POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Outils de codesign pour la vérification de partitionnement par hyperviseur sur système embarqué matériel-logiciel de criticités mixtes
Auteur: Author:	Fabien Portas
Date:	2023
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Portas, F. (2023). Outils de codesign pour la vérification de partitionnement par hyperviseur sur système embarqué matériel-logiciel de criticités mixtes [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/57015/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/57015/
Directeurs de recherche: Advisors:	Guy Bois, & Yvon Savaria
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Outils de codesign pour la vérification de partitionnement par hyperviseur sur système embarqué matériel-logiciel de criticités mixtes

FABIEN PORTAS
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2023

© Fabien Portas, 2023.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Outils de codesign pour la vérification de partitionnement par hyperviseur sur système embarqué matériel-logiciel de criticités mixtes

présenté par **Fabien PORTAS**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Tarek OULD-BACHIR, président
Guy BOIS, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Michel DAGENAIS, membre

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche Guy Bois, qui m'a accompagné et sagement conseillé tout au long de ma maîtrise. Sa grande expérience et son expertise ont grandement enrichi mes connaissances, et m'ont de nombreuses fois permis de sortir d'impasses lors de mon travail.

Je veux également remercier mon co-directeur de recherche Yvon Savaria, qui, malgré son rattachement assez tardif à mon travail, m'a donné de nombreux conseils inestimables sur la méthodologie et la rédaction scientifique.

Également, je remercie Anthony Dentringer, ingénieur chez SpaceCodesign Systems Inc, qui m'a grandement accompagné et aidé sur les aspects techniques des aspects d'instrumentation de cet ouvrage, et sans qui je n'aurais sans doute pas pu achever ce travail. Je suis également reconnaissant envers Hubert Guérard, PDG et directeur technique de SpaceCodesign Systems Inc, qui m'a également apporté son support au niveau des problématiques de simulation de ce travail.

Enfin, je me dois de remercier le programme Mitacs Accelerate et SpaceCodesign Systems Inc, qui m'ont permis, grâce à leur contribution financière, de bénéficier d'une bourse d'études pour subvenir à mes besoins durant mon travail de maîtrise.

RÉSUMÉ

Les systèmes embarqués modernes tendent à embarquer de plus en plus de fonctionnalités complexes sur un même système, en utilisant des hyperviseurs de partitionnement pour s'assurer que les applications plus et moins critiques n'interfèrent pas entre elles lors de leur fonctionnement. Par ailleurs, pour respecter les contraintes exigeantes du domaine embarqué en performance et consommation énergétique, ces systèmes intègrent fréquemment du matériel programmable sous la forme d'un FPGA, devenant des systèmes hétérogènes. Ces deux facteurs combinés rendent ainsi la conception de tels systèmes extrêmement complexe.

Cette montée en complexité a amené des nouvelles méthodes de conception qui élèvent le niveau d'abstraction auquel on conçoit le système. On s'intéresse ici à la méthodologie ESL, qui considère la conception du système en partant du niveau algorithmique de description du système, directement en langage de programmation haut niveau, plutôt que du niveau RTL très bas niveau.

Cependant, cette méthodologie nécessite un certain nombre d'outils afin d'être réellement efficace. Or, dans le cas des systèmes basés un hyperviseur, les outils existants trouvent des défauts. D'abord au niveau des outils de simulation, qui permettent dans la méthodologie une validation fonctionnelle rapide sans passage à l'implémentation. Ensuite, au niveau des outils de mesure de performances, qui permettent de détecter certains problèmes de conception tels que les interférences de partitionnement de l'hyperviseur. Ceux-ci sont soit très lourds, soit peu adaptés à la méthodologie, requérant une descente à un niveau d'abstraction plus bas, annulant possiblement les gains espérés de la méthodologie haut niveau.

Nous proposons dans un premier temps un outil d'instrumentation de performances qui permet de récupérer et manipuler des informations sur le comportement temporel via une API unifiée en logiciel et en matériel (grâce à un environnement de synthèse matérielle haut niveau). Celle-ci s'intègre dans la plateforme de développement SpaceStudio, appliquant la méthodologie ESL, et repose sur divers mécanismes d'analyse statique et de génération de code pour obtenir une transparence totale de la cible pour l'utilisateur.

Nos tests démontrent la légèreté de ces outils en ce qui a trait à la consommation de ressources, et leur faible impact sur le système observé, une nécessité pour obtenir des mesures précises sur des systèmes contraints en ressources. Aussi, nous démontrons sur un cas simple l'utilité d'un tel outil pour détecter des problèmes de performances sur un système hétérogène avec hyperviseur de partitionnement. En revanche, les outils restent globalement pour l'instant rudimentaires, laissant une charge importante à l'utilisateur pour pleinement les exploiter.

Des abstractions de plus haut-niveau pourraient être proposées pour faciliter la prise en main de l'outil par le développeur.

Nous proposons par la suite une simulation efficace de Xng, un hyperviseur de partitionnement commercial, sur deux plateformes embarquées, l'une étant le Cortex-A9, bien établie et ayant un support préliminaire important, et la seconde la plateforme Ng-Ultra, très jeune et disposant d'un support logiciel plus faible, mais plus à jour en ce qui a trait aux évolutions technologiques modernes. Ce support est spécifiquement ciblé pour Xng, mais les techniques de virtualisation simulées devraient aider au support de n'importe quel autre hyperviseur de partitionnement.

Après des tests fonctionnels, nous obtenons un bon support de l'hyperviseur sur la première plateforme, et un support préparatoire à la seconde, le port de l'hyperviseur sur celle-ci étant indisponible au moment de l'écriture de ce rapport. La simulation présente cependant la limite d'avoir une précision faible en termes de timing, et pourrait être retravaillée pour permettre une meilleure évaluation des performances dès la simulation.

Nous avons donc atteint l'objectif de proposer de nouveaux outils pour aider au développement des plateformes avec hyperviseurs par la méthodologie ESL, à la fois par la simulation et de la mesure de performances. Tous deux présentent cependant des limites de précision et de sur-simplicité, mais apportent tout de même une véritable valeur ajoutée dans le flot de conception ESL.

ABSTRACT

The rise in the complexity of modern embedded systems has led to the integration of more and more functionality on shared hardware, with a partitioning hypervisor ensuring isolation between critical applications. Further increasing design complexity is the trend of integrating FPGAs to reach better performance while remaining within the stringent energy-efficiency requirement of the embedded world.

Development of such complex systems has seen the development of new design methodologies, which propose a rise in the abstraction level to allow the developer to better conceptualize the many interacting software and hardware components. One of such high-level methodlogies is the ESL methodology, in which the development starts from algorithmic code expressed in high-level languages, and proceeds to achieve a final implementation through successive refinements of the system. This expresses a more ”top-down” design process rather than the traditional ”bottom-up” methodologies which start at a lower abstraction level such as RTL to design the system.

When it comes to hypervisor-based systems, however, the ESL method shows a few shortcomings. First, the efficient simulation models required for functional validation in this methodology show a lack of support for virtualization features required for most partitioning hypervisors, making their simulation impossible in the current state. Moreover, the tools to analyze performance metrics, which are crucial to finding partitioning issues related to the hypervisor, are either too heavy in hardware resources or are not fit for use in high-level code central to the ESL methodology.

We set on to address the two problems with two proposals.

First is the proposal of new performance monitoring tools that fit the ESL methodology by providing a unified C/C++ API to retrieve and manipulate timing information both in software and hardware (through the use of high-level synthesis tools). It integrates in the SpaceStudio platform, following a variant of the ESL methodology. Methods such as static analysis and code generation ensure full transparency to the developer using the API, whether it targets software or hardware implementations.

Testing using resource estimation proves that the tools are relatively lightweight regarding hardware resource utilization. We also show the ability of the tool to help diagnose a performance problem linked with bus contention in a hypervisor-based mixed criticality SoC-FPGA system. However, the tools implemented in this work are somewhat barebones and leave the

burden of exploiting timing information to the user. Creating a higher level of abstraction for measuring delays through the code would facilitate future usage of our tools.

The other proposal is the improvement of the support for partitioning hypervisors on QEMU, an efficient system simulator, for two target platforms, the first, the Cortex-A9 being older but benefiting from more industrial support, and the other one, the Ng-Ultra, which uses more future-proof technology but that lacks software support at this time. While this support is targeted at a specific commercial hypervisor, the virtualization features that are simulated form a base for the simulation of any partitioning hypervisor on the market.

Functional testing exhibits good support for the hypervisor on the older platform, and tentative support for the newer one, although the Xng port for it is still a work in progress, and roundabout ways of testing the simulation had to be used. The simulation, however, shows a great lack of precision when it comes to timing simulation, which would need work to diagnose performance issues directly in simulation.

Overall, the goal of making the development of complex hypervisor-based mixed-criticality heterogeneous platforms has been achieved through improvements to simulation and runtime performance monitoring tooling. While both aspects still present a few flaws, they nonetheless show a definite advantage over the status quo.

TABLE DES MATIÈRES

REMERCIEMENTS iii

RÉSUMÉ iv

ABSTRACT vi

TABLE DES MATIÈRES viii

LISTE DES TABLEAUX x

LISTE DES FIGURES xi

LISTE DES SIGLES ET ABRÉVIATIONS xii

CHAPITRE 1 INTRODUCTION 1
1.1 Contexte 1
1.2 Problématiques 1
1.3 Méthodologie 2
1.4 Résumé des contributions 2
1.5 Structure du mémoire 3

CHAPITRE 2 MISE EN CONTEXTE ET REVUE DE LITTÉRATURE 4
2.1 Systèmes hétérogènes de criticités mixtes 4
2.1.1 Codesign et plateformes virtuelles dynamiques 6
2.1.2 Hyperviseurs de partitionnement 9
2.1.3 Interférences dans les systèmes hétérogènes 11
2.1.4 Outils diagnostics actuels et lacunes pour les méthodologies haut-niveau 12
2.1.5 Lacunes de la simulation des systèmes avec hyperviseurs 15
2.2 Objectifs 15

CHAPITRE 3 PREMIER THÈME - OUTIL D'INSTRUMENTATION DE PERFORMANCES 16
3.1 Spécification des requis 16
3.2 Architecture et implémentation 18
3.3 Résultats expérimentaux 23

3.3.1 Environnement de test 23
3.3.2 Étude de la consommation de ressources matérielles du module Spaceclock 23
3.3.3 Étude de la consommation de ressources matérielles de la manipulation des points de temps 25
3.3.4 Application de l'outil à un cas d'interférence dans un système partitionné par hyperviseur 27
3.3.5 Application à l'exploration architecturale et à la validation de la simulation 30
3.4 Conclusion du chapitre 32

CHAPITRE 4 SECOND THÈME - PLATEFORME VIRTUELLE POUR HYPERVISEUR 33
4.1 Modèle de co-simulation avec QEMU & SystemC 33
4.1.1 Le simulateur QEMU 33
4.2 Amélioration d'un modèle de simulation existant 34
4.2.1 Plateforme Cortex-A9 34
4.2.2 Fonctionnalités présentes et manquantes 34
4.2.3 Résultats expérimentaux : validation de la simulation 35
4.3 Création d'un modèle pour une plateforme moderne 36
4.3.1 Plateforme Ng-Ultra 36
4.3.2 Fonctionnalités nécessaires 37
4.3.3 Résultats expérimentaux : validation fonctionnelle de la simulation . 39

CHAPITRE 5 CONCLUSION 41
5.1 Synthèse des travaux 41
5.2 Limitations de la solution proposée 41
5.3 Améliorations futures 42

RÉFÉRENCES 43

LISTE DES TABLEAUX

Tableau 3.1 Consommation de ressources post-implémentation selon le nombre de modules connectés au module Spaceclock, en valeurs absolues et en proportion des ressources disponibles sur la Zedboard 24
Tableau 3.2 Consommation de ressources du système de moniteurs 26
Tableau 3.3 Indices de performance de multiplication matricielle en matériel sans interférences d'un DMA secondaire (valeurs en cycles) 29
Tableau 3.4 Indices de performance de multiplication matricielle en matériel *avec* interférences d'un DMA secondaire (valeurs en cycles) et comparaison aux valeurs sans interférences dans le tableau 3.3 29
Tableau 3.5 Temps d'exécution moyen (en millisecondes) de chaque section du code de l'accélérateur matriciel selon s'il s'exécute en matériel et logiciel, sur l'implémentation et en simulation logicielle 31

LISTE DES FIGURES

Figure 2.1 Diagramme simplifié de la structure d'un système hétérogène SoC-FPGA 5
Figure 2.2 Illustration schématique des niveaux de raffinement de la modélisation d'un système, avec un exemple de raffinements successifs possibles . . 7
Figure 2.3 Illustration schématique des avantages des plateformes virtuelles en terme d'efforts en fonction du temps de mise sur le marché 9
Figure 2.4 Organisation des deux différents types d'hyperviseurs 10
Figure 3.1 Architecture du routage des accès aux estampilles de temps sous-jacentes à un point de temps selon la cible 18
Figure 3.2 Diagramme de classe simplifié de l'API d'estampille de temps. En rouge clair sont les sections déterminées par de la génération de code automatique 20
Figure 3.3 Exemple de design simpliste illustrant les connexions du module Spaceclock 22
Figure 3.4 Architecture des outils abstraits de mesure de performance implémentés au niveau utilisateur sur un module d'accélération de multiplication matricielle 26
Figure 3.5 Diagramme simplifié de l'architecture du système de test de contention complet 28
Figure 4.1 Diagramme simplifié de l'architecture de la plateforme Ng-Ultra . . . 36
Figure 4.2 Diagramme d'architecture et exécution en simulation du RTOS RTEMS sur le Ng-Ultra dans la plateforme SpaceStudio 40

LISTE DES SIGLES ET ABRÉVIATIONS

A | C | E | F | H | I | L | M | O | R | S | T | U | V

A

API Application-Programmer Interface vi, 10, 13, 16, 17, 23, 41

ARINC Aeronautical Radio, INCorporated 10

AXI Advanced eXtensible Interface 11

C

CPU Central Processing Unit 5, 33

E

ELA Embedded Logic Analyzer 12, 13

ESL Embedded System Level iv–vi, 6, 8, 12, 14–16, 32

F

FA Federated Avionics 5

FPGA Field-Programmable Gate Array iv, vi, 1, 3–9, 11, 12, 17, 18, 21, 23, 30, 31, 34, 37, 39, 41

H

HLS High-Level Synthesis 1, 2, 6, 8, 12, 13, 16, 18, 19

I

IMA Integrated Modular Avionics 5

L

LTTng Linux Trace Toolkit : next generation [1] 12

M

MMU Memory Management Unit 34, 37

MPU Memory Protection Unit 37–39

O

OS Operating System 10

R

RTL Register Transfer Level iv, 6, 7, 13, 22, 42

RTOS Real-Time Operating System xi, 39, 40

S

SoC System-on-Chip 4, 11, 23, 37

T

TLM Transaction-Level Modelling 7, 9, 34

U

UART Universal Asynchronous Receiver Transmitter 38

V

VCR Virtualization Control Register 34, 35

CHAPITRE 1 INTRODUCTION

Ce chapitre introduit rapidement le contexte des systèmes embarqués hétérogènes à criticités mixtes, amène les problématiques liées à leur conception, et résume les contributions de ce mémoire. Ces thèmes seront approfondis dans le chapitre 2.

1.1 Contexte

Les systèmes embarqués modernes gagnent continuellement en complexité, embarquant de plus en plus de fonctionnalités sur un seul système. Qui plus est, la tendance est à l'intégration de composants matériels programmables, sous la forme d'un FPGA (Field-Programmable Gate Array) pour atteindre des objectifs stricts en termes de performances et de consommation énergétique. Ce mélange de composants matériels et logiciels rend ainsi la conception de tels systèmes hétérogènes plus complexe, nécessitant du travail supplémentaire au niveau des communications entre les deux domaines, et plus de temps de validation pour s'assurer du respect de contraintes temps-réel.

Une autre complexité arrive du fait que les applications regroupées sur un unique système n'ont pas nécessairement des criticités identiques. Des outils logiciels viennent pallier ‘a ce problème : un hyperviseur de partitionnement assure la répartition équitable des ressources du système et de l'isolation en cas de faute d'une application. L'hyperviseur évite alors par exemple qu'une application peu critique en état de faute cause un défaut dans une application plus critique.

Pour faciliter le développement de systèmes hétérogènes matériel-logiciel, les dernières décennies ont vu l'émergence de méthodes de conception élevant le niveau d'abstraction auquel on conçoit un tel système, afin de réduire leur temps de développement. Ces méthodologies exploitent notamment des méthodes de simulation des systèmes pour effectuer une validation fonctionnelle rapide et de synthèse de matériel haut niveau (ou HLS pour High-Level Synthesis), qui permet une génération de la portion matérielle à partir de code haut-niveau proche du logiciel.

1.2 Problématiques

À l'intersection des deux domaines cités en 1.1, les systèmes hétérogènes matériel-logiciel à criticité mixtes partitionnés par un hyperviseur présentent un manque important de support des outils de développement haut-niveau cités précédemment.

D'abord, au niveau de la simulation, celle-ci supporte mal les mécanismes de virtualisation utilisés par les hyperviseurs, ce qui oblige une descente à l'implémentation sur le matériel pour effectuer une validation fonctionnelle préliminaire, une opération longue et parfois impossible en l'absence d'un prototype matériel.

Ensuite, au niveau de l'implémentation même, l'intégration de composants matériels ajoute des points de conflits possible entre les applications de criticités différentes, que l'hyperviseur ne peut gérer. Or, les outils de diagnostics de ces problématiques de performances montrent des manquements quant aux méthodologies haut-niveau, notamment au niveau du matériel synthétisé grâce au HLS.

1.3 Méthodologie

Pour pallier à ces problématiques, nous proposons deux outils facilitant la conception de systèmes hétérogènes matériel-logiciel à criticités mixtes.

Dans un premier temps, nous proposons un outil d'instrumentation de performance universel, qui fournit une abstraction permettant un usage identique pour des cibles logicielles et matérielles via le HLS, donc parfaitement adaptés aux méthodologies de conception haut-niveau.

Dans un second temps, nous proposons une simulation performante à haut-niveau de tels systèmes basés sur un hyperviseur de partitionnement.

1.4 Résumé des contributions

Ce travail a abouti à trois principales contributions :

- Dans un premier temps, nous proposons un outil qui permet l'intégration transparente d'instruments de mesure de performances à la fois en logiciel et en matériel sur les systèmes SoC-FPGA adapté à une méthodologie de conception de matériel à haut-niveau. À notre connaissance, une telle abstraction unifiée n'a jamais été proposée dans la littérature.
- Deuxièmement, nous avons amélioré QEMU, un simulateur de jeu d'instructions open-source performant, afin de supporter la simulation des systèmes basés sur les hyperviseurs de partitionnement.
- Comme corollaire de la combinaison des deux contributions précédentes, notre outil d'instrumentation de performance permet une méthode de validation de la précision d'une simulation d'un système.

1.5 Structure du mémoire

Ce mémoire est organisé en quatre chapitres, en excluant la présente introduction.

Le chapitre 2 commence par expliciter plus en détail les concepts essentiels à la compréhension du contexte du domaine, et il présente en même temps une revue des travaux précédents sur le sujet.

Ensuite, dans le chapitre 3, nous présentons une spécification des requis de l'outil d'instrumentation de performances que nous proposons. Ensuite, nous définissons l'architecture requise en termes plus techniques, pour détailler par la suite ses conditions d'implémentation en pratique dans la plateforme SpaceStudio. Nous terminons ensuite par des résultats sur des exemples d'utilisation de l'outil, afin de souligner sa précision et, dans le cas du ciblage matériel, de son faible usage de ressources du FPGA.

Le chapitre 4 qui suit traite des aspects d'adaptation d'une plateforme virtuelle existante à l'utilisation d'un hyperviseur de partitionnement, d'abord avec une plateforme matérielle standard existante, puis sur une nouvelle plateforme en développement, donc dont les prototypes physiques n'existent pas encore, montrant l'intérêt d'une telle simulation.

Enfin, un dernier chapitre 5 vient conclure cet ouvrage en résumant les travaux achevés, ainsi que leurs limitations, ouvrant sur des pistes d'améliorations futures permettant d'y pallier.

CHAPITRE 2 MISE EN CONTEXTE ET REVUE DE LITTÉRATURE

Ce chapitre explicite plus en détail les concepts centraux à cet ouvrage que constituent les systèmes embarqués hétérogènes matériel-logiciel de criticités mixtes. Nous fournissons également une analyse de l'état de l'art des outils d'instrumentation de performances et leurs lacunes, qui permet de préciser notre problématique.

2.1 Systèmes hétérogènes de criticités mixtes

Les systèmes embarqués modernes intègrent toujours plus de fonctionnalités variées, et simultanément, chacune de ces fonctionnalités gagne en complexité. Cette tendance amène les méthodes de conception classiques à leurs limites. C'est pourquoi des innovations dans la conception et l'architecture de ces systèmes sont apparues afin d'exécuter ces applications toujours plus lourdes en calcul en respectant les contraintes très restrictives du domaine en termes de poids, de consommation énergétique et de sûreté.

Systèmes hétérogènes matériel-logiciel

Dans un premier temps, on remarque que la tendance est à l'intégration sur un système d'un ou plusieurs processeurs généralistes et de logique programmable (sous la forme d'un FPGA) sur la même plateforme, ce qui nous donne des systèmes hétérogènes SoC-FPGA. Cela a l'avantage de permettre d'exploiter au mieux les avantages des deux cibles.

D'abord, le processeur apporte une grande flexibilité et une aisance de programmation, mais une efficacité plus limitée dans les tâches lourdes en calcul, particulièrement en considérant les restrictions de consommation énergétique d'un système embarqué, qui limitent la fréquence maximale des processeurs utilisables. Ainsi, pour la gestion des tâches d'entrée-sortie et de gestion globale du système il est plus facile d'utiliser un système d'exploitation temps-réel.

Le FPGA associé au processeur offre l'avantage d'une faible consommation énergétique et de bien meilleures performances sur des tâches calculatoires grâce à un plus grand parallélisme possible en matériel. Ceci vient cependant au prix d'une flexibilité réduite et d'un plus grande difficulté pour les programmer. On le préférera donc pour déporter des sections particulièrement exigeantes en calcul avec des contraintes de latence fortes, sous forme d'accélérateurs ou co-processeurs implémentés sur celui-ci.

Cette intégration de composantes matérielle et logicielle augmente significativement la complexité du développement d'une application pour un tel système, l'hétérogénéité apportant

notamment des considérations au niveau de la communication entre le CPU et le FPGA, en lien avec le découpage du calcul entre les deux unités, etc….

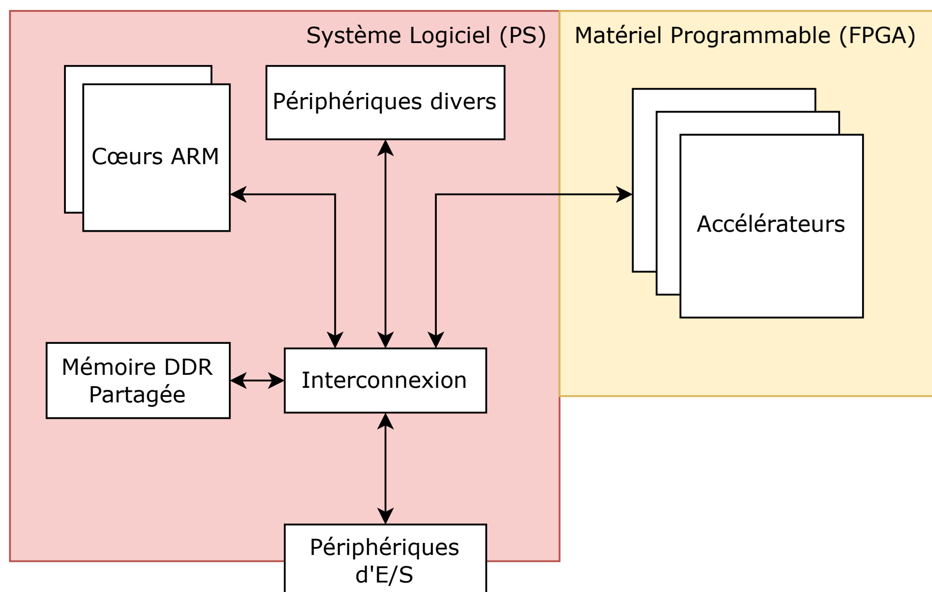

FIGURE 2.1 Diagramme simplifié de la structure d'un système hétérogène SoC-FPGA

Système à criticités mixtes

Pour des raisons d'économie de poids et de coût, les systèmes modernes tendent à regrouper plusieurs applications sur du matériel commun [2]. À titre d'exemple de cette tendance, on voit dans le domaine aérospatial, et comme ce fut le cas précédemment pour l'avionique, une évolution des architectures historiques, dites « fédérées », ou FA, qui utilisent plusieurs systèmes informatiques distincts connectés par un bus de communication, vers les « architectures modulaires intégrées », ou IMA. Les secondes privilégient le partage des ressources informatiques, particulièrement le processeur, généralement multi-cœur, ou encore la mémoire [3], entre plusieurs applications précédemment physiquement isolées. On économise ainsi le poids, le coût et la consommation énergétique du matériel redondant.

Or, toutes les applications ainsi regroupées n'auront alors pas toutes la même importance, et les conséquences de pannes ou de défauts seront possiblement radicalement différentes, selon que la tâche accomplie est plus ou moins critique. Un tel système est donc qualifié de système à criticités mixtes. Cependant, les applications n'étant plus physiquement isolées sur du matériel distinct, les interférences entre applications peu et fortement critiques deviennent un nouvel aspect qu'il est crucial d'étudier pour l'ingénieur concevant un tel système.

2.1.1 Codesign et plateformes virtuelles dynamiques

Codesign

La conception d'un système hétérogène SoC-FPGA doit donc nécessairement prendre en compte à la fois les aspects matériel et logiciel de la plateforme. La méthodologie dite de codesign matériel-logiciel [4] prescrit d'approcher ces deux facettes du systèmes simultanément, plutôt que séquentiellement, afin d'exploiter au mieux les avantages des deux pour atteindre les objectifs de conception (en termes de latence, consommation énergétique, etc…) plus rapidement, et de réduire ainsi les coûts associés à une mise sur le marché tardive.

Par ailleurs, un élément rendant la programmation de ces systèmes plus complexe, particulièrement pour un ingénieur logiciel, est la programmation des accélérateurs matériels implantés sur le FPGA. En effet, cela peut entre autres impliquer de la programmation RTL (en VHDL ou Verilog par exemple), souvent étrangère aux développeurs de logiciels embarqués. Qui plus est, face à la complexité des applications modernes, la conception d'accélérateurs matériels devient même un frein pour des ingénieurs familiers avec ces technologies.

Le niveau système électronique, ou ESL

La complexité toujours grandissante des systèmes électroniques explique pourquoi on a vu émerger dans les dernières décennies des méthodologies élevant le niveau d'abstraction du processus de développement, à l'image de ce qu'on a observé précédemment en ingénierie logicielle en passant du langage assembleur à des langages de plus haut-niveau comme le C ou le Java.

Nous nous intéressons ici à une de ces méthodologies haut-niveau, la méthodologie de conception au niveau du système électronique (ou ESL - Embedded System Level). Celle-ci consiste en une approche dite du « haut vers le bas », commençant la conception par une modélisation en langage haut-niveau du système, au niveau de l'algorithme, sans considération sur les détails d'implémentation tels que les communications. Ceci est ensuite suivi par une descente vers les détails et optimisations pour l'implémentation par raffinement successifs.

Cette approche se démarque d'une méthodologie plus traditionnelle du « bas vers le haut », où l'on va concevoir nos accélérateurs et composants logiciels directement à bas niveau, puis les intégrer dans le système complet par la suite.

Un premier aspect de cette méthodologie est l'exploitation de la technologie de synthèse de matériel à haut niveau (ou HLS, pour High-Level Synthesis). Cette technologie permet d'accélérer drastiquement le développement des modules du système implémentés en matériel,

en permettant de synthétiser des architectures matérielles directement depuis du code source haut-niveau, tels que le C/C++. Des travaux ont effectivement démontré des gains de productivité d'un facteur de quatre par rapport à une méthodologie RTL de plus bas niveau, avec une qualité similaire de l'architecture matérielle finale [5].

Le second aspect important est l'exploitation de plateformes virtuelles, qui fournissent une simulation du système SoC-FPGA développé au complet. Ainsi, on simule à la fois le processeur grâce à un simulateur de jeu d'instructions, mais aussi les accélérateurs matériels destinés à être implémentés sur le FPGA. Ce modèle de simulation, d'abord de très haut niveau, ne simulant le système qu'à un niveau purement comportemental grâce au modèle algorithmique initial, sans notion réelle de performance temporelle, peut ensuite être raffiné successivement pour introduire des notions de temps approximatif, en simulant par exemple les communications au niveau de la transaction sur le bus (modélisation TLM pour Transaction-Level Modelling) voire précis au cycle près, se rapprochant alors du modèle RTL, afin de mieux estimer les performances du système une fois implémenté. Une telle succession de raffinement est illustrée dans la figure 2.2.

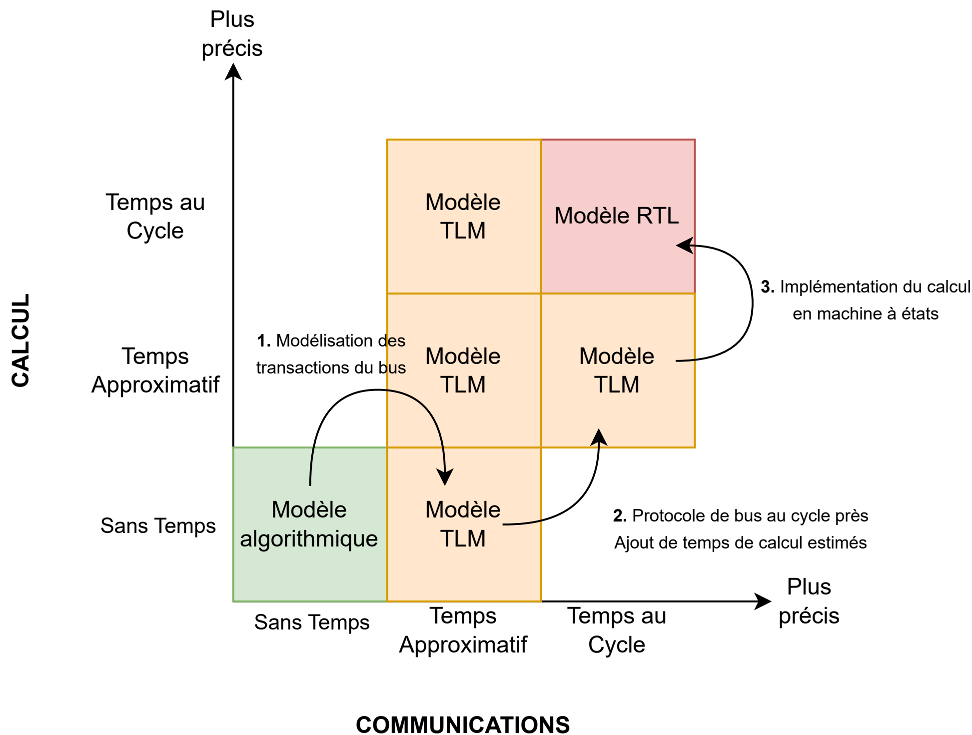

FIGURE 2.2 Illustration schématique des niveaux de raffinement de la modélisation d'un système, avec un exemple de raffinements successifs possibles

La simulation du système permet également d'avoir une exécution dite en « boîte blanche »

d'un système, qui permet d'y déceler des bogues bien plus facilement que sur une implémentation, où l'exécution est bien plus opaque, particulièrement au niveau du FPGA, où l'analyse de l'état du système lors de l'exécution pour le déboguer est particulièrement complexe.

Le modèle de simulation accélère également le processus de conception, puisqu'il permet d'effectuer les premiers choix techniques et de commencer le développement du système même sans avoir accès à la plateforme matérielle, et d'éviter ainsi l'attente pour obtenir des prototypes physiques, qui peuvent se révéler dispendieux et complexes d'accès.

L'environnement SpaceStudio

SpaceStudio est un environnement de développement implémentant la méthodologie ESL [6]. Celui-ci propose, à partir d'une même base de code en C/C++ (exploitant la bibliothèque SystemC), de traverser toutes les étapes de conception d'un système. En partant de l'algorithme conçu en langage de haut-niveau, on peut ensuite perfectionner le partitionnement des blocs entre logiciel et accélérateur matériel (le code étant synthétisé en matériel grâce au HLS, le déplacement d'un bloc fonctionnel du logiciel au matériel est facilité), puis effectuer une validation fonctionnelle et des tests de performance grossiers grâce à un modèle de simulation, et enfin générer automatiquement les binaires d'implémentation du système complet.

La plateforme virtuelle SpaceStudio présente une particularité par rapport aux autres présentes sur le marché dans le fait qu'elle est dynamique, c'est-à-dire qu'elle va automatiquement s'adapter aux choix faits pour l'application, sans besoin d'apporter manuellement des modifications au modèle. En comparaison, sur une plateforme virtuelle fixe, on conçoit d'abord le système matériel, puis on en réalise un modèle sous forme d'une plateforme virtuelle, et c'est seulement après cette étape que le développement de l'application commence. Ainsi, un changement de besoin de l'application (un changement de partitionnement matériel-logiciel suite à un défaut de performances, par exemple) va nécessiter de retravailler la plateforme virtuelle en adéquation avec ce changement, impliquant une perte de temps, qui pourrait être importante. De l'autre côté, une plateforme virtuelle dynamique est adaptée automatiquement en fonction des besoins de l'application. Cela facilite grandement l'exploration architecturale, et permet de paralléliser encore davantage les phases de conception du matériel, réduisant le temps global de conception.

Sur le plan technique, dans SpaceStudio, cette plateforme virtuelle dynamique est réalisée par une utilisation conjointe de deux outils. D'abord, la partie logicielle repose sur QEMU qui est un simulateur de jeu d'instructions généraliste très performant grâce à son usage de la traduction de binaire dynamique. Ses performances permettent la simulation efficace

de systèmes complexes, comme ceux embarquant un système d'exploitation basé sur Linux, cependant au détriment d'une précision qui n'est plus possible au cycle près (on pourrait se considérer dans la catégorie des simulateurs au cycle approximatif). De l'autre côté, au niveau matériel, la simulation se fait sous la forme de l'exécution directe du modèle SystemC, avec une modélisation des transactions avec le logiciel au niveau TLM, représentant donc toute la flexibilité possible du FPGA sur la plateforme physique.

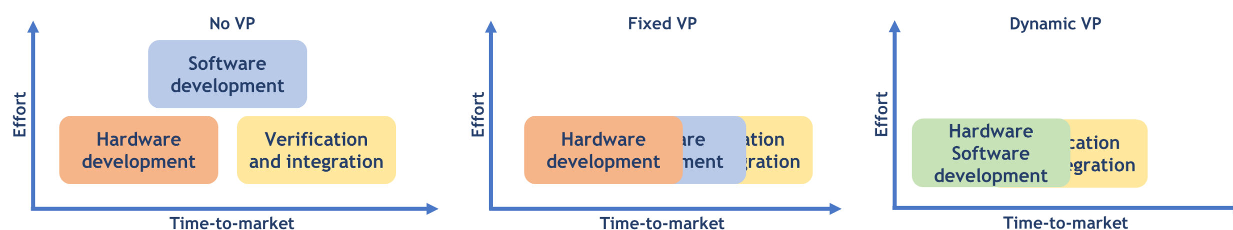

FIGURE 2.3 Illustration schématique des avantages des plateformes virtuelles en terme d'efforts en fonction du temps de mise sur le marché

2.1.2 Hyperviseurs de partitionnement

Les interférences entre applications de criticités mixtes précédemment mentionnées en 2.1 peuvent être classées dans deux principales catégories : temporelles et spatiales. Les interférences temporelles sont celles concernant une répartition inappropriée du temps de calcul, un service peu critique pouvant alors monopoliser le temps processeur et bloquer l'exécution d'une tâche plus critique (de façon similaire à une inversion de priorité sur un système multitâche).

Les interférences spatiales, elles, concernent des accès mémoire qui pourraient modifier le comportement d'une autre tâche, par exemple une écriture erronée à un pointeur qui causerait un plantage d'une autre application (possiblement plus critique).

La manière la plus commune de pallier ces interférences est par le biais d'un outil de virtualisation appelé hyperviseur. De manière générale, ceux-ci orchestrent l'exécution de plusieurs systèmes d'exploitation (niveau de privilège « superviseur » d'où le nom) sur un même matériel, de manière plus ou moins transparente pour les systèmes d'exploitation dits « invités ». C'est par exemple eux qui permettent l'exécution de machines virtuelles sur les systèmes d'exploitation modernes.

On distingue deux types d'hyperviseurs [7], dont les différences architecturales sont illustrées à la figure 2.4. Ceux dits de type 2 existent au-dessus d'un système d'exploitation hôte et n'interagissent par conséquent pas directement avec le matériel, limitant ainsi leurs capacités

et performances à ce que ce dernier permet. Nous nous intéressons cependant ici plus aux hyperviseurs dits de type 1. Ceux-ci exploitent directement le matériel, se plaçant en dessous de tout système d'exploitation du système. Ainsi, les systèmes invités n'ont toujours pas une vue directe du système, devant encore passer par les services de l'hyperviseur pour traiter avec le matériel, mais la couche hyperviseur de type 1 est normalement bien plus légère que le recours à système d'exploitation complet, ses services fournissant des abstractions de niveau plus bas.

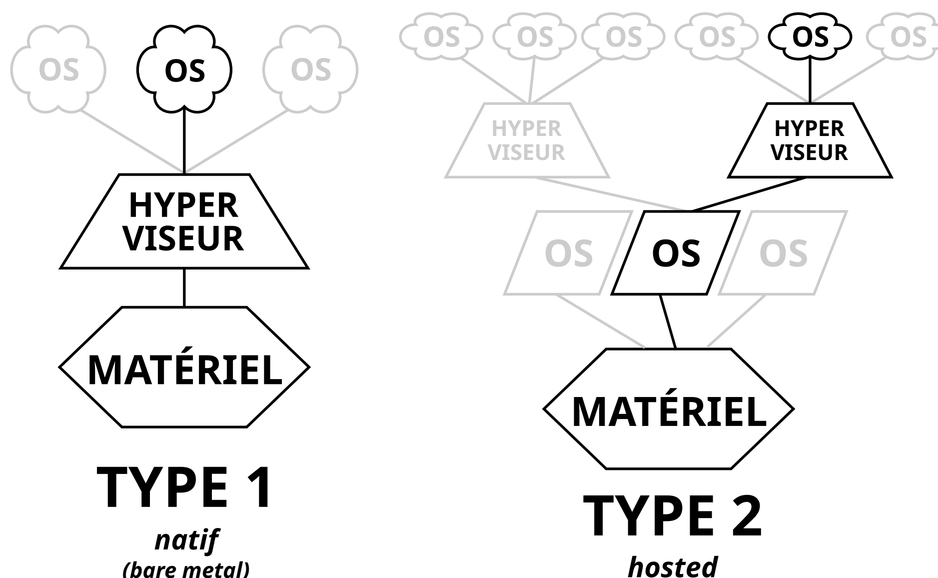

Figure 2.4 Organisation des deux différents types d'hyperviseurs

Les hyperviseurs dits de partitionnement, aussi appelés noyaux d'isolation, exploitent ce contrôle complet sur le matériel afin de fournir, comme leur nom l'indique, une isolation entre les systèmes invités. Ceux-ci sont largement utilisés dans le domaine aéronautique, où ces critères d'isolation, ainsi que les API de tels hyperviseurs de partitionnement sont fixés par la norme ARINC-653 [8]. Celle-ci définit une isolation déclinée sur deux dimensions : spatiale et temporelle.

L'isolation spatiale assure la protection des plages mémoires utilisées par chaque application, renforçant à la fois la sécurité et le risque d'erreurs dues à une application dysfonctionnelle en perturbant une autre partition (possiblement plus critique). D'un autre côté, l'isolation temporelle assure la répartition du temps de calcul du processeur entre les OS invités, empêchant une monopolisation du temps de calcul disponible par une seule tâche qui pourrait empêcher une tâche critique de s'exécuter dans un délai correct.

Ces mécanismes d'isolation reposent sur des fonctionnalités de virtualisation et de protection

mémoire directement implémentées en matériel dans le processeur. Cela assure donc un faible coût en ce qui a trait aux performances, ainsi qu'une grande robustesse.

2.1.3 Interférences dans les systèmes hétérogènes

Nous avons vu que les hyperviseurs de partitionnement apportent une fiabilité très élevée au niveau logiciel. C'est sans compter, cependant, sur leur utilisation dans des systèmes hétérogènes matériel-logiciel, qui amène des nouvelles problématiques.

En effet, le partitionnement assuré par l'hyperviseur s'arrêtant au niveau logiciel, les interactions avec les accélérateurs matériels implantés dans le FPGA ne sont pas isolées. Cela peut donc devenir un facteur pouvant mener à des interférences entre les partitions logicielles, et, dans le pire des cas, briser les contraintes de temps-réel d'une partition hautement critique.

Des travaux se sont déjà penchés sur certains de tels facteurs d'interférences. Un de ces facteurs est une contention possible au niveau de l'interconnexion, qui causerait une hausse des latences d'un accélérateur exploité par une partition. Par exemple, [9], démontre sur une interconnexion AXI [10], standard de-facto sur les systèmes SoC-FPGA, qu'en jouant sur la longueur de rafale (« burst size » en anglais), il est possible, avec les mécanismes d'arbitrage de bus standard, d'obtenir une répartition totalement inégale de la bande-passante du bus entre plusieurs accélérateurs matériels de manière ciblée sur un maître précis. Cela a des répercussions évidentes sur le temps de réponse d'un accélérateur, et peut causer des violations de contraintes temps-réel qui, sur une application critique, pourrait avoir de graves conséquences. D'autres travaux [11, 12] montrent la variété des facteurs de contention possibles lors du partage d'un FPGA par de multiples applications.

Des travaux ont par la suite mené à des solutions pour les problèmes soulevés précédemment, notamment en créant un module d'interconnexion adapté aux contraintes d'isolation des hyperviseurs de partitionnement, appelé AXI HyperConnect [13]. Cependant, cela ne règle pas entièrement la problématique de contention une fois pour toutes. En effet, ce nouveau module peut certes remplacer les interconnexions implémentées sur le FPGA vu que celui-ci est programmable, mais il reste néanmoins les points de contention que sont l'interconnexion centrale entre les processeurs et le FPGA ou encore la mémoire centrale partagée, qui sont implémentées « en dur », et dont la politique d'arbitrage ne peut pas être modifiée à volonté.

Ainsi, on peut conclure que l'analyse des problèmes d'interférences reste importante lors de la conception d'un système hétérogène de criticités mixtes.

2.1.4 Outils diagnostics actuels et lacunes pour les méthodologies haut-niveau

Les interférences de partitionnement restent cependant globalement un problème difficile à détecter et diagnostiquer. En effet, ces problèmes sont difficiles à prévoir à l'avance via par exemple de l'analyse statique du code source, de par le fait que leur apparition est liée à un mélange complexe de facteurs durant l'exécution.

Par ailleurs, comme la contention découle de comportements précis du bus en termes de timing, il est difficile de la modéliser dans la simulation à haut-niveau par transaction tout en gardant des performances de simulation satisfaisantes.

On voit donc tout l'intérêt d'outils d'analyse permettant d'extraire des métriques lors de l'exécution de l'application sur le système, possiblement sans l'interrompre ou le perturber. À cet égard, le côté logiciel possède plusieurs outils de profilage, de traçage et de surveillance de performances à l'exécution, tels que Perf, GProf ou encore LTTng [1]. Cependant, lorsqu'une partie du calcul est déportée en FPGA, les outils logiciels ne suffisent plus à eux seuls, car ils ne peuvent pas analyser précisément le comportement interne du FPGA, leur vue du matériel se résumant globalement à des délais d'accès aux interfaces projetées en mémoires. Aussi, les outils du côté matériel ne sont pas adaptés à la méthodologie ESL, étant donné que la synthèse haut-niveau, base de la méthodologie pour la synthèse de matériel depuis un langage haut-niveau, accuse d'un important manque d'outils adaptés [14].

Nous allons donc dans les prochains paragraphes analyser les lacunes des outils existants d'analyse de performances en matériel, et passer en revue les outils adaptés au HLS développés dans d'autres travaux, afin de déceler les besoins pour un nouvel outil plus adapté à les résoudre.

Les analyseurs de logique embarquée Les outils traditionnels d'analyse de performances sur FPGA sont les analyseurs de logique embarqués (ou ELA pour Embedded Logic Analyzer). Les deux fournisseurs majeurs de matériel programmable supportent leur propres solutions : l'« Integrated Logic Analyzer » de Xilinx [15] et le système SignalTap II de Intel [16]. Un ELA fonctionne en intégrant un module matériel qui va échantillonner les valeurs des signaux auxquels il est connecté et les sauvegarder dans une mémoire embarquée dans le FPGA, soit continuellement soit via un mécanisme de déclenchement sur certaines conditions (à la manière d'un oscilloscope). Les données sont ensuite rapatriées vers le processeur ou vers les ports d'entrée-sortie de débogage pour être analysées par le développeur.

Cette approche présente d'abord le problème d'utiliser des ressources du FPGA déjà contraintes, particulièrement les blocs de mémoire embarquée si l'on demande une fenêtre d'échantillonage

adéquatement large, ce qui peut empêcher l'analyse d'applications gourmandes en ressources, puisque le système à déboguer ne rentrerait plus dans les ressources après addition du module ELA.

La seconde limitation de cette approche, sûrement la plus importante, est le fait qu'on analyse directement les signaux numériques dans le FPGA. Il faut donc pour cela connaître précisément quels signaux analyser. Ceci est parfaitement adapté à une méthodologie RTL, car la correspondance entre un signal RTL et son équivalent post-implémentation matérielle est assez évidente, moyennant quelques optimisations. En revanche, pour une méthodologie basée sur la synthèse haut-niveau, le code source de l'accélérateur subit de lourdes transformations avant d'atteindre une implémentation FPGA, qui va rendre la sélection des signaux à analyser et leur interprétation très complexe, et ce pour deux raisons.

Premièrement, une seule variable dans le code HLS va rarement être synthétisée en un seul signal dans le code RTL généré par l'outil de synthèse HLS. La synthèse matérielle et les optimisations subséquentes pourraient même simplifier le signal en une constante et le faire entièrement disparaître de l'implémentation finale.

Ensuite, même si la correspondance entre les données du code haut-niveau et les signaux en implémentation était facile à déterminer, cette méthode d'analyse requiert l'analyse de formes d'ondes complexes afin de comprendre la source d'un potentiel problème de performances. Par exemple, l'analyse de protocoles de communication complexes tels que AXI4, qui implique des dizaines de signaux pour le transfert de données, peut se révéler très complexe même pour un ingénieur matériel expérimenté, et pour ainsi dire impossible pour un développeur plus orienté logiciel.

Ainsi, les outils de l'état de l'art pour le débogage matériel habituellement utilisés pour la méthodologie RTL sont inadaptés au diagnostic de problèmes de performance de code haut-niveau destiné à la synthèse de matériel, en raison de leur approche trop bas niveau. Cela montre bien le besoin d'outils spécifiquement destinés à une méthodologie plus haut-niveau.

Outils d'instrumentation de haut niveau pour l'évaluation des performances Plusieurs travaux ont proposé des outils d'instrumentation permettant une analyse de performances dans le FPGA qui s'insèrent directement à haut niveau dans le code HLS, tentant ainsi d'atteindre le niveau d'abstraction que l'on retrouve du côté des outils logiciels.

Un premier outil qui fut proposé est SoCLog [17]. Il permet une journalisation d'évènements définis par l'utilisateur à une précision au cycle près, via une API en C pour la synthèse haut-niveau. L'outil propose également une analyse de l'utilisation du bus en scrutant les signaux du protocole AXI. Les données de journal sont collectées en mémoire et peuvent être

analysées à l'exécution par le logiciel sur le processeur. Cependant, cet outil se concentre uniquement sur les aspects matériels, donc l'outil ne peut pas être utilisé en logiciel, et la comparaison entre des points de temps matériel et logiciel n'est pas possible, limitant l'utilité de l'outil dans une méthodologie ESL.

Un autre outil développé est HLScope [18]. Celui-ci permet de mesurer le temps d'exécution de sections arbitraires de code haut-niveau directement en implémentation. Le temps d'exécution peut également être estimé en simulation, avec une précision raisonnable sur les exemples présentés. En revanche, l'analyse des résultats ne peut se faire qu'après l'exécution, et l'application elle-même ne peut manipuler les données de performances, ce qui limite grandement les possibilités de cet outil.

Enfin, un travail similaire propose une instrumentation de performance pour les accélérateurs matériels développés en OpenCL [19]. L'outil insère des points d'instrumentation légers directement dans le code OpenCL pour générer une trace précise au cycle près de l'exécution de l'accélérateur sur le FPGA. L'API OpenCL pouvant cibler le logiciel aussi bien que du matériel, on pourrait penser pouvoir utiliser ces outils dans les deux domaines, mais le fait que le comportement des instruments soit spécifié en RTL rend la portabilité impossible.

Les outils présentés jusqu'à présent dans la littérature présentent donc des méthodologies très spécifiques à la synthèse haut niveau, qui sont largement incompatibles avec le grand écosystème logiciel existant. Or, cette incompatibilité des bibliothèques existantes avec la synthèse haut-niveau constitue un facteur limitant important à l'adoption plus large de la synthèse de matériel à haut-niveau dans l'industrie [14], car elle complexifie la formation des ingénieurs logiciels, qui doivent se familiariser avec de nouveaux concepts, et augmentent la charge de travail nécessaire au portage de code existant pour cibler des synthétiseurs HLS.

Une solution possible aux problèmes évoqués serait de proposer une abstraction unifiée pour la mesure de performances sur le matériel et le logiciel. Un travail précédent [20] s'y est penché en adaptant une API logicielle existante pour accéder à des compteurs de performances sur le FPGA, en plus des compteurs de performances existants du processeur. Cependant, cette extension se concentre sur l'accès de compteurs matériels en logiciel, et n'instrumente pas directement le code HLS, se limitant aux compteurs existant déjà dans le module matériel. Cela limite la visibilité de débogage et nécessite du travail supplémentaire en RTL pour construire les instruments à la main.

2.1.5 Lacunes de la simulation des systèmes avec hyperviseurs

La simulation logicielle d'un système est, comme expliqué en 2.1.1, un élément crucial dans la méthodologie ESL, et donc un premier pas vers l'adaptation de celle-ci aux systèmes à criticités mixtes utilisant des hyperviseurs pour le partitionnement de leurs applications. Cependant, une exploration préalable montre que deux de ces hyperviseurs communément utilisés en aérospatial, XtratuM [21] et Xng, une version commerciale améliorée du précédent, ne parviennent pas à s'exécuter sur le simulateur QEMU.

Des simulateurs bas-niveau (notamment la simulation RTL au cycle près du processeur et du matériel) parviennent à exécuter ce système, mais avec des performances très faibles inadaptées à la simulation d'un système complexe au complet, contrairement à QEMU, qui, grâce à sa simulation haut-niveau, atteint d'excellentes performances même en simulant un système très complexe. Ce problème de performance est principalement dû à des manquements dans la simulation des mécanismes de virtualisation, sur lesquels les hyperviseurs reposent.

2.2 Objectifs

Ainsi, au vu des deux obstacles que sont le faible support des plateformes virtuelles pour les hyperviseurs et le manque d'outils d'instrumentation adaptés à une méthodologie de conception à haut-niveau, nous proposons d'y pallier en développant des outils aidant au développement de telles plateformes matériel-logiciel. Ceci se fera en deux temps.

D'abord, nous proposons le développement d'outils d'instrumentation pour l'analyse de performances unifiés, c'est-à-dire fournissant une abstraction commune à la fois en matériel et logiciel, s'intégrant donc mieux à la méthodologie ESL, et plus particulièrement à celle de SpaceStudio, et permettant, entre autres, une validation plus aisée du partitionnement en cas de criticités mixtes.

Puis par la suite, nous proposons une simulation logicielle plus complète de tels systèmes, apportant un support pour la virtualisation imbriquée plus poussé au simulateur QEMU. Ce mécanisme nécessaire à la simulation d'un hyperviseur, facilite ainsi l'intégration des hyperviseurs à une méthodologie ESL.

CHAPITRE 3 PREMIER THÈME - OUTIL D'INSTRUMENTATION DE PERFORMANCES

3.1 Spécification des requis

L'outil développé utilisera une API en C/C++. Cela permettra la compatibilité avec les systèmes HLS les plus répandus, notamment Vivado HLS de Xilinx et Catapult HLS de Siemens, et donc une intégration plus aisée dans la méthodologie ESL utilisée par SpaceStudio. La spécificité de l'outil que l'on souhaite développer est son universalité. Ainsi, il doit pouvoir être utilisé de manière identique au niveau du code quelle que soit la cible du module, matérielle ou logicielle. Toute différence découlant de spécificités de la cible doit donc être abstraite par l'outil via divers procédés automatisés afin d'être complètement invisible pour l'utilisateur.

Estampilles de temps

Pour ce travail, la fonctionnalité au cœur de l'outil proposé a été restreinte à la récupération et la comparaison de points de temps – c'est à dire un objet abstrait représentant un instant précis dans un référentiel temporel quelconque – ce qui permet de calculer divers délais d'exécution dans le code par l'utilisateur.

L'outil est en cela flexible, car il permet à l'utilisateur de collecter des informations de timing et les traiter à son désir. Ces points de temps seront extraits de différents référentiels selon la cible (par exemple les minuteries du processeur en logiciel), mais respecteront au moins des critères basiques que sont la monotonie (leur valeur ne peut pas revenir en arrière), et de précision relativement élevée.

Domaines d'horloge

Les points de temps collectés sont compartimentés en domaines, qui correspondent à des ensembles de points comparables entre eux. En effet, les points de temps pourront avoir une représentation interne, ou estampille de temps, différente sur chaque cible (par exemple sur Linux le nombre de nanosecondes depuis le 1$^{\text{er}}$ janvier 1970 sur un entier 64-bit signé).

Par conséquent, une estampille de temps issue du matériel et une issue du processeur (ou même de différents cœurs du processeur) ne sont pas comparables entre elles sans conversion explicite dans un référentiel commun (en secondes depuis un point donné par exemple). Cette opération peut se révéler très coûteuse, particulièrement en matériel, puisqu'elle implique

souvent des calculs sur des nombres exprimés en virgule flottante. Ainsi, l'API prendra soin de mettre en place des mécanismes marquant la séparation entre ces domaines afin d'éviter des mélanges accidentels de domaines d'horloge incomparables, sauf sur demande explicite de l'utilisateur de convertir un point de temps en secondes.

Les points de temps peuvent être convertis en secondes. Cependant, pour des raisons de simplicité nécessaire, particulièrement en matériel, cette conversion n'a de sens que dans le référentiel du domaine, donc convertir un point de temps seul le convertira en secondes depuis son époque. Par conséquent, cette conversion n'a d'utilité que pour comparer des délais entre domaines, et on ne peut pas directement comparer deux points de deux domaines différents.

Ces domaines correspondront donc, en matériel, aux ensembles de modules reliés à une même fréquence d'horloge, et en logiciel à chaque cœur du processeur. Le logiciel ne présente pas un unique domaine puisque les minuteries des différents cœurs ne sont pas nécessairement synchronisées.

Par ailleurs, l'outil permettra les accès à des points de temps dans un domaine externe à celui du module depuis lequel on fait appel à l'API dans certains cas, notamment entre deux domaines différents sur le matériel (deux zones du FPGA connecté à des horloges différentes), ou des points de temps matériels depuis le logiciel (à des fins par exemple de calibration des conversions en secondes).

Cibles

Enfin, les outils seront portables à diverses plateformes, en logiciel, on cible les systèmes d'exploitation Linux et MicroC, ainsi que directement sur le processeur en baremetal. En matériel, les outils fonctionnent en implémentation sur l'outil de synthèse Vivado HLS, avec un support futur prévu pour Catapult HLS, et sont également adaptés sur la simulation haut-niveau du matériel avec SystemC.

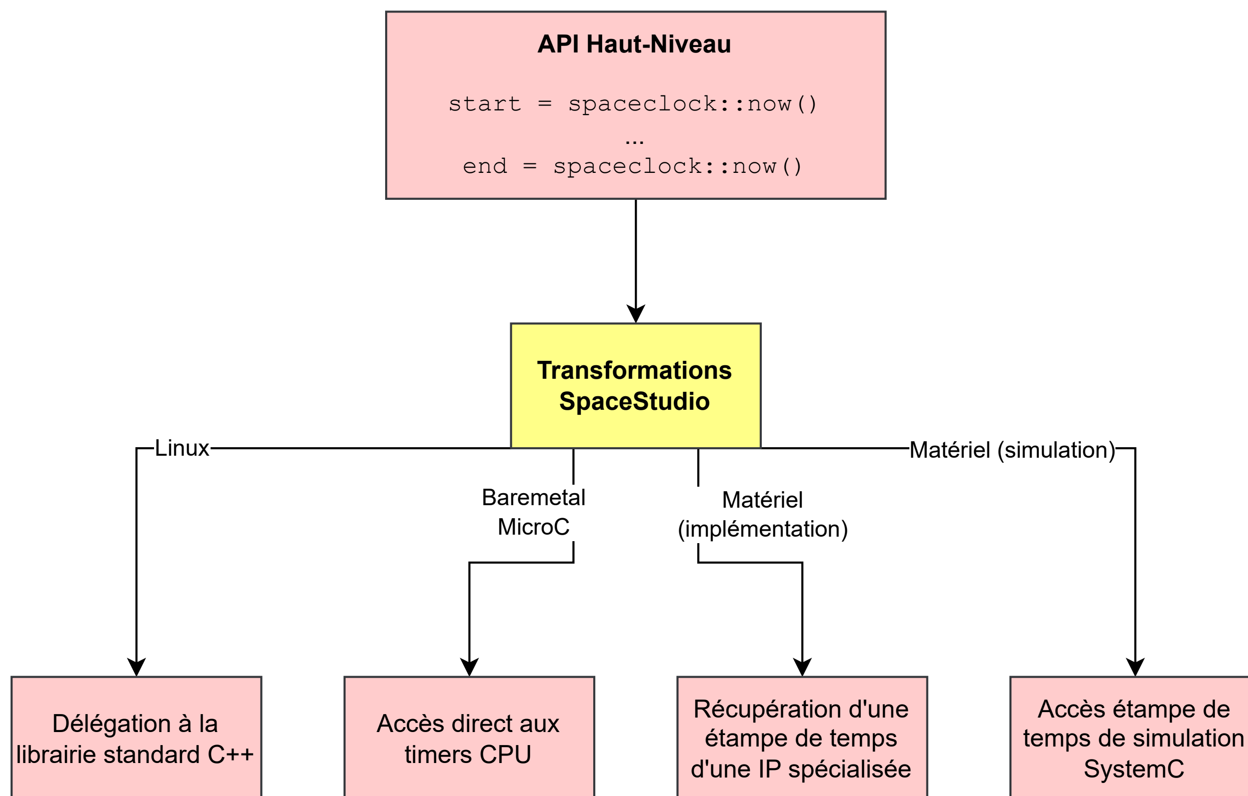

FIGURE 3.1 Architecture du routage des accès aux estampilles de temps sous-jacentes à un point de temps selon la cible

Consommation de ressources et performances

Un objectif important est également que les outils soient aussi légers que possible. Ainsi, en matériel, ils devront utiliser un minimum de ressources sur le FPGA en implémentation, et en logiciel, il faudra ajouter le moins de surcoûts par rapport à l'utilisation des appels standard du système d'exploitation.

3.2 Architecture et implémentation

L'API C++ s'inspire de celle de la librairie standard (`std::chrono`). Cela facilite grandement son usage pour les développeurs logiciels y étant déjà habitué.

Les outils reposent sur un usage important des templates C++. On voit par exemple dans 3.1 aux lignes 3 et 6 que le domaine d'horloge auquel on souhaite accéder est fourni en paramètre de template. Ainsi la destination des accès est déterminée statiquement dès la compilation, ce qui va non seulement économiser du temps de débogage en cas d'erreurs, mais est également crucial pour les outils HLS, puisque ceux-ci requièrent une définition explicite de toutes les

Listing 3.1 Exemple simple de l'utilisation de l'accès à un point de temps

```
1        DataReceive(&buf, DATA_SIZE);
2
3        auto c_start = clk_t<SPACE_ID_SELF>::now();
4        ProcessData(&buf);
5        auto c_end = clk_t<SPACE_ID_SELF>::now();
6        auto compute = c_end - c_start;
7
8        DataSend(&buf, DATA_SIZE);
9
10       double compute_secs = compute.to_sec<double>();
```

interfaces d'un module à la compilation, y compris les interfaces vers le module matériel fournissant la référence de temps du domaine.

Les templates, associés à un système d'analyse statique et de génération de code, permettent le routage des lectures de temps vers le canal correct, selon qu'on lit une étampe matérielle ou logicielle, en implémentation ou en simulation, ou depuis le matériel ou le logiciel.

Génération de code

Les outils reposent lourdement sur la génération de code pour leur fonctionnement. Celle-ci nous permet de cacher les abstractions permettant l'accès à un système unifié d'estampilles de temps. Ainsi par exemple les classes d'accès aux horloges visibles par l'utilisateur sont automatiquement générées pour hériter de la classe qui utilise l'interface correcte pour lire le point de temps selon que l'on soit en matériel ou en logiciel.

Ensuite, l'association des interfaces selon la destination est générée en créant des spécialisations explicites de templates de base avec toutes les combinaisons de paramètres valides (destination, domaine). Cela permet de vérifier que l'utilisateur fait bien appel à des destinations existantes dans son design directement à la compilation, évitant de possibles erreurs accidentelles.

Un aspect important de ce choix de conception, conjointement avec le système de templates de C++, est que le chemin de code emprunté pour effectuer l'accès est connu dès la compilation, et non résolu à l'exécution. Ceci est important pour le support des outils HLS, qui ne peuvent généralement pas synthétiser efficacement un chemin de données qui serait déterminé à l'exécution.

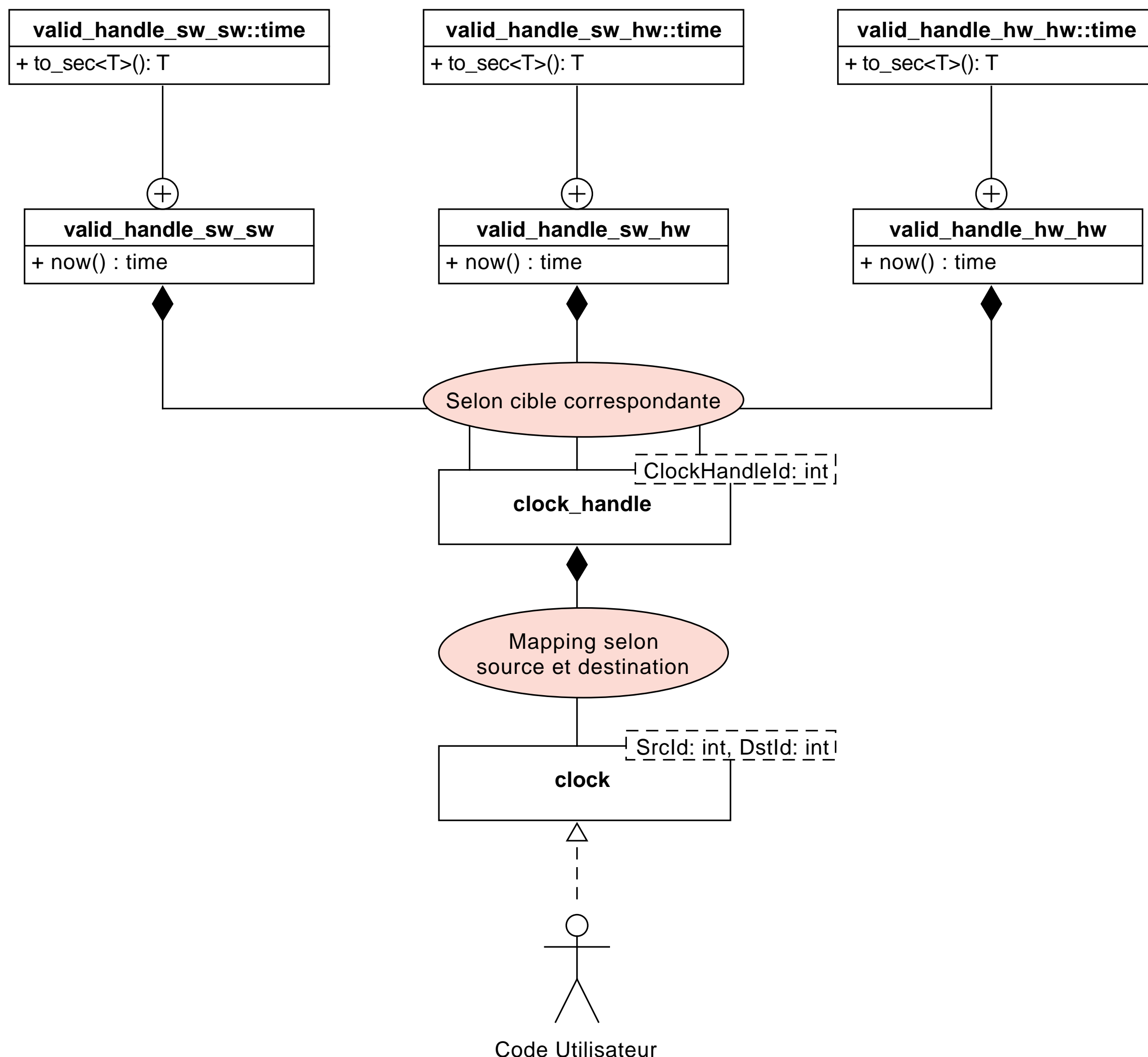

FIGURE 3.2 Diagramme de classe simplifié de l'API d'estampille de temps. En rouge clair sont les sections déterminées par de la génération de code automatique

Références de temps logicielles

Pour atteindre les objectifs de très faibles surcoûts en logiciel par rapport aux accès aux estampilles de temps fournies par le système, notre outil va, en cas d'accès, simplement rediriger l'accès vers les méthodes du système d'exploitation de la cible. Sur Linux par exemple, l'appel est directement converti à un appel à la librairie standard `std::chrono`. On peut supposer que le surcoût de l'appel intermédiaire, très simple, et étant effectué par des templates déduites à la compilation, devrait être simplifié par le compilateur à la manière d'une défini-

tion du préprocesseur et ne causer aucun délai supplémentaire dans la lecture par rapport à l'usage direct de l'API logicielle.

En simulation de matériel, on fait simplement suivre l'appel d'accès à un point de temps à une récupération du temps écoulé du simulateur SystemC, causant un surcoût également négligeable considérant les optimisations du compilateur C++.

Horloge matérielle

La complexité de trouver une référence pour obtenir des estampilles de temps arrive en matériel, étant donné qu'aucune infrastructure de minuterie n'est embarquée par défaut. Il nous faut donc intégrer un module fournissant nos estampilles de temps de manière précise et atomique dans le FPGA.

Les modules de minuterie existantes présentent généralement leur valeur à travers une interface mappée en mémoire (de type AXI ou AXI-Lite par exemple), ce qui introduit des délais, des problèmes de contention en raison du passage par une interconnexion, et donc des difficultés à garantir l'atomicité des accès aux estampilles de temps.

Nous avons donc conçu un module d'horloge sur mesure, nommé Spaceclock, pour régler ce problème. De conception relativement simple afin d'utiliser le moins de ressources possible, celui-ci fournit simplement une estampille de temps de 64 bits, instanciée à zéro à la réinitialisation, et incrémentée à chaque front de l'horloge à laquelle il est connecté (qui est implicite sur le diagramme 3.3). Ainsi, un module Spaceclock est instancié par domaine d'horloge distinct configuré sur le FPGA.

Il présente son estampille de temps en parallèle sur un nombre personnalisable d'interfaces AXI-Stream de 64 bits de large, à raison d'une par module en matériel qui effectue une lecture d'un point de temps. On les voit groupées en `stream_master` sur le diagramme de la figure 3.3, connectées aux interfaces auto-générées `spaceclock_stream_slave` des modules utilisateurs. Ce protocole est utilisé en raison de sa simplicité qui permet de réduire la logique de lecture et ainsi utiliser moins de ressources matérielles et assurer l'absence de latence qui pourrait fausser les lectures.

Une interface AXI-Lite (`axi_slave` sur le diagramme 3.3) est également fournie afin de permettre au processeur d'effectuer des lectures des étampes fournies, et ainsi par exemple calibrer la conversion en secondes de celles-ci en comparaison aux minuteries système.

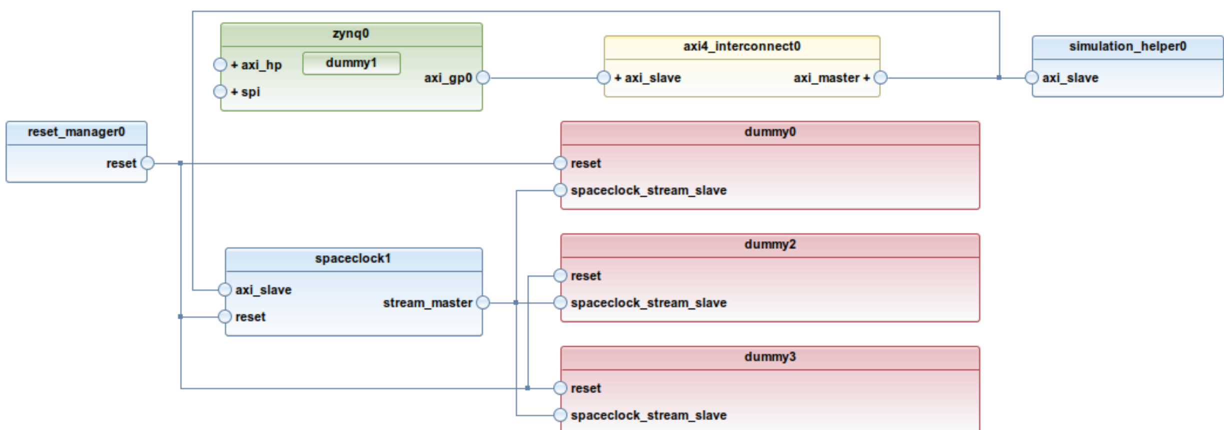

FIGURE 3.3 Exemple de design simpliste illustrant les connexions du module Spaceclock

Pour des raisons d'efficacité en termes de ressources et de timing, le module Spaceclock a été écrit directement au niveau RTL, en VHDL. Cela permet une manipulation directe du signal d'horloge, complexe en code HLS, et l'adaptation de la logique des protocoles aux interfaces. Par exemple, le système de handshaking de AXI-Stream a été simplifié, puisque des données seront toujours disponibles du côté du module Spaceclock.

Une fonctionnalité de verrouillage de l'horloge a également été prévue, pour permettre à de futures évolutions d'effectuer des lectures atomiques plus efficaces via le port AXI-Lite. En effet, comme le bus peut avoir une largeur plus faible que celle de l'estampille de temps. À titre d'exemple, pour un bus de 32 bits encore très utilisé, deux lectures sont nécessaires pour lire une estampille de temps de 64 bits.

Dans la version actuellement proposée, n'exploitant pas ce verrouillage, il faudra, pour s'assurer que la lecture est atomique, suivre un protocole effectuant au moins trois lectures, d'abord du mot le plus signifiant, puis du mot le moins signifiant, et relire le mot supérieur pour s'assurer qu'il est égal à la première lecture et qu'un dépassement de capacité du mot le moins signifiant au plus signifiant ne s'est pas déroulé entre les deux lectures. Le cas échéant, il faudra recommencer ce processus de lecture jusqu'à ce qu'un dépassement de capacité n'ait pas lieu. Ainsi, un délai de lecture non-déterministe peut s'ajouter et causer des imprécisions lors de la lecture.

Analyse statique

Afin d'effectuer la génération de code mentionnée précédemment, et également pour effectuer les diverses transformations de l'architecture nécessaires, par exemple l'ajout de ports sur le

module Spaceclock (où il faut même le générer si aucun module ne le requiert), il faut extraire diverses informations sur l'usage de l'API dans le code de l'utilisateur.

Initialement, SpaceStudio utilise déjà un système d'analyse statique basé sur les outils LLVM, afin de déterminer notamment les canaux de communication entre modules. Celui-ci a été adapté par un ingénieur de SpaceCodesign afin de fournir les informations liées aux accès Spaceclock. Ceux-ci fournissent ainsi l'ensemble des appels vers la méthode de récupération d'un point de temps `clock::now()`, agrémenté du domaine d'origine (le module lisant le temps) et de destination de la lecture.

Par ailleurs, diverses transformations du design de l'utilisateur sont effectuées en conséquence des résultats de l'analyse statique. On crée alors les ports d'accès à la minuterie sur les modules utilisateurs (`spaceclock_stream_slave` vu sur le diagramme 3.3) et on les connecte à l'horloge de domaine correspondante, sur laquelle a également été généré le port AXI-Stream maître correspondant (`stream_master` sur la figure 3.3).

Cette analyse permet en outre d'éviter de créer des connexions en trop, qui pourraient consommer des ressources supplémentaires sur le matériel si on se rend jusqu'à l'implémentation, et donc rendre l'instrumentation plus lourde que nécessaire.

3.3 Résultats expérimentaux

3.3.1 Environnement de test

Les tests et les analyses effectués dans cette section ont été réalisés avec comme cible le système SoC-FPGA Zynq-7000, précisément le XC7Z7020 embarqué sur le kit de développement Zedboard, avec le FPGA cadencé à une fréquence d'horloge de 100 MHz. Outre la suite d'outils SpaceStudio modifiée (faisant parti de la version de base 4.2.9), la synthèse de matériel se base sur les outils Vivado (incluant Vivado HLS) de Xilinx en version 2018.3, donc dans leur dernière version supportant la synthèse de code C++ basé sur SystemC.

3.3.2 Étude de la consommation de ressources matérielles du module Spaceclock

Montage expérimental

Nous proposons d'abord de vérifier un des objectifs des outils développés, soit de minimiser leur utilisation de ressources matérielles. On a ainsi établi un design de test simple, avec des modules factices qui ne font qu'effectuer une lecture d'un point de temps et l'envoyer vers un consommateur (afin d'éviter des optimisations qui supprimeraient tout simplement le module Spaceclock).

Nous allons également faire varier le nombre de ces modules lecteurs. En effet, la génération automatique ajoute un port au module Spaceclock par lecteur (car le protocole AXI-Stream, de par sa simplicité, gère difficilement plus de deux modules), et donc il contiendra plus de logique, qui utilisera plus de ressources à l'implémentation. Il faut donc s'assurer que cet usage reste raisonnable, même avec un grand nombre de lecteurs.

Analyse des résultats

	Nombre de lecteurs	**1**	**2**	**4**	**8**
Spaceclock	FF	104 (0.1%)	104 (0.1%)	104 (0.1%)	104 (0.1%)
	LUT	40 (0.08 %)	40 (0.08 %)	40 (0.08 %)	40 (0.08 %)
	BRAM	0	0	0	0
	DSP	0	0	0	0
Total	FF	1928 (1.81 %)	2255 (2.12 %)	2831 (2.55 %)	3981 (3.74 %)
	LUT	1784 (3.35 %)	1962 (3.69 %)	2254 (4.24 %)	2861 (5.38 %)
	BRAM	0.5 (0.18 %)	1 (0.36 %)	2 (0.71 %)	4 (1.43 %)
	DSP	0	0	0	0

Tableau 3.1 Consommation de ressources post-implémentation selon le nombre de modules connectés au module Spaceclock, en valeurs absolues et en proportion des ressources disponibles sur la Zedboard

On remarque que la consommation de ressources du module Spaceclock est très faible, et correspond à respectivement 0.08 % et 0.1 % des ressources de LUT et de Flip-Flops sur la carte relativement modeste que nous utilisons.

Par ailleurs, on voit que même en variant le nombre de lecteurs, les ressources occupées par la Spaceclock est identique. Cela peut s'expliquer par la simplicité du protocole de communication utilisé, qui ne nécessite même pas le handshaking de AXI-Stream, ce qui permet à l'optimisation en un seul signal connecté à tous les modules, ne requérant aucune ressource supplémentaire.

On peut donc conclure de ce test que l'objectif de légèreté niveau ressources matériel est bien atteint au niveau du composant Spaceclock lui-même.

3.3.3 Étude de la consommation de ressources matérielles de la manipulation des points de temps

Montage expérimental

La récupération de point de temps en elle-même est donc à l'évidence peu consommatrice en ressources. Il s'agit après tout d'une lecture d'un signal. C'est la manipulation de ceux-ci qui pourrait impliquer des coûts. Or, cette tâche est laissée à l'utilisateur dans cette architecture basique.

Ainsi, nous proposons ici une architecture de traitement des points de temps plus abstraite, implémentée comme des modules de l'utilisateur. Cette architecture se base sur des moniteurs de temps, c'est-à-dire des objets qui permettent de calculer des statistiques sur délais entre deux points du code, définis par un départ et une arrivée.

Le suivi de ces points se fait dans un module matériel d'agrégation de statistiques, qui reçoit des commandes de début et fin de sections des autres modules et effectue le calcul des statistiques désirées (minimum, maximum, total, moyenne) à travers un protocole de communication basique via un ensemble de registres. Ces statistiques se limitent à celles calculables en temps constant afin de limiter la consommation de temps et de ressources matérielles.

À la fin de l'exécution, le module d'agrégation renvoie ses données au processeur, qui les affiche à l'utilisateur via le port série de la carte.

Nous testons l'architecture sur un système utilisant un accélérateur de multiplication de matrices de flottants carrées déployé en matériel. L'application utilise quatre zones de mesure différentes : l'attente de données, la réception des données, le calcul, et le renvoi des données au processeur.

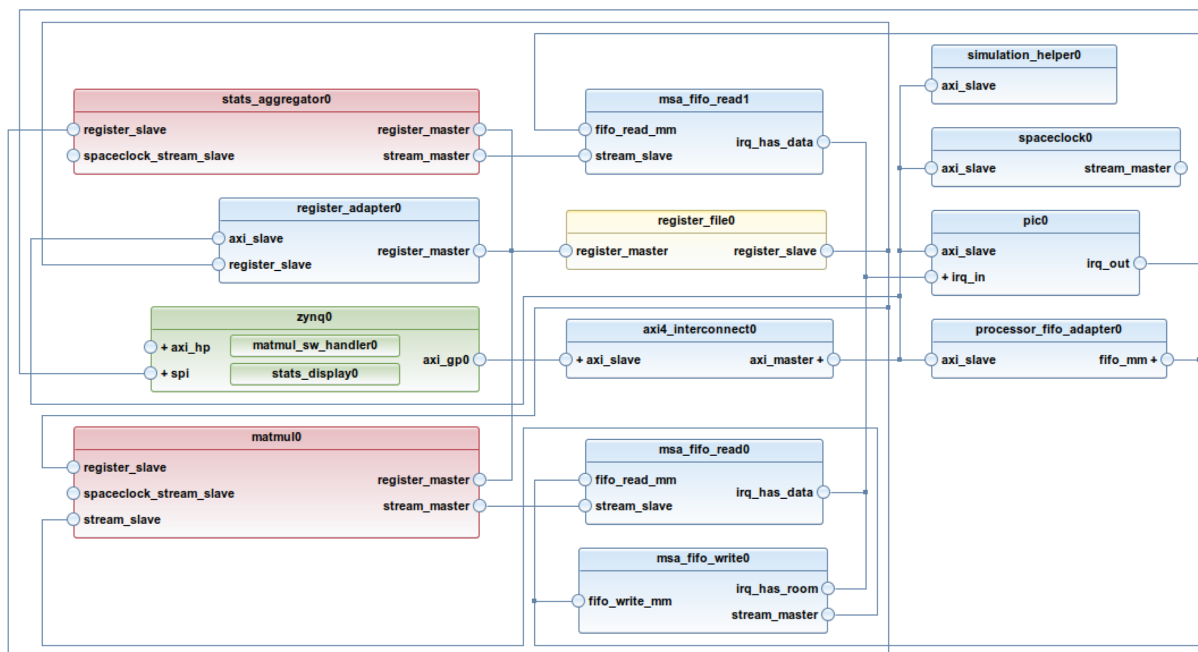

FIGURE 3.4 Architecture des outils abstraits de mesure de performance implémentés au niveau utilisateur sur un module d'accélération de multiplication matricielle

Analyse des résultats

Nous mesurons l'impact de ces outils plus abstraits via une estimation de leur consommation de ressources matérielles. Nous sélectionnons uniquement les modules liés aux outils, donc le module d'agrégation, le module Spaceclock, et les modules de communication. On obtient les données qui suivent :

Composant	FF	(%)	LUT	(%)	BRAM	(%)	DSP	(%)
`stats_aggregator0`	1531	1.44	1131	2.13	0	0	0	0
`register_file0`	520	0.49	247	0.46	0	0	0	0
`spaceclock0`	105	0.1	41	0.08	0	0	0	0
`msa_fifo_read1`	171	0.16	124	0.23	1	0.36	0	0
Total	2327	**2.19**	1543	**2.9**	1	**0.36**	0	**0**
Disponible	106400		53200		280		220	

TABLEAU 3.2 Consommation de ressources du système de moniteurs

On constate donc effectivement un usage plus important de ressources, montant jusqu'à presque 3 % des ressources de LUT de la carte. Cependant, ce surcoût reste globalement très acceptable. En effet, cette architecture est très naïve et ne compte aucune instruction d'optimisation de la synthèse, donc ce résultat est encourageant pour un développeur optimisant un tant soit peu le banc de test. Par ailleurs, nous effectuons nos tests sur une carte relativement modeste, donc la consommation sur une carte plus haut-de-gamme serait négligeable.

3.3.4 Application de l'outil à un cas d'interférence dans un système partitionné par hyperviseur

Montage expérimental

Afin de démontrer les capacités de détection de problèmes de performances de notre outil, nous avons concu un système générant artificiellement de la congestion sur le bus central du système SoC-FPGA.

Le système de base dans lequel on cherche à introduire des interférences est composé d'un accélérateur de multiplication matricielle à virgule flottante 32 bits sur des matrices carrées de grande taille (50 par 50). Afin d'exploiter l'intégralité de la bande-passante du bus, l'accélérateur est exploité par une première partition de l'hyperviseur par l'intermédiaire d'un module DMA également implémenté en matériel dans le FPGA.

La contention est ensuite générée par une seconde partition, connectée à un second module DMA, qui va envoyer un grand nombre de requêtes à la fois de lecture et d'écriture vers la mémoire partagée du système. Les transactions sont configurées avec une grande taille de rafale AXI (AXI Burst Size), ce qui a été noté comme un facteur important de contention par [9]. Cette partition traite ici de données factices générées aléatoirement, mais peut dans un système réel représenter une partition traitant des entrées-sorties avec une bande-passante élevée.

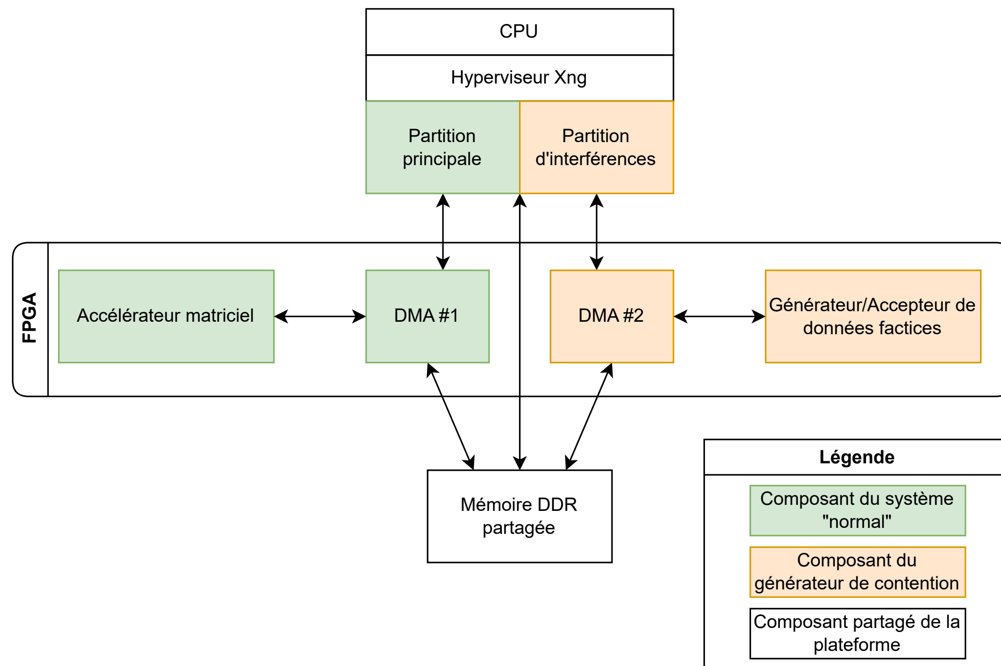

FIGURE 3.5 Diagramme simplifié de l'architecture du système de test de contention complet

Le système d'extraction de points de temps offert par notre outil est ensuite utilisé pour bâtir une architecture plus complète d'analyse de performance, similaire à celle étudiée en 3.3.3. Ainsi, nous pouvons extraire des statistiques sur les temps d'exécution de certaines sections du code HLS de notre accélérateur matériel.

Dans notre cas, nous regardons les temps d'exécution de quatre sections de code de l'accélérateur : 1) l'attente de l'envoi d'un calcul à effectuer par le processeur, 2) la réception des données, 3) le calcul de la multiplication matricielle, et 4) le transfert du résultat en mémoire partagée. Ces 4 sections sont respectivement nommées ”Attente”, ”Reception”, ”Calcul” et ”Envoi” dans les tableaux 3.3 et 3.4.

À l'exécution, un test de performance effectuant de nombreuses opérations sur l'accélérateur est lancé, d'abord en ayant désactivé la partition d'interférences, puis il est relancé en ayant activé l'envoi de trafic d'interférence sur la seconde partition. Les résultats extraits par notre outil sont ensuite comparés.

Analyse des résultats

Section	**Minimum**	**Maximum**	**Moyenne**
Attente	537	245 855 079	25 230
Reception	5 973	6 190	6 190
Calcul	62 838	62 838	62 838
Envoi	2 512	2 512	2 512

Tableau 3.3 Indices de performance de multiplication matricielle en matériel sans interférences d'un DMA secondaire (valeurs en cycles)

Section	**Minimum**	**Maximum**	**Average**
Attente	554 (+3%)	1 040 944 529 (+323%)	108 310 (+329%)
Reception	10 281 (+72%)	12 267 (+98%)	11 983 (+97%)
Calcul	62 838	62 838	62 838
Envoi	2 512	2 512	2 512

Tableau 3.4 Indices de performance de multiplication matricielle en matériel *avec* interférences d'un DMA secondaire (valeurs en cycles) et comparaison aux valeurs sans interférences dans le tableau 3.3

Les résultats de l'analyse en implémentation sont regroupés dans les tableaux 3.3 et 3.4.

On peut rapidement depuis ces résultats la source de l'augmentation du temps de réponse de l'accélérateur au niveau du processeur. Le temps de calcul n'est pas en cause, étant donné qu'il reste constant peu importe les données (cela pourrait être le cas pour des algorithmes plus complexes).

Cependant, le temps de réception des données augmente significativement, doublant presque lorsque l'on génère de la contention sur le bus. On constate également que le temps d'attente de calcul augmente également, ce qui suggère que les performances du processeur lui-même sont également affectées, et qu'il envoie ses requêtes moins fréquemment en raison d'une saturation de la mémoire centrale.

Le temps d'envoi du résultat cependant n'augmente pas. Deux facteurs peuvent expliquer cela. D'abord, l'envoi en retour a un volume de données deux fois moins important (une matrice seulement contre deux en réception), ce qui explique qu'il ne soit pas autant affecté par la contention. Par ailleurs, il est ici probable que l'envoi, géré par un module DMA, ai été

complété en temps constant du point de vue de l'accélérateur car les données ont simplement été copiées dans un tampon d'écriture du DMA.

En tout cas, les valeurs de performances extraites nous on permis un diagnostic rapide du problème de performances. On peut en conclure comme solution simple qu'il faudrait augmenter la capacité du lien de communication dans la direction processeur vers FPGA. Le lien en sens inverse n'est manifestement pas affecté, et il serait inutile d'y ajouter de la capacité.

La transparence dans l'exécution sur le FPGA permise par notre outil a été essentielle pour permettre d'exclure par exemple une augmentation du temps de calcul sur l'accélérateur, qui aurait été pris en bloc par une analyse depuis le logiciel. Notre instrumentation aurait également pu permettre de découper notre analyse en phases du calcul lui-même, afin de déterminer un possible goulot d'étranglement.

À noter que le temps d'attente semble atteindre un maximum très élevé, mais cette valeur est liée au fait que lors du démarrage de la carte et de l'hyperviseur, l'accélérateur est resté en attente, puisque le processeur n'a pas soumis de calcul. Cette valeur extrême a donc du sens, mais pourrait être évitée dans le futur par un mécanisme réinitialisant cette valeur au début effectif de l'application, car elle biaise fortement la moyenne du temps d'attente.

3.3.5 Application à l'exploration architecturale et à la validation de la simulation

Montage expérimental

Nous effectuons un fragment d'une exploration architecturale, afin de déterminer si un gain de performance pourrait être obtenu en implémentant un accélérateur en matériel plutôt qu'en logiciel, ou si les surcoûts liés à la déportation du module en matériel domine les possible gains.

Ainsi, nous considérons un accélérateur simple de multiplication matricielle à virgule flottante de 50 par 50, dénué de toute optimisation en l'état, que nous instrumentons en utilisant notre outil, avec une méthode similaire à celle présentée dans la section 3.3.4. L'accélérateur est ensuite déployé à la fois en matériel et en logiciel, et les mesures de performances sont comparées.

Afin d'effectuer en simultané une validation de la précision de notre simulateur, nous déployons ensuite l'architecture résultante à la fois sur carte et sur une simulation. À noter que notre modèle n'est annoté d'aucune information sur le timing.

Résultats et analyse

	Simulation		Implémentation	
Section	Logiciel	Matériel	Logiciel	Matériel
Reception	4.929	0.261 (-1789%)	1.764	6.217 (+346%)
Calcul	2.470	2.319 (-6.5%)	1.006	0.628 (-60%)
Envoi	3.899	0.129 (-2922%)	0.628	3.416 (+444%)
Total	11.298	2.709 (-76%)	3.398	10.261 (+201.97%)

Tableau 3.5 Temps d'exécution moyen (en millisecondes) de chaque section du code de l'accélérateur matriciel selon s'il s'exécute en matériel et logiciel, sur l'implémentation et en simulation logicielle

En implémentation, au global, donc en comptant les temps de communication, le temps d'exécution de l'accélérateur est trois fois plus lent qu'en logiciel. Or, grâce à notre instrumentation, on constate que le calcul en lui-même est bien plus rapide sur le FPGA, d'environ 60%, mais que cette accélération ne permet pas de dépasser le temps de communication, qui lui est multiplié par un facteur de plus de quatre dans les deux directions.

On voit ici l'utilité de l'instrumentation dans le FPGA même grâce à notre outil, car des outils fournissant un grain moins fin d'analyse, comme des outils logiciels qui n'auraient obtenu que le total, nous auraient amené à la conclusion qu'il est absolument inutile de déporter notre calcul en matériel. À l'opposé, notre outil montre que le calcul est bien accéléré, mais que la communication est un goulot d'étranglement, et on pourrait donc les optimiser pour possiblement obtenir un résultat plus satisfaisant.

Au niveau de la simulation, notre outil permet d'immédiatement remarquer la faible précision de ce premier raffinement, car les temps d'exécution en matériel simulé n'ont rien à voir avec ceux en implémentation, et on ne trouve même pas un facteur consistant d'accélération entre logiciel et matériel sur la simulation et l'implémentation.

Ce premier test pourrait cependant permettre d'effectuer de la « back-annotation », et d'ajouter des délais estimés dans la simulation, selon ce que l'on observe en implémentation. Cela montre que notre outil peut aider à évaluer la qualité de notre simulation, et même contribuer à l'améliorer.

3.4 Conclusion du chapitre

Nous avons présenté dans ce chapitre une architecture pour un outil d'instrumentation de mesure de performance universelle proposant une abstraction commune en logiciel et matériel de manière transparente. L'utilité et l'efficacité en ressources de l'outil a été démontrée sur des exemples d'applications, notamment dans la détection de problématiques d'interférences de partitionnement de systèmes matériel-logiciel partitionnés avec un hyperviseur.

Cependant, beaucoup d'aspects de l'usage de l'outil n'ont pas été analysés par manque de temps ou parce qu'ils dépassaient les objectifs du présent mémoire. Ces aspects incluent notamment une exploration plus approfondie de l'impact de l'instrumentation sur le chemin critique en matériel, et la possible réduction de la fréquence maximale d'un système instrumenté, ce qui rendrait notre outil trop intrusif et contreviendrait à nos objectifs initiaux. Également, la gestion de domaines matériels multiples n'a été implémentée que partiellement. Enfin, une instrumentation de très grands systèmes pourrait soulever des problèmes liés à des biais d'horloge, qui n'ont pas été étudiés ici.

Par ailleurs, le lecteur a pu se rendre compte que dès lors qu'un hyperviseur était impliqué, nous nous sommes concentrés sur des expérimentations en implémentation, ignorant les lancements en simulation, pourtant supportée comme cible par nos outils de mesure de performances. Cela est dû à un manque de support des simulateurs de jeu d'instructions pour les hyperviseurs de partitionnements embarqués.

Cette impossibilité de simuler le système instrumenté avec hyperviseur réduit un des principaux intérêts de la méthodologie ESL, qui exploite la simulation pour valider divers critères du système sans avoir recours à l'implémentation. En effet, le fait de pouvoir travailler en simulation permet dans un premier temps d'éviter le recours à la synthèse matérielle d'un système complexe pouvant demander plus d'une dizaine d'heures, et également des ressources financières importantes pour l'acquisition de de cartes de prototype souvent extrêmement dispendieux pour des domaines spécialisés comme celui de l'espace.

Nous proposons donc dans le chapitre suivant une amélioration d'un simulateur de jeu d'instructions existant pour lui ajouter un support d'un hyperviseur de partitionnement. Ceci permettra ainsi par ailleurs d'extraire plus aisément les résultats de performances grâce à nos outils directement en simulation, à condition bien sûr que celle-ci atteigne une précision suffisante pour que les résultats soient pertinents à l'analyse.

CHAPITRE 4 SECOND THÈME - PLATEFORME VIRTUELLE POUR HYPERVISEUR

Nous nous intéressons dans ce chapitre à fournir un modèle de simulation (ou plateforme virtuelle) des systèmes avec hyperviseurs. Nous verrons que cela requiert des modifications aux modèles existants par rapport aux systèmes sans hyperviseurs, puisque les modèles performants actuels ne possèdent de support pour la simulation des mécanismes de virtualisation (sur lesquels reposent les hyperviseurs), peu utile dans les usages généraux.

Le support de cette simulation se fera en deux temps, sur deux plateformes différentes. D'abord, nous allons intégrer la simulation du matériel nécessaire à un hyperviseur sur une plateforme mature, ce qui permettra de se familiariser avec les mécanismes et l'architecture du simulateur choisi. Dans un second temps, nous travaillerons sur la simulation d'une plateforme bien plus moderne, mais dont le support en simulation est pour l'instant quasi inexistant.

4.1 Modèle de co-simulation avec QEMU & SystemC

Nous nous proposons d'intégrer les plateformes à un modèle de simulation particulier. Celui-ci intègre la simulation logicielle performante avec QEMU et la simulation de la partie matérielle en exécutant le modèle algorithmique annoté grâce à la librairie SystemC..

4.1.1 Le simulateur QEMU

Nous nous proposons d'utiliser le simulateur QEMU [22]. Ce choix est basé sur ses excellentes performances de simulation, rendues possibles grâce à la technique de traduction binaire dynamique qu'il utilise, ainsi que son support déjà existant pour la co-simulation matériel-logiciel contribué par Xilinx.

Ce simulateur, à l'origine uniquement un simulateur de jeu d'instructions, se double également d'une simulation de système complet, et fournit donc un modèle des périphériques variés des plateformes en plus du CPU seul, comme les minuteries, les périphériques d'entrées-sorties et cartes graphiques.

QEMU dispose également, grâce à la contribution de Xilinx, d'un support pour la simulation de la partie FPGA d'un système SoC-FPGA. Celui-ci permet donc la connexion d'un simulateur SystemC qui simule la partie matérielle du système à haut niveau à QEMU qui simule le reste du système. Les communications entre le logiciel et le matériel ainsi simulées sont modélisées à haut niveau, précisément au niveau de la transaction sur le bus central du

système (ou TLM).

4.2 Amélioration d'un modèle de simulation existant

4.2.1 Plateforme Cortex-A9

La plateforme matérielle de base que nous voulons simuler est la plateforme Zynq-7000 de Xilinx. Celle-ci présente deux processeurs ARM Cortex-A9, associés à un FPGA de Xilinx. Les processeurs utilisent le jeu d'instructions 32-bit ARMv7-A, aujourd'hui très mature au vu de son âge, celui-ci ayant été introduit en 2007.

4.2.2 Fonctionnalités présentes et manquantes

Au vu de la maturité de la plateforme, la plateforme Zynq-7000 est d'ores-et-déjà très bien supportée en simulation. Ainsi, la simulation est complète pour les fonctionnalités centrales telles que le jeu d'instruction, l'unité de gestion de la mémoire (ou MMU), les périphériques d'entrées-sorties de base, etc....

Cependant, le développement de cette simulation s'était concentré sur le support du système d'exploitation Linux, ignorant les fonctionnalités plus marginales que celui-ci n'utilise pas.

Registre de contrôle de virtualisation

L'hyperviseur que nous ciblons pour cette simulation, Xng, utilise certaines de ces fonctionnalités. La première est le support des extensions de virtualisation du Cortex-A9. Ces extensions sont très basiques, et sont un précurseur des extensions de virtualisation ajoutées plus tard à l'architecture ARMv7-A, qui ne sont pas supportées sur le Cortex-A9, plus ancien.

Alors que les extensions de virtualisation de ARMv7-A fournissent un niveau de privilège spécifiquement dédié à l'hyperviseur, la version réduite du Cortex-A9 ne se résume qu'à un mécanisme d'aide à la virtualisation des interruptions via un registre permettant d'ignorer le masquage d'interruptions, le registre VCR (pour Virtualization Control Register).

Plus précisément, ce registre permet d'ignorer les drapeaux de masquage d'interruptions indiqués normalement dans le registre de contrôle du système lorsque le système est dans un mode de bas privilèges. Ainsi, l'hyperviseur peut intercepter les interruptions même si un système d'exploitation invité les a masquées, et ensuite en faire un traitement correct (par exemple en les redirigeant vers un invité ne les ayant pas masquées).

Par conséquent, les effets de ce registre ont été assez simplement simulés, par une vérification

des cas où le masque d'interruptions est ignoré dans le code de routage des interruptions de QEMU.

Autres détails

Étant donné que le registre VCR est plus ou moins la plus grande dépendance de Xng distincte du matériel de base, peu d'autres éléments ont dû être modifiés sur QEMU pour intégrer cet hyperviseur, et relèvent du détail. Cependant, pour être exhaustif, elles sont listées ici :

— Modification du mode d'exécution et du niveau de privilèges au démarrage du système simulé, qui ne correspondait pas à la spécification ARM.
— Réparation du décodage de l'instruction SMC (Secure Monitor Call, appel système au mode superviseur sécurisé), qui attendait des bits à zéro à des positions qui devraient être ignorées.
— Connexion des lignes d'interruptions rapides « FIQ », non utilisées par Linux, mais fortement exploitées par Xng.

4.2.3 Résultats expérimentaux : validation de la simulation

Tests fonctionnels unitaires

La seule nouvelle fonctionnalité du support Xng sur cette plateforme étant le registre VCR, c'est sur celui-ci que se sont concentrés les tests de simulation unitaires. On rappelle que ce registre permet d'ignorer le masquage d'une interruption d'un certain type, normalement spécifié par le registre de statut du système.

Le code de test réalise, selon plusieurs assignations du registre VCR, un test consistant à lever une interruption du type démasqué dans le VCR en utilisant un timer système, et vérifier l'arrivée de celle-ci malgré le masquage dans le registre système lorsque le processeur est en mode non-privilégié (et aussi que le traitement de l'interruption n'est pas affecté en mode privilégié).

Tests fonctionnels globaux

La validation de la simulation du système avec hyperviseur plus globale s'est effectuée simplement par l'exécution de programmes développés pour Xng sur le simulateur, et en vérifiant leur comportement et leurs sorties.

Ces tests ont démontré la cohérence de la simulation même dans les cas les plus complexes impliquant plusieurs partitions lancées sur les deux cœurs du A9 simulé simultanément qui

communiquent entre elles.

Si la simulation semble donc correcte sur le plan fonctionnel, nous remarquons sur cette plateforme une certaine imprécision des minuteries du système simulé, parfois lents, parfois rapides, ce qui cause des irrégularités d'ordonnancement et limite l'utilisation de la simulation pour la validation des aspects temporels. Ce problème est ardu à résoudre, en raison de l'architecture même de QEMU, et sera pour l'instant laissé comme piste d'amélioration future.

4.3 Création d'un modèle pour une plateforme moderne

La plateforme Zynq de la section précédente est très mature, mais à cause de son âge, ne sera pas nécessairement à l'abri de devenir obsolète dans les prochaines années. Nous proposons donc d'implémenter le support en simulation d'hyperviseurs de partitionnement pour une plateforme embarquée plus moderne.

4.3.1 Plateforme Ng-Ultra

La plateforme moderne que nous nous proposons de simuler est le Ng-Ultra. Celle-ci est une nouvelle plateforme dont le développement est mené par l'Agence Spatiale Européenne. Un diagramme simplifié de son architecture est présenté dans la figure 4.1.

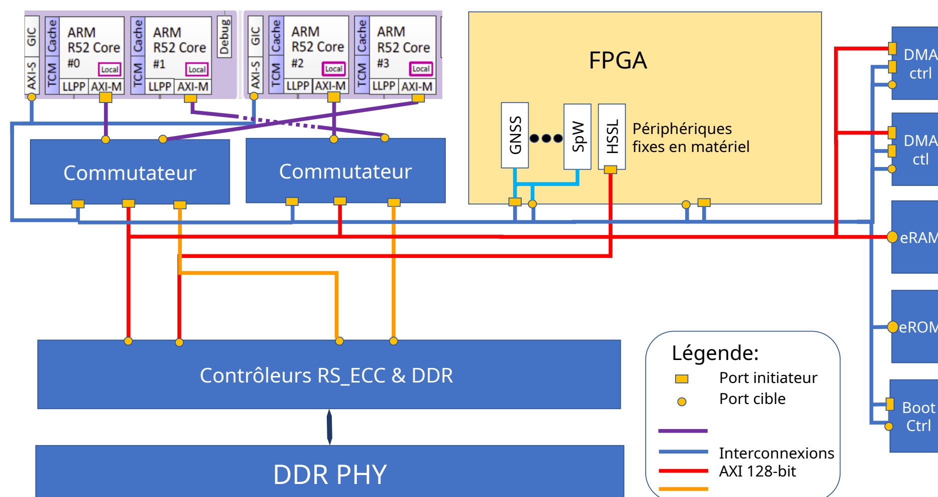

Figure 4.1 Diagramme simplifié de l'architecture de la plateforme Ng-Ultra

Cette plateforme, destinée à l'espace, intègre quatre cœurs Cortex-R52 sur une plateforme résistante aux radiations. Ces processeurs implémentent l'architecture ARMv8-R, variante optimisée pour le temps-réel et les tâches critiques de l'architecture ARM généraliste ARMv8-

A. Ainsi, au lieu d'une unité de gestion de la mémoire (ou MMU), ceux-ci présentent une unité de protection de la mémoire (ou MPU), bien plus simpliste. En effet, une MMU effectue une traduction de chaque adresse mémoire accédée selon une table de traduction établie par le système d'exploitation, et peut ainsi créer des espaces d'adressage dits « virtuels » entièrement distincts de l'agencement de la mémoire physique. La MPU, quant à elle, ne retient que les aspects de protection de la mémoire. Elle va donc uniquement rechercher, pour une adresse mémoire accédée, les permissions de lecture/écriture/exécution, une opération bien plus simple en termes de complexité de calcul, et surtout plus prévisible au niveau des délais d'accès, une propriété très intéressante pour les systèmes temps-réel.

En sus des quatre cœurs R52, la plateforme est renforcée contre les radiations et intègre un FPGA performant, le tout communicant par un réseau-sur-puce comprenant plusieurs interconnexions entre les processeurs, le FPGA et les périphériques. On a donc bien ici un système SoC-FPGA comme vu en introduction.

De plus, le prix de liste de cette nouvelle carte pouvant être embarquée dans des satellites en orbite stationnaire (GEO) à plus de 36,000 km de la terre, est d'environ 20,000 euro. Ce qui justifie l'importance d'une plate-forme virtuelle du Ng-Ultra et d'une simulation haut niveau lorsqu'un industriel souhaite faire un premier prototype sans avoir à faire immédiatement un tel déboursé. De telles simulations peuvent donc être utiles pour valider une fonctionnalité, développer des pilotes (drivers) ou encore explorer des premiers choix de partitionnements logiciel/matériel.

4.3.2 Fonctionnalités nécessaires

Étant donné l'état précoce de la plateforme Ng-Ultra, aucun hyperviseur de partitionnement n'y a été porté au moment de l'écriture de ce mémoire. Nous avons donc établi un ensemble de fonctionnalités de l'architecture et de composants matériels que nous jugeons nécessaires au support d'un hyperviseur, en s'inspirant notamment de ceux utilisés sur la plateforme Zynq vue précédemment. Ensuite, nous implémentons leur simulation dans QEMU si elles n'y sont pas déjà présentes, pour espérer avoir une simulation supportant déjà l'hyperviseur une fois celui-ci porté sur la plateforme.

Les fonctionnalités retenues sont les suivantes :

— Le jeu d'instruction de base du processeur, essentiel à la simulation de tout code logiciel.
— L'unité de protection de la mémoire (ou MPU), utilisée pour assurer le partitionnement spatial entre les zones mémoires des différentes partitions.
— L'unité de gestion des horloges et de la réinitialisation (ou « Clock and Reset Ma-

nager »), utilisée sur la plateforme pour effectuer le réveil des cœurs secondaires du processeur.
— Le module émetteur-récepteur universel asynchrone (ou UART), qui permet d'obtenir des entrées-sorties basiques sous la forme d'un terminal sériel.
— Les minuteries privées des cœurs du processeur, qui génèrent les interruptions régulières qui cadencent l'ordonnancement de l'hyperviseur.

D'abord, au niveau le plus fondamental qu'est le jeu d'instruction AArch32 implémenté par le Cortex-R52, il est identique à celui des anciennes versions de l'architecture ARM, donc celui-ci est largement supporté sur le simulateur QEMU. Il en est de même avec le support matériel de la virtualisation, qui reste identique au profil généraliste de l'architecture, hormis en ce qui concerne la MPU.

La MPU de l'architecture ARMv8-R, suivant la spécification architecturale PMSAv8-32, a donc été implémentée dans QEMU. Celle-ci consiste en un nombre fini de zones de mémoires, définies par leurs adresses de début et de fin, auxquelles sont associées des permissions de lecture, écriture et d'exécution. La virtualisation ajoute un deuxième niveau de permissions, définies par l'hyperviseur, et qui sont ensuite combinées de façon la plus restrictive avec le premier niveau.

Périphériques de la plateforme

L'étape finale afin de supporter une simulation d'un hyperviseur sur la plateforme Ng-Ultra est la modélisation des périphériques externes au processeur seul cités en 4.3.2. Or, sur le Ng-Ultra, n'utilise quasiment que des composants développés en interne, et aucun composant « off-the-shelf ».

Ainsi, nous avons développé une simulation de base de ces nouveaux périphériques dans QEMU que nous avons jugé essentiels à un hyperviseur.

La simulation de la fonctionnalité est complète pour le module UART, incluant la génération d'interruption lors de la réception et de l'envoi de données.

Le module de gestion des horloges et de la réinitialisation, plus complexe, et incluant des fonctionnalités de plus bas niveau que nous ne souhaitons pas simuler comme la gestion du voltage des processeurs, a vu sa simulation limitée à la fonctionnalité permettant d'exécuter la procédure de de réveil des processeurs secondaires, dans l'optique de supporter la simulation d'un hyperviseur multi-cœurs.

4.3.3 Résultats expérimentaux : validation fonctionnelle de la simulation

La plateforme Ng-Ultra est très récente. En plus d'être dispendieuses, les premières cartes de prototypage physiques sont à peine sortis de production au moment de l'écriture de ce mémoire. Étant donné la limitation de la quantité de cartes disponibles, elles sont pour le moment réservées aux 2 grands maîtres d'œuvre (Airbus Defense and Space et Thales Alenia Space) ayant participé à sa création. Ainsi, le support logiciel est encore très limité. Par exemple, l'hyperviseur ciblé, Xng, est toujours en cours de portage pour le Ng-Ultra, et n'est pas disponible pour effectuer les tests fonctionnels nécessaires à la validation de la simulation.

Ainsi, nous avons dû nous reporter à d'autres cibles pour valider la simulation par proxy. Nous avons d'abord utilisé le système d'exploitation temps-réel (ou RTOS) Zephyr. Celui-ci n'est pas porté vers la cible Ng-Ultra en tant que tel, mais il existe un port vers une plateforme virtuelle de référence de l'architecture ARMv8-R. Par conséquent, l'exécution avec succès de ce système temps-réel nous assure un certain niveau de fonctionnalité de simulation en ce qui relève du jeu d'instruction, des mécanismes architecturaux de base tels que les niveaux de privilèges, mais surtout du support de la MPU.

Le RTOS Zephyr, cependant, n'exploite pas les fonctionnalités de virtualisation (il laisse immédiatement la main du mode hyperviseur au mode superviseur), ni les périphériques spécifiques du Ng-Ultra ou le FPGA. Ainsi, pour valider ces points, nous avons écrit un noyau de tests en baremetal. Celui-ci initialise le système, et vérifie le fonctionnement des divers sous-systèmes, comme la MPU, l'UART, ou encore tente des transactions de base avec le FPGA.

Avec l'ensemble des tests effectués, nous avons pu établir un seuil de fonctionnalité basique de la simulation. Il faut tout de même noter que ces tests ne sont pas exhaustifs, et que la simulation reste encore à affiner, notamment au niveau des délais.

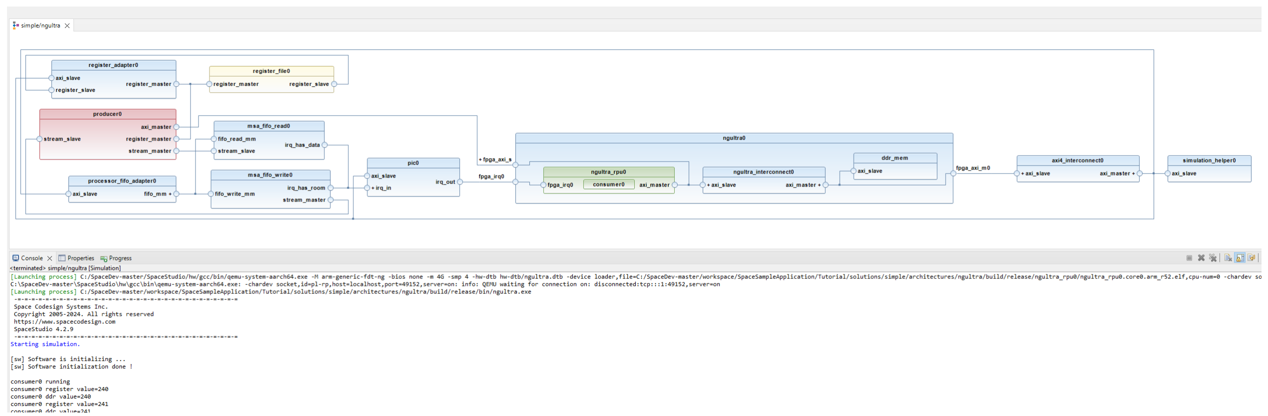

FIGURE 4.2 Diagramme d'architecture et exécution en simulation du RTOS RTEMS sur le Ng-Ultra dans la plateforme SpaceStudio

Ainsi, grâce à la simulation présentée dans ce chapitre, une plateforme virtuelle pour le Ng-Ultra a pu, en collaboration avec SpaceCodesign Systems Inc, être intégrée dans SpaceStudio. On peut ainsi observer sur la figure 4.2 une architecture exploitant cette plateforme, et un exécution en simulation d'un système d'exploitation temps réel sur cette architecture. Cependant, la conception de systèmes qui eux reposent sur un hyperviseur, et non un simple RTOS, n'est pas encore intégrée à la plateforme SpaceStudio et demeure pour l'instant parmi les travaux futurs à explorer.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Les travaux présentés ont atteint leurs objectifs de facilitation du développement d'applications embarquées sur système hétérogène sur les deux fronts proposés.

D'abord, un outil d'instrumentation de métriques de performance ont été développés. Ceux-ci permettent dans leur version ici précoce à un utilisateur de récupérer et manipuler des points de temps avec une API unifiée que ce soit en logiciel ou en matériel, et sur plusieurs cibles de systèmes d'exploitation. Cela permet alors une bien meilleure visibilité de l'exécution du programme sur le FPGA, qui est atteinte bien plus facilement qu'avec les outils existants. Par ailleurs, nous avons montré que l'universalité de l'outil facilite entre autre l'exploration architecturale et la validation de nos modèles de simulation.

Ensuite, nous avons obtenu, par amélioration de QEMU, une simulation efficace des systèmes hétérogènes matériel-logiciel tournant avec un hyperviseur de partitionnement. Un hyperviseur commercial a été démontré comme fonctionnel en simulation d'une plateforme existante mature et un modèle de simulation a été créé pour une plateforme plus moderne, mais au support encore limité.

Le faible impact des outils sur la consommation de ressources matériel et sur les performances des applications a été démontré sur plusieurs exemples simples. Ainsi, le compteur ajouté en matériel ne consomme que très peu des ressources matérielles, de même que leur lecture depuis un module matériel, grâce à un protocole de communication simple. Le seul aspect pouvant consommer des ressources matérielles est la manipulation des points de temps, mais celle-ci reste assez légère en ressources si correctement implémentée par l'utilisateur.

5.2 Limitations de la solution proposée

Les outils de mesure de performance développés sont, comme mentionnés précédemment, encore précoces. En effet, la récupération de points de temps est une fonctionnalité cruciale et donne une grande liberté dans leur manipulation par l'utilisateur, mais la charge de leur manipulation, par exemple pour mesurer des délais, reste dans les mains de l'utilisateur, ce qui peut complexifier leu utilisation.

Par ailleurs, un problème reste également à la conjugaison des deux enjeux, c'est-à-dire au niveau de la mesure des métriques de performances en simulation, qui, bien qu'elle soit permise

par nos outils, est au finale très peu représentative de la réalité. La cause est le compromis effectué en faveur des performances du simulateur. Ainsi, pour obtenir des métriques précises, il faut aller à l'implémentation, ce qui implique de la synthèse de matériel, qui peut durer très longtemps pour des projets complexes, et donc allonger le temps de développement.

5.3 Améliorations futures

Il existe des méthodes afin de régler le problème de précision en termes de temps de notre simulation. On pourrait par exemple exploiter des données de temps d'exécution extraites d'un profilage sur le matériel préalable, et les appliquer aux opérations dans la simulation, ou encore utiliser un système hybride combinant un simulateur performant comme QEMU pour les sections du programme les plus exigeantes en calcul, puis transférer l'état simulé à un simulateur de jeu d'instruction précis au cycle près, bien plus lent, mais convenable pour des sections de calcul peu intensif (qui seraient déléguées au matériel).

Du côté des outils de mesure de performances, il serait intéressant de développer, sur la base créée dans ces travaux, des abstractions supplémentaires. Celles-ci pourraient prendre la forme de « moniteurs de temps », qui mesureraient des métriques de durée (minimum, maximum, moyenne, etc…) sur des segments de code fixés par l'utilisateur. Ainsi, la charge de la manipulation des points de temps serait délestée du développeur, et faciliterait son travail. Qui plus est, cette manipulation, si effectuée par l'outil SpaceStudio directement, pourrait être optimisée par une implémentation directement en RTL, et donc être plus légère qu'une implémentée par l'utilisateur comme vu en 3.3.3.

Tel que mentionné à la section 3.4, le travail du présent mémoire a servi à jeter les bases d'une méthodologie d'instrumentation. Beaucoup de travail, déjà en cours de réalisation par d'autres étudiants gradués, reste à compléter. Parmi ces spécialisations on pense à l'étude de l'intrusivité au niveau matériel des instruments insérés, ou sur des possibles biais d'horloges ou des décalages entre domaines matériels imprévisibles. La mise en place de techniques d'agrégation de l'information afin d'éviter la contention sur les interconnexions est aussi prévue.

Enfin, il serait intéressant d'explorer les possibilités d'exploitation de ces outils afin d'effectuer un raffinement automatique d'architectures via les mesures recueillies. Les délais mesurés pourraient par exemple être des facteurs guidant des algorithmes de partitionnement entre matériel et logiciel.

RÉFÉRENCES

[1] M. Desnoyers et M. Dagenais, “Low Disturbance Embedded System Tracing with Linux Trace Toolkit Next Generation.” Embedded Linux Conference 2006.

[2] C. B. Watkins et R. Walter, “Transitioning from federated avionics architectures to integrated modular avionics,” dans *2007 IEEE/AIAA 26TH DIGITAL AVIONICS SYSTEMS CONFERENCE, VOLS 1-3*, ser. Digital Avionics Systems Conference. IEEE ; AIAA, 2007, p. 241–250, 26th Digital Avionics Systems Conference, Dallas, TX, OCT 21-25, 2007.

[3] P. Bieber *et al.*, “New Challenges for Future Avionic Architectures.” *Aerospace Lab*, n°. 4, p. p. 1–10, mai 2012. [En ligne]. Disponible : https://hal.science/hal-01184101

[4] G. De Michell et R. Gupta, “Hardware/software co-design,” *Proceedings of the IEEE*, vol. 85, n°. 3, p. 349–365, 1997.

[5] S. Lahti *et al.*, “Are we there yet ? a study on the state of high-level synthesis,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, n°. 5, p. 898–911, 2019.

[6] G. B. Adrien Thierry, Hubert Guérard, “Dynamic virtual platform for HW/SW partitioning on MPSoC platforms.” Embedded Real Time Systems (ERTS).

[7] R. P. Goldberg, “Architectural Principles for Virtual Computer Systems.”

[8] ARINC, “ARINC 653 Avionics Application Software Standard Interface,” Aeronautical Radio, Incorporated, Rapport technique.

[9] F. Restuccia *et al.*, “Is Your Bus Arbiter Really Fair ? Restoring Fairness in AXI Interconnects for FPGA SoCs,” vol. 18, n°. 5s, oct 2019. [En ligne]. Disponible : https://doi.org/10.1145/3358183

[10] *AMBA AXI Protocol Specification*, ARM.

[11] F. Restuccia *et al.*, “Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs,” dans *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, 2020, p. 12 :1–12 :23.

[12] ——, “Bounding Memory Access Times in Multi-Accelerator Architectures on FPGA SoCs,” *IEEE Transactions on Computers*, vol. 72, n°. 1, p. 154–167, 2023.

[13] ——, “AXI HyperConnect : A Predictable, Hypervisor-level Interconnect for Hardware Accelerators in FPGA SoC,” dans *PROCEEDINGS OF THE 2020 57TH ACM/E-*

DAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC), ser. Design Automation Conference DAC, 2020.

[14] J. Cong *et al.*, "FPGA HLS Today : Successes, Challenges, and Opportunities," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, nº. 4, aug 2022.

[15] Xilinx ILA IP Product Page (Retrieved 2023-05-03). [En ligne]. Disponible : https://web.archive.org/web/20230503191303/https://www.xilinx.com/products/intellectual-property/ila.html

[16] *Intel® Quartus® Prime Pro Edition User Guide : Debug Tools Version 22.4*.

[17] I. Parnassos *et al.*, "SoCLog : A real-time, automatically generated logging and profiling mechanism for FPGA-based Systems On Chip," dans *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, p. 1–4.

[18] Y.-K. Choi et J. Cong, "HLScope : High-Level Performance Debugging for FPGA Designs," dans *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, p. 125–128.

[19] H. Bensalem, Y. Blaquière et Y. Savaria, "In-FPGA Instrumentation Framework for OpenCL-Based Designs," *IEEE Access*, vol. 8, p. 212 979–212 994, 2020.

[20] L. Suriano *et al.*, " Unified Hardware/Software Monitoring Method for Reconfigurable Computing Architectures using PAPI," dans *Proceedings of the 2018 13th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*, 2018.

[21] M. Masmano *et al.*, "Xtratum : a hypervisor for safety critical embedded systems." 09 2009.

[22] F. Bellard, "QEMU, a fast and portable dynamic translator," dans *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA : USENIX Association, avr. 2005. [En ligne]. Disponible : https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator