein d'un même circuit un processeur et des accélérateurs matériels que les applicatifs peuvent exploiter afin d'accélérer le traitement de certaines tâches. Toutefois, les couches de support protocolaire sont aujourd'hui uniquement implantées de manière logicielle et peu d'approches proposent de tirer parti d'accélérateurs matériels pour les taches de type analyse protocolaire. Les approches proposées dans la littérature se focalisent sur l'accélération de l'analyse des flux de type XML sans prendre en compte la relation avec l'applicatif exploitant les informations extraites du flux.

8.1 Contributions

Dans le cadre de nos travaux, nous avons présenté dans ce manuscrit une approche innovante basée sur une implantation conjointe des applications réseau au sein de systèmes sur puce. Cette approche conjointe logicielle-matérielle repose d'une part sur la conception d'un système sur puce et d'autre part sur une architecture logicielle rendant

le développement du support protocolaire accessible au programmeur. La plateforme matérielle intègre un processeur généraliste pour le déploiement d'applications logicielles ainsi que des accélérateurs matériels dédiés à l'accélération des tâches d'analyses des flux réseaux. Les accélérateurs matériels sont générés et le système sur puce est configuré à partir des descriptions des protocoles à supporter. Un langage dédié dont la syntaxe est proche de celle utilisée pour la spécification des protocoles a été développé. Un intergiciel rendant la gestion des accélérateurs matériels transparente pour les applicatifs réseaux a de plus été développé.

Cette transparence permet l'exploitation facile et efficace de l'architecture matérielle par un développeur d'applications réseau.

Afin de valider notre approche, nous avons conçu un système sur puce sur une plateforme FPGA à des fins de protoypage. Au sein de ce circuit FPGA, nous avons implanté un processeur généraliste de type LEON-3 et différents accélérateurs matériels. Cette plateforme intègre en plus un ensemble de composants nécessaires à l'exécution d'applicatifs réseau : contrôleur mémoire, interface Ethernet, interfaces d'entrée/sortie.

En adéquation avec ce prototype fonctionnant sur circuit FPGA, nous avons défini un processus d'évaluation complet de notre approche. L'évaluation s'est focalisée sur les performances des couches de support protocolaires accélérées matériellement vis à vis des approches traditionnelles. Dans un premier temps, nous avons comparé l'efficacité de notre approche avec celle d'approches logicielles existantes. Les résultats obtenus ont démontré que notre approche accélère l'exécution des tâches d'analyse protocolaire d'un facteur 4 à 11 (2 pour l'analyse du protocole SMTP et 11 pour le protocole HTTP). Les gains au niveau applicatif sont légèrement inférieurs. En effet, le facteur d'accélération est de 3.7 pour le protocole SMTP et de 4.80 pour protocole HTTP. Dans un second temps, nous avons évalué la qualité des accélérateurs matériels générés. Cette comparaison s'est basée sur des outils de synthèse de haut niveau qui permettent de générer des accélérateurs matériels à partir d'algorithmes décrits en langage C. Les outils que nous avons évalués n'ont pas été en mesure de générer un coprocesseur, à spécification d'entrée égale, excepté l'outil LegUP dont les coprocesseurs générés consomment 6 à 10 fois plus de ressources matérielles que ceux générés par l'approche Zebra. L'ensemble de ces expérimentations a démontré la faisabilité et l'intérêt de l'approche proposée.

8.2 Perspectives

Ce manuscrit dresse la faisabilité d'une approche conjointe pour le développement du support protocolaire d'applications réseaux. Au cours de nos travaux, nous avons recensé deux types d'approches à ce domaine : les approches logicielles telles que Zebu [BRLM07], GAPAL [BBW07] ou Binpac[PPSP06] et les approches majoritairement matérielles telles que celles proposées par Mitra *et. al* pour le traitement XML [MVB+09] ou Moscola *et. al* pour le routage [MLLP03].

Les perspectives des travaux présentés s'articulent autour de l'affinement de l'approche que nous proposons. Lors de la conception de notre plateforme, nous nous sommes attachés à garantir une efficacité du traitement des requêtes. Une prochaine direction consiste à forger les réponses à l'aide des coprocesseurs : fournir une vue du

message à l'intergiciel afin que celui-ci puisse construire les réponses aux requêtes des clients.

Une deuxième perspective découle de l'architecture intrinsèque de la plateforme matérielle. En effet, lors du déploiement des coprocesseurs, le fichier *bitstream* servant à programmer la carte FPGA inclue les coprocesseurs dédiés à différents protocoles. Lors d'un changement de coprocesseur (et donc de protocole), il est nécessaire de re-programmer la plateforme afin qu'elle puisse prendre en compte ce nouveau protocole. Nous avons dore et déà commencé à explorer cette direction dont les premiers résultats sont prometteurs : la capacité à pouvoir reprogrammer à la volée une partie de la carte afin d'avoir une plateforme dynamique s'adapatant aux nécessités des clients.

Bibliographie

[apa] « Postfix SMTP Server ».

[Ash08] Peter J. ASHENDEN. *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 édition, 2008.

[Bac02] Godmar BACK. « DataScript - A Specification and Scripting Language for Binary Data ». Dans *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 66–77, London, UK, UK, 2002. Springer-Verlag.

[BAMR10] Yosi BEN-ASHER, Danny MEISLER, et Nadav ROTEM. « Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs ». *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 3(3) :15 :1–15 :19, 2010.

[BBW07] Nikita BORISOV, David J. BRUMLEY, et Helen J. WANG. « A Generic Application-Level Protocol Analyzer and its Language ». Dans *In 14th Annual Network &amp ; Distributed System Security Symposium*, 2007.

[BdW00] Mick BAUER et Brenno de WINTER. « Using Postfix for Secure SMTP Gateways ». *Linux J.*, 2000(78es), octobre 2000.

[BLC95] T. BERNERS-LEE et D. CONNOLLY. « RFC 1866 : Hypertext Markup Language — 2.0 », novembre 1995. Status : PROPOSED STANDARD.

[BLFF96] T. BERNERS-LEE, R. FIELDING, et H. FRYSTYK. « RFC 1945 : Hypertext Transfer Protocol — HTTP/1.0 », mai 1996. Status : INFORMATIONAL.

[Bon03] O. BONAVENTURE. « SIP express router (SER) [Book Review] ». *Network, IEEE*, 17(4) :9–9, 2003.

[BPSM+08] Tim BRAY, Jean PAOLI, C. Michael SPERBERG-MCQUEEN, Eve MALER, et François YERGEAU. « Extensible Markup Language (XML) 1.0 (Fifth Edition) ». World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[BR99] Vaughn BETZ et Jonathan ROSE. « FPGA routing architecture : segmentation and buffering to optimize speed and density ». Dans *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, FPGA '99, pages 59–68, New York, NY, USA, 1999. ACM.

[BRLM07] Laurent BURGY, Laurent RÉVEILLÈRE, L. LAWALL, Julia, et Gilles MULLER. « A Language-Based Approach for Improving the Robustness of Network

Application Protocol Implementations ». Research Report RR-6167, INRIA, 2007.

[BRLM09] Yérom-David BROMBERG, Laurent RÉVEILLÈRE, Julia L. LAWALL, et Gilles MULLER. « Automatic generation of network protocol gateways ». Dans *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 2 :1–2 :20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[BRLM11] Laurent BURGY, Laurent RÉVEILLÈRE, Julia LAWALL, L., et G. MULLER. « Zebu : A Language-Based Approach for Network Protocol Message Processing ». *IEEE Transactions on Software Engineering*, 37(4) :575–591, 2011.

[Cam01] Patterson CAMERON. « High Performance DES Encryption in Virtex(tm) FPGAs Using Jbits(tm) ». Dans *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 113–, Washington, DC, USA, 2001. IEEE Computer Society.

[CB74] Donald D. CHAMBERLIN et Raymond F. BOYCE. « SEQUEL : A structured English query language ». Dans *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[CBM04] Vlad COROAMA, Jürgen BOHN, et Friedemann MATTERN. « Living in a smart environment - implications for the coming ubiquitous information society ». Dans *SMC (6)*, pages 5633–5638. IEEE, 2004.

[CC97] Steve CONNER et Diane CONNER. *Programmer's Guide to the NCP (NetWare Core Protocol)*. Annabooks, 1997.

[CCA+11] Andrew CANIS, Jongsok CHOI, Mark ALDHAM, Victor ZHANG, Ahmed KAMMOONA, Jason H. ANDERSON, Stephen BROWN, et Tomasz CZAJKOWSKI. « LegUp : high-level synthesis for FPGA-based processor/accelerator systems ». Dans *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 33–36, New York, NY, USA, 2011. ACM.

[CCB+08] Philippe COUSSY, Cyrille CHAVET, Pierre BOMEL, Dominique HELLER, Eric SENN, et Eric MARTIN. GAUT : A High-Level Synthesis Tool for DSP Applications. Dans Philippe COUSSY et Adam MORAWIEC, éditeurs, *High-Level Synthesis*, pages 147–169. Springer Netherlands, 2008.

[CE00] Krzysztof CZARNECKI et Ulrich W. EISENECKER. *Generative programming : methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CE02] M. CORTES et J. R. ENSOR. « Narnia : A virtual machine for multimedia communication services ». Dans *Proceedings of the Fourth International Symposium on Multimedia Software Engineering*, pages 246–254, 2002.

[CEE04] M. CORTES, J.R. ENSOR, et J.O. ESTEBAN. « On SIP Performance ». Rapport Technique, Bell Labs Technical Journal 3, 2004.

[che] « Cherokee Web Server ».

[CM] Satish CHANDRA et Peter J MCCANN. « Packet Types ».

[CO97] D. CROCKER, ED. et P. OVERELL. « RFC 2234 : Augmented BNF for Syntax Specifications : ABNF », novembre 1997. Status : PROPOSED STANDARD.

[Cor] Sony CORPORATION. « Virtual Mobile Engine (VME) ».

[Cri94] M. CRISPIN. « Internet Message Access Protocol - Version 4 ». RFC 1730 (Proposed Standard), décembre 1994. Obsoleted by RFCs 2060, 2061.

[CRW01] Antonio CARZANIGA, David S. ROSENBLUM, et Alexander L. WOLF. « Design and evaluation of a wide-area event notification service ». *ACM Trans. Comput. Syst.*, 19(3) :332–383, août 2001.

[DAF+03] Yanlei DIAO, Mehmet ALTINEL, Michael J. FRANKLIN, Hao ZHANG, et Peter FISCHER. « Path sharing and predicate evaluation for high-performance XML filtering ». *ACM Trans. Database Syst.*, 28(4) :467–516, décembre 2003.

[Est02] G. ESTRIN. « Reconfigurable computer origins : the UCLA fixed-plus-variable (F+V) structure computer ». *Annals of the History of Computing, IEEE*, 24(4) :3–9, 2002.

[FG05] Kathleen FISHER et Robert GRUBER. « PADS : a domain-specific language for processing ad hoc data ». Dans *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[FGM+99] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, et T. BERNERS-LEE. « Hypertext Transfer Protocol – HTTP/1.1 ». RFC 2616 (Draft Standard), juin 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[FK05] Felix FUENTES et Dulal C. KAR. « Ethereal vs. Tcpdump : a comparative study on packet sniffing tools for educational purpose ». *J. Comput. Sci. Coll.*, 20(4) :169–176, avril 2005.

[FKM+04] Anja FELDMANN, Nils KAMMENHUBER, Olaf MAENNEL, Bruce MAGGS, Roberto DE PRISCO, et Ravi SUNDARAM. « A methodology for estimating interdomain web traffic demand ». Dans *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 322–335, New York, NY, USA, 2004. ACM.

[Fow10] Martin FOWLER. *Domain Specific Languages*. Addison-Wesley Professional, 1st édition, 2010.

[Gai10] GAISLER RESEARCH. « *GRLIB IP Library User's Manual* », 2010.

[Hai02] Jed HAILE. « An Introduction To Gateway Intrusion Detection Systems - Hogwash GIDS », 2002.

[Hau00] John Reid HAUSER. « Augmenting a Microprocessor with Reconfigurable Hardware ». Rapport Technique, BRASS, 2000.

[HFC02] B.L. HUTCHINGS, R. FRANKLIN, et D. CARVER. « Assisting network intrusion detection with reconfigurable hardware ». Dans *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, pages 111–120, 2002.

[HFS+03] Michael HUCKA, Andrew FINNEY, Herbert M SAURO, Hamid BOLOURI, John C DOYLE, Hiroaki KITANO, Adam P ARKIN, Benjamin J BORNSTEIN, Dennis BRAY, Athel CORNISH-BOWDEN, et OTHERS. « The systems biology markup language (SBML) : a medium for representation and exchange of biochemical network models ». *Bioinformatics*, 19(4) :524–531, 2003.

[HO04] Nataliya HRISTOVA et G. M. P. O'HARE. « Ad-me : Wireless Advertising Adapted to the User Location, Device and Emotions ». Dans *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*, HICSS '04, pages 90285.3–, Washington, DC, USA, 2004. IEEE Computer Society.

[HTHT09] Yuko HARA, Hiroyuki TOMIYAMA, Shinya HONDA, et Hiroaki TAKADA. « Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. ». *JIP*, 17 :242–254, 2009.

[Inc] Xilinx INC..

[Joh79] Stephen C. JOHNSON. « Yacc : Yet Another Compiler-Compiler ». Rapport Technique, 1979.

[JS04] Jaeyeon JUNG et Emil SIT. « An Empirical Study of Spam Traffic and the Use of DNS Black Lists ». Dans *Internet Measurement Conference*, Taormina, Italy, October 2004.

[KF02] T. KINDBERG et A. FOX. « System software for ubiquitous computing ». *Pervasive Computing, IEEE*, 1(1) :70–81, 2002.

[Kle01] J. KLENSIN. « Simple Mail Transfer Protocol ». RFC 2821 (Proposed Standard), avril 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[KMC+00] Eddie KOHLER, Robert MORRIS, Benjie CHEN, John JANNOTTI, et M. Frans KAASHOEK. « The click modular router ». *ACM Trans. Comput. Syst.*, 18(3) :263–297, août 2000.

[KMKC09] F. KIYAK, B. MOCHIZUKI, E. KELLER, et M. CAESAR. « Better by a HAIR : Hardware-amenable internet routing ». Dans *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 83–92, Oct 2009.

[Knu64] Donald E. KNUTH. « backus normal form vs. Backus Naur form ». *Commun. ACM*, 7(12) :735–736, décembre 1964.

[KR06] Ian KUON et Jonathan ROSE. « Measuring the gap between FPGAs and ASICs ». Dans *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

[Kri06] S. KRISHNAMURTHY. « TinySIP : Providing Seamless Access to Sensor-based Services ». Dans *3rd International Conference on Mobile and Ubiquitous Systems : Networking and Services*, numéro 4611 dans Lecture Notes in Computer Science, pages 1–9, 2006.

[KT08] S. KELLY et J.P. TOLVANEN. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley, 2008.

[Law02] Julia L. LAWALL. « Capturing OS expertise in an Event Type System : the Bossa experience ». Dans *In Tenth ACM SIGOPS European Workshop 2002 (EW2002*, pages 54–61, 2002.

[LBCO04] Christian LENGAUER, Don S. BATORY, Charles CONSEL, et Martin ODERSKY, éditeurs. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 de *Lecture Notes in Computer Science*. Springer, 2004.

[LCMC08] Qiang LIU, George A. CONSTANTINIDES, Konstantinos MASSELOSY, et Peter Y. K. CHEUNG. « Compiling C-like languages to FPGA hardware : some novel approaches targeting data memory organisation ». Dans *Proceedings of the 2008 international conference on Visions of Computer Science : BCS International Academic Conference*, VoCS'08, pages 295–303, Swinton, UK, UK, 2008. British Computer Society.

[LDD+96] H. Dan LAMBRIGHT, Saumya K. DEBRAY, H. DAN, Lambright SAUMYA, et K. DEBRAY. « APF : A Modular Language for Fast Packet Classification », 1996.

[LL90] Peter S. LANGSTON et Peter S. LANGSTON. « Little Languages for Music », 1990.

[LMHR05] S. LEGGIO, J. MANNER, A. HULKKONEN, et K. RAATIKAINEN. « Session Initiation Protocol deployment in ad-hoc networks : A decentralized approach ». Dans *2nd International Workshop on Wireless Ad-hoc Networks*, 2005.

[LMW+07] J.W. LOCKWOOD, N. MCKEOWN, G. WATSON, G. GIBB, P. HARTKE, J. NAOUS, R. RAGHURAMAN, et Jianying LUO. « NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing ». Dans *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, pages 160–161, 2007.

[LS90] M. E. LESK et E. SCHMIDT. UNIX Vol. II. Chapitre Lex a lexical analyzer generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[LY99] Tim LINDHOLM et Frank YELLIN. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 1999.

[Mar67] J. MARTIN. *Design of real-time computer systems*. Prentice-Hall series in automatic computation. Prentice-Hall, 1967.

[MC00] Peter J. MCCANN et Satish CHANDRA. « Packet types : abstract specification of network protocol messages ». Dans *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. ACM.

[MFW+07] Yitzhak MANDELBAUM, Kathleen FISHER, David WALKER, Mary F. FERNÁNDEZ, et Artem GLEYZER. « PADS/ML : a functional data description language. ». Dans Martin HOFMANN et Matthias FELLEISEN, éditeurs, *POPL*, pages 77–83. ACM, 2007.

[MHD+07] Anil MADHAVAPEDDY, Alex HO, Tim DEEGAN, David SCOTT, et Ripduman SOHAN. « Melange : creating a "functional" internet ». *SIGOPS Oper. Syst. Rev.*, 41(3) :101–114, mars 2007.

[MHS05] Marjan MERNIK, Jan HEERING, et Anthony M. SLOANE. « When and how to develop domain-specific languages ». *ACM Comput. Surv.*, 37(4) :316–344, décembre 2005.

[Mil96] D. MILLS. « RFC 2030 : Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI », octobre 1996. Obsoletes RFC1769 [**?**]. Status : INFORMATIONAL.

[ML07] Nilo MITRA et Yves LAFON. « SOAP Version 1.2 Part 0 : Primer (Second Edition) ». World Wide Web Consortium, Recommendation REC-soap12-part0-20070427, April 2007.

[MLLP03] James MOSCOLA, John LOCKWOOD, Ronald P. LOUI, et Michael PACHOS. « Implementation of a Content-Scanning Module for an Internet Firewall ». Dans *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '03, pages 31–, Washington, DC, USA, 2003. IEEE Computer Society.

[MRC+00] F. MÉRILLON, L. RÉVEILLÈRE, C. CONSEL, R. MARLET, et G. MULLER. « Devil : An IDL for Hardware Programming ». Dans *4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, California, octobre 2000.

[MVB+09] Abhishek MITRA, Marcos R. VIEIRA, Petko BAKALOV, Walid A. NAJJAR, et Vassilis J. TSOTRAS. « Boosting XML Filtering with a Scalable FPGA-based Architecture ». *CoRR*, abs/0909.1781, 2009.

[oca] « Objective Caml (OCaml) programming language website. ». http ://-caml.inria.fr/.

[OKH00] J. OBERG, A. KUMAR, et A. HEMANI. « Grammar-based hardware synthesis from port-size independent specifications ». *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(2) :184–194, 2000.

[PADF+07] Erwann POUPART, Ali ABOU DIB, Louis FÉRAUD, Pierre BAZEX, Ileana OBER, Christian PERCEBOIS, et Thierry MILLAN. « Vers une abstraction d'une famille de DSL ». *Génie Logiciel*, 81 :24–31, juin 2007.

[PCRL07] Nicolas PALIX, Charles CONSEL, Laurent RÉVEILLÈRE, et Julia L. LAWALL. « A stepwise approach to developing languages for SIP telephony service creation ». Dans *IPTComm*, pages 79–88, 2007.

[PG03] M. P. PAPAZOGLOU et D. GEORGAKOPOULOS. « Introduction : Service-oriented computing ». *Commun. ACM*, 46(10) :24–28, octobre 2003.

[POJK03] A. PELINESCU-ONCIUL, J. JANAK, et Jiri KUTHAN. « SIP Express Router (SER) ». *IEEE Network Magazine*, 17(4) :9, July/August 2003.

[Pos81a] J. POSTEL. « RFC 788 : Simple Mail Transfer Protocol », novembre 1981. Obsoleted by RFC0821 [**?**]. Obsoletes RFC0780 [**?**]. Status : UNKNOWN. Not online.

[Pos81b] J. POSTEL. « RFC 791 : Internet Protocol », septembre 1981.

[Pos81c] J. POSTEL. « RFC 793 : Transmission Control Protocol », septembre 1981.

[PPSP06] Ruoming PANG, Vern PAXSON, Robin SOMMER, et Larry PETERSON. « binpac : a yacc for writing application protocol parsers ». Dans *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 289–300, New York, NY, USA, 2006. ACM.

[QPP02] Xiaohu QIE, Ruoming PANG, et Larry PETERSON. « Defensive programming : using an annotation toolkit to build DoS-resistant software ». *SIGOPS Oper. Syst. Rev.*, 36(SI) :45–60, décembre 2002.

[Rab00] Jan M. RABAEY. « Low-power silicon architecture for wireless communications : embedded tutorial ». Dans *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, ASP-DAC '00, pages 377–380, New York, NY, USA, 2000. ACM.

[RAKD12] Teemu RINTA-AHO, Mika KARLSTEDT, et Madhav P. DESAI. « The Click2NetFPGA toolchain ». Dans *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[Res01] P. RESNICK. « RFC 2822 : Internet Message Format ». Rapport Technique, IETF, 2001.

[Roe99] Martin ROESCH. « Snort - Lightweight Intrusion Detection for Networks ». Dans *Proceedings of the 13th USENIX conference on System administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association.

[RSC+02] J. ROSENBERG, H. SCHULZRINNE, G. CAMARILLO, A. JOHNSTON, J. PETERSON, R. SPARKS, M. HANDLEY, et E. SCHOOLER. « SIP : Session Initiation Protocol », 2002.

[rts] « RTSP Gstreamer ».

[Ré01] Laurent RÉVEILLÈRE. « *Approche langage au développement de pilotes de périphériques robustes* ». PhD thesis, Rennes 1, IFSIC, 2001. Th. : informatique : ; 2618.

[Sai11] K. SAINI. *Squid Proxy Server 3.1 : Beginner's Guide*. Packt Publishing, Limited, 2011.

[Sat01] M. SATYANARAYANAN. « Pervasive computing : vision and challenges ». *Personal Communications, IEEE*, 8(4) :10–17, 2001.

[SBRA07] P. STUEDI, M. BIHR, A. REMUND, et G. ALONSO. « SIPHoc : Efficient SIP Middleware for Ad Hoc Networks ». Dans *Proceedings of the 8th ACM/IFIP/USENIX International Conference on Middleware*, 2007.

[SGD+02] Stefan SAROIU, Krishna P. GUMMADI, Richard J. DUNN, Steven D. GRIBBLE, et Henry M. LEVY. « An analysis of Internet content delivery systems ». *SIGOPS Oper. Syst. Rev.*, 36(SI) :315–327, décembre 2002.

[SGL+11] G. STITT, A GEORGE, H. LAM, C. REARDON, M. SMITH, B. HOLLAND, V. AGGARWAL, Gongyu WANG, J. COOLE, et S. KOEHLER. « An End-to-End Tool Flow for FPGA-Accelerated Scientific Computing ». *Design Test of Computers, IEEE*, 28(4) :68–77, July 2011.

[Som03] Robin SOMMER. « Bro : An Open Source Network Intrusion Detection System. ». Dans Jan von KNOP, Wilhelm HAVERKAMP, et Eike JESSEN, éditeurs, *DFN-Arbeitstagung über Kommunikationsnetze*, volume 44 de *LNI*, pages 273–288. GI, 2003.

[SP01] Reetinder SIDHU et Viktor K. PRASANNA. « Fast Regular Expression Matching Using FPGAs ». Dans *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '01, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[SPA99] SPARC INTERNATIONAL INC.. « *The SPARC Architecture Manual* », ver. 8, 1999.

[SRL98] H. SCHULZRINNE, A. RAO, et R. LANPHIER. « RFC 2326 : Real Time Streaming Protocol (RTSP) », avril 1998. Status : PROPOSED STANDARD.

[Sta02] P. STARK. « RFC 3236 :The application/xhtml+xml Media Type », janvier 2002.

[TD04] Marie THILLIEZ et Thierry DELOT. « Evaluating Location Dependent Queries Using ISLANDS. ». Dans Félix F. RAMOS, Herwig UNGER, et Victor LARIOS, éditeurs, *ISSADS*, volume 3061 de *Lecture Notes in Computer Science*, pages 125–136. Springer, 2004.

[Thu06] Adrian D. THURSTON. « Parsing Computer Languages with an Automaton Compiled from a Single Regular Expression ». Dans *Proceedings of the 11th International Conference on Implementation and Application of Automata*, CIAA'06, pages 285–286, Berlin, Heidelberg, 2006. Springer-Verlag.

[vDF97] A. van DEURSEN et A FINANCIAL. « Domain-Specific Languages versus Object-Oriented Frameworks : A Financial Engineering Case Study ». 1997.

[vDKV00] Arie van DEURSEN, Paul KLINT, et Joost VISSER. « Domain-specific languages : an annotated bibliography ». *SIGPLAN Not.*, 35(6) :26–36, juin 2000.

[Wei91] Mark WEISER. Human-computer interaction. Chapitre The computer for the 21st century, pages 933–940. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[Wei93] Mark WEISER. « Hot topics—Ubiquitous Computing ». *Computer*, 26(10) :71–72, oct 1993.

[Wei94] Mark WEISER. « The world is not a desktop ». *Interactions*, 1(1) :7–8, 1994.

[Wei99] Mark WEISER. « Some computer science issues in ubiquitous computing ». *Mobile Computing and Communications Review*, 3(3) :12, 1999.

[Wil04] David WILE. « Lessons learned from real DSL experiments ». *Sci. Comput. Program.*, 51(3) :265–290, juin 2004.

[WM98] E. WHITEHEAD et M. MURATA. « RFC 2376 : XML Media Types », juillet 1998. Status : INFORMATIONAL.

[WSKW07] S. WANKE, M. SCHARF, S. KIESEL, et S. WAHL. « Measurement of the SIP Parsing Performance in the SIP Express Router ». Dans *Dependable and Adaptable Networks and Services*, numéro 4606 dans Lecture Notes in Computer Science, pages 103–110, 2007.

[Xil12] XILINX. « *Vivado Design Suite User Guide : Synthesis* ». UG901 (v2012.2), July 25 2012.

[Y E10] Y Explorations (XYI), San Jose, CA. « *eXCite C to RTL Behavioral Synthesis 4.1(a)* », 2010.