Titre: Title:	Architectures multiprocesseurs de décodeurs séquentiels à pile
Auteur: Author:	Normand Bélanger
Date:	1989
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Bélanger, N. (1989). Architectures multiprocesseurs de décodeurs séquentiels à pile [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/57924/

UNIVERSITE DE MONTREAL

ARCHITECTURES MULTIPROCESSEURS DE DÉCODEURS SÉQUENTIELS À PILE

par

NORMAND BÉLANGER

DEPARTEMENT DE GENIE ELECTRIQUE

ECOLE POLYTECHNIQUE

MEMOIRE PRESENTE EN VUE DE L'OBTENTION

DU GRADE DE MAITRE ES SCIENCES APPLIQUEES (M.Sc.A.)

Juillet 1989

ISBN 0-315-52697-1

Canada

UNIVERSITE DE MONTREAL

ECOLE POLYTECHNIQUE

Ce mémoire intitulé:

ARCHITECTURES MULTIPROCESSEURS DE DÉCODEURS SÉQUENTIELS À PILE

présenté par: Normand Bélanger

en vue de l'obtention du grade de: maître es sciences appliquées (M.Sc.A.)

a été dûment accepté par le jury d'examen constitué de:

Bozena Kaminska	Ph. D., président-rapporteur
Yvon Savaria	Ph. D., directeur
David Haccoun	Ph. D., co-directeur
Jean Conan	Ph. D.

Sommaire

Le décodage de codes convolutionnels utilisant l'algorithme de Zigangirov-Jelinek (à pile) est très performant et nécessite un faible effort de calcul moyen. Cependant, cet effort de calcul est variable, ce qui est indésirable dans un environnement électronique numérique. Les inconvénients que cela peut entraîner sont: la nécessité d'incorporer des tampons d'entrée et de sortie au décodeur et la possibilité de déborder la capacité de mémoire de la pile du décodeur. On cherche à diminuer cette variabilité en étudiant des architectures multiprocesseurs.

La méthode utilisée consiste à définir des critères qualitatifs d'évaluation des architectures. On s'attache ensuite à trouver des architectures qui soient prometteuses lorsque confrontées à ces critères d'évaluation. Il reste ensuite à créer et mettre en oeuvre un programme informatique permettant de simuler les architectures les plus intéressantes.

Finalement, on interprète les résultats de ces simulations pour en tirer des conclusions sur l'intérêt des architectures au niveau des performances et du rapport performance/coût, sur les limites de ce type d'architectures au niveau des performances et on discute des possibilités qu'ouvrent ces architectures.

Abstract

The Zigangirov-Jelinek (stack) algorithm allows decoding convolutional codes with few computations compared to the optimum Viterbi algorithm but it has a computational variability that is undesirable. This thesis describes new multi-processor architectures which implement multi-path-like algorithms for reducing this variability.

These architectures have linear and tree structures. The first one pairs "extenders" and priority stacks (the modules which, respectively, extend tree nodes and memorize them in sorted metric order) in a linear structure. In the second architecture, the extenders are placed in a binary tree layout with its leaves connected to those of a similar tree made of priority stacks while, in the third architecture, several trees of extenders have their leaves connected to a row of priority stacks.

After defining evaluation criteria for potential architectures, we show that the proposed architectures have good potential for reducing the computational variability without adding much overhead to the system.

Simulations have shown that these architectures effectively reduce computational variability as the number of processors increases even for a relatively large number of extenders. They have also shown that the linear architecture has a better ratio of improvement of computational variability over cost compared to the tree architectures, that using 4 to 16 extenders are good choices and that, with a higher investment, one of the tree architecture can reduce the effective computational variability to zero with a probability of memory overflow around 1% if around 38% of throughput loss is acceptable.

Remerciements

Je voudrais tout d'abord remercier mon directeur, Yvon Savaria, pour m'avoir guidé au long de mes travaux et pour m'avoir accordé sa confiance malgré mes résultats scolaires mitigés. De même, j'aimerais remercier mon co-directeur, David Haccoun, dont l'interaction m'a permis de mieux saisir ce champ de connaissances que sont les communications numériques. Je tiens également à souligner que la précision des commentaires émis par ces deux personnes au sujet de versions préliminaires du présent mémoire m'a permis de l'améliorer considérablement et aussi m'a donné une compréhension beaucoup plus précise des objectifs à rencontrer lors de la rédaction d'un tel document.

Ma gratitude va également vers Nicolas Arel dont l'expérience en programmation m'a aidé à concevoir et mettre au point mon simulateur, pour avoir consacré du temps à m'aider à cette mise au point et pour son travail sur l'implantation du décodeur minimal qui a servi de modèle pour développer les modèles de coûts des architectures multiprocesseurs.

Je ne voudrais pas passer sous silence la participation de Jean Belzile à la réalisation de l'extenseur, de même pour avoir réalisé, en collaboration avec Mylène Toulgoat, la file prioritaire systolique qui m'a permis de donner validité et précision aux modèles de coûts développés dans le présent mémoire. Egalement, le travail de Pierre Lavoie, qui a défini le décodeur minimal et une première version de l'extenseur, m'a donné une base sur laquelle m'appuyer pour implanter l'extenseur et pour développer les architectures multiprocesseurs.

Table des Matières

Liste des Tableaux

Liste des Figures

Chapitre 1

Introduction

Les techniques de correction d'erreurs de transmission d'information par codage sont de plus en plus répandues, par exemple, dans les communications par satellites. Il existe deux types de codage: par blocs et convolutionnel. Les codes convolutionnels sont intéressants pour leur faible probabilité d'erreur et pour leur facilité d'implantation.

Le codage convolutionnel consiste, pour chaque cycle de codage, à introduire un ou plusieurs bits d'information dans un registre à décalage, calculer des symboles à transmettre en fonction des bits contenus dans le registre pour enfin émettre ces symboles dans le canal de transmission. Un exemple de ce type de codeur est illustré à la figure 1.1. Le nombre de cellules de mémoire de ce registre est appelé *longueur de contrainte*, le nombre de bits introduits divisé par le nombre de symboles émis est le *taux de codage* et les connexions donnent le *code*. Ce processus d'encodage peut être visualisé grâce à *l'arbre d'encodage* qui représente tous les messages possibles d'une longueur donnée. Tel que représenté à la figure 1.2, un message est le chemin parcouru, en partant de la racine de l'arbre et en allant de noeud en noeud, en suivant la branche du haut (par convention) si le bit correspondant du message

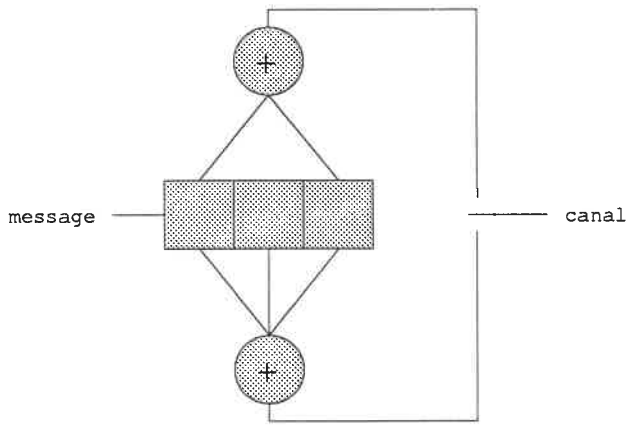


Figure 1.1: Structure d'un codeur

est 0 et en suivant celle du bas si le bit est 1 (le chemin tracé en trait gras est donc le message "1011"). Sur les branches de l'arbre sont écrits les symboles transmis dans le canal lorsque l'encodeur (qui utilise le code de l'exemple de la figure 1.1) est dans l'état correspondant à ce chemin dans l'arbre et que le bit de message courant est celui donné par la branche que l'on suit. On remarque qu'après avoir suivi des chemins identiques sur une portion de l'arbre, des chemins différents finissent par émettre les mêmes symboles tant qu'ils suivent les mêmes branches; cela est dû au fait que, le registre à décalage étant de longueur limitée, si on suit la même portion de chemin sur un nombre de branches égal à la mémoire du code (qui est la longueur de contrainte moins un), les bits des messages qui diffèrent sont tous expulsés du registre à décalage. Alors, on ne peut différencier les deux chemins (du point de vue de l'encodeur). Le diagramme en treillis (figure 1.3) est une représentation de l'arbre qui tient compte de cette *reconvergence.*

Le décodage de ces codes peut se faire selon trois approches: le décodage à seuil, le décodage par l'algorithme de Viterbi et le décodage séquentiel. Ces trois approches tentent de copier le travail que l'encodeur a exécuté pour le bloc à décoder

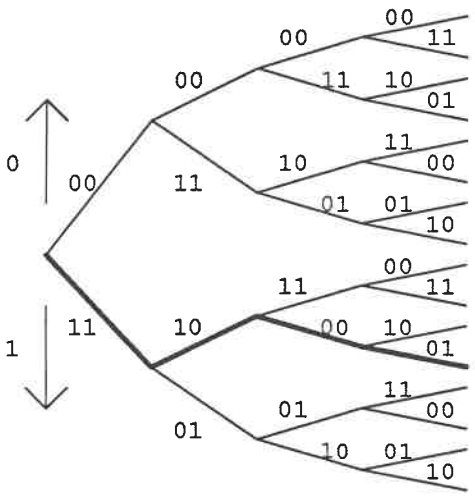


Figure 1.2: Arbre de codage

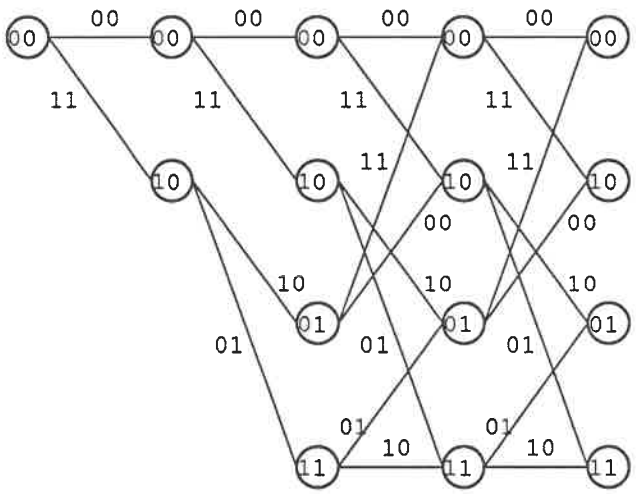


Figure 1.3: Représentation en treillis

en encodant les deux hypothèses (0 et 1) qui suivent l'état courant, et en comparant les symboles calculés avec ceux reçus.

Le décodage à seuil [1] consiste, à l'aide d'un code systématique (i.e. un code qui transmet le message en plus des parités calculées), à encoder les bits d'information reçus, calculer la somme modulo-2 de chacun de ses symboles avec chacune des parités transmises (ce qui forme les *syndrômes*), calculer une "majorité" sur un sous-ensemble des derniers syndrômes calculés et, si cette majorité dépasse un certain seuil, inverser le bit de message correspondant. C'est donc un algorithme qui nécessite un nombre de cycles de décodage constant pour un nombre de bits de message donné. Bien qu'attrayante et utilisée en pratique la technique de décodage à seuil est moins performante que les techniques de décodage probabilistes. En effet, on doit utiliser des codes orthogonaux et donc, le gain de codage n'est pas aussi élevé qu'il pourrait l'être pour une même longueur de contrainte.

Le décodage par l'algorithme de Viterbi et le décodage séquentiel (qui sont des techniques de décodage probabilistes) assignent aux hypothèses de décodage une *valeur de vraisemblance* (aussi appelée métrique) qui sera d'autant plus élevée que les symboles reçus et les symboles hypothétiques seront semblables.

L'algorithme de Viterbi [2] consiste à comparer les métriques des chemins arrivant au même noeud du treillis et à ne garder que celui qui possède la métrique la plus élevée; on obtient donc un et un seul chemin par noeud du treillis et on avance d'un niveau dans le treillis lorsque tous les chemins du niveau courant ont été étendus et comparés. Cette recherche est donc exhaustive puisqu'un chemin éliminé n'aurait jamais pu voir la métrique de ses successeurs dépasser celle des successeurs du chemin survivant puisque, les chemins ayant reconvergés, les métriques de leurs successeurs garderont toujours le même différence de valeur; les sucesseurs des noeuds éliminés ont donc été explorés implicitement ce qui démontre que la

recherche est exhaustive. On obtient donc un bloc décodé en un intervalle de temps fixe puisqu'avancer d'un pas dans le treillis demande un nombre de calculs constant et que la profondeur du treillis est fixée par le nombre de bits du message. Cependant, cet algorithme ne peut être utilisé qu'avec des codes de longueur de contrainte relativement petite (de 8 à 10 au maximum) car le nombre de calculs à effectuer à chaque cycle est proportionnel au nombre d'états possibles 2^{k-1}, où k est la longueur de contrainte. Donc l'effort de calcul croît avec cette longueur de contrainte. L'utilisation de codes de longueur de contrainte faible est, pour certaines applications, de peu d'intérêt puisque la performance d'erreur s'améliore exponentiellement avec l'augmentation de la longueur de contrainte; on serait donc intéressé à avoir un algorithme où l'effort de calcul n'est pas influencé par la longueur de contrainte.

Le décodage séquentiel consiste à calculer les métriques des chemins descendants du chemin le plus vraisemblable (celui qui a la métrique la plus grande) parmi les noeuds explorés jusqu'à ce moment. C'est donc une recherche sous-optimale mais qui demande un effort de calcul beaucoup plus faible que l'algorithme de Viterbi. Le défaut principal de cette approche est que, étant donné qu'on étend le chemin le plus vraisemblable, il se peut que celui-ci soit situé à une profondeur dans l'arbre qui soit inférieure à celle du chemin exploré à l'étape précédente; ce qui entraîne que le nombre de cycles nécessaires pour décoder un bloc n'est pas fixe. La mémoire du décodeur peut donc déborder et on devra incorporer des tampons d'entrée et de sortie au décodeur pour régulariser le débit de l'information.

Les deux principaux algorithmes de décodage séquentiels sont: l'algorithme de Fano [3] et l'algorithme de Zigangirov-Jelinek [4]. L'algorithme de Zigangirov-Jelinek consiste à garder les données concernant les noeuds explorés dans une pile (qui est typiquement une liste ordonnée) et, à chaque cycle, retirer de la pile le

noeud ayant la meilleure métrique, calculer la métrique des noeuds-successeurs de ce dernier et insérer ces noeuds dans la pile. Les étapes du décodage sont donc:

1. retirer de la file le noeud qui a la meilleur métrique
2. si ce noeud est à la fin du bloc de données, son chemin dans l'arbre donne le message décodé; sinon on calcule la métrique des successeurs du noeud
3. on insère les sucesseurs dans la file

Lorsque le noeud le plus vraisemblable n'est pas l'un des descendants du noeud qui vient d'être prolongé, on effectue un saut dans l'arbre de décodage. Il n'y a cependant pas de délai associé à ce bond (si on néglige le temps de tri de la pile) car on n'a eu qu'à trier la pile et ressortir le meilleur noeud.

L'algorithme de Fano ne conserve pas les noeuds observés dans une pile; lorsque le noeud courant subit une chute de métrique, il retrace plutôt son chemin dans l'arbre jusqu'à ce qu'il trouve un chemin de métrique plus grande. Il n'utilise aucune mémoire mais il effectue plus de calculs lorsque le canal est bruité.

Les trois principales techniques de décodage sont donc: le décodage à seuil qui demande un effort de calcul peu élevé, permet un débit fixe mais fournit un faible gain de codage, l'algorithme de Viterbi qui demande un effort de calcul élevé et de débit fixe mais qui limite la longueur de contrainte utilisable et, finalement, l'algorithme de Zigangirov-Jelinek qui demande peu d'effort de calcul, qui a un fort gain de codage et qui ne limite en rien la longueur de contrainte mais qui implique une variabilité de l'effort de calcul nécessaire pour décoder un bloc.

Plusieurs implantations de l'algorithme de Viterbi ainsi que certaines variantes ont été proposées [5,6,7,8] sa régularité permettant une implantation simple; on a également pu implanter l'algorithme de Fano [9] et l'algorithme à seuil [10] alors que, jusqu'à tout récemment, l'algorithme à pile pouvait difficilement être implanté

de façon "élégante" (bien que cela soit possible en utilisant une variante de cet algorithme [4,11]) dû à la grande quantité de mémoire requise et, surtout, à cause de la difficulté de garder la pile triée sans ralentir le décodeur de façon catastrophique. Des architectures de piles ont été proposées récemment [12,13] qui permettent de récupérer de la pile le meilleur noeud en un temps qui soit du même ordre de grandeur que celui nécessaire au calcul des métriques des successeurs d'un noeud. Ces architectures et la densité croissante des circuits électroniques permettent l'implantation de décodeurs séquentiels à pile à l'aide de quelques dizaines de circuits intégrés seulement (par exemple, voir [14]).

Avec le décodage séquentiel, on a un décodeur puissant, car il permet d'utiliser une grande longueur de contrainte, et qui nécessite, en moyenne, un effort de calcul très faible. On voudrait cependant diminuer la variabilité de l'effort de calcul inhérente à ce type de décodeur.

Une variante de l'algorithme de Zigangirov-Jelinek a été proposée [15] qui permet de ne pas avoir de débordement de pile ou de tampons du décodeur tout en diminuant la variabilité de l'effort de calcul. Cet algorithme, appelé multi-piles, consiste à décoder de façon habituelle jusqu'à ce que la pile déborde. A ce moment, on retire un certain nombre de noeuds du sommet de la pile qu'on introduit dans une autre pile et on poursuit le décodage à l'aide de cette pile. Si cette pile déborde à son tour, on procède de la même façon avec une troisième pile. On procède ainsi jusqu'à ce qu'on obtienne un message complètement décodé; à ce moment, on mémorise ce message, on vide la pile courante et on continue le décodage avec la pile précédente. Lorsqu'on obtient un autre message décodé, on retient celui qui a la métrique la plus grande et le décodage se termine lorsque le message obtenu provient de la première pile. Cet algorithme a comme désavantage de faire un compromis entre la probabilité d'erreur et la variabilité de l'effort de calcul. Par

exemple, les auteurs comparent la probabilité d'erreur obtenue par simulation de cet algorithme avec celle d'un décodeur de Viterbi de longueur de contrainte égale à 4 ce qui est très peu puisqu'on peut réaliser des décodeurs de ce type supportant des longueurs de 7 ou 8. On a donc un décodeur ayant une cumulative de l'effort de calcul de forme exponentielle plutôt que Pareto mais qui a un taux d'erreurs assez élevé.

Un autre moyen de diminuer la variabilité de l'effort de calcul a été proposé: cet algorithme, nommé multi-chemins [16]. consiste à étendre un certain nombre des meilleurs noeuds explorés plutôt qu'un seul à chaque cycle de décodage; donc, plutôt que de réduire la variabilité de l'effort de calcul en sacrifiant le taux d'erreurs, nous optons plutôt d'ajouter des processeurs ce qui ne fait qu'augmenter les coûts du système sans augmenter le taux d'erreurs, de débordement ou sacrifier tout autre critère de performance.

Cependant, cet algorithme multi-chemins pourrait difficilement être implanté de façon efficace puisqu'il nécessiterait d'introduire et de retirer de nombreux noeuds de la pile à chaque cycle d'horloge ce qui allongerait de beaucoup son temps d'accès. On pourrait par contre étendre des noeuds qui seraient de bons noeuds sans être nécessairement les meilleurs. Une façon de le faire consiste à répartir les noeuds dans plusieurs piles et à étendre, à chaque cycle, le meilleur noeud de chaque pile. Si les noeuds de métrique élevée sont bien répartis entre les piles, on a un algorithme très près de l'algorithme multi-chemins. On obtient alors un décodeur ayant un faible taux d'erreurs puisqu'il est comparable à celui de l'algorithme de Zigangirov-Jelinek et qui nécessite un effort de calcul relativement faible mais qui a une probabilité de débordement non-négligeable par rapport à l'algorithme multi-piles.

Dans le but de réduire la variabilité de l'effort de calcul et les problèmes qui lui sont associés, le présent mémoire décrit des architectures permettant d'implanter

des décodeurs pseudo-multi-chemins. On montre que ces architectures réduisent effectivement la variabilité de l'effort de calcul tout en conservant une probabilité d'erreur très faible. On montre également que la densité de transistors qu'il est possible d'atteindre sur un circuit intégré avec les technologies courantes permet la réalisation de systèmes multiprocesseurs et qu'il est possible de le faire de façon peu coûteuse. Donc, la contribution apportée par ce mémoire est de proposer une technique de décodage séquentiel permettant d'avoir une faible variabilité de l'effort de calcul tout en conservant un gain de codage élevé et un taux d'erreur très faible et ce, seulement au détriment des coûts du système; on montre également que ces coûts peuvent être assez près de ceux d'un système minimal.

La structure du présent mémoire consiste, dans le chapitre 2, à discuter d'une implantation de l'algorithme de Zigangirov-Jelinek et, en particulier, du circuit extenseur qui effectue le calcul de la métrique des noeuds explorés. Le chapitre 3 propose trois architectures implantant l'algorithme pseudo-multi-chemins et discute certains détails de fonctionnement. Les chapitres 4 et 5 quant à eux décrivent respectivement un simulateur qui a été créé pour simuler ces architectures et les résultats de simulations qui ont été effectuées pour déterminer le comportement des architectures en question. Des conclusions générales sur ces résultats et sur les travaux qu'il serait intéressant de poursuivre sont élaborées au chapitre 6.

Chapitre 2

Implantation d'un décodeur minimal

2.1 Architecture du décodeur

L'implantation d'un premier décodeur, à architecture minimale, est motivée par le besoin d'étudier les contraintes matérielles dues à ce genre de systèmes et les variantes envisageables car, il ne suffit pas qu'on ait un algorithme qui soit plus performant, il doit pouvoir être implanté de façon efficace et économique.

Ce décodeur minimal [14], est constitué de quatre modules et son architecture est illustrée par la figure 2.1. Les modules sont: la file prioritaire, l'historique, le module bloc et l'extenseur:

- Le module *file prioritaire* ordonne les noeuds visités en fonction de leur métrique et, à chaque cycle, fournit le noeud ayant la meilleure métrique.
- Le module *historique* tient à jour une séquence chronologique de chaque pas de décodage et permet la récupération de la séquence décodée par le biais

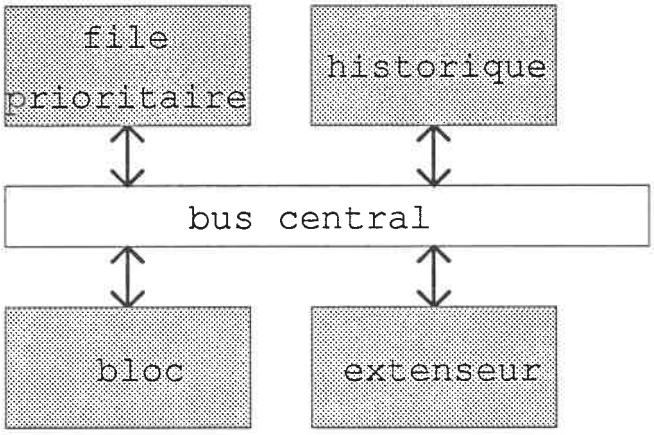


Figure 2.1: Structure générale du décodeur

d'une liste chaînée.

- Le module *bloc* garde en mémoire la séquence codée reçue pour toute la durée du décodage. Chaque paire de symboles reçue, correspondant à un bit d'information, peut être consultée directement.

- Le module *extenseur* fait simultanément l'extension des deux branches descendantes d'un noeud.

L'architecture du système est centrée sur un bus de données qui est aussi accessible au monde extérieur ce qui donne une architecture flexible et facilement testable.

Le décodage d'un bloc se fait selon les étapes suivantes:

1. le monde extérieur fournit au décodeur un bloc de symboles reçus,

2. le décodeur en effectue le décodage et

3. il renvoie le bloc décodé au monde extérieur.

On constatera que le décodeur ne contient aucun tampon ni à l'entrée ni à la sortie, le monde extérieur doit donc effectuer le tamponnage nécessaire. Les caractéristiques d'un décodeur qui est en cours d'implantation sont explicitées au tableau 2.1.

Paramètre	*min*	*max*
métrique accumulée	$+0$	$+32K-1$
métrique de symbole	-256	$+255$
métrique de branche	-512	$+510$
longueur N d'un bloc en branches	2	2047
taille de l'historique	0	$32K-1$
longueur de contrainte K	0	24
nombre Q de niveaux de quant.	2	8
numérateur U du taux de codage R_N	1	16
dénominateur V du taux de codage	2	32

Tableau 2.1: Caractéristiques du décodeur

2.2 Architecture de l'extenseur

2.2.1 Description générale

Dans le décodeur, le module extenseur est utilisé pour étendre le noeud sorti de la file prioritaire selon les branches 0 et 1 correspondant aux bits d'information 0 et 1 respectivement. L'extenseur calcule essentiellement la métrique des "fils" du noeud reçu à l'aide des symboles reçus et d'une table de métriques. Il utilise un taux de codage égal à 1/2 et, grâce à des structures permettant d'implanter des codes perforés, il permet des codes de taux allant de 17/32 jusqu'à 15/16 (pour une description des codes perforés, voir [17]). La longueur de contrainte maximale qu'il

supporte est de 24 et il permet une quantification du canal allant de deux à huit niveaux.

2.2.2 Fonction de l'extenseur

Lors du décodage, l'extenseur lit du bus central l'état (ST), la métrique du noeud-père (NM), les deux symboles reçus ($S1$ et $S2$), le bit indiquant si la branche est dans la queue du bloc (IQ) et la valeur du compteur de perforation (PE) (pour connaître la signification des abréviations utilisées, voir l'annexe B). Le compteur de perforation permet de savoir quel est le patron de perforation courant. L'extension débute par le calcul, à l'aide de l'état et de deux arbres de ou-exclusifs, des deux symboles transmis potentiels, et ce, pour chacune des branches. En comparant les quatre estimés obtenus avec les symboles reçus, l'extenseur choisit quatre métriques de symboles qui peuvent ensuite être perforées (i.e. forcées à zéro) selon la valeur du compteur de perforation et le contenu de la mémoire de perforation. La somme de deux des métriques de symboles donne une métrique de branche qui est ajoutée à la métrique du noeud-père pour donner la métrique de chaque noeud-successeur.

Etant donné que la queue du message ne contient que des zéros, lorsque le noeud sorti de la file prioritaire est dans la queue ($IQ = 1$), on sait que la branche 1 n'est pas un noeud pertinent; alors la métrique du noeud 1 est forcée à zéro pour l'éliminer. Similairement, lorsque le décodeur est dans l'état décodage marche-avant [14], la métrique du noeud successeur 1 est encore forcée à zéro, tandis que la métrique de la branche 0 est forcée à zéro pour s'assurer que la métrique accumulée du noeud successeur 0 reste la meilleure et ainsi empêcher les retours en arrière.

L'extenseur utilise des blocs de mémoire pour déterminer les conditions où il se trouve. Ces blocs sont chargés lorsque le décodeur est dans l'état initialisation et contiennent les données suivantes: les connexions des deux arbres de ou-exclusifs

(i.e. le code utilisé), les métriques de symboles, le numérateur du taux de codage et les patrons de perforation. Ces données sont identifiées au Tableau 2.2.

mémoire	*abrév.*	*bit(s)*
connexions de l'arbre 1	$C1$	24
connexions de l'arbre 2	$C2$	24
métriques de symboles	SM	8×9
numérateur du taux de codage	RN	4
patrons de perforation	PP	16×2

Tableau 2.2: Données mémorisées par l'extenseur

Notons finalement, qu'étant donnée la valeur maximale de métrique de noeud (NM), les métriques de symboles utilisées en pratique et la longueur maximale des blocs, il est impossible d'avoir un débordement de métrique de noeud et, par conséquent, aucune structure n'a été implantée pour détecter cet état.

2.2.3 Architecture et états de l'extenseur

Un schéma détaillé de l'extenseur apparaît à la figure 2.2. L'extension du noeud reçu (décrit par ST, NM et PE) commence par l'évaluation des symboles possiblement transmis à l'aide d'arbres de ou-exclusifs (module *XOR trees*). En parallèlle avec cette opération, le module *met. sel.* adresse la mémoire de métriques à l'aide des symboles reçus. Normalement, on adresse cette mémoire à l'aide des symboles reçus seulement si le résultat de l'arbre de ou-exclusif pertinent est 0, sinon, on l'adresse à l'aide de l'inverse des symboles. Cependant, étant donné qu'un premier prototype de l'extenseur a été conçu et simulé, on a constaté qu'il est le module le plus lent du système; donc, dans le but de l'accélérer, on accède la mémoire de métriques selon les deux adresses potentielles qui pointent à des zones de mémoire

distinctes (puisqu'elles sont l'invese l'une de l'autre); par exemple, si les symboles reçus sont 1 et 5, on adressera la mémoire à 1 ou 6 et 5 ou 2. Les métriques ainsi lues sont multiplexées en utilisant les résultats des arbres de ou-exclusifs pour garder les métriques pertinentes. Ceci permet de sortir l'adressage de la mémoire du chemin critique et, ainsi, accélérer l'extenseur. Les métriques ainsi calculées sont alors perforées (si nécessaire) pour être ensuite additionnées deux à deux à la métrique du noeud reçu ce qui donne les métriques des noeud-fils de ce dernier.

L'extenseur contient des blocs de mémoire conservant les données nécessaires à son fonctionnement. Ces blocs sont faits d'éléments de mémoire statique dont la structure est montrée à la figure 2.3. Le contenu de cet élément ne peut être forcé à une valeur qu'en décalant son contenu puisque la structure de son port d'écriture est celle d'un élément de registre à décalage. Donc, en reliant tous les éléments de mémoire sériellement à l'aide de leur port d'écriture, on obtient un très long registre à décalage ne requérant qu'une broche pour mémoriser les données, ce qui est un très grand avantage étant donné que le nombre de broches est très limité. On notera que les phases ϕ_1 et ϕ_2 illustrées sont en fait conditionnées de façon à effectuer le décalage lorsque l'extenseur est en mode d'initialisation; dans le cas contraire, ϕ_2 et ϕ_1 sont maintenues à 0 et 1 respectivement.

La figure 2.4 illustre "l'intérieur" du bloc intitulé *XOR trees*, ce bloc contient deux arbres de ou-exclusifs, les mémoires de connexions de ces arbres et les cellules NANDs permettant de sélectionner les bits correspondant au code utilisé. Deux des quatre symboles codés hypothétiques (ceux concernant le noeud-successeur 0) sont calculés en utilisant les arbres de ou-exclusifs, l'état (ST) reçu de l'extérieur et toutes les connexions sauf celles concernant le bit le moins significatif de chaque arbre. Pour calculer les hypothèses du noeud 1, on n'a qu'à calculer le ou-exclusif entre les deux résultats précédents et les bits les moins significatifs des connexions.

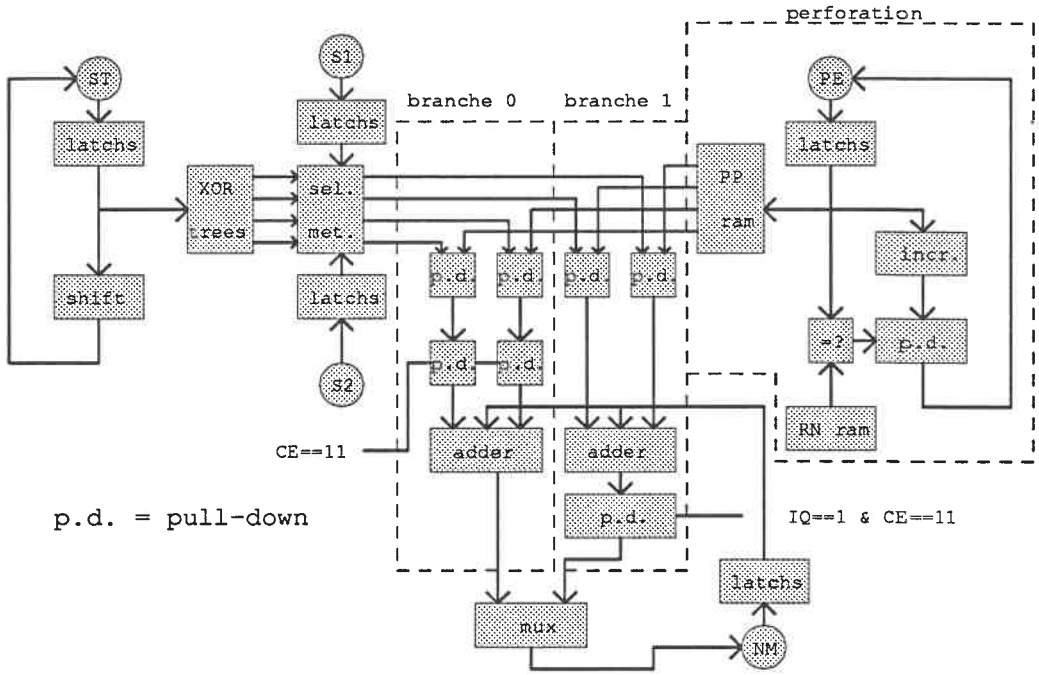


Figure 2.2: Schéma détaillé de l'extenseur

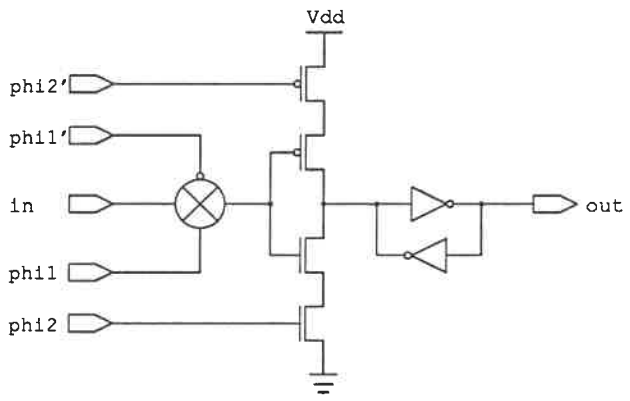


Figure 2.3: Elément de mémoire utilisé dans l'extenseur

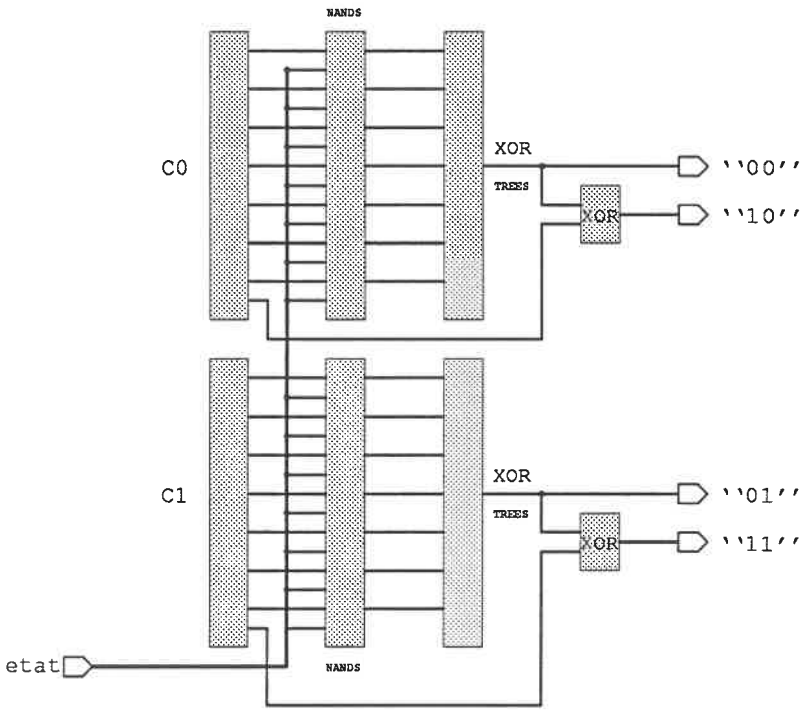


Figure 2.4: Arbres de ou-exclusifs et logique associée

La figure 2.5 illustre le bloc nommé *met. sel.*. Ce bloc est constitué de trois types de "sous-blocs": le bloc de mémoire de métriques qui est divisé en deux groupes de quatre mots de neuf bits, deux copies d'un bloc de logique et de quatre blocs de neuf multiplexeurs. Les blocs de logique adressent les blocs de mémoire selon chacun deux adresses qui sont l'inverse logique l'une de l'autre car on sait que la métrique qui sera choisie pour un symbole est à l'adresse donnée soit par le symbole reçu ou par son inverse logique selon le résultat donné par l'arbre de ou-exclusifs impliqué. La séquence des opérations est donc: calculer le ou-exclusif du premier et du troisième bit du symbole et de même pour le deuxième et le troisième (on veut s'assurer d'adresser le bloc de mémoire qui est aux adresses basses à l'aide de l'adresse dont le troisième bit est 0), adresser le bloc contenant les mots d'adresses basses avec le nombre composé de ces deux résultats et adresser l'autre bloc avec l'inverse logique de ce nombre et, finalement, sélectionner la métrique pertinente à

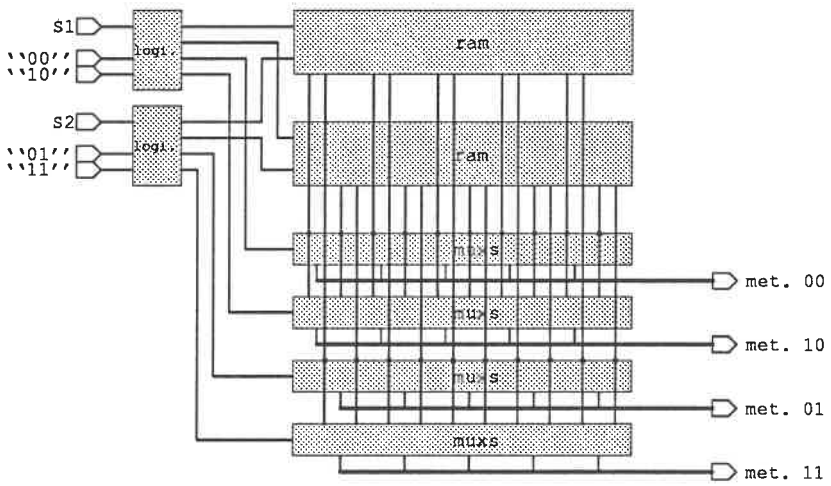


Figure 2.5: Mémoires de métriques et logique associée

l'aide du résultat de l'arbre de ou-exclusif concerné.

Par exemple, si on reçoit les symboles 2 et 6, on adresse les métriques situées aux adresses 2 et 1 (l'inverse de 6), dans le bloc d'adresse inférieure, et les métriques d'adresse 5 (l'inverse de 2) et 6 dans le bloc d'adresse supérieure. Pour s'assurer d'adresser le bloc d'adresse inférieure à l'aide de 1 et 2, on doit, tel qu'expliqué ci-haut, inverser les deux bits les moins significatifs de 2 et 6 si leur bit le plus significatif est 1 (ce qui est le cas pour 6 seulement) et on adresse le bloc d'adresse supérieure à l'aide de l'inverse de ces adresses.

Sachant qu'on lit deux métriques par symbole reçu et qu'on a deux symboles, on voit qu'on doit lire quatre métriques (deux par bloc de métriques) et, donc, on doit avoir deux ports de lecture sur les blocs de mémoire et que quatre métriques sont finalement sélectionnées (certaines pouvant être identiques).

Les modules qui reçoivent les métriques calculées selon l'algorithme qui vient d'être décrit sont les additionneurs. Ils reçoivent chacun deux des quatre métriques de branches qu'ils doivent additionner à la métrique du noeud-père. Ayant trois nombres à sommer, chaque additionneur doit contenir deux additionneurs; le choix

qui a été fait concernant la structure de ces additionneurs consiste à utiliser un additionneur "carry-save" qui ramène de trois à deux le nombre de données à additionner; on a choisi ce type d'additionneur parce que c'est celui qui est le plus rapide qu'on puisse concevoir car le délai maximum qu'il impose est celui d'une seule cellule d'additionneur. Pour obtenir le résultat final, on utilise un additionneur "carry-select". La structure de celui-ci est: un bloc de quatre additionneurs de un bit connectés en cascade et de quatre blocs de cinq additionneurs connectés de la même façon. L'additionneur complet consiste à utiliser la retenue de sortie de l'additionneur de quatre bits pour sélectionner un résultat parmi ceux de deux des additionneurs de cinq bits et la retenue de sortie ainsi sélectionnée sert à choisir le résultat du bon additionneur parmi les deux restant tel que montré par la figure 2.6. On a opté pour cet additionneur à cause de sa relativement grande rapidité et du fait qu'il a une structure statique. Un additionneur "carry-look-ahead" a été envisagé mais certaines des cellules logiques qui le composent ont un grand nombre d'entrées ce qui rend l'additionneur très lent. Etant donné qu'on désirait n'avoir que des cellules statiques, nous avons rejeté les additionneurs à calcul de retenue rapide de style "manchester" et autres.

L'implantation de la perforation utilise les blocs *PP ram*, *RN ram*, *incr.*, *"=?"* et *"p.d."* de la figure 2.2. Au niveau du système, la perforation consiste à insérer des symboles bidon aux endroits où l'encodeur doit retirer des symboles et, lors de l'extension, à forcer la métrique des symboles bidon à zéro de façon à ce que ces symboles n'influencent pas la valeur relative des métriques des noeuds visités.

Le bloc *PP ram* contient les patrons de perforation et est structuré en quatre rangées et quatre colonnes de deux cellules (un mot) de mémoire à un port de lecture. A l'aide de signaux de sélection de colonne, un mot est lu pour chaque rangée et un parmi ces quatre résultats est sélectionné à l'aide de portes de transmission

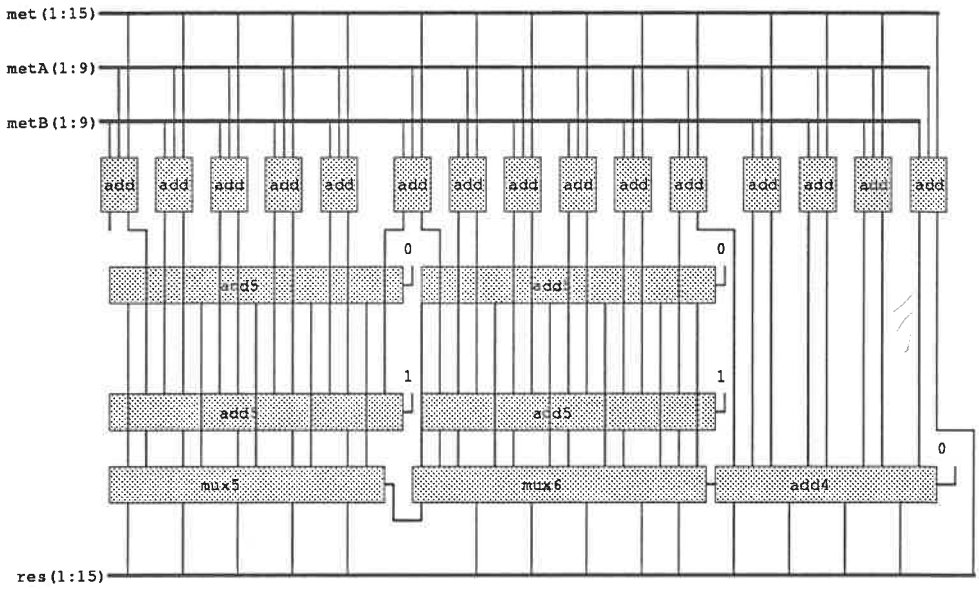


Figure 2.6: Structure des additionneurs

et de signaux de sélection de rangée.

Le bloc *RN ram* (qui contient le numérateur du taux de codage) est constitué de quatre éléments de mémoire n'ayant pas de port de lecture c'est-à-dire que leurs contenus activent toujours les lignes auxquelles ils sont connectés.

Le bloc *"incr."* est constitué de quatre demi-additionneurs connectés en cascade et avec la retenue d'entrée fixée à 1. Sa fonction est donc d'incrémenter de un le nombre qu'il reçoit.

Quant au bloc *"=?"*, il compare deux nombres de quatre bits grâce à quatre cellules XNOR2 dont les sorties alimentent un NAND4, donc le résultat est "1" si les nombres sont différents (il fonctionne en logique inversée).

Les cellules *"p.d."* ont pour fonction de forcer, si désiré, un nombre à 0. Leur structure est composée d'une porte de transmission au travers de laquelle passe (ou non) un bit du nombre à forcer à 0 et d'un transistor "n" connecté entre la sortie de

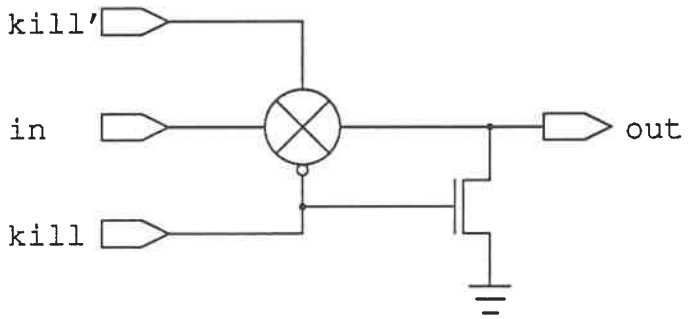


Figure 2.7: Structure des cellules "pull-down"

cette porte de transmission et la masse et commandé par l'inverse de la commande de la porte de transmission. Donc, si le signal passe la porte de transmission, le transistor est ouvert et le signal n'est pas affecté sinon la sortie de la porte de transmission est forcée à 0. Cette cellule est illustrée à la figure 2.7.

Finalement, la cellule "latch" est une porte de transmission suivie de deux inverseurs cascadés ce qui donne le noeud dynamique dont on a besoin.

L'extenseur a donc quatre modes de fonctionnement gouvernés par les signaux CE et IQ:

CE=00: *(initialisation)* si le circuit est actif, les blocs de mémoire présents sur le circuit sont initialisés grâce à la structure qui consiste à les relier en un registre à décalage,

CE=01 et IQ=0: *(décodage libre)* l'extenseur effectue le travail pour lequel il a été conçu c'est-à-dire recevoir les données concernant un noeud et en faire l'extension en ses deux noeuds-fils,

CE=01 et IQ=1: le travail effectué est le même que pour l'état précédent sauf que la métrique du noeud 1 est forcée à zéro,

CE=11: *(décodage avant)* l'extenseur fait sensiblement le même travail que lorsque

CE=01 sauf que les métriques des noeuds-fils 1 et 0 sont forcées à zéro et à la valeur de la métrique du noeud-père respectivement,

CE=10: état inutilisé.

Lorsque l'extenseur est dans un des trois derniers états, les données transférées sont: à ϕ_1, il lit ST, NM, CE, S1, S2 et PE, à ϕ_2 et ϕ_3, il écrit PE, NM, ST correspondant aux noeuds successeurs 0 et 1 respectivement. Durant CE=00, les phases ϕ_1 et ϕ_2 sont utilisées pour faire fonctionner le registre à décalage constitué de tous les blocs de mémoire présents sur le circuit.

L'extenseur a été implanté sur un circuit de 4.5mm par 4.5mm à l'aide d'une technologie à 3 μm, a été encapsulé dans un boîtier de 68 broches et il contient environ 7000 transistors. Il été soumis pour fabrication mais n'a pas encore été testé; on sait cependant qu'il contient deux erreurs qui empècheront le test complet du circuit mais les blocs qui ne pourront pas être testé ne sont qu'une assez petite partie du circuit.

Un décodeur implantant l'algorithme de Zigangirov-Jelinek ayant été décrit, le chapitre suivant discute d'extensions multiprocesseurs de cette architectures.

Chapitre 3

Extensions architecturales

Le système multiprocesseur qui serait idéal du point de vue implantation est un système ayant plusieurs extenseurs connectés à une seule file prioritaire; il permettrait d'avoir une file toujours triée et de fournir à tous les extenseurs un noeud à étendre à chaque cycle d'horloge. La puissance de calcul effective des extenseurs serait ainsi maximisée parce qu'ils seraient toujours alimentés en données et que ces données seraient celles de noeuds de métrique élevée; c'est ainsi qu'on pourrait diminuer au maximum la variabilité de l'effort de calcul.

Cependant, il serait extrêmement difficile de concevoir une file prioritaire pouvant accepter des extenseurs et leur fournir plus de trois ou quatre noeuds si on désire avoir un système ayant une fréquence d'horloge comparable à celle du système minimal. Une façon d'y remédier est d'avoir plusieurs files prioritaires qui, chacune, interagirait avec un petit nombre d'extenseurs et, par conséquent, n'aurait à lire et écrire qu'un ou deux noeuds par cycle. Le point faible de cette approche est que les noeuds de métrique élevée pourrait être mal répartis entre les files prioritaires et, ainsi, certains extenseurs pourraient recevoir des noeuds de métrique faible ce qui diminuerait la puissance effective de calcul; résoudre ce problème résulterait en

un système dont la fréquence d'horloge élevée permettrait de tirer le maximum des extenseurs et des files prioritaires. On obtiendrait alors un système performant et qui s'approcherait des performances théoriques de l'algorithme multi-chemins.

Les sections suivantes décrivent respectivement des critères d'évaluation d'architectures pseudo-multi-chemins permettant d'atteindre les objectifs que l'on vient de mentionner en plus de décrire trois architectures qui semblent bien s'y conformer.

3.1 Critères d'évaluation

Le but premier des architectures qui sont proposées est de diminuer la variabilité de l'effort de calcul inhérente au décodage séquentiel à pile classique. Dans le but d'optimiser l'utilisation des processeurs (extenseurs et files prioritaires), on cherche également à maximiser les caractéristiques suivantes:

- communications régulières ne ralentissant pas le travail des extenseurs et des files prioritaires,
- on veut étendre des noeuds ayant une métrique élevée (par rapport aux autres noeuds disponibles),
- juste partage des bons noeuds entre les files prioritaires,
- facilité d'implantation d'un algorithme multi-chemins sans retours en arrière.

On entend par communications régulières des communications qui requièrent un minimum de cycles d'horloge pour être effectuées, le but ultime étant un comportement similaire à celui du décodeur minimal, c'est-à-dire que les échanges de données se font à l'intérieur d'un cycle normal sans nécessiter de cycles suplémentaires; donc les processeurs n'attendent jamais avant de recevoir leurs données.

On désire que les noeuds étendus soient parmi ceux ayant les meilleures métriques (ce qui n'est pas assuré parce qu'on utilise plus d'une file prioritaire) dans le but de maximiser la puissance de calcul effective, car le fait d'avoir plusieurs processeurs permet d'étendre des noeuds qui le seraient plus tard sur une architecture mono-extenseur. Cependant, si les processeurs supplémentaires travaillent sur des noeuds qui ne devraient pas être étendus, on perd cette puissance de calcul.

Le juste partage des bons noeuds entre les files prioritaires vise premièrement à minimiser la taille des files et donc les coûts d'implantation et, deuxièmement, à mieux utiliser les extenseurs comme expliqué ci-haut.

Le but principal des architectures que l'on propose étant de diminuer la variabilité de l'effort de calcul, on aimerait l'annuler complètement; une façon de le faire est d'implanter un algorithme multi-chemins mais sans retours en arrière. Cependant, l'implantation d'un tel algorithme pourait s'avérer coûteuse; on désirerait donc une architecture résultant en un algorithme multi-chemins sans retours en arrière tout en étant simple (et peu coûteuse) à construire.

3.2 Variantes architecturales

3.2.1 Architecture linéaire

Description du décodeur

L'architecture linéaire est illustrée par la figure 3.1. Elle est composée des mêmes modules que le décodeur minimal soit: l'extenseur (E), la file prioritaire (F), le module historique (H) et le module bloc (B). Les données concernant les noeuds étendus vont de chaque extenseur à la file prioritaire associée à ϕ_2 et ϕ_3 et de chaque file à l'extenseur situé à droite de l'extenseur associé à ϕ_1 (sauf pour la file

de droite qui envoie ses données à l'extenseur de gauche). De plus, elle contient un module nommé "tri" dont la fonction est décrite plus loin. On peut voir que cette architecture est très simple à implanter. En effet, les seuls changements par rapport au décodeur minimal sont que:

- le module bloc est divisé en sous-blocs mais le monde extérieur l'accède de la même manière que dans le cas du décodeur minimal; il y a donc peu de complexité ajoutée,
- les données stockées par l'historique contiennent plus d'un bit d'information (symbole décodé) par enregistrement,
- les extenseurs et les files prioritaires reçoivent un signal indiquant si les données reçues concernent des noeuds valides (car, durant les premiers cycles de décodage, certaines files seront vides; elles n'enverront que des données factices aux extenseurs et réciproquement),
- on devra ajouter un état au système permettant, une fois le décodage terminé, de faire avancer les noeuds dans la chaîne et, ainsi, pouvoir sortir le meilleur noeud,
- on doit ajouter un module *tri* qui servira à trouver le noeud qui, parmi les noeuds qui seront à la fin de l'arbre de décodage, aura la plus grande métrique (voir le point précédent).

Donc, grâce aux similitudes entre l'architecture minimale et celle-ci, on pourra réaliser cette architecture rapidement et ce, avec des risques d'erreurs plus faibles.

Si on confronte cette architecture aux critères décrits plus haut, on constate que:

- les communications sont régulières et n'impliquent aucun ralentissement du travail des processeurs,

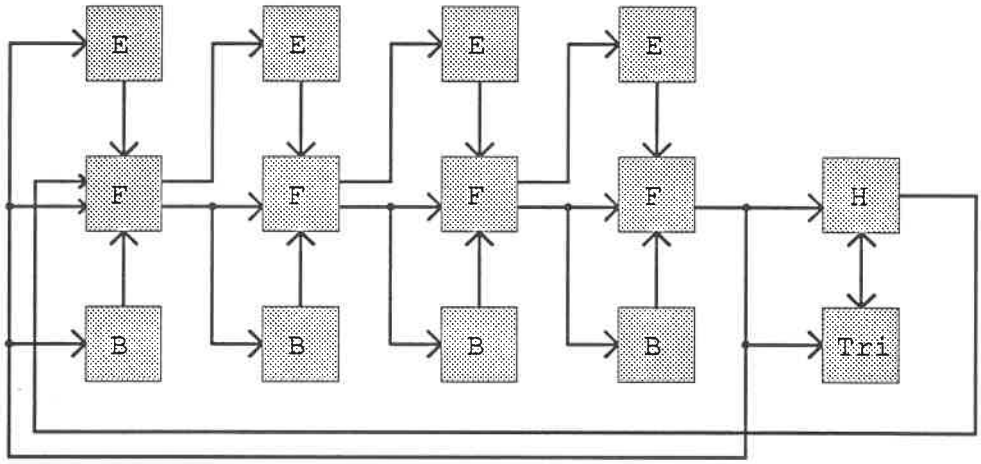


Figure 3.1: Structure générale du décodeur "linéaire"

- les noeuds étendus sont parmi les meilleurs car ils sont triés à chaque cycle et les noeuds les meilleurs ressortent rapidement des files prioritaires pour être étendus à nouveau,

- le partage des bons noeuds entre les files prioritaires est potentiellement sous-optimal parce que les bons noeuds sont toujours envoyés vers la droite mais le meilleur noeud de la file de droite est envoyé à l'extenseur de gauche donc on s'attend à ce que, statistiquement, le partage des noeuds soit bon,

- l'implantation d'un algorithme multi-chemins sans retours en arrière est très simple puisqu'on n'a qu'à trier les noeuds selon le nombre obtenu en concaténant la métrique à la profondeur du noeud dans l'arbre, cette dernière occupant les bits les plus significatifs et, éventuellement, en enlevant un certain nombre de bits à la profondeur ce qui donnerait un pouvoir discriminatoire variable (i.e. le nombre de chemins ne serait plus fixé par le nombre de processeurs).

Les étapes du décodage d'un bloc de données

La première étape consiste à initialiser les différents modules, c'est-à-dire charger leurs mémoires de configuration. Cette étape comprend également l'introduction les données relatives à la racine de l'arbre de décodage dans la dernière file prioritaire de la chaîne. On veillera à initialiser les autres files prioritaires de façon à signaler qu'elles ne contiennent aucune donnée valide.

On doit ensuite entrer les symboles reçus dans les modules *bloc*. On doit mettre dans le même *bloc* les symboles dont la position dans le message modulo le nombre de paires extenseur-file prioritaire donne le même résultat. Pour ce faire, on active le bon module en utilisant les bits les moins significatifs de la position des symboles et on adresse la mémoire du *bloc* à l'aide des bits les plus significatifs. On notera que l'utilisation d'un nombre de paires extenseur-file prioritaire qui soit une puissance de deux facilite grandement cet adressage. La fin de cette étape est signalée par le monde extérieur. On passe alors en mode de décodage.

Le système passe automatiquement à l'étape suivante lorsque l'historique constate que le dernier noeud reçu est à la fin du bloc. A ce moment, l'historique signale au module *tri* de commencer à mémoriser les données concernant les noeuds sortants et il indique au premier extenseur que les données qu'il reçoit ne sont pas valides.

L'étape suivante consiste (à l'aide du module *tri*) à comparer les métriques des noeuds interceptés et prendre celui qui a la métrique la plus grande (si ce module est suffisamment rapide, cette étape sera intégrée à la précédente). Finalement, la dernière étape consiste à retracer le chemin correspondant à ce noeud à l'aide de l'historique comme dans le cas du décodeur minimal, le dernier symbole de ce chemin (car on retrace à rebours) étant donné par le module *tri*.

L'algorithme implanté par cette architecture est: à chaque cycle, le meilleur

noeud de chaque pile est retiré et est prolongé, les noeuds résultants sont envoyés à la pile située à droite de la pile courante et ils sont triés avec les autres noeuds contenus dans cette pile. Ces extensions se font simultanément et, donc, les noeuds créés durant un cycle ne deviendront candidats à une autre extension qu'au cycle suivant.

Les avantages de ce système sont:

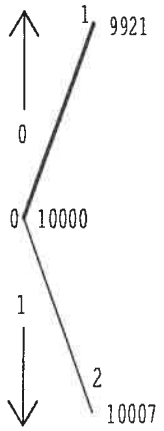
- il implante un algorithme s'approchant du multi-chemin donc avec une variabilité de l'effort de calcul réduite,
- il évite le goulot d'étranglement que représenterait l'utilisation d'une seule pile,
- on est certain d'étendre le meilleur noeud à chaque cycle mais les autres noeuds ne sont pas nécessairement les meilleurs bien qu'ils soient très près de l'être généralement,
- les extensions se font en parallèlle donc il n'y a pas de ralentissement causé par le nombre de processeurs,
- les piles fonctionnent de façon très similaire à celle du système minimal ce qui permet d'utiliser les même circuits intégrés.

Exemple de décodage

Dans le but de montrer la dynamique de ce décodeur, on présente dans cette section un exemple de décodage du début d'un bloc. On illustre chaque pas de décodage par l'arbre montrant les extensions faites jusqu'à cette étape et par un tableau donnant le numéro des noeuds reçus et produits jusqu'au cycle courant (ces numéros permettent de faire le lien entre un arbre et le tableau correspondant). On montre

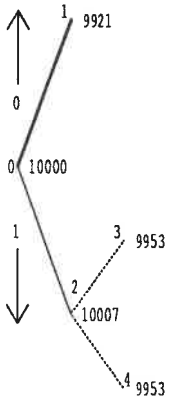
les huit premiers cycles de décodage et, sur les arbres, les extensions effectuées durant le même cycle ont le même patron de ligne (cependant, par manque de patrons différents, certains cycles partagent le même patron). Les métriques des noeuds sont également affichées sur les arbres.

Le premier arbre permet de voir que, durant le premier cycle, un seul extenseur est occupé puisque le seul noeud disponible est la racine de l'arbre.



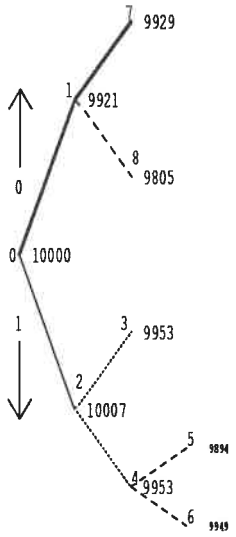
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—

Lors du deuxième cycle également, un seul noeud est étendu puisque les noeuds successeurs de la racine de l'arbre sont dans la même file prioritaire.



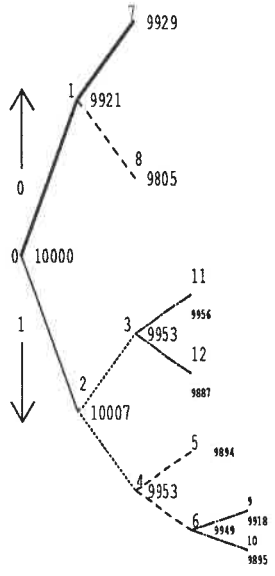
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—

Le troisième cycle voit le deuxième noeud-fils de la racine ressortir de sa file (ce qui est normal puisqu'il est le seul noeud de cette file) alors qu'un des fils de son "frère" est étendu par un autre extenseur.



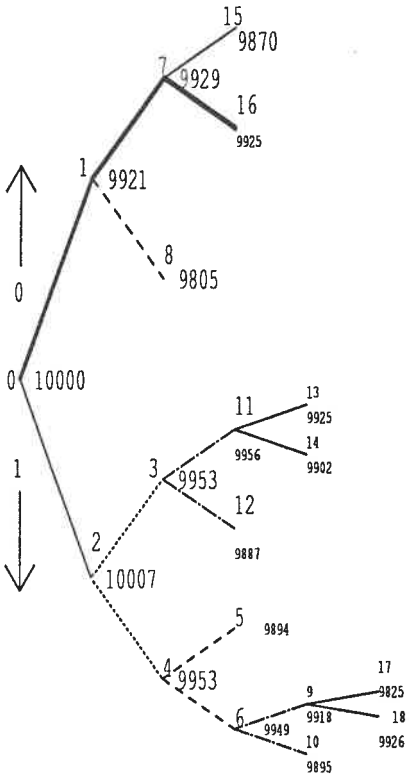
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—

Au quatrième cycle, les deux premiers extenseurs sont inactifs puisque la racine de l'arbre et ses noeuds-fils ont été étendus, alors que les autres étendent un deuxième noeud de troisième génération et un premier de quatrième génération.



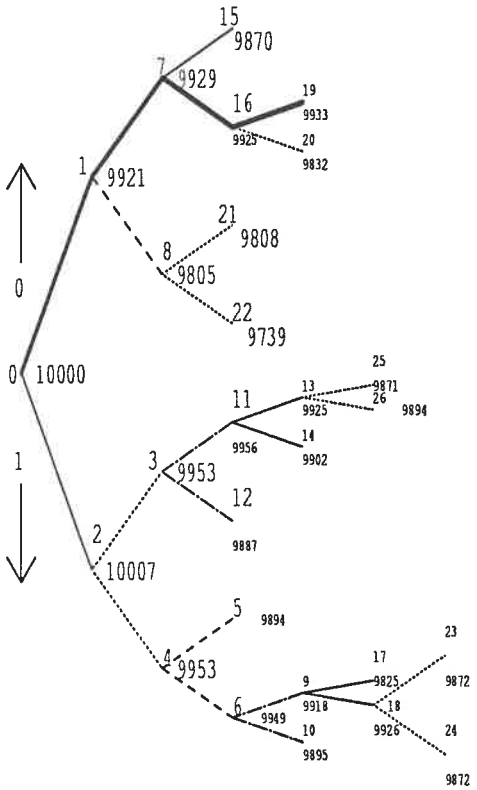
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—
3	—	—	$3 \rightarrow 11,12$	$6 \rightarrow 9,10$

Au cycle suivant, le premier extenseur reçoit un noeud (situé à une profondeur de 4 dans l'arbre) de la quatrième file prioritaire pendant que les autres extenseurs sont aussi actifs sauf le deuxième qui n'a pas encore reçu de noeud venant de la deuxième vague qui s'amorce pendant ce cycle.



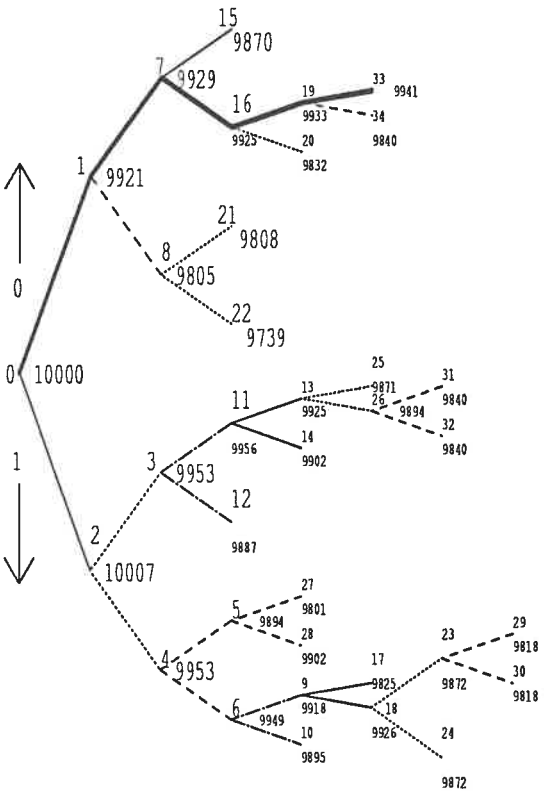
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—
3	—	—	$3 \rightarrow 11,12$	$6 \rightarrow 9,10$
4	$9 \rightarrow 17,18$	—	$7 \rightarrow 15,16$	$11 \rightarrow 13,14$

Durant ce sixième cycle, tout les extenseurs sont occupés. Les deux premiers procèdent à l'extension de noeuds issus de la deuxième vague alors que les deux autres étendent des noeuds de la première vague.



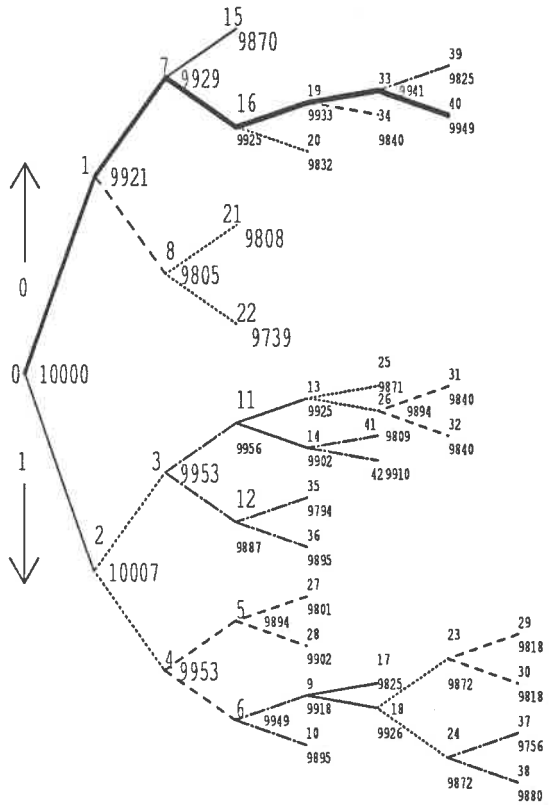
cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—
3	—	—	$3 \rightarrow 11,12$	$6 \rightarrow 9,10$
4	$9 \rightarrow 17,18$	—	$7 \rightarrow 15,16$	$11 \rightarrow 13,14$
5	$13 \rightarrow 25,26$	$18 \rightarrow 23,24$	$8 \rightarrow 21,22$	$16 \rightarrow 19,20$

Lors du septième cycle, seul le dernier extenseur reçoit un noeud de première vague. On remarque également qu'ils sont tous occupés et ils le seront jusqu'à la fin du décodage de ce bloc. On voit également que, à partir de ce moment, les noeuds contenus dans une même file pourront être à différentes profondeurs, ce qui est d'ailleurs le cas pour les noeuds de la troisième file.



cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—
3	—	—	$3 \rightarrow 11,12$	$6 \rightarrow 9,10$
4	$9 \rightarrow 17,18$	—	$7 \rightarrow 15,16$	$11 \rightarrow 13,14$
5	$13 \rightarrow 25,26$	$18 \rightarrow 23,24$	$8 \rightarrow 21,22$	$16 \rightarrow 19,20$
6	$19 \rightarrow 33,34$	$26 \rightarrow 31,32$	$23 \rightarrow 29,30$	$5 \rightarrow 27,28$

Finalement, le huitième cycle nous fait voir deux extenseurs étendant des noeuds de la premième vague soit: le dernier, car la métrique du noeud 12 est plus grande que celle des noeuds 29 et 30 et le premier car la dernière file ne contient que des noeuds de la première vague.



cycle	ext. 0	ext. 1	ext. 2	ext. 3
0	$0 \rightarrow 1,2$	—	—	—
1	—	$2 \rightarrow 3,4$	—	—
2	—	$1 \rightarrow 7,8$	$4 \rightarrow 5,6$	—
3	—	—	$3 \rightarrow 11,12$	$6 \rightarrow 9,10$
4	$9 \rightarrow 17,18$	—	$7 \rightarrow 15,16$	$11 \rightarrow 13,14$
5	$13 \rightarrow 25,26$	$18 \rightarrow 23,24$	$8 \rightarrow 21,22$	$16 \rightarrow 19,20$
6	$19 \rightarrow 33,34$	$26 \rightarrow 31,32$	$23 \rightarrow 29,30$	$5 \rightarrow 27,28$
7	$14 \rightarrow 41,42$	$33 \rightarrow 39,40$	$24 \rightarrow 37,38$	$12 \rightarrow 35,36$

On voit donc que les noeuds qui sont étendus par un même extenseur sont situés à des profondeurs dans l'arbre dont le reste de leur division par le nombre d'extenseurs est identique. De même, l'information circule de gauche à droite et, lorsque rendue à la fin de la chaîne de file prioritaire, les noeuds sont envoyés à l'extenseur de gauche. Donc, un seul extenseur (à la fois) étend le bon chemin (évidemment!) alors que les autres font avancer des chemins qui sont situés à des profondeurs dans l'arbre qui sont différentes diminuant ainsi la longueur des retours en arrière produits lorsque le noeud courant subit une chute de métrique.

On remarque que, malgré le fait que durant les premiers cycles certains extenseurs ne sont pas utilisés, le bon chemin (illustré en traits gras) est décodé sur 6 branches en 8 cycles donc il y a peu de latence malgré le nombre relativement faible de processeurs; de plus, le système minimal aurait besoin de 12 cycles pour avancer de ces 6 branches ce qui illustre bien le gain obtenu même pour un nombre assez limité d'extenseurs.

On voit également que, étant donné qu'à chaque cycle on n'a besoin qu'elles ne rendent que le meilleur noeud qu'elles contiennent, les files prioritaires ne sont pas obligées de trier tous leurs noeuds, tel que dit à la section 2.1.

3.2.2 Architecture arborescente reconvergente

Description du décodeur

Cette architecture arborescente est illustrée à la figure 3.2. On peut y voir que les extenseurs sont disposés en arbre. Les files prioritaires sont également disposées en un arbre, mais celui-ci est reconvergent. On devra, encore, modifier les extenseurs et files prioritaires pour pouvoir leur indiquer si les données reçues sont valides.

Si on confronte cette architecture aux critères décrits plus haut, on constate les

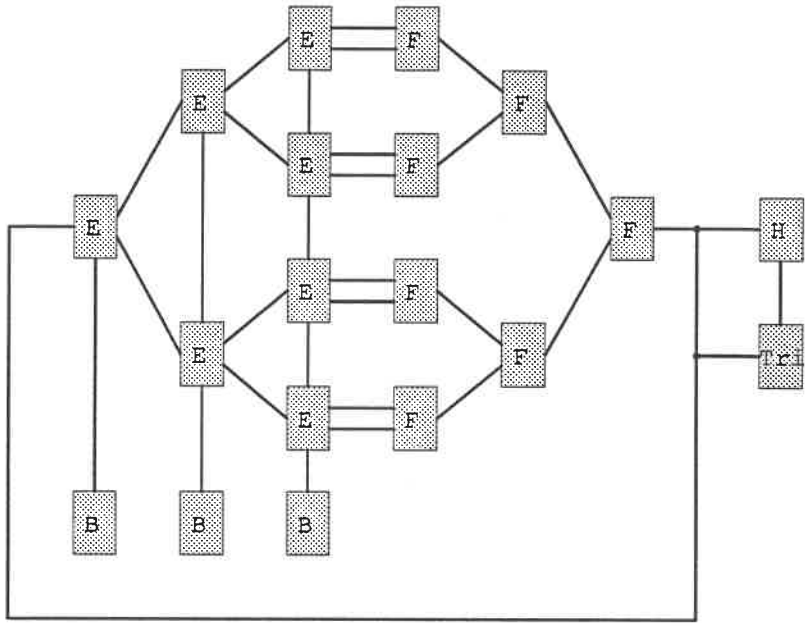


Figure 3.2: Structure du décodeur "arborescent reconvergent"

points suivants:

- les communications sont efficaces,

- le partage des bons noeuds entre les files prioritaires est mauvais parce que les bons noeuds sont envoyés vers la droite donc la file à la racine de l'arbre contiendra une grande portion des bons noeuds,

- plus les arbres sont profonds, plus la quantité de noeuds ayant une métrique faible augmente,

- elle offre une grande facilité d'implantation d'un algorithme multi-chemins sans retours en arrière qui consiste à faire les mêmes modifications que pour l'architecture linéaire.

Les étapes du décodage d'un bloc de données

On initialise d'abord les différents modules. Cela consiste à initialiser toutes les mémoires de configuration et à forcer les données relatives à la racine de l'arbre de décodage dans la file prioritaire à la racine de l'arbre alors que les autres files prioritaires sont initialisées de façon à signaler qu'elles ne contiennent aucune donnée valide.

On doit ensuite transmettre les symboles reçus aux modules *bloc*. Pour ce faire, on procède de la même manière que dans le cas de l'architecture linéaire c'est-à-dire que les symboles ayant la même valeur de position modulo la profondeur des arbres d'extenseurs sont transmis au même *bloc*.

Lors de la troisième étape, on doit simplement mettre le décodeur en mode de décodage lorsque le monde extérieur signale la fin du bloc de données. Ensuite, le système passe à l'étape suivante lorsque l'historique signale que le dernier noeud reçu est à la fin du bloc. A ce moment, le décodage continue alors que l'historique signale au module *tri* d'intercepter les données concernant les noeuds sortant de la file prioritaire et envoie au premier extenseur un signal indiquant que les données reçues ne sont pas valides.

Les dernières étapes consistent à comparer les métriques des noeuds interceptés, prendre celui qui a la métrique la plus grande et à retracer ce chemin dans l'arbre à l'aide de l'historique et du symbole de départ fourni par le module *tri* comme dans le cas de l'architecture linéaire.

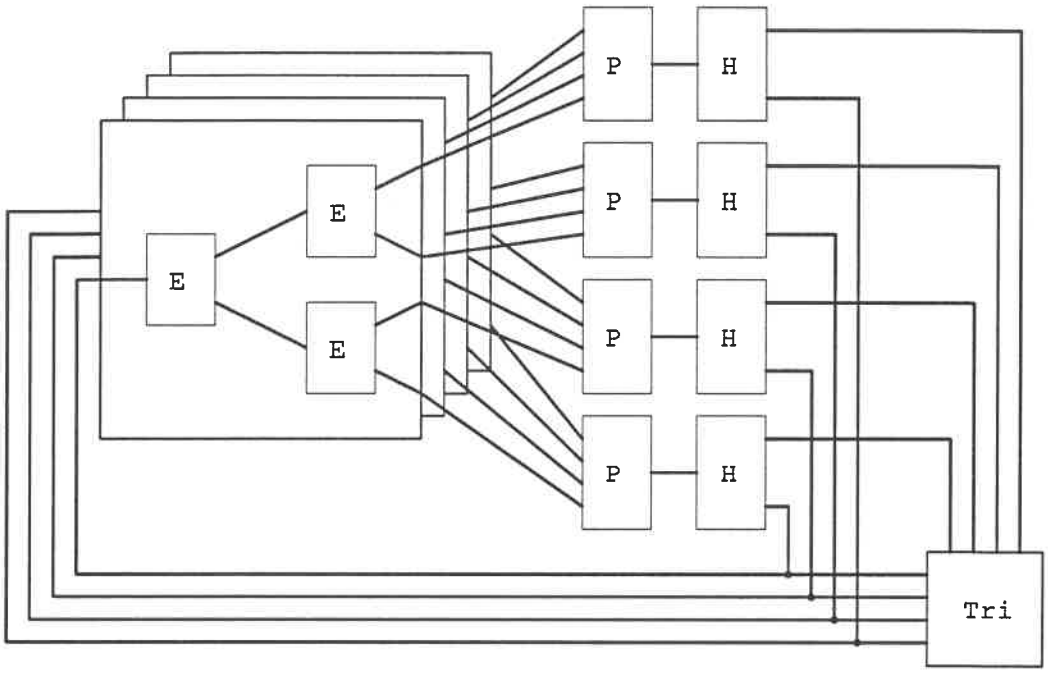


Figure 3.3: Architecture arborescente non-reconvergente

3.2.3 Architecture arborescente non-reconvergente

Description du décodeur

De la discussion de la section précédente, on voit que, pour obtenir une architecture en arbre plus efficace, on doit effectuer les changements suivants: dans un premier temps, on doit limiter la profondeur des arbres, ensuite, on doit s'assurer que les résultats seront bien répartis entre les files; pour ce faire, on peut utiliser plusieurs arbres et envoyer leurs résultats aux files selon, non pas leur chemin dans l'arbre, mais selon la valeur des métriques de ces noeuds en essayant d'éviter que des noeuds dont les métriques sont assez semblables ne se rencontrent pas. Finalement, un groupe de files n'ayant pas une structure en arbre permet une meilleure utilisation des extenseurs puisque les noeuds passent moins de temps hors de ces extenseurs.

L'architecture obtenue est illustrée par la figure 3.3 (on notera que les modules *bloc* ne sont pas illustrés pour ne pas surcharger la figure). On y voit que les files

prioritaires peuvent accepter plus de deux noeuds en un cycle d'horloge. Le but de cette modification est de permettre de "mélanger" les résultats des extensions en transmettant à chaque file prioritaire le meilleur noeud issu d'un des arbres, le deuxième noeud d'un second arbre, le troisième d'un autre arbre et, enfin, le moins bon du dernier arbre. Cette façon de procéder permet d'optimiser l'utilisation des files prioritaires. Une façon d'implanter ce tri est de modifier les extenseurs de manière à ce qu'il ressortent toujours le meilleur noeud-fils en premier. On obtient alors un tri partiel mais qui n'impose pas de ralentissement important. Cela impose, évidemment, d'ajouter de la logique aux extenseurs permettant de comparer et d'aiguiller les noeuds étendus selon la valeur de leur métrique. On devra, encore, modifier les extenseurs et files prioritaires pour pouvoir leur indiquer si les données reçues sont valides.

Si on confronte cette architecture aux critères décrits plus haut, on constate les points suivants:

- les communications sont régulières mais un cycle d'horloge doit avoir plus de trois phases pour permettre aux files prioritaires d'accepter plus de deux noeuds par cycle, ce qui ralentit le système,

- la "qualité" des noeuds étendus est légèrement sous-optimale car les noeuds sont étendus sur plus d'une branche avant d'être triés,

- le partage des bons noeuds entre les files prioritaires est bon parce que les noeuds sont envoyés aux files prioritaires selon la valeur de leur métrique,

- elle offre une grande facilité d'implantation d'un algorithme multi-chemins sans retours en arrière: tout comme pour l'architecture linéaire.

Les étapes du décodage d'un bloc de données

On initialise d'abord les différents modules. Cela consiste à initialiser toutes les mémoires de configuration et à forcer les données relatives à la racine de l'arbre de décodage dans une des files prioritaires alors que les autres files prioritaires sont initialisées de façon à signaler qu'elles ne contiennent aucune donnée valide.

On doit ensuite transmettre les symboles reçus aux modules *bloc*. Pour ce faire, on procède de la même manière que dans le cas de l'architecture linéaire, c'est-à-dire que les symboles ayant la même valeur de position dans le bloc modulo la profondeur des arbres d'extenseurs sont transmis au même *bloc*.

Lors de la troisième étape, on doit simplement mettre le décodeur en mode de décodage lorsque le monde extérieur signale la fin du bloc de données. Ensuite, le système passe automatiquement à l'étape suivante lorsqu'un des historiques signale que le dernier noeud reçu est à la fin du bloc. A ce moment, le décodage continue alors que l'historique signale au module *tri* d'intercepter les données concernant les noeuds sortant des files prioritaires (et qui sont à la fin du bloc de données) et envoie à l'extenseur concerné un signal indiquant que les données reçues ne sont pas valides.

Les dernières étapes consistent à comparer les métriques des noeuds interceptés, prendre celui qui a la métrique la plus grande et à retracer ce chemin dans l'arbre à l'aide de l'historique et de ce symbole de départ fourni par le module *tri* comme dans le cas du décodeur minimal. On notera que les adresses servant à retracer les noeuds dans les historiques comprennent un nombre permettant d'accéder les historiques séparément. Egalement, le module *tri* devra recevoir et comparer plus de deux noeuds par cycle d'horloge.

3.3 Détails d'implantation

3.3.1 Le contrôle du décodeur

Les états du système

De la discussion des sections précédentes, on constate qu'il faut six états pour contrôler chacun de ces systèmes:

initialisation Tous les modules sont en attente des données d'initialisation et du signal leur indiquant que ces données les concernent.

lecture Le module *bloc* reçoit les symboles d'un bloc de données et les mémorise alors que les autres modules sont en attente.

décodage Les modules entrent en action en effectuant les opérations pour lesquelles ils ont été conçus sauf le module *tri* qui reste en attente.

décodage suivi Les modules continuent leur travail de décodage, alors que le module *tri* entre en action et mémorise les données sorties de la (des) dernière(s) file(s) prioritaire(s) pour un nombre de cycles égal au nombre de files prioritaires ou à la profondeur de(s) l'arbre(s) d'extenseur(s) (selon l'architecture).

tri Le module *tri* trouve le noeud ayant la métrique la plus élevée parmi les noeuds reçus alors que les autres modules sont en attente.

écriture Le module *historique* envoie le bloc décodé au monde extérieur en se servant de l'adresse donnée par le module *tri* alors que les autres modules sont en attente.

Les descriptions précédentes permettent de voir qu'à chaque état du système il ne correspond pas un état pour chaque module; certains modules ont le même état

système	extenseur	file	tri	historique	bloc
init.	init.	init.	init.	init.	init.
lecture	init.	init.	init.	init.	lecture
décodage	déc.	déc.	init.	déc.	déc.
décodage suivi	déc.	déc.	acc.	déc.	déc.
tri	init.	init.	tri	init.	init.
écriture	init.	init.	init.	écriture	init.

Tableau 3.1: Etats du système

module	état
extenseur	init. déc.
file prio.	init. déc.
tri	init. acc. tri
historique	init. déc. écriture
bloc	init. lecture déc.

Tableau 3.2: Etats nécessaires aux modules

pour différents états du système. Le tableau 3.1 établit la relation entre ces états et le tableau 3.2 donne la liste des états nécessaires pour chaque module.

On remarque que trois des cinq types de bloc nécessitent trois états; on doit donc utiliser deux signaux d'états pour les commander ce qui donne quatre états disponibles, le quatrième état peut alors être utilisé pour implanter un mode de test.

Les réactions de chaque module aux commandes du système se résument comme suit:

extenseur

init. Si le contrôleur s'adresse à l'extenseur, celui-ci active la chaîne de balayage utilisée pour le configurer.

déc. Si les données reçues sont valides, il fait l'extension de ce noeud et en transmet les résultats, sinon il signale que les données sont invalides.

file prioritaire

init. Le cas échéant, elle lit les données sur le bus et les mémorise.

déc. Elle écrit le noeud situé à son sommet sur le bus et, ensuite, lit deux (ou plusieurs) autres noeuds.

tri

init. S'il est actif, ce module lit sur le bus un nombre indiquant le nombre de paires extenseur-file prioritaire (ou la profondeur des arbres d'extenseurs) contenues dans le système et remet son compteur d'adresse à zéro.

acc. Il lit un noeud sur le bus et incrémente son compteur d'adresse.

tri Il tri les noeuds qu'il contient et retient celui ayant la plus grande métrique.

historique

init. Il met à zéro son compteur d'adresse.

déc. Il lit les données sur le bus et incrémente son compteur d'adresse.

écriture Il envoie au monde extérieur le bloc décodé en se servant du noeud retenu par le module *tri*.

bloc

init. Si le contrôleur s'adresse à ce module, il met à zéro son compteur d'adresse.

lecture Il lit deux symboles sur le bus et incrémente son compteur d'adresse.

déc. Il lit sur le bus un nombre indiquant une profondeur dans l'arbre et retourne les symboles correspondants.

On constate que, dans les tableaux qui précèdent, l'état *décodage avant* de l'extenseur a été complètement escamoté; cela est dû aux résultats de certaines simulations qui ont montré que, lorsqu'on ajuste correctement la profondeur de l'historique en fonction de celle de la (des) file(s) prioritaire(s), on peut faire en sorte que les blocs qui provoquent un débordement de l'historique sont les blocs pour lesquels on a de toute façon perdu le bon chemin. Donc, ne pas faire de décodage avant mais demander une retransmission (lorsque l'historique déborde) et bien ajuster les dimensions de la (des) file(s) prioritaire(s) et de l'historique permet d'avoir un taux d'erreurs, en pratique, nul (surtout si on utilise une grande longueur de contrainte). Plus de détails sur les simulations citées sont donnés au chapitre 5.

Les états des trois systèmes sont les mêmes puisque la seule différence architecturale significative est que plusieurs files prioritaires envoient leurs données à plusieurs historiques dans le cas de l'architecture arborescente non-reconvergente. Ce fait n'impose pas de changement aux états du système car le module *tri* aura une structure différente permettant de compenser cet état de fait.

Donc, tout les modules sauf *tri* pourraient être utilisés indifféremment dans chacun des systèmes décrits dans ce mémoire.

Les signaux de contrôle

Le module *contrôleur* doit générer les signaux de contrôle suivants:

MA "adresse" du module auquel sont destinées les données sur le bus; ce signal est utilisé dans l'état d'initialisation et dans l'état de lecture pour choisir le module *bloc* auquel on s'adresse,

PC ligne donnant l'état de la file prioritaire et de l'extenseur car on remarque, d'après le tableau 3.1, qu'ils changent d'état en même temps,

SC 2 lignes donnant l'état du module *tri*,

HC 2 lignes donnant l'état de l'historique,

BC 2 lignes donnant l'état du module *bloc*.

Les signaux générés par les autres modules sont:

BF signal généré par le(s) module(s) *historique(s)* pour indiquer que le noeud sorti de la file prioritaire est à la fin du bloc de données (ce signal est envoyé au module *tri* et au contrôleur),

FF signal généré par le module *tri* indiquant que la lecture des noeuds de fin de bloc est terminée,

SF signal généré par le module *tri* indiquant que le tri des noeuds est terminé (le décodage est fini),

RF commande du monde extérieur indiquant la fin de la lecture du bloc de données,

IF commande du monde extérieur indiquant la fin de l'initialisation,

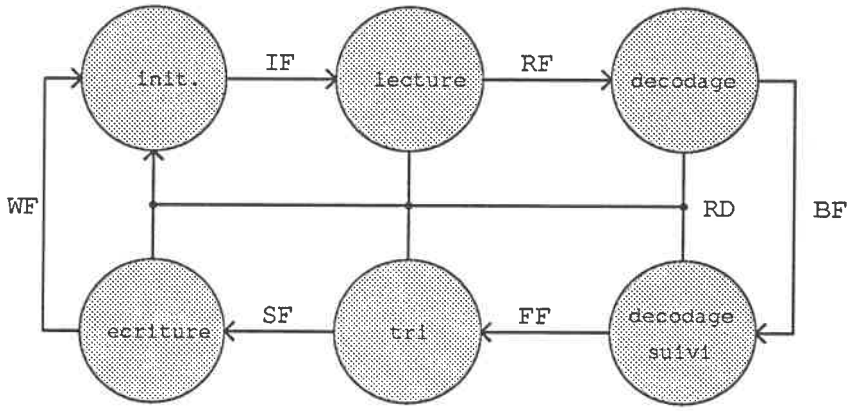


Figure 3.4: Diagramme d'état

WF signal provenant du(des) module(s) historique(s) indiquant si l'écriture est terminée,

RD met le contrôleur en mode d'initialisation, à ce moment, les autres signaux de transitions d'états sont inactifs puisqu'ils sont aussi émis par le monde extérieur ou par un module qui est en mode d'initialisation et, par conséquent, qui force le signal à zéro. De plus, le fait que l'état du système est commandé par le contrôleur, les courses internes au système sont sans conséquence puisque l'initialisation annulera tout résultat néfaste dû à ces courses éventuelles.

Pour une description plus complète des communications inter-module se référer à l'annexe A.

3.3.2 Diagramme d'état

Ayant décrit les états et les signaux de contrôle, nous sommes maintenant prêts à présenter le diagramme de transition d'état qui est illustré à la figure 3.4.

On peut y voir que, quelque soit l'état courant, on entre en état *initialisation* lorsque la ligne RD est vraie. Le passage de cet état à l'état lecture est effectué par le monde extérieur, via le signal IF, lorsqu'il a fini d'initialiser les modules. La transition à l'état *décodage* est commandée par le monde extérieur à l'aide de la ligne RF lorsqu'il a terminé la transmission des données aux modules *blocs*. Par contre, le passage à l'état *décodage suivi* est forcé par le bloc *historique* (via la ligne BF) lorsque le(s) noeud(s) sorti(s) de la(les) file(s) est(sont) à la fin du bloc de données. Le système passe à l'état *tri* lorsque le module *tri* active le signal de contrôle FF; de même, on passe à l'état *écriture* lorsque le module *tri* signale la fin du tri des noeuds à l'aide de la ligne SF. Finalement, la transition *écriture-initialisation* se fait lorsque le signal WF est activé par le module *historique*.

Chapitre 4

Simulateur

Pour évaluer de façon plus précise les capacités des architectures qui ont été décrites, on a conçu un simulateur qui est décrit dans les sections qui suivent; les deux premières sections donnent le format des données et des résultats, respectivement, alors que la troisième décrit sommairement les algorithmes mis en oeuvre par le simulateur.

4.1 Format des données

Les données doivent être stockées dans un fichier, une par ligne (elles peuvent être suivies de commentaires) dans l'ordre suivant:

fichier d'erreurs nom du fichier de sortie où seront écrits les messages d'erreurs advenant un problème lors de la simulation,

fichier de résultats nom du fichier où seront sauvegardés les résultats de la simulation,

u numérateur du taux de codage,

v dénominateur du taux de codage,

k longueur de contrainte,

nquant nombre de niveaux de quantification,

c1 premier mot du code utilisé,

c2 deuxième mot du code utilisé,

h facteur multiplicateur de métrique,

h_depth profondeur de(s) l'historique(s),

q_depth profondeur de la (des) file(s) prioritaire(s),

lin_len ou arbo_depth nombre de paires extenseur-file prioritaire ou profondeur du(des) arbre(s) d'extenseurs,

n_blocs nombre de blocs de données,

n_bits nombre de bits par bloc de données (excluant la queue du message),

métrique initiale métrique assignée à la racine de l'arbre de décodage,

semence message semence initiale du générateur de nombres pseudo-aléatoires utilisé pour générer les blocs de message,

semence canal semence initiale du générateur de bruit du canal de transmission,

Eb_No? les rapports signal/bruit qui suivent sont-ils donnés en E_b/N_0 ou E_n/N_0 (1 ou 0)?,

s/n initial premier rapport s/n (en dB) pour lequel on veut effectuer la simulation,

s/n final dernier rapport s/n (en dB) pour lequel on veut effectuer la simulation,

n. de s/n nombre de rapports s/n pour lesquels on veut effectuer la simulation,

seq. de zero? utilise-t-on des séquences de zéros ou des messages aléatoires? (1 ou 0),

ppram[1] premier patron de deux bits indiquant si on doit perforer un, deux ou aucun des deux symboles,

⋮

ppram[u] dernier patron de la matrice de perforation (voir [17]) (on doit en avoir un nombre égal au numérateur du taux de codage "**u**").

Un exemple de fichier est donné au tableau 4.1. La mise en marche du simulateur consiste à l'appeler (le nom du programme est "decodeur") et à lui donner en paramètre le nom complet du fichier de données approprié. Si on ne lui fournit pas de nom de fichier, il tentera de lire le fichier "decodeur.dat".

4.2 Format des résultats

Le fichier de résultats répète les données suivantes: k, u, v, nquant, c1, c2, lin_len ou arbo_depth, h_depth, q_depth, n_blocs, n_bits, semence canal, semence message, seq. de zéros?. Il donne ensuite les résultats proprement dits des simulations qui sont, pour chaque rapport signal/bruit:

Eb/No rapport s/n (en dB),

Rcomp comme son nom l'indique R_{comp} (computationnal cut-off rate),

```
decoerrlin.dat fichier d'erreur
decoutlin.dat  fichier de resultats
1           u
2           v
24          k
8           nquant
55076157    c1 (octal)
75501351    c2 (octal)
8           h
10000       h_depth
50          q_depth
1           lin_len ou arbo_depth
20          nblocs
505         nbits par bloc (sans la queue)
16000       metrique initiale
32767       seed_message
731         seed_canal
1           snrb ( Eb/No ?)
2.698       rapport s/n initial
3.634       rapport s/n final   (doit etre > initial)
4           n. de rapport s/n   (doit etre positif)
0           seq_zero
0           ppram[1] IL DOIT Y EN AVOIR u
```

Tableau 4.1: Données d'entrées fournies au simulateur

cap. capacité du canal,

métriques p[trans.] métriques, probabilités de transitions du canal et cumulatives des probabilités de transitions,

n. déb. nombre de blocs ayant débordés,

semence finale du canal semence finale du générateur de nombres aléatoires de bruit du canal,

n_itera nombre moyen de cycles pour décoder un bloc,

variance variance de ce nombre de cycles,

n_erreur nombre total de bits en erreur,

B.E.R. taux d'erreurs,

transitions exp. transitions expérimentales du canal,

statistiques tableau donnant, pour chaque nombre de cycles pertinent (ou intervalle), le nombre de blocs ayant nécessité ce nombre de cycles pour être décodés, la cumulative de l'effort de calcul, le nombre de bits en erreur, le taux d'erreurs et la moyenne du nombre de calculs (si l'historique est limité à ce nombre de cycles) pour les blocs déjà donnés.

Les statistiques mentionnées ci-haut peuvent être imprimées sous forme tabulée ou pour chaque nombre de cycles pour lequel il y a au moins un bloc; ce choix est inclus dans le code du simulateur et un changement nécessite donc une recompilation. Le nombre d'intervalles est aussi inclus dans le code et est présentement fixé à 50.

Un exemple de fichier de sortie est donné à l'annexe F.

4.3 Algorithmes du simulateur

Les sections suivantes décrivent de façon très intuitive les principales routines utilisées dans le simulateur, une description plus systématique (sous forme d'algorithme) est donnée à l'annexe C.

4.3.1 Routine principale

Le travail de cette routine consiste à lire les données, générer, coder et bruiter les messages, lancer le décodage, recueillir les résultats et compiler les statistiques. Elle procède selon les étapes suivantes:

lecture et validation des données (routine "lire_donnees")

pour chaque rapport signal/bruit ...

initialisation du canal (routine "init_canal")

initialisation des statistiques

pour chaque bloc de données ...

génération du message

codage du message

bruitage du message (routine "generer_sequence_bruitee")

décodage du message (routine "decodeur")

si le bloc n'a pas fait déborder l'historique, comptage du nombre de ... bits en erreur

ajustement des statistiques

génération des résultats

4.3.2 Routine "decodeur"

Cette routine reçoit un bloc de données codées et bruitées, en effectue le décodage et retourne le bloc décodé ainsi que le nombre de cycles d'horloge nécessaire au décodage du bloc. Elle signale un débordement de l'historique en incrémentant un compteur de débordements reçu en paramètre. Elle effectue ce travail de la manière suivante:

initialisation des files prioritaires (routine "q_init")

initialisation de(s) l'historique(s) (routine "init_hist")

initialisation des variables qui contiendront les noeuds sortis des

... files prioritaires

boucler ...

pour chaque file prioritaire ...

mise en oeuvre de la fonction des files (entrée, tri et sortie) (routine ... "queue")

pour chaque extenseur ...

simulation du module bloc (i.e. lecture des symboles et ajustement ... du bit indiquant si les symboles sont dans la queue du message)

extension du noeud reçu (routine "extenseur")

incrémentation du nombre de cycles

pour chaque historique (le cas échéant) ...

si le noeud reçu est valide, enregistrement dans l'historique

si le noeud est à la fin du bloc de données

mémorisation de son adresse d'historique

sortie de la boucle

si l'historique déborde

incrémentation du compteur de débordement

signalisation du débordement

sortie de la boucle

s'il n'y a pas eu de débordement, extraction du bloc de l'historique

retour à la routine appelante en donnant le nombre de cycles de décodage

4.3.3 Routine "extenseur"

Cette routine reçoit les données concernant un noeud et en fait l'extension en ses deux noeuds-fils. Elle calcule la métrique des noeuds-fils, leur adresse de perforation et elle effectue, le cas échéant, la perforation des symboles. L'algorithme qui y est implanté est le suivant:

copie du noeud reçu dans les noeuds-fils

si le noeud-père n'est pas valide, retour à la routine appelante

calcul de la profondeur des noeuds-fils dans l'arbre de décodage

calcul de l'état des noeud-fils

codage de l'état des noeud-fils

calcul des métriques de branches

perforation, si nécessaire, de ces métriques

calcul des métriques des noeuds-fils

si on est dans la queue du message, mise à zéro de la métrique du noeud 1

ajustement du compteur de perforation

4.3.4 Routines des files prioritaires

Il y a plusieurs routines pour gérer les files prioritaires dans le but de faciliter cette gestion selon les différentes architectures et selon les différents états (initialisation et décodage normal). La routine "q_init" simule l'état d'initialisation, la routine "queue" effectue la gestion des noeuds à insérer dans les files prioritaires alors que "insere_noeud" et "enleve_noeud" effectuent respectivement l'insertion et le retrait des noeuds dans les files.

"q_init"

pour chaque file ...

pour chaque noeud de la file autre que la tête et la queue ...

ajuster les pointeurs aux noeuds précédent et suivant

indiquer que le noeud est invalide

pour la tête de la file ...

indiquer qu'il n'y a pas de noeud précédent

ajuster le pointeur au noeud suivant

indiquer que le noeud est invalide

pour la queue de la file ...

indiquer qu'il n'y a pas de noeud suivant

ajuster le pointeur au noeud précédent

indiquer que le noeud est invalide

ajuster l'entête en lui assignant les adresses de la tête et de la queue de la file

retourner à la routine appelante un pointeur à la première entête de file

"queue"

trier les noeuds en ordre décroissant de métrique

trouver le nombre de noeuds valides

insérer les noeuds dans la file à l'aide de la routine "insere_noeud"

retirer le noeud qui est au sommet de la file (routine "enleve_noeud")

"insere_noeud"

trouver le nombre de noeuds qui seront insérés dans la file (i.e. qui ont

... une métrique plus élevée que les derniers noeuds de la file)

trouver l'endroit dans la file où devra être inséré le premier noeud

pour chaque noeud ...

trouver l'endroit où il doit être inséré

briser les liens du dernier noeud de la file (il "tombe" de la file)

recréer ses liens au bon endroit dans la file

copier dans cet élément de file les données concernant le noeud reçu

"enleve_noeud"

mémoriser l'adresse du noeud du sommet de la file

copier ses données à l'adresse du paramètre de résultat

ajuster les pointeurs impliqués (sommet de file, queue de file, précédent et suivant du noeud déplacé, etc.)

4.3.5 Routines de l'historique

Dans le but de faciliter la gestion de l'historique, sa fonction a été divisée en trois routine: "init_hist", "push_hist" et "pop_hist". Leur fonction consiste, respectivement, à initialiser l'historique, à insérer un enregistrement dans l'historique et à lire un enregistrement de l'historique.

"init_hist"

pour chaque historique (le cas échéant) ...

initialiser le pointeur d'enregistrement courant

"push_hist"

incrémenter le pointeur d'enregistrement courant

insérer à cet endroit la donnée et l'adresse reçues

retourner le contrôle à la routine appelante en lui donnant la valeur du

... pointeur d'enregistrement courant

"pop_hist"

mémoriser l'adresse reçue

retourner les données trouvées à l'adresse reçue

4.3.6 Routine "init_canal"

calculer les probabilités de transitions du canal

calculer les probabilités cumulatives de transitions du canal

calculer les métrique associées

calculer R_{comp}

calculer la capacité du canal

4.3.7 Routine "generer_sequence_bruitee"

pour chaque bit du message ...

générer un nombre pseudo-aléatoire entre zéro et un

à partir de la transition dont la probabilité est la plus grande, trouver celle dont la probabilité est la plus grande tout en étant inférieure au nombre aléatoire

bruiter le bit d'une valeur correspondant à cette transition

Chapitre 5

Résultats des simulations

Les sections qui suivent présentent les résultats des simulations pour chacune des architectures proposées. Les simulations ont toutes été faites avec un code de Johannesson [18] de longueur de contrainte 24 (vecteurs générateurs: 55076157, 75501351), de taux 1/2 alors que les blocs ont une longueur d'environ 500 bits et qui varie selon les architectures pour avoir une longueur qui soit un multiple de la longueur du chemin parcouru entre deux enregistrements dans l'historique. Ceci est nécessaire pour que les chemins qui arrivent à la fin du bloc ressortent immédiatement de leur file prioritaire et soit détectés par l'historique comme indicateur de fin de décodage. On a choisi de simuler un canal ayant des rapports signal sur bruit (E_b/N_0) de 2.698, 3.010, 3.322 et 3.634 dB; ceux-ci correspondent à des valeurs de R/R_{comp} de 0.99, 0.94, 0.89 et 0.85 environ. Dans le texte, on présente les courbes obtenues lorsque E_b/N_0 vaut 2.698 dB alors que les autres courbes sont données à l'annexe D.

Les choix qui ont ainsi été faits visaient à avoir des conditions aussi près que possible des conditions dans lesquelles ces systèmes pourraient être appelés à fonctionner. D'autres conditions seraient évidemment intéressantes à explorer mais

n'ont pu l'être dû au manque de temps.

5.1 Architecture linéaire

Etant donné qu'une absence de variabilité de l'effort de calcul impliquerait une courbe de cumulative de l'effort de calcul (en fonction du nombre de **cycles d'horloge** pour décoder les blocs) en forme d'échelon, les premières simulations effectuées visaient à déterminer si une augmentation arbitraire du nombre de processeurs entraînait toujours une amélioration de cette courbe. Les résultats, pour un nombre de processeurs allant jusqu'à seize, sont montrés à la figure 5.1. On y constate qu'avoir un seul processeur donne sensiblement la même courbe que l'algorithme de Zigangirov-Jelinek, ce qui était prévisible, et qu'augmenter le nombre de processeurs est toujours (dans les limites explorées) profitable. Cette conclusion est corroborée par les données du tableau 5.1 qui montrent que le nombre moyen de cycles pour décoder un bloc diminue constamment à mesure qu'on augmente le nombre de processeurs. On peut également voir que cette architecture se comporte comme l'algorithme multi-chemins puisque, comme décrit dans [16], le fait d'ajouter des processeurs (donc d'augmenter le nombre de chemins) ne fait que déplacer la courbe et ne change pas la pente asymptotique (comportement Pareto) de la courbe.

Une question qui se pose maintenant est de savoir si, malgré cette amélioration, l'efficacité d'utilisation des processeurs diminue c'est-à-dire que cette amélioration pourrait être inférieure à l'augmentation du nombre de processeurs. Pour répondre à cette question, il suffit de tracer les courbes précédentes en multipliant le nombre de cycles par le nombre de processeurs pour obtenir le nombre total d'extensions effectuées. Le résultat est illustré par la figure 5.2. On voit que, lorsqu'on a suffisamment de blocs difficiles à décoder, les courbes rejoignent la même asymptote;

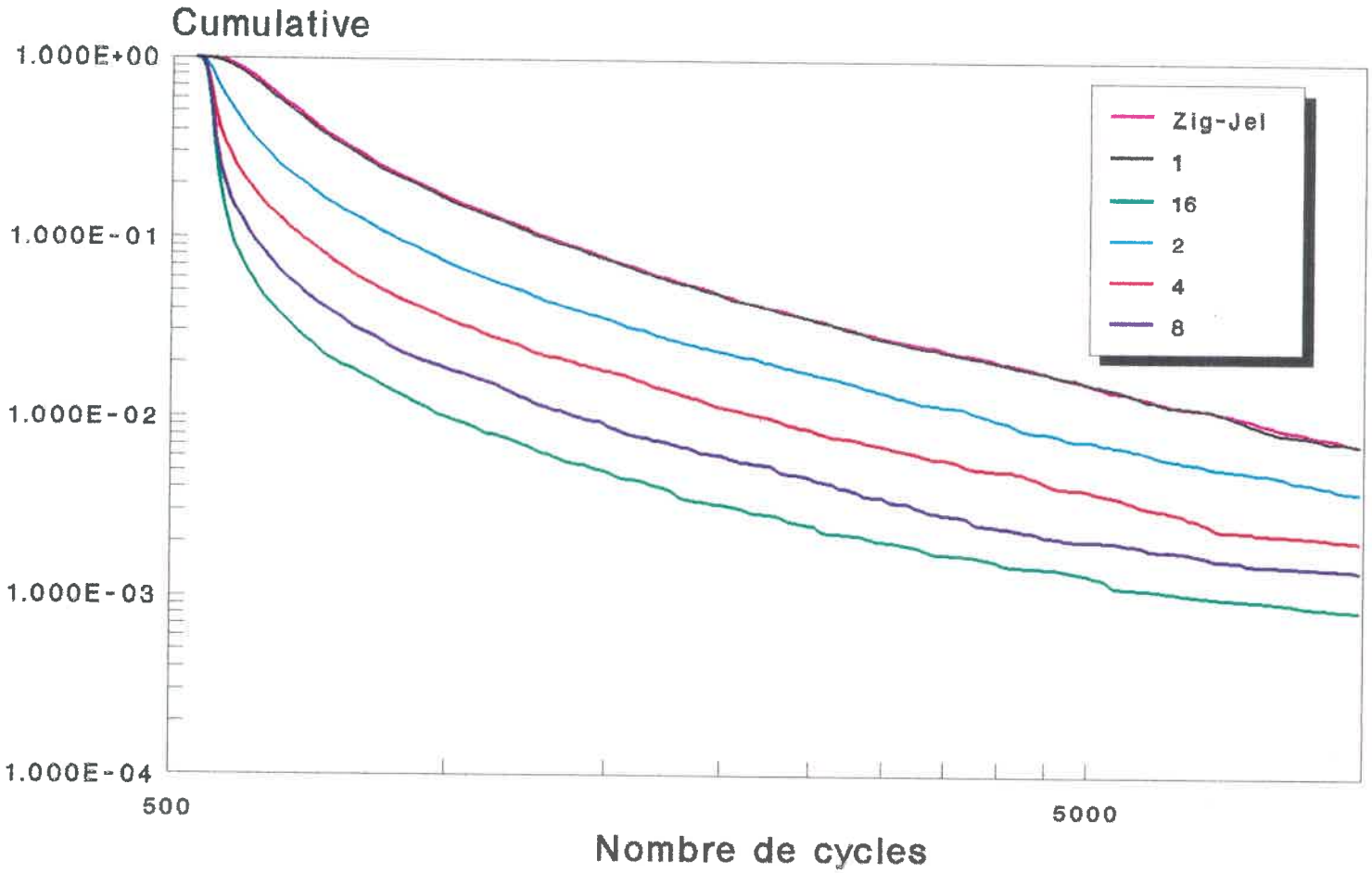


Figure 5.1: Cumulative de l'effort de calcul selon le nombre de cycles

Architecture linéaire
$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

n. de proc.	n. moyen d'itérations
1	950.7
2	744.2
4	647.1
8	602.2
16	580.9

Tableau 5.1: Nombre moyen d'itérations pour différents nombre de processeurs

architecture linéaire
$E_b/N_0 = 2.698dB$

donc, sous cette condition, les extenseurs ne semblent pas faire d'extensions inutiles et la rapidité du système (pour ces blocs difficiles) est proportionnelle au nombre de processeurs, par conséquent, si on fait abstraction des coûts qui augmentent, avoir autant de processeurs que possible est désirable.

Si le travail des extenseurs devient quasi optimal pour les blocs difficiles à décoder, il est naturel de se demander ce qu'il en est de l'utilisation des files prioritaires. C'est-à-dire qu'en ayant plusieurs files plutôt qu'une seule de dimension égale à la somme de celle des autres, on s'attend à ce que, les noeuds n'étant pas parfaitement partagés entre les files, certaines reçoivent plus de bons noeuds et, par conséquent, la probabilité de perdre le bon chemin augmente. Pour savoir à quel taux est perdu le bon chemin, on n'a qu'à simuler un système ayant des files peu profondes par rapport à la profondeur de l'historique. On constate alors, sur la courbe de la cumulative de l'effort de calcul, la présence d'un plateau signifiant que, dans un certain intervalle du nombre de cycles, aucun bloc n'a été décodé en ces nombres de cycles de décodage. De plus, les blocs décodés avec erreurs sont *tous* situés à droite de ce plateau et ceux décodés sans erreur sont à sa gauche; donc, la

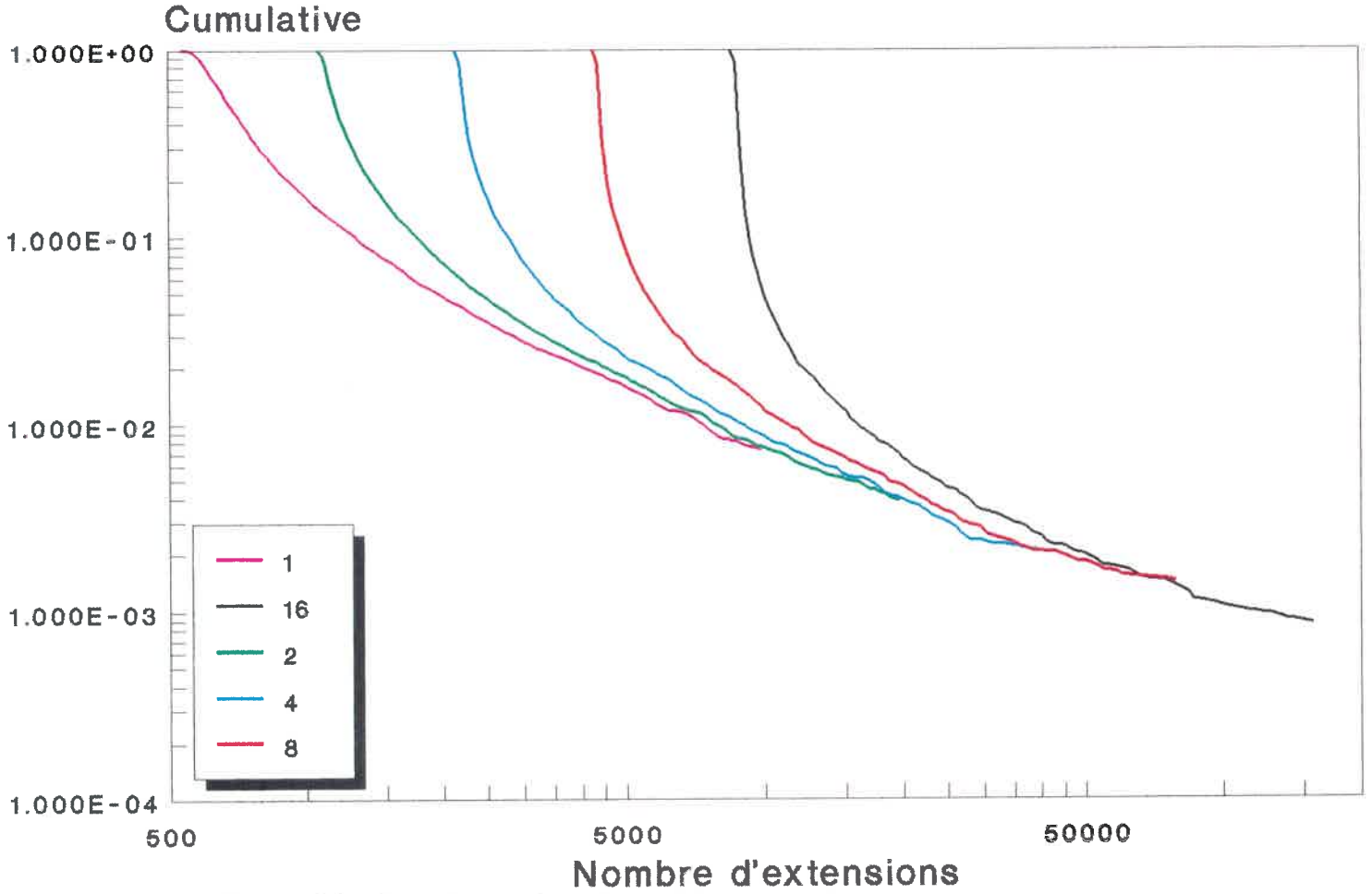


Figure 5.2: Cumulative de l'effort de calcul selon le nombre d'extensions

Architecture linéaire
$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

n. de proc.	prof. des files	hauteur du plateau (%)	B.E.R.
1	100	0.072500	0.0173
2	55	0.073000	0.0171
4	27	0.075500	0.0178
8	14	0.076500	0.0187
16	9	0.074500	0.0182

Tableau 5.2: Taux d'erreurs pour des files de profondeur limitée

architecture linéaire
$E_b/N_0 = 2.698dB$

hauteur du plateau indique le pourcentage de blocs pour lesquels on a perdu le bon chemin. On en déduit qu'ajuster la hauteur du plateau à une même valeur pour un nombre variable de processeurs donnera un même taux de perte du bon chemin et, par conséquent, une même efficacité des files prioritaires. La figure 5.3 donne les résultats de simulations faites pour un nombre de processeur allant de 1 à 16 et pour une profondeur de files allant de 100 à 9. Les taux d'erreurs expérimentaux qu'on obtient sont donnés au tableau 5.2.

On voit que les taux d'erreurs sont assez près bien que non-identiques; cela est dû au manque de "résolution" c'est-à-dire que changer la profondeur des files de 14 à 15, par exemple, entraîne un changement drastique de la hauteur du plateau donc, on ne peut avoir des systèmes ayant exactement le même comportement lorsque les files ont une profondeur aussi faible. De plus, la valeur relative de ces taux d'erreurs n'est pas significative puisque les plateaux ne sont pas à la même hauteur; donc on ne peut pas tirer de conclusions de l'augmentation apparente de ce taux avec le nombre de processeurs.

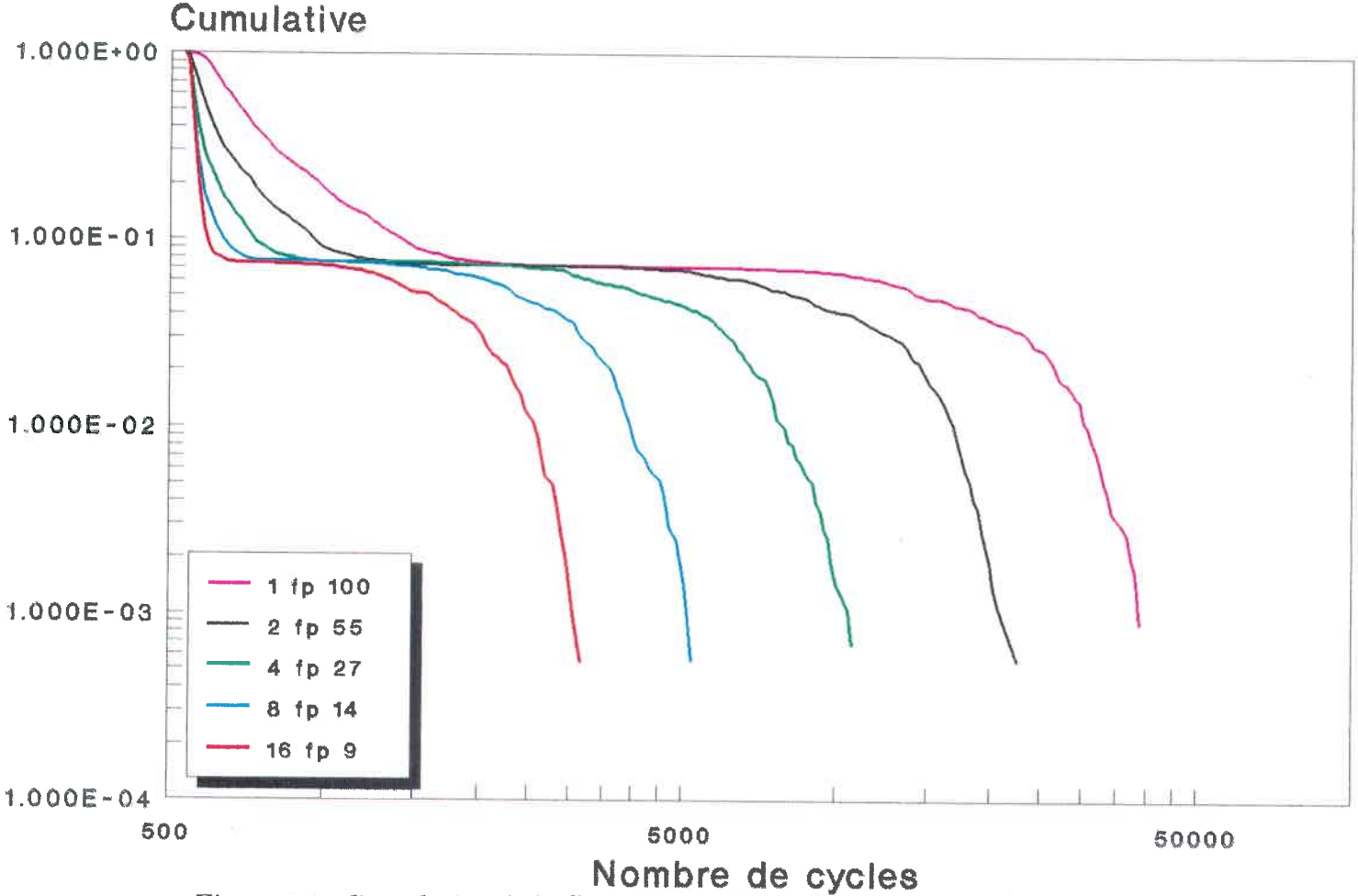


Figure 5.3: Cumulative de l'effort de calcul pour une file de profondeur variée

Architecture linéaire
$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de l'historique = 100000

On constate aussi que, comme prévu, il y a une certaine perte d'efficacité car la profondeur totale des files est de 100, 110, 108, 112 et 144 pour un nombre de processeurs de 1, 2, 4, 8 et 16 respectivement. Donc, la perte d'efficacité est significative mais elle est somme toute assez faible sauf pour le cas de 16 processeurs où elle semble augmenter dramatiquement (n'oublions pas la faiblesse de la résolution!).

En résumé, on voit que, plus on a de processeurs, plus la variabilité de l'effort de calcul est réduite, que le travail des extenseurs reste efficace lorsque les blocs sont, en moyenne, assez difficiles à décoder et que la perte d'efficacité des files prioritaires est telle que la somme de la profondeur des files n'augmente que légèrement en fonction du nombre de processeurs (sauf, éventuellement, pour un très grand nombre de processeurs). La décision quant au nombre optimal de processeurs à utiliser sera donc dépendante des circonstances: le taux de débordement de l'historique désiré, le nombre moyen désiré de cycles pour décoder un bloc, les ressources mises en oeuvre, les limites de volume et de poids admissibles etc.

Pour prendre cette décision, on a donc besoin de déterminer les coûts en fonction du taux de débordement. Pour cela, il faut savoir la profondeur des files prioritaires et de l'historique, le nombre de processeurs et autres paramètres du système. On a aussi besoin d'un (ou plusieurs) modèle(s) de coûts.

Deux modèles de coûts ont été définis ayant chacun deux variantes: le premier consiste à évaluer les coûts selon le nombre de transistors que contient le système et le deuxième tient plutôt compte du nombre de circuits intégrés. Les deux variantes consistent en des systèmes ayant des files prioritaires composées d'une matrice de circuits de file systolique (voir [14]), pour la première, et composées d'une colonne de files systoliques permettant de trier les noeuds et de circuits de mémoire standards permettant de mémoriser les données autres que les métriques, pour la deuxième (les files systoliques ne mémorisent donc que les métriques et des pointeurs à la

mémoire standard). L'évaluation des coûts pour le deuxième modèle est basée sur un système qui est en cours d'implantation à l'Ecole Polytechnique de Montréal alors que le premier modèle ne diffère du deuxième que par les files prioritaires tel qu'expliqué ci-haut.

Les paramètres architecturaux nécessaires à l'évaluation de ces coûts et la valeur qui leur ont été assignée sont:

nombre de transistors par extenseur	7000
nombre de transistors par enregistrement de file (sur un circuit)	857
nombre d'enregistrements par circuit intégré de file	21
nombre de circuits pour avoir des enregistrements de file complets	5
nombre de transistors par bit d'historique	6
nombre de transistors par module *bloc*	27000
nombre de bits par bloc de message	505

Le deuxième des paramètres ci-haut donne le nombre de transistors qui constitue la portion d'un enregistrement de file prioritaire qu'on retrouve sur un circuit intégré, alors que le quatrième paramètre indique combien de circuits de file systolique ont doit concaténer pour allonger les enregistrements de façon suffisante pour les besoins présents.

Le nombre de transistors par extenseur est celui du circuit qui a été conçu, de même pour le nombre de transistors par enregistrement de file systolique, pour le nombre d'enregistrements par file systolique et pour le nombre de transistors (et de circuits) par module *bloc* et historique. Le nombre de circuits par enregistrement de file systolique est tiré de [14] (p. 6). On a fixé le nombre de transistors par bit d'historique à 6 puisque c'est le nombre de transistors que l'on retrouve souvent dans les circuits de mémoire statique comme ceux qui seront utilisés pour l'assemblage de

l'historique du système qui est en cours de réalisation. Le nombre de bits par bloc de message est celui qui a été utilisé dans les simulations qui ont servi à trouver les valeurs optimales de profondeur de files prioritaires et d'historique qui sont, avec le nombre d'extenseurs, les variables dont on a besoin pour calculer les coûts.

Les équations utilisées pour le premier modèle, qui sera par la suite dénommé "A", sont:

$$\text{coûts totaux} = \text{cout ext.} + \text{cout file} + \text{cout histo.} + \text{cout bloc} \quad (1)$$

$$\text{cout ext.} = \text{n. trans. par ext.} * \text{n. proc.} \quad (2)$$

$$\text{cout file} = \text{n. trans. par enre.} * \lceil \text{prof. file/n. enre. par c.i.} \rceil * \text{n. enre. par c.i.} * \text{n. c.i. par enre.} * \text{n. proc.} \quad (3)$$

$$\text{cout histo.} = \text{n. trans. par bit} * \lceil (\text{n. proc.} + \lceil \log_2 \text{prof. histo.} \rceil)/8 \rceil * 8 * 1024 * 2^{min(\lceil \log_2(\text{prof. histo.}/1024) \rceil, 5)} * \lceil \text{prof. histo.}/2^{min(\lceil \log_2(\text{prof. histo.}/1024) \rceil, 5)} \rceil \quad (4)$$

$$\text{cout bloc} = \text{n. trans. par bloc} * \text{n. proc.} \quad (5)$$

où:

cout ext. = coût des extenseurs

cout file = coût des files prioritaires

cout histo. = coût de l'historique

cout bloc = coût des modules *bloc*

n. trans. par ext. = le nombre de transistors par extenseur

n. proc. = le nombre de processeurs (extenseurs ou files prioritaires)

n. trans. par enre. = nombre de transistors par enregistrement de circuit de file prioritaire

prof. file= profondeur des files prioritaires

n. enre. par c.i.= nombre d'enregistrements par circuit de file prioritaire

n. c.i. par enre.= nombre de circuits de file systolique nécessaire pour avoir des enregistrements complets

n. trans. par bit= nombre de transistors par bit d'historique

prof. histo.= profondeur de l'historique

n. trans. par bloc= nombre de transistors par module *bloc*

Les opérations $log_2(\cdots)$, $\lceil \cdots \rceil$ et $min(\cdots)$ visent à trouver le nombre de circuits intégrés nécessaire pour remplir les conditions de profondeur de file et d'historique, c'est-à-dire, pour tenir compte du fait qu'on a des circuits intégrés standards et, donc, qu'on a un nombre déterminé d'enregistrement sur ces circuits. On considère que les circuits de mémoire standards qui sont disponibles ont des capacités qui sont des multiples de 8 kilobits et qui s'échelonnent de cette valeur jusqu'à 256 kilobits (par puissance de deux) et qu'ils ont la capacité de lire ou écrire 8 bits à la fois.

Les équations utilisées pour le deuxième modèle, qui sera par la suite dénommé "C", sont:

$$\text{coûts totaux} = \text{cout ext.} + \text{cout file} + \text{cout histo.} + \text{cout bloc} \quad (6)$$

$$\text{cout ext.} = 2 * \text{n. trans. par ext.} \quad (7)$$

$$\text{cout file} = \text{n. proc.} * \text{n. c.i. par enre.} * \lceil \text{prof. file/n. enre. par c.i.} \rceil \quad (8)$$

$$\text{cout histo.} = 10 + \lceil(\lceil(\log_2 \text{prof. histo.} + \text{n. proc.})/8\rceil * \lceil\text{prof. histo.}/(1024 * 2^{min(\lceil\log_2(\text{prof. histo.}/1024)\rceil,5)})\rceil)\rceil \quad (9)$$

$$\text{cout bloc} = 7 * \text{n. proc.} \quad (10)$$

Les équations régissant les files prioritaires pour la deuxième variante sont, pour le premier et le deuxième modèle respectivement:

$$\text{cout file} = \text{n. proc.} * \text{n. trans. par enre.} * \lceil\text{prof. file/n. enre. par c.i.}\rceil * 8 * \text{n. enre. par c.i.} * 1024 * 2^{min(\lceil\text{prof. file}/1024\rceil,5)} * \text{n. proc.} * 6 \quad (11)$$

$$\text{cout file} = \text{n. proc.} * (\lceil\text{prof. file/n. enre. par c.i.}\rceil + \lceil(34 + \lceil\log_2 \text{n. bits par bloc}\rceil + \lceil\log_2 \text{prof. file}\rceil)/8\rceil * 2^{min(\lceil\log_2(\text{prof. file}/1024)\rceil,5)}) \quad (12)$$

Ces variantes seront dénommées "B" et "D" dans le reste du texte. La description complète du programme et des données qu'il utilise pour calculer ces coûts est donnée à l'annexe E.

Les résultats de ces évaluations de coûts sont montrés aux figures 5.4 à 5.7. Ces coûts ont été calculés avec une profondeur de l'historique correspondant au début du plateau (voir la figure 5.3); on a donc un décodage pratiquement sans erreur et on évite le travail inutile nécessaire pour dépasser le plateau de la courbe de la cumulative de l'effort de calcul. On notera cependant que les points obtenus donnent une borne sur le début du plateau et que cette borne est peu serrée dû au faible nombre de blocs simulés (2000). Ce fait est illustré par le point singulier que l'on peut voir sur les courbes correspondant à un nombre de processeur égal à un.

La première constatation qui se dégage de l'analyse de ces quatre courbes est que, lorsqu'on a un taux de débordement inférieur à 1%, les coûts montent en flèche

(notons que les courbes ont une échelle logarithmique sur l'axe "x", ce qui atténue cette montée abrupte).

Pour ce qui est du nombre optimal de processeurs, on peut voir que quatre processeurs est un choix qui n'est pas déraisonnable étant donné que les coûts correspondants sont très concurentiels (et parfois meilleurs) lorsque comparés aux coûts des architectures à un et deux processeurs et que la variabilité de l'effort de calcul est nettement plus faible. Pour ce qui est de choisir huit processeurs, on serait tenté de croire que les coûts "légèrement" supérieurs pourraient être intéressants considérant que le nombre moyen de cycles nécessaires pour décoder un bloc est plus faible (voir la figure 5.8), cependant, on constate que ce gain est de l'ordre de 5% ce qui est faible. On remarque également sur cette figure que plus le système contient de processeurs et moins l'effort moyen de calcul change en fonction de la probabilité de débordement.

On est également amené à croire *a priori* que seize processeurs est nettement trop (en particulier lorsqu'on s'attarde au nombre de circuits intégrés); une analyse plus détaillée des coûts nous montre, par contre, qu'environ 50% des circuits sont utilisés pour les modules *bloc*. Ceci est dû à la logique de contrôle de ces modules qui doit être répétée. Donc, intégrer ces modules aux extenseurs, permettrait de rendre cette architecture concurentielle. On notera également que cette opération est réalisable sachant que diviser le module *bloc* en seize implique d'ajouter quelques centaines d'éléments de mémoire seulement sur chaque extenseur en plus de quelques dizaines de portes logiques. Etudier davantage cette architecture amène également la constatation que, les files étant très peu profondes (moins de cent entrées, voir l'annexe F pour les valeurs précises), il est envisageable de les mettre sur le même circuit intégré que l'extenseur et le module *bloc*; on aurait alors un nombre de circuits de seize plus ceux nécessaires à la construction de l'historique ce qui est, de

loin, mieux que les autres architectures.

Cela nous montre que les modèles perdent leur précision lorsqu'on atteint le nombre de seize processeurs (et aussi, dans une moindre mesure, pour huit processeurs puisque les files ont alors une profondeur inférieure à cent cinquante enregistrements). On voit donc qu'une architecture à seize extenseurs n'est pas à négliger surtout si on considère que, en combinant une file, un extenseur et un module *bloc* sur un circuit intégré, on obtient un circuit inprimé beaucoup plus simple donc moins dispendieux à produire, à tester et à installer. De plus, cette architecture a une variabilité de l'effort de calcul en fonction du rapport signal à bruit qui est plus faible; les coûts dus aux tampons d'entrées et de sortie pourraient donc également être réduits.

Compte tenu de ces résultats, on peut se demander si avoir une architecture avec encore plus de processeurs serait intéressant. Nous ne croyons pas que ce soit le cas car, les files prioritaires devenant très peu profondes, on s'attend à ce que la perte d'efficacité de ces file devienne suffisante pour entraîner une autre non-linéarité dans les courbes de coûts et pour remettre encore en question les modèles utilisés. Par contre, si on désire un taux de débordement très faible, cette option pourrait être intéressante.

Finalement, on peut voir qu'un nombre de processeurs qui soit de huit ou de seize entraîne un taux de débordement inférieur à 1% pour un historique d'environ 1000 entrées (1500 pour huit processeurs). Donc, le rapport entre le nombre de cycles de décodage d'un bloc très facile à décoder et d'un bloc très difficile est d'environ deux pour un (trois pour huit processeurs) ce qui est intéressant puisque ceci impose une borne assez serrée sur la variabilité de l'effort de calcul.

National Library
of Canada

Canadian Theses Service

Bibliothèque nationale
du Canada

Service des thèses canadiennes

NOTICE

QUALITY OF THIS MICROFICHE
EAVILY DEPENDENT UPON THE
[TY OF THE THESIS SUBMITTED
MICROFILMING.

RTUNATELY THE COLOURED
STRATIONS OF THIS THESIS
ONLY YIELD DIFFERENT TONES
REY.

AVIS

LA QUALITE DE CETTE MICROFICHE DEPEND GRANDEMENT DE LA QUALITE DE LA THESE SOUMISE AU MICROFILMAGE.

MALHEUREUSEMENT, LES DIFFERENTES ILLUSTRATIONS EN COULEURS DE CETTE THESE NE PEUVENT DONNER QUE DES TEINTES DE GRIS.

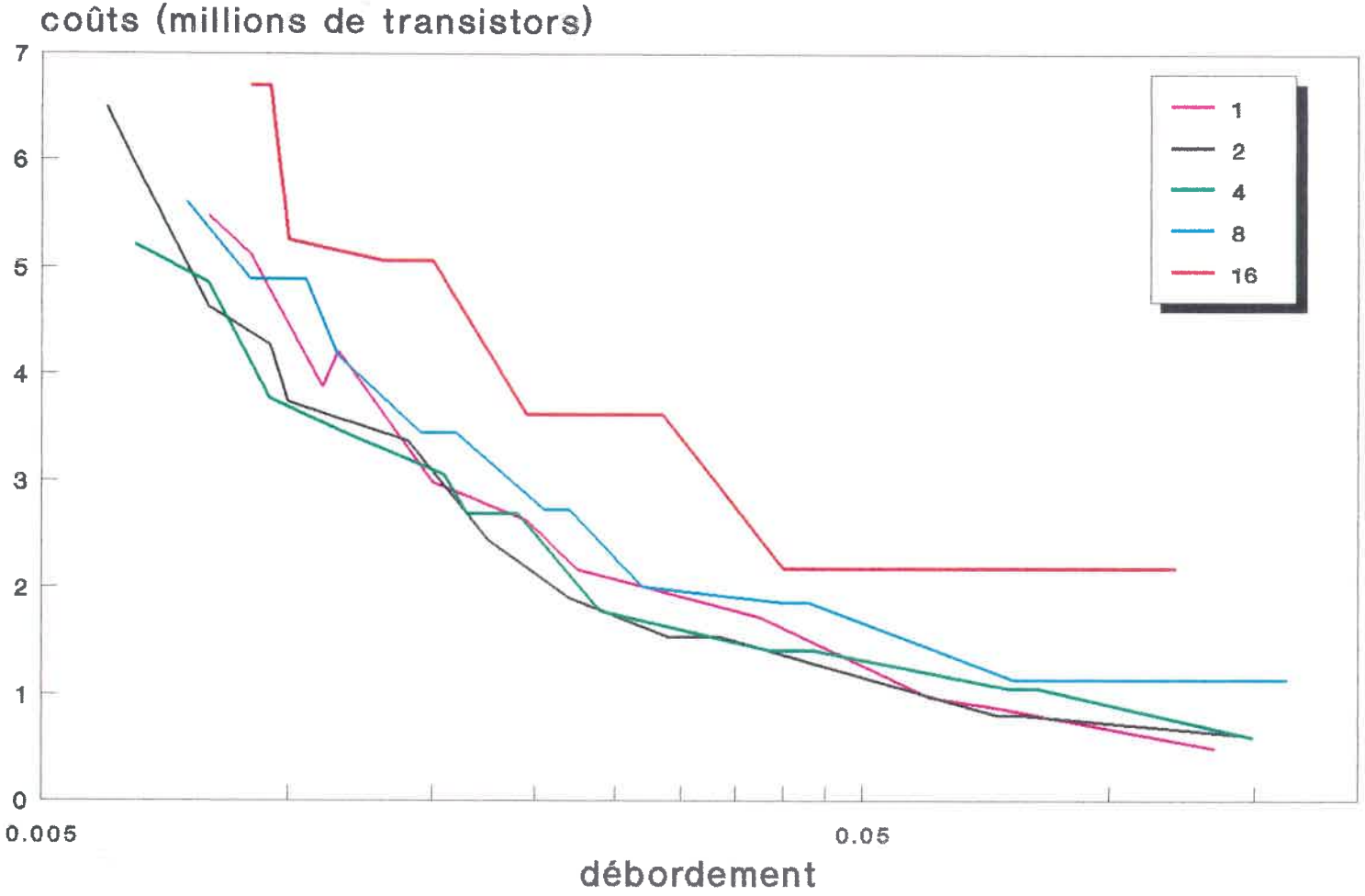


Figure 5.4: Coûts en fonction du taux de débordement selon le modèle "A"

Architecture linéaire
$E_b/N_0 = 2.698dB$

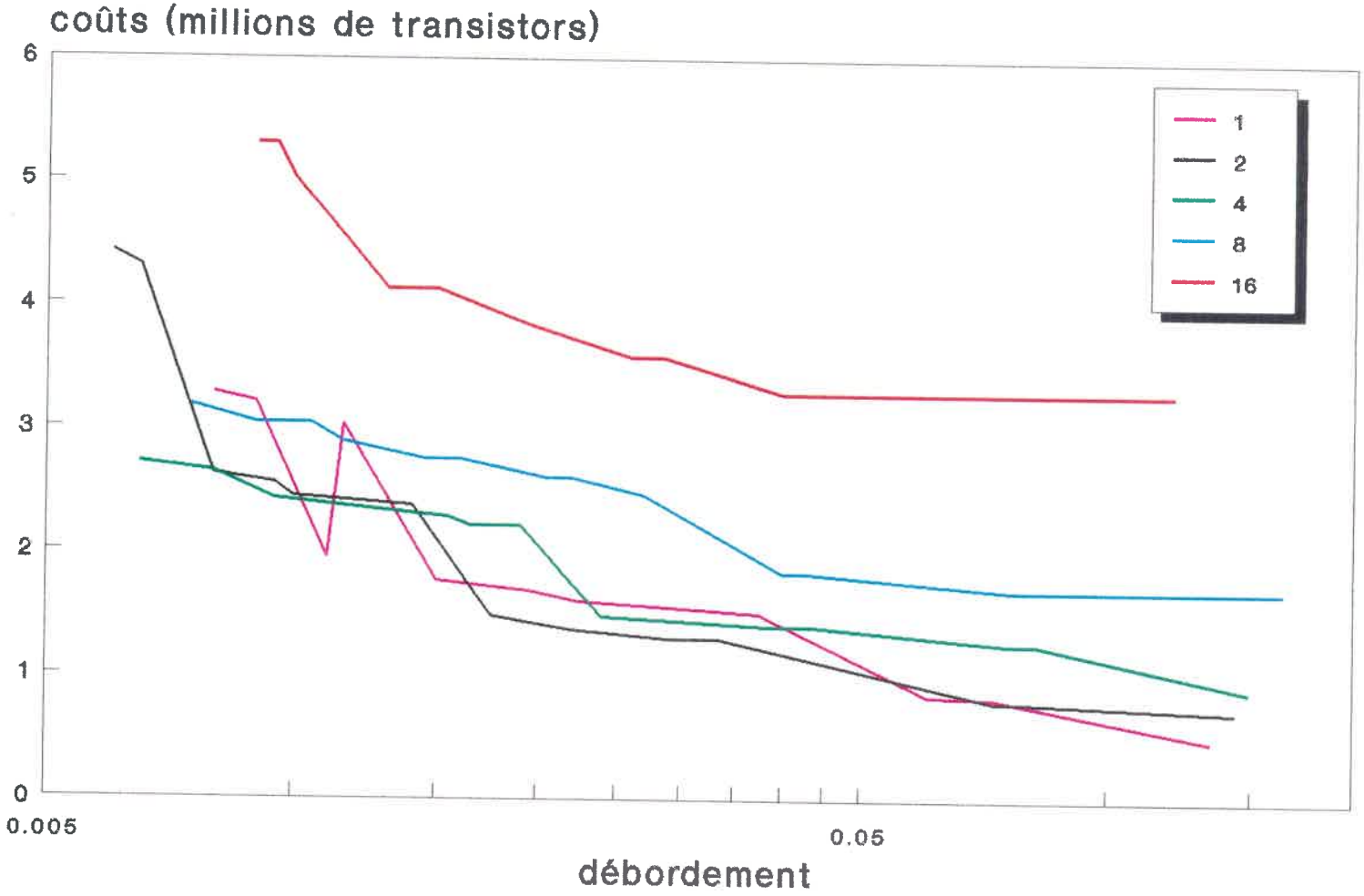


Figure 5.5: Coûts en fonction du taux de débordement selon le modèle "B"

Architecture linéaire
$E_b/N_0 = 2.698dB$

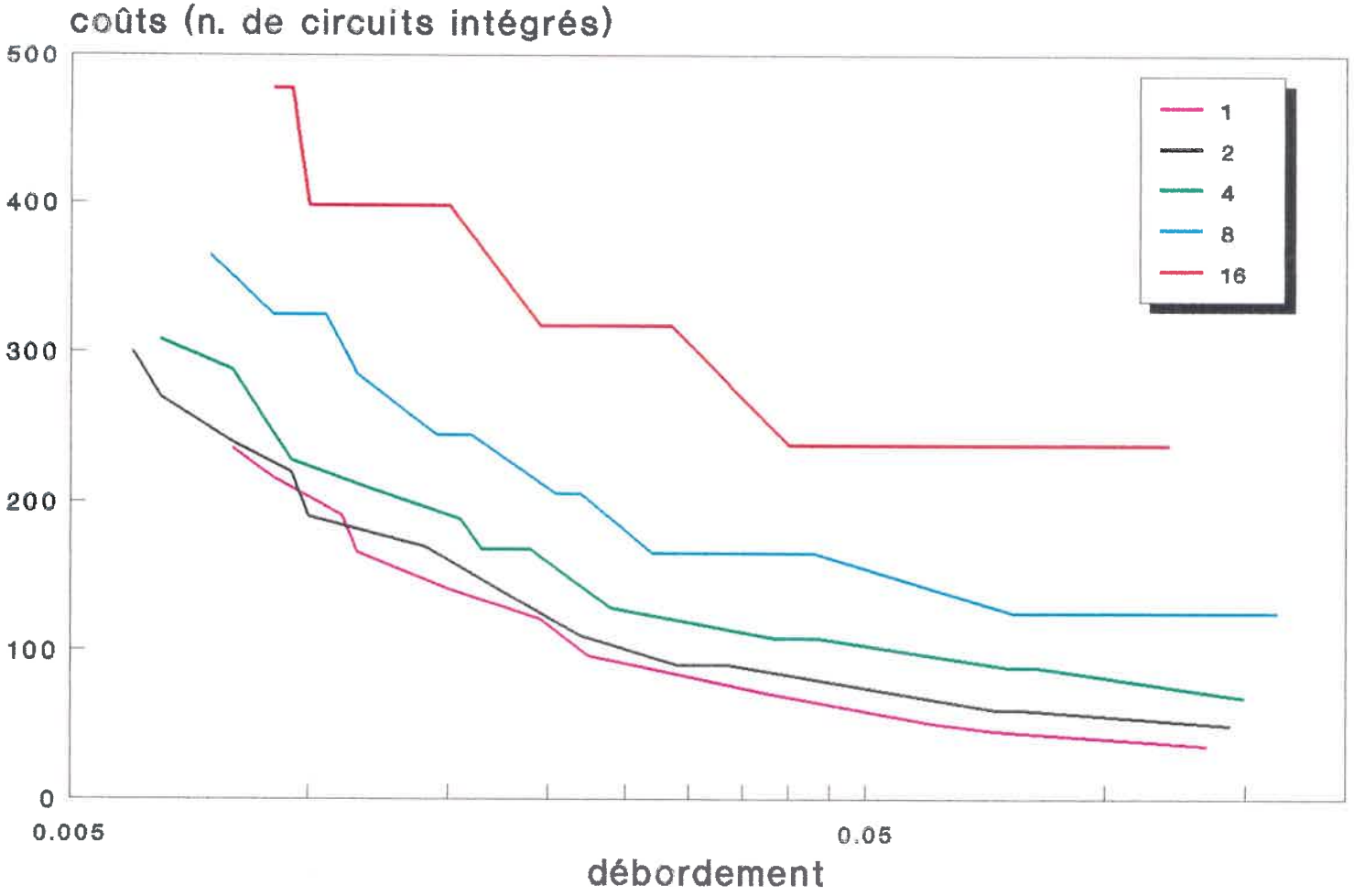


Figure 5.6: Coûts en fonction du taux de débordement selon le modèle "C"

Architecture linéaire
$E_b/N_0 = 2.698dB$

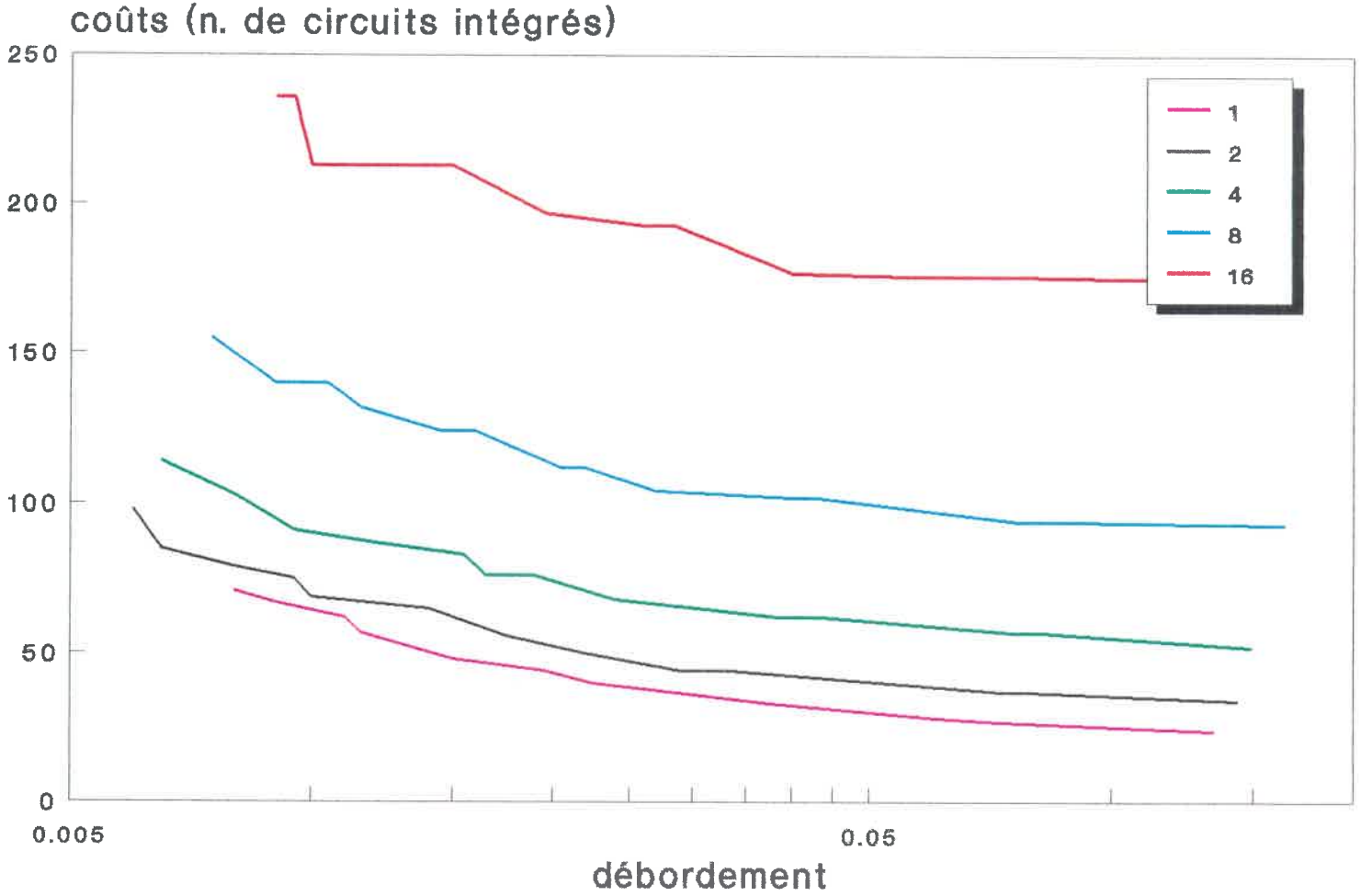


Figure 5.7: Coûts en fonction du taux de débordement selon le modèle "D"

Architecture linéaire
$E_b/N_0 = 2.698dB$

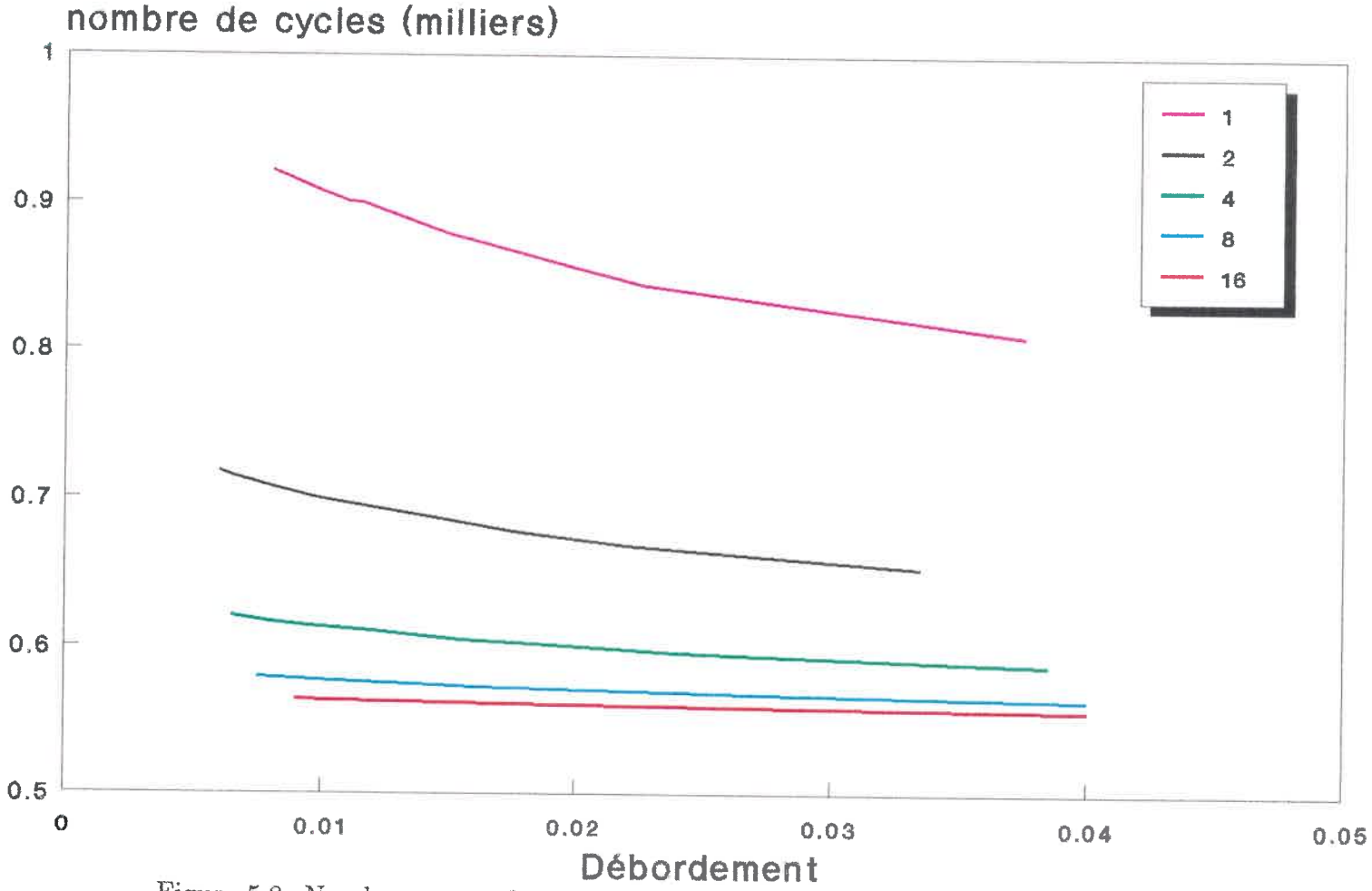


Figure 5.8: Nombre moyen de cycles pour décoder un bloc de données en fonction du taux de débordement

Architecture linéaire
$E_b/N_0 = 2.698dB$

5.2 Architectures arborescentes

Dans le but de s'assurer que l'architecture arborescente reconvergente est aussi peu intéressante que les critères qualitatifs d'évaluation le laissaient croire, une série de simulations ont été effectuées et les résultats sont illustrés par la figure 5.9. La distance qui existe entre le minimum de cycles pour décoder un bloc pour chaque courbe est due au fait qu'il faut $2n - 1$ cycles (où n est la profondeur des arbres) à un chemin pour traverser les deux arbres; donc, le délai que les noeuds subissent dans l'arbre de files prioritaires devient plus important à mesure que la profondeur de cet arbre augmente.

On constate sur ces courbes que, même avec des files prioritaires de mille enregistrements, on a un début de plateau, donc on commence à perdre le bon chemin, alors qu'on n'en a pas pour l'architecture linéaire ce qui pourrait signifier que la file qui se trouve à la racine de l'arbre est bel et bien un goulôt d'étranglement car elle accumule trop de bons noeuds. Cependant, on doit noter que ces simulations ont été effectuées pour deux mille blocs contre vingt mille pour l'architecture linéaire ce qui implique que la position du plateau est définie avec moins de précision. Néanmoins, à la figure 5.10, on voit que l'architecture arborescente de profondeur quatre (qui contient quinze extenseurs et quinze files prioritaires) nécessiterait un historique énorme avant de rattraper éventuellement l'architecture linéaire de longueur seize et qu'elle ne rejoint celle de longueur quatre qu'après environ 3500 cycles. De plus, le système de profondeur six (qui possède 63 extenseurs!) aurait besoin (en l'absence de plateau) d'un historique environ 4500 cycles pour rattraper le système linéaire de longueur seize. On notera également que le gain obtenu en augmentant la profondeur de l'arbre est beaucoup plus faible que pour l'architecture linéaire pour un nombre d'extenseurs qui double pratiquement d'une courbe à l'autre.

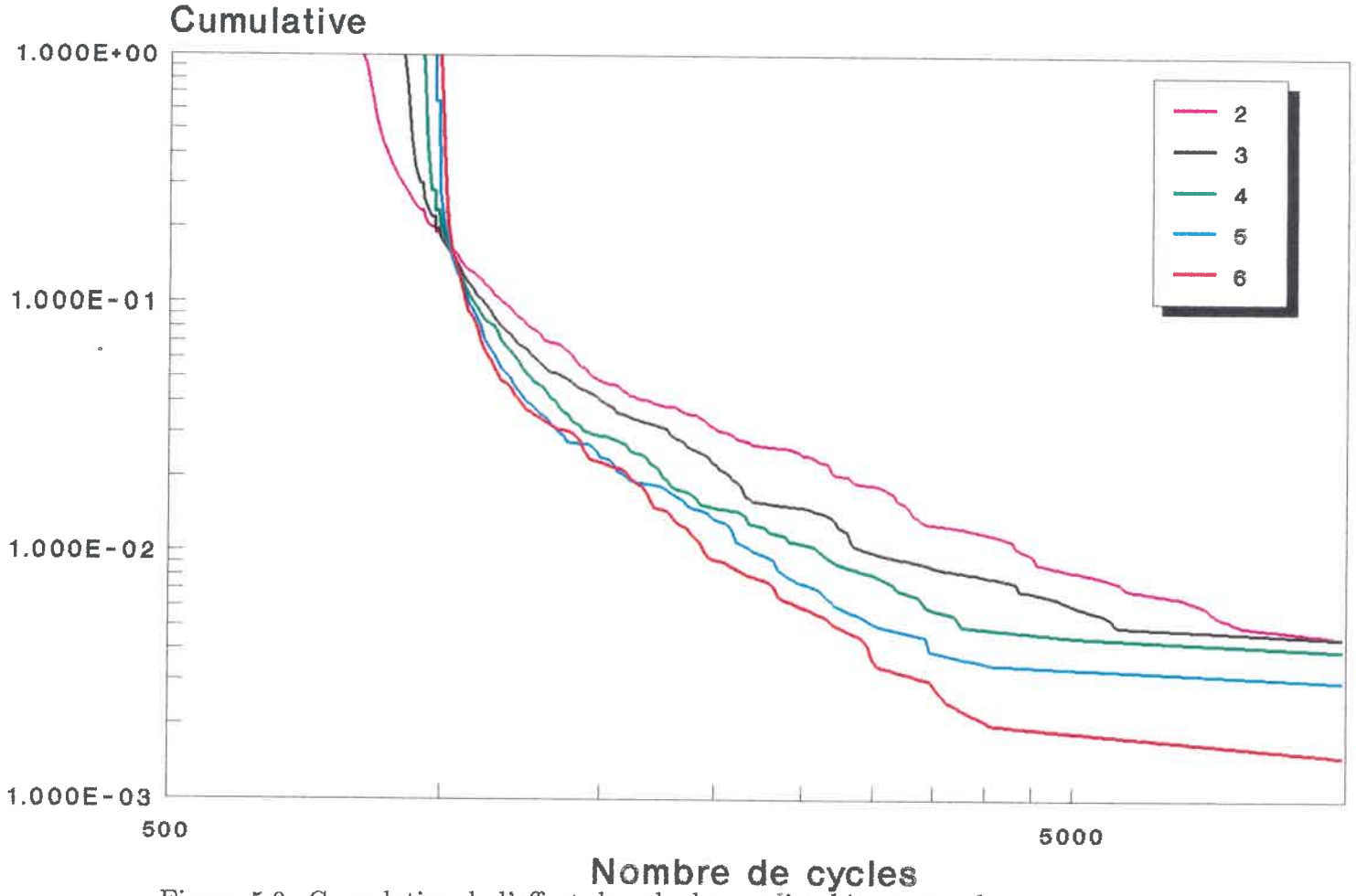


Figure 5.9: Cumulative de l'effort de calcul pour l'architecture arborescente reconvergente

$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

arborescente		
n. de proc.	q_depth	P[déb.]
2	100	0.0555
3	100	0.0465
4	100	0.0395
5	100	0.0275
6	100	0.0260
linéaire		
n. de proc.	q_depth	P[déb.]
1	200	0.0375
2	165	0.0220
4	189	0.0095
8	140	0.0075
16	81	0.0090

Tableau 5.3: Taux d'erreurs pour des files de profondeur constante

$$E_b/N_0 = 2.698$$

Le tableau 5.3 montre que, pour un système arborescent reconvergent ayant des files de profondeur de cent et une profondeur d'arbre augmentant, la probabilité de débordement baisse considérablement moins que dans une situation similaire pour l'architecture linéaire. On en déduit que les files prioritaires sont beaucoup moins bien utilisées. Un système ayant des files prioritaires de profondeur variant avec leur profondeur dans l'arbre pourrait rendre cette architecture plus intéressante. Donc, c'est là une architecture coûteuse (puisque très difficile à intégrer) et qui n'a pas un comportement particulièrement intéressant.

Pour ce qui est de l'architecture arborescente non-reconvergente, on constate, à l'étude de la figure 5.11, que les courbes concernées ont une pente très accentuée qui fait en sorte que la valeur de la cumulative se compare favorablement à celles de

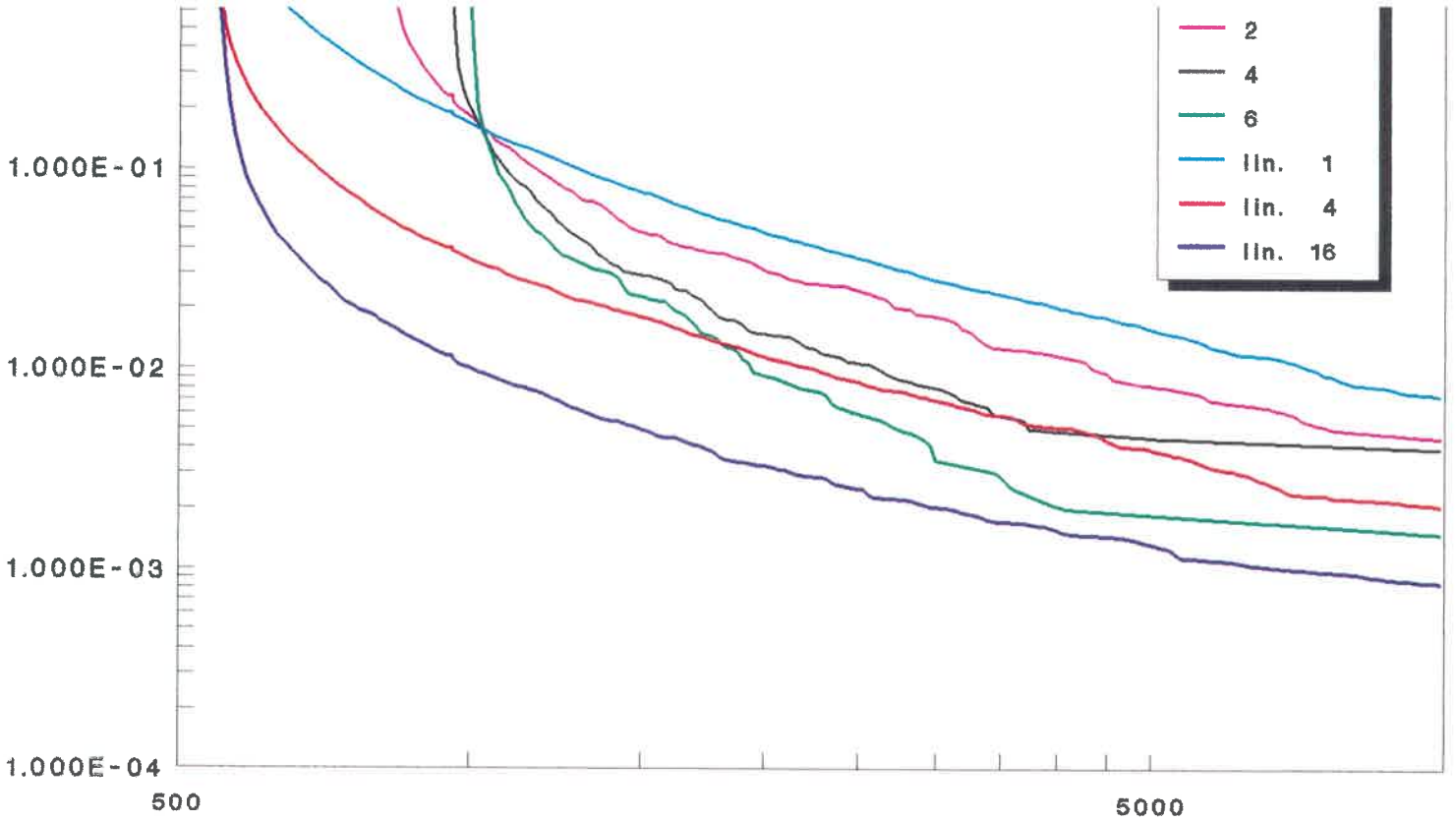


Figure 5.10: Comparaison entre les architectures linéaire et arborescente reconvergente (en terme de la cumulative de l'effort de calcul)

$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
architecture linéaire = 20000
architecture arborescente reconvergente = 2000

l'architecture linéaire car l'architecture contenant des arbres de profondeur 2 (qui possède 12 extenseurs) a une cumulative qui se compare à celle de l'architecture linéaire ayant 16 extenseurs. On notera que les simulations de l'architecture non-reconvergente ont été faites pour seulement 500 blocs; ceci a pour résultat les irrégularités des courbes en question et qu'on ne sait pas exactement où se situent les courbes théoriques.

On notera que cette architecture a, pour les cas simulés, 12 et 56 extenseurs, 4 et 8 files prioritaires et autant d'historiques; il est par conséquent difficile de comparer cette architecture aux deux autres en terme de coûts. Cependant, le fait de ne pas pouvoir intégrer plus d'un extenseur par circuit intégré et la multiplicité des modules *bloc* imposeraient des coûts nettement plus élevés. On notera également que, pour des arbres de profondeur trois, chaque file prioritaire doit accepter 8 noeuds à chaque cycle ce qui ralentit le système. L'aspect intéressant de cette architecture est sa faible variabilité de l'effort de calcul. En effet, pour une profondeur d'arbre de trois, 90% des blocs sont décodés en 539 cycles ou moins et 95% le sont en 554 cyles ou moins. Si on compare ces nombres au minimum de 529 cycles nécessaires au décodage d'un bloc, on conclut que des taux de débordement de 5 et 10% imposent une variabilité **maximale** d'environ 1.9 et 4.7% donc le monde extérieur pourrait ne s'occuper du décodeur qu'après ces nombres de cycles et ainsi ne voir aucune variabilité de l'effort de calcul. Notons finalement que, si on désire un taux de débordement plus raisonnable de 1%, on aura tout de même une variabilité maximale de 38% (730 cycles); on pourrait donc utiliser la tactique décrite ci-haut si une perte de vitesse de 38% était acceptable.

On voit donc que cette architecture mériterait une étude plus approfondie; cependant, dû au manque de temps, cela n'a pas été fait. Cette étude pourrait comprendre une évaluation des coûts (à l'aide d'un nouveau modèle), une étude

de l'efficacité des files prioritaires et on pourrait s'attarder à l'impact qu'aurait, lorsqu'on utilise une profondeur d'arbre de deux, l'utilisation d'une version des extenseurs qui ferait l'extension des noeuds sur deux branches plutôt qu'une seule. On devrait alors s'attarder a l'analyse de l'impact, sur la fréquence de l'horloge, du fait qu'on étend sur deux branches et qu'on doit sortir quatre noeuds des extenseurs plutôt que deux à chaque cycle d'horloge.

5.3 Simulation du module "tri"

Des simulations ont été effectuées pour tenter de mesurer le gain, au niveau du taux d'erreurs, apporté par le module *tri*. Ces simulations ne semblaient pas fonctionner puisque, même pour un faible nombre de blocs à décoder, les simulations bouclaient sans terminer. Après investigation, nous avons constaté que c'était dû au fait que, lors du décodage de la queue du message, les chemins incorrects subissent une forte chute de métrique en fonction de leur avancement dans l'arbre de décodage; donc, on en venait à explorer quasi-exhaustivement les sous-arbres (de profondeur 23) correspondant à la queue du message. Donc, le premier noeud situé à la fin de la queue à sortir de la chaîne est le meilleur noeud. En conséquence, le module *tri* et les modifications à faire aux architectures (états de décodage et structure) ne sont plus de mise. Les modifications à apporter au décodeur minimal sont donc (outre l'ajout de files prioritaires et d'extenseurs):

- le module bloc est divisé en sous-blocs bien que le monde extérieur l'accède de la même manière,

- les données stockées par l'historique contiennent plus d'un bit d'information (symbole décodé) par enregistrement,

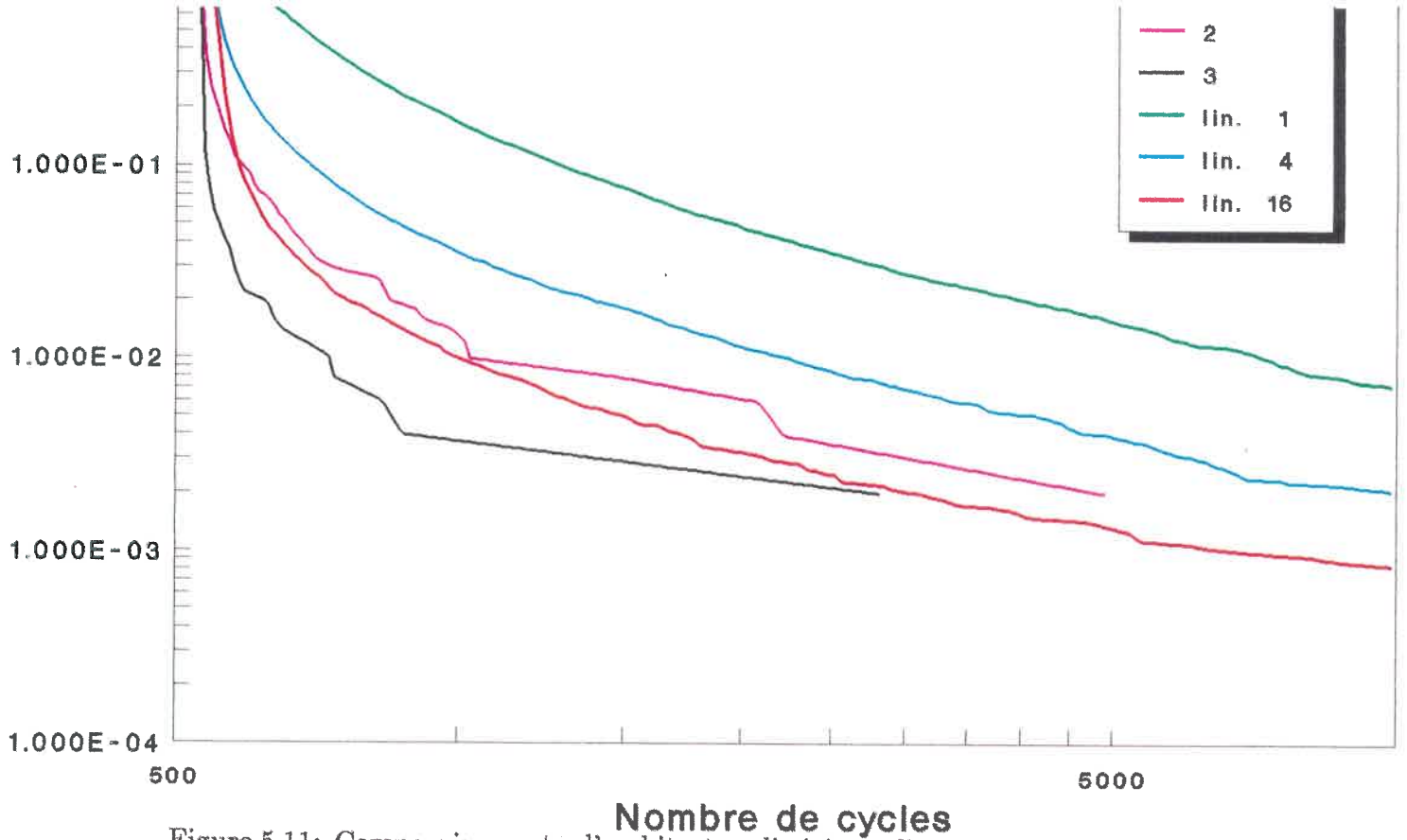


Figure 5.11: Comparaison entre l'architecture linéaire et l'architecture arborescente non-reconvergente

$E_b/N_0 = 2.698dB$, k = 24, code de Johannesson
nombre de bits par blocs = 499
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
linéaire = 20000
arborescente = 500

- les extenseurs et les files prioritaires reçoivent un signal indiquant si les données reçues concernent des noeuds valides.

ce qui est minime.

On notera que ce phénomène peut être dû à la longueur de contrainte qui est relativement élevée et qu'une longueur de contrainte de 7 ou 8 pourrait impliquer un autre comportement.

5.4 Conclusions

En résumé, on a vu que l'architecture linéaire voit ses performances s'améliorer avec le nombre de processeurs au point de vue de la variabilité de l'effort de calcul, tout en conservant un taux d'utilisation des extenseurs élevé (lorsque les blocs sont en moyenne assez difficiles à décoder). De plus, avec cette architecture, le partage des noeuds de métrique élevée entre les files prioritaires est bon et les coûts totaux augmentent moins que linéairement avec le nombre de processeurs. On a également vu que quatre processeurs entraînent des coûts qui sont sensiblement les mêmes que pour un ou deux processeurs tout en étant plus performant et qu'avoir seize processeurs implique qu'on peut intégrer un extenseur, une file et un module *bloc* sur un seul circuit ce qui impliquerait des frais réduits de beaucoup et rendrait probablement cette architecture concurrentielle, en terme de coûts, par rapport aux autres tout en ayant une variabilité de l'effort de calcul très réduite.

Pour ce qui est de l'architecture arborescente reconvergente, on a constaté qu'elle a une courbe de la cumulative de l'effort de calcul qui est de valeur plus élevée que pour l'architecture linéaire, pour un nombre égal de processeurs. Les coûts plus élevés de cette architecture combinés à ses piètres performances font en sorte qu'elle offre peu d'intérêt.

L'architecture arborescente non-reconvergente, par contre, est intéressante à cause de sa très faible variabilité de l'effort de calcul bien qu'elle impose des coûts élevés. Donc, si le critère principal de sélection est la faiblesse de la variabilité de l'effort de calcul, on optera plutôt pour cette architecture.

L'architecture linéaire se dégage donc comme étant la plus intéressante tant du point de vue des performances (en fonction des coûts) que des possibilités d'intégration. Il semble aussi qu'utiliser cette architectures avec de quatre à seize extenseurs (ou files prioritaires) serait un choix judicieux; cependant, des simulations avec plus de processeurs et pour des rapports signal sur bruit plus faibles serait indiquées puisqu'il est très difficile d'extrapoler les courbes obtenues. On pourrait alors tirer des conclusions plus précises et on pourrait savoir si un nombre de processeurs élevé serait un atout en cas de canal très bruité.

Chapitre 6

Conclusions

6.1 Sommaire des principaux résultats obtenus

Un circuit intégré dénommé extenseur a donc été conçu et décrit dans le présent mémoire; on a montré qu'il est flexible quant au taux de codage, au nombre de niveaux de quantification du canal et de la plage de métriques qu'il supporte et qu'il a été doté d'une architecture lui permettant d'atteindre de bonnes performances pour un circuit de cette complexité.

On a également décrit des architectures multiprocesseurs qui implantent des algorithmes qui s'approchent de l'algorithme multi-chemins; on a montré qu'en fonction d'une série de critères d'évaluation qualitatifs, on pouvait s'attendre à un bon comportement de ces architectures.

Par la suite, un programme de simulation a été décrit. Ce programme permet d'étudier le comportement des architectures sous toutes conditions d'opérations réalistes; c'est-à-dire pour des codes de longueur qu'on puisse utiliser en pratique, pour un nombre de processeurs qui ne soit pas utopique et pour des rapports signal sur bruit qui sont usuels.

Ce simulateur a permis d'effectuer des simulations qui ont montré que l'architecture linéaire est très intéressante tout en restant peu coûteuse. On a vu également qu'on pourrait difficilement, pour cette architecture, limiter le nombre de cycles permis pour décoder un bloc à moins de deux fois la longueur d'un bloc si on veut garder un taux de débordement de l'historique inférieur à 1%. On a également constaté que l'architecture arborescente non-reconvergente permet, à des coûts plus élevés, de réduire sensiblement la variabilité de l'effort de calcul tout en conservant un taux de débordement qui soit du même ordre de grandeur. Elle permettrait donc de réduire les frais associés aux tampons d'entrée et de sortie. On pourrait même réduire les tampons à leur strict minimum et amener la variabilité de calcul apparente à zéro si on est prêt à augmenter légèrement (38% dans les pires cas) le temps de décodage d'un bloc, en forçant le décodeur à attendre s'il a décoder le bloc en un temps inférieur d'un cycle (ou plus) à la capacité de l'historique.

6.2 Avenues de recherche intéressantes

Ayant obtenu un décodeur dont la variabilité de l'effort de calcul est aussi réduite que désiré et qui, si on le veut, est peu coûteux; on voudrait éventuellement un décodeur pouvant fonctionner à grand débit (pour les communications satellites notamment). On sait cependant que ce type de décodeur ne peut pas, avec une techonologie CMOS, décoder à de tels débits; on devra donc utiliser plusieurs de ces décodeurs en parallèle pour obtenir le débit d'information désiré.

Une tactique qui semblerait *a priori* intéressante pour utiliser un tel système multi-décodeur, étant donné que la plupart des blocs sont faciles à décoder, consisterait à utiliser plusieurs décodeurs possédant des ressources limitées et un ou un petit nombre de décodeurs doté(s) de beaucoup de ressources. Cependant,

l'architecture linéaire pourrait être dotée de ressources assez considérables et ne coûter que marginalement plus cher (ou même être moins dispendieuse) qu'un système minimal. On peut mentionner qu'avoir des décodeurs "asymétriques" implique des frais supplémentaires au niveau du système. De plus, l'architecture arborescente non-reconvergente (avec sa variabilité de l'effort de calcul pratiquement nulle) permettrait de décoder les messages avec un gain de codage élevé tout en évitant d'avoir à resynchroniser les blocs à la sortie de ces décodeurs évitant ainsi les tampons de sortie nécessaires (et qui peuvent devenir assez considérables) dus à la variabilité de l'effort de calcul. Des simulations supplémentaires et une étude de coûts plus fine permettraient de juger du choix optimum selon les besoins des différentes applications.

Donc, les travaux à effectuer qui donneraient une vue d'ensemble du champ d'étude défriché dans le présent texte sont: des simulations de l'architecture linéaire avec plus de processeurs et avec plus de blocs (pour mieux definir la position des "plateaux"), introduire un ou plusieurs modèles de coûts plus précis et des simulations multi-décodeurs permettant de cerner le comportement de cette architecture et de l'architecture arborescente non-reconvergente dans ledit système. Une avenue qui serait également intéressante à explorer consisterait à étudier le comportement de l'architecture linéaire lorsqu'on force un décodage avant (donc sans variabilité de l'effort de calcul). Deux façons d'implanter cet algorithme ont été proposées au chapitre 3 et l'une d'elles a été implantée dans le simulateur mais n'a pu être mise en oeuvre faute de temps.

On pourrait penser qu'il serait intéressant d'imaginer des architectures ayant une structure différente (en anneau ou en étoile, par exemple); cependant, sachant que les architectures proposées réflètent la structure de l'arbre de décodage (même l'architecture linéaire car elle permet *l'avancement* simultané des meilleurs noeuds),

les autres architectures imaginables ne pourraient qu'imposer des contraintes de communications inter-module qui ralentiraient le système de façon à renverser les gains potentiels.

Annexe A

Description détaillée des communications inter-module

Les tableaux A.1 et A.2 donnent la liste complète des signaux de contrôle et données échangés par les modules dans le système linéaire et les systèmes arborescents respectivement.

modules		signal	direction
file	extenseur	NM	$\leftrightarrow$
		ST	$\leftrightarrow$
		PE	$\leftrightarrow$
		IQ	$\rightarrow$
		S1	$\rightarrow$
		S2	$\rightarrow$
file	historique	ST	$\rightarrow$
		NA	$\leftrightarrow$
file	file	NA	$\leftrightarrow$
		ND	$\leftrightarrow$
bloc	file	S1	$\rightarrow$
		S2	$\rightarrow$
		ND	$\leftarrow$
		IQ	$\rightarrow$
tri	historique	BF	$\leftarrow$
		BN	$\rightarrow$
tri	contrôleur	FF	$\rightarrow$
		SF	$\rightarrow$
		SC	$\leftarrow$
file	contrôleur	PC	$\leftarrow$
extenseur	contrôleur	PC	$\leftarrow$
extenseur	monde ext.	DA	$\leftarrow$
historique	contrôleur	HC	$\leftarrow$
		BF	$\rightarrow$
		WF	$\rightarrow$
bloc	contrôleur	BC	$\leftarrow$
tous	contrôleur	MA	$\leftarrow$
monde ext.	contrôleur	RD	$\rightarrow$
		IF	$\rightarrow$
		RF	$\rightarrow$

Tableau A.1: Communications inter-module, système linéaire

modules		signal	direction
file	extenseur	NM	$\leftrightarrow$
		ST	$\leftrightarrow$
		PE	$\leftrightarrow$
		IQ	$\rightarrow$
		S1	$\rightarrow$
		S2	$\rightarrow$
extenseur	extenseur	NM	$\leftrightarrow$
		ST	$\leftrightarrow$
		PE	$\leftrightarrow$
file	historique	ST.	$\rightarrow$
		NA	$\leftrightarrow$
bloc	file	S1	$\rightarrow$
		S2	$\rightarrow$
		ND	$\leftarrow$
		IQ	$\rightarrow$
tri	historique	BF	$\leftarrow$
		BN	$\rightarrow$
tri	contrôleur	FF	$\rightarrow$
		SF	$\rightarrow$
		SC	$\leftarrow$
file	contrôleur	PC	$\leftarrow$
extenseur	contrôleur	PC	$\leftarrow$
extenseur	monde ext.	DA	$\leftarrow$
historique	contrôleur	HC	$\leftarrow$
		BF	$\rightarrow$
		WF	$\rightarrow$
bloc	contrôleur	BC	$\leftarrow$
tous	contrôleur	MA	$\leftarrow$
monde ext.	contrôleur	RD	$\rightarrow$
		IF	$\rightarrow$
		RF	$\rightarrow$

Tableau A.2: Communications inter-module, systèmes arborescents

Annexe B

Signification des symboles utilisés

Le tableau B.1 donne la signification des mnémoniques utilisées dans les sections 2.2.2, 2.2.3 et 3.3.1.

symbole	signification
IQ	inside queue
NA	node address
ND	node depth
NM	node metric
PE	perforate
ST	state
S1	symbol 1
S2	symbol 2
BC	bloc control
BN	best node
DA	data
FF	follow finish
HC	history control
IF	input finish
BF	bloc finished
PC	processors control
RD	reset decoder
RF	read finish
SC	sort control
SF	sort finish
MA	module address
WF	write finish

Tableau B.1: Liste des symboles utilisés

Annexe C

Algorithmes détaillés

Cette annexe donne le nom de chaque fichier contenant le code-source du simulateur décrit au chapitre 4 et, pour chaque routine qu'il contient, son nom, ses entrées, ses sorties et son algorithme. On notera qu'une ligne qui est décalée et qui commence par trois points signifie qu'elle continue la ligne précédente.

fichier: decodeur.c

routine: main

entrées: nom de fichier de paramètres (éventuellement).

sorties: statistiques envoées dans les fichiers de sorties et
... d'erreurs dont les noms apparaissent dans le fichier de
... paramètres.

algorithme:

si un nom de fichier a été donné
    lire les données de ce fichier (routine "lire_donnees")
sinon
    lire les données du fichier "decodeur.dat" (routine "lire_donnees")
si on est en mode de "déverminage"
    ouvrir le fichier de résultats du "déverminage"
si on n'a pas réussi à ouvrir ce fichier
    afficher un message d'erreur et terminer l'exécution
pour un nombre de cycles égal à la longueur de contrainte ...
    mettre le bit correspondant du masque d'état à "1"
inverser la séquence des bits du code (car les bits de message sont
    ... entrés à droite alors que code est spécifié dans l'ordre inverse)
masquer le code (au cas où on aurait spécifier un code plus grand que
    ... la longueur de contrainte)
"imprimer" dans le fichier de sortie les paramètres de la simulation
pour chaque niveau de bruit désiré ...
    calcul du rapport s/n
    initialiser le canal (routine "init_canal")
    "impression" dans le fichier de résultats de l'entête pour ce
        ... rapport s/n (E_b/N_0, R_{comp}, capacité et métriques)
    initialiser la semence du générateur de nombres pseudo-aléatoires
    si on désire "déverminer le générateur de nombre aléatoire
        "imprimer" dans le fichier de sortie l'entête des semences
    initialiser les compteurs de blocs ayant débordés, d'itérations

```
    ... et d'erreurs totales à zéro
pour un indice variant de 0 à la profondeur de l'historique ...
    mettre à zéro l'enregistrement correspondant du tableau d'erreurs
        ... (si on désire des statistiques tabulées, on initialise les deux tableaux)
pour chaque valeur du niveau de quantification ...
    mettre à zéro l'élément correspondant du tableau des transitions
        ... expérimentales
pour chaque bloc à décoder ...
    mettre à zéro le compteur d'erreurs
    si on désire "déverminer" le générateur de nombre aléatoire
        imprimer la semence
    initialiser l'état à zéro
    pour chaque bit du message ...
        si on désire des séquence de zéro
            mettre le bit à zéro
        sinon
            générer un nombre aléatoire
            s'il est plus grand que 0.5
                mettre le bit à zéro
            sinon
                le mettre à un
            insérer le bit dans l'état (le registre à décalage)
            masquer l'état
            coder l'état et mettre le résultat dans les bits
                ... correspondants de la table des symboles
    pour chaque bit de la queue du message ...
```

```
            mettre le bit à zéro
            insérer le bit (zéro) dans l'état (le registre à décalage)
            coder l'état et mettre le résultat dans les bits
                ... correspondants de la table des symboles
        bruiter le message codé (routine "generer_sequence_bruitee")
        décoder le bloc (routine "decodeur")
        ajouter le nombre d'itérations nécessaires pour décoder
            ... ce bloc au nombre total d'itérations
        incrémenter l'enregistrement pertinent du tableau de compteur
            ... d'itérations
        si le bloc n'a pas débordé
            pour chaque bit du message ...
                si le bit décodé est différent du bit original
                    incrémenter le compteur d'erreurs
        additionner le nombre d'erreurs du bloc au bon enregistrement du
            ... tableau d'erreurs
        additionner le nombre d'erreurs au nombre total d'erreurs
    s'il y a des blocs qui ont débordés
        imprimer un message indiquant qu'il y a eu débordement
    "imprimer" dans le fichier de sortie le nombre de blocs ayant débordés
    trouver le nombre d'itérations par bloc le plus faible
    trouver le nombre d'itérations par bloc le plus grand
    mettre la variance à zéro
    calculer la moyenne en divisant le nombre total d'itérations par le
        ... nombre de blocs
    pour chaque valeur d'itérations entre celui du bloc ayant le plus
```

... faible et celui ayant le plus élevé ...
s'il y a eu des blocs ayant nécessités ce nombre d'itérations
additionner à la variance le carré de l'écart du
... nombre d'itérations à la moyenne multiplié par
... le nombre de blocs correspondant
diviser la variance par la moyenne
"imprimer" dans le fichier de sortie la semence finale
"imprimer" dans le fichier de sortie la moyenne du nombre d'itérations
"imprimer" dans le fichier de sortie la variance du nombre d'itérations
"imprimer" dans le fichier de sortie le nombre total d'erreurs
"imprimer" dans le fichier de sortie le taux d'erreurs
"imprimer" dans le fichier de sortie l'entête du tableau des
... probabilités de transitions expérimentales
pour chaque niveau de quantification ...
"imprimer" dans le fichier de sortie la probabilité de
transition correspondante
mettre la cumulative du nombre de blocs à un
mettre le compteur de blocs à zéro
mettre le compteur des erreurs dans les statistiques à zéro
mettre le poids (dans le compte de la moyenne) à zéro
"imprimer" dans le fichier de sortie l'entête du tableau de la cumulative
si on désire des statistiques stabulées
si on a une "distance" de moins de cent entre les nombres
... d'itérations minimal et maximal
pour chaque nombre d'itérations pertinent ...
si on a eu des blocs ayant nécessités ce nombre d'itérations

```
                additionner au nombre de blocs (pour les statistiques)
                    ... le nombre de blocs ayant eu ce nombre d'itérations
                additionner au nombre d'erreurs (pour les statistiques)
                    ... le nombre d'erreurs des blocs pertinents
                calculer le taux d'erreurs courant en divisant le nombre
                    ... d'erreurs ci-dessus par le produit du nombre de blocs
                    ... ci-dessus et du nombre de bits par bloc
                additionner au poids le produit du nombre de blocs ayant
                    ... eu ce nombre d'itérations par le nombre d'itérations
                calculer la moyenne provisoire en divisant le poids par le
                    ... nombre de blocs ci-dessus
                "imprimer" dans le fichier de sortie le nombre d'itérations,
                    ... le nombre de blocs, la cumulative, le nombre d'erreurs,
                    ... le taux d'erreurs provisoire et la moyenne provisoire
                à la cumulative soustraire le quotient du nombre de
                    ... blocs par le nombre total de blocs à décoder
sinon
    mettre le pointeur de début d'intervalle au nombre
        ... d'itérations le plus faible
pour un numéro d'intervalle variant entre 0 et 49 ...
        mettre le nombre de blocs tabulé à zéro
        calculer la fin de l'intervalle
        pour chaque nombre d'itérations de l'intervalle ...
            accumuler le nombre total de blocs de l'intervalle
            accumuler le nombre total d'erreurs de l'intervalle
        s'il y a eu des blocs dans l'intervalle
```

```
                additionner au nombre de blocs (pour les statistiques)
                    ... le nombre de blocs de l'intervalle
                additionner au nombre d'erreurs (pour les statistiques)
                    ... le nombre d'erreurs des blocs de l'intervalle
                calculer le taux d'erreurs courant en divisant le nombre
                    ... d'erreurs ci-dessus par le produit du nombre de blocs
                    ... ci-dessus et du nombre de bits par bloc
                additionner au poids le produit du nombre de blocs de
                    ... l'intervalle par le nombre d'itérations
                calculer la moyenne provisoire en divisant le poids par le
                    ... nombre de blocs ci-dessus
                "imprimer" dans le fichier de sortie l'intervalle,
                    ... le nombre de blocs, la cumulative, le nombre d'erreurs,
                    ... le taux d'erreurs provisoire et la moyenne provisoire
                à la cumulative soustraire le quotient du nombre de
                    ... blocs par le nombre total de blocs à décoder
                    ... calculer le début du prochain intervalle
sinon
    pour chaque nombre d'itérations pertinents
        s'il y a eu des blocs
            additionner au nombre de blocs (pour les statistiques)
                ... le nombre de blocs ayant eu ce nombre d'itérations
            additionner au nombre d'erreurs (pour les statistiques)
                ... le nombre d'erreurs des blocs
            calculer le taux d'erreurs courant en divisant le nombre
                ... d'erreurs ci-dessus par le produit du nombre de blocs ci-dessus
```

... et du nombre de bits par bloc

additionner au poids le produit du nombre de blocs par le

... nombre d'itérations

calculer la moyenne provisoire en divisant le poids par le

... nombre de blocs ci-dessus

"imprimer" dans le fichier de sortie l'intervalle,

... le nombre de blocs, la cumulative, le nombre d'erreurs,

... le taux d'erreurs provisoire et la moyenne provisoire

à la cumulative soustraire le quotient du nombre de

... blocs par le nombre total de blocs à décoder

"imprimer" dans le fichier de sortie la fin du tableau

fermer les fichier de sortie et d'erreurs

afficher un message de fin de décodage

terminer l'exécution

fichier: decodeur_module.c

routine: decodeur

entrées: pointeur à une structure décrivant l'environnement du décodeur
pointeur à une structure décrivant le système complet
un pointeur aux structures des files prioritaires
un pointeur aux vecteurs contenant les symboles reçus

sorties: le message une fois décodé
le nombre de blocs ayant débordé (on l'aura
... incrémenté le cas échéant)

algorithme:

initialiser le files prioritaires (routine "q_init")
initialiser le(s) historique(s) (routine "hist_init")
pour chaque file ...
    indiquer que le noeud sorti est invalide
initialiser le noeud sorti de la première file aux valeurs concernant la
    ... racine de l'arbre de décodage
si on veut simuler le module "tri"
    mettre le compteur de noeuds à zéro
mettre le fanion de débordement à FAUX
mettre le compteur de cycles à zéro

```
boucler (tant que le décodage n'est pas terminé, ce qui est testé dans la boucle) ...
    pour chaque file (dans l'ordre inverse au flux de données) ...
        insérer les noeuds reçus et retirer le meilleur
    pour chaque extenseur (dans le sens inverse au flux de données) ...
        lire les symboles pertinents au noeud (simulation du module bloc)
        effectuer l'extension du noeud (routine "extenseur")
    copier le noeud sorti de la dernière file de la variable temporaire
        ... à la variable qu'il doit occuper (ceci pour éviter d'étendre un
        ... noeud sur deux branches en un cycle)
    si le noeud en question est valide
        l'enregistrer dans l'historique
    incrémenter le compteur de cycles
    si le noeud est valide et s'il est à la fin du bloc
        si on veut simuler le module "tri"
            conserver les données consernant le noeud
            incrémenter le compteur de noeuds de fin de blocs
            si on a suffisamment de noeud de fin de bloc
                mettre l'adresse d'historique à celle du dernier noeud sorti
                mettre la métrique du meilleur noeud à celle du dernier noeud sorti
                pour chacun des noeuds de fin de bloc ...
                    si sa métrique est supérieur à celle du meiller noeud
                        mettre l'adresse du meilleur noeud à celle de ce noeud
                        mettre la métrique du meilleur noeud à celle de ce noeud
            mettre l'adresse d'historique à celle du meilleur noeud
            sortir de la boucle
        sinon
```

```
                mettre l'adresse d'historique à celle du noeud
                sortir de la boucle
        si le noeud est valide et s'il y a débordement de l'historique
            mettre l'adresse d'historique à celle du noeud
            incrémenter le compteur de débordement
            mettre à VRAI le fanion de débordement
            sortir de la boucle
        s'il n'y a pas eu de débordement ...
            pour chaque bit du message (dans l'ordre inverse) ...
                lire en remontant la liste chaînée l'enregistrement précédent de l'historique
                pour chaque extension conservée dans cet enregistrement ...
                    extraire le bit décodé
                    indiquer qu'on a un autre bit d'extrait
retourner le contrôle à la routine appelante en donnant le nombre de cycles
        ... nécessaire pour décoder le bloc
```

fichier: codeur.c

routine: codeur

entrées: état
un pointeur à la structure décrivant le système

sorties: les parités générées

algorithme:

masquer l'état avec les mots donnant les connexions des codes
calculer le mot (16 bits) obtenu en faisant le ou-exclusif entre le mot le
... plus significatif et celui qui est le moins significatif d'un état masqué
calculer l'octet en effectuant la même opération sur les deux octets
... du résultat précédent
calculer les quatre bits par la même opération
calculer les deux bits de la même manière
calculer la parité de la même manière
recommencer ces opérations de ou-exclusifs pour le deuxième état masqué
masquer ces deux résultats pour ne laisser que le bit d'intérêt
retourner le contrôle à la routine appelante

fichier: init_canal.c

routine: init_canal

entrées: pointeur à une structure décrivant le système utilisé
le rapport s/n
le facteur multiplicateur de métrique

sorties: structure décrivant le canal

algorithme:

calculer les probabilités de transitions du canal (routine "dmc_prob_trans")
calculer les métriques (routine "dmc_metrique")
calculer R_{comp} (routine "dmc_rcomp")
calculer la capacité du canal (routine "dmc_capacity")
calculer le rapport s/n en dB (routine "dmc_snrdb")
retourner le contrôle à la routine appelante

routine: dmc_prob_trans

entrées: pointeur à une structure décrivant le système utilisé
le rapport s/n

sorties: structure décrivant le canal

algorithme:

calculer E_b/N_0 (en dB) en additionnant à E_n/N_0 le logarithme
... en base 10 de l'inverse du taux de codage
calculer E_b/N_0 (en "unité") en élevant 10 à la
... puissance donnée par son dixième
calculer une distance comme étant quatre divisé par le nombre de
... niveaux de quantification
calculer la valeur de départ de la boucle qui suit
pour chaque niveau de quantification autre que le premier ...
calculer la probabilité courante
calculer la probabilité de transition courante
calculer la cumulative de probabilité de transition courante
mettre à jour la une variable temporaire de cumulative de probabilité
calculer la première probabilité de transition et la cumulative correspondante
... (i.e. "1")
retourner le contrôle à la routine appelante

routine: prob

entrées: un nombre en format "virgule flottante"

sorties: un nombre en virgule flottante

algorithme:

retourner le contrôle à la routine appelante en lui donnant comme
... valeur le nombre obtenu en divisant par deux le résultat de la fonction
... "erfc" appliquée au quotient du nombre reçu divisé par la racine
... carrée de deux

routine: dmc_capacity

entrées: pointeur à une structure décrivant le système utilisé
structure décrivant le canal

sorties: la capacité du canal

algorithme:

initialiser la capacité du canal à zéro
pour chaque niveau de quantification ...
additionner à la capacité le produit de la probabilité de
... transition correspondante au logarithme du quotient du double de la
... probabilité de transition par la somme de la probabilité de
... transition et de celle qui la précède
retourner le contrôle à la routine appelante en lui donnant la capacité

routine: dmc_metrique

entrées: pointeur à une structure décrivant le système utilisé
structure décrivant le canal
facteur multiplicateur de métrique

sorties: structure décrivant le canal (modifiée)

algorithme:

pour chaque niveau de quantification ...
    calculer la métrique
    si la métrique est négative
        ajouter un biais de -0.5
    sinon
        ajouter un biais de +0.5
retourner le contrôle à la routine appelante

routine: dmc_rcomp

entrées: pointeur à une structure décrivant le système utilisé
structure décrivant le canal

sorties: R_{comp}

algorithme:

initialiser R_{comp} à zéro
pour chaque paire (opposée) de niveaux de quantification ...
    calculer la moyenne de leur racine carrée
    ajouter à R_{comp} le carré de cette moyenne
R_{comp} devient l'inverse du quotient du logarithme de son double et du
    ... logarithme de deux
retourner le contrôle à la routine appelante en lui donnant R_{comp}

routine: dmc_snrdb

entrées: pointeur à une structure décrivant le système utilisé
E_n/N_0

sorties: E_b/N_0

algorithme:

retourner le contrôle à la routine appelante en lui donnant la somme
    ... de E_n/N_0 et du produit de dix par le logarithme en base dix de
    ... l'inverse du taux de codage

routine: snrdb_a_symdb

entrées: pointeur à une structure décrivant le système utilisé
E_n/N_0

sorties: E_b/N_0

algorithme:
retourne le contrôle à la routine appelante en lui donnant la somme de
... E_b/N_0 et du produit de dix par le logarithme en base dix de
... l'inverse du taux de codage

fichier: generer.c

routine: generer_sequence_bruitee

entrées: pointeur à une structure décrivant le système utilisé
structure décrivant le canal
le message à bruiter
la longueur du message (incluant la queue)
la semence du générateur de nombre aléatoire
un pointeur à la table de transitions expérimentales

sorties: le message bruité
la table de transitions expérimentales

algorithme:

```
pour chaque bit du message ...
    générer un nombre aléatoire
    si on ne doit pas perforer le symbole
        pour chaque niveau de quantification ...
            si le nombre aléatoire est plus petit que la cumulative de la
                ... probabilité de transition du canal
                si le symbole à bruité est "1"
                    bruiter en assignant à ce symbole la différence
                        ... entre le niveau maximum de quantification et
                        ... l'indice correspondant à la cumulative courante
```

sinon

bruiter en assignant au symbole l'indice correspondant à

... la cumulative courante

incrémenter la position courante du tableau de transition expérimentale

sortir de la boucle

retourner le contrôle à la routine appelante

routine: rando

entrées: semence d'entree

sorties: semence de sortie

nombre aléatoire

algorithme:

multiplier la semence par 65539

si le résultat est négatif

lui additionner 2147483647 puis 1

retourner le contrôle à la routine appelante en lui donnant le

... produit de la nouvelle semence par 0.4656612875e-9

routine: rando_long

entrées: semence d'entree

sorties: semence de sortie (qui est aussi le nombre aléatoire)

algorithme:

multiplier la semence par 65539
si le résultat est négatif
    lui additionner 2147483647 puis 1
retourner le contrôle à la routine appelante en lui donnant la nouvelle semence

fichier: globale.c

routine: lire_donnees

entrées: nom du fichier de données

sorties: pointeur à un fichier de sortie
pointeur à un fichier d'erreurs
changements dans la structure du système
changements dans la structure de l'environnement

algorithme:

```
demander l'ouverture du fichier de données
mettre à zéro les pointeurs aux fichiers de sortie et d'erreurs
si l'ouverture du fichier de données a été réussie
    lire le nom du fichier de résultats
    si la lecture n'a pas réussi
        terminer en affichant un message d'erreur (routine "aff_erreur_fin")
    lire la fin de la ligne dans un tampon "dummy"
    lire le nom du fichier d'erreur
    si la lecture n'a pas réussi
        terminer en affichant un message d'erreur (routine "aff_erreur_fin")
    lire la fin de la ligne dans un tampon "dummy"
    demander l'ouverture des fichiers de sortie et d'erreur
```

```
si l'ouverture d'un des fichiers n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire le numérateur du taux de codage
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le dénominateur du taux de codage
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la longueur de contrainte
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le nombre de niveau de quantification
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le premier générateur du code utilisé
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le deuxième générateur du code utilisé
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
```

```
lire le facteur multiplicateur de métrique
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la profondeur de(s) l'historique(s)
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la profondeur des files prioritaires
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
si on a une architecture linéaire
    lire le nombre de processeurs
    si la lecture n'a pas réussi
        terminer en affichant un message d'erreur (routine "aff_erreur_fin")
    lire la fin de la ligne dans un tampon "dummy"
sinon
    lire la profondeur des arbres
    si la lecture n'a pas réussi
        terminer en affichant un message d'erreur (routine "aff_erreur_fin")
    lire la fin de la ligne dans un tampon "dummy"
lire le nombre de blocs à simuler
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
```

lire le nombre de bits par blocs
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la métrique de la racine de l'arbre
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la semence initiale du générateur de nombres aléatoires pour créer le message
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire la semence initiale du générateur de nombres aléatoires pour bruiter le message
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le type de rapport s/n
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le rapport s/n initial
si la lecture n'a pas réussi
    terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le rapport s/n final
si la lecture n'a pas réussi

terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le nombre de rapport s/n
si la lecture n'a pas réussi
terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
lire le fanion "sequence de zéro?"
si la lecture n'a pas réussi
terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
valider les données (routine "valider_donnees")
pour chaque entier de zéro au numérateur de taux de codage ...
lire le patron de perforation
si la lecture n'a pas réussi
terminer en affichant un message d'erreur (routine "aff_erreur_fin")
lire la fin de la ligne dans un tampon "dummy"
tenter de fermer le fichier de données
si la fermeture n'a pas réussi
terminer en affichant un message d'erreur (routine "aff_erreur_fin")
sinon
terminer en affichant un message d'erreur (routine "aff_erreur_fin")
retourner le contrôle à la routine appelante

routine: valider_donnees

entrées: pointeur à la structure du système
pointeur à la structure de l'environnement
pointeur au fichier de sortie
pointeur fichier d'erreur

sorties: aucune

algorithme:

si le numérateur du taux de codage est supérieur ou égal au dénominateur
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le numérateur est plus grand que la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le numérateur est inférieur à la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le dénominateur est plus grand que la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le dénominateur est inférieur à la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le nombre de niveau de quantification est inférieur à la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le nombre de niveau de quantification est supérieur à la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si la profondeur de(s) l'historique(s) est inférieur à la limite admissible
    afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si la profondeur de(s) l'historique(s) est supérieur à la limite admissible

```
        afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si la profondeur de(s) file(s) est inférieur à la limite admissible
        afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si la profondeur de(s) file(s) est supérieur à la limite admissible
        afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si on a un système linéaire
        si le nombre de processeurs est inférieur à la limite admissible
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
        si le nombre de processeurs est supérieur à la limite admissible
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
        si le modulo de la longueur des blocs (avec la queue) avec le nombre de
                ... processeurs est différent de zéro
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
sinon
        si la profondeur des arbres est inférieur à la limite admissible
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
        si la profondeur des arbres est supérieure à la limite admissible
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
        si le modulo de la longueur des blocs (avec la queue) avec la profondeur
                ... des arbres est différent de zéro
                afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le nombre de bits par blocs est supérieureà la limite admissible
        afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le nombre de rapport s/n est négatif
        afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si le rapport s/n final est supérieur au rapport s/n initial
```

afficher un message d'erreur et terminer (routine "aff_erreur_fin")
si on a donné des rapports s/n qui sont E_b/N_0

convertir en E_n/N_0 le premier et le dernier rapport s/n (routine "snrdb_a_symdb")
retourner le contrôle à la routine appelante

fichier: extenseur.c

routine: extenseur

entrées: noeud à étendre
pointeur à la structure du système

sorties: les deux noeuds résultants de l'extension

algorithme:

```
copier les données du noeud reçu dans un noeud temporaire
copier le noeud-père dans les noeuds-fils (les données
    ... non-calculées restent les mêmes par défaut)
si le noeud reçu n'est pas valide
    retourner le contrôle à la routine appelante
incrémenter le compteur de profondeur dans l'arbre des noeuds-fils
calculer l'état des noeuds-fils
calculer les parités correspondant à ces états (routine "codeur")
si la parité correspondant au premier générateur du code et au
        ... premier noeud-fils est "1"
    assigner à la métrique correspondante celle trouvée à
        ... l'adresse inverse de celle donnée par le premier symbole
sinon
    assigner à la métrique correspondante celle trouvée à
```

... l'adresse donnée par le premier symbole
si la parité correspondant au deuxième générateur du code et au
... premier noeud-fils est "1"
assigner à la métrique correspondante celle trouvée à
... l'adresse inverse de celle donnée par le deuxième symbole
sinon
assigner à la métrique correspondante celle trouvée à
... l'adresse donnée par le deuxième symbole
si la parité correspondant au premier générateur du code et au
... deuxième noeud-fils est "1"
assigner à la métrique correspondante celle trouvée à
... l'adresse inverse de celle donnée par le premier symbole
sinon
assigner à la métrique correspondante celle trouvée à
... l'adresse donnée par le premier symbole
si la parité correspondant au deuxième générateur du code et au
... deuxième noeud-fils est "1"
assigner à la métrique correspondante celle trouvée à
... l'adresse inverse de celle donnée par le deuxième symbole
sinon
assigner à la métrique correspondante celle trouvée à
... l'adresse donnée par le deuxième symbole
si le premier bit du patron de perforation courant est VRAI (i.e. on doit perforer)
perforer les métriques de branches correspondant au premier générateur du code
si le deuxième bit du patron de perforation courant est VRAI (i.e. on doit perforer)
perforer les métriques de branches correspondant au deuxième générateur du code

```
additionner aux métriques de noeuds les métriques de branches
si on est dans la queue du message
    mettre à zéro la métrique du deuxième noeud-fils
    signaler que ce noeud ne contient pas de données valides
incrémenter les compteurs de perforation
s'ils ont atteint la valeur du numérateur du taux de codage
    les mettre à zéro
si on désire trier les noeuds et si le deuxième noeud-fils a une
métrique supérieure à celle de son frère ...
    interchanger le contenu de leurs structures
retourner le contrôle à la routine appelante
```

fichier: q_init.c

routine: q_init

entrées: profondeur des files prioritaires

sorties: pointeur aux "entêtes" des files prioritaires

algorithme:

pour chaque file prioritaire ...
    pour chaque élément de la file sauf le premier et le dernier ...
        mettre dans le pointeur "précédent" l'adresse de l'élément qui précède
        ... dans le tableau
        mettre dans le pointeur "suivant" l'adresse de l'élément qui suit
        ... dans le tableau
        indiquer que le noeud est invalide
    mettre dans le pointeur "précédent" du premier élément l'adresse NULL
    mettre dans le pointeur "suivant" du premier élément l'adresse du deuxième
    ... élément
    indiquer que ce noeud est invalide
    mettre dans le pointeur "précédent" du dernier élément l'adresse de
    ... l'avant-dernier élément
    mettre dans le pointeur "suivant" du premier élément l'adresse NULL
    indiquer que ce noeud est invalide

mettre dans l'élément "premier" de l'entête de la file l'adresse du premier élément

mettre dans l'élément "dernier" de l'entête de la file l'adresse du dernier élément

mettre le fanion indiquant si le dernier élément est valide à FAUX

retourner le contrôle à la routine appelante en lui donnant l'adresse de la première entête

fichier: queue.c

routine: queue

entrées: l'adresse du premier noeud à insérer
l'adresse de l'entête de la file
l'adresse où on doit mettre les données du noeud extrait
le nombre de noeud à insérer

sorties: modification à l'adresse du noeud de sortie

algorithme:

trier les noeuds à insérer en ordre décroissant de métriques
... (en utilisant éventuellement la routine "qsort")
initialiser le nombre de noeud valides au nombre de noeuds à insérer
... en partant du dernier noeud à insérer (celui qui a la métrique la
... plus faible) et tant que le noeud est invalide et que le nombre de noeud est
... positif ou nul ...
décrémenter le nombre de noeuds valides
si le nombre de noeuds valides est supérieur à zéro ...
insérer les noeuds dans la file (routine "insere_noeud")
retirer le noeud du sommet de la file (routine "enleve_noeud")
retourner le contrôle à la routine appelante

fichier: noeud_ins_enl.c

routine: insere_noeud

entrées: un pointeur au premier noeud à insérer
un pointeur à la entête de la file
le nombre de noeud à insérer

sorties: aucune

algorithme:

initialiser le nombre de noeuds à insérer à zéro
initialiser le pointeur de noeud courant au dernier de la file
tant que le nombre de noeuds à insérer est plus petit que le nombre de
... noeuds valides et que le noeud courant est invalide ou que sa métrique
... est inférieure ou égale à celle du noeud
... à insérer courant ("pointé" par le nombre de noeuds à
... insérer) faire ...
incrémenter le nombre de noeuds à insérer
ajuster le pointeur de noeud à celui qui le précède
si le pointeur au dernier noeud valide de la file est invalide ou que le dernier noeud
... de la file est invalide ou que la métrique du premier noeud à insérer est
supérieure à celle du premier noeud de la file ou que la différence
... de métrique entre le premier noeud à insérer et le dernier

... noeud de la file est supérieure à celle entre le premier noeud
... de la file et celle du premier noeud à insérer
assigner au premier noeud à chercher l'adresse du premier noeud de la file
sinon
en partant du dernier noeud valide de la file, tant que le premier noeud à
... chercher est valide et que sa métrique est inférieure ou égale
... à celle du premier noeud à insérer ...
remonter le premier noeud à chercher au noeud précédent
assigner au premier noeud à chercher le noeud qui le suit dans la file
pour chaque noeud à insérer ...
en partant du noeud courant et tant que ce noeud est valide et que sa
... métrique est supérieure à celle du noeud à insérer ...
aller au noeud suivant dans la file
si le noeud ne doit pas être inséré à la place du dernier élément de la file ...
retirer le dernier noeud de la file
si le pointeur au dernier noeud valide "vise" le dernier noeud de la file ...
ajuster le pointeur au dernier noeud valide de la file
ajuster le pointeur de dernier noeud de la file
sinon
ajuster le pointeur de dernier noeud de la file
si on veut insérer le noeud en tête de la file ...
ajuster les pointeurs de tête de file, au précédent de
... l'ancien premier, du suivant du nouveau premier et de son
... précédent en conséquence
sinon si le noeud courant est différent du dernier noeud de la file
... (alors on insère "dans" la file)...

```
        ajuster les pointeurs de tête de file, au précédent de
            ... l'ancien premier, du suivant du nouveau premier et de son
            ... précédent en conséquence
    si les données précédente dans l'enregistrement sont invalides
        ... (on vient d'insérer le noeud à la fin des données valides) ...
        assigner au pointeur du dernier noeud valide de la file l'adresse du
            ... noeud que l'on vient d'insérer
    copier les données du noeud dans l'enregistrement
    assigner au noeud courant celui qui le précède dans la file (dans
        ... l'énoncé de la boucle)
    le noeud à insérer devient le noeud suivant celui que l'on vient
        ... d'insérer (dans l'énoncé de la boucle)
retourner le contrôle à la routine appelante


routine:        enleve_noeud

entrées:    pointeur à la tête de la file d'où on doit retirer le noeud

sorties:    on met les données relatives au noeud dans la
            structure dont on a reçu le pointeur

algorithme:

mémoriser l'adresse du sommet de la file
copier les données du noeud du sommet à l'adresse reçue
```

retirer le noeud du sommet de la file
insérer cet élément à la fin de la file
signaler que ses données ne sont plus valides
ajuster le pointeur du dernier noeud de la file
retourner le contrôle à la routine appelante

fichier: historique.c

routine: hist_init

entrées: aucune
sorties: aucune

algorithme:

pour chaque historique
    mettre le pointeur d'enregistrement courant à "-1"
retourner le contrôle à la routine appelante

routine: push_hist

entrées: une donnée à enregistrer
une adresse à enregistrer
(éventuellement) le numéro de l'historique
... où on doit effectuer l'enregistrement
un numéro d'historique à enregistrer (éventuellement)

sorties: le pointeur d'enregistrement courant

algorithme:

incrémenter le pointeur d'enregistrement courant
mettre à cette adresse la donnée à enregistrer
mettre à cette adresse l'adresse à enregistrer
mettre à cette adresse le numéro d'historique à enregistrer (éventuellement)
retourner le contrôle à la routine appelante en lui donnant le
pointeur d'enregistrement courant

routine: pop_hist

entrées: une adresse d'enregistrement
un numéro d'historique où doit se faire le retrait (éventuellement)

sorties: la donnée trouvée à cette adresse

algorithme:

mettre dans une variable temporaire le pointeur à l'adresse où doit
... s'effectuer le retrait
(éventuellement) mettre dans une variable temporaire le pointeur à
... l'historique où doit s'effectuer le retrait
mettre dans l'adresse l'adresse de l'enregistrement du noeud-père
(éventuellement) mettre dans le pointeur au numéro d'historique le
... numéro de l'historique où se trouve le noeud-père
retourner le contrôle à la routine appelante en lui donnant la

... donnée se trouvant à l'adresse reçue en paramètre

Annexe D

Courbes supplémentaires

Les courbes présentées dans la présente annexe n'ont pas été incluses dans le corps du texte pour ne pas l'allourdir. Ce sont les courbes tracées pour d'autres rapports signal sur bruit dans le but de vérifier que les comportements ne changent pas dramatiquement en fonction de ce rapport et que les conclusions tirées tiennent quelque soit sa valeur.

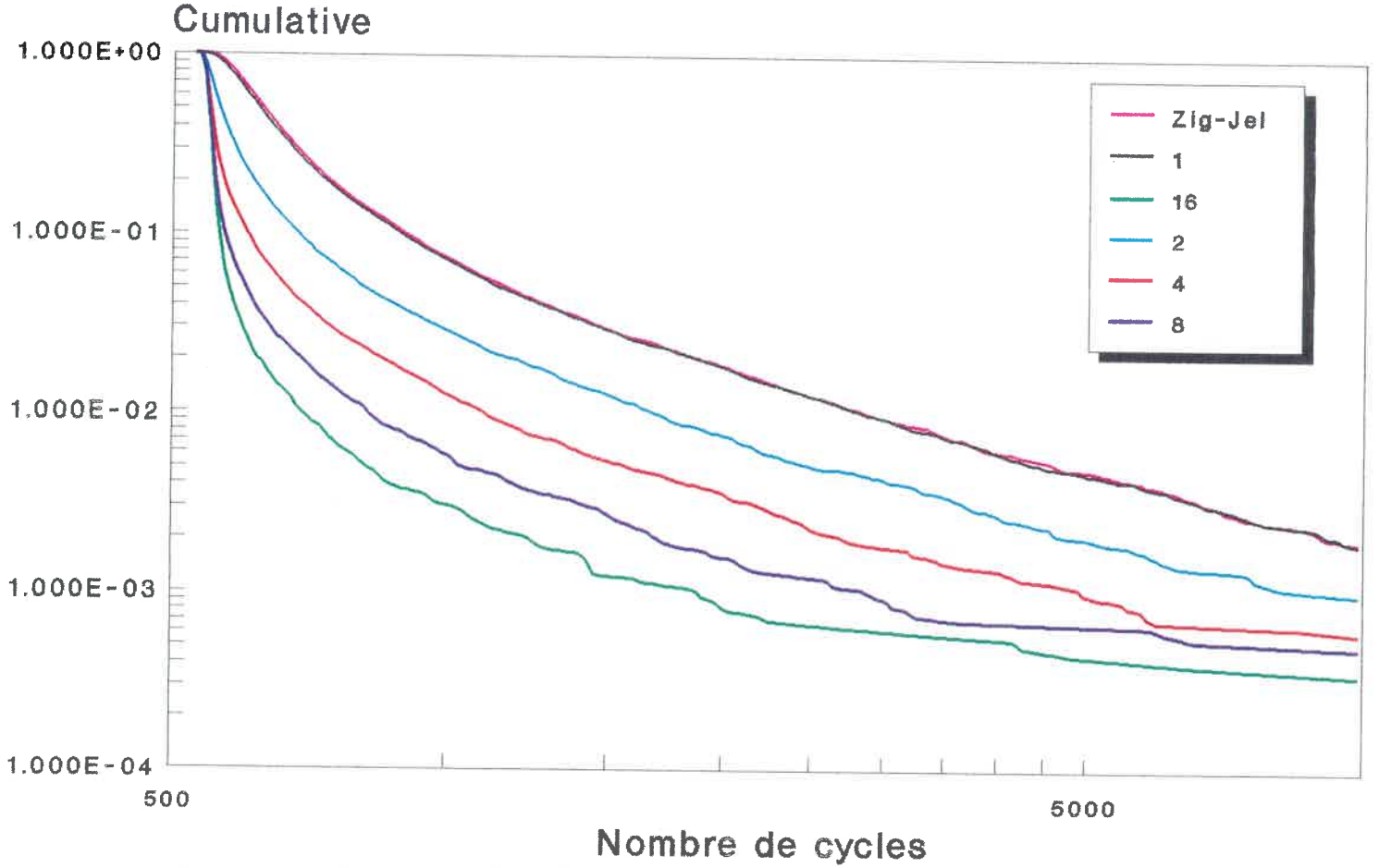


Figure D.1: Cumulative de l'effort de calcul selon le nombre de cycles $E_b/N_0 = 3.010dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

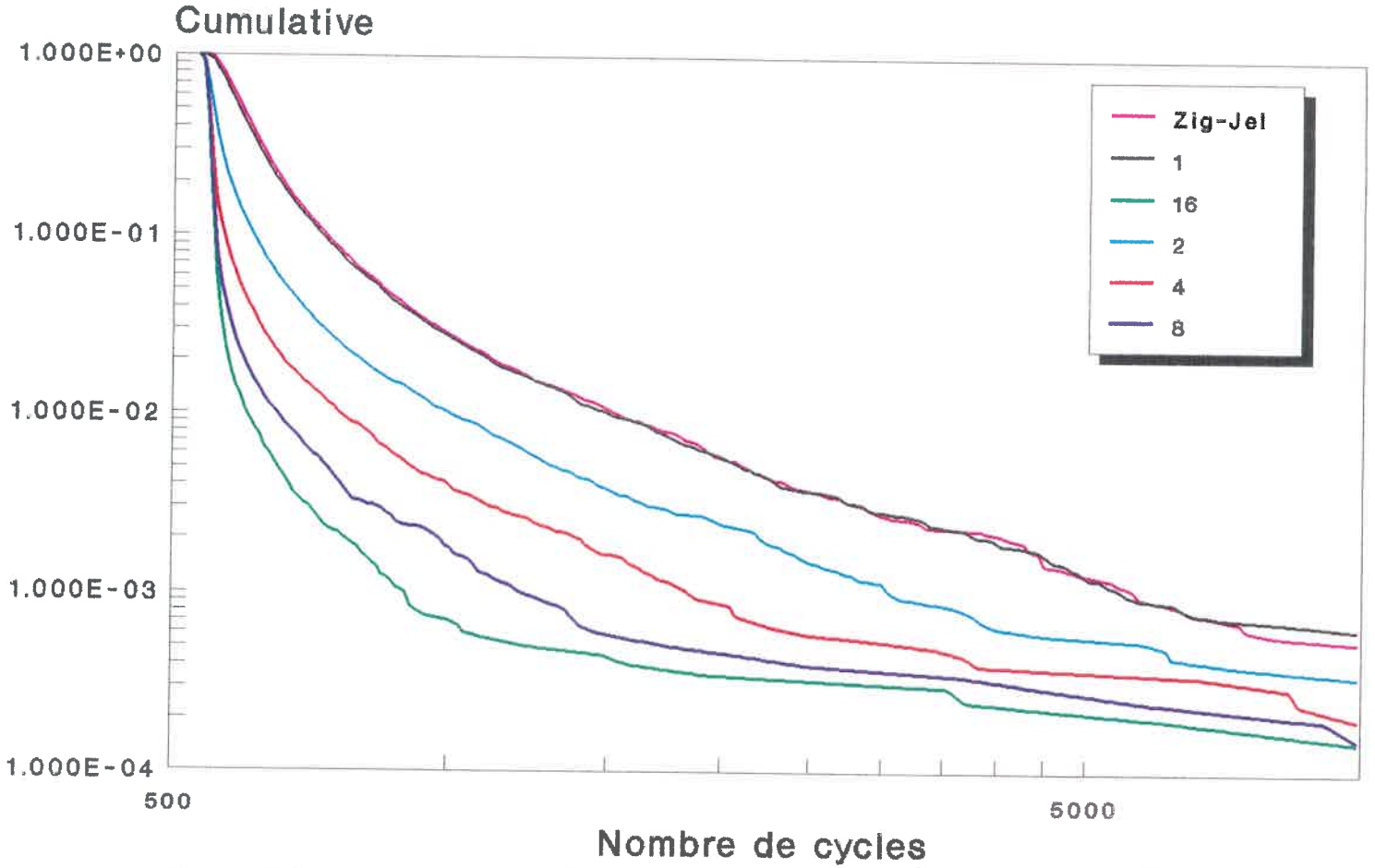


Figure D.2: Cumulative de l'effort de calcul selon le nombre de cycles $E_b/N_0 = 3.322dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

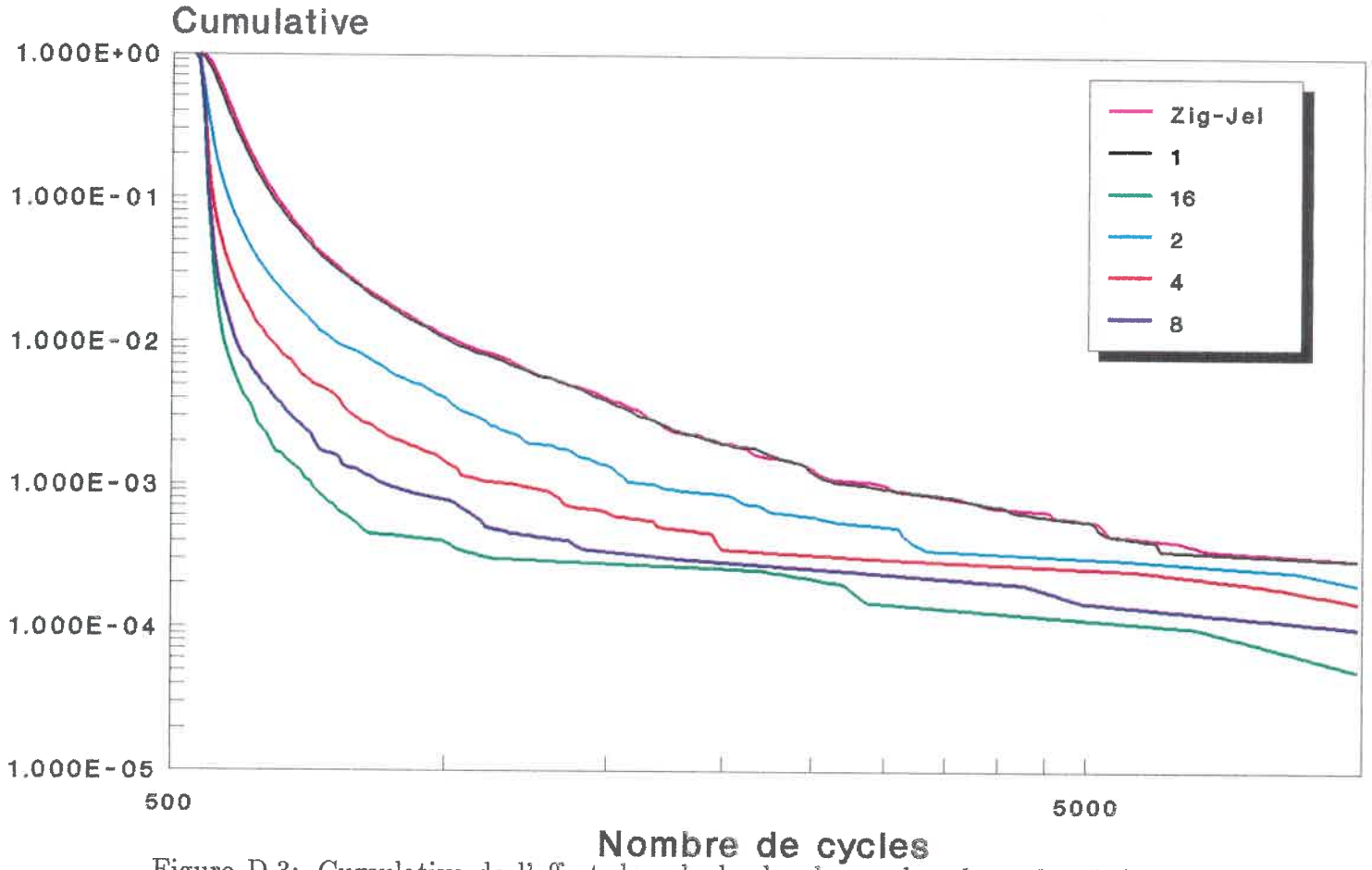


Figure D.3: Cumulative de l'effort de calcul selon le nombre de cycles $E_b/N_0 = 3.634dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

On voit sur les figures D.7, D.8 et D.9 que le "plateau" où commence à se produire des erreurs de décodage ne se déplace pas au même taux selon le nombre de processeurs. De plus, comme prévu, les taux d'erreurs s'éloignent en fonction de l'écart entre les plateaux tel que montré au tableau D.2.

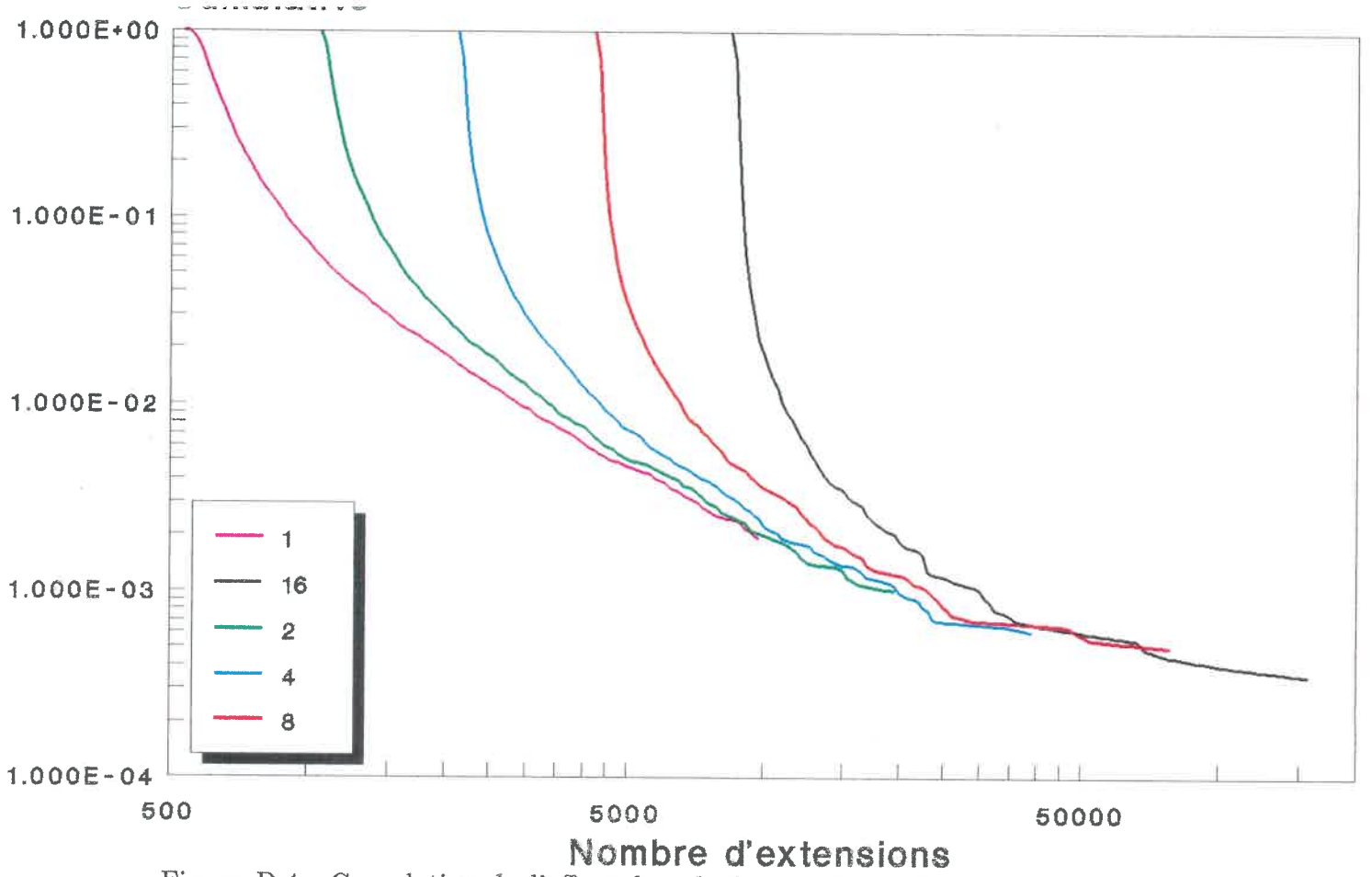


Figure D.4: Cumulative de l'effort de calcul selon le nombre d'extensions pour $E_b/N_0 = 3.010dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

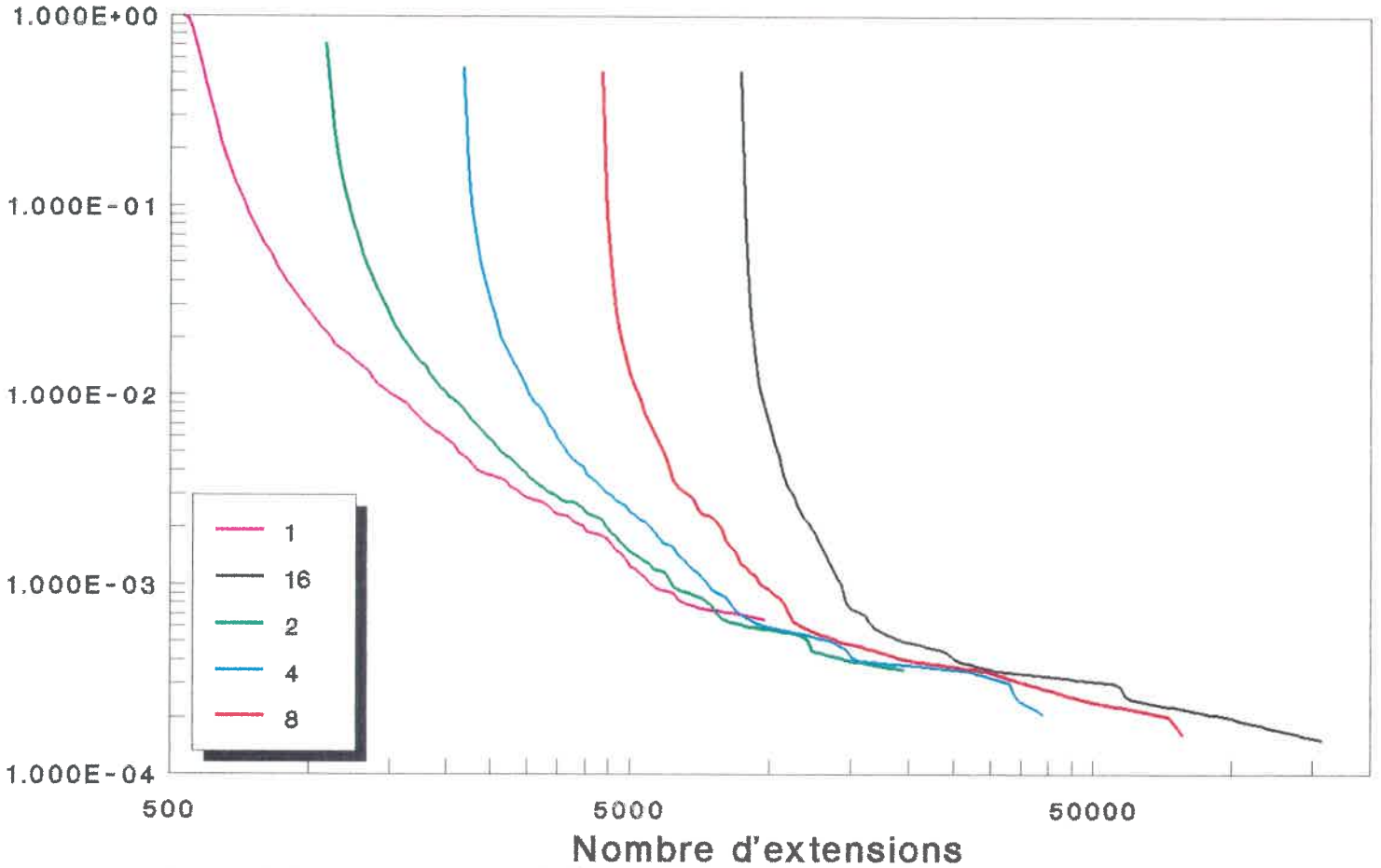


Figure D.5: Cumulative de l'effort de calcul selon le nombre d'extensions pour $E_b/N_0 = 3.322dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

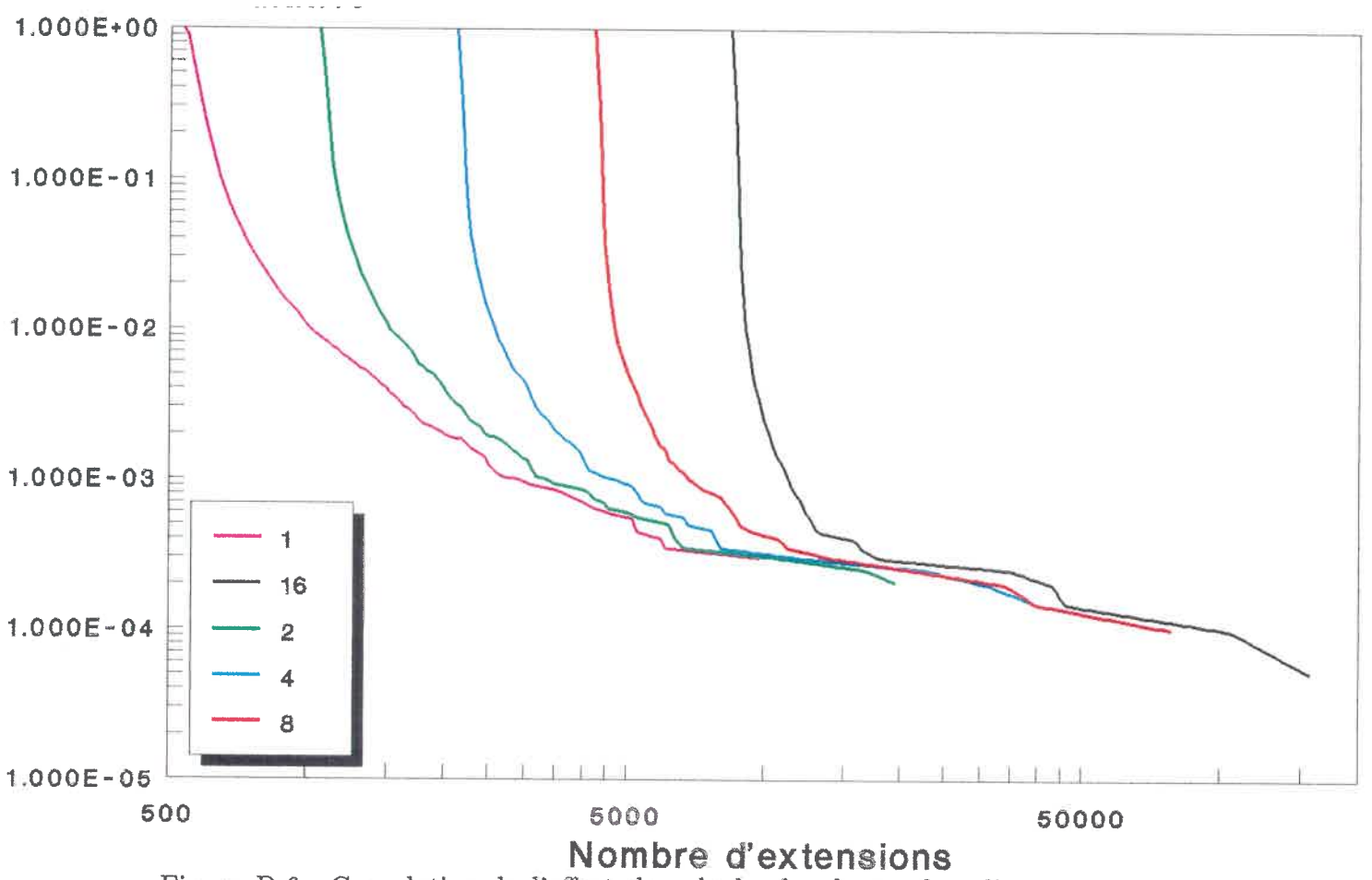


Figure D.6: Cumulative de l'effort de calcul selon le nombre d'extensions pour $E_b/N_0 = 3.634dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 20000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

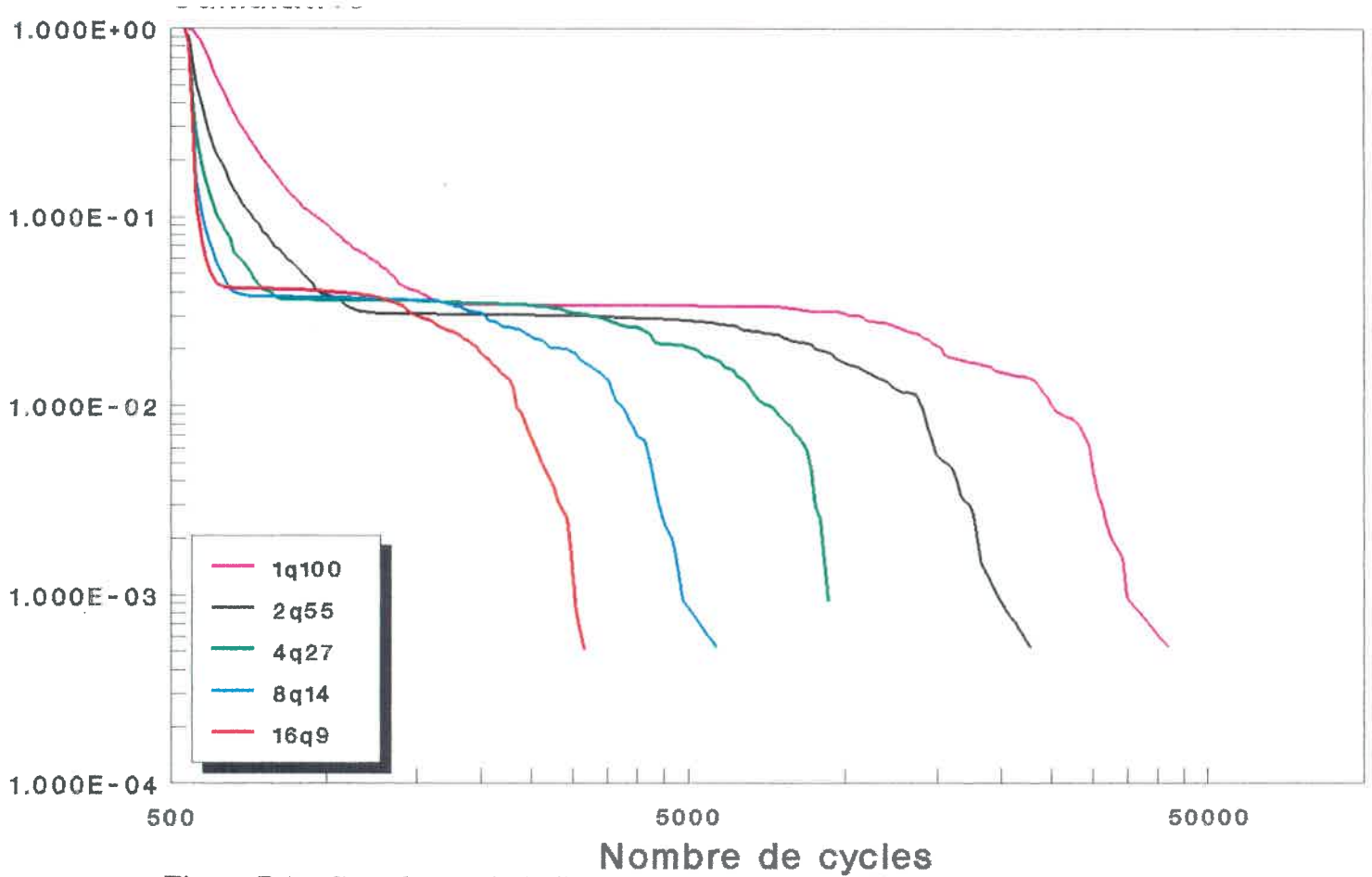


Figure D.7: Cumulative de l'effort de calcul pour une file de profondeur variée et $E_b/N_0 = 3.010dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de l'historique = 100000

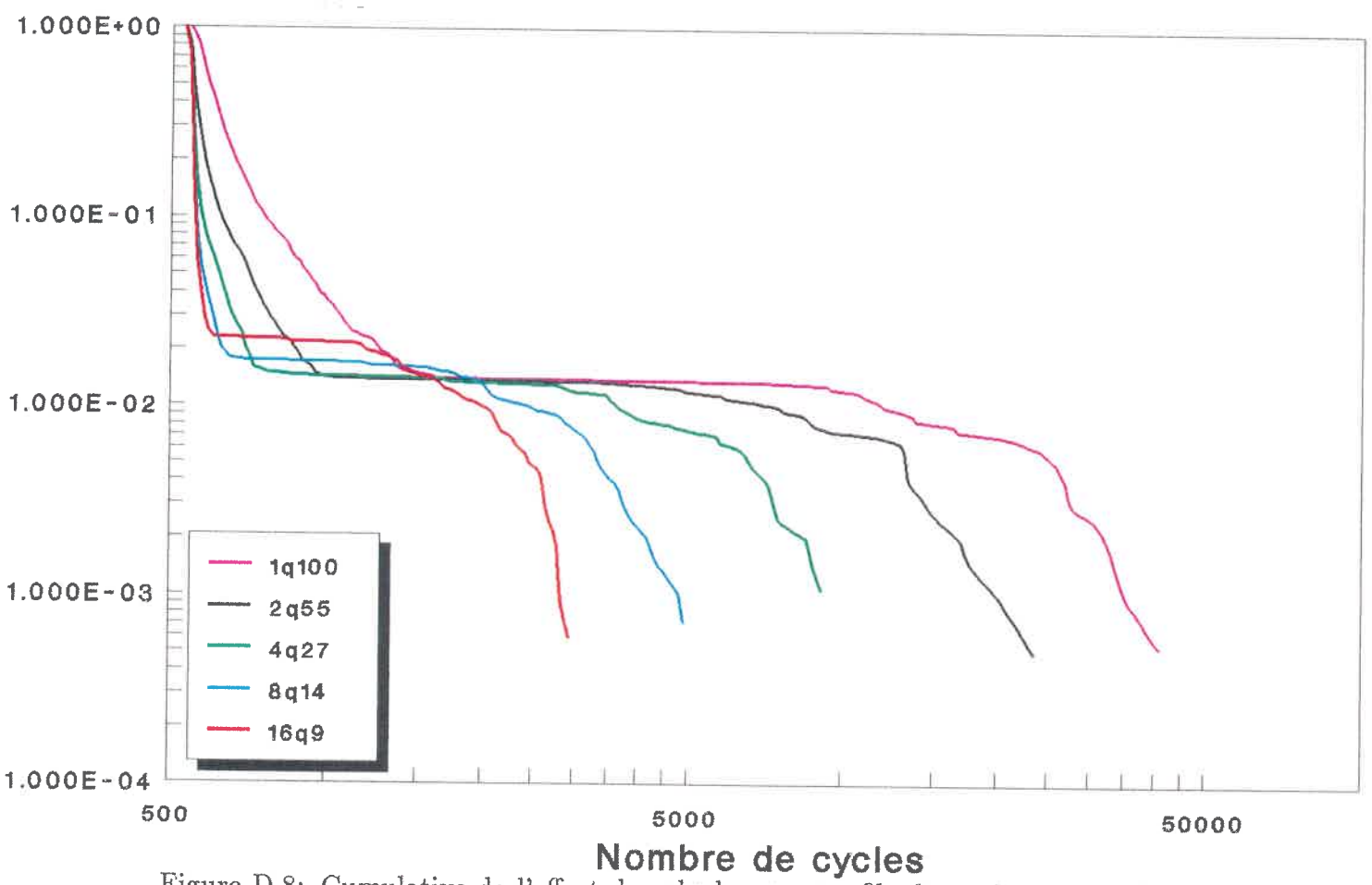


Figure D.8: Cumulative de l'effort de calcul pour une file de profondeur variée et $E_b/N_0 = 3.322dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de l'historique = 100000

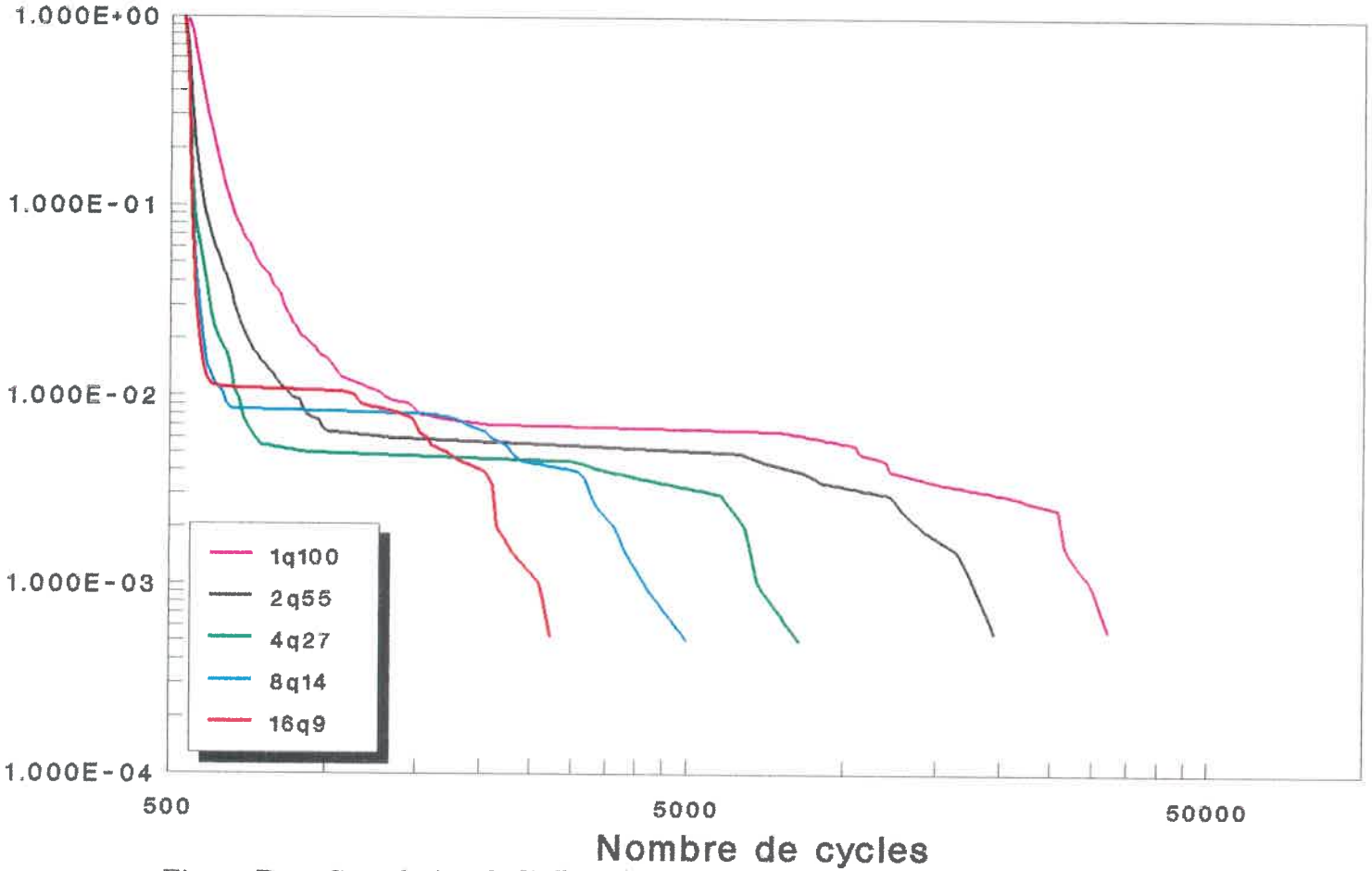


Figure D.9: Cumulative de l'effort de calcul pour une file de profondeur variée et $E_b/N_0 = 3.634dB$

Architecture linéaire
k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de l'historique = 100000

n. de proc.	$E_b/N_0(dB)$		
	3.010	3.322	3.634
1	742.9	643.8	595.9
2	628.0	578.5	557.6
4	582.4	556.9	546.6
8	564.5	549.4	542.9
16	557.0	546.9	541.6

Tableau D.1: Nombre moyen d'itérations pour différents nombres de processeurs

architecture linéaire

n. de proc.	$E_b/N_0(dB)$		
	3.010	3.322	3.634
1	0.0071	0.0030	0.0012
2	0.0067	0.0029	0.0011
4	0.0083	0.0029	0.0010
8	0.0090	0.0034	0.0016
16	0.0099	0.0050	0.0020

Tableau D.2: Taux d'erreurs en fonction du nombre de processeurs et du rapport signal sur bruit

Architecture linéaire

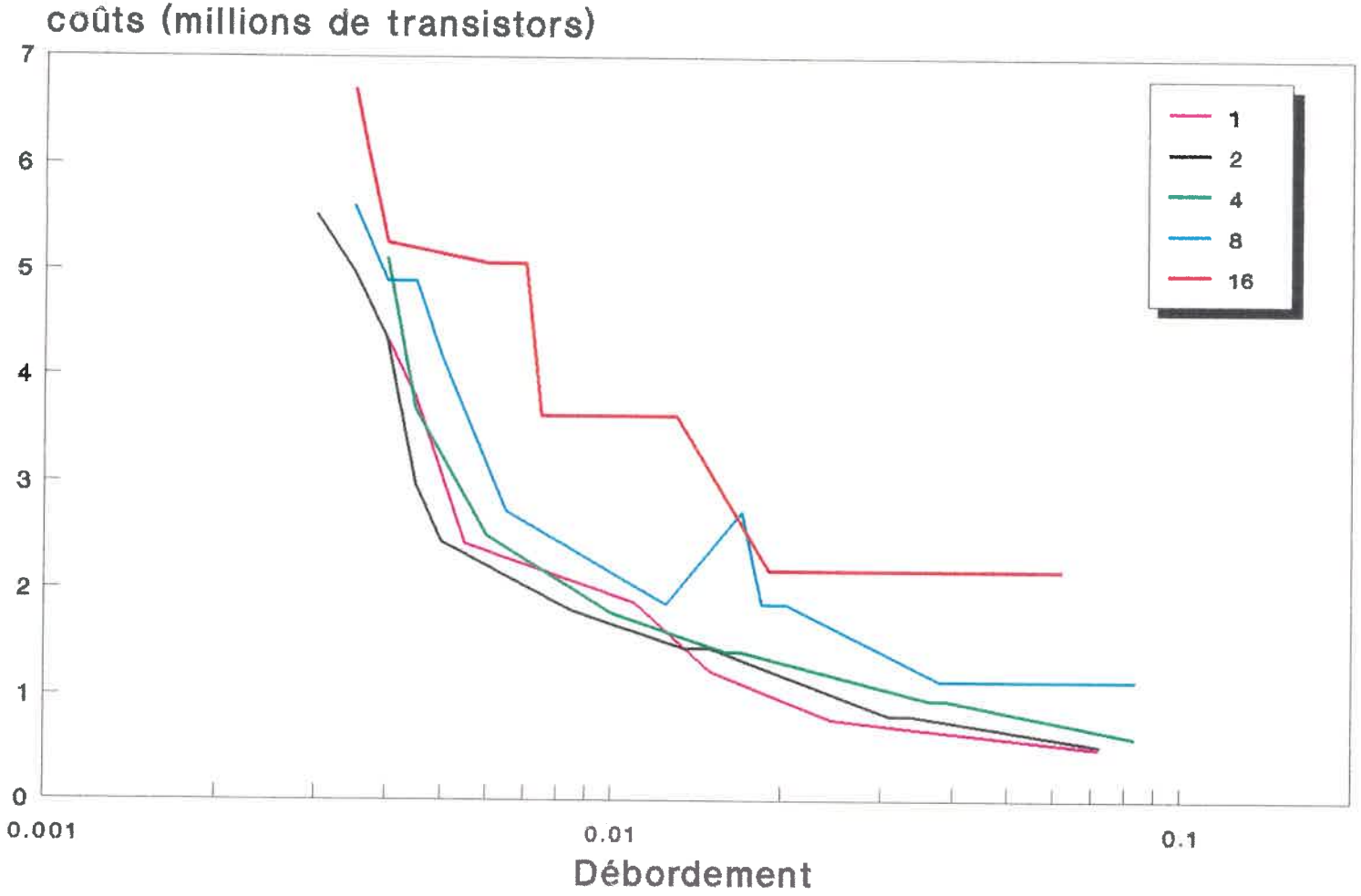


Figure D.10: Coûts en fonction du taux de débordement selon le modèle "A" $E_b/N_0 = 3.010dB$

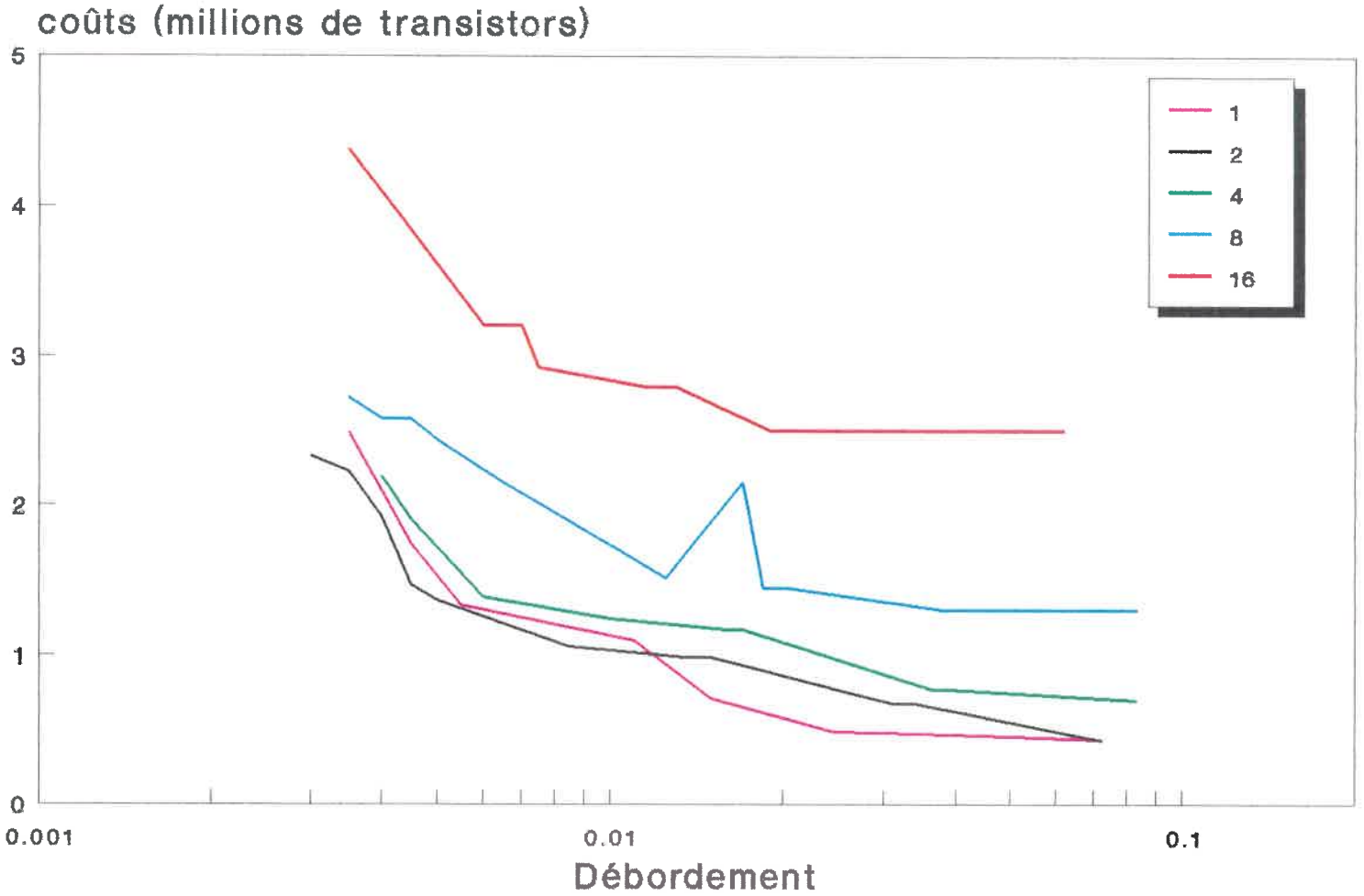


Figure D.11: Coûts en fonction du taux de débordement selon le modèle "B" $E_b/N_0 = 3.010dB$

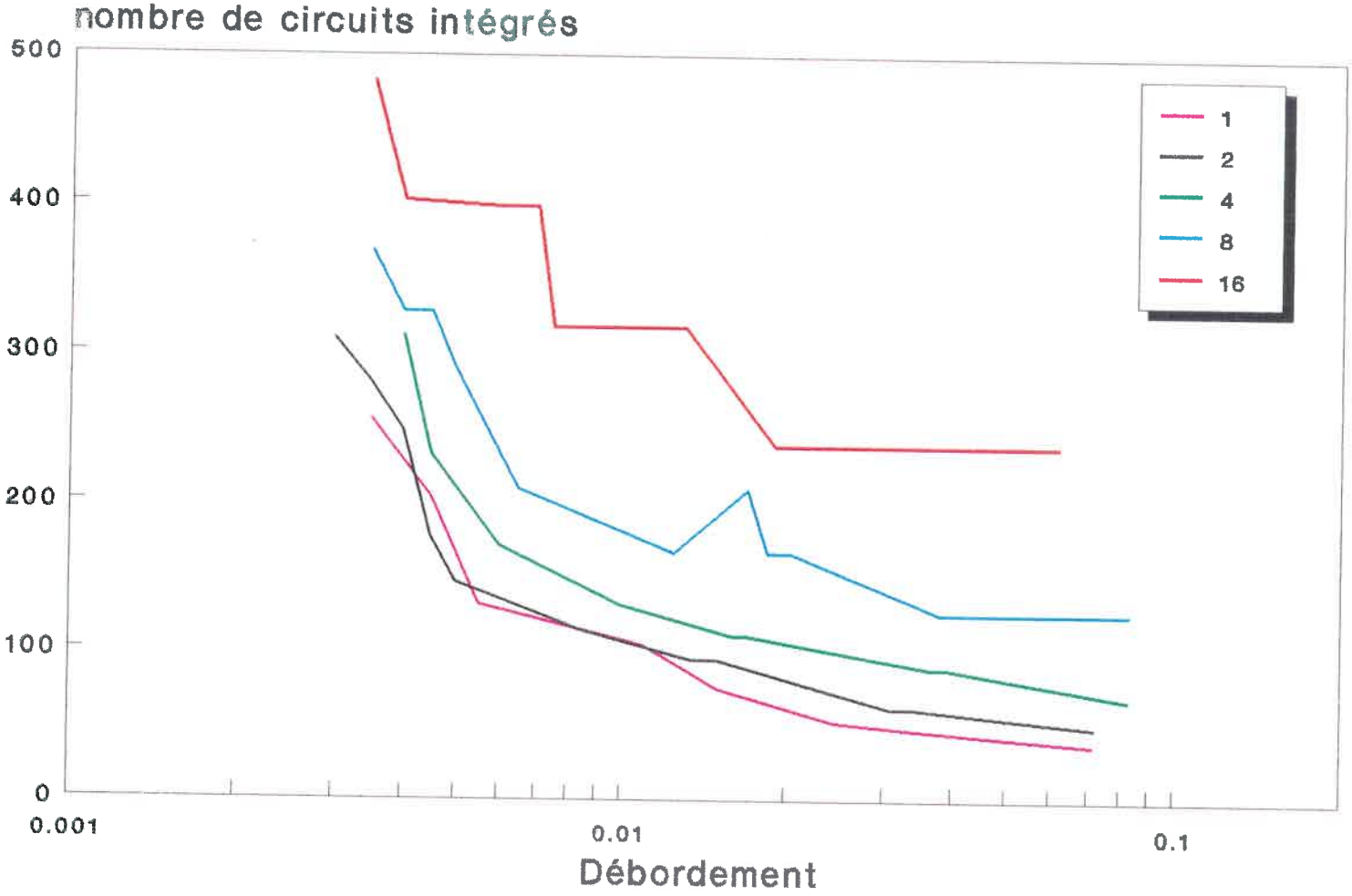


Figure D.12: Coûts en fonction du taux de débordement selon le modèle "C" $E_b/N_0 = 3.010dB$

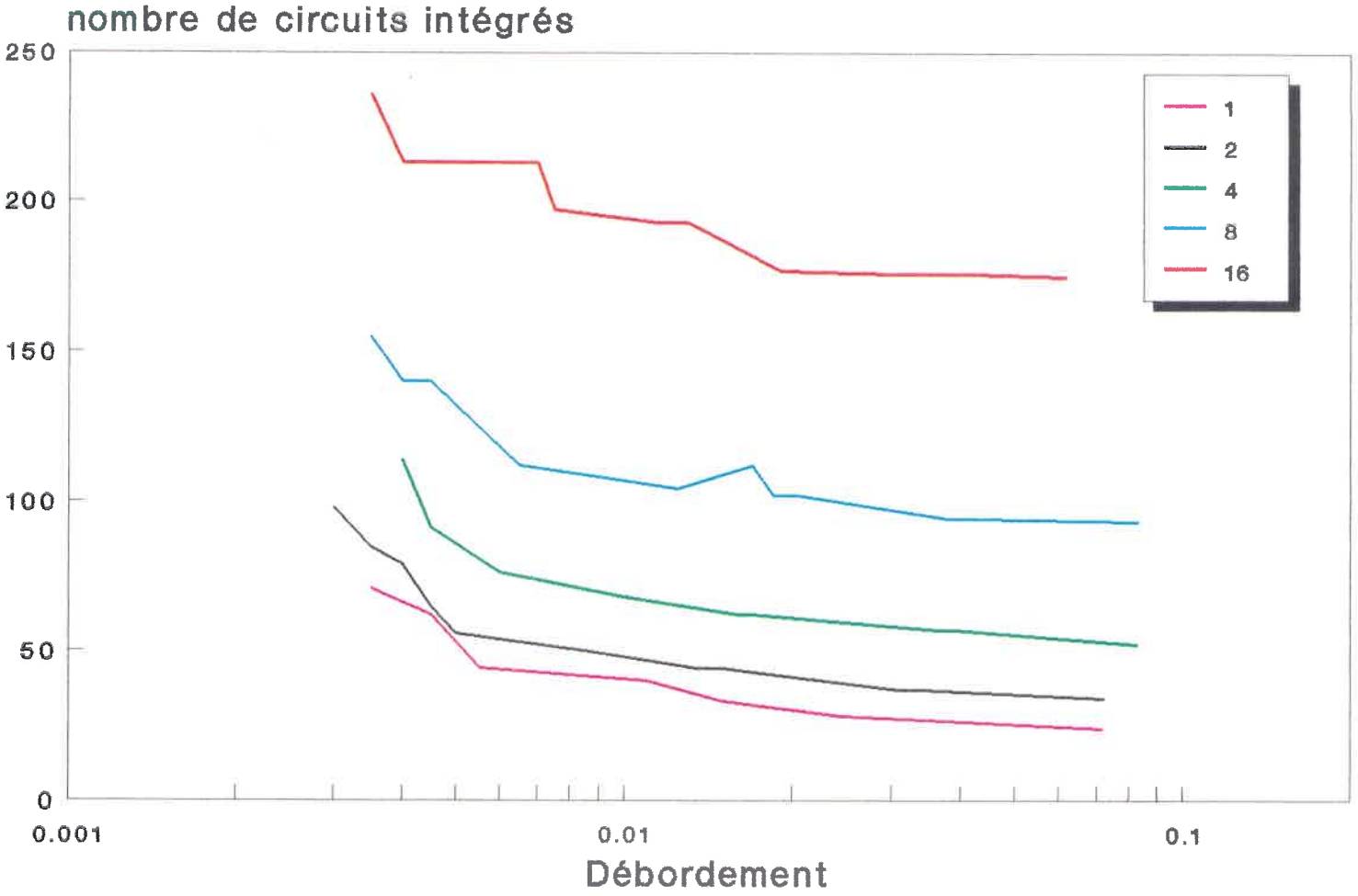


Figure D.13: Coûts en fonction du taux de débordement selon le modèle "D" $E_b/N_0 = 3.010dB$

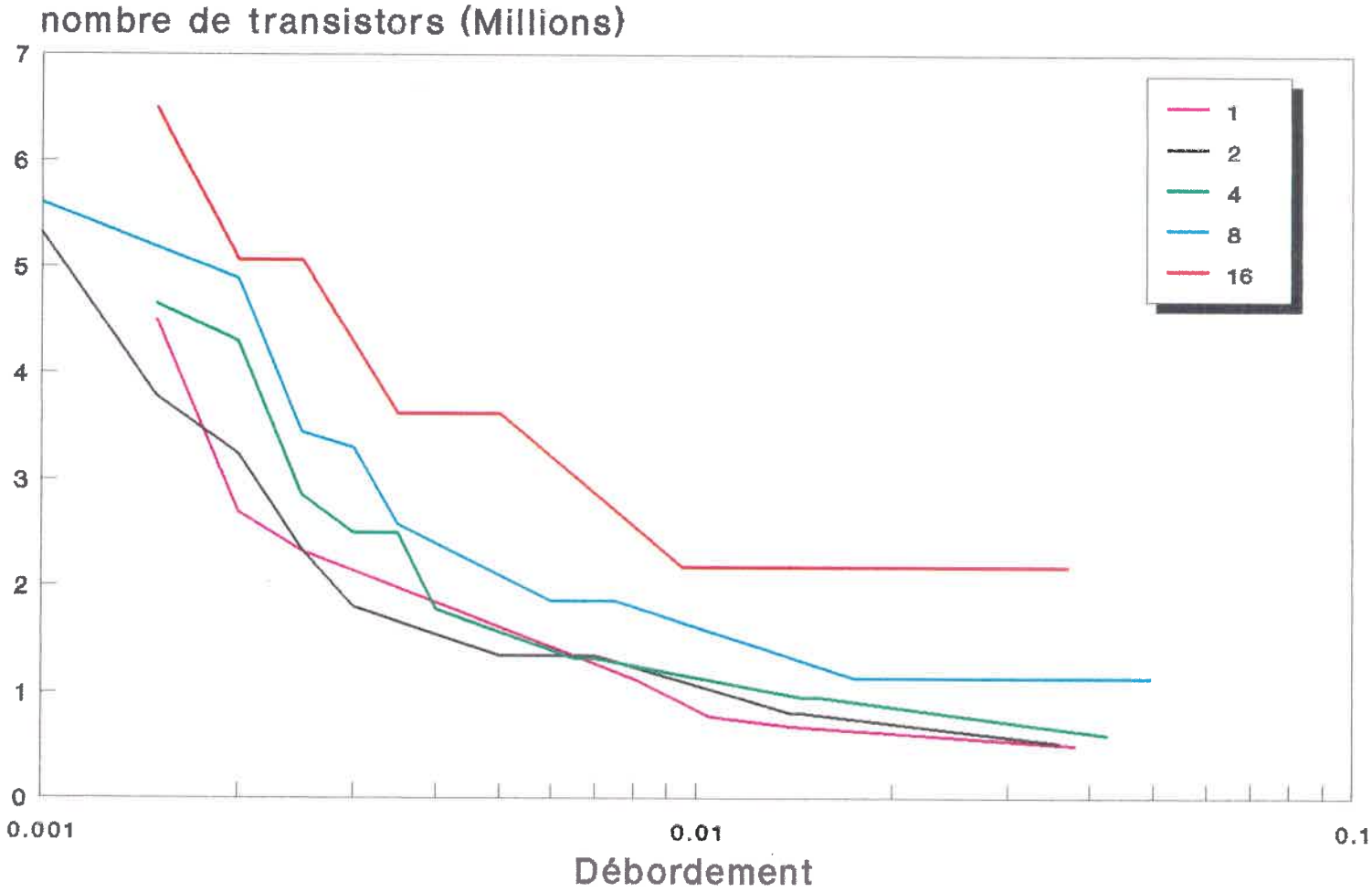


Figure D.14: Coûts en fonction du taux de débordement selon le modèle "A" $E_b/N_0 = 3.322dB$

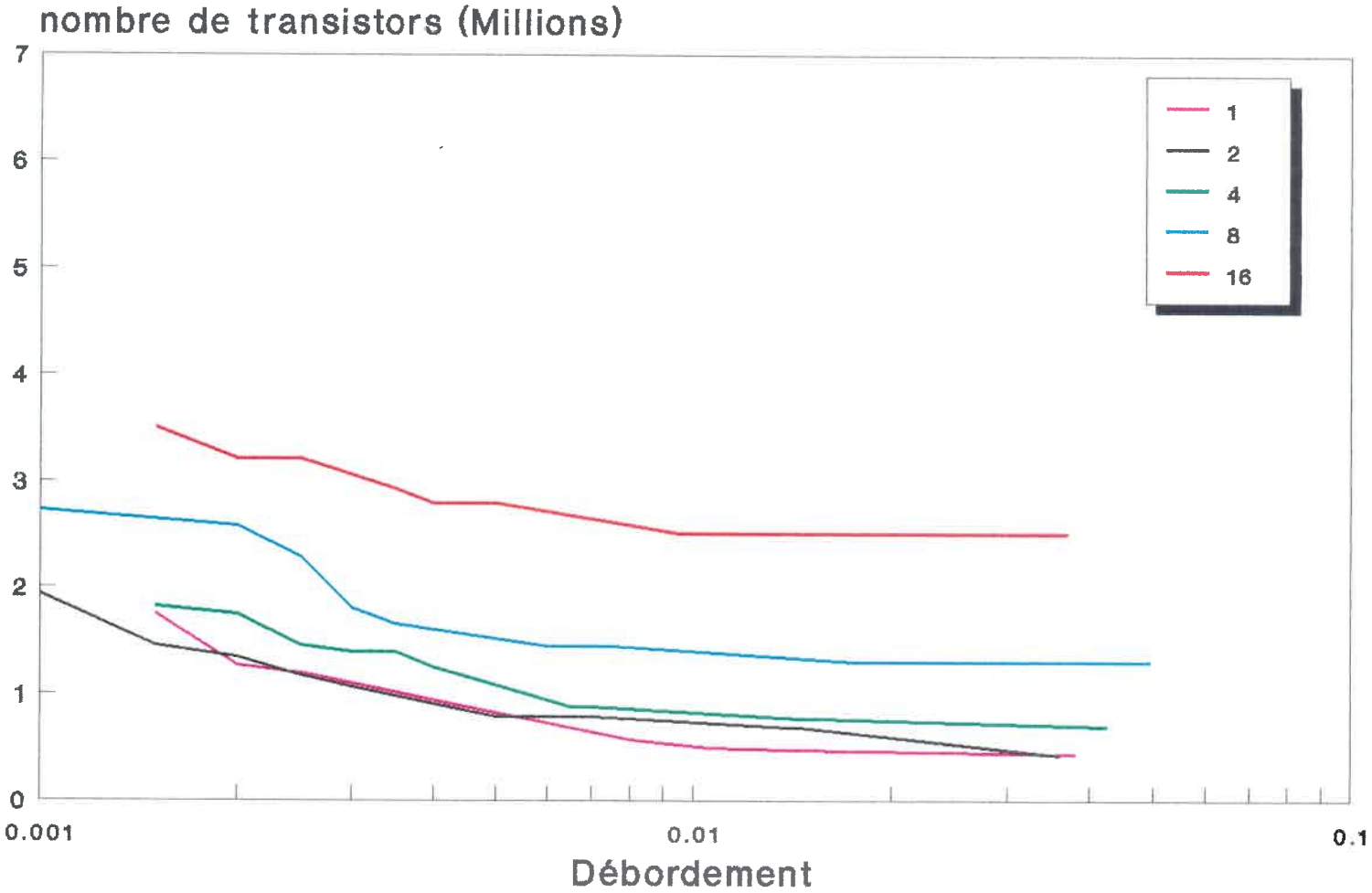


Figure D.15: Coûts en fonction du taux de débordement selon le modèle "B" $E_b/N_0 = 3.322dB$

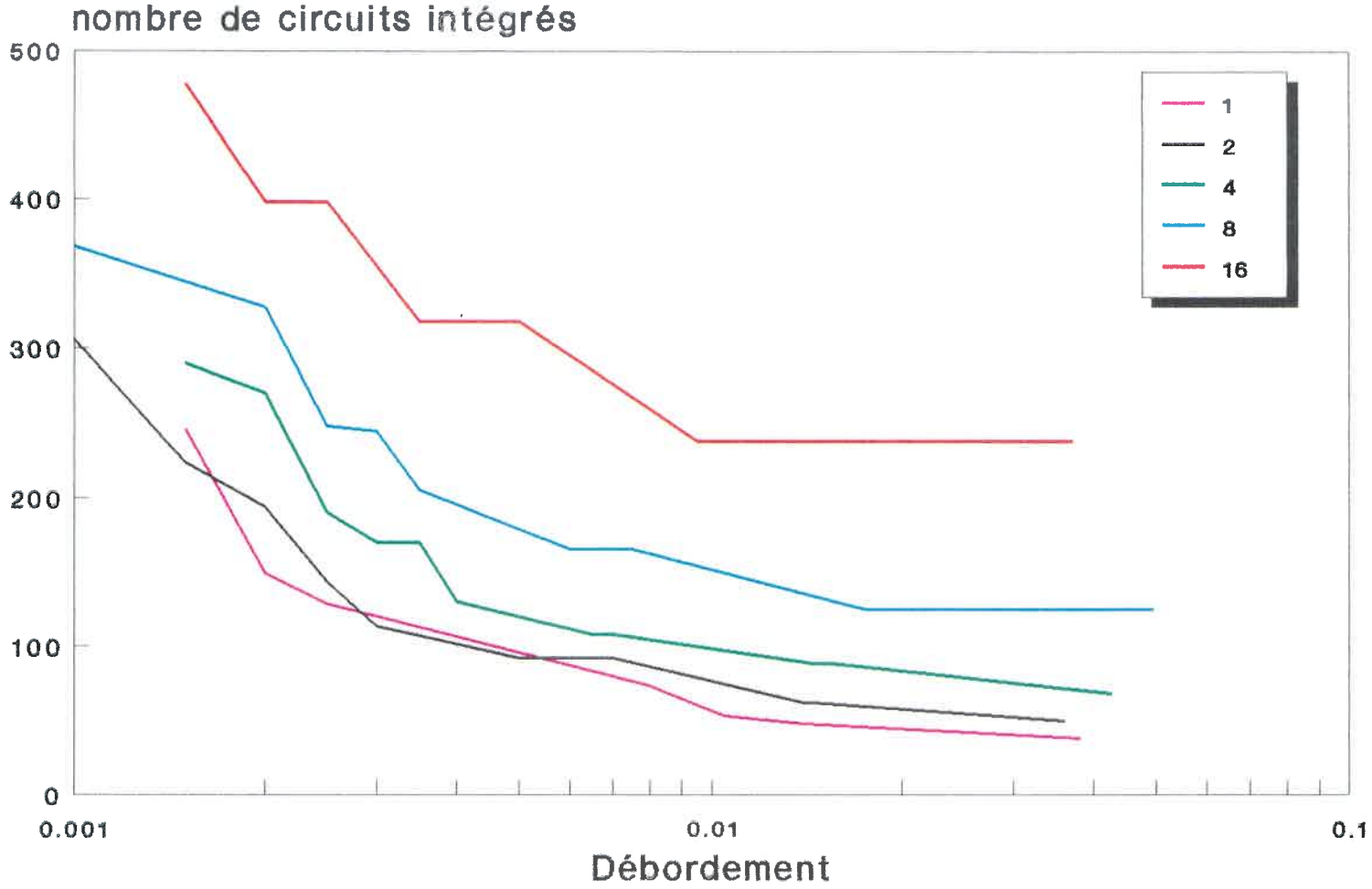


Figure D.16: Coûts en fonction du taux de débordement selon le modèle "C" $E_b/N_0 = 3.322dB$

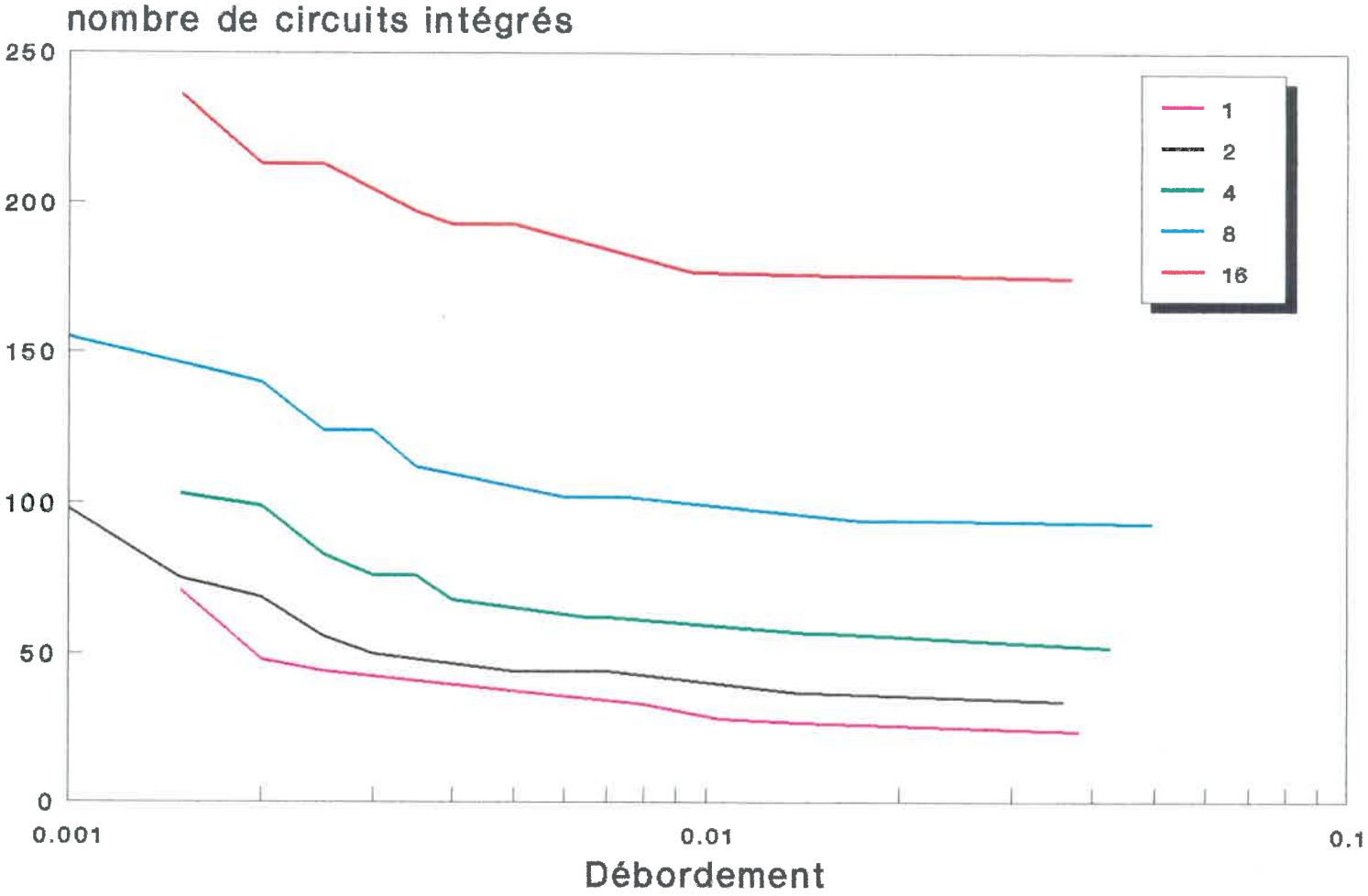


Figure D.17: Coûts en fonction du taux de débordement selon le modèle "D"
$E_b/N_0 = 3.322dB$

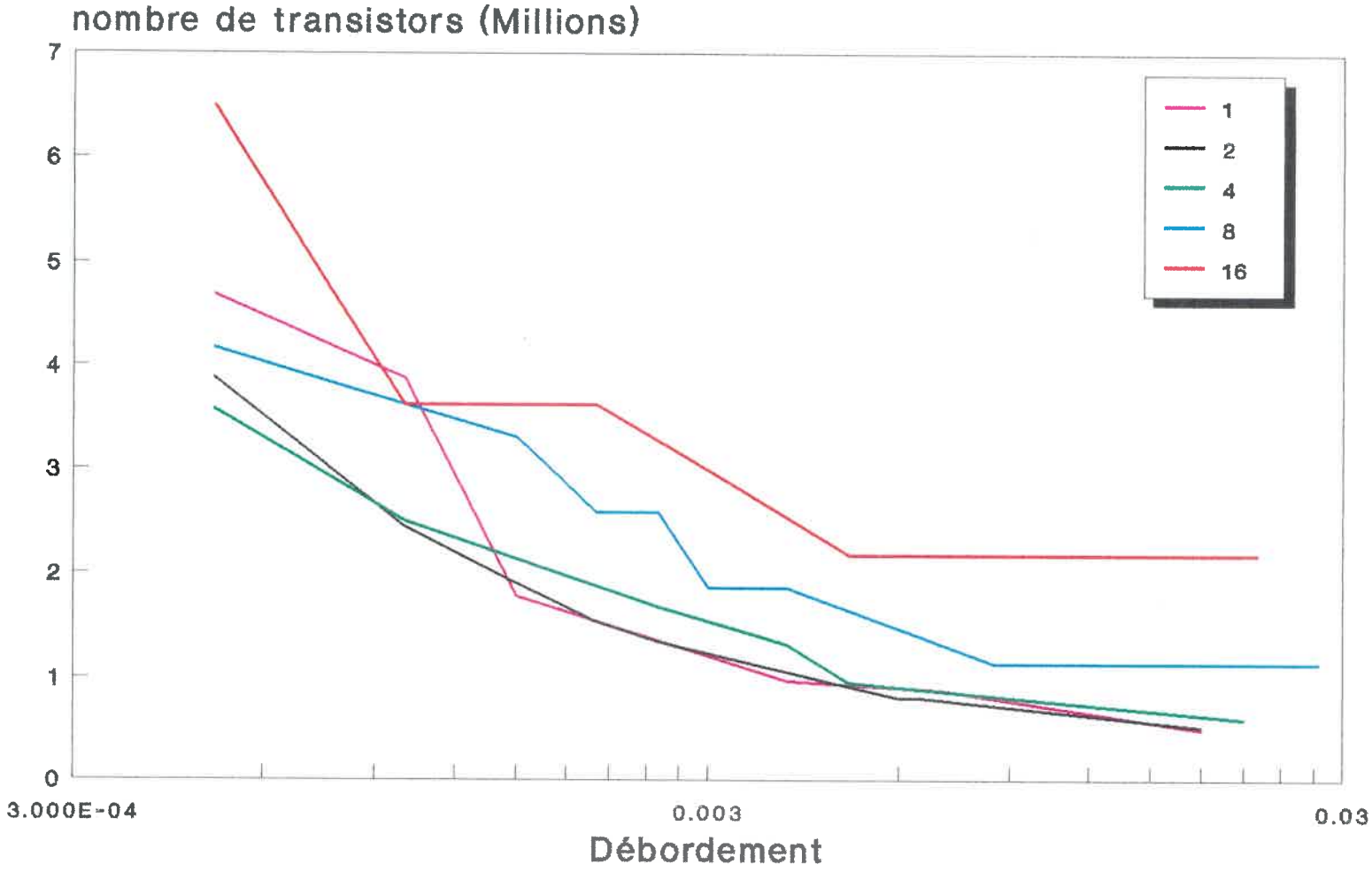


Figure D.18: Coûts en fonction du taux de débordement selon le modèle "A"
$E_b/N_0 = 3.634dB$

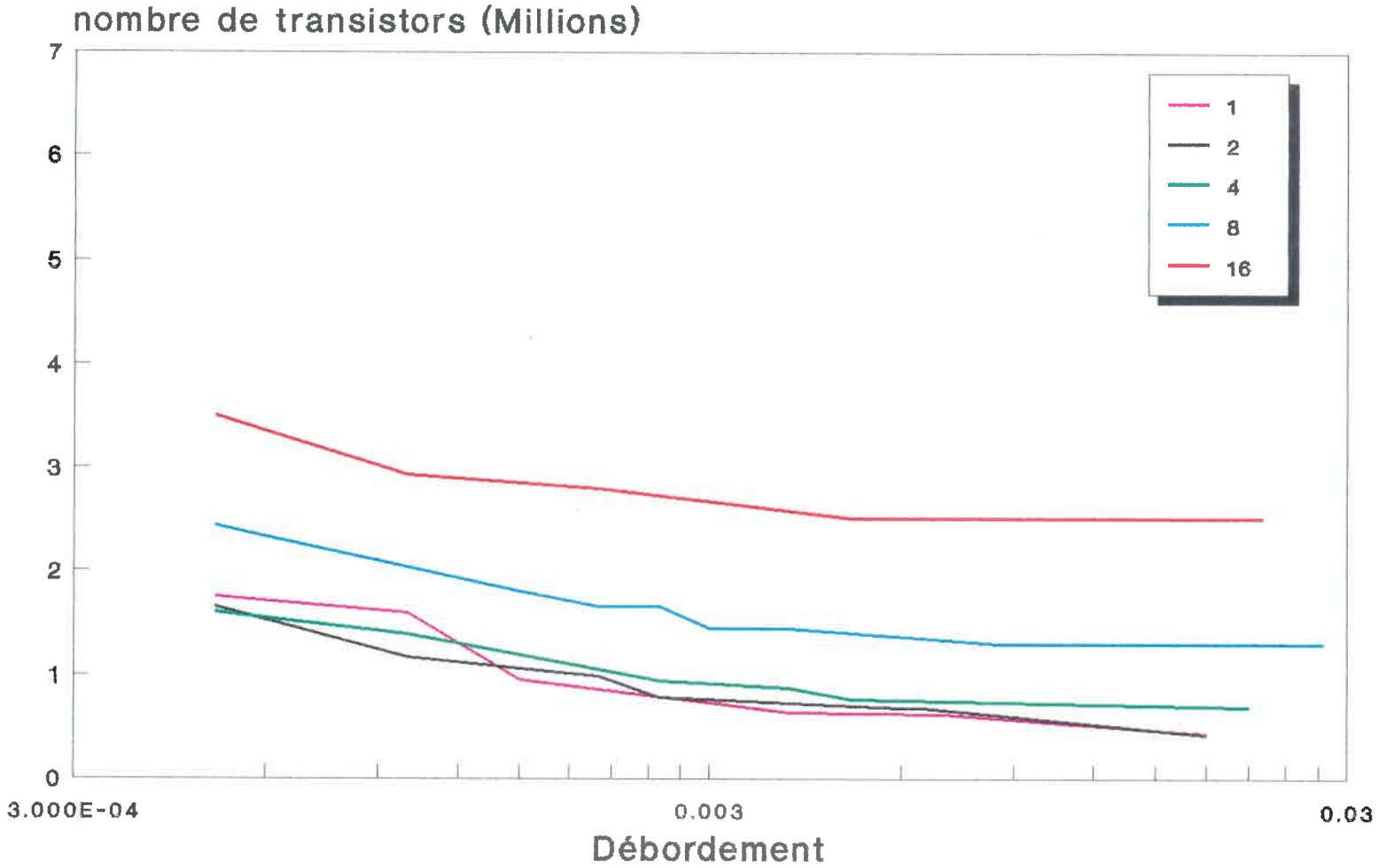


Figure D.19: Coûts en fonction du taux de débordement selon le modèle "B" $E_b/N_0 = 3.634dB$

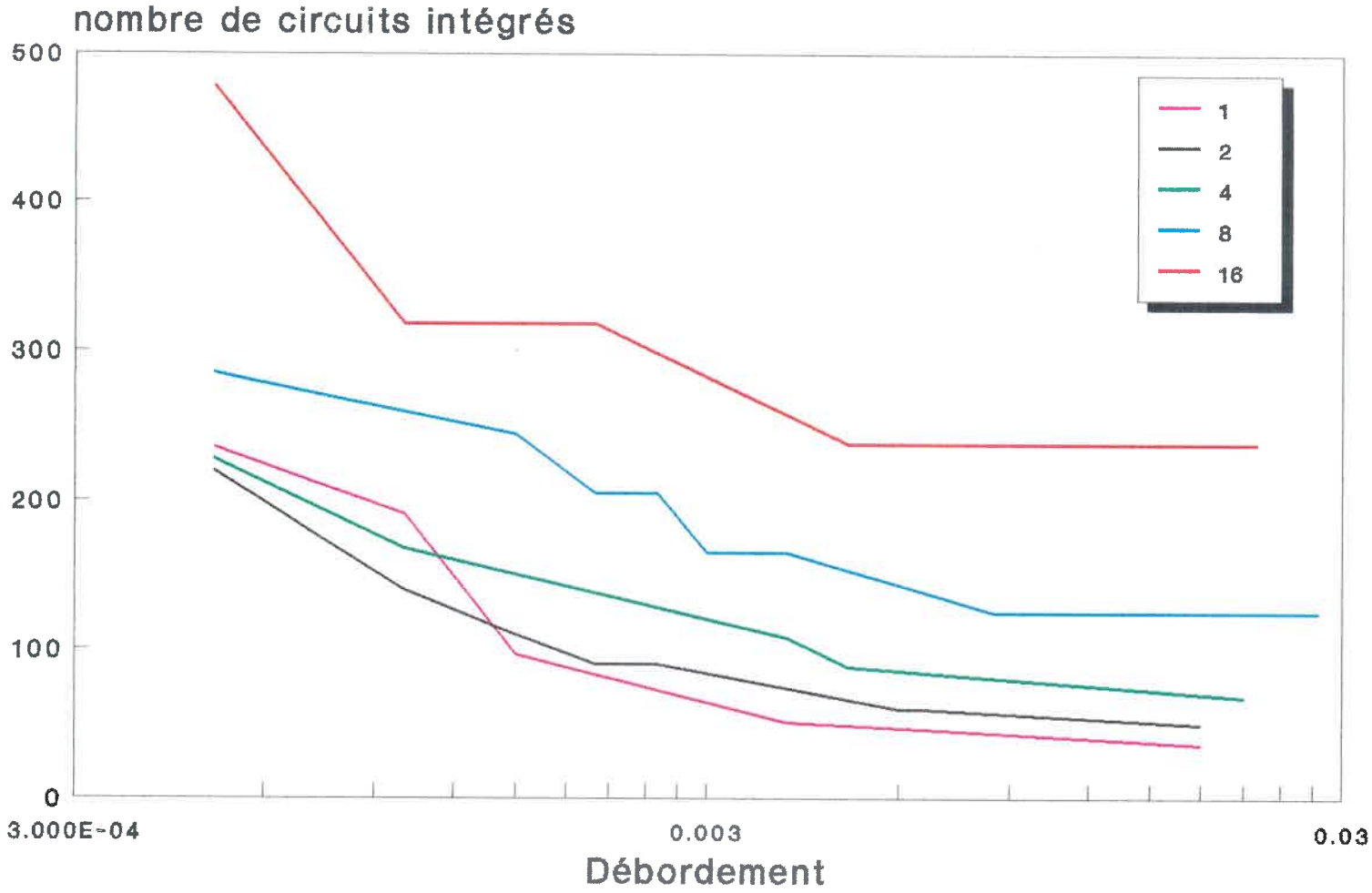


Figure D.20: Coûts en fonction du taux de débordement selon le modèle "C" $E_b/N_0 = 3.634dB$

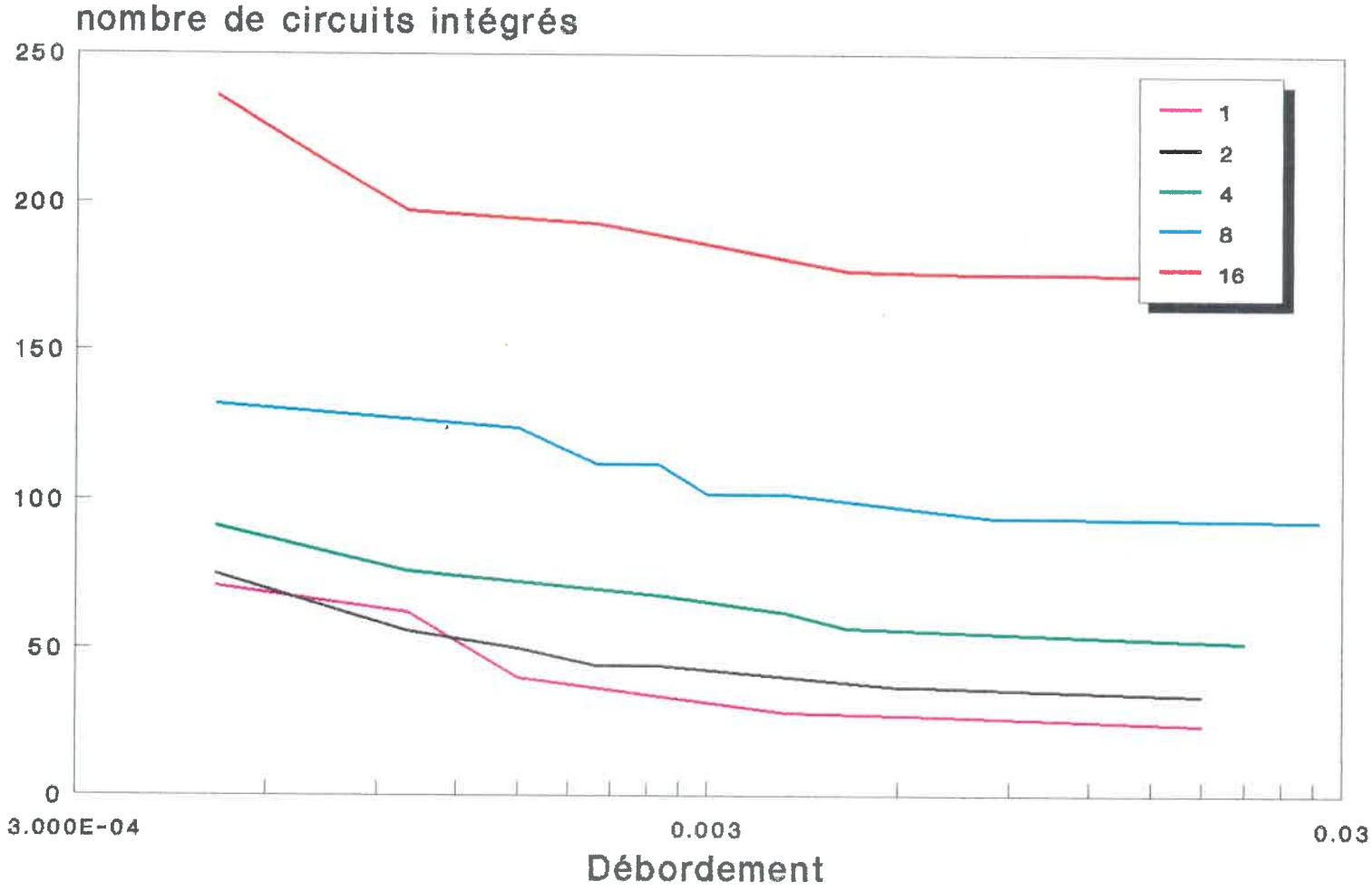


Figure D.21: Coûts en fonction du taux de débordement selon le modèle "D"
$E_b/N_0 = 3.634dB$

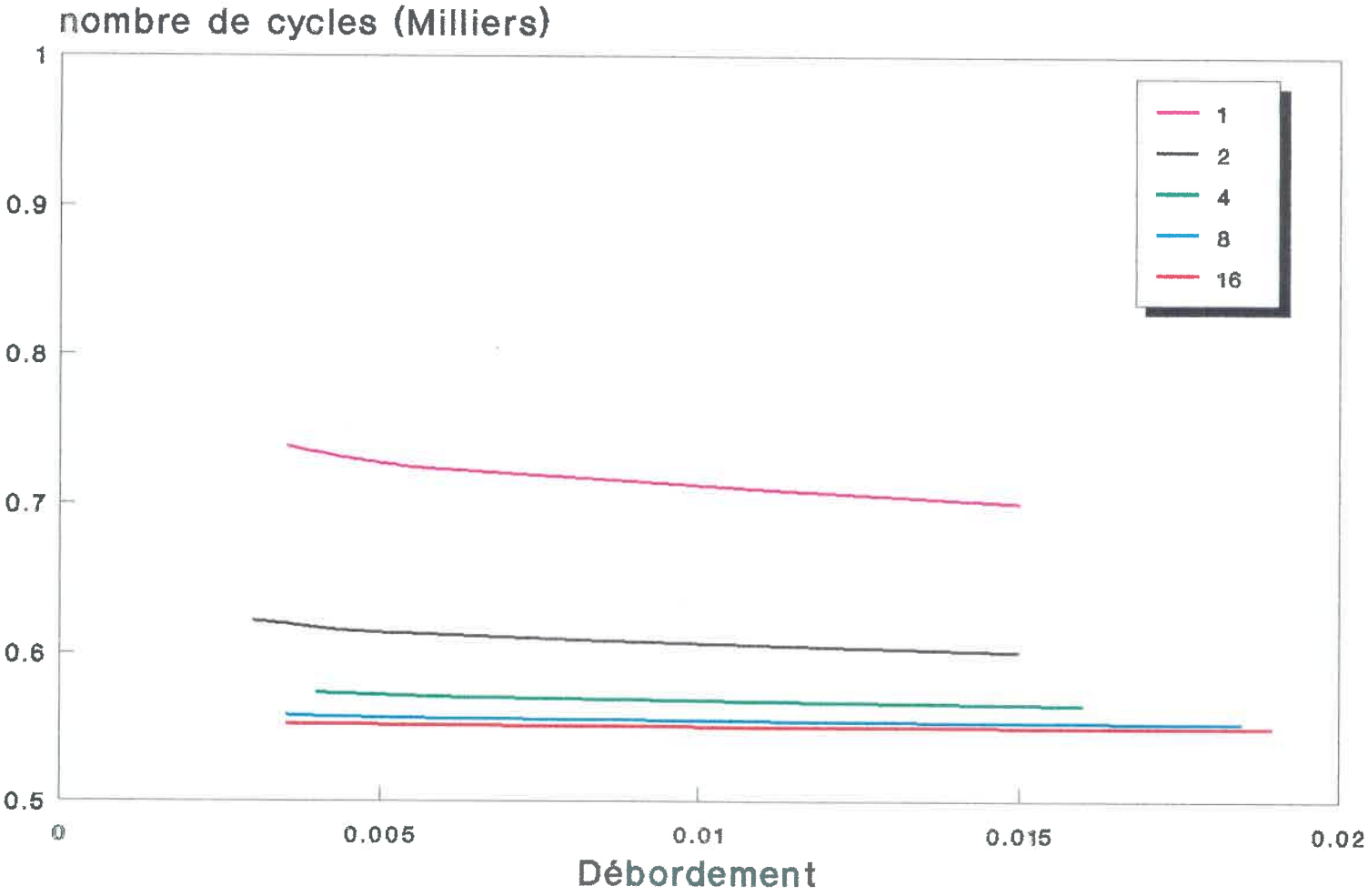


Figure D.22: Nombre moyen de cycles pour décoder un bloc de données en fonction du taux de débordement $E_b/N_0 = 3.010dB$

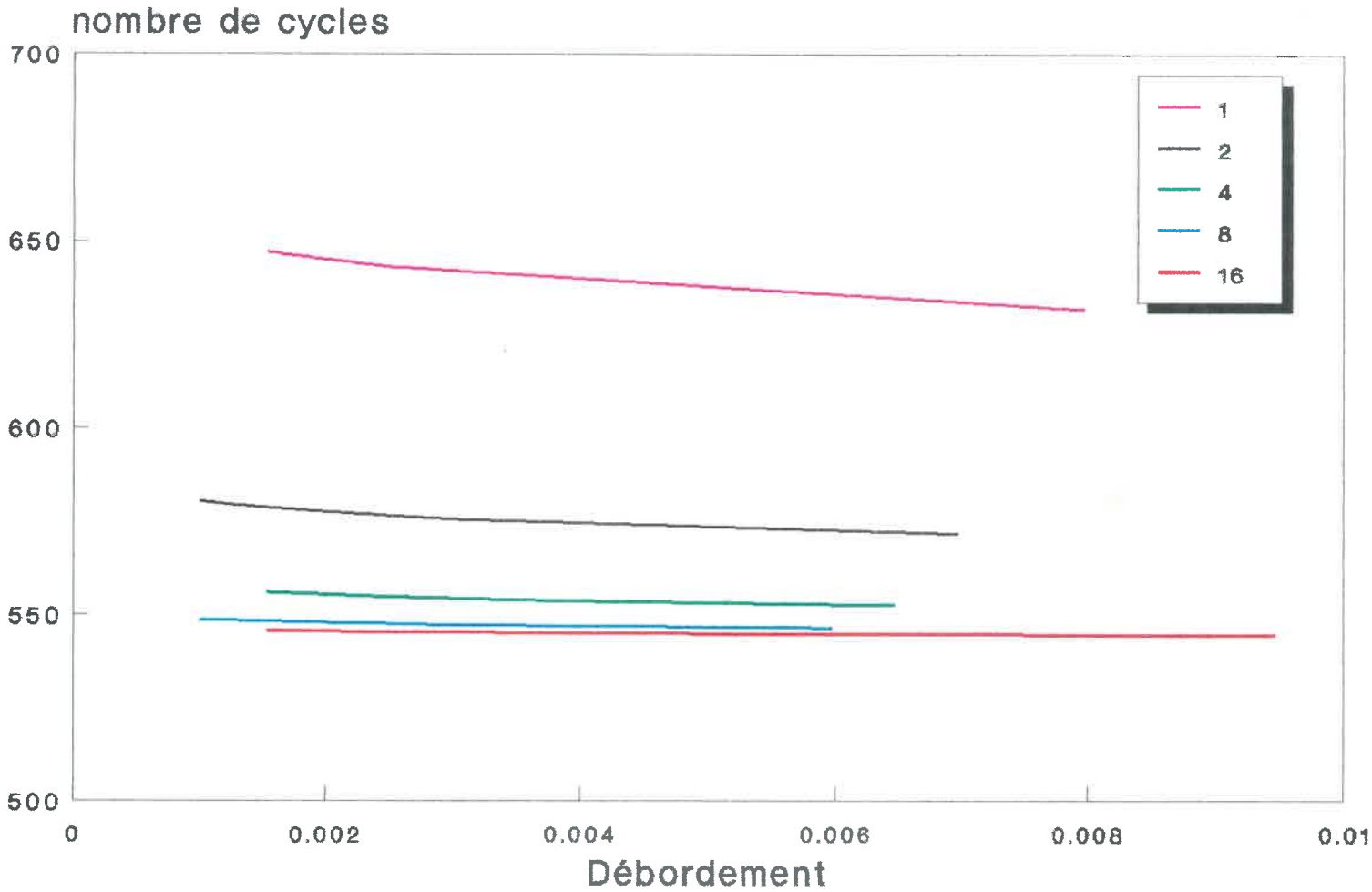


Figure D.23: Nombre moyen de cycles pour décoder un bloc de données en fonction du taux de débordement $E_b/N_0 = 3.322dB$

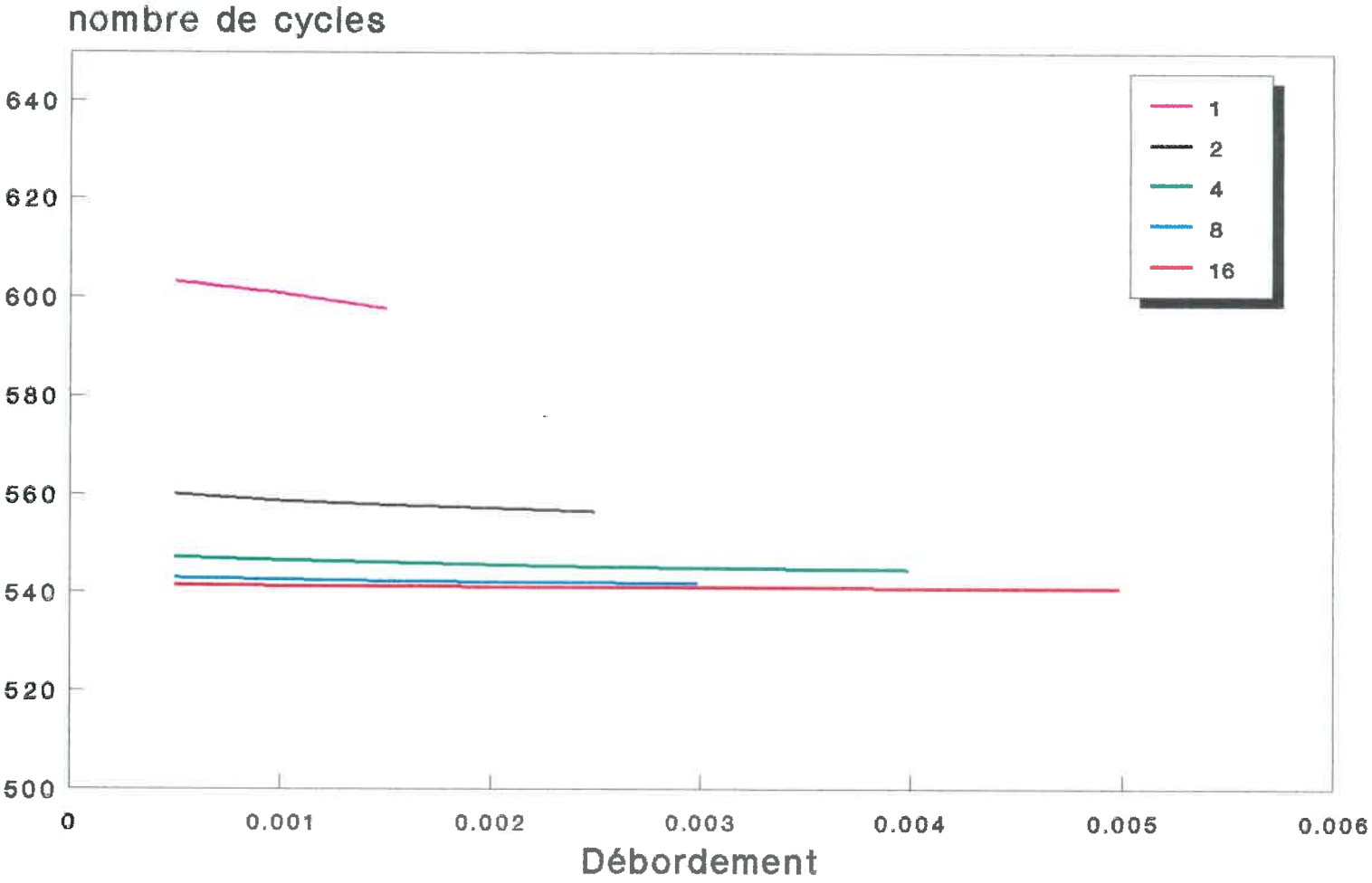


Figure D.24: Nombre moyen de cycles pour décoder un bloc de données en fonction du taux de débordement $E_b/N_0 = 3.634dB$

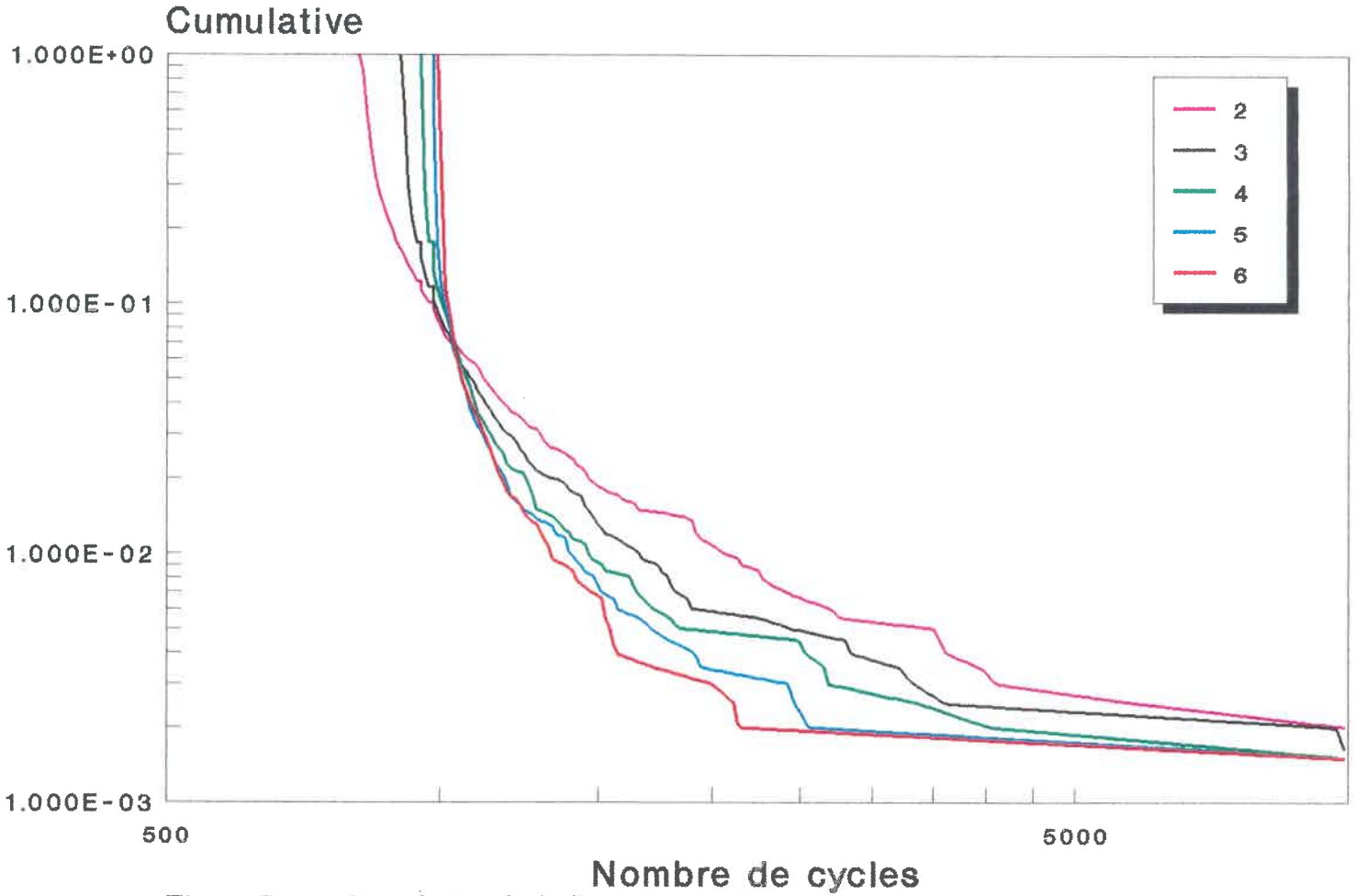


Figure D.25: Cumulative de l'effort de calcul pour l'architecture arborescente reconvergente et pour $E_b/N_0 = 3.010dB$

k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

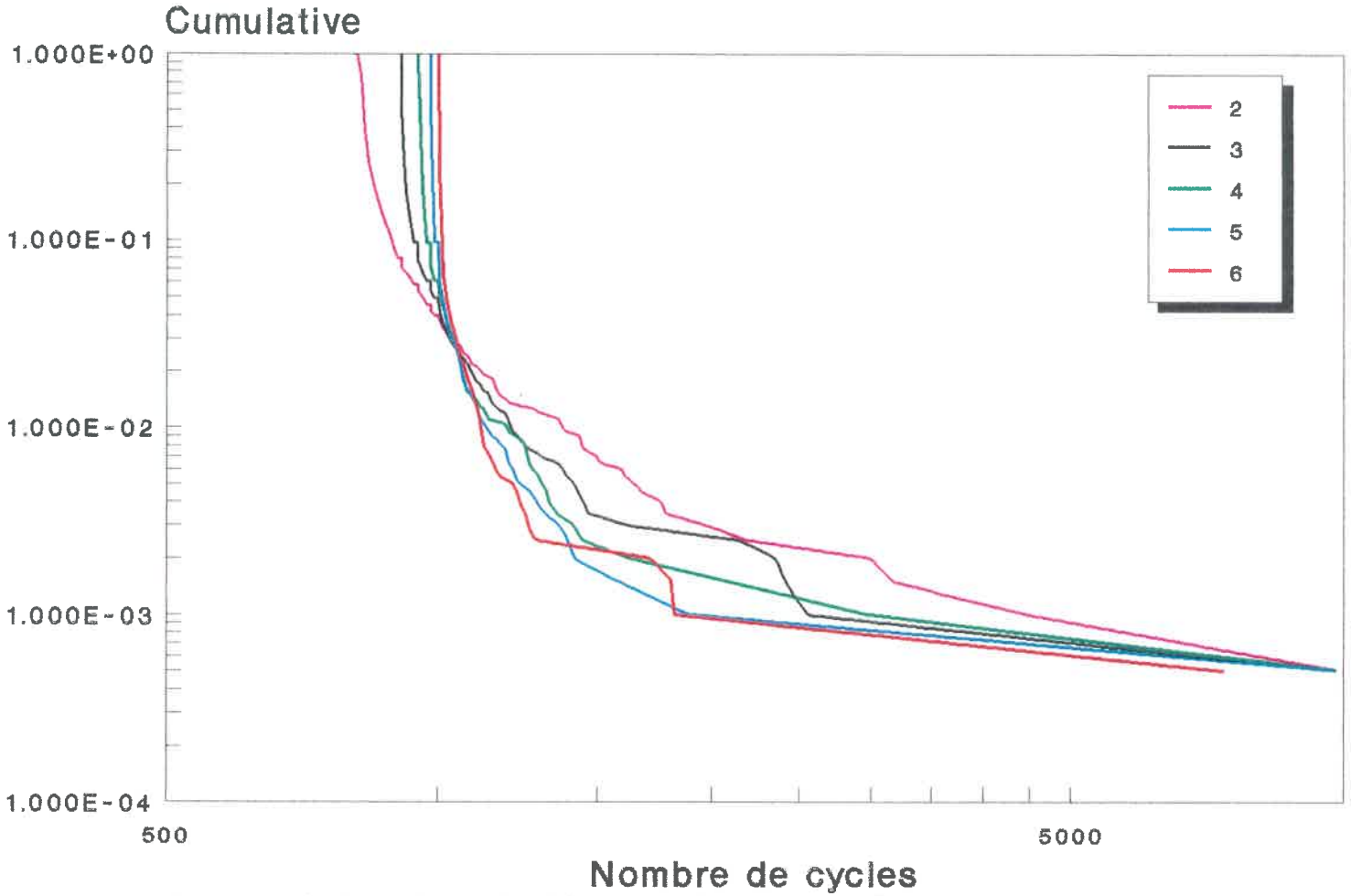


Figure D.26: Cumulative de l'effort de calcul pour l'architecture arborescente reconvergente et pour $E_b/N_0 = 3.322dB$

k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

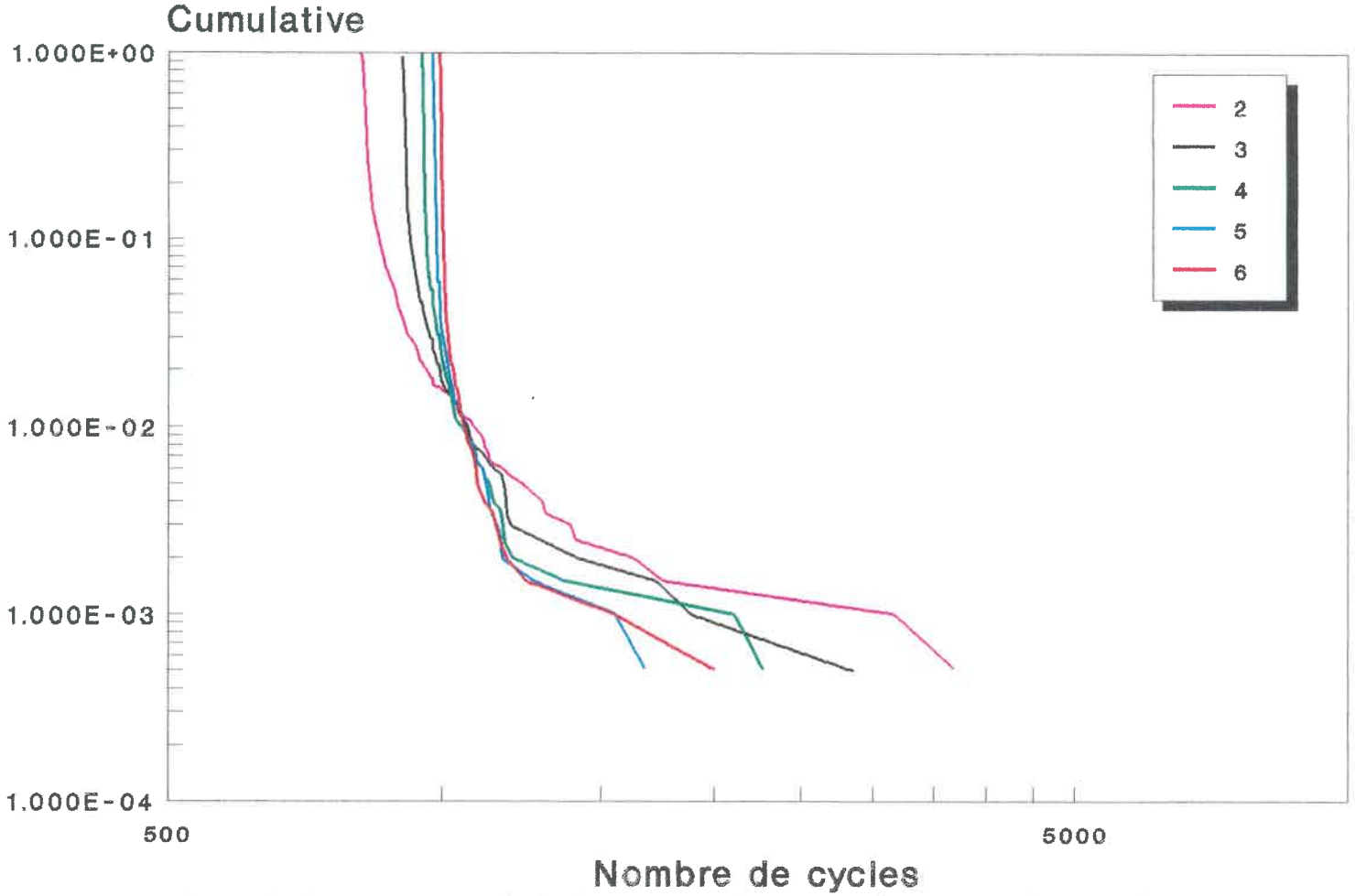


Figure D.27: Cumulative de l'effort de calcul pour l'architecture arborescente reconvergente et pour $E_b/N_0 = 3.634dB$

k = 24, code de Johannesson
nombre de blocs = 2000, nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000

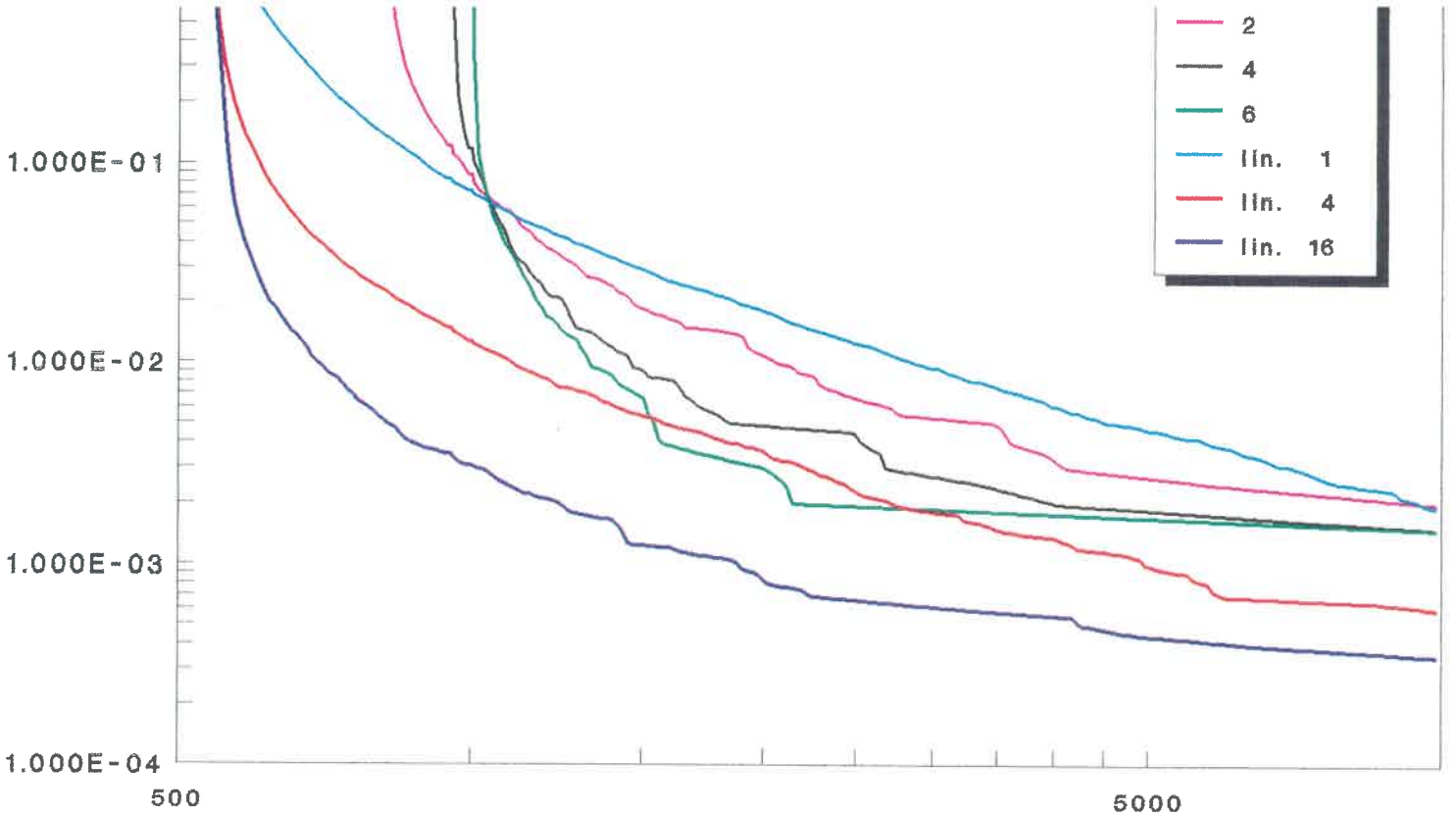


Figure D.28: Comparaison entre les architectures linéaire et arborescente reconvergente $E_b/N_0 = 3.010dB$

k = 24, code de Johannesson
nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
architecture linéaire = 20000
architecture arborescente reconvergente = 2000

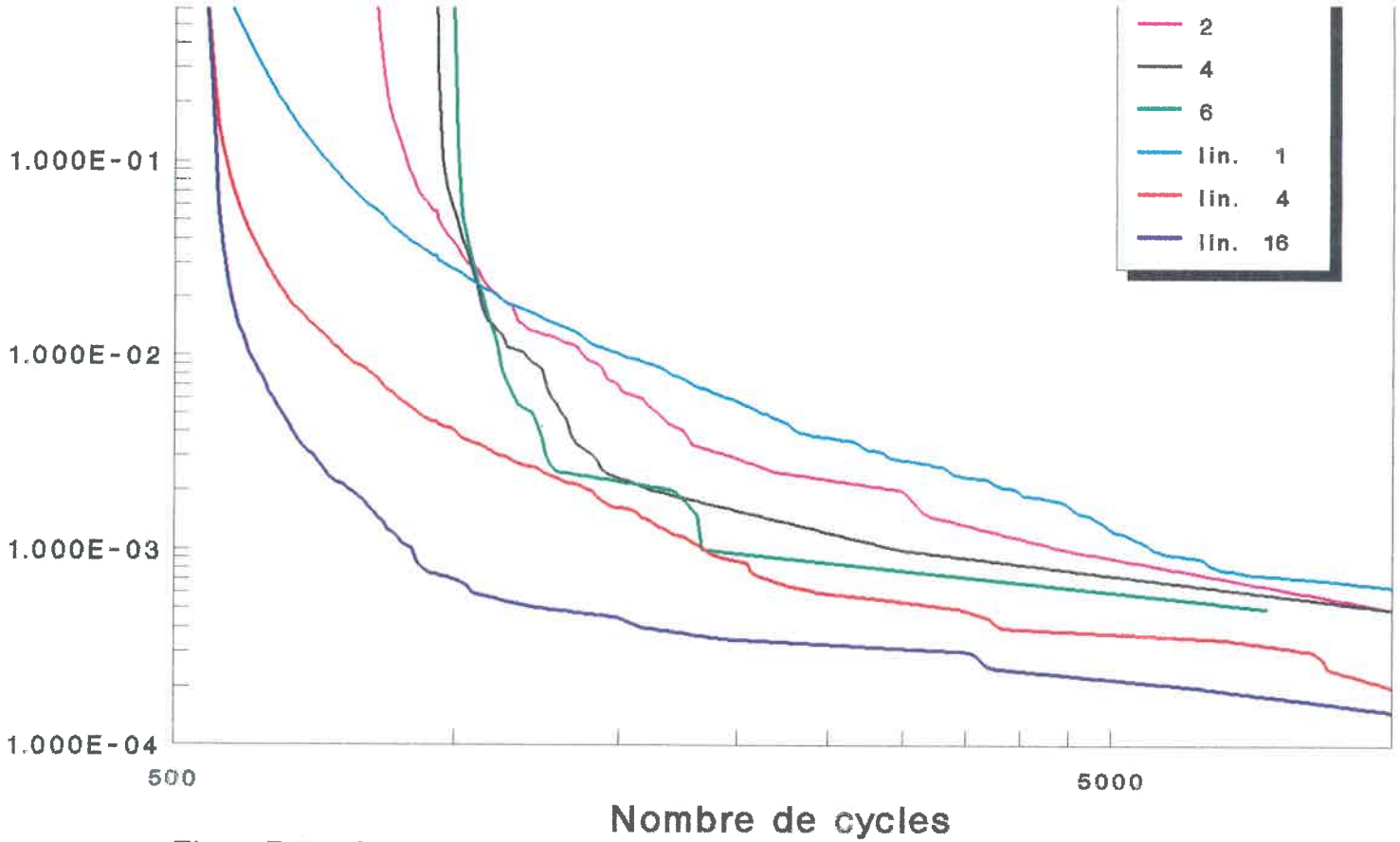


Figure D.29: Comparaison entre les architectures linéaire et arborescente reconvergente $E_b/N_0 = 3.322dB$

k = 24, code de Johannesson
nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
architecture linéaire = 20000
architecture arborescente reconvergente = 2000

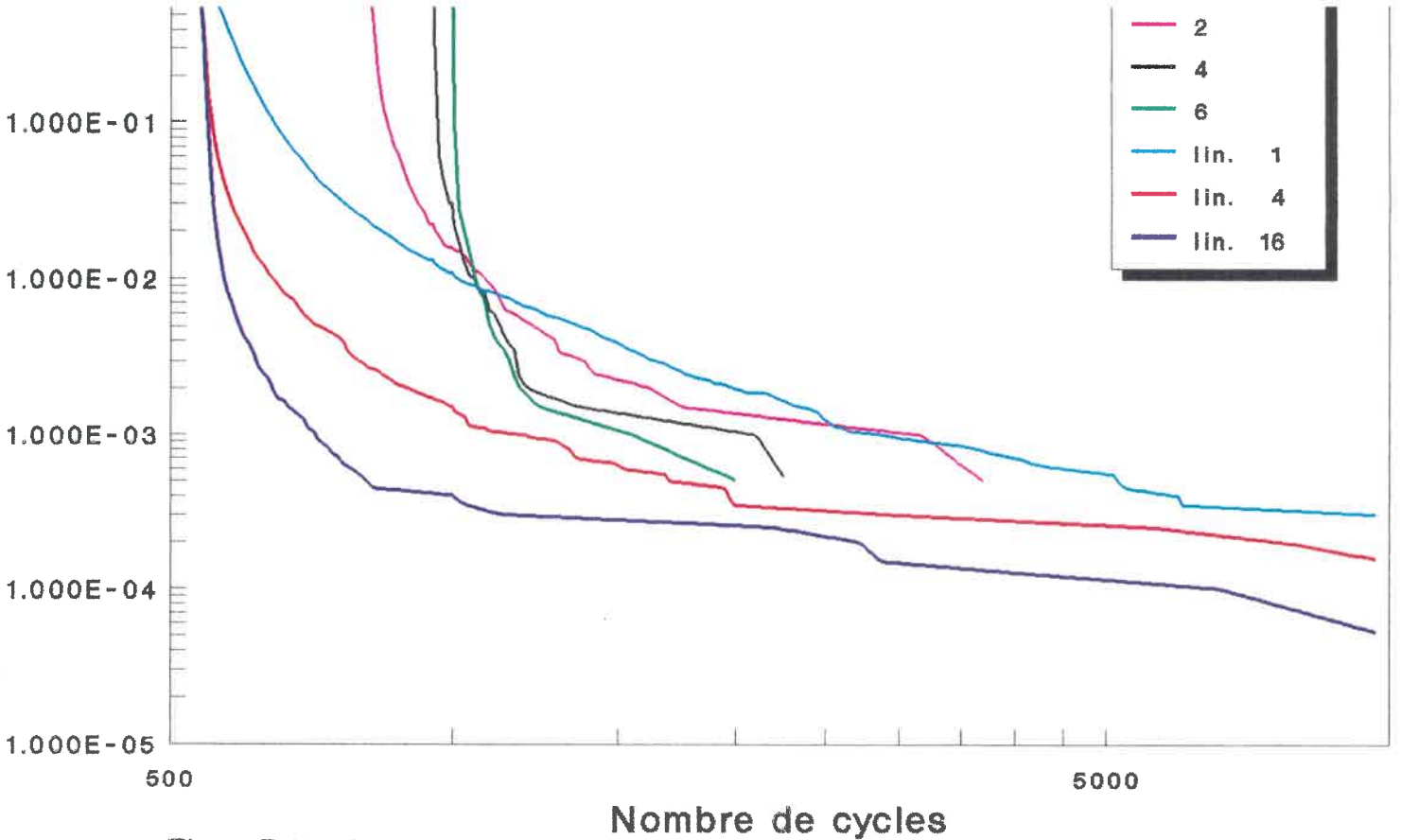


Figure D.30: Comparaison entre les architectures linéaire et arborescente reconvergente $E_b/N_0 = 3.634dB$

k = 24, code de Johannesson
nombre de bits par blocs = 505
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
architecture linéaire = 20000
architecture arborescente reconvergente = 2000

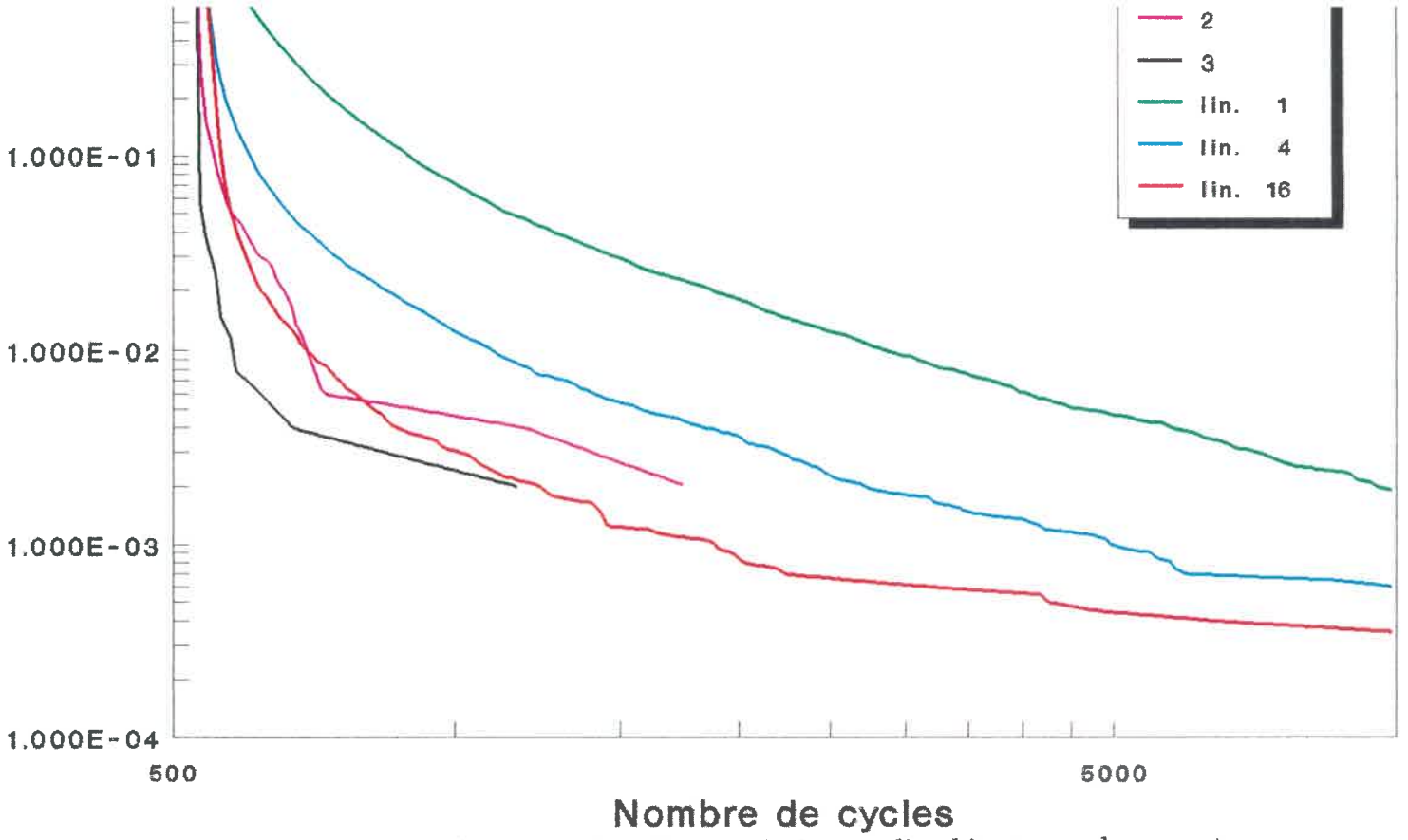


Figure D.31: Comparaison entre l'architecture linéaire et l'architecture arborescente non-reconvergente $E_b/N_0 = 3.010dB$

k = 24, code de Johannesson
nombre de bits par blocs = 499
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
linéaire = 20000
arborescente = 500

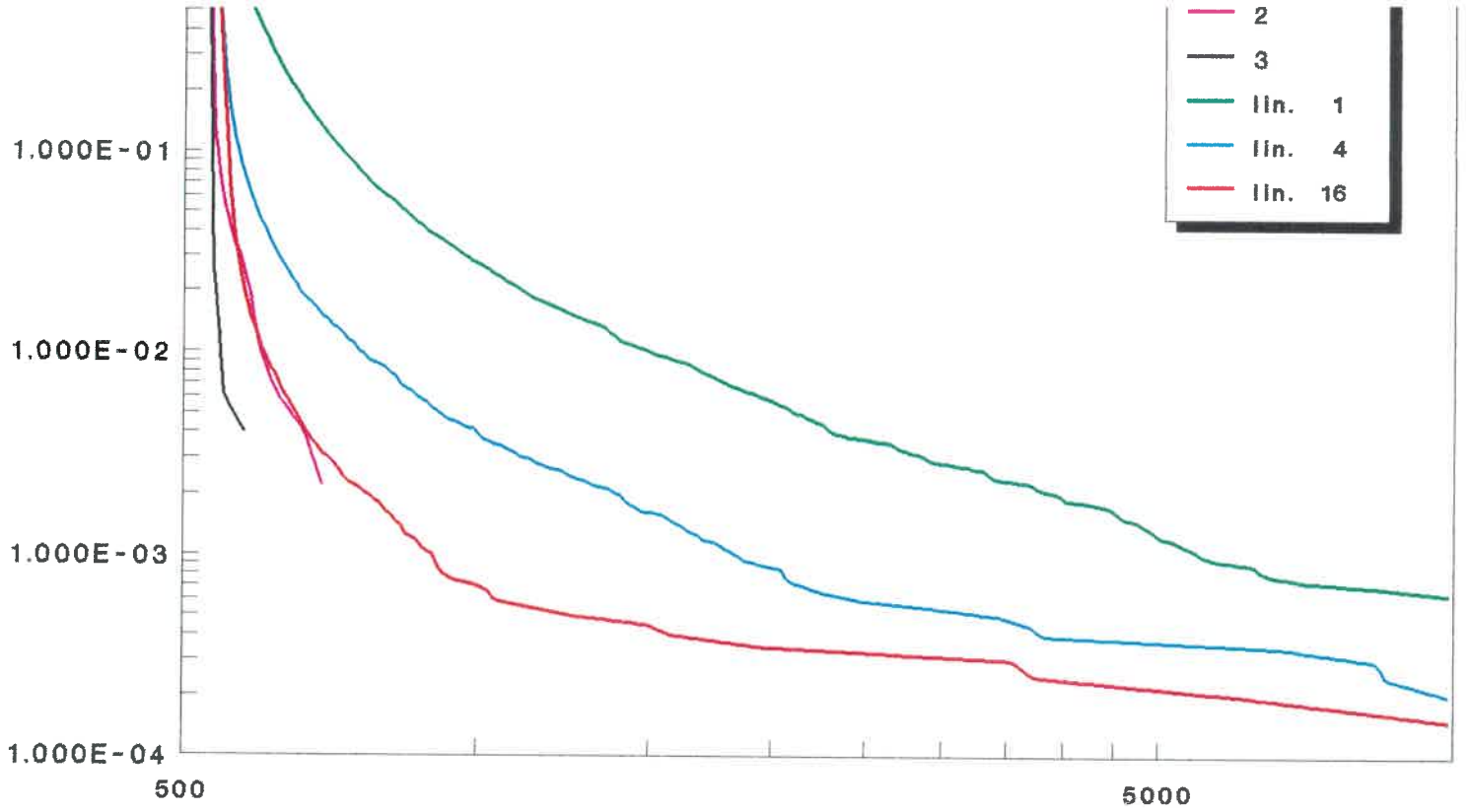


Figure D.32: Comparaison entre l'architecture linéaire et l'architecture arborescente non-reconvergente $E_b/N_0 = 3.322dB$

k = 24, code de Johannesson
nombre de bits par blocs = 499
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
linéaire = 20000
arborescente = 500

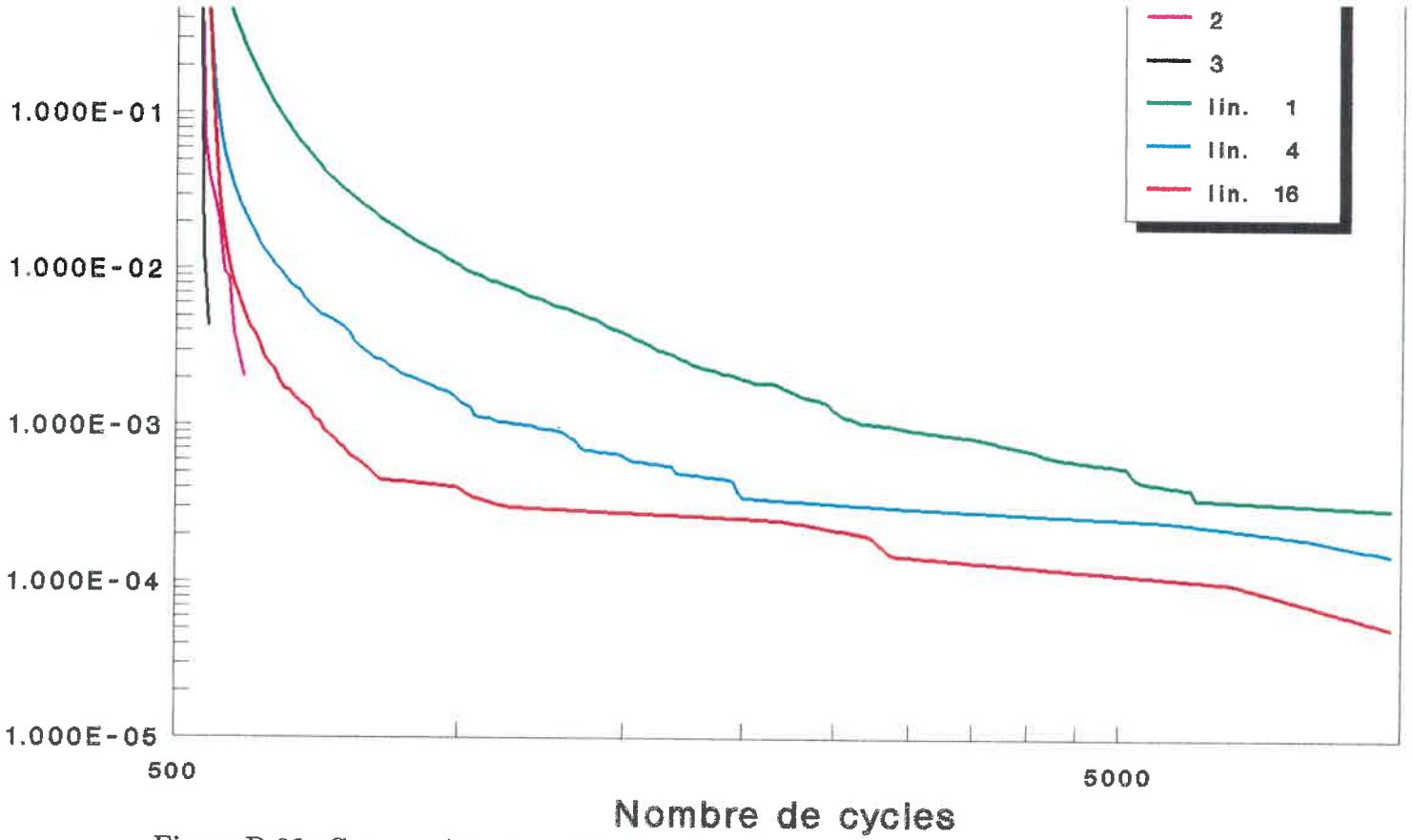


Figure D.33: Comparaison entre l'architecture linéaire et l'architecture arborescente non-reconvergente $E_b/N_0 = 3.634dB$

k = 24, code de Johannesson
nombre de bits par blocs = 499
profondeur de files = 1000, profondeur de l'historique = 10000
nombre de blocs:
linéaire = 20000
arborescente = 500

Annexe E

Programme d'évaluation de coûts

Le code-source qui suit est celui du programme qui a permit d'évaluer les coûts de chaque système.

```
#include <stdio.h>
#include <math.h>

/* cc -O -o cout cout.c -lm */

#define min(a,b) (((a) < (b)) ? (a) : (b))

void aff_erreur_fin();

main(argc,argv)
int argc;
char **argv;
   {static double p_deb,n_calcul;
   static double n_proc,q_depth,h_depth,n_enre_ic_fp,n_ic_enre;
   static double cout_bloc,n_bits_bloc;
   static char tempo[256];
   static char dummy[256];
   static FILE *in1, *in2, *out;

   static double cout_file,cout_histo,cout_ext,cout;


   if(argc != 5)
      {printf("\n   cout <type de couts> <fichier de parametre>\n");
      printf("            <fichier d'entree> <fichier de sortie>\n");
      printf("\n");
```

```
    printf("<type de couts>: a: n. de trans. avec f.p. compl.\n");
    printf("              b: n. de trans. avec f.p. econome\n");
    printf("                  c: n. de C.I.   avec f.p. compl.\n");
    printf("              d: n. de C.I.   avec f.p. economique\n");
    printf("\n");

    exit(1);
    }
else if(argv[1][1] != '\0')
    {printf("\n      cout <type de couts> <fichier de param.>\n");
    printf("            <fichier d'entree> <fichier de sortie>\n");
    printf("\n");
    printf("<type de couts> = a, b, c ou d\n");
    printf("\n");

    exit(1);
    }


in1 = fopen(argv[2],"r");
if(!in1)
    {printf("\n cout: incapable d'ouvrir le fichier de param.\n");
    exit(2);
    }

in2 = fopen(argv[3],"r");
```

```
if(!in2)
   {printf("\n cout: incapable d'ouvrir le fichier d'entree\n");
   exit(2);
   }

out = fopen(argv[4],"w");
if(!out)
   {printf("\n cout: incapable d'ouvrir le fichier de sortie\n");
   exit(2);
   }


if(!fscanf(in1, "%lf", &(cout_ext)))
   aff_erreur_fin("Erreur  fichier de param.: cout_ext",*out);
   fscanf(in1,"%[^\n]s",dummy);

if(!fscanf(in1, "%lf", &(cout_file)))
   aff_erreur_fin("Erreur  fichier de param.: cout_file",*out);
   fscanf(in1,"%[^\n]s",dummy);

if(!fscanf(in1, "%lf", &(n_enre_ic_fp)))
  aff_erreur_fin("Erreur  fichier de param.:n_enre_ic_fp",*out);
   fscanf(in1,"%[^\n]s",dummy);

if(!fscanf(in1, "%lf", &(n_ic_enre)))
   aff_erreur_fin("Erreur  fichier de param.: n_ic_enre",*out);
```

```
      fscanf(in1,"%[^\n]s",dummy);

   if(!fscanf(in1, "%lf", &(cout_histo)))
      aff_erreur_fin("Erreur  fichier de param.: cout_histo",*out);
      fscanf(in1,"%[^\n]s",dummy);

   if(!fscanf(in1, "%lf", &(cout_bloc)))
      aff_erreur_fin("Erreur  fichier de param.: cout_bloc",*out);
      fscanf(in1,"%[^\n]s",dummy);

   if(!fscanf(in1, "%lf", &(n_bits_bloc)))
      aff_erreur_fin("Erreur  fichier de param.:n_bits_bloc",*out);
      fscanf(in1,"%[^\n]s",dummy);

/* printf("%lf%lf%lf%lf%lf%lf%lf%lf%lf\n",cout_ext,cout_file,
      n_enre_ic_fp,n_ic_enre,cout_histo,cout_bloc,n_bits_bloc);  */

   fgets(tempo,256,in2);
   fputs(tempo,out);
   fgets(tempo,256,in2);
   fputs(tempo,out);
   fscanf(in2,"%[^\n]s",tempo);
   strcat(tempo,"     couts\n");
   fputs(tempo,out);

   switch(argv[1][0])
```

```
{case 'a': while(fscanf(in2,"%lf %lf %lf %10lf %10lf%[^\n]s",
     &n_proc,&q_depth,&h_depth,&p_deb,&n_calcul,dummy) != EOF)

   {cout =  n_proc*cout_ext +
            cout_file*n_proc*ceil(q_depth/n_enre_ic_fp)
            n_enre_ic_fp*n_ic_enre+
            cout_histo*ceil((n_proc+ceil(log2(h_depth)))/8.)*
               8.*1024.*exp2(min(ceil(log2(
               h_depth/1024.)),5.0)) +
            n_proc*cout_bloc;

   fprintf(out," %2.0lf   %5.0lf  %5.0lf  %-10.8lg   %9.6lg
      %12.0lf\n",n_proc,q_depth,h_depth,p_deb,n_calcul,cout);
   }
   break;

case 'b': while(fscanf(in2,"%lf %lf %lf %10lf %10lf%[^\n]s",
            &n_proc,&q_depth,&h_depth,&p_deb,&n_calcul,dummy)
            != EOF)

   {cout =  n_proc*cout_ext +
            cout_file*n_proc*ceil(q_depth/n_enre_ic_fp)*
               n_enre_ic_fp + ceil((34.+ ceil(log2(n_bits_bloc
               ))+ ceil(log2(q_depth)))/8.)*8.0*1024.*exp2(
               min(ceil(q_depth/1024.),5.0))*n_proc*6.0 +
            cout_histo*ceil(n_proc+ceil(log2(h_depth))/8.)*8.*
```

```
                1024.*exp2(min(ceil(log2(h_depth/1024.)),5.0))+
            n_proc*cout_bloc;

    fprintf(out," %2.0lf  %5.0lf  %5.0lf  %-10.8lg  %9.6lg
        %12.0lf\n",n_proc,q_depth,h_depth,p_deb,n_calcul,cout);
    }
    break;

case 'c': while(fscanf(in2,"%lf %lf %lf %10lf %10lf%[^\n]s",
                &n_proc,&q_depth,&h_depth,&p_deb,&n_calcul,
                dummy) != EOF)

    {cout =  2.*n_proc +
            n_proc*n_ic_enre*ceil(q_depth/n_enre_ic_fp) +
            10. + ceil((ceil(log2(h_depth)) + n_proc)/8.)*
                ceil(h_depth/(1024.*exp2(min(ceil(log2(h_depth/
                1024.)),5.0))))+
            n_proc*7.;

    fprintf(out," %2.0lf  %5.0lf  %5.0lf  %-10.8lg  %9.6lg
          %12.0lf\n",n_proc,q_depth,h_depth,p_deb,n_calcul,
          cout);
    }
    break;

case 'd': while(fscanf(in2,"%lf %lf %lf %10lf %10lf%[^\n]s",
```

```
                &n_proc,&q_depth,&h_depth,&p_deb,&n_calcul,
                dummy) != EOF)

      {cout =  2.*n_proc +
               n_proc*(ceil(q_depth/n_enre_ic_fp) +
                  ceil((34.+ceil(log2(n_bits_bloc))+ceil(log2(
                  q_depth)))/8.)*exp2(min(ceil(log2(q_depth/1024.
                  )),5.0))) +
               10. + ceil((ceil(log2(h_depth)) + n_proc)/8.)*
                  ceil(h_depth/(1024.*exp2(min(ceil(log2(h_depth/
                  1024.)),5.0))))+
               n_proc*7.;

      fprintf(out,"  %2.0lf   %5.0lf  %5.0lf  %-10.8lg  %9.6lg
         %12.0lf\n",n_proc,q_depth,h_depth,p_deb,n_calcul,cout);
      }
      break;

   default: printf("Le type de couts doit etre: 'a', 'b', 'c' ou
         'd'.\n");
   }

exit(0);
}
```

```
void aff_erreur_fin(message,fpout1)
char *message;
FILE *fpout1;
{
   if(fpout1)
         {fprintf(fpout1,"\nFIN ANORMALE\n");
         fprintf(fpout1,"%s\n",message);
         }
   fprintf(stderr,"\nFIN ANORMALE\n");
   fprintf(stderr,"%s\n",message);

   exit(1);
}
```

7000	cout_ext	n. trans./extenseur
857	cout_file	n. trans./enregistrement 18000/21
21	n_enre_ic_fp	n. d'enre./f.p.
5	n_ic_enre	n. de C.I./enre. de f.p.
6	cout_histo	n. trans./bit d'histo.
27000	cout_bloc	n. trans./bloc
505	n_bits_bloc	n. de bits par bloc de message

Tableau E.1: Paramètres du programme d'évaluation des coûts

Le tableau E.1 illustre le fichier de paramètres utilisé lors de ces évaluations de coûts.

Annexe F

Fichier de sortie et résultats des simulations

Les données qui suivent ont été extraites des résultats des simulations et permettant de voir où se situent les "plateaux" des cumulatives de l'effort de calcul. Certains enregistrements contiennent des données partielles parce que le simulateur a été modifié entre temps pour fournir plus de résultats.

Eb_No = 2.698000 dB

# n. proc.	q_depth	h_depth	p[debord.]	E[n. de calculs]
1	50	1834	0.134000	???????
1	125	3729	0.060500	???????
1	100	2780	0.072500	???????
1	200	4611	0.037500	810.221
1	300	5694	0.022500	845.047
1	400	6240	0.019500	858.456
1	500	6710	0.015000	878.490
1	600	11767	0.011500	900.123
1	700	7714	0.011000	900.778
1	800	9538	0.009000	914.597
1	900	9990	0.008000	921.979
2	25	1049	0.143500	???????
2	50	1658	0.078500	???????
2	125	2745	0.029000	???????
2	55	1719	0.073000	???????
2	110	2654	0.033500	653.416
2	165	3008	0.022000	668.655

2	220	3354	0.017500	677.975
2	275	5551	0.014000	687.803
2	330	6428	0.010000	698.946
2	385	7276	0.009500	701.439
2	440	7977	0.008000	707.865
2	495	8561	0.006500	714.874
2	550	9223	0.006000	718.346
4	12	770	0.149000	???????
4	25	1039	0.082000	???????
4	50	1518	0.043500	???????
4	125	2071	0.016500	???????
4	27	1106	0.075500	???????
4	54	1528	0.038500	587.759
4	81	1775	0.024000	596.279
4	108	2711	0.019000	601.606
4	135	2140	0.015500	604.477
4	162	3502	0.012000	610.324
4	189	2890	0.009500	613.889
4	216	2997	0.008000	616.440
4	243	3127	0.008000	616.815
4	270	3141	0.006500	619.940
8	6	638	0.164000	???????

8	12	707	0.087000	???????
8	25	960	0.043000	???????
8	50	1199	0.022000	???????
8	14	738	0.076500	???????
8	28	990	0.040000	564.320
8	42	1101	0.027000	568.027
8	56	1258	0.020500	570.495
8	70	1340	0.016000	572.865
8	84	1400	0.014500	573.888
8	98	1447	0.011500	575.706
8	112	1477	0.010500	576.599
8	126	1516	0.009000	577.795
8	140	1760	0.007500	579.420
16	6	629	0.120000	???????
16	12	733	0.056000	???????
16	25	930	0.028500	???????
16	50	1008	0.013500	???????
16	9	696	0.074500	???????
16	18	765	0.040000	557.021
16	27	941	0.026000	559.241
16	36	976	0.019500	560.764

16	45	996	0.015000	561.887
16	54	1007	0.013000	562.518
16	63	1039	0.010000	563.662
16	72	1065	0.009500	563.969
16	81	1181	0.009000	564.341

Les pages qui suivent donnent un exemple de fichier de simulation où on peut voir le "plateau". Remarquez la distance entre le dernier bloc bien décodé et le premier bloc mal décodé par rapport aux autres distances. On notera que le fichier a été amputé pour éviter l'ajout de pages inutiles. Les endroits où ont été effectués ces changements sont illustrés par trois points.

```
# ****************************
# *                          *
# *    decodage sequentiel   *
# *      multiprocesseur     *
# *                          *
# ****************************
```

```
k            = 24          #longueur de contrainte
u            = 1           #num. du taux de codage
v            = 2           #deno. du taux de codage
n_quant      = 8           #n. de niveaux de quant.
c1           = 55076157    #code utilise
c2           = 75501351    # "      "
lin_len      = 16          #n. de processeurs
h_depth      = 10000       #prof. de l'historique
q_depth      = 63          #prof. de la queue
n_blocs      = 2000        #n. de blocs
n_bits       = 505         #n. de bits par bloc
seed_canal   = 731         #semence pour bruiter le canal
seed_message = 32767       #semence pour le message
seq_zero     = 0           FAUX
```

```
        ******************

        Eb_No = 2.698000 dB          #rapport signal/bruit
        rcomp = 0.505074 #bit/sym.
        cap   = 0.477627 #nat./sym.

# metriques  p_trans  p_cum

#          4  0.446017 1.000000
#          4  0.196154 0.553983
#          3  0.164109 0.357829
#         -1  0.107479 0.193720
#         -8  0.055099 0.086241
#        -21  0.022109 0.031142
#        -35  0.006943 0.009033
#        -58  0.002090 0.002090



n_debord = 13 #DEBORDEMENTS DE L'HISTORIQUE!

#  semence finale = 65706715

 n_moy_itera =     649.3 #n. moyen d'iterations
 var_itera   = 2376009.2 #variance du n. d'iterations
```

```
 n_erreur     =        415 #n. d'erreurs
 B_E_R        = 4.136e-04 #"bit error rate"

#       transitions experimentales

#         0->0         0.445906
#         0->1         0.195900
#         0->2         0.164527
#         0->3         0.107206
#         0->4         0.055407
#         0->5         0.022000
#         0->6         0.006916
#         0->7         0.002140


# n. de calculs n. de blocs cumulative nerreur p[erreur] moyenne du
#                    de l'effort de calcul              n. de calculs

DEBUT CUM_BLOC
       532        1          1.000000       0      0.000e+00     532.000
       533        1          0.999500       0      0.000e+00     532.500
       534        5          0.999000       0      0.000e+00     533.571
       535        6          0.996500       0      0.000e+00     534.231
       536        6          0.993500       0      0.000e+00     534.789
       537       11          0.990500       0      0.000e+00     535.600
       538       21          0.985000       0      0.000e+00     536.588
```

539	37	0.974500	0	0.000e+00	537.602
540	31	0.956000	0	0.000e+00	538.227
541	57	0.940500	0	0.000e+00	539.125
		.			
		.			
		.			
901	1	0.014500	0	0.000e+00	561.783
913	1	0.014000	0	0.000e+00	561.961
932	1	0.013500	0	0.000e+00	562.149
947	1	0.013000	0	0.000e+00	562.344
949	1	0.012500	0	0.000e+00	562.539
971	1	0.012000	0	0.000e+00	562.746
1003	1	0.011500	0	0.000e+00	562.969
1011	1	0.011000	0	0.000e+00	563.195
1012	1	0.010500	0	0.000e+00	563.422
1039	1	0.010000	0	0.000e+00	563.662
6806	1	0.009500	49	4.896e-05	566.811
7864	1	0.009000	63	1.118e-04	570.491
8995	1	0.008500	58	1.697e-04	574.737
9154	1	0.008000	80	2.494e-04	579.059
9161	1	0.007500	76	3.250e-04	583.381
9740	1	0.007000	89	4.136e-04	587.989
10000	13	0.006500	0	4.109e-04	649.167

FIN CUM_BLOC

```
       ******************

       Eb_No = 3.010000 dB          #rapport signal/bruit
       rcomp = 0.532230 #bit/sym.
       cap   = 0.494523 #nat./sym.

# metriques  p_trans  p_cum

#          4  0.465799 1.000000
#          4  0.194825 0.534201
#          3  0.159062 0.339377
#         -1  0.101658 0.180315
#         -9  0.050857 0.078657
#        -21  0.019914 0.027800
#        -36  0.006102 0.007886
#        -60  0.001783 0.001783



n_debord = 6 #DEBORDEMENTS DE L'HISTORIQUE!

#  semence finale = 65706715

 n_moy_itera =     584.6 #n. moyen d'iterations
```

```
 var_itera    = 1012758.8 #variance du n. d'iterations
 n_erreur     =        46 #n. d'erreurs
 B_E_R        = 4.568e-05 #"bit error rate"

#       transitions experimentales

#         0->0         0.465664
#         0->1         0.194573
#         0->2         0.159431
#         0->3         0.101351
#         0->4         0.051323
#         0->5         0.019720
#         0->6         0.006107
#         0->7         0.001830


# n. de calculs n. de blocs cumulative nerreur p[erreur] moyenne du
#                       de l'effort de calcul           n. de calculs

DEBUT CUM_BLOC
       531        1          1.000000       0      0.000e+00     531.000
       532        7          0.999500       0      0.000e+00     531.875
       533        2          0.996000       0      0.000e+00     532.100
       534       12          0.995000       0      0.000e+00     533.136
       535       19          0.989000       0      0.000e+00     534.000
       536       28          0.979500       0      0.000e+00     534.812
```

537	53	0.965500	0	0.000e+00	535.762
538	72	0.939000	0	0.000e+00	536.593
539	89	0.903000	0	0.000e+00	537.350
		.			
		.			
		.			
756	1	0.008500	0	0.000e+00	550.945
760	1	0.008000	0	0.000e+00	551.050
784	1	0.007500	0	0.000e+00	551.167
789	1	0.007000	0	0.000e+00	551.287
809	1	0.006500	0	0.000e+00	551.416
850	1	0.006000	0	0.000e+00	551.567
867	1	0.005500	0	0.000e+00	551.725
912	1	0.005000	0	0.000e+00	551.906
995	1	0.004500	0	0.000e+00	552.129
1206	1	0.004000	0	0.000e+00	552.457
8073	1	0.003500	46	4.568e-05	556.228
10000	6	0.003000	0	4.554e-05	584.559

FIN CUM_BLOC

```
       ******************

       Eb_No = 3.322000 dB          #rapport signal/bruit
       rcomp = 0.559981 #bit/sym.
       cap   = 0.511116 #nat./sym.

# metriques  p_trans  p_cum

#         4  0.486393 1.000000
#         4  0.192958 0.513607
#         3  0.153598 0.320649
#        -1  0.095711 0.167051
#        -9  0.046684 0.071340
#       -22  0.017822 0.024656
#       -38  0.005325 0.006834
#       -63  0.001509 0.001509



n_debord = 2 #DEBORDEMENTS DE L'HISTORIQUE!

#  semence finale = 65706715

 n_moy_itera =     559.4 #n. moyen d'iterations
```

```
 var_itera    =  469300.1 #variance du n. d'iterations
 n_erreur     =        76 #n. d'erreurs
 B_E_R        = 7.532e-05 #"bit error rate"

#      transitions experimentales

#         0->0        0.486139
#         0->1        0.192860
#         0->2        0.153895
#         0->3        0.095515
#         0->4        0.047020
#         0->5        0.017716
#         0->6        0.005315
#         0->7        0.001540


# n. de calculs n. de blocs cumulative nerreur p[erreur] moyenne du
#                      de l'effort de calcul           n. de calculs

DEBUT CUM_BLOC
       530        2         1.000000       0     0.000e+00    530.000
       531        1         0.999000       0     0.000e+00    530.333
       532       19         0.998500       0     0.000e+00    531.773
       533       15         0.989000       0     0.000e+00    532.270
       534       51         0.981500       0     0.000e+00    533.273
       535       60         0.956000       0     0.000e+00    533.973
```

536	88	0.926000	0	0.000e+00	534.729
537	96	0.882000	0	0.000e+00	535.386
538	115	0.834000	0	0.000e+00	536.058
539	145	0.776500	0	0.000e+00	536.779
		.			
		.			
		.			
658	1	0.006500	0	0.000e+00	544.586
659	1	0.006000	0	0.000e+00	544.644
663	1	0.005500	0	0.000e+00	544.703
668	1	0.005000	0	0.000e+00	544.765
718	1	0.004500	0	0.000e+00	544.852
727	1	0.004000	0	0.000e+00	544.943
729	1	0.003500	0	0.000e+00	545.036
759	1	0.003000	0	0.000e+00	545.143
777	1	0.002500	0	0.000e+00	545.259
794	1	0.002000	0	0.000e+00	545.384
9688	1	0.001500	76	7.532e-05	549.959
10000	2	0.001000	0	7.525e-05	559.409

FIN CUM_BLOC

```
       ******************

       Eb_No = 3.634000 dB          #rapport signal/bruit
       rcomp = 0.588204 #bit/sym.
       cap   = 0.527317 #nat./sym.

# metriques  p_trans  p_cum

#         4  0.507778 1.000000
#         4  0.190515 0.492222
#         3  0.147724 0.301707
#         0  0.089665 0.153983
#        -9  0.042601 0.064319
#       -23  0.015842 0.021718
#       -39  0.004610 0.005876
#       -65  0.001266 0.001266



n_debord = 0 #DEBORDEMENTS DE L'HISTORIQUE!

#  semence finale = 65706715

 n_moy_itera =     541.4 #n. moyen d'iterations
```

```
 var_itera    =     821.0 #variance du n. d'iterations
 n_erreur     =         0 #n. d'erreurs
 B_E_R        = 0.000e+00 #"bit error rate"

#      transitions experimentales

#         0->0        0.507373
#         0->1        0.190728
#         0->2        0.147875
#         0->3        0.089445
#         0->4        0.042929
#         0->5        0.015738
#         0->6        0.004616
#         0->7        0.001296


# n. de calculs n. de blocs cumulative nerreur p[erreur] moyenne du
#                      de l'effort de calcul             n. de calculs

DEBUT CUM_BLOC
       530        4          1.000000       0      0.000e+00     530.000
       531       16          0.998000       0      0.000e+00     530.800
       532       31          0.990000       0      0.000e+00     531.529
       533       53          0.974500       0      0.000e+00     532.279
       534      101          0.948000       0      0.000e+00     533.127
       535      132          0.897500       0      0.000e+00     533.861
```

536	171	0.831500	0	0.000e+00	534.581
537	175	0.746000	0	0.000e+00	535.201
538	162	0.658500	0	0.000e+00	535.737
539	186	0.577500	0	0.000e+00	536.326
		.			
		.			
		.			
622	1	0.004500	0	0.000e+00	540.738
632	1	0.004000	0	0.000e+00	540.784
634	1	0.003500	0	0.000e+00	540.830
639	1	0.003000	0	0.000e+00	540.880
643	1	0.002500	0	0.000e+00	540.931
700	1	0.002000	0	0.000e+00	541.011
708	1	0.001500	0	0.000e+00	541.094
779	1	0.001000	0	0.000e+00	541.213
967	1	0.000500	0	0.000e+00	541.426

FIN CUM_BLOC

Bibliographie

[1] James L. Massey. *Threshold Decoding.* MIT Press, Cambridge, Mass., 1963.

[2] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorihtm. *IEEE Trans. Inform. Theory*, IT-13:260–269, April 1967.

[3] Robert M. Fano. A heuristic discussion of probabilistic decoding. *IEEE Trans. Inform. Theory*, IT-9:64–74, April 1963.

[4] F. Jelinek. Fast sequential decoding algorithm using a stack. *IBM J. Res. Develop.*, 13:675–685, November 1969.

[5] J. Bibb Cain and Rob A. Kriete. A VLSI R=1/2, K=7 Viterbi decoder. In *Proc. IEEE 1984 National Aerospace and Electronic Conf.*, pages 20–27, Dayton, OH, May 21–25, 1984.

[6] R. M. Orndorff, P. C. Chou, J. D. Kremarik, T. W. Doak, and R. Koralek. Viterbi decoder VLSI integrated circuit for bit error correction. In *Proc. National Telecommunications Conf.*, pages 149–152, New Orleans, Lousiana, Nov. 29–Dec. 3, 1981.

[7] Stanley J. Simmons. A nonsorting vlsi structure for implementing the (M, L) algorithm. *IEEE J. Select. Areas Commun.*, SAC-6(3):538–546, April 1988.

[8] Yutaka Yasuda, Yasuo Hirata, Katsuhiro Nakamura, and Susumu Otani. Development of variable-rate Viterbi decoder and its performance characteristics. In *Proc. Sixth Int'l Conf. Digital Satellite Commun.*, pages XII/24–31, Phoenix, AZ, Sept. 19–23, 1983.

[9] G. David Forney, Jr. and Edward K. Bower. A high-speed sequential decoder: prototype design and test. *IEEE Trans. Commun. Technol.*, COM-19(5):821–835, October 1971.

[10] David Haccoun et Pierre Lavoie et Yvon Savaria. New architectures for fast convolutional encoders and threshold decoders. *IEEE J. Select. Areas Commun.*, 6(3):547–557, 1988.

[11] David Haccoun. *Décodeur Séquentiel Haute Vitesse: Réalisation et Tests Préléminaires.* Technical Report EP84-R-19, Editions de l'Ecole Polytechnique de Montréal, Québec, Canada, June 1984.

[12] C. Y. Chang and K. Yao. Systolic array architecture for the sequential stack decoding algorithm. In *Proc. SPIE vol. 696, Advanced Alg. and Arch. for Signal Proc.*, pages 196–203, 1986.

[13] Pierre Lavoie, David Haccoun, and Yvon Savaria. A systolic architecture for fast stack sequential decoders. In *Proceedings of the 14th Biennial Symposium on Communications*, pages D.5.5–D.5.8, Queen's University, Kingston, Ontario, Canada, May 29–June 1, 1988.

[14] Pierre Lavoie et David Haccoun et Yvon Savaria. *Spécification d'un Décodeur Séquentiel Rapide Utilisant une Queue Prioritaire Systolique.* Technical Report EPM/RT-88/11, Editions de l'Ecole Polytechnique de Montréal, Québec, Canada, March 1988.

[15] Pierre R. Chevillat and Daniel J. Costello, Jr. A multiple stack algorithm for erasurefree decoding of convolutional codes. *IEEE Trans. Commun.*, COM-25(12):1460–1470, December 1977.

[16] David Haccoun and Michael J. Ferguson. Generalized stack algorithms for decoding convolutional codes. *IEEE Trans. Inform. Theory*, IT-21(6):638–651, November 1975.

[17] David Haccoun and Guy Bégin. *High-Rate Punctured Convolutional Codes.* Technical Report EPM/RT-88/1, Editions de l'Ecole Polytechnique de Montréal, Québec, Canada, January 1988.

[18] Rolf Johannesson. Communication privée.