DELAGE Florian

Année 2024–2025
Promotion 31

Rapport de stage – Loria
Période : du 05 mai au 13 juin 2025

Optimisation d'un algorithme de calcul de bornes de latences pour accélérer la génération de preuves

Encadrants :
AMET Matthieu
THOMAS Ludovic

Coordonnées de l'entreprise :
Loria, 54506 Vandœuvre-lès-Nancy Cedex
Tél : +33 3 83 59 20 00

Remerciements

Je tiens à remercier chaleureusement l'ensemble des personnes qui ont contribué au bon déroulement de mon stage au sein du Loria. Je remercie tout particulièrement monsieur Matthieu AMET, mon maître de stage, ainsi que monsieur Ludovic THOMAS pour leur accueil, leur disponibilité, leur conseil et l'accompagnement qu'ils m'ont apporté tout au long de cette expérience. Leur encadrement m'a permis de progresser aussi bien sur le plan professionnel que personnel. Je souhaite également exprimer ma gratitude à toute l'équipe SIMBIOT, pour leur bienveillance, leur écoute et leur esprit collaboratif, qui ont largement facilité mon intégration. Enfin, je remercie monsieur Théo DOCQUIER pour son suivi et son soutien dans la réalisation de ce stage.

Résumé

Au cours de mon stage au sein de l'équipe SIMBIOT du laboratoire Loria (Université de Lorraine), j'ai travaillé sur l'optimisation d'un algorithme de calcul de bornes de latence (Total Flow Analysis – TFA) dans le contexte des réseaux temps-réels. L'objectif était d'accélérer la génération de preuves à divulgation nulle de connaissance (Zero-Knowledge Proofs) à l'aide de l'outil Risc Zero, sans compromettre l'exactitude des résultats.

Mes contributions principales ont été les suivantes :

— Analyse des performances de l'algorithme TFA existant, en identifiant les goulots d'étranglement dans l'algorithme ;
— Optimisation du code en Rust, pour maximiser l'efficacité des calculs ;
— Exploration et intégration de techniques de réduction des coûts cryptographiques, afin d'accélérer la génération des ZKP (Zero-Knowledge Proofs) tout en respectant les contraintes liées à la génération de preuves de Risc Zero.

J'ai pu également aider à l'implémentation d'une extension moins pessimiste de TFA (TFA++) Ce projet s'inscrit dans une démarche d'innovation visant à rendre les applications critiques utilisant des réseaux temps-réels viables à grande échelle.

Table des matières

Chapitre 1

Introduction

À mesure que la technologie progresse, de plus en plus de systèmes reposent sur des échanges d'informations rapides, fiables et prévisibles. C'est notamment le cas des voitures autonomes, des satellites, ou encore des chaînes de production automatisées. Dans ces environnements, il est essentiel que les données circulent dans des délais garantis. Un simple retard de quelques millisecondes peut entraîner des dysfonctionnements majeurs, voire des situations dangereuses. C'est pour répondre à ces besoins qu'ont été développés les réseaux temps-réels qui visent à offrir des garanties strictes sur la latence maximale des communications.

Contrairement aux réseaux classiques de type Internet, les réseaux temps-réels doivent fournir des garanties de latence dans le pire cas. Pour cela, des algorithmes comme Total Flow Analysis (TFA) ont été conçus afin d'en calculr des bornes de latence. Ces dernières seront utilisées afin de valider le respect des contraintes.

Cependant dans des environnements collaboratifs, une nouvelle problématique se pose : comment prouver à l'un que les contraintes de latence sont respectées par l'autre, sans pour autant lui révéler sa configuration interne ? C'est ici qu'intervient une approche innovante issue de la cryptographie : les Zero-Knowledge Proofs (ZKP). Ces preuves permettent de démontrer qu'un calcul est correct, sans pour autant exposer les données utilisées pour l'effectuer.

L'objectif principal de ce stage est d'optimiser un algorithme de calcul de latence afin d'accélérer la génération de ZKP. Ceci reste aujourd'hui très coûteux en temps et en ressources. Il s'agit donc d'identifier les points les plus lents de l'algorithme, et d'explorer des techniques pour rendre le processus plus rapide.

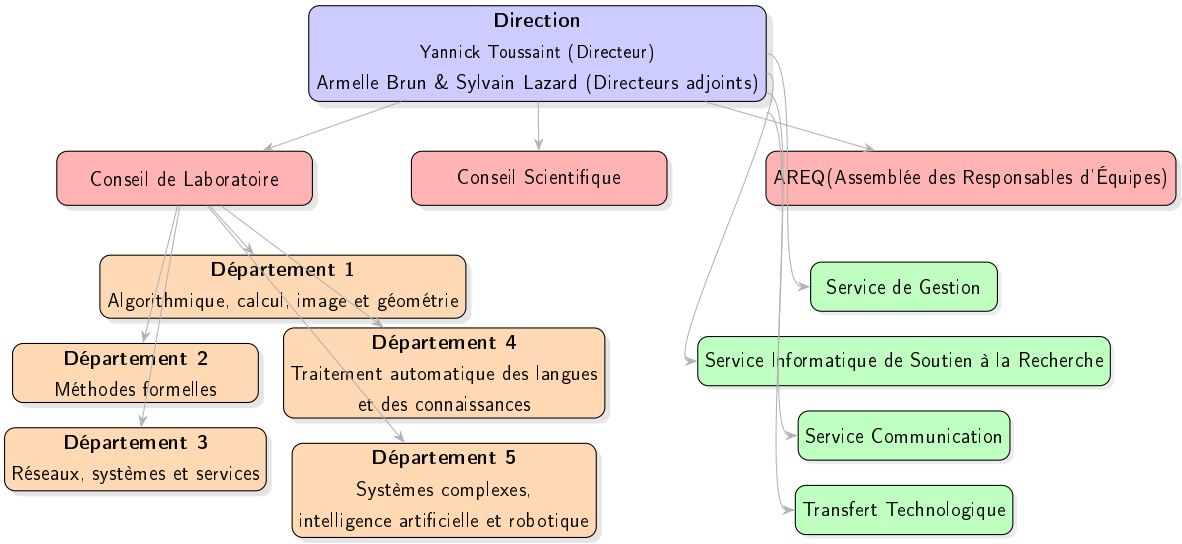
Ce travail s'inscrit dans une perspective plus large : rendre les réseaux temps-réels vérifiables, transparents et fiables à grande échelle, ce qui pourrait faciliter leur adoption dans

des domaines industriels ou critiques, où la confiance entre acteurs est primordiale.

Présentation du laboratoire

Le Loria, **Laboratoire lorrain de Recherche en Informatique et ses Applications** est une Unité Mixte de Recherche, commune à plusieurs établissements : le CNRS, l'Université de Lorraine, CentraleSupélec et Inria.
Leurs travaux scientifiques sont divisés en thématiques de recherche à travers 500 personnes au sein de 5 départements.

Ci-dessous un organigramme reprenant l'organisation du laboratoire :



Au cours de mon stage, j'ai rejoint l'équipe SIMBIOT, dont la mission est la conception et la validation de systèmes cyber-physiques dits « intelligents ». Les travaux de l'équipe se concentrent principalement sur les propriétés d'adaptabilité de ces systèmes, tant au niveau du calcul que des communications, dans le but de renforcer leur autonomie.

Chapitre 2

Sujet du stage

Dans cette partie, nous allons étudier les différents éléments clés permettant le calcul des bornes de latences ainsi que la génération de preuves de ces bornes en questions.

Calcul des bornes de latences

L'un des premiers défis de mon stage a été l'étude et la compréhension de l'algorithme **Total Flow Analysis (TFA)**. Il s'agit d'un algorithme d'analyse déterministe (affirmant, avec certitude, qu'on ne peut dépasser une certaine valeur) utilisé pour estimer des **bornes de latence dans le pire des cas**. Ces réseaux sont couramment rencontrés dans des environnements critiques où il est nécessaire de garantir le respect de délais stricts, comme dans les systèmes embarqués, les réseaux industriels ou les communications critiques.

Objectif. L'algorithme TFA sert à estimer, pour chaque flux de données dans un réseau, le temps maximum qu'un message peut mettre pour arriver à destination, en tenant compte de la manière dont le réseau est organisé. Il est conçu pour des systèmes où certains messages sont plus prioritaires que d'autres, et où les messages les plus urgents passent toujours en premier. Dans notre cas, il existe 8 niveaux de priorité, 7 étant la plus grosse priorité et 0 la plus faible.

Hypothèses. L'algorithme repose sur les hypothèses suivantes :

- Priorité stricte non préemptive (ne pouvant être interrompu) avec 8 niveaux (0 à 7).
- Absence de dépendance cyclique dans le graphe du réseau.
- Comportement FIFO (First in First Out) par classe de priorité.
- Liens de communication en duplex (pouvant à la fois recevoir et envoyer).
- Affectation statique des flux à une classe de priorité.

Algorithm 1 Algorithme Total Flow Analysis (TFA)

1: **for** $c \in [7 \dots 0]$ **do**
2: **for all** $n \in \mathcal{N}$ **do**
3: $$R_n^c \leftarrow R_n - \sum_{c'>c} r_n^{c'}$$
4: $$T_n^c \leftarrow T_n + \frac{\sum_{c'>c} b_n^{c'}}{R_n^c} + \frac{lmax_{c''\leq c}}{R_n}$$
5: $$r_n^c \leftarrow \sum_{f\in c, f\in n} r_f$$
6: $$b_n^c \leftarrow \sum_{f\in c, f\in n} b_n^f$$
7: $$D_n^c \leftarrow T_n^c + \frac{b_n^c}{R_n^c}$$
8: $$\forall f \in c, f \in n, b_{n+1}^f \leftarrow b_f^n + r_f \times D_n^c$$
9: **end for**
10: **end for**

Principe. L'algorithme parcourt les priorités dans l'ordre décroissant (du plus prioritaire au moins prioritaire), puis chaque nœud du réseau. À chaque étape, il :

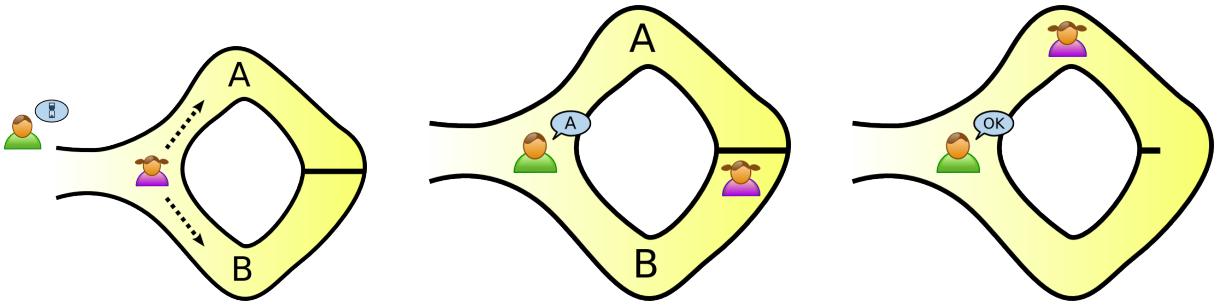
1. calcul le **débit restant** au nœud, c'est-à-dire le débit disponible une fois soustraits les débits des flux plus prioritaires.
2. calcul la **latence de service** en prenant en compte la latence technologique, les latences dues aux flux de priorités supérieures et le délai d'un paquet de priorité inférieure.
3. calcul le **débit et la burst** des flux de la priorité courante.
4. calcul la **borne de latence** en sommant la latence de service et le temps de traitement du burst.
5. **Propage le burst** aux nœuds suivants selon la formule $b_f^{n+1} = b_f^n + r_f \cdot D_n^c$.

Résultat. L'algorithme fournit une borne supérieure pour le délai que peut subir un flux à travers le réseau. Il tient compte de l'influence des flux plus prioritaires sur les flux de priorité inférieure, garantissant ainsi une analyse conservatrice, mais fiable du comportement temporel du réseau.

Dans le cadre de mon stage, je n'ai pas eu à implémenter l'algorithme TFA, une implémentation en Rust été déjà présente. Cependant, une bonne compréhension était nécessaire pour résoudre les problèmes liés à l'implémentation, visant à améliorer son exécution.

Preuve à divulgation nulle de connaissance

L'une des façons les plus célèbres d'expliquer le fonctionnement des **preuves à divulgation nulle de connaissance (Zero-Knowledge Proofs)** repose sur une métaphore inspirée du conte *Ali Baba et les quarante voleurs*, proposée par Quisquater et Guillou en 1989.
Imaginons une grotte en forme de boucle avec deux entrées, A et B, reliées par un tunnel fermé par une porte magique invisible de l'extérieur. Pour passer de A à B ou de B à A, il faut connaître un mot magique permettant d'ouvrir cette porte. Peggy (le **prouveur**) veut prouver à Victor (le **vérificateur**) qu'elle connaît le mot magique — sans pour autant le lui révéler.



(a) Étape 1 (b) Étape 2 (c) Étape 3

FIGURE 2.1 – Illustration du protocole Zero-Knowledge avec la grotte d'Ali Baba

Protocole. Le protocole se déroule en plusieurs tours identiques :

1. **Engagement - (a)** : Peggy entre seule dans la grotte, en choisissant aléatoirement l'entrée A ou B.
2. **Défi - (b)** : Victor, qui reste à l'extérieur, crie « A » ou « B », demandant à Peggy de ressortir par l'une des deux entrées.
3. **Réponse - (c)** :
   — Si Peggy connaît le mot magique, elle peut ressortir par n'importe quelle sortie, même si elle doit passer par la porte.
   — Si elle ne le connaît pas, elle ne pourra satisfaire la requête de Victor que si celui-ci demande la sortie correspondant à sa position initiale.

Interprétation. Si Peggy ne connaît pas le mot magique, elle ne peut réussir le test que dans 50 % des cas. En répétant l'expérience plusieurs fois, Victor peut devenir presque

certain que Peggy possède le mot magique. En effet, la probabilité qu'elle réussisse par pur hasard N fois de suite est $\left(\frac{1}{2}\right)^N$.

Conclusion. Cette interaction permet à Victor de **vérifier que Peggy connaît le secret**, sans jamais apprendre quoi que ce soit sur le mot magique lui-même. Ce principe est à la base des Zero-Knowledge Proofs, qui sont largement utilisés en cryptographie moderne pour prouver des identités, des calculs ou des connaissances, sans rien dévoiler d'autre que la validité de l'affirmation.

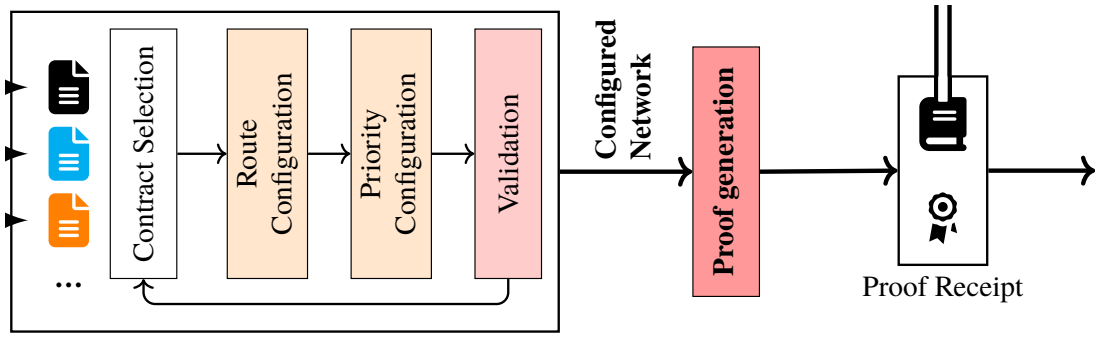
Dans notre cas, cela permet de prouver le calcul de notre borne de latence, sans avoir à divulguer une quelconque information sur le réseau utilisé.

Utilité des preuves

Dans un contexte décentralisé, où les acteurs ne se font pas nécessairement confiance, la génération de preuve devient un outil clé pour établir une confiance automatisée et vérifiable.
Imaginons un scénario typique impliquant trois entités :

- **Le client**, qui souhaite transmettre des données à travers un réseau avec des garanties strictes sur les délais.
- **L'opérateur**, chargé de configurer et de superviser le réseau.
- **La blockchain**, qui héberge une application décentralisée : un contrat intelligent jouant le rôle d'arbitre automatisé.

L'enjeu est simple : comment le client peut-il s'assurer que l'opérateur a bien configuré le réseau de façon à respecter les délais annoncés, sans pour autant avoir accès à tous les détails (souvent confidentiels) de la topologie ou de la charge du réseau ?



C'est ici que la **génération de preuve** entre en jeu.

- L'opérateur exécute l'algorithme TFA, pour évaluer si les contraintes de latence sont respectées.
- Il génère ensuite une preuve, qui atteste que ce calcul a été effectué de manière honnête.

— Cette preuve est publiée sur la blockchain, pour que l'ensemble des acteurs puissent vérifier sa validité.
— Le contrat intelligent vérifie automatiquement la validité de cette preuve.

En résumé, la génération de preuve permet de transformer une promesse ("je respecte les délais") en une garantie vérifiable et infalsifiable. C'est un composant essentiel pour automatiser la confiance dans les systèmes distribués critiques, en s'appuyant sur des preuves objectives plutôt que sur des déclarations ou une surveillance constante.

Ayant désormais un contexte sur l'utilité des bornes de latences et de la génération de preuve à divulgation nulle de connaissances, nous pouvons désormais nous concentrer sur l'objectif principal de mon stage, l'optimisation de la génération de ces dites preuves.

Chapitre 3

Travaux effectués

Le temps requis pour générer une preuve d'exécution est directement lié au nombre de cycles de l'algorithme. Nous cherchons donc à réduire ce nombre afin d'accélérer le processus de génération.

3.1 Optimisations pour une machine virtuelle

3.1.1 Profilage de code

Le profilage de code est une étape essentielle dans le processus d'optimisation d'un programme informatique. Il consiste à analyser l'exécution du programme afin d'identifier les parties les plus coûteuses en ressources, telles que le temps de calcul ou la mémoire utilisée.

Cette analyse permet de repérer les 'goulots d'étranglement', c'est-à-dire les sections du code qui ralentissent globalement l'exécution. Grâce à ces informations, il est possible de concentrer les efforts d'optimisation sur ces portions spécifiques, maximisant ainsi l'amélioration des performances.

Pour argumenter et discuter des différentes optimisations à réaliser, nous allons nous appuyer sur cet outil, mais notamment sur la documentation de 'Risc Zero' (la machine virtuelle utilisée pour la génération de ZKPs), informant sur des optimisations divergentes des optimisations sur des processeurs d'ordinateur classiques.

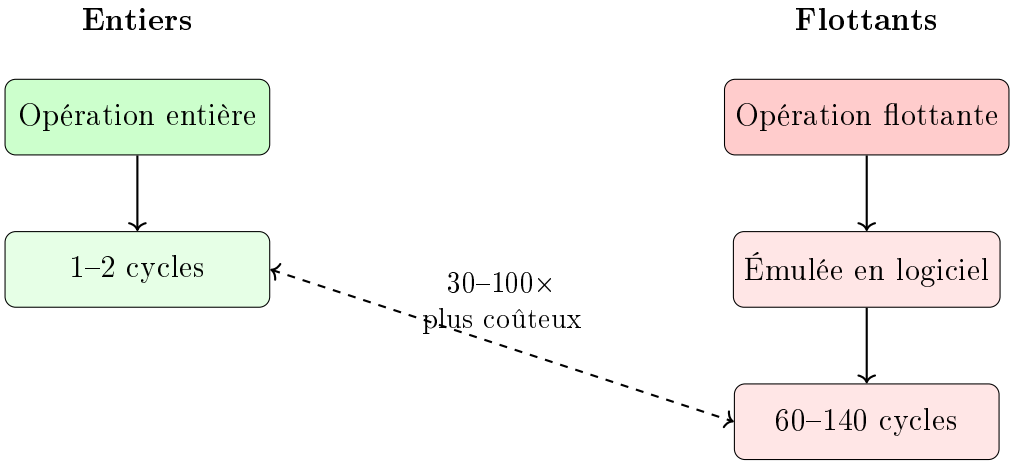
3.1.2 Conversion des flottants vers des entiers

Une première amélioration, non visible par profilage, est l'utilisation d'entiers à la place de flottants. Dans une machine virtuelle, comme celle de **RISC Zero**, le choix du type de données utilisées au sein du code à un impact significatif sur ses performances. Plus précisément, d'après la documentation de RISC Zero, les opérations sur des **entiers** sont plus efficaces que celles sur les **nombres flottants**.

En effet, la machine virtuelle de RISC Zero n'inclut pas les instructions pour le calcul en virgule flottante. Cela signifie qu'elle ne sait pas exécuter directement les opérations comme l'addition de flottants.

Puisqu'il n'y a pas de prise en charge directe, toutes les opérations flottantes sont simulées en logiciel. Cela veut dire que, pour effectuer une opération flottante, la machine virtuelle doit exécuter une série d'instructions entières qui imitent le comportement d'une addition en virgule flottante.

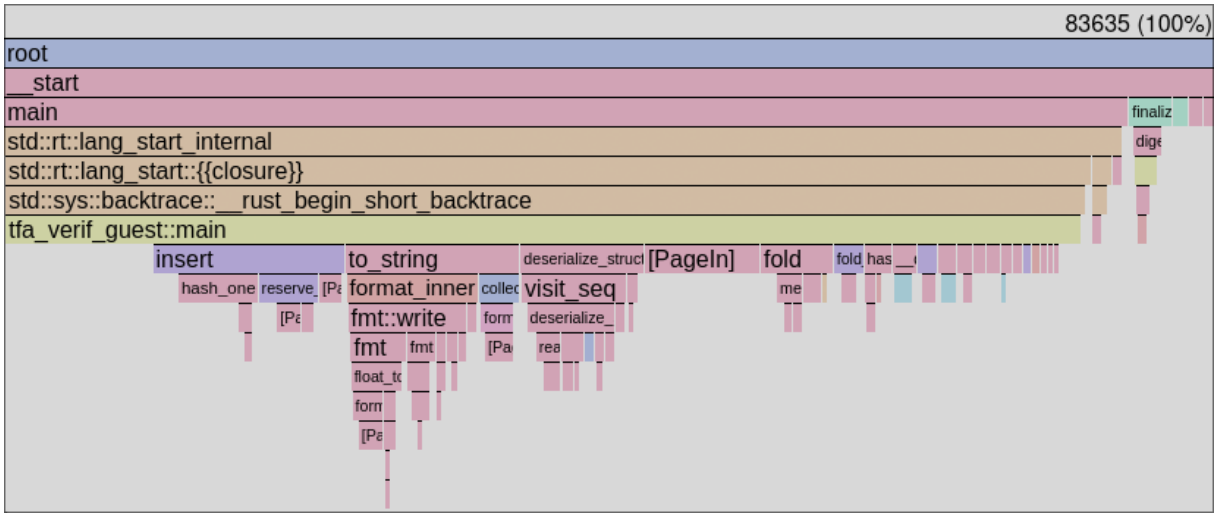
En conséquence, une opération entière prendra seulement 1 à 2 cycles, alors qu'une opération flottante consommera 60 à 140 cycles, soit 30 à 100 fois plus de temps.



Cependant, l'utilisation d'entiers n'est pas sans compromis : si l'addition, la soustraction et la multiplication ne sont pas des problèmes avec ce type de données, la division en est une. En Rust, une division d'entiers est performé en tronquant sa partie décimale. Par exemple, si avec des flottants $1.0/2.0 = 0.5$, avec des entiers $1/2 = 0$. Malheureusement, on calcule des bornes de latences, on ne peut pas se permettre de sur-estimer cette dernière en tronquant systématiquement la partie décimale.

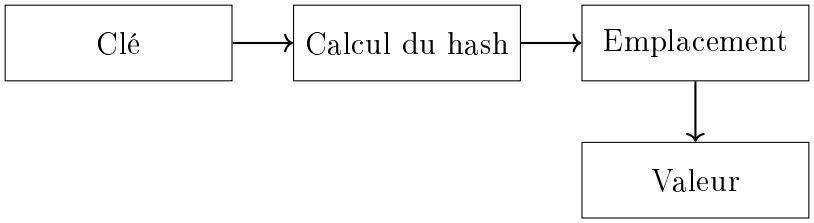
Pour pallier ce problème, il a fallu dans un premier temps convertir l'ensemble des données en entrées sur une unité plus faible pour conserver la précision apporté par leurs parties décimales (ex : 3.5ms = 3500µs) puis créer une fonction réalisant une division entière en arrondissant à l'entier supérieur. ($1/2 = 1$). Dû au changement d'unité, la perte induit par l'arrondi est négligeable.

3.1.3 Optimisation des structures de données

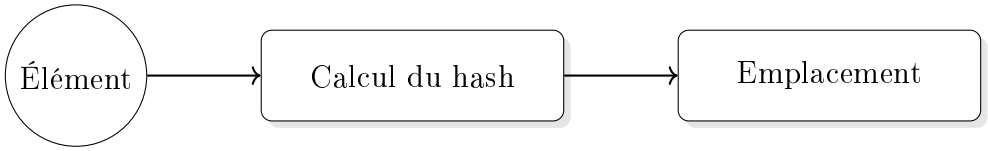


En étudiant le profilage du code, présent ci-dessus, on observe qu'une importante partie des cycles totaux est dû aux insertions (dû au bloc 'insert', étant le bloc le plus à gauche dans le profilage) dans nos structures de données : la HashMap et le HashSet.

La *HashMap* associe des clés à des valeurs en utilisant une table de hachage. Chaque clé est transformée par une fonction de hachage en un indice qui indique où stocker la valeur. Cela permet un accès rapide, mais le calcul du hachage peut être coûteux.

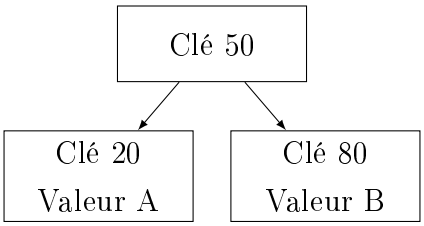


Le *HashSet* stocke uniquement des éléments uniques, sans valeurs associées. Il utilise aussi une table de hachage pour déterminer la position de chaque élément. Le fonctionnement est donc similaire au HashMap, mais sans la valeur.

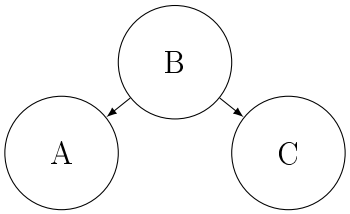


En regardant plus en profondeur le profilage, on remarque que ce qui est couteux durant les insertions est le hachage effectué sur la clé (HashMap) ou l'élément (HashSet). Pour éviter ce coût élevé en cycle, la documentation de RISC Zero recommande d'utiliser des BTreeMap et des BTreeSet.

La *BTreeMap* organise les clés et valeurs dans un arbre équilibré. Les clés sont conservées dans un ordre trié et les accès se font par comparaison (plus petit → à gauche, plus grand → à droite). Il n'y a pas de calcul de hachage, ce qui réduit certains coûts.

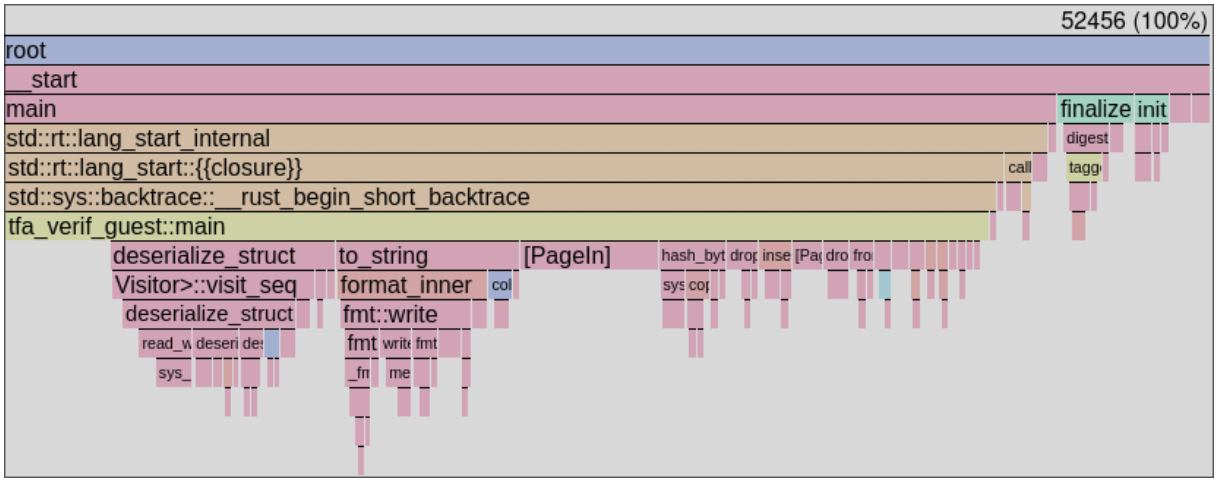


Le *BTreeSet* stocke des éléments uniques dans un arbre équilibré. Les éléments sont triés et les opérations utilisent la comparaison sans hachage.



En remplaçant l'intégralité des HashMap et des HashSet par des BTreeMap et des BTreeSet, on réduit considérablement le coût des insertions en supprimant l'utilisation de fonctions de hachage.

La réalisation d'un deuxième profilage, présent ci-dessous, vient confirmer la réduction du nombre total de cycles (voir en haut à droite), mais notamment la réduction du nombre de cycles dû aux insertions (absence du bloc 'insert' présent dans le profilage précédent).



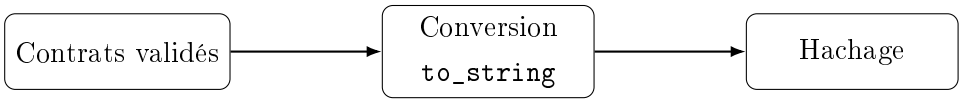
3.1.4 Changement de fonction de hachage

Lors du profilage précédent, deux blocs importants restent visibles : `deserialize_struct` et `to_string`. Le premier, indispensable, correspond à la lecture d'un fichier `JSON` par notre code, permettant de récupérer la topologie du réseau ainsi que les différents contrats le traversant.

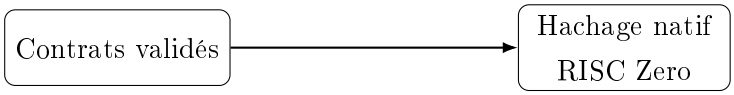
En revanche, le bloc `to_string` peut être optimisé. Cette instruction correspond à la conversion des contrats validés par l'algorithme en chaînes de caractères, en vue de leur hachage pour un envoi sécurisé.

En remplaçant cette conversion par l'utilisation directe de la fonction de hachage native de RISC Zero, on peut supprimer l'étape intermédiaire de transformation en chaîne de caractères. Cette optimisation permet de réduire significativement le nombre de cycles nécessaires au hachage des contrats validés de par la suppression des cycles dû à la conversion.

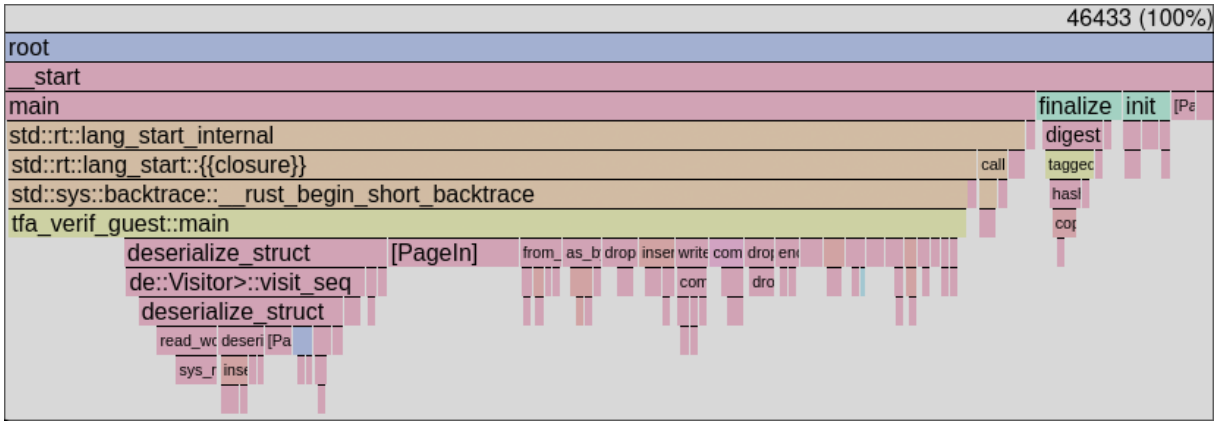
Avant optimisation



Après optimisation



Ci-dessous, le profilage final avec l'ensemble des modifications apportées, montrant ainsi une diminution importante du nombre total de cycles (83635 → 46433, soit 1,8x plus rapide).



3.2 Optimisations structurelles

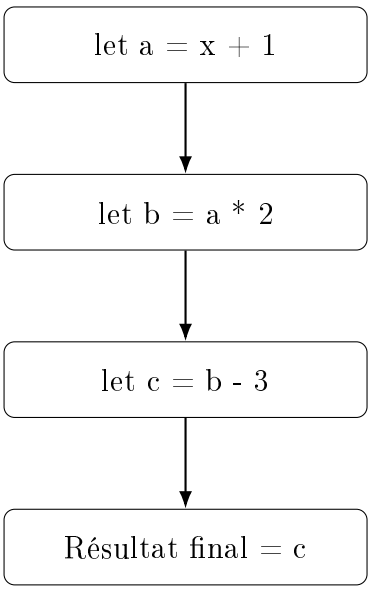
Si les optimisations précédentes était spécifiques à la machine virtuelle de RISC Zero, les optimisations structurelles, elles, sont indépedantes de la plateforme d'exécution est profitable pour n'importe quel environnement.

Elles visent à transformer le code source afin de le rendre plus efficace en termes de structure logique ou de complexité algorithmique. Cela inclut, par exemple, l'élimination de code non utilisé, la simplification d'expressions, ou encore la réduction du nombre d'itérations dans des boucles.

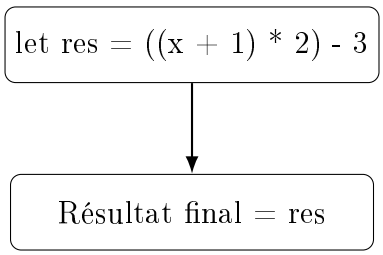
Parmi la liste des optimisations structurelles effectuées sur le code originel, la majorité permettent surtout une meilleure lisibilité du code en lui-même (notamment en utilisant de la programmation fonctionnelle). On retrouve cependant deux changements majeurs dans le code reflétant une accélération non négligeable.

La première est la réduction du nombre de variables initialisées au sein du code. Créer une variable implique une allocation en registre ou en mémoire. Chaque écriture ou lecture d'une variable coûte ainsi des cycles supplémentaires. Travailler directement sur les données en évitant au maximum la création de variables intermédiaires, permet de réduire considérablement le nombre de cycles. Ci-dessous un exemple de code non optimisé et optimisé en réduisant le nombre de créations de variables.

Code non optimisé



Code optimisé



Le calcul est direct.
Moins de variables intermédiaires.
Moins d'instructions à exécuter.
Exécution plus rapide et plus efficace.

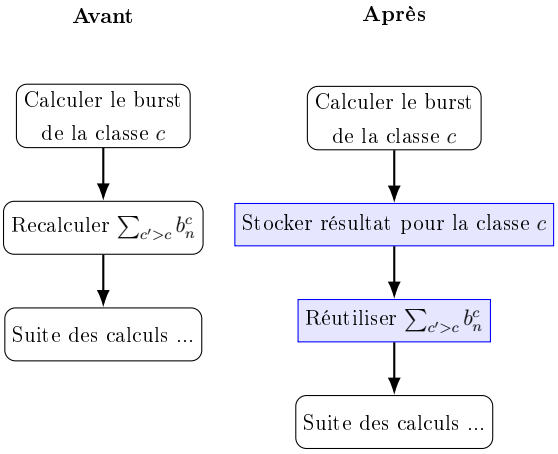
La deuxième amélioration, en lien avec la première, consiste à sauvegarder des calculs déjà effectués et qui sont destinés à être réutilisés par la suite. Regardons plus en détails les deux optimisations possibles dans l'algorithme.

Somme des bursts de priorités supérieures $\sum_{c'>c} b_n^c$

Le principe est que pour une classe de priorité c, à un nœud du réseau n, on additionne les bursts arrivant à ce nœud seulement pour les classes de priorités supérieures à celle sur laquelle on se positionne.

Originellement, on recalculait cette somme pour chaque classe de priorité. Cependant, de par le fait que l'on commence à la classe de priorité la plus élevée, nous pouvons stocker au fur et à mesure le burst de la classe calculé pour pouvoir le réexploiter dans le calcul de la somme des bursts de priorités supérieures.

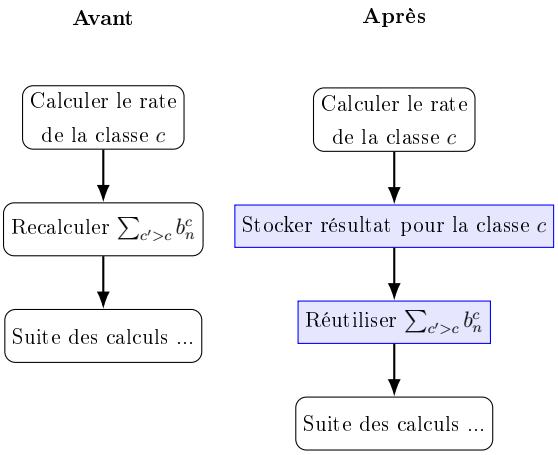
Ci-contre un schéma synthétisant la nouvelle méthode du calcul de la somme des bursts de priorités supérieures.



Somme des rates de priorités supérieures $\sum_{c'>c} r_n^c$

La deuxième optimisation consiste à réaliser un raisonnement analogue à la somme des bursts de priorités supérieures, mais cette fois-ci pour le rate.

Ci-contre un schéma synthétisant la méthode du calcul de la somme des rates de priorités supérieures, analogue à celle des bursts.



Les dernières optimisations réalisées marquant la fin de cette partie sont des optimisations propres au compilateur de Rust, appelé `rustc`, elles sont presque négligeables et n'ont pas d'intérêts à être développées dans ce rapport.

Chapitre 4

Évaluation des performances

4.1 Qu'est-ce qu'un benchmark ?

Un **benchmark** est un test ou un ensemble de tests standardisés permettant d'évaluer les performances d'un système, d'un programme ou d'un algorithme. Il permet de mesurer des métriques clés (comme le temps d'exécution, l'utilisation mémoire, etc.) en fonction de différents paramètres, afin de comparer, analyser ou améliorer une solution informatique.

Paramètres du benchmark

Dans notre cas, le benchmark fait varier deux paramètres principaux :

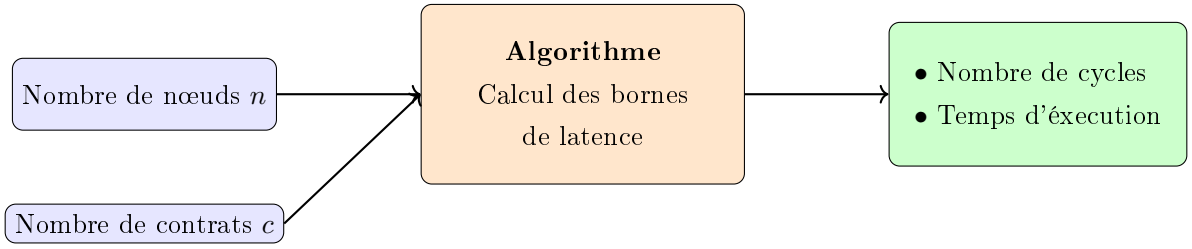
- **Le nombre de nœuds** dans le réseau (n), représentant la taille du système distribué.
- **Le nombre de contrats** (flux) (c) circulant dans le réseau.

Ces paramètres influencent directement la complexité du calcul des bornes, étant donné que le calcul des bornes de latences à une complexité de $n * c$.

Mesures effectuées

Pour chaque configuration (n, c), on mesure :

- la **nombre de cycles** enregistrés.
- le **temps d'exécution** du benchmark.



4.2 Résultats expérimentaux

Comme dit précédemment, la complexité de l'algorithme TFA dépend du nombre de nœuds comprenant le réseau, ainsi que le nombre de contrats les parcourant.
Ci-dessous deux graphiques mettant en avant le nombre de cycles enregistrés, pour chaque optimisation discutées précédemment, en faisant varier uniquement le nombre de nœuds ou le nombre de contrats.

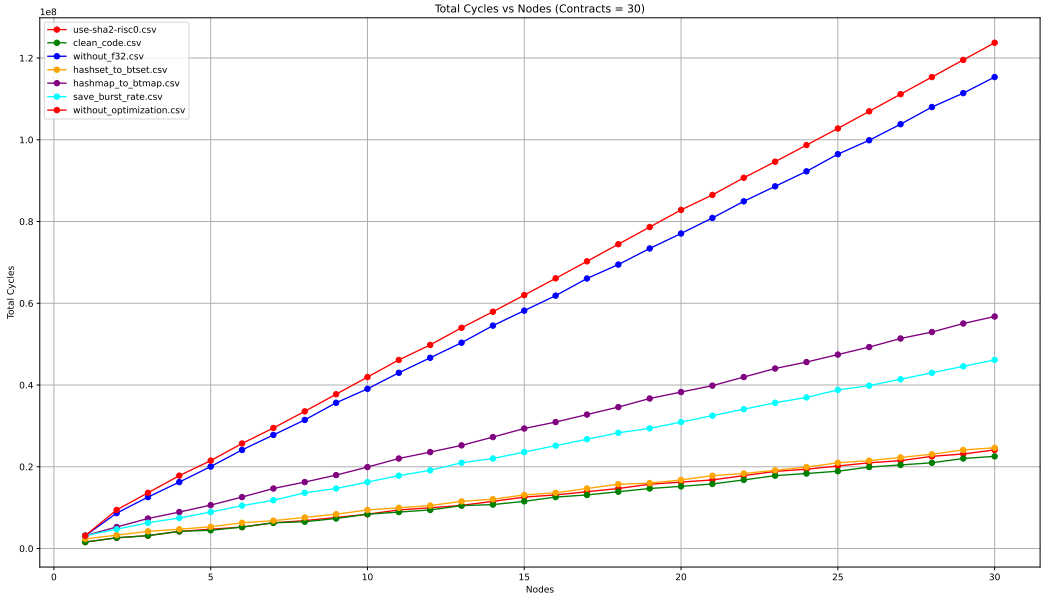


FIGURE 4.1 – Temps de calcul des bornes de latences selon le nombre de nœuds

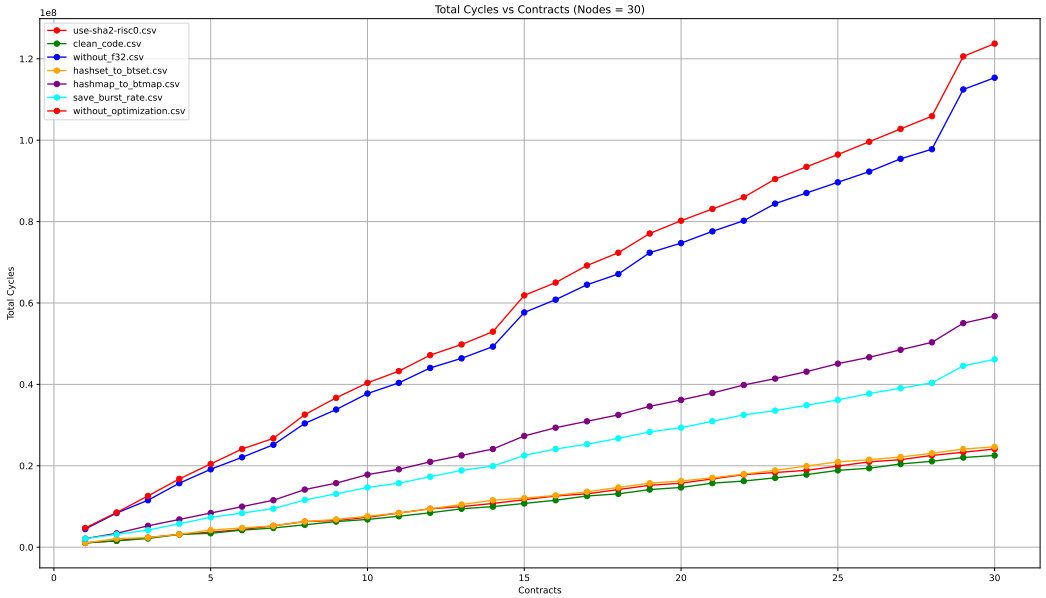


FIGURE 4.2 – Temps de calcul des bornes de latences selon le nombre de contrats

À noter que les différentes optimisations sont cumulatives, elles s'appuient sur les précédentes.

Voici un rappel des optimisations correspondantes aux différentes courbes présentent sur les deux graphiques :

- **without_optimization:** code initial, aucune optimisation
- **without_f32:** conversion de l'ensemble des flottants en entiers
- **hashmap_to_btmap:** conversion de l'ensemble des HashMap en BTreeMap
- **save_burst_rate:** optimisation de la méthode de calcul du rate et du burst
- **hashset_to_btset:** conversion de l'ensemble des HashSet en BTreeSet
- **use_sha2_risc0:** changement de fonction de hachage
- **clean_code:** suppression de code redondant et inutilisé

On observe un gain important avec l'optimisations des structures de données utilisées au sein du code, ce qui est cohérent étant donné leurs importantes présences.
L'échelle étant très importante ($1e8 = 100000000$ cycles), certaines optimisations peuvent être discutables, mais elles apportent toutes un gain de performance non négligeable.
Voici un tableau synthétisant l'ensemble des résultats de benchmark :

Optimisation	**Cycles moyens**	**Accélération**	**Réduction (%)**
without_optimization.csv	31941281.56	1.00x	0.00%
without_f32.csv	29764448.71	1.07x	6.82%
hashmap_to_btmap.csv	14783811.13	2.16x	53.72%
save_burst_rate.csv	12123104.14	2.63x	62.05%
hashset_to_btset.csv	6886004.05	4.64x	78.44%
use_sha2_risc0.csv	6505704.11	4.91x	79.63%
clean_code.csv	6158727.40	5.19x	80.72%

TABLE 4.1 – Résultats de benchmark triés par ordre croissant de réduction des cycles

Ci-dessous un diagramme en 3D mettant en opposition le code initial ainsi que le code final afin d'étudier leurs complexités en faisant varier à la fois le nombre de nœuds et le nombre de contrats.

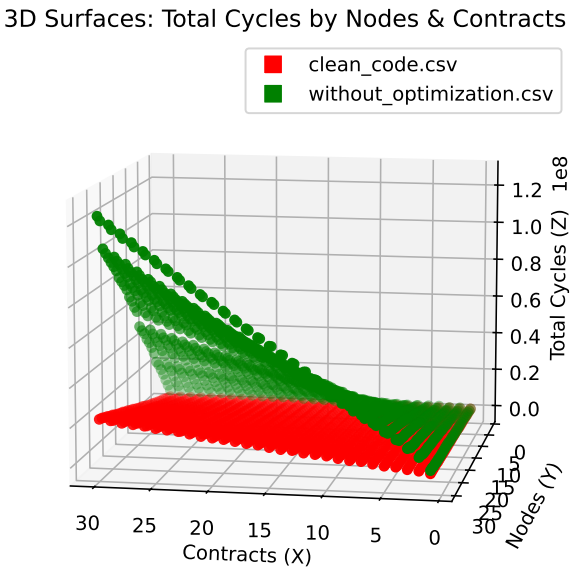


FIGURE 4.3 – Temps de calcul des bornes de latences selon le nombre de nœuds **et** le nombre de contrats

À noter que l'ensemble des benchmarks ont été fait sur un réseau paramétrique linéaire, où le nombre de contrats et de nœuds sont variables. Ces benchmarks n'évaluent pas leurs performances dans un contexte réaliste, mais permettent d'étudier le comportement des différentes optimisations apportées au code.

Cependant, nous pouvons observer sur le graphique ci-dessous, que les optimisations sont encore valables sur un exemple de réseau industriel, où les contrats sont envoyés un par un.

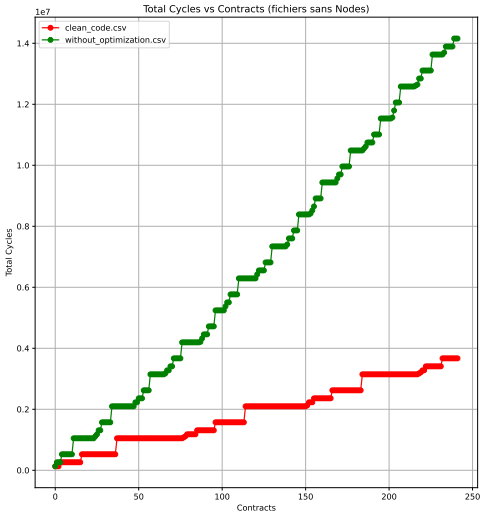


FIGURE 4.4 – Temps de calcul des bornes de latences sur un réseau industriel, (on observe des pics dans les courbes, ce sont des arrondissements de cycles pour la génération de preuves, étant donné qu'elle travail avec des cycles ronds).

Chapitre 5

Ouverture sur l'algorithme TFA++

Après l'objectif principal du stage qui était d'optimiser la génération de preuves. J'ai pu intégrer un deuxième algorithme : TFA++.

TFA++ est extension plus précise de TFA, qui **intègre explicitement les temps de transmission sur les liens entre les nœuds**. Autrement dit, l'algorithme modélise la progression réelle des paquets dans le réseau. Grâce à cette modélisation plus fine, TFA++ fournit des **bornes de délai moins pessimistes** et donc plus proches du comportement réel du système.

Ce gain en précision a toutefois un coût : en tenant compte des liens, le traitement est plus complexe, ce qui se traduit par un **temps de calcul plus long**. Mais dans les systèmes critiques, par exemple les réseaux embarqués dans l'aéronautique ou l'automobile, cette précision accrue est souvent indispensable pour garantir des propriétés de sûreté.

Ainsi, TFA++ **représente un compromis** entre rapidité d'analyse et fidélité du modèle, en choisissant de sacrifier un peu de performance pour obtenir des résultats significativement plus réalistes.

5.1 Méthode de calcul avec TFA++

Avant d'implémenter ce nouvel algorithme, il faut d'abord comprendre comment ce dernier diverge dans les calculs par rapport à **TFA**.

Avec TFA, pour chaque classe de priorité, et à chaque nœud, on récupérait l'ensemble des flux arrivant à ce dit nœud.

TFA fait ensuite la somme de ces flux entrants α_B, afin d'obtenir la courbe d'entrée du nœud étudiée. Une fois cette courbe obtenue, il nous suffit de récupérer la courbe de service β_B du nœud, c'est-à-dire la courbe représentant la capacité d'envoi du nœud.

Enfin, on calcule la distance horizontale maximale entre la courbe α_B et la courbe de service β_B, représentant le temps D_B pour qu'un bit soit envoyé par le nœud.

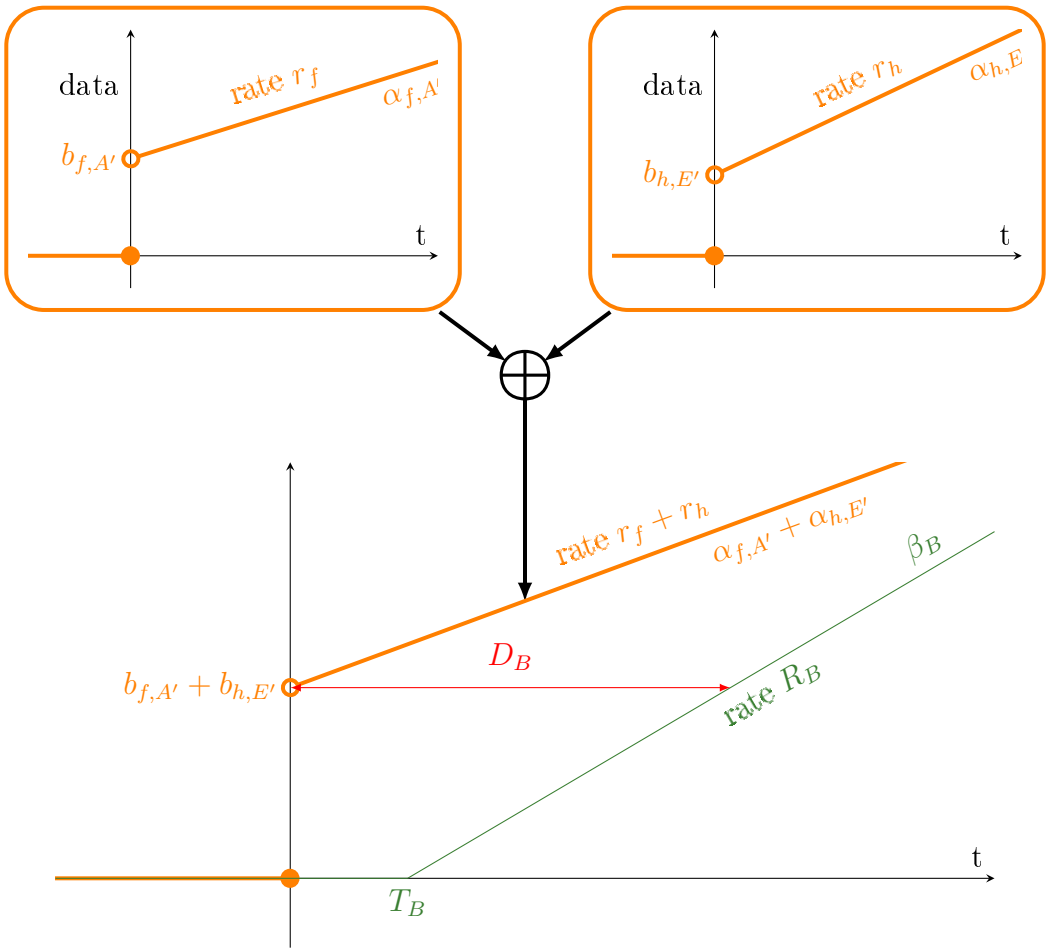


FIGURE 5.1 – Schéma synthétisant le comportement de TFA

Pour TFA++, nous ne faisons plus la somme des courbes des flux entrants, mais on construit la courbe minimale de ces courbes. La courbe de service du nœud est calculée de la même manière, et la recherche de la distance maximale entre la courbe d'entrée et la courbe de service est obtenue en récupérant le maximum des différents distances horizontales entre les points de changement de pente et la courbe de service.

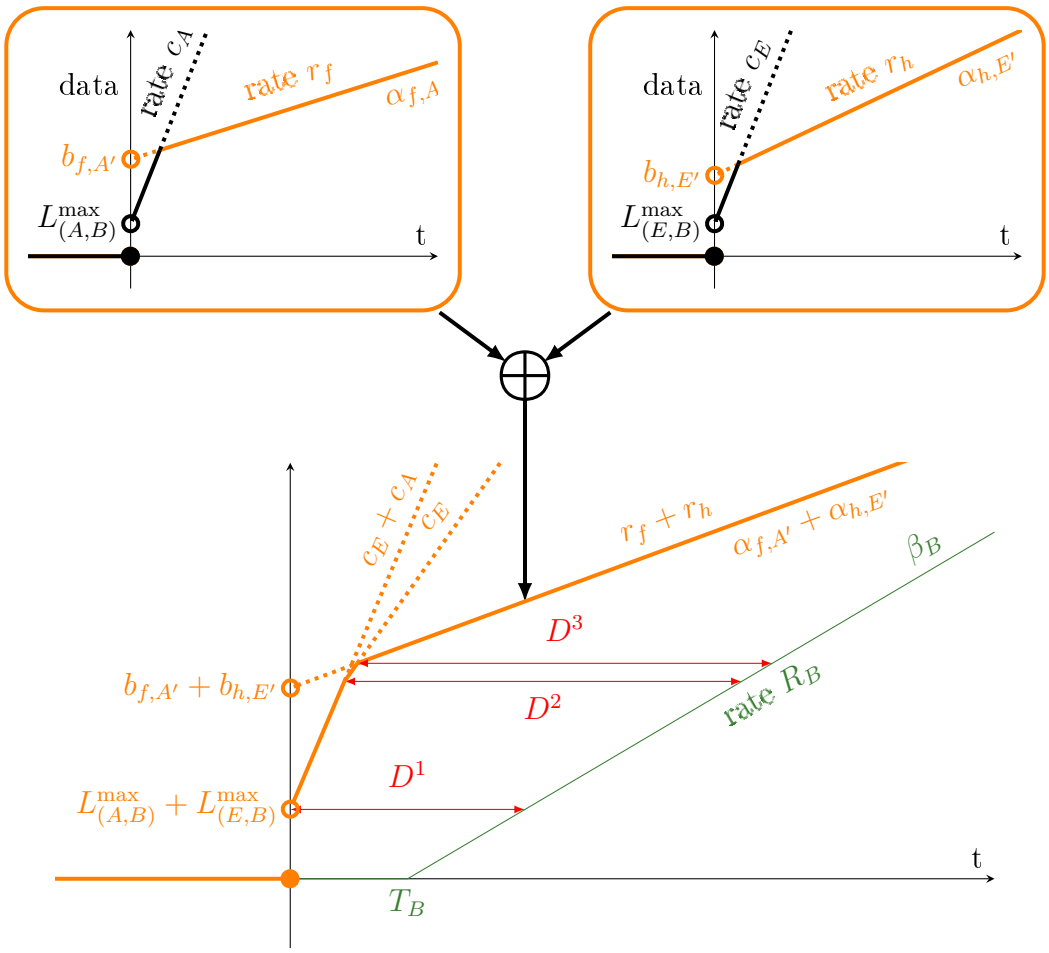


FIGURE 5.2 – Schéma synthétisant le comportement de TFA++

5.2 Implémentation de TFA ++

Dans cette partie, nous allons voir les différentes structures et méthodes qui ont été ajouté et adapté dans le code initiale de TFA pour transitionner sur le comportement de TFA++.

1. Structures de base

Segment

Chaque segment représente une droite de la forme :

$$\alpha(t) = b + r \cdot t$$

où b est le *burst* et r le *rate*.

Curve

Une courbe est une suite ordonnée de segments. Chaque segment est ajouté avec soin pour garantir que la courbe reste minimale.

2. Méthodes principales

add_segment

Cette méthode **ajoute un segment** à la courbe existante en vérifiant plusieurs propriétés :

- Supprimer les segments dominants.
- Maintenir l'ordre croissant des points d'intersection.

sum_curve

Permet de **combiner deux courbes**. Pour chaque paire de segments (s_1, s_2), on génère un nouveau segment s tel que :

$$s.\text{rate} = s_1.\text{rate} + s_2.\text{rate}, \quad s.\text{burst} = s_1.\text{burst} + s_2.\text{burst}$$

La courbe résultante est ensuite minimisée par `add_segment`.

get_intersection_points

Cette méthode retourne les points d'intersection entre segments successifs :

$$x = \frac{b_2 - b_1}{r_1 - r_2}, \quad y = r_1 \cdot x + b_1$$

Ces points sont utiles pour déterminer les ponts de changement de pente dans la courbe, nécessaires pour le calcul de la borne de délai.

compute_class_delay_bound

Calcule la borne de délai maximale selon le service d'un nœud n :

$$D_n = T_n + \frac{\alpha_n(t)}{R_n} - t$$

où :

- D_n est le délai maximum subit par un bit entrant,
- T_n est la latence de service,
- R_n le débit de service,
- $\alpha_n(t)$ est la courbe d'entrée.
- t désigne les moments correspondant aux changements de pente.

5.3 Performances de TFA++

Chapitre 6

Conclusion

[Bilan, récapitulatif des apports, ouverture.]
[TODO : Citer ludovic sur les captions Travaille réalisé avec Mathieu AMET sur les captions du code Faire Bibliographie Et citer le boss docquier]