UNIVERSITÉ DE MONTRÉAL

EFFICIENT METHODS FOR FINDING OPTIMAL CONVOLUTIONAL SELF-DOUBLY ORTHOGONAL CODES

GILBERT KOWARZYK
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE ÉLECTRIQUE)
AOÛT 2013

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

EFFICIENT METHODS FOR FINDING OPTIMAL CONVOLUTIONAL
SELF-DOUBLY ORTHOGONAL CODES

présentée par: KOWARZYK Gilbert
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

To Jacqueline and Natasha, with love…

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my supervisors and mentors, Dr. Yvon Savaria and Dr. David Haccoun. You have been a role model for what can be accomplished by hard work and creative thinking, and your attitude towards research has had a deep impact in my life and career. I would like to thank you for having offered me the opportunity to work on this project, for your invaluable guidance and patience, and for your financial support. I would like to also thank you for having given me the privilege to attend several conferences and workshops throughout the world, which has undeniably contributed to my development as a researcher.

I would particularly like to thank Dr. Normand Bélanger. Thank you for your help, your precious advice for writing journal articles, your support and deep technical insights, and for the interesting conversations about computer engineering. Thank you for providing an invaluable expertise and a contagious enthusiasm for new projects and challenges.

I am deeply grateful to Eric, for all the explanations and illuminating discussions regarding error-correcting systems, and for making me feel part of a team.

I wish to also thank my friends, for their ongoing support, for helping me maintain my sanity and for providing much needed distractions.

To my parents, thank you for your unconditional support, endless love and devotion. Special thanks to my sisters, for your moral support, patience, and for keeping your sense of humour when I had lost mine.

Last but not least, I would like to thank CMC Microsystems for providing software, equipment, support and workshops, which were all essential throughout my research, and to the Fonds de Recherche du Québec - Nature et Technologies for their generous financial support.

RÉSUMÉ

Au cours des dernières années, la hausse sans précédent du nombre d'ultrabooks et d'appareils mobiles s'est accompagnée d'un besoin toujours croissant d'accès aux technologies permettant des communications sans-fil fiables et à haut débit. Pour atténuer ou éliminer les erreurs induites par les interférences et le bruit dans les canaux de communication, il est important de développer des systèmes de codage efficaces pour la correction d'erreurs. En effet, lors de communications de données numériques sur un canal ayant un faible rapport signal sur bruit, ces *codes* permettent de conserver un taux d'erreur faible tout en augmentant le débit des transmissions et/ou en diminuant la puissance d'émission requise. Ceci contribue grandement à améliorer l'efficacité énergétique de ces dispositifs électroniques sans-fil et, ainsi, à prolonger leur autonomie.

Dans cette thèse par articles, nous présentons un algorithme de recherche efficace pour trouver deux types de codes correcteurs d'erreur : les codes convolutionnels doublement orthogonaux (CDO) et les codes convolutionnels doublement orthogonaux simplifiés (S-CDO). En effet, ces codes sont utilisés dans un système de contrôle d'erreurs ayant un décodage à seuil itératif différent de la procédure de décodage Turbo classique, puisqu'il ne nécessite aucun entrelaceur, ni à l'encodage, ni aux étapes de décodage. Néanmoins, son processus de décodage à seuil nécessite que ces codes convolutionnels systématiques satisfassent des propriétés dites de « double orthogonalité », allant au-delà des conditions requises par les codes « simplement orthogonaux », bien connus et habituellement utilisés lors d'un décodage à seuil non-itératif. Afin de pouvoir construire des *codecs* à haute performance et à faible latence avec ces codes, il est important de minimiser leur *longueur de contrainte* ou « *span* » pour un nombre J de connexions donné. Bien que trouver des codes CDO et S-CDO ne soit pas difficile, déterminer les codes ayant un span *minimal* (dit *optimal*) pour un *ordre* J donné est mathématiquement très complexe. En effet, la construction directe de codes CDO / S-CDO à span court/optimal reste un problème ouvert et qui est soupçonné d'être *NP-complet.*

Cette thèse présente un total de trois articles : deux articles publiés dans *IEEE Transac-*

tions on Communications[1,2] et un article soumis au journal *IEEE Transactions on Parallel and Distributed Systems*[3]. Dans ces articles, nous décrivons un nouvel algorithme de recherche parallèle, efficace et implicitement-exhaustif pour trouver des codes CDO et S-CDO systématiques, à taux $R = \frac{1}{2}$ et ayant un span plus court, voire minimal, c.à.d. optimal. Comparé à l'algorithme de recherche implicitement-exhaustif de référence, l'algorithme de recherche à haute performance proposé reste exhaustif mais fournit un facteur d'accélération très important, supérieur à 16300 pour les codes CDO ($J = 7$) et supérieur à 6300 pour les codes S-CDO ($J = 8$).

Ces accélérations sont atteintes grâce à des améliorations algorithmiques permettant la réduction déterministe de l'espace de recherche, ainsi qu'à une fonction de validation grandement améliorée et utilisant une structure de données nouvelle lui permettant de faire un calcul incrémental et une réutilisation des données. Comparée à la fonction de validation de référence, cette nouvelle fonction de validation offre un facteur d'accélération très substantiel, supérieur à 190000 pour les codes CDO ($J = 17$) et supérieur à 60000 pour les codes S-CDO ($J = 17$). De plus, comparée à la fonction de validation de codes CDO la plus rapide et qui est utilisée dans les algorithmes de recherche pseudo-aléatoires de haute performance, la fonction de validation proposée offre un facteur d'accélération supérieur à 2000 pour les codes CDO ($J = 17$). La combinaison d'optimisations et de techniques d'équilibrage de charge proposée nous a permis d'exploiter plusieurs centaines de coeurs de calcul afin d'effectuer une recherche exhaustive sur un espace de recherche environ 10^{14} fois plus grand qu'auparavant, nous donnant ainsi le moyen de trouver, dans un délai de temps de calcul raisonnable, de nouveaux codes CDO et S-CDO plus courts voire optimaux.

Nous fournissons les codes CDO et S-CDO à span optimal obtenus, ayant respectivement un ordre $J \in \{6, ..., 9\}$ et $J \in \{9, ..., 12\}$. Nous fournissons aussi les codes CDO / S-CDO ayant les spans les plus courts publiés à date pour $J \in \{10, ..., 17\}$ et $J \in \{13, ..., 20\}$ respectivement. Grâce à cet algorithme, nous avons pu réduire la longueur du span de 14% en moyenne pour les codes CDO et de 26% en moyenne pour les codes S-CDO, permettant ainsi une réduction de latence de la même ampleur dans les systèmes correcteurs d'erreurs qui leur sont destinés.

[1] *IEEE Transactions on Communications*, *Transactions Letters*, vol. 60, no. 1, January 2012, pp. 3-8.
[2] *IEEE Transactions on Communications*, vol. 61, no. 3, March 2013, pp. 865-876.
[3] *IEEE Transactions on Parallel and Distributed Systems* - submitted August 18, 2013.

Nous comparons le span de ces nouveaux codes aux bornes inférieures théoriques connues et présentons les performances de correction d'erreur de certains de ces codes, ainsi que l'amélioration en longueur de span obtenue lorsque l'on utilise un code S-CDO au lieu d'un code CDO ayant le même ordre J. Nous montrons que, bien que les codes CDO offrent une performance d'erreur légèrement supérieure, d'un point de vue technique, les codes S-CDO offrent clairement des avantages : une latence de décodage beaucoup plus faible pour une performance d'erreur semblable. Nous confirmons également que, pour des valeurs de E_b/N_0 modérées (soit $E_b/N_0 > 3$dB), les codes CDO / S-CDO offrent une performance d'erreur concurrentielle aux codes Turbo et par conséquent une alternative convaincante : leurs courbes de performance d'erreur ont une région « plancher[4] » plus basse que celle des codes Turbo, fournissant ainsi une meilleure performance d'erreur tout en ayant une latence de décodage inférieure et permettant une mise en oeuvre moins complexe.

Enfin, nous présentons l'évolution de la performance d'erreur des codes CDO / S-CDO en fonction de leur ordre J. Nous montrons que, bien qu'une augmentation de la valeur de J conduise à une amélioration de la performance d'erreur, cela est réussi au prix d'un déplacement de la zone « cascade[5] » des performances d'erreur à une région où E_b/N_0 a une valeur plus élevée.

[4]En anglais : « floor » region.
[5]En anglais : « waterfall » region.

ABSTRACT

In recent years, the rise of ultrabooks and mobile devices has been accompanied by an ever increasing need for reliable high-bandwidth wireless communications. To mitigate or eliminate the errors that are invariably introduced due to noise and interference in the communication channels, it is important to develop efficient error-correcting coding schemes. Indeed, these *codes* may be used to preserve the error performance while allowing the data-rate of digital communications to be increased and the transmission power at lower signal-to-noise ratios to be reduced, thereby improving the overall power efficiency of these devices.

In this *manuscript-based thesis*, we present an efficient search algorithm for finding optimal/short-span *Convolutional Self-Doubly Orthogonal* (CDO) codes and *Simplified Convolutional Self-Doubly Orthogonal* (S-CDO) codes. These error-correcting codes are employed in an iterative error-control coding scheme that differs from the classical Turbo code procedure, as it does not require any interleaver, neither at the encoding nor at the decoding stages. However, its iterative threshold decoding procedure requires that these systematic convolutional codes satisfy some "double orthogonality properties", beyond those of the well-known orthogonal codes used in the usual non-iterative threshold decoding. In order to build high-performance, low-latency codecs with these codes, it is important to minimize the constraint length, also called "*span*", for a given number J of generator connections. Although finding CDO/S-CDO codes is not difficult, determining the optimal/short-span codes for a given order J is computationally very challenging. The direct construction of optimal or shortest-span CDO and S-CDO codes has so far eluded analysis, and the search for these codes is believed to be an *NP-complete* problem.

The thesis presents a total of three articles: two articles that were published in *IEEE Transactions on Communications*[6,7], and one article that was submitted for publication to *IEEE Transactions on Parallel and Distributed Systems*[8]. In these articles, we describe a novel efficient and parallel implicitly-exhaustive search algorithm for finding rate $R = \frac{1}{2}$ sys-

[6] *IEEE Transactions on Communications*, *Transactions Letters*, vol. 60, no. 1, January 2012, pp. 3-8.
[7] *IEEE Transactions on Communications*, vol. 61, no. 3, March 2013, pp. 865-876.
[8] *IEEE Transactions on Parallel and Distributed Systems* - submitted August 18, 2013.

tematic optimal/short-span CDO and S-CDO codes. The high-performance search algorithm is still exhaustive in nature, yet it provides an impressive speedup that is larger than 16300 (CDO, J=7) and 6300 (S-CDO, J=8) over the reference implicitly-exhaustive search algorithm, and larger than 2000 (CDO, J=17) over the fastest known CDO validation function used in high-performance pseudo-random search algorithms.

These speedups are achieved through algorithmic enhancements in the deterministic search-space reduction, and a vastly improved validation function that makes use of a novel data structure for enabling data-reuse and incremental computations. The resulting validation function speedup is larger than 60000 (S-CDO, J=17) and 190000 (CDO, J=17) when compared to its reference implementation. The combination of optimizations and load-balancing techniques allowed us to leverage hundreds of processor cores in order to perform an exhaustive search over a search space that is some 10^{14} times larger than what was previously possible, yielding new and improved codes in a reasonable computation time.

We provide optimal-span CDO/S-CDO codes having orders $J \in \{6, ..., 9\}$ and $J \in \{9, ..., 12\}$ respectively, as well as CDO/S-CDO codes having the shortest spans published to date for $J \in \{10, ..., 17\}$ and for $J \in \{13, ..., 20\}$ respectively. Using this algorithm, we were able to reduce the spans of these codes by an average of 14% for CDO codes and by an average of 26% for S-CDO codes, resulting in a latency reduction of the same magnitude in the error-correcting systems for which they are intended.

We compare the spans of our new codes to known theoretical lower-bounds, and provide the error-correction performance for some of these codes, along with the span improvements obtained when using S-CDO codes instead of CDO codes of the same order. We show that although CDO codes perform slightly better than S-CDO codes, from an engineering point of view, S-CDO codes clearly offer a much lower decoding latency for a similar error performance. We also confirm that for moderate E_b/N_0 values (i.e. $E_b/N_0 > 3$dB), CDO/S-CDO codes do offer a competitive error performance and a compelling alternative to Turbo codes, since their error performance curves yield a lower "floor" region than that of Turbo codes, thus providing a better error performance along with a lower latency and reduced implementation complexity.

Finally, we present the evolution of the error performance of CDO/S-CDO codes as a function of their order J. We show that although the greater the value of J, the better the error performance, this is achieved at the cost of having the "waterfall" region of the error performance move to higher values of E_b/N_0.

CONDENSÉ EN FRANÇAIS

Introduction

La *théorie de l'information* trouve son origine scientifique dans l'article révolutionnaire publié en 1948 par l'ingénieur électricien et mathématicien américain *Claude E. Shannon* [1]. Ce domaine s'intéresse, entre autres, au transfert fiable d'informations sur un canal de communication bruité. En effet, un média de transmission d'information (ou *canal de communication*) présente certaines propriétés physiques qui vont introduire des erreurs dans les messages transmis entre un émetteur et un récepteur.

Afin d'augmenter l'efficacité et la fiabilité des transmissions de données numériques, il est possible d'utiliser des codes correcteurs d'erreur [2, 3]. Cette technique de codage, basée sur la redondance, consiste à « encoder » les messages en ajoutant des symboles de parité aux bits d'information transmis. Les symboles de parité sont générés par un *codeur de canal* à partir des bits d'information et de certaines règles précises préétablies. À la réception, ces symboles de parité seront utilisés par un décodeur pour détecter et éventuellement corriger un certain nombre d'erreurs se produisant dans la transmission. De plus, lors de la transmission de données numériques dans un canal très bruité, ces *codes* permettent de conserver un taux d'erreur faible tout en augmentant le débit de transmission et/ou en diminuant la puissance d'émission requise. Ainsi, ils contribuent à améliorer l'efficacité énergétique des dispositifs électroniques et, par conséquent, à prolonger leur autonomie.

Un code est dit *systématique* lorsque la séquence d'information à l'entrée du codeur se retrouve à l'une de ses sorties sans avoir été modifiée [1]. Le *gain de codage* d'un code est défini, pour un même *taux d'erreur* (BER), comme la différence entre le *rapport signal sur bruit* (SNR) d'une transmission non-codée et le SNR d'une transmission encodée avec ce code [3]. De plus, le *taux de codage* d'un code est défini comme $R = \frac{k}{n}$, où k est le nombre de bits d'information à l'entrée du codeur de canal, et n est le nombre de bits transmis à la sortie de cet encodeur [3]. Par conséquent, un codeur de canal systématique ajoutera $n - k$ bits redondants aux k bits d'information à son entrée. Enfin, un *code* est dit *convolutionnel* lorsque les symboles de parité ajoutés dépendent non seulement des bits d'information à l'entrée, mais aussi des bits d'information précédemment émis [1].

Dans cette *thèse par articles*, nous présentons un algorithme de recherche efficace pour trouver deux types de codes systématiques de taux de codage $R = \frac{1}{2}$: les codes convolutionnels doublement orthogonaux (CDO) [4] et les codes convolutionnels doublement orthogonaux simplifiés (S-CDO) [5]. En effet, ces codes sont utilisés dans un système de contrôle d'erreurs [6, 7] ayant un décodage à seuil itératif différent de la procédure de décodage Turbo classique [8, 9], puisqu'il ne nécessite aucun entrelaceur, ni à l'encodage, ni aux étapes de décodage. Néanmoins, son processus de décodage à seuil nécessite que ces codes convolutionnels systématiques satisfassent des propriétés dites de « double orthogonalité » [4, 5], allant au-delà des conditions requises par les codes « simplement orthogonaux » [10], bien connus et habituellement utilisés lors d'un décodage à seuil non-itératif [11]. Afin de pouvoir construire des *codecs* à haute performance et à faible latence avec ces codes, il est important de minimiser leur *longueur de contrainte* ou « *span* » pour un nombre J de connexions donné [4]. Bien que trouver des codes CDO et S-CDO ne soit pas difficile, déterminer les codes ayant un span *minimal* (dit *optimal*) pour un *ordre* J donné est mathématiquement très complexe. En effet, la construction directe de codes CDO / S-CDO à span court/optimal reste un problème ouvert et qui est soupçonné d'être *NP-complet* [12].

Objectifs de recherche

Nous nous concentrons sur le développement d'un algorithme pour la recherche de deux types de codes systématiques à taux de codage $R = \frac{1}{2}$: les codes Convolutionels Doublement Orthogonaux (CDO), et les codes CDO simplifiés (S-CDO). De part leur définition, ces codes correcteurs d'erreur doivent satisfaire certaines conditions de « double orthogonalité » [4, 5]. Leur *performance d'erreur* dépend surtout de leur « ordre » J, le nombre de connexions reliant l'additionneur modulo-2 au registre à décalage du codeur, et leur *latence* de décodage est proportionnelle au « span » du code, c.à.d. à la longueur du registre à décalage du codeur [4]. Par conséquent, afin de pouvoir construire des *codecs* à haute performance et à faible latence avec ces codes, il est important de minimiser leur span pour un ordre J donné, un problème complexe qui est soupçonné d'être *NP-complet* [12], et qui est relié à la recherche de règles de Golomb optimales [13].

Les objectifs de recherche sont donc de :

1. développer et mettre en oeuvre un algorithme de recherche haute-performance et efficace pour trouver de nouveaux codes CDO et S-CDO ayant un span optimal et/ou un span plus court que tout autre code publié précédemment pour un même ordre J ;
2. trouver, pour $J \leq 20$, de nouveaux codes CDO/S-CDO à span optimal, et de nouveaux codes CDO/S-CDO ayant un span plus court que tout autre code publié auparavant ;
3. caractériser la performance de correction d'erreurs de ces nouveaux codes, ainsi que l'évolution de leur performance d'erreur en fonction de l'augmentation de J.

Algorithme de recherche efficace, parallèle et implicitement-exhaustif

Nous présentons un nouvel algorithme de recherche haute-performance *efficace*, *parallèle* et *implicitement-exhaustif* pour la recherche de nouveaux codes CDO et S-CDO systématiques à taux $R = \frac{1}{2}$, et ayant un span court, voire optimal. L'algorithme de recherche proposé est beaucoup plus rapide que les meilleurs algorithmes de recherche exhaustifs et pseudo-aléatoires existant auparavant, et il utilise des techniques analytiques et d'ingénierie informatique pour offrir des améliorations synergiques importantes conduisant à trouver de nouveaux codes ayant un span amélioré (c.à.d. plus court).

L'algorithme de recherche effectue, de façon plus efficace, un parcours *implicitement-exhaustif* de l'arbre de recherche : il applique, de façon *dynamique*, des techniques d'élagage identifiant et ciblant la recherche *uniquement* sur les codes potentiellement valides, permettant ainsi de réduire la taille de l'espace de recherche. Afin de faciliter l'élagage, des bornes *inférieures*, *de point milieu* et *supérieures* sont définies pour les noeuds de l'arbre de recherche, diminuant ainsi de plusieurs ordres de grandeur la complexité de la recherche. L'algorithme de recherche proposé est un type *d'algorithme par séparation et évaluation*[9] : bien qu'il ne teste pas toutes les branches de l'arbre de recherche, il réalise effectivement une recherche exhaustive, assurant ainsi que le span des codes trouvés à la fin de la recherche soit optimal. En effet, compléter une recherche exhaustive dans un temps de calcul raisonnable n'était possible auparavant que pour des codes ayant une très faible valeur de J.

[9]En anglais : « branch and bound » algorithm.

L'algorithme de recherche proposé utilise une fonction de validation de codes CDO/S-CDO considérablement améliorée, et qui emploie une *nouvelle structure de données* pour effectuer, de façon efficace, un *calcul incrémental* et une *réutilisation des données* déjà calculées. En effet, cette structure de données permet de faire un suivi, avec une complexité temporelle en $O(1)$, des résultats de calcul partiels pertinents, facilitant ainsi la réutilisation des données : pour chaque nouveau code à valider, la fonction de validation proposée ne calcule que les *résultats partiels qui sont nouveaux* par rapport à la validation du code précédent, et réutilise les résultats partiels communs aux deux validations, déjà stockés dans la structure de données. De ce fait, le degré de l'équation polynomiale décrivant le nombre de calculs partiels à effectuer pour chaque validation est *réduit de un*, c.à.d. de J^4 à J^3.

La nouvelle fonction de validation met l'emphase sur une *invalidation rapide* de codes ne satisfaisant pas les conditions requises, assurant ainsi que, lors du processus de validation, un mauvais code puisse être éliminé aussitôt que possible. De plus, en utilisant des techniques de méta-programmation lors de la compilation, nous éliminons les boucles et les branchements dans la fonction de validation, éliminant ainsi aussi les pénalités associées à une mauvaise prédiction des branchements dans les microprocesseurs modernes.

Afin de réduire encore plus le temps de calcul, l'algorithme effectue une recherche *parallèle et coopérative*, de sorte à calculer plus de branches d'arbre de recherche en même temps et ainsi pouvoir converger plus rapidement vers un arbre de recherche réduit. En effet, l'algorithme de recherche a une très bonne *capacité de monter en charge* : en utilisant une *technique d'équilibrage de charge* efficace, il est capable de profiter de la puissance offerte par plusieurs centaines de coeurs de calcul. En outre, pour compenser le faible *temps moyen entre pannes*[10] des ordinateurs effectuant la recherche, l'algorithme de recherche proposé met en oeuvre des mesures basiques de *tolérance aux pannes* : des instantanés de l'état actuel de la recherche sont stockés régulièrement, permettant ainsi que celle-ci soit arrêtée et redémarrée sans perte significative de progrès. Les instantanés sont enregistrés dans un *format XML vérifiable*, garantissant ainsi la possibilité d'une récupération en cas de corruption de fichiers, et permettant la reprise de la recherche après un plantage ou un redémarrage du système.

Nous caractérisons, par rapport aux algorithmes publiés antérieurement, l'accélération

[10]En anglais : MTBF, or « Mean Time Between Failures ».

spectaculaire obtenue avec le nouvel algorithme de recherche. En utilisant la combinaison d'optimisations algorithmiques et les techniques d'équilibrage de charge décrites dans cette thèse, nous avons été en mesure de compléter une recherche exhaustive sur un espace de recherche environ 10^{14} fois plus grand qu'auparavant. En effet, comparé à l'algorithme de recherche implicitement-exhaustif de référence, l'algorithme de recherche à haute performance proposé reste exhaustif mais fournit un facteur d'accélération très important, supérieur à 16300 pour les codes CDO d'ordre $J = 7$, et supérieur à 6300 pour les codes S-CDO d'ordre $J = 8$. En outre, comparé à la fonction de validation de codes CDO et S-CDO de référence et qui est utilisée dans les algorithmes de recherche exhaustifs, cette nouvelle fonction de validation offre un facteur d'accélération très substantiel, supérieur à 190000 pour les codes CDO d'ordre $J = 17$, et supérieur à 60000 pour les codes S-CDO d'ordre $J = 17$. Enfin, comparativement à la fonction de validation de codes CDO la plus rapide et qui est utilisée dans les algorithmes de recherche pseudo-aléatoires à haute performance, la fonction de validation proposée offre un facteur d'accélération supérieur à 2000 pour les codes CDO d'ordre $J = 17$.

Nouveaux codes CDO et S-CDO obtenus

Nous fournissons de nouveaux codes CDO et S-CDO systématiques à taux $R = \frac{1}{2}$ et ayant un span plus court que tout autre code du même ordre J publié auparavant. En utilisant le nouvel *algorithme de recherche efficace, parallèle et implicitement-exhaustif*, nous avons pu déterminer de nouveaux codes CDO à span optimal d'ordre $J \in \{6, 7, 8, 9\}$, et de nouveaux codes S-CDO à span optimal d'ordre $J \in \{9, 10, 11, 12\}$. De plus, nous avons pu trouver plusieurs nouveaux codes CDO et S-CDO ayant les spans les plus courts publiés à ce jour pour $J \in [10; 17]$ et $J \in [13; 20]$ respectivement. Grâce à cet algorithme, la réduction maximale de la longueur de span obtenue fut de 32% pour les codes CDO, et de 34% pour les codes S-CDO. Par ailleurs, nous avons obtenu une réduction de la longueur de span de 14% en moyenne pour les codes CDO, et de 26% en moyenne pour les codes S-CDO. Bien entendu, dans les systèmes de contrôle d'erreurs utilisant ce type de codes, ces améliorations en longueur de span obtenues se traduiront par une réduction de la même ampleur en latence de décodage.

Nous décrivons aussi certaines des caractéristiques des codes CDO et S-CDO obtenus. Nous comparons le span de ces nouveaux codes aux bornes inférieures théoriques connues, et présentons les performances de correction d'erreur de certains de ces codes, ainsi que l'amélioration en longueur de span obtenue lorsque l'on utilise un code S-CDO au lieu d'un code CDO ayant le même ordre. Nous montrons que, bien que les codes CDO offrent une performance d'erreur légèrement supérieure, d'un point de vue strictement technique, les codes S-CDO offrent clairement des avantages : une latence de décodage beaucoup plus faible pour une performance d'erreur semblable. Nous confirmons également que, pour des valeurs de E_b/N_0 modérées (c.à.d. $E_b/N_0 > 3$ dB), les codes CDO et S-CDO offrent une performance d'erreur concurrentielle aux codes Turbo et par conséquent une alternative convaincante : leurs courbes de performance d'erreur ont une région « plancher[11] » plus basse que celle des codes Turbo, fournissant ainsi une meilleure performance d'erreur tout en ayant une latence de décodage inférieure, et permettant une mise en oeuvre moins complexe.

Evolution de la performance d'erreur des codes CDO/S-CDO

Nous présentons l'évolution de la performance d'erreur des codes CDO et S-CDO en fonction de leur ordre J. Nous montrons que bien qu'une augmentation de la valeur de J conduise à une amélioration de la performance d'erreur, cela est réussi au prix d'un déplacement de la zone « cascade[12] » des performances d'erreur à une région où E_b/N_0 a une valeur plus élevée, ce qui devra être pris en considération selon l'application concernée. En effet, il est possible que, pour un ordre $J > 20$, il ne soit pas avantageux d'utiliser des codes CDO ou S-CDO puisque la région « cascade » se retrouverait dans une région où E_b/N_0 a une valeur trop élevée pour l'usage prévu.

Conclusions et Recommandations

Dans cette thèse par articles, nous présentons deux articles publiés dans *IEEE Transactions on Communications* [14, 15], et un article soumis pour publication à *IEEE Transactions on Parallel and Distributed Systems* [16].

L'algorithme de recherche haute performance que nous avons développé et présenté offre un très grand facteur d'accélération comparé à l'algorithme de référence, et nous a permis

[11]En anglais : « floor » region.
[12]En anglais : « waterfall » region.

de trouver de nouveaux codes CDO / S-CDO systématiques à taux de codage $R = \frac{1}{2}$ ayant un span plus court que tout autre code du même ordre $J \leq 20$ publié auparavant. Nous avons aussi trouvé plusieurs nouveaux codes à span optimal, ayant pu traverser un espace de recherche 10^{14} fois plus grand que ce qui était possible auparavant. Nous avons caractérisé la performance de correction d'erreur de ces nouveaux codes, ainsi que l'évolution de leur performance d'erreur en fonction de l'augmentation de leur ordre J. Notre analyse révèle la complexité et les enjeux de ce sujet et suggère que des conclusions importantes peuvent être attendues suite à une enquête plus approfondie.

En effet, nous estimons que de nombreuses améliorations peuvent être apportées à l'algorithme de recherche présenté, et que plusieurs outils peuvent être développés pour aider à mieux comprendre les codes CDO et leurs variantes.

Par exemple, des résultats préliminaires ont montré qu'une réduction du temps de calcul d'environ 18% peut être obtenue en réarrangeant l'ordre des calculs partiels générés dans la fonction de validation. De plus, le développement d'un algorithme de recherche de codes CDO/S-CDO basé sur le *Shift Algorithm* [17] permettrait d'obtenir un facteur d'accélération encore plus important.

Nous recommandons le développement d'un nouveau simulateur de performance de correction d'erreurs pour les codes CDO et S-CDO. En effet, le simulateur actuel est très lent et limite notre capacité à simuler la performance d'erreur des codes CDO/S-CDO pour des valeurs de SNR supérieures à $\frac{E_b}{N_0} > 4.0$ dB.

Enfin, nous recommandons d'adapter autant l'algorithme de recherche comme le simulateur pour pouvoir supporter les codes Convolutionnels Doublement Orthogonaux Récursifs (RCDO) [18]. En effet, à faible SNR, ces codes et leurs variantes offrent une bien meilleure performance d'erreur comparativement aux codes CDO/S-CDO traditionnels [4].

Ces codes RCDO haute performance seront utilisés dans le développement de systèmes de correction d'erreurs encore plus puissants et efficaces.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Forward Error Correction	FEC
Signal-to-Noise Ratio	SNR
Bit Error Rate	BER
Convolutional Self-Doubly Orthogonal	CDO
Simplified Convolutional Self-Doubly Orthogonal	S-CDO
CDO and/or S-CDO	(S-)CDO
Central Processing Unit	CPU
Graphics Processing Unit	GPU
Field-Programmable Gate Array	FPGA
Garry/Vanderschel ANTenna	GVANT
Garry's Adaptation of Rado's Searching Principles	GARSP
Feiri/Levet Enhanced Garsp Engine	FLEGE
Lookup Table	LUT
Unique Identifier	UID
Identification	ID
Computational Improvement Rate	CIR
Mean Time Before Failures	MTBF
General-Purpose computing on GPUs	GPGPU
Recursive Convolutional Self-Doubly Orthogonal	RCDO
Simplified Recursive Convolutional Self-Doubly Orthogonal	S-RCDO
RCDO and/or S-RCDO	(S-)RCDO

CHAPTER 1

INTRODUCTION

1.1 Background

In 1948, the groundbreaking paper presented by *Claude E. Shannon*, an American mathematician and electronics engineer, gave birth to the field of *Information Theory* [1]. A central paradigm in this field is the engineering problem of the reliable transmission of information over a noisy channel. Indeed, a transmission medium (or *communication channel*), has certain physical characteristics that will introduce errors into the information flow as it travels from the transmitter to the receiver. For example, in Fig. 1.1, the information (y_i) obtained at the Receiver, is the data (u_i) sent by the Transmitter, but corrupted by the errors (e_i) induced by the Noisy Communication Channel.

In order to increase the reliability of data transmissions, *forward error correction* (FEC, or *channel coding*) may be used [2]. In this scheme, the sender "encodes" the information by adding systematically generated redundant parity check symbols to its messages. These additional symbols are then used at the receiving end to detect and/or correct a limited number of channel errors occurring in the transmission. For example, in Fig. 1.2, the Encoder adds redundant symbols (v_i) to the data stream (u_i) sent by the Transmitter. The Noisy Communication Channel adds errors to the original data stream $\{u_i, v_i\}$, resulting in a data stream $\{\tilde{u}_i, \widetilde{v}_i\}$ containing errors. The redundant symbols are then used by the Decoder to detect and/or correct up to a certain limited number of errors within a given interval of the data stream. Following the error-correction scheme at the Decoder, the decoded sequence

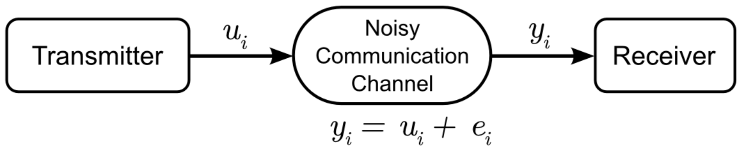


Figure 1.1 Simplified diagram depicting a communication over a noisy channel.

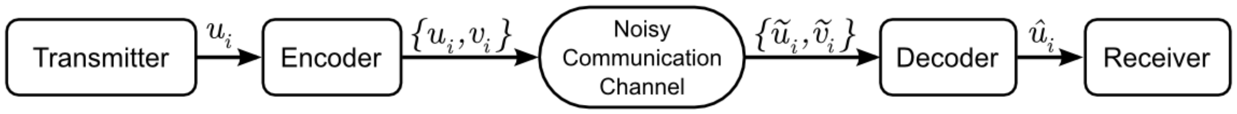


Figure 1.2 Simplified diagram depicting a communication using forward error correction over a noisy channel.

and "most likely value of u_i", $\hat{u}_i$, is then delivered to the Receiver.

In his paper [23], Shannon presents the *noisy-channel coding theorem*, which states that as long as the transmission rate is kept below some computable maximum rate, it is possible to use a sophisticated coding technique to communicate discrete data (or digital information) nearly error-free through a noisy channel [3]. The theorem describes the maximum information transfer rate of a channel for a given noise level, and thus the maximum possible efficiency of error-correcting methods for that given noise level: this theoretical threshold is known as the *Shannon limit* (or *channel capacity*) [3]. The *channel capacity*, expressed in *bits/s*, can be calculated from the physical properties of a transmission medium.

Unfortunately, Shannon's work does not provide a description on how to construct error-correcting codes and systems reaching this efficiency, a problem which gave rise to the field of *Coding Theory*, and more particularly *Channel Coding Theory* (a sub-field of *Coding Theory*).

1.2 Channel Coding Theory - A Quick Overview

The main goal of *Channel Coding Theory* is to find codes and encoding/decoding methods that allow reliable and efficient data transmissions over different types of communication channels, that is, finding codes that minimize the effect of the channel noise and allow the transmission of data with an arbitrarily small coding error at a rate near the *channel capacity* [24].

Indeed, depending on the transmission channel, the codes will require different properties. For example, deep space communications are affected by thermal noise, which is of a continuous nature; DVDs on the other hand will encounter bursts of errors wherever dust or scratches are present; and the high frequencies used by cell phones can cause rapid fading of the signal [3, 25]. As long as the transmission rate is below the *Shannon capacity*, the maximum ratio of the number of errors that can be corrected over the total number of transmitted symbols is determined by the design of the FEC code [26]. Therefore, based on the type of application that is of interest, different codes and/or code combinations may be suitable.

In addition to offering more reliable data transmissions, the use of FEC also provides other significant advantages: for example, it opens the door to higher data-rate communications that would otherwise be impossible with uncoded transmissions; it also allows for communications over very long distances to take place, which is useful for space exploration and high-speed transcontinental data links; finally, mobile devices (such as smartphones and tablets) can use FEC to reduce their transmission power in wireless communications, and thus extend their battery life. Therefore, finding new and better codes is of critical importance, especially given that mobile devices requiring high data-rate communications are becoming mainstream.

A very brief overview of some *forward error correction* terms and two common types of codes, *block codes* and *convolutional codes,* is now presented.

1.2.1 Some Forward Error Correction Terms

Forward Error Correction (FEC) follows a predetermined algorithm to add just the right kind of redundancy needed to efficiently and reliably transmit data across a noisy communication channel [3]. Indeed, these redundant bits can be used to reconstruct the original data, were it to contain errors.

A code is said to be *systematic* if the original information sequence can be found in the encoded output. Otherwise, it is said to be *non-systematic* [1]. The *coding gain* of a code is defined as the measure in the difference between the *signal-to-noise ratio* (SNR) level of an uncoded transmission, and the SNR level of the coded transmission at the same *bit error rate* (BER) [3]. Furthermore, the *coding rate* is defined as $R = \frac{k}{n}$, where k is the number of information bits at the input of the encoder, and n is the number of transmitted bits at the output of the encoder [3]. Thus, a systematic encoder will add $n - k$ redundancy bits to the k information bits at its input.

In this thesis, we will mainly be dealing with rate $R = \frac{1}{2}$ *systematic codes.*

1.2.2 Block Codes

As their name implies, *block codes* work on fixed-size blocks of data of predetermined size [1]. During the encoding, a message is divided into a set of fixed-length sequences called information blocks. Each block is encoded separately, and with the added redundancy bits, it will form a larger fixed-length block that will be transmitted over the noisy channel. Since blocks are independent of each other, practical implementations of these codes are able to make heavy use of parallel processing techniques [27].

1.2.3 Convolutional Codes

Convolutional codes work on bit streams of arbitrary length: the added redundancy bits are computed as a function of the last k input bits in the stream [1]. Their main advantage is that they tend to offer a greater simplicity of implementation than block codes of equal power [3]. A more in-depth description of a few types of convolutional codes is presented in Chapter 2.

1.3 Research Objectives

We focus on developing a search algorithm for finding optimal/short-span rate $R = \frac{1}{2}$ systematic *Convolutional Self-Doubly Orthogonal* (CDO) codes and *Simplified Convolutional Self-Doubly Orthogonal* (S-CDO) codes. These convolutional error-correcting codes, described in Chapter 2, must satisfy some "double orthogonality properties", beyond those of the well-known orthogonal codes [10]. The error-correcting performance of these codes depends mostly on J, the number of generator connections (also known as the "order" of the code), and their decoding latency is proportional to their memory length (also known as the "span" of the code) [4]. Therefore, in order to build high-performance/low-latency codecs with these codes, it is important to minimize their span for a given order J. While finding CDO/S-CDO codes is relatively easy, determining shortest-span codes for a given order J is computationally very challenging. In fact, the direct construction of optimal-span (or shortest-span) CDO and S-CDO codes has so far eluded analysis and the search for these codes is believed to be an *NP-complete* problem (see Chapter 2).

The research objectives for this thesis will thusly involve:

1. Developing and implementing an efficient high-performance search algorithm for finding new optimal-span CDO and S-CDO codes, and new CDO/S-CDO codes having shorter spans than any previously published codes for the same order J.

2. Finding, for $J \leq 20$, novel optimal-span codes and new codes with spans that are shorter than previously published codes.

3. Characterizing the error-correcting performance of these novel codes, as well as the evolution of their error performance as J increases.

1.4 Research Contributions

In this thesis, three articles are presented:

1. G. Kowarzyk, N. Bélanger, D. Haccoun, and Y. Savaria, "Efficient Search Algorithm for Determining Optimal R=1/2 Systematic Convolutional Self-Doubly Orthogonal Codes," *IEEE Transactions on Communications*, vol. 60, no. 1, pp. 3-8, 2012.

2. G. Kowarzyk, N. Bélanger, D. Haccoun, and Y. Savaria, "Efficient Parallel Search Algorithm for Determining Optimal R=1/2 Systematic Convolutional Self-Doubly Orthogonal Codes," *IEEE Transactions on Communications*, vol. 61, no. 3, pp. 865-876, 2013.

3. G. Kowarzyk, N. Bélanger, D. Haccoun, and Y. Savaria, "Optimizing the Parallel Tree-Search for Finding Shortest-Span CDO Codes of Order J," *IEEE Transactions on Parallel and Distributed Systems - submitted August 18, 2013.*

The first article [15] proposes an *efficient implicitly-exhaustive search algorithm* that applies *dynamic search-space reduction techniques* to yield new optimal-span CDO and S-CDO codes ($J \in \{6, 8, 9\}$ and $J \in \{9\}$ respectively), and new codes having shorter spans than any published codes of this class with the same order ($J \in \{10, 11\}$ and $J \in \{14, 15\}$ respectively). The error-correction performance of some of these codes is shown and their spans are compared to known bounds.

In the second article [14], we present a high-level overview of the *high-performance parallel and efficient implicitly-exhaustive search algorithm* that we have developed. The novel search algorithm provides a very significant speedup over previous search algorithms. This is achieved through the use of a *stricter set of constraints* to identify and concentrate the search on only potentially valid codes, and by performing a *parallel search* using *incremental computation* with *data-reuse*. The novel algorithm was able to yield new optimal-span CDO/S-CDO codes having order $J \in \{9\}$ and $J \in \{10, 11\}$ respectively, and new codes with the shortest published spans having order $J \in [10; 20]$. Their span is compared to known bounds, and the error-correction performance for some of these codes is presented. Finally, the evolution of the error performance for CDO/S-CDO codes as a function of J, $J \leq 20$, is shown.

In the third article [16], we focus on describing the *optimization techniques* that were applied to the search algorithm in [14] to reduce the time required for finding optimal-span CDO/S-CDO codes. We *characterize the speedup* obtained and show that using the novel algorithm and its efficient implementation, a very substantial *speedup of more than four orders of magnitude* is achieved. We explain the method by which the codes are validated using a *novel data structure* to enable incremental computation and data-reuse. The combination of optimizations and load-balancing techniques allowed us to complete the search over a search-space that is some 10^{14} times larger than what was previously possible.

The list of research contributions presented in this *manuscript-based thesis* can be grouped into the following three categories:

1. The development of a novel parallel and implicitly-exhaustive search algorithm for determining optimal/short-span rate $R = \frac{1}{2}$ systematic CDO/S-CDO codes. The algorithm described in [14, 15, 16] features the following synergistic improvements that led to finding new and improved codes:

   (a) An improved search-tree traversal:

   - uses an *implicitly-exhaustive tree traversal* [14, 15];
   - executes a *parallel search* to further reduce the computation time [14, 16];

- uses an effective *load-balancing technique* to scale efficiently over hundreds of processing cores [14, 16].

(b) A drastically improved (S-)CDO code validation function [14, 16]:

- uses compile-time meta-programming techniques to *remove the branches and loops* in the validation function, thus eliminating the associated branch-misprediction penalty on modern microprocessors;
- computes only *one element* of the second-order difference pairs, thereby reducing the number of computed second-order differences by half;
- focuses on *invalidating* a code rather than validating a code, thus ensuring that a code is discarded as early as possible during the validation process;
- performs an *incremental computation* with *data-reuse.*

(c) *Basic fault-tolerance measures* to counteract the low mean time between failures of computers running the search:

- performs *regular snapshots* of the current state of the search, which are efficiently saved in a *verifiable* XML format;
- uses the XML state-files to allow for the search to be *stopped* and *resumed without a significant loss of progress.*

(d) Offers a *very significant speedup* [16] over previously published algorithms [5, 20, 21]:

- *overall speedup* factor for $J = 7$ CDO codes: > 16300;
- *overall speedup* factor for $J = 8$ S-CDO codes: > 6300;
- *validation function speedup* factor compared to reference algorithm:
  - CDO codes, $J = 17$: > 190000;
  - S-CDO codes, $J = 17$: > 60000;
- *validation function speedup* factor compared to fastest published CDO code-only validation function used in high-performance pseudo-random search algorithms, for CDO codes, $J = 17$: > 2000;

- *size of largest search space exhaustively searched*: completed the search over a search space that is some 10^{14} times larger than previously possible.

2. In [14, 15], we provide new optimal and short-span (S-)CDO codes that have a shorter span than previously published codes having the same order [20, 21, 28]:

   (a) CDO codes:

   - novel *optimal-span* CDO codes for $J \in \{6, 7, 8, 9\}$;
   - new *short-span* CDO codes for $J \in [10; 17]$;
   - *maximal span* reduction for CDO codes: 32%;
   - *average span* reduction for CDO codes: 14%.

   (b) S-CDO codes:

   - novel *optimal-span* S-CDO codes for $J \in \{9, 10, 11, 12\}$;
   - new *short-span* S-CDO codes for $J \in [13; 20]$;
   - *maximal span* reduction for S-CDO codes: 34%;
   - *average span* reduction for S-CDO codes: 26%.

3. In [14, 15], we describe some of the characteristics of these (S-)CDO codes:

   - their spans are compared to *known theoretical lower-bounds*;
   - the *bit error-correction performance* for some of these codes is presented, confirming that they offer an interesting alternative at medium SNR values ($\frac{E_b}{N_0} \geq 3$ dB);
   - the *evolution of their error-performance as J increases* is presented: although the error floor seems to be lowered as J becomes larger, the "*waterfall*" region progressively moves to higher $\frac{E_b}{N_0}$ values, a fact that will need to be considered when selecting one of these codes for use in a given application of interest.

1.5 Thesis Layout

This manuscript-based thesis is composed of seven chapters. Chapters 3, 4, and 5 introduce articles [15], [14] and [16] respectively. Following this introductory chapter, the document is subdivided as follows:

- In Chapter 2, we briefly define Turbo codes, Convolutional Self-Orthogonal codes (also known as *Golomb rulers*), Convolutional Self-Doubly Orthogonal (CDO) codes and Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes. A short literature review on search algorithms for finding optimal-span Golomb rulers is provided. We describe the reference CDO code pseudo-random search algorithm and the reference (S-)CDO code exhaustive search algorithm: their tree-traversal and validation function are briefly explained in order to position our work. Then, we introduce Chapters 3, 4, and 5, which correspond to the three papers for this manuscript-based thesis.

- In Chapter 3, we present a novel implicitly-exhaustive search algorithm for finding rate $R = \frac{1}{2}$ optimal-span CDO and S-CDO codes. The algorithm defines a set of tree-pruning techniques to reduce the size of the (S-)CDO code search space, thus yielding new optimal-span CDO and S-CDO codes for $J \in \{6, 7, 8\}$ and $J = 9$ respectively. Furthermore, we were able to find CDO codes ($J \in \{10, 11\}$) and S-CDO codes ($J \in \{14, 15\}$) having spans that are shorter than any previously published codes. The spans of these codes are compared to known theoretical bounds, and their error-correction performance is shown.

- In Chapter 4, we present an *efficient* and *parallel* implicitly-exhaustive search algorithm for determining rate $R = \frac{1}{2}$ optimal-span CDO and S-CDO codes. This novel algorithm uses a stricter set of tree-pruning techniques, and a *parallel execution* with *incremental computation* and *data reuse* to speed up the search and yield new codes. We provide a very high-level overview of the algorithm, and mainly focus on the codes that were obtained: new optimal-span CDO/S-CDO codes (having order $J = 9$ and $J \in \{10, 11\}$ respectively), as well as new codes having the shortest published spans for $J \in \{10, 12, ...17\}$ and $J \in \{12, ..., 20\}$ respectively. The new codes and their error-performance are provided, and an evolution of the CDO/S-CDO code error performance as J increases is presented.

- In Chapter 5, we describe the optimizations and enhancements used for the algorithm presented in Chapter 4, which led to a drastic reduction in the time required for finding optimal/short span CDO/S-CDO codes. The resulting high-performance parallel implementation provides a speedup over the reference implicitly-exhaustive search algorithm that is greater than 16300 for $J = 7$ CDO codes, and greater than 6300 for $J = 8$ S-CDO codes. We focus on the *vastly improved validation function*, which makes use of a novel data structure for enabling data-reuse and incremental computations, thus achieving a speedup greater than 190000 and 60000 for $J = 17$ CDO and S-CDO codes respectively. We also describe improvements made on the tree-traversal and load-balancing of computations, and show that the algorithm scales well with the number of processor cores used: the combination of techniques allowed us to leverage hundreds of processor cores in order to complete an exhaustive search over a search-space that is some 10^{14} times larger than what was previously possible.

- Chapter 6 presents a general discussion[1] of the thesis: it provides a brief overview of the overall objectives achieved, without going into the details discussed in previous chapters.

- Chapter 7 presents the concluding remarks of this thesis and discusses several suggestions for future work.

[1]As per the guidelines of the *Département de génie électrique*, *École Polytechnique de Montréal*.

CHAPTER 2

DEFINITIONS AND LITERATURE REVIEW

In this chapter, we first define a few types of codes that are of interest in the discussion: Convolutional Self-Orthogonal (CSO) codes, Turbo codes, Convolutional Self-Doubly Orthogonal (CDO) codes, and Simplified CDO (S-CDO) codes. Then, we provide an overview of search algorithms for finding optimal-span Golomb rulers, also known as optimal-span CSO codes. Finally, the reference CDO code pseudo-random search algorithm and the reference (S-)CDO code exhaustive search algorithm are described: their tree-traversal and validation function are briefly explained in order to position our work.

2.1 Definitions for some codes of interest

2.1.1 Convolutional Self-Orthogonal (CSO) codes

Systematic *Convolutional Self-Orthogonal* (CSO) codes were first introduced by *J. L. Massey* [11] in 1963. These codes have the advantage of offering a simple decoding scheme.

A systematic Convolutional Self-Orthogonal (CSO) code of coding rate $R = \frac{1}{2}$, order J, and span α_J is defined as the set $\Omega = \{\alpha_1, \alpha_2, \dots, \alpha_J\}$ of J ascendingly ordered positive integers ($\alpha_1 < \alpha_2 < ... < \alpha_J$) such that the elements in S, the set of first-order differences

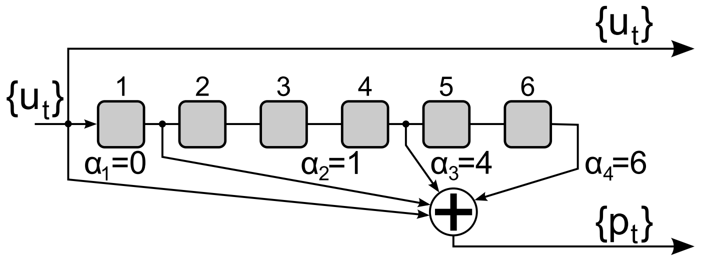


Figure 2.1 Example of a CSO code encoder: $R = \frac{1}{2}$, $J = 4$, $M = 6$, $\Omega = \{0, 1, 4, 6\}$.

between these integers, are all distinct:

$$S = \{s_{k,l} = (\alpha_k - \alpha_l) \ : \ k \neq l\}. \tag{2.1}$$

As with other systematic rate $R = \frac{1}{2}$ convolutional codes, the encoding can be done with a simple shift register (see Fig. 2.1): a parity check bit p_t is generated for every information input bit u_t by modulo-2 adding u_t to $J - 1$ register outputs within the shift register. The locations of these shift register "taps" are specified by the CSO code itself: each $\alpha_i = k$ element of the code represents the i-th connection going from the k-th register to the modulo-2 adder. The first element, α_1, has always the value zero and represents the connection between input bit u_t and the modulo-2 adder. The value of α_J represents the memory window, i.e. the number of information bits currently stored in the shift register. The information bit u_t and the redundant bit p_t are then multiplexed to form the encoded data stream.

Massey's decoder design focuses on simplicity [11]. Nevertheless, it is important to note that the error-correction power of the code is a function of d_{min}, the minimum distance of the code, where $d_{min} = J + 1$, and that the decoding latency is proportional to α_J, the span of the code [5]. Thus, in order to have a good error-correction performance and a low latency, one has to maximize J and minimize α_J, a problem related to the search of *optimal Golomb rulers*.

Golomb rulers are named after *Solomon W. Golomb* [13], an American mathematician and engineer, but they were also discovered independently by *Simon Szidon* (1932) [29] and *Wallace C. Babcock* (1953). Aside from being employed as CSO codes, they are of mathematical interest [30] and have found a surprising number of other uses: from rope cutting [31], to gaining insight into diffraction patterns that arise in x-ray crystallography [30] and the construction of spectrometers [32, 33], to being used in the fields of graceful graph labeling [30, 34] and numbered undirected graphs [35], and even for the development of optimal recovery schemes in fault-tolerant distributed computing [36, 37]. They are also used for reducing intermodulation distortion [38, 39], implementing carrier spacing in fiber optic systems [40], developing radars/sonars, generating Costas arrays [30, 41], and in radio astronomy and signal processing [42].

A *Golomb ruler* is defined as a set of marks at integer positions along an imaginary ruler, such that no two pairs of marks are the same distance apart [43]. By convention, the first mark is at position zero on the ruler. A Golomb ruler's *order*, J, is defined as the number of marks on the ruler, and its *length* (or *span*) is the largest distance between two of its marks. A Golomb ruler of order J is said to be *optimal* when no other ruler with a shorter length exists for that order. One can easily see that a Golomb ruler is in fact a convolutional self-orthogonal code of coding rate $R = \frac{1}{2}$. Thus, finding optimal length Golomb rulers of order J is equivalent to the problem of minimizing the span of a convolutional code of coding rate $R = \frac{1}{2}$ and order J.

Although creating a Golomb ruler is easy, finding *optimal* Golomb rulers is computationally very challenging and believed[1] to be *NP-complete* [46, 47]. Since this problem is also of interest to the fields of *Applied Physics* and *Mathematics*, massively parallel searches have been undertaken by the *Distributed.net OGR* project [13], and optimal Golomb rulers of orders up to 26 have been found. *Distributed.net* is currently searching for optimal golomb rulers of order 27, a computation that is expected to take 7 years to complete [13].

2.1.2 Turbo codes

C. Berrou, *A. Glavieux*, and *P. Thitimajshima* introduced *Turbo codes* in 1993. This novel class of high-performance FEC codes is able to approach the *Shannon limit* within a fraction of a decibel [8, 48]. The breakthrough BER performance was achieved through the use of two or more convolutional encoders and an interleaver, which is designed to make the encoder output sequences be statistically independent from each other [3]. Fig. 2.2 depicts a simplified diagram of a systematic Turbo encoder: note that the Interleaver ensures that the inputs at Encoder #1 and Encoder #2 are statistically independent from each other. A detailed description of its iterative soft-decision decoding algorithm is beyond the scope of this document [8]. Nevertheless it is important to understand that it comprises two decoders that will exchange information iteratively until a given number of iterations is reached. With each iteration, the estimate of the message bits improves, and usually it converges after some

[1]Although papers from the heuristics community [44, 45] claim that the *Golomb Ruler* problem is *NP-complete* or *NP-hard* [12], a mathematical proof of this is still an open problem.

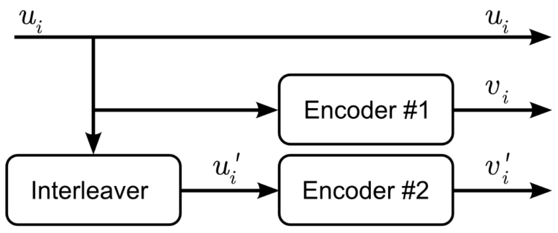


Figure 2.2 Simplified diagram of a systematic Turbo Encoder.

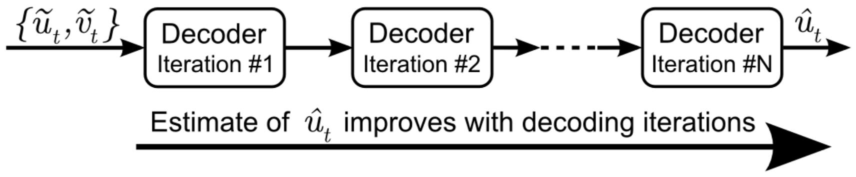

$\{\tilde{u}_t, \tilde{v}_t\}$
Decoder Iteration #1
Decoder Iteration #2
Decoder Iteration #N
$\hat{u}_t$
Estimate of $\hat{u}_t$ improves with decoding iterations

Figure 2.3 Iteratively decoding - one bit at a time.

number of iterations to the correct original information bits [3].

Although these codes offer an excellent error-correcting performance in low SNR environments, they suffer from three main drawbacks. First, their implementation complexity is relatively high. Second, because in order to obtain a good error performance they require a large interleaver (of several thousands of bits), as well as many iterations, these systems are plagued by a high decoding latency. Finally, their error-correcting performance hits an "error floor" at larger SNR values, where it is not possible to improve the BER even with more iterations: this may be unacceptable for data transmissions requiring very low bit error rates [5].

2.1.3 Convolutional Self-Doubly Orthogonal (CDO) codes and Simplified CDO (S-CDO) codes

The novel iterative error-control coding scheme presented in [4, 6, 7] differs from the classical Turbo code procedure invented in 1993 [8, 9], as it does not use any interleaver, neither at the encoding nor at the decoding process. The iterative threshold decoding algorithm it uses employs a new class of systematic convolutional codes that must satisfy double orthogonality properties, beyond those of the well-known orthogonal codes used in the usual non-iterative threshold decoding [10].

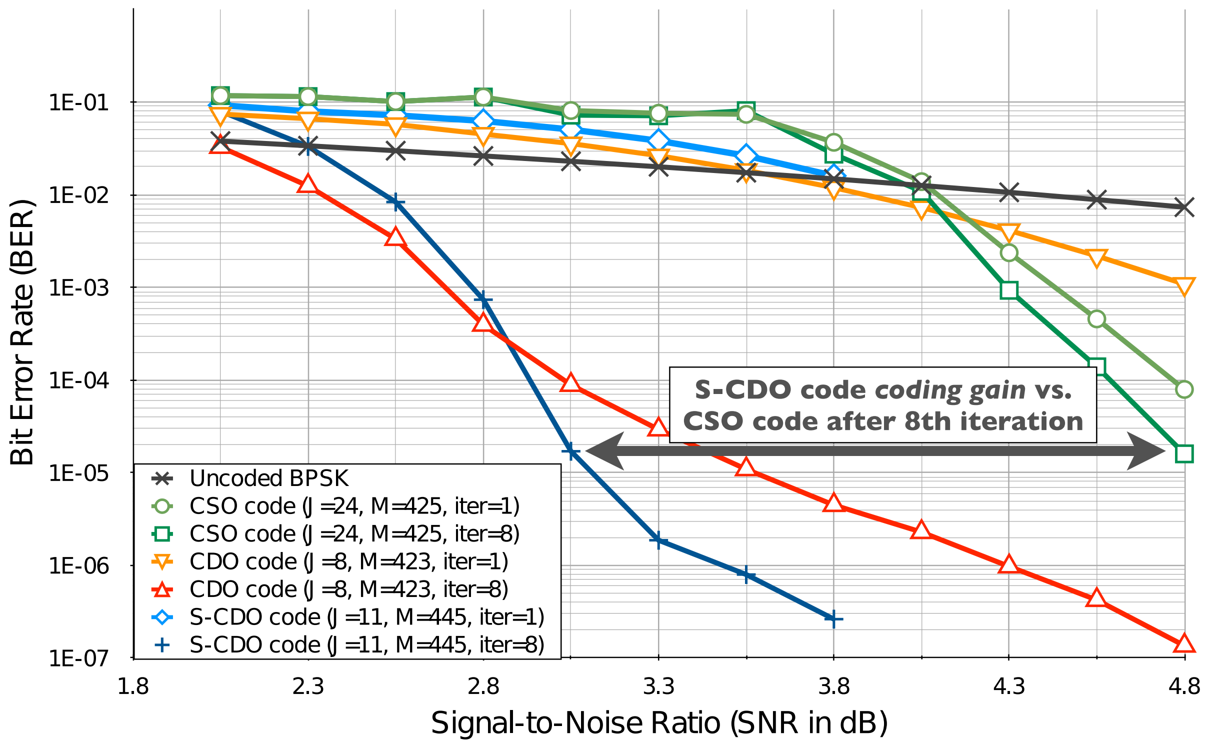


Figure 2.4 Error-correction performance after *one* and after *eight* decoding iterations for three rate $R = \frac{1}{2}$ systematic codes having a similar span value M, and $\frac{E_b}{N_0} \in [2.0; 4.8]$ (dB): a CSO code ($J = 24$), a CDO code ($J = 8$) and an S-CDO code ($J = 11$). CDO and S-CDO codes can benefit from iterative decoding, thus offering a significant coding gain over the CSO code.

The added so-called double orthogonality properties required from the codes ensure a quasi-independence of the observables over the first two decoding iterations, thereby allowing the use of an iterative decoding procedure (see Fig. 2.3), and hence attractive trade-offs between complexity, latency, and a good error performance [6]. Figure 2.4 shows, for $\frac{E_b}{N_0} \in [2.0; 4.8]$ (dB), the error-correction performance after *one* and after *eight* decoding iterations for three rate $R = \frac{1}{2}$ systematic codes having a similar span value M: a CSO code ($J = 24$, $M = 425$), a CDO code ($J = 8$, $M = 423$) and an S-CDO code ($J = 11$, $M = 445$). One can clearly see that iterative decoding does not significantly improve the error performance of the CSO code. However, for CDO and S-CDO codes, it allows for a considerable error-performance improvement, thus resulting in a coding gain of about 1.75 dB when compared to the CSO code.

As the error-correcting capability of CDO and S-CDO codes depends essentially on the

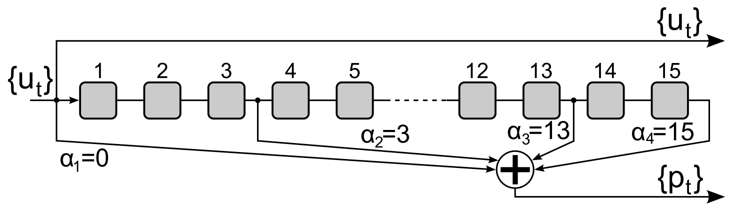


Figure 2.5 Example of a CDO code encoder: $R = \frac{1}{2}$, $J = 4$, $M = 15$, $\Omega = \{0, 3, 13, 15\}$.

number J, the dimension of the vector generator of the $R = \frac{1}{2}$ code [20], and because the code constraint length (or span of the code) has a direct impact on the latency of the system, it is of great interest to search for rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes having the shortest possible span for any given J number of connections. Since no systematic deterministic method for solving this problem is currently known, the code searching must be conducted using heuristic search algorithms [20, 28]. Although finding a CDO code is relatively easy, determining the shortest span codes for a given J has eluded analysis and is still an open problem. In fact, the search for optimal CDO codes (and their variants) is far more computationally challenging than the problem of finding “optimal” simply orthogonal codes (a.k.a. the *Golomb ruler* problem), which is believed[2] to be *NP-complete* [44, 45] or *NP-hard* [12]. Indeed, CDO codes may be viewed as second-order Golomb rulers.

Please note that since this is a *manuscript-based thesis*, articles are required[3] to be presented without modifications. Therefore, CDO and S-CDO code definitions will be provided several times: in this chapter (below), and then again in Chapters 3, 4, and 5.

2.1.3.1 Convolutional Self-Doubly Orthogonal (CDO) codes

A systematic Convolutional Self-Doubly Orthogonal (CDO) code of coding rate $R = \frac{1}{2}$ and order J is defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers ($\alpha_1 < \alpha_2 < ... < \alpha_J$) such that the following conditions are satisfied [6, 20, 21, 49]:

1. The elements in S, the set of first-order differences between these integers, are all distinct:

$$S = \{s_{k,l} = (\alpha_k - \alpha_l) \; : \; k \neq l\}; \tag{2.2}$$

[2]Please recall that a mathematical proof of this is still an open problem [46, 47].

[3]As per the guidelines of the *Département de génie électrique*, *École Polytechnique de Montréal*.

2. The elements in D, the set of second-order differences (the differences between the differences), are all distinct from one another, with the exception of the unavoidable differences caused by the permutations of indices (l, m) or (k, n):

$$D = \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \ : \\ k \neq l, m \neq n, k \neq m, l \neq n\}; \tag{2.3}$$

3. The elements in sets S and D are distinct from one another ($D \cap S = \emptyset$).

The α_i elements ($i \in [1; J]$) represent the connections between the encoder shift register and the modulo-2 adder, i.e. the generator connections. By convention α_1, the first integer in our set, is always equal to zero ($\alpha_1 = 0$). The span M of a CDO code is equal to α_J, the largest integer in Ω, and corresponds to the length of the encoder shift register (see Fig. 2.5), that is, α_J is the code memory length [20]. The number J of elements in Ω is equal to the number of generator connections of the code and is called the order of the CDO code.

An optimal CDO code of a given order J is defined as a CDO code whose span M_{opt} is the *smallest* span that exists for that order. However, an optimal CDO code may not be unique, and hence there may be more than one optimal CDO code of a given order J.

By definition, since the validity of a CDO code depends only on the relationship between the J successive elements composing it, any subset of L consecutive elements from the set defining a CDO code also forms a valid CDO code, albeit one of smaller order L, $L < J$. For example:

$$\begin{aligned} \text{CDO}_{J=5} &= \{0, 1, 24, 37, 53\} \\ \text{CDO}_{J=4} &= \{0, 1, 24, 37\} \\ \text{CDO}_{J=3} &= \{0, 1, 24\} \end{aligned} \tag{2.4}$$

are all valid, although not optimal, CDO codes. This property will be leveraged to speed-up the algorithms presented in this thesis.

When calculated directly as per the definition, first-order and second-order differences come in pairs of *equal magnitude* but *opposite sign.* The number of *positive* first-order differ-

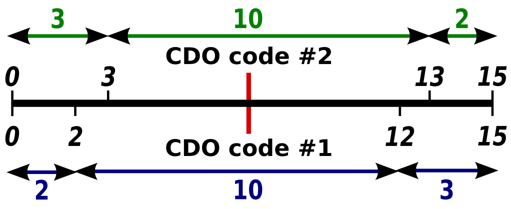


Figure 2.6 Golomb ruler symmetry: CDO code #1 and CDO code #2 are symmetrical (mirror) equivalents.

ences (N_S) and second-order differences (N_D) that exist for a code are a function of J, given by [20]:

$$N_S^J = \frac{J(J-1)}{2} \tag{2.5}$$

$$N_D^J = \frac{J(J^3-2J^2+3J-2)}{8}. \tag{2.6}$$

For simplicity, assuming that the cost of computing a first and a second order difference is the same, the total number of *positive* differences as a function of J is given by the sum of (2.5) and (2.6):

$$N_{S+D}^J = N_S^J + N_D^J = \frac{J^4-2J^3+7J^2-6J}{8}. \tag{2.7}$$

Recall that directly computing the exact span of optimal codes, whether simply or doubly orthogonal, is still an unsolved problem [13, 46]. However, a loose lower bound for the span of a CDO code has been developed in in [20, 50] and can be expressed as a function of J, the order of the code, using (2.7) as follows:

$$\alpha_J \geq \alpha_J^* = \left\lceil \frac{N_{S+D}^J}{2} \right\rceil. \tag{2.8}$$

Furthermore, any CDO code has a symmetrical (mirror) equivalent composed of integers with the same differences but in the reverse order, a property shared with the so-called *Golomb ruler* problem they are related to [20]: the symmetrical equivalent of {0, 2, 12, 15} would therefore be {0, 3, 13, 15} (see Fig. 2.6). This property will also be used to speed-up

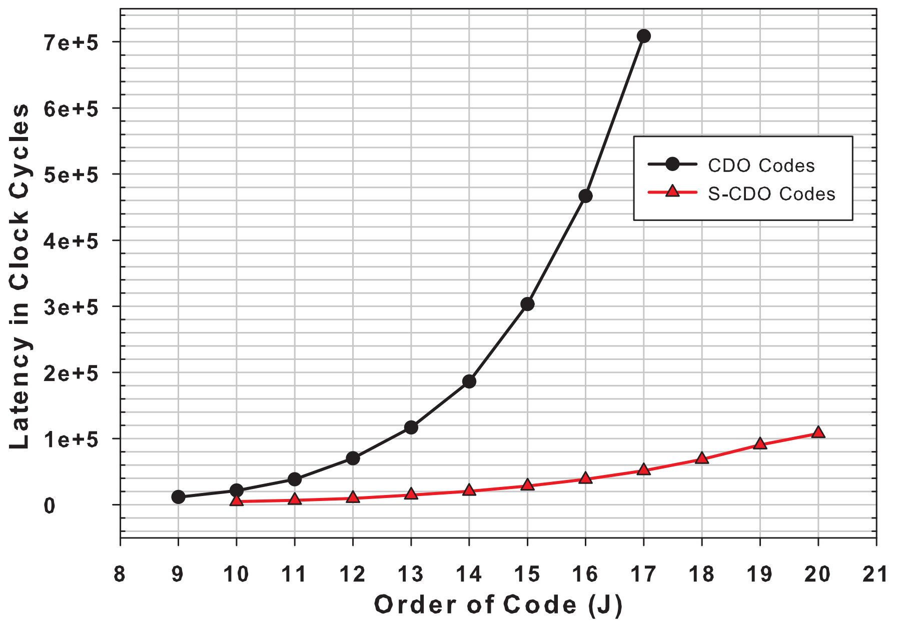


Figure 2.7 Approximate decoding latency of CDO and S-CDO codes after 14 decoding iterations (assuming one decoded bit per clock cycle).

the novel search algorithms presented in this thesis.

2.1.3.2 Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes

Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes are obtained by relaxing the second CDO condition, yielding codes with shorter spans than regular CDO codes. The latency of the decoding process is in direct proportion to the span of the code and the number of iterations used for reaching a given error performance. Therefore, using S-CDO instead of CDO codes, with the same number of iterations, leads to a substantially reduced decoding latency (see Fig. 2.7) at the cost of only a very small degradation of the error-correction performance [5, 21, 51].

A systematic S-CDO code of coding rate $R = \frac{1}{2}$ and order J is thus defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers such that it satisfies the *first* and *third* CDO conditions, and a *modified version of the second* condition, as follows [5, 51]:

2b) The set D of second-order differences between the integers in Ω, defined as:

$$\begin{aligned} D = \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n)\ : \\ k \neq l, m \neq n, k \neq m, l \neq n\} \end{aligned} \tag{2.9}$$

is composed of $2N_D$ second-order differences (of which $2N^e_D$ have an equal value in the set D), computed by excluding the unavoidable second-order differences caused by the permutations of indices (l, m) or (k, n). We define δ, the simplification coefficient, as:

$$\delta = \frac{N^e_D}{N_D} \tag{2.10}$$

where N^e_D is the number of second-order differences having an equal value in the set D, $N^e_D < N_D$ and $0 \leq \delta \leq 1 - \frac{N_S}{N_D}$ [5].

Clearly, a CDO code may be viewed as an S-CDO code for which $\delta = 0$, and thus S-CDO and CDO codes share most of their properties. A loose lower bound on the span of an S-CDO code has been derived as a function of J and δ, and is expressed, using (2.5) and (2.6), as [5, 21]:

$$\alpha^*_J = \left\lceil \frac{N_S + (1-\delta) \cdot N_D}{2} \right\rceil . \tag{2.11}$$

An optimal S-CDO code of order J and simplification coefficient δ is thus an S-CDO code having the smallest span, M_{opt}, which exists for that order and δ. Again, there may be more than one optimal S-CDO code for a given order J and simplification coefficient δ.

In this document, the notation "(S-)CDO" will be used when referring to both CDO and S-CDO codes. We now present a brief literature review on search algorithms for finding optimal-span Golomb rulers, followed by a short description of the tree traversal and validation function of the reference CDO code pseudo-random search algorithm and the reference (S-)CDO code exhaustive search algorithm.

2.2 Overview of Golomb ruler search algorithms

We recall that finding optimal length Golomb rulers of order J is equivalent to the problem of minimizing the span of a convolutional self-orthogonal code of coding rate $R = \frac{1}{2}$ and order J. Although several Golomb ruler construction methods are readily available [52], finding *optimal* span Golomb rulers (or *near-optimal* Golomb rulers) is computationally very challenging [12, 44, 45]. For example, the search for an optimal $J = 19$ Golomb ruler required approximately 36200 computing hours on a Sun Sparc Classic workstation using a very specialized algorithm [17], and the distributed and massively-parallel search for optimal span $J = 27$ Golomb rulers organized by *Distributed.net* is expected to necessitate several years of computing time [13].

Several heuristics however, *stochastic* and *deterministic*, have been developed in an attempt to improve Golomb ruler search algorithms:

- *stochastic* search algorithms, such as tabu search [53], evolutionary algorithms [44, 45, 54, 55, 56, 57], and hybrid techniques [58, 59, 60, 61, 62] have been used to find *near-optimal* (albeit not *optimal*) span Golomb rulers;

- *deterministic* (or *exhaustive*) search algorithms, based on constraint programming [63, 64], linear programming [65], or improved versions of the Shift Algorithm [66], have been able to either find or prove the optimality of the best Golomb rulers known to date ($J \leq 27$).

In this thesis, one of our objectives is finding new optimal-span (S-)CDO codes. Therefore, exhaustive search algorithms that have led to proving the optimality of Golomb rulers with the largest values of J currently known are of particular interest, since they would be the fastest exhaustive search algorithms.

The *Shift Algorithm*, which was invented by D. McCracken for his thesis, is derived from an algorithm that was first published in the december 1985 issue of the *Scientific American* magazine. In 1995, A. Dollas, W. T. Rankin and D. McCracken of Duke University employed an improved parallel version of the Shift Algorithm exploiting a reduced search space to prove the optimality of the shortest known $J = 19$ Golomb ruler [12, 17, 67]. In 1996, M. Garry and D. Vanderschel enhanced the Shift Algorithm, thus creating the *GVANT* algorithm, which was used to prove the optimality for the shortest known $J \in \{20, 21, 22, 23\}$ Golomb rulers. This algorithm was then significantly improved by M. Garry and R. Adorni, leading to the development of the *GARSP* algorithm, used by *Distributed.net* to prove the optimality of the shortest known $J \in \{24, 25\}$ Golomb rulers. Finally, in 2007/2008, M. Feiri and D. Levet developed the *FLEGE* algorithm [66], which brought significant enhancements to the *GARSP* algorithm by reducing the search space even further. The *FLEGE* algorithm was used to prove the optimality of the $J = 26$ Golomb ruler in 2009, and is currently being used for the optimality proof of the $J = 27$ Golomb ruler. These *Shift Algorithm* based algorithms are in fact quite clever and efficient: for example, for the *GARSP* and *FLEGE* algorithms, the distances, marks, and next-mark locations are represented as three bitmaps on which basic operations such as *SHIFTs* and bitwise *ORs* are performed [66]. The simplicity of these algorithms has led to the development of parallel hardware-software implementations [68, 69, 70], which have provided significant speedups for $J \leq 25$ compared to the software-only algorithms, but have been of limited use due to the cost and availability of large Field-Programmable Gate Array (FPGA) boards.

Although using a simple and efficient *FLEGE*-like algorithm for finding new optimal-span (S-)CDO codes would clearly allow for a high-performance implementation, correctly expressing and computing the first *and* second order differences with bitmaps and SHIFT/OR operations has proven to be very difficult. Unfortunately, this has led us to dropping the bitmap-shift based component of these algorithms in favor of the algorithms presented in this thesis.

2.3 Reference (S-)CDO code searching algorithms

Before presenting the novel tree-pruning and search-time reducing techniques that allowed us to find new optimal-span (S-)CDO codes, we describe the (S-)CDO code search space and briefly introduce some previous significant search algorithms applicable to this problem.

2.3.1 (S-)CDO Code Search Space

The algorithm described in this thesis performs the search for (S-)CDO codes using a tree-like structure (see Fig. 2.8 for $J = 3$). The *root node*[4] of the tree has always value 0 and is located at depth 0 of the tree. The rest of the tree is composed of nodes which must have a value

[4]The *root node* represents the *first* integer in our set Ω, i.e. α_1, which by convention has a value of *zero*.

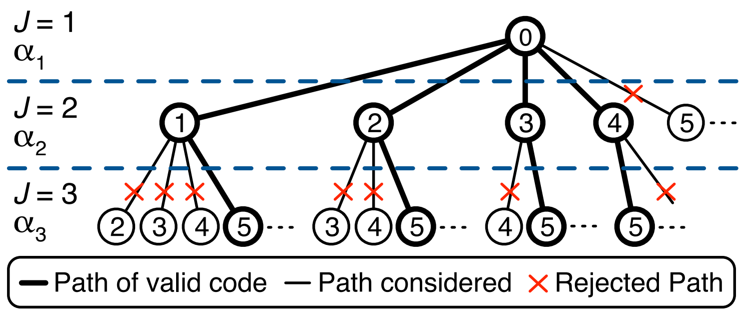


Figure 2.8 (S-)CDO search-tree - searching for a CDO with $J = 3$

larger than their parent node and their sibling[5] nodes to the left. The tree *depth* ranges from 0 to $J-1$, and represents the total number of connections J: all nodes at depth $J-1$ are *leaf-nodes*. The values of the nodes on a path from the root node to a leaf-node represent the elements of $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$, the set of positive integers defining the code. A *valid path* in this search-tree starts at the root node and ends at a leaf-node that has a value not larger than M_{curr}, the current and smallest known span value.

The total search-tree size depends on the current span, M_{curr}, and on J. There is only one path leading from the root node to a leaf-node. Thus, the number of leaf-nodes in the search-tree, N_L, represents the total number of possible paths (or (S-)CDO code candidates). Since the set Ω defining a code always starts with zero (the root node), the number of possible combinations of $J-1$ nodes with integer values smaller than or equal to M_{curr} may be expressed as "M_{curr} choose $J-1$". Hence, we can write:

$$N_L = \begin{pmatrix} M_{curr} \\ J-1 \end{pmatrix} = \frac{(M_{curr})!}{(J-1)!\,(M_{curr}-J+1)!} \tag{2.12}$$

Table 2.1 illustrates the number of leaf-nodes (and thus possible paths) on the (S-)CDO search-tree for different values of J and M_{curr}. It can be seen that the number of leaves quickly explodes as M_{curr} and J increase, thus making the search for optimal-span codes of order $J+1$ exponentially more complex. Clearly, tree-pruning techniques must be used as much as possible.

[5]Sibling nodes are nodes with the same parent node.

Table 2.1 Number of leaf-nodes as a function of J and M_{curr}

J	Example M_{curr} †	Number of leaf-nodes (or possible paths) ‡
3	5	10
4	15	455
5	41	101,270
6	100	75,287,520
7	222	155,308,696,543
8	459	813,381,183,503,226
9	912	11,509,802,721,558,333,270
10	1698	316,557,845,045,813,572,898,840
11	3467	68,254,814,954,346,235,795,154,297,723

† shortest known CDO code spans as per [5, 21]
‡ computed as per (2.1)

2.3.2 (S-)CDO code searching algorithms - pseudo-random vs. exhaustive

The first technique for finding CDO codes used a projective geometry approach to determine valid codes [50, 71]. However, this approach yielded codes with excessively large spans, thus requiring the development of new code-searching methods [20, 50].

Recent code-searching algorithms can be divided into two categories, *exhaustive* and *pseudo-random*, as discussed below:

- *exhaustive* search algorithms: this type of search guarantees, if a sufficiently large initial span M_{curr} is used, that the optimal span for a given J number of connections is found. This is done by testing all of the root-to-leaf node paths on the search-tree (see Fig. 2.8). However, for values of J larger than 7, the very rapidly increasing computational effort required to obtain optimal (S-)CDO codes led to dropping this type of algorithm in favor of more practical pseudo-random search algorithms [5, 50];

- *pseudo-random* search algorithms: these searching techniques are based on the use of a pseudo-random rejection criterion that can easily be modified and tuned in order to shorten the spans of the codes obtained [20]. Note that this type of algorithm cannot guarantee that minimal-span codes have been found. With the use of a span reduction method based on modulo operations [50] and a carefully chosen lower bound, the technique has provided codes with some of the shortest known spans, albeit possibly not optimal codes [5, 20]. The highest order for which an exhaustive search has been completed is $J = 5$ for CDO codes [50] and $J = 8$ for S-CDO codes [5].

Since one of our research objectives is to find new optimal-span (S-)CDO codes, or at least new codes with shorter span values than previously reported codes for a same order J, we focus on developing methods for further improving the computational performance of the *exhaustive search* algorithms. A short overview of the reference algorithms follows.

2.3.2.1 Pseudo-Random search algorithms

2.3.2.1.1 The state of pseudo-random search algorithms A first algorithm was proposed by B. Baechler in 2000 [50, 71]. It uses a pseudo-random construction method for determining valid CDO codes of order J: starting from a set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$ of $N < J$ integers forming a valid CDO code, an element taken among the natural integers arranged in ascending order is appended. Should the new set $\{\alpha_1, \alpha_2, \ldots, \alpha_N, \alpha_{N+1}\}$ of $N + 1$ integers form a valid CDO code, a test is performed to determine whether this additional integer is retained or not: the test consists in comparing a pseudo-randomly generated number to an arbitrary threshold value. If, on the other hand, the new set is not a valid CDO code, the integer is discarded. The procedure is repeated until all J elements are obtained.

Various improvements have been made on the choice of the discarding threshold value, each one yielding CDO codes of shorter span. A first version used a fixed-value threshold [50], later improving on the algorithm by using a threshold that linearly increases with N. Then, [20] further increased the efficiency of the algorithm by using a nonlinear (polynomial) discarding threshold that rejects α_N integers having smaller values of N with a much higher probability than α_N elements having larger values of N. This ensures that the search is spread-out over the search space, thus increasing the chances of finding new and improved codes, while also considering the fact that the time required for finding valid CDO codes greatly increases as N grows larger.

In [5], a pseudo-random search algorithm for determining valid S-CDO codes is presented.

Instead of using a discarding threshold, it uses a set of heuristics to generate, for each element α_i, a list of integers Γ_i: these integers are chosen such that, when added to α_i to compute α_{i+1}, the likelihood of forming a valid code is maximized. Then, using a pseudo-random selection process, an integer from Γ_i is chosen and α_{i+1} is computed. If the resulting code is valid, the α_{i+1} element is kept; otherwise, a new α_{i+1} value is computed by pseudo-randomly selecting the next integer from Γ_i prior to adding its value to α_i. The process is repeated until all J elements are obtained.

In essence, these pseudo-random search algorithms explicitly skip areas of the search space under the assumption that the search space will never be fully explored due to its considerable size. Instead, they attempt to be more effective at finding new codes with improved spans by spreading-out the search efforts over pseudo-randomly chosen search-space areas.

2.3.2.1.2 A high-performance CDO code validation function The fastest CDO code validation function published prior to this work is described in [20]. It provides a large performance advantage over the reference CDO code validation function, which is based on a set of nested *for-loops* that sequentially test all three CDO code conditions.

The significant performance improvement offered by this validation function is achieved by means of two algorithmic enhancements. First, only the second CDO code condition is tested, as ensuring that it is met also ensures that the first and third CDO conditions are met [4, 50]. This reduces from three to one the number of conditions to test. Then, it generates and classifies the first-order differences into a matrix that is partitioned into regions, such that regions that do not need to be verified are identified and implicitly discarded. First-order differences are only computed once for each code validation. Subsequently, they are read from memory to generate the second-order differences. The computational time is reduced by using this classification system to restrict the computations performed to only the regions of the matrix that need to be verified. This effectively reduces the number of second-order differences that are computed and compared from $2N_D^J$ to N_D^J (see (2.6)), thus substantially improving the validation function's performance. Nevertheless, a new classification matrix has to be created for each code validation, and all resulting second-order differences have to to be compared with each other to ensure that no two equal values exist, both operations resulting in a considerable computing overhead.

Since this validation function only tests the second condition of the CDO code definition, it cannot be used for validating S-CDO codes. However, due to its speed for validating CDO codes, its performance is compared to the novel (S-)CDO code validation functions presented in Chapter 5.

2.3.2.2 Fully-Exhaustive Search Algorithm

The most basic way of performing an exhaustive search is to choose an arbitrary initial maximal span value M_{curr} for the (S-)CDO code order J that is being searched for, and then to test all the combinations of potential (S-)CDO codes that have a smaller or equal span value. The value of M_{curr} may either be chosen by means of an educated guess, or by using a known valid span value, for example the span of a valid (S-)CDO code obtained through projective geometry or by using a pseudo-random search algorithm. Potential (S-)CDO code candidates are then tested for validity by checking that all the conditions defining them are met.

The most sensitive aspect of this algorithm is the choice of the initial span value, M_{curr}, as the number of codes that have to be tested is equal to N_L (see (2.12)): if the chosen M_{curr} value is too small, no valid codes with that number of elements will be found; if the chosen value is too large, the number of codes that need to be tested quickly explodes, thus exponentially increasing the time required for a fully-exhaustive search-tree exploration.

Since all of the root-to-leaf node paths on the search-tree are tested (see Fig. 2.8), we can be certain that if the initial value M_{curr} is chosen large enough, the optimal-span (S-)CDO codes for that order J are found. However in practice, because of its brute-force approach and extreme inefficiency, this algorithm is not used.

2.3.2.3 Improved Reference Exhaustive-Search for (S-)CDO codes

The main goal of the exhaustive-search algorithm presented in [5] is to find the (S-)CDO codes with the shortest possible spans for a specific set of J connections. While the search is in progress, codes with a span shorter than or equal to the current best span are gradually obtained. Upon completion of the search (i.e. when all the potential code candidates have been evaluated), the algorithm guarantees that the optimal span (S-)CDO codes for that J have been found. In order to reduce the computational overhead, the δ value is not evaluated during the search for (S-)CDO codes. Instead, it is simply computed once the code has been found.

Since to our knowledge this is the fastest published (S-)CDO code exhaustive-search algorithm, it will be used as a reference for comparison with the algorithm presented in this thesis.

2.3.2.3.1 Reference tree traversal The *improved reference exhaustive-search* algorithm's tree-traversal [5] uses a depth-first search algorithm [72], as shown in Figures 2.8 and 2.9, to progressively *assemble* a valid (S-)CDO code. Indeed, the algorithm leverages the relationship between consecutive elements in a code (see (2.4) and Section 2.3.1): a code with $N + 1$ connections is created by using a valid (S-)CDO code with N connections and appending an α_{N+1} element, such that the newly formed code is both valid (see Section 2.1.3) and has a span with smaller or equal value than M_{curr}, the shortest known span for that order J.

The reference exhaustive-search algorithm's pseudo-code shown in Fig. 2.9 is now briefly described for a code with $J = 3$ connections (see Fig. 2.8). The value of M_{curr} is initialized at some given large value[6], since we assume that no $J = 3$ (S-)CDO codes are currently known. Naturally, any prior knowledge of a valid $J = 3$ code can accelerate the search, but this fact is not exploited to avoid biasing reported results with an unfair advantage. Starting at the root node $\alpha_1 = 0$, the first available child node $\alpha_2 = 1$ is appended. The validation routine is executed on the $\{0, 1\}$ code, and because it satisfies the double orthogonality conditions as per the validation process, the node is kept (see Fig. 2.8). The current number of connections being smaller than J, the next available node $\alpha_3 = \alpha_2 + 1 = 2$ is appended. Since $\{0, 1, 2\}$ fails the validation test, the node is therefore discarded. Nodes can be discarded either because the validation test fails or because their span is larger than M_{curr}. The process of adding, testing for validity, and discarding a node is repeated for all sibling nodes on a path until either the

[6]As with the fully-exhaustive search algorithm, the value of M_{curr} may either be chosen by means of an educated guess, or by using a known valid span value, for example the span of a valid (S-)CDO code obtained through projective geometry or by using a pseudo-random search algorithm.

```
procedure REFERENCE_TREE_TRAVERSAL( $InitShortestKnownSpan$ )
    $ShortestKnownSpan \leftarrow InitShortestKnownSpan$
    $CurrentNode \leftarrow RootNode$
    → Move down one level in the tree          ▷ ($CurrentNode \leftarrow FirstChildNode$)

    while $True$ do
        if $CurrentNode.getValue() > ShortestKnownSpan$ then
            → Discard node
            → Move up one level in the tree          ▷ ($CurrentNode \leftarrow ParentNode$)
            if $CurrentNode == RootNode$ then
                → Terminate search
            else
                → Discard node
                → Move to next sibling node          ▷ ($CurrentNode \leftarrow NextSiblingNode$)
            end if
        end if

        if $Code.isValid()$ then
            if $CurrentNode.isLeafNode()$ then
                if $Code.getSpan() \leq ShortestKnownSpan$ then
                    $ShortestKnownSpan \leftarrow Code.getSpan()$
                    → Add current code to list of codes found
                    → Discard node
                    → Move to next sibling node          ▷ ($CurrentNode \leftarrow NextSiblingNode$)
                end if
            else
                → Move down one level in the tree          ▷ ($CurrentNode \leftarrow FirstChildNode$)
            end if
        else
            → Discard node
            → Move to next sibling node          ▷ ($CurrentNode \leftarrow NextSiblingNode$)
        end if
    end while
end procedure
```

Figure 2.9 Pseudo-code for the reference tree-traversal algorithm.

added node forms a valid (S-)CDO code, in which case the node is kept and its children are evaluated, or no more such siblings exist, in which case the next parent is evaluated. If the current valid code has J connections and its span value is smaller than the best known span, M_{curr} is updated, leading to a very substantial tree pruning for all paths evaluated from that point on. A path through the next parent is evaluated until no more paths exist, at which point we know there is no (S-)CDO code for that J with a span shorter than M_{curr}. The list of optimal (S-)CDO codes will be the codes with a span equal to M_{curr}.

Although it is not defined as such in [5], this search algorithm is *implicitly*-exhaustive because it is among the “branch and bound” class of algorithms [73]. It does not need to test

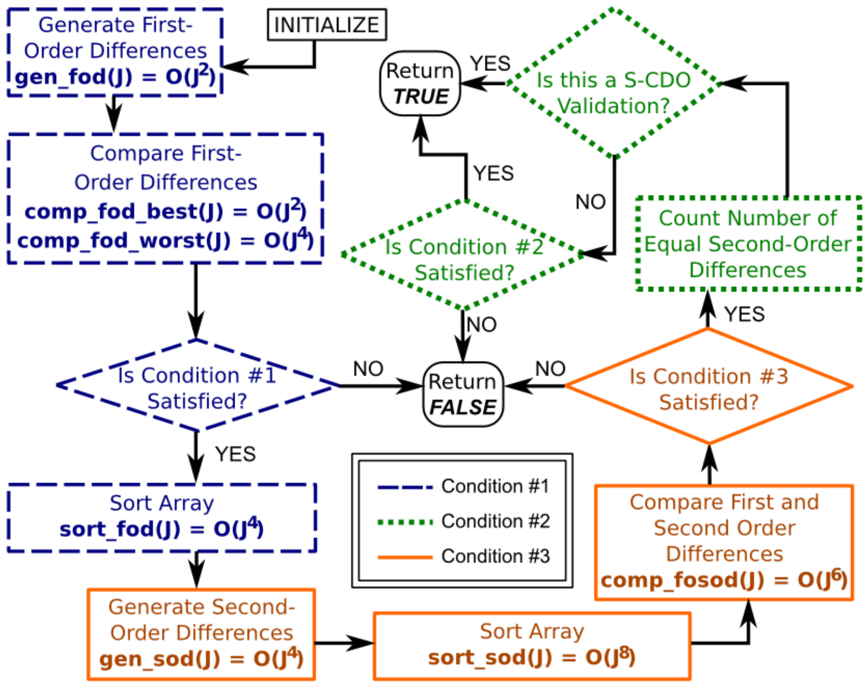

INITIALIZE
Generate First-Order Differences gen_fod(J) = O(J2)
Compare First-Order Differences comp_fod_best(J) = O(J2) comp_fod_worst(J) = O(J4)
Is Condition #1 Satisfied?
NO
YES
Sort Array sort_fod(J) = O(J4)
Generate Second-Order Differences gen_sod(J) = O(J4)
Sort Array sort_sod(J) = O(J8)
Compare First and Second Order Differences comp_fosod(J) = O(J6)
Is Condition #3 Satisfied?
NO
YES
Count Number of Equal Second-Order Differences
Is this a S-CDO Validation?
YES
NO
Is Condition #2 Satisfied?
YES
NO
Return TRUE
Return FALSE
Condition #1
Condition #2
Condition #3

Figure 2.10 Simplified diagram of the reference validation function's algorithm.

all the nodes and paths, but still performs an exhaustive search while reducing the search complexity by several orders of magnitude: if a node addition fails the validation test, the validation routine is not applied to any children nodes on the sub-branches starting at that node since their inclusion cannot lead to a valid (S-)CDO code (see (2.4)), and thus the node and its children can safely be discarded. Indeed, invalidating nodes close to the root node first is very advantageous: since the number of differences that have to be computed is a function of J (see (2.7)), the time required to validate a (S-)CDO code increases with J. Furthermore, updating M_{curr} when a shorter span is found is also highly beneficial, as the number of paths sharply increases as J and M_{curr} increase[7]: the effective result is a substantial reduction in the number of computations performed. Finally, given the fact that implicitly-exhaustive search algorithms still perform an exhaustive search, we can be certain that when the process completes, the codes that are obtained are *proven to be optimal*.

[7]Recall that the total size of the search-space is determined by (2.12). Therefore, the exhaustive-search for optimal-span (S-)CDO codes having order $J+1$ is exponentially more complex than it is for codes having order J.

2.3.2.3.2 Reference Algorithm - Validation Function A detailed analysis of the reference validation algorithm in [5] is beyond the scope of this document. Nevertheless, in order to better understand and position our work, it is briefly described below.

The validation is performed in three consecutive steps, each corresponding to one of the (S-)CDO code orthogonality conditions (see Fig. 2.10). As first and second order differences are generated (see Fig. 2.11), they are stored into their respective arrays: fo_array and so_array. These arrays are initialized and cleared only once, at the beginning of the search, since all relevant data is overwritten with each validation: relevant data resides at indices having value smaller than fo_count and so_count respectively. For every new code, the reference algorithm computes two new sets of differences: *positive* first-order differences, S^+_{ref}, and *positive and negative* second-order differences, D_{ref}. However, only *positive* second-order difference values are stored into so_array.

In order to validate the first condition, the positive first-order differences are generated and stored into fo_array (see Fig. 2.10). Each first-order difference in fo_array is then compared to all other differences, to ensure that no two equal elements exist. If two elements were found to be equal to one another, a flag is raised, but the test proceeds until all comparisons for that element are completed. Then, the flag is checked: if raised, the condition's test fails and *False* is returned; otherwise, the next first-order difference is compared to the rest, until all elements have been compared to each other. The number of comparisons required is a function of N_S (see (2.5)), and thus the complexity of the comparison as a function of J is quadratic in its best case (the first element having an equal) or polynomial in its worst case (the last two elements being equal). If the condition is verified, the array is sorted using a variant of the *Bubble Sort* algorithm[8], which also has a quadratic complexity as a function of N_S and thus a polynomial complexity as a function of J.

Next, the third condition is tested (see Fig. 2.10). The reference second-order difference generation requires four *for-loops* and five *if-statement* tests. Even though all $2N_D$ second-order differences are computed, only half are stored into so_array, i.e. only D^+_{ref}, the set of *positive* second-order differences. In order to reduce the number of arithmetic operations,

[8]Bubble Sort is a simple but inefficient sorting algorithm that has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted [74]. Therefore its use is not recommended for practical applications.

one of the second-order difference terms is kept in memory while the other term is updated, thus avoiding the computation of *both* terms at each iteration of the inner loop. The array is sorted using a variant of the *Bubble Sort* algorithm, and then each second-order difference is compared to each first-order difference to ensure that no element in D^+_{ref} has an equal in S^+_{ref}. If two equal values are found, a flag is raised, but the validation function proceeds until all elements in D^+_{ref} have been compared to all elements in S^+_{ref}, at which point *False* will be returned.

Finally the second condition is tested (see Fig. 2.10). Having sorted *so_array*, it is scanned to count the number of differences having an equal value in the array (i.e. N^e_D, see (2.3)). The condition is verified differently depending on whether a S-CDO or a CDO code is to be validated: for S-CDO codes, the validation function will always return *True* and the number of equal second-order difference values will be made available so that δ, the simplification coefficient, can be computed; for CDO codes, the validation function returns *True* only if the number of equal second-order differences is zero, otherwise *False* is returned.

There are several factors greatly limiting the performance of the reference exhaustive-search (S-)CDO code validation function. First, sorting the second-order differences is done with an $O(J^8)$ time complexity, and comparing second-order differences with first-order differences is done with an $O(J^6)$ time complexity. Then, since the function focuses on *validating* a code, a significant computational overhead is incurred while processing *invalid* codes (i.e. clearly most codes). For example, the third condition test will *fail* only after all the comparisons have been made, even if two equal differences are found early-on in the comparison process. Furthermore, although the reference validation function generates both *positive* and *negative* second-order differences (for a total of $2N_D$ differences), only the *positive* second-order differences (i.e. N_D differences) are stored and used for the validation process, thus implying unnecessary computations. Finally, as shown in Fig. 2.11, the reference algorithm for generating first and second order differences consist of a set of nested *for-loops* and includes several *if-statements*. These hinder the execution speed of the difference generation: on the one hand, the nested *for-loops* are difficult for compilers to fully unroll, thus limiting their ability to reschedule instructions to reduce memory access latencies and eliminate the overhead caused by instructions controlling the loop; on the other hand, each branch test may result in a branch mispredictions on modern microprocessors, further limiting the maximum performance that would otherwise be achievable.

The limitations of the *reference exhaustive-search algorithm* are all addressed and circumvented in the novel *parallel and efficient implicitly-exhaustive* search algorithm presented in Chapters 3, 4, and 5.

```
function REF_FO_DIFF_GEN($code$)
    $fo\_count \leftarrow 0$
    for ($i = 0$; $i < code.length() - 1$; $i{+}{+}$) do
        for ($j = i + 1$; $j < code.length()$; $j{+}{+}$) do
            $fo\_array[fo\_count] = code[j] - code[i]$
            $fo\_count \leftarrow fo\_count + 1$
        end for
    end for
    return ($fo\_array$, $fo\_count$)
end function

function REF_SO_DIFF_GEN($code$)
    $so\_count \leftarrow 0$
    $t1 \leftarrow 0$
    $t2 \leftarrow 0$
    for ($i = 0$; $i < code.length()$; $i{+}{+}$) do
        for ($j = 0$; $j < code.length()$; $j{+}{+}$) do
            if $i \neq j$ then
                $t1 \leftarrow code[i] - code[j]$
                for ($k = 0$; $k \leq j$; $k{+}{+}$) do
                    if $k \neq i$ then
                        for ($n = 0$; $n \leq i$; $n{+}{+}$) do
                            if ($n \neq j$) then
                                if ($n \neq k$) then
                                    $t2 \leftarrow code[k] - code[n]$
                                    if $t1 \geq t2$ then
                                        $so\_array[so\_count] \leftarrow t1 - t2$
                                        $so\_count \leftarrow so\_count + 1$
                                    end if
                                end if
                            end if
                        end for
                    end if
                end for
            end if
        end for
    end for
    return ($so\_array$,$so\_count$)
end function
```

Figure 2.11 Pseudo-code of the reference exhaustive-search algorithm's generation of first and second order differences.

CHAPTER 3

IMPROVING THE TREE-TRAVERSAL OF THE IMPLICITLY-EXHAUSTIVE SEARCH ALGORITHM

3.1 Overview

In order to speed up the search for new optimal-span (S-)CDO codes, a parallel version of the algorithm described in Section 2.3.2.3 was developed. In this algorithm [19], the computation time is reduced by means of a very basic simultaneous exploration of independent regions in the search-tree. Indeed, this preliminary brute-force parallel approach showed that the problem lends itself well to parallel computing, as it exhibited a *linear* and at times *super-linear* speedup [75] with respect to the number of computing threads used (see Fig. 3.1). Nevertheless, given the speed of the reference algorithm, it quickly became clear that linear speed improvements would not be able to address the very rapidly increasing size[1] of the search space, and that in order to obtain new optimal-span codes, a more capable algorithm would have to be devised.

To that end, we developed the algorithm described in the first article of this thesis [15], presented and included verbatim in Section 3.2: it uses a more effective implicitly-exhaustive searching technique for efficiently reducing the size of the search space without compromising the exhaustive nature of the search. Indeed, when used with an initial span value of $M_{curr} = 100$ during the search for optimal-span CDO codes of order $J = 6$, compared to the fully-exhaustive search algorithm, more than a 150-fold reduction in the number of leaves explored was achieved.

The combination of tree-pruning techniques that were applied allowed the algorithm to prove the optimality of the rate $R = \frac{1}{2}$ systematic optimal-span $J \in \{6, 7, 8\}$ CDO codes and $J = 9$ S-CDO codes that were found. Furthermore, the algorithm was also able to yield $J \in \{10, 11\}$ CDO codes and $J \in \{14, 15\}$ S-CDO codes with shorter spans than

[1]We recall that the size of the search space associated with the search for optimal-span (S-)CDO codes of order J is defined by $N_L = \begin{pmatrix} M_{curr} \\ J-1 \end{pmatrix}$, where M_{curr} is the shortest span currently known.

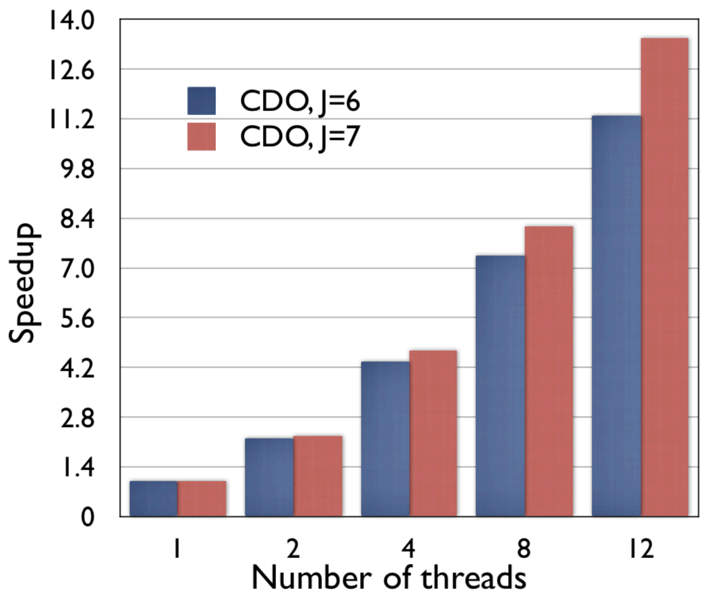


Figure 3.1 Speedup observed with parallel version [19] of the algorithm in Section 2.3.2.3

previously published, resulting in a span reduction, and thus a decoding latency reduction, of up to 26%. In order to reduce the computation time required for simulating the error-correction performance of these codes, several instances of the (S-)CDO code simulator were run concurrently and on different computers: due to inefficiencies in our (S-)CDO code simulation software, each error-performance curve required several weeks of computation time to deliver the $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) bit error rates obtained.

We now present verbatim the article published in [15]: the more efficient search algorithm yielding novel (S-)CDO codes is described, and the codes obtained and their error-correction performance are provided.

3.2 Article #1: Efficient Search Algorithm for Determining Optimal $R = 1/2$ Systematic Convolutional Self-Doubly Orthogonal Codes

G. Kowarzyk, N. Bélanger, D. Haccoun, Y. Savaria

École Polytechnique de Montréal

{*gilbert.kowarzyk, normand.belanger, david.haccoun, yvon.savaria*}*@polymtl.ca*

Publication source: IEEE Transactions on Communications, Transactions Letters, vol. 60, no. 1, January 2012, pp. 3-8.

Abstract

A novel implicitly-exhaustive search algorithm for finding, in systematic form, rate $R = \frac{1}{2}$ optimal-span Convolutional Self-Doubly Orthogonal (CDO) codes and Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes is presented. In order to build high-performance low-latency codecs with these codes, it is important to minimize their constraint length (or "span") for a given J number of generator connections. The proposed algorithm is exhaustive in nature and its improvements over the best previously published searching techniques allowed it to yield new optimal-span CDO/S-CDO codes (having order $J \in \{6,7,8\}$ and $J \in \{9\}$ respectively), as well as a span reduction for codes with a higher J value ($J \in \{10,11\}$ and $J \in \{14,15\}$ for CDO and S-CDO respectively).

Index Terms: *Convolutional codes, self-doubly orthogonal codes, systematic codes, threshold decoding.*

3.2.1 Introduction

The novel iterative error-control coding scheme presented in [4, 6, 7] differs from the classical Turbo code procedure invented in 1993 [8, 9], as it does not use any interleaver, neither at the encoding nor at the decoding process. The iterative threshold decoding algorithm it uses employs a new class of convolutional codes in systematic form that must satisfy double orthogonality properties, beyond those of the well-known orthogonal codes used in the conventional non-iterative threshold decoding [10]. Throughout this paper, the widely used terminology *systematic codes* is used to represent *convolutional codes in systematic form*, that is, codes whose encoders are systematic. Since we only consider rate $R = \frac{1}{2}$ codes, only one generator vector is provided. The additional so-called double orthogonality properties required from the codes ensure a quasi-independence of the observables over the first two decoding iterations, thereby allowing the use of an iterative decoding procedure and hence a good error performance while allowing attractive trade-offs between complexity, latency, and error performance [6].

As the error-correcting capability of the new codes depends essentially on the dimension J of the vector generator of the $R = \frac{1}{2}$ code [20], and because the code constraint length (or span of the code) has a direct impact on the latency of the system, it is of great importance to search for rate $R = \frac{1}{2}$ systematic Convolutional Self-Doubly Orthogonal (CDO) codes (and their variants) having the shortest possible span for any given J number of connections. Since no systematic deterministic method for solving this problem is currently known, the code searching is usually conducted using heuristic search algorithms [20, 28]. Although finding a CDO code is relatively easy, determining the shortest span codes for a given J has eluded analysis and is still an open problem. In fact, the search for optimal CDO codes (and their variants) is far more computationally challenging than the problem of finding "optimal" simply orthogonal codes, a.k.a. the *Golomb ruler* problem, which has an *NP-hard* complexity [13, 46]. Indeed, CDO codes may be viewed as second-order Golomb rulers.

Pseudo-random and exhaustive search algorithms have been developed to obtain good, i.e. short span, CDO codes [5, 20, 28]. However, using these algorithms to find even shorter span or eventually optimal span (i.e. *shortest* span) codes requires a computational time that becomes rapidly excessive, especially as the number J of generator connections increases beyond $J = 5$.

This paper presents a novel and efficient implicitly-exhaustive search algorithm that greatly reduces the computational time required for finding optimal-span CDO codes and their variants. To increase the speed and efficiency of the search process, the algorithm exploits significant algorithmic improvements such as an enhanced dynamic search-space reduction technique and a stricter set of constraints to identify and concentrate the search on only potentially valid codes.

This faster technique allowed finding new optimal-span codes. Moreover, short of obtaining optimal codes for some higher J values, we have also been able to obtain, within an acceptable amount of computation time, new codes with significantly shorter spans than the ones previously published.

The paper is organized as follows: in Section 3.2.2, CDO and S-CDO codes (a CDO code variant) are defined to establish the notation used in this paper. A novel and more efficient implicitly-exhaustive search algorithm is described in Section 3.2.3. In Section 3.2.4, new optimal-span codes, as well as novel codes with a shorter span than previously obtained [20, 21, 28] are presented.

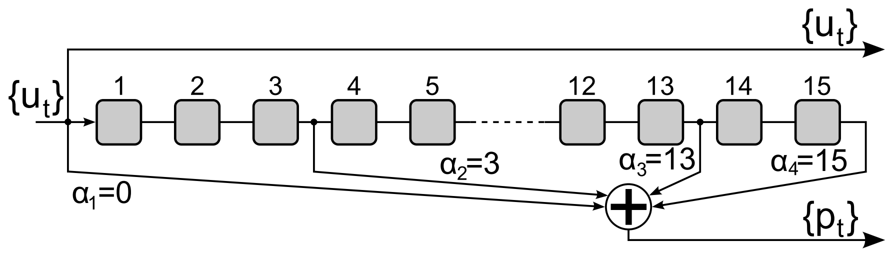


Figure 3.2 Example of a CDO code encoder: $R = \frac{1}{2}$, $J = 4$, $M = 15$, $\Omega = \{0, 3, 13, 15\}$

3.2.2 Definitions

In this section, we provide the necessary definitions on the vector generator of the systematic CDO and S-CDO codes of coding rate $R = \frac{1}{2}$ [4, 49], which, as previously stated, may be viewed as second-order Golomb rulers [43].

3.2.2.1 Convolutional Self-Doubly Orthogonal (CDO) codes

A systematic Convolutional Self-Doubly Orthogonal (CDO) code of coding rate $R = \frac{1}{2}$ and order J is defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers $(\alpha_1 < \alpha_2 < ... < \alpha_J)$ such that the following conditions are satisfied [6, 20, 21, 49]:

1. The elements in S, the set of first-order differences between these integers, are all distinct:
$$S = \{s_{k,l} = (\alpha_k - \alpha_l) \ : \ k \neq l\}$$
2. The elements in D, the set of second-order differences (the differences between the differences), are all distinct from one another, with the exception of the unavoidable differences caused by the permutations of indices (l, m) or (k, n):
$$D = \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \ : \ k \neq l, m \neq n, k \neq m, l \neq n\}$$
3. The elements in sets S and D are distinct from one another ($D \cap S = \emptyset$).

The α_i elements ($i \in [1; J]$) represent the connections between the encoder shift register and the modulo-2 adder, i.e. the generator connections. By convention α_1, the first integer in our set, is always equal to zero ($\alpha_1 = 0$). The span M of a CDO code is equal to α_J, the largest integer in Ω. It corresponds to the length of the encoder shift register (see Fig. 3.2), that is, the code memory length [20]. The number J of elements in Ω is equal to the number of generator connections of the code and is called the order of the CDO code. An optimal CDO code of a given order J is defined as a CDO code whose span M_{opt} is the *smallest* span that exists for that order; an optimal CDO may not be unique and hence there may be more than one optimal CDO code for any given order J.

Since the validity of a code as a CDO code depends only on the relationship between the J elements composing it, any subset of L consecutive elements from the set defining a CDO code also forms a valid CDO code, albeit one of smaller order L, $L < J$. For example:

$$\text{CDO}_{J=5} = \{0, 1, 24, 37, 53\} \qquad \text{CDO}_{J=4} = \{0, 1, 24, 37\} \qquad \text{CDO}_{J=3} = \{0, 1, 24\} \tag{3.1}$$

are all valid, although not optimal, CDO codes. This property will be leveraged to speed-up the proposed algorithm.

The number of *positive* first-order differences (N_S) and second-order differences (N_D) that exist for a code are a function of J, given by [20]:

$$N_S^J = \frac{J(J-1)}{2} \tag{3.2}$$

$$N_D^J = \frac{J(J^3 - 2J^2 + 3J - 2)}{8} \tag{3.3}$$

Directly computing the exact span of optimal codes, whether simply or doubly orthogonal, is still an unsolved problem [13, 46]. However, a loose lower bound for the span of a CDO code has been developed in [20, 50] and can be expressed as a function of J, the order of the code, using (3.2) and (3.3) as follows:

$$\alpha_J^* = \left\lceil \frac{N_S^J + N_D^J}{2} \right\rceil \tag{3.4}$$

Furthermore, any CDO code has always a symmetrical (mirror) equivalent composed of integers with the same differences but in the reverse order, a property shared with the so-called Golomb ruler problem they are related to [20].

3.2.2.2 Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes

Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes are obtained by relaxing the second CDO condition, yielding codes with shorter spans than regular CDO codes. The latency of the decoding process is in direct proportion to the span of the code and the number of iterations used. Thus, using S-CDO instead of CDO codes reduces the decoding latency at the cost of only a very small degradation of the error-correction performance [5, 21, 51].

A systematic S-CDO code of coding rate $R = \frac{1}{2}$ and order J is thus defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers such that it satisfies the *first* and *third* CDO conditions, and a *modified version of the second* condition, as follows [5, 51]:

2b) The set D of second-order differences between the integers in Ω, defined as:

$$D = \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \; : \; k \neq l, m \neq n, k \neq m, l \neq n\}$$

is composed of $2N_D$ second-order differences (of which $2N^e_D$ have an equal value in the set D), computed by excluding the unavoidable second-order differences caused by the permutations of indices (l, m) or (k, n). We define δ, the simplification coefficient, as $\delta = \frac{N^e_D}{N_D}$, where $N^e_D < N_D$ and $0 \leq \delta \leq 1 - \frac{N_S}{N_D}$ [5].

A CDO code may be viewed as a S-CDO code for which $\delta = 0$. Thus, they share most of their properties. Furthermore, a loose lower bound on the span of a S-CDO code has been derived as a function of J and δ, and is expressed, using (3.2) and (3.3), as [5, 21]:

$$\alpha^*_J = \left\lceil \frac{N_S + (1-\delta) \cdot N_D}{2} \right\rceil \tag{3.5}$$

An optimal S-CDO code of order J and simplification coefficient δ is thus a S-CDO code having the smallest span, M_{opt}, which exists for that order and δ. Again, there may be more than one optimal S-CDO code for a given order J and simplification coefficient δ.

In this paper, the notation "(S-)CDO" will be used when referring to both CDO and S-CDO codes. We now present an efficient search procedure for obtaining $R = \frac{1}{2}$ systematic (S-)CDO codes with a span shorter than codes previously published.

3.2.3 Novel Efficient Implicitly-Exhaustive Search Algorithm

3.2.3.1 Overview of previous search algorithms

The first technique for finding CDO codes used a projective geometry approach to yield valid codes [50, 71]. However, this approach yielded codes with excessively large spans, and thus new code-searching methods were devised [20, 50].

Recent code-searching algorithms can be divided into two categories, exhaustive and pseudo-random. *Exhaustive* search algorithms guarantee, if a sufficiently large initial M_{curr} is used, that the optimal span for a given J number of connections is found by testing all the potentially valid codes in the search space. However, as J becomes larger, the very rapidly increasing computational effort required to obtain optimal (S-)CDO codes led to dropping this type of algorithm in favor of more practical pseudo-random search algorithms [5, 50]. *Pseudo-random* search algorithms are based on the use of a pseudo-random rejection criterion that can be modified in order to shorten the spans of the codes obtained [20]. Note that this type of algorithm cannot guarantee that minimal-span codes have been found. With the use of a span reduction method based on modulo operations [50] and a carefully chosen lower bound, the technique has provided codes with the shortest known spans, albeit possibly not optimal codes [5, 20].

We have focused on developing methods for further improving the computational performance of the exhaustive search algorithms in order to find new optimal codes or at least codes with a shorter span than the ones that have been previously reported for the same order. The (S-)CDO code exhaustive-search algorithm described in this paper uses a tree-like structure to perform the search (see Fig. 3.3 for $J = 3$). The tree is composed of nodes which must have a value larger than their parent node and their sibling nodes to the left. Sibling nodes are nodes with the same parent. The tree depth represents the total number of connections J, while the values of the nodes on a path from the root to a leaf represent the elements of $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$, the set of positive integers defining the code. All nodes at depth "$J - 1$" are leaf-nodes. A *valid path* in this search-tree starts at the root node and ends at a leaf-node that has a value not larger than M_{curr}, the currently smallest known span value.

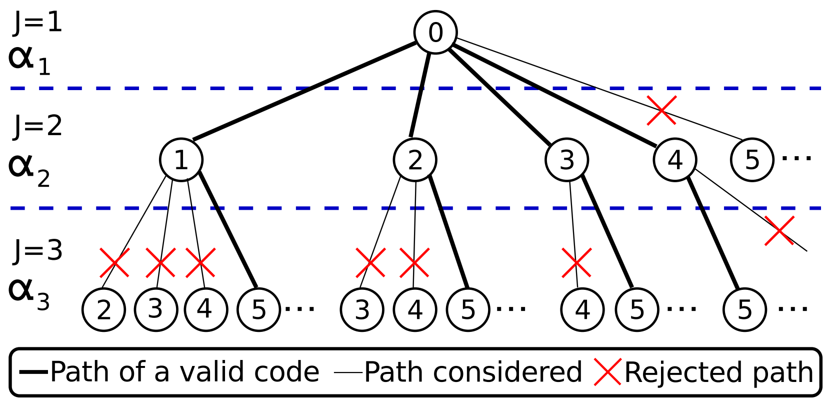


Figure 3.3 (S-)CDO search-tree - searching for a CDO with $J = 3$

3.2.3.2 An "implicitly-exhaustive" search algorithm

The (S-)CDO code search-tree size depends on the current span, M_{curr}, and on J. Since there is only one path leading from the root node to a leaf-node, the number of leaf-nodes in the search-tree, N_L, also represents the total number of possible paths (or (S-)CDO code candidates). Since the set Ω defining a code always starts with zero (the root node), the number of possible combinations of "$J-1$" nodes with integer values ranging from 1 to M_{curr} may be expressed as: $N_L = \begin{pmatrix} M_{curr} \\ J-1 \end{pmatrix} = \frac{(M_{curr})!}{(J-1)!(M_{curr}-J+1)!}$. It can be seen that the number of leaves quickly explodes as M_{curr} and J increase. Clearly, to improve search efficiency, tree-pruning techniques must be used as much as possible.

The novel algorithm described in this section proposes an exhaustive searching technique that is faster at finding rate $R = \frac{1}{2}$ systematic (S-)CDO codes with a shorter span than the best pseudo-random and exhaustive search algorithms presently known, yielding optimal-span codes for higher J order values than previously known ($J \leq 8$ for CDO, $J \leq 9$ for S-CDO). For codes with a large order ($J \geq 9$), where an exhaustive-search can be prohibitive, this new search-method also allows to gradually obtain codes with significantly shorter spans than previously known, thus allowing to find better codes within the same computation time.

The novel algorithm presented in this section is *implicitly*-exhaustive and among the "branch and bound" class of algorithms [73]: it does not need to test all the nodes and paths, but still performs an exhaustive search while reducing the search complexity by several orders of magnitude. This class of algorithms has helped to improve the processing time required to obtain good practical solutions of important *NP-complete* problems by several orders of magnitude in the field of circuit testing. For instance, the *PODEM* algorithm allowed a breakthrough in the size of the circuits that could be tested within a reasonable amount of computation time [76]. Since an implicitly-exhaustive algorithm still performs an exhaustive search, the codes that are obtained are *proven to be optimal.*

3.2.3.3 Dynamic reduction of the number of branches explored - an implicitly-exhaustive search

The novel algorithm presented in this section provides several search-tree pruning enhancements over the *reference exhaustive-search* algorithm's tree-traversal [5]. It reduces the computational time required to search for optimal (S-)CDO codes by further reducing the number of branches that are explored: the validation routine does not need to be applied to children nodes of nodes that have been discarded. The substantial computational savings are obtained by using the three search-tree pruning techniques described below.

A node whose value is larger than M_{curr} minus the search-tree depth remaining from that node to a leaf-node, must have descendants such that their leaf-nodes have a value larger than M_{curr}, the current best known span. Thus, these nodes can be safely discarded since they cannot lead to codes with a shorter span than the shortest known span for that order J. This is proven in the following theorem:

Theorem 3.1. *Let V_{max}^d be defined as: $V_{max}^d = M_{curr} - ((J-1) - d)$. Then, any node at depth $d > 1$ with a value greater than V_{max}^d will result in leaf-nodes with a value greater than M_{curr}, and hence these nodes can be discarded because they would result in codes with a larger span than the best known span.*

Proof. V_{max}^d assumes that nodes at depth $2 \leq d \leq J-1$ have a node value increment of one (i.e. the smallest possible increment) between each parent and child nodes and that the leaf-node at depth $d = J-1$ has a node value M_{curr}: only one such code exists but it does not meet the criteria for being a valid (S-)CDO code. Any code with node values at depth $d > 1$ larger than V_{max}^d must thus have a larger span than M_{curr}, and therefore can be discarded since they cannot be optimal-span codes. *QED* □

A second search-tree pruning technique consists in making use of the symmetry property of the codes (see Section 3.2.2) in order to discard some codes whose symmetrical code has already been encountered during the tree exploration, as stated in Theorem 3.2 below.

Theorem 3.2. *Let $V_{max}^{\alpha_2}$ be defined as: $V_{max}^{\alpha_2} = \left\lceil \frac{M_{curr}}{2} \right\rceil - 1$. Then, any code with an α_2 node having a value larger than $V_{max}^{\alpha_2}$ would have a* symmetrical *equivalent within the codes in the search-tree having $\alpha_2 \leq V_{max}^{\alpha_2}$.*

Proof. Assuming that the tree is traversed in the same order as in the reference exhaustive search algorithm, $V_{max}^{\alpha_2}$ can be seen as a symmetry center for all the possible connection patterns to an encoder shift register with a span M_{curr} (see Fig. 3.2). Thus, all search-tree branches with an α_2 node value larger than $V_{max}^{\alpha_2}$ may be safely skipped because if they were to lead to an optimal-span code, that code's symmetrical equivalent would have already been encountered earlier in the tree exploration. *QED* □

Theorem 3.3 below states that it is possible to reduce the search space by attempting to choose a higher starting value than the *current node value plus one* when generating the list of children nodes to be evaluated. Improvements on this starting value may be obtained through the use of the lower bound as per (3.4) (for CDO codes only), or, if such a value is known, by using the optimal span for a code of order equal to the depth of the child node plus one.

Theorem 3.3. *Let V_{min}^{d} be a lower bound for node values having $d < J - 1$. Then, V_{min}^{d} can be defined as the largest of the following three values:* the optimal-span corresponding to the node's depth, M_{opt}^{d}, if it is known; the lower bound calculated as per (3.4) (for CDO codes only); the node value of the parent node plus one.

Proof. By definition, the smallest value that a node at depth $d = N$ can have while being part of a valid (S-)CDO code is equal to M_{opt}^{d}, the optimal-span for a (S-)CDO code of order $J = N + 1$. Also, for CDO codes, the lower bound α_J^* (see (3.4)) has already been shown to be a loose lower bound [20, 50]. Finally, during the search, the value of the parent node plus one is obviously a lower bound as well. Thus, the largest of these three lower bounds is the tightest lower bound among them. *QED* □

The new codes obtained with this improved algorithm are presented in the next section.

Table 3.1 Summary of new rate $R = \frac{1}{2}$ systematic (S-)CDO codes obtained with the novel technique

J	Code Type	Optimal-span?	New (S-)CDO codes	Lower Bound	Prev. best span (span reduction)	δ
6	CDO	YES	{0, 1, 17, 70, 95, 100}*	68 †	100 (0.00%)	0
7	CDO	YES	{0, 4, 34, 81, 195, 206, 211}	126 †	222 (4.95%)	0
8	CDO	YES	{0, 3, 30, 98, 278, 394, 416, 423}	217 †	459 (7.84%)	0
8	CDO	YES	{0, 5, 53, 74, 300, 346, 414, 423}	217 †	459 (7.84%)	0
9	S-CDO	YES	{0, 15, 20, 46, 125, 132, 190, 207, 208}	183 ‡	208 (0.00%)	0.5075
9	S-CDO	YES	{0, 1, 17, 26, 127, 138, 185, 204, 208}*	188 ‡	208 (0.00%)	0.4895
10	CDO	?	{0, 1, 5, 33, 543, 913, 1216, 1354, 1398, 1477}	540 †	1698 (13.02%)	0
11	CDO	?	{0, 1, 5, 21, 72, 1388, 1569, 1809, 2109, 2423, 2559}	798 †	3467 (26.19%)	0
14	S-CDO	?	{0, 1, 4, 13, 32, 71, 156, 353, 827, 927, 1034, 1099, 1357, 1475}	1125 ‡	1967 (25.01%)	0.4845
15	S-CDO	?	{0, 1, 4, 13, 32, 71, 124, 218, 642, 1025, 1178, 1349, 1652, 1739, 2001}	1469 ‡	2653 (24.58%)	0.4911

* code first presented in [5] but was not known to be optimal

† lower bound calculated as per (3.4)

‡ lower bound calculated as per (3.5)

3.2.4 Results

In this section, new optimal span rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes are provided, as well as codes with a shorter span than previously known. Their spans are compared to known theoretical lower-bounds [5, 20, 50], and the error-correction performance for some of these codes is presented.

3.2.4.1 New rate $R = \frac{1}{2}$ systematic (S-)CDO codes

The novel algorithm allowed us to determine that the rate $R = \frac{1}{2}$ systematic CDO code for $J = 6$ published in [5, 21] is in fact optimal. Although the code was known, its optimality was not. In addition, the proposed algorithm allowed finding the rate $R = \frac{1}{2}$ systematic optimal-span CDO codes for $J = 7$ and $J = 8$ (see Table 3.1), which have a span of 211 and 423 respectively. Table 3.1 also includes the lower bound as per (3.4), as well as two rate $R = \frac{1}{2}$ systematic CDO codes with a shorter span than previously published codes in [5, 20, 21] for $J \in \{10, 11\}$. These new shorter-span codes have a span of 1477 and 2559 respectively.

Furthermore, Table 3.1 presents the *shortest span* rate $R = \frac{1}{2}$ systematic optimal-span S-CDO codes for $J = 9$ (i.e. irrespective of δ), which have a span of 208, including the first published rate $R = \frac{1}{2}$ systematic S-CDO code with a simplification coefficient δ that is greater than $\frac{1}{2}$. These new optimal-span codes have a span that is between 11% and 14% greater than the calculated lower-bound for the given values of J and δ. Finally, Table 3.1 presents rate $R = \frac{1}{2}$ systematic S-CDO codes for $J = 14$ and $J = 15$ with a span about 25% shorter than the shortest ones published in [5] and [21] for those orders, with spans of 1475 and 2001 (previous best spans of 1967 and 2653 respectively). They have a span that is between 31% and 36% greater than the calculated lower-bound for that J and δ.

3.2.4.2 Error correction performance for the $R = \frac{1}{2}$ systematic codes

The error correcting performance for the novel codes presented in this paper is shown in Figs. 3.4, 3.5, and 3.6. The decoding was iterated until no significant performance improvement was observed. Fig. 3.4 shows how the error performance improves as the order of the code is increased from $J = 6$ to $J = 14$ for $\frac{E_b}{N_0} \geq 3$. The codes having order $J = 10$ (CDO) and $J = 14$ (S-CDO) have a respective span that is 13.02% and 25.02% shorter than previously known spans [20, 21].

Fig. 3.5 comprises two sets of curves. The first set is composed of three $J = 8$ CDO codes, which can be seen to have a very similar performance, despite the novel codes having a span that is 7.84% shorter than the code presented in [20]. The second set is composed of two $J = 11$ CDO codes, also with similar error correction performance, despite the novel code having a span that is 26.68% shorter than the one presented in [20]. As expected, the codes with a larger J value have a better error performance.

In Fig. 3.6, two sets of curves can be observed. The first set is composed of the two novel optimal-span $J = 9$ S-CDO codes, and the novel optimal-span $J = 7$ CDO code. The three

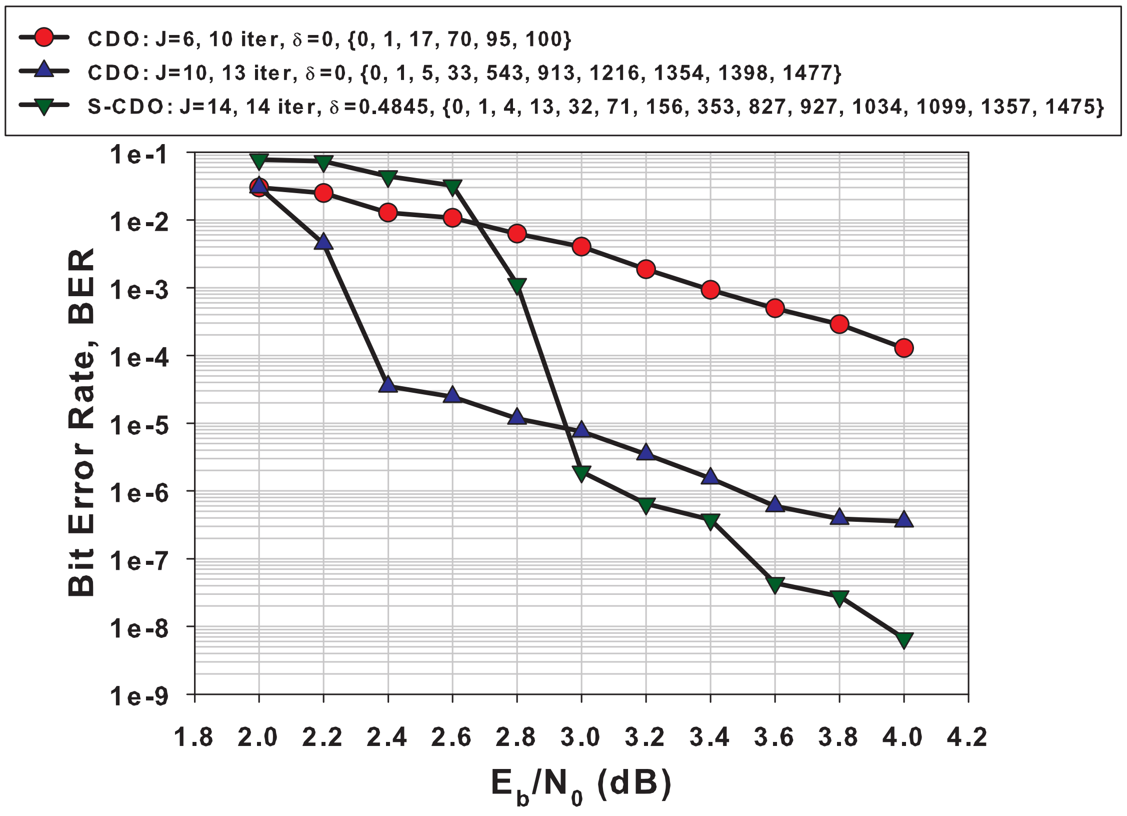


Figure 3.4 Rate $\frac{1}{2}$ systematic CDO and S-CDO code error correction performance for $J \in \{6, 10\}$, $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) and $J = 14$, $\frac{E_b}{N_0} \in [2.0; 3.6]$ (dB), after 10, 13 and 14 iterations respectively.

codes have similar spans of 208 and 211 respectively. As with the $J = 10$ (CDO) and $J = 14$ (S-CDO) codes in Fig. 3.4, it can be seen that for a similar given span, it is possible to use a S-CDO code with a higher J value than with CDO codes, and that the former offers a better error correction since their performance more heavily depends on the order J than on the span of the code. The second set is composed of the two $J = 15$ S-CDO codes. It can be seen that despite the novel code having a higher simplification coefficient δ and a span that is shorter by 24.58% than the code in [21], their error correction performance is very similar. For both, the $J = 9$ and the $J = 15$ S-CDO codes, it can be observed that their error correction performance does not change very much for small differences in the value of δ.

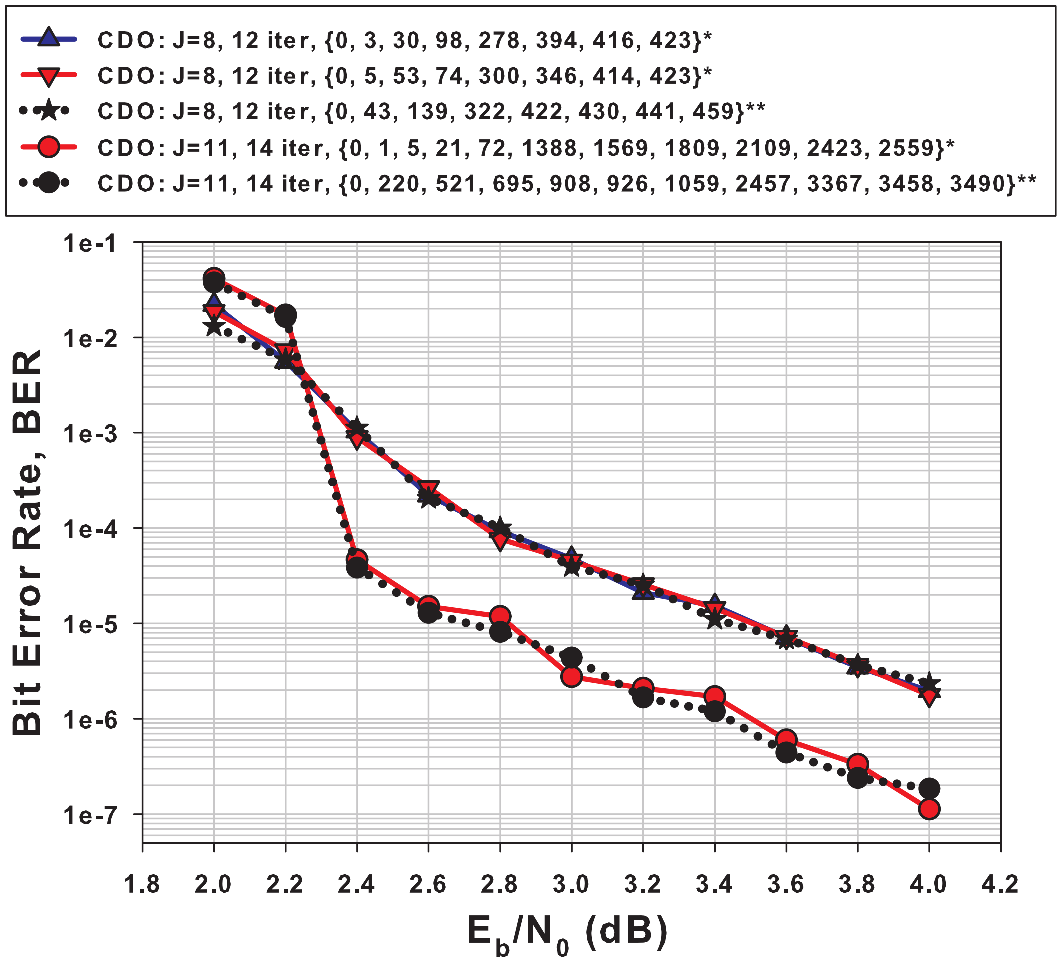


Figure 3.5 Rate $\frac{1}{2}$ systematic CDO code error correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) after the 12th ($J = 8$) and the 14th ($J = 11$) decoding iteration. Novel codes presented are marked with a single asterisk ('*'): the codes of order $J = 8$ have an optimal span. Codes presented in [20] are marked with a double asterisk ('**').

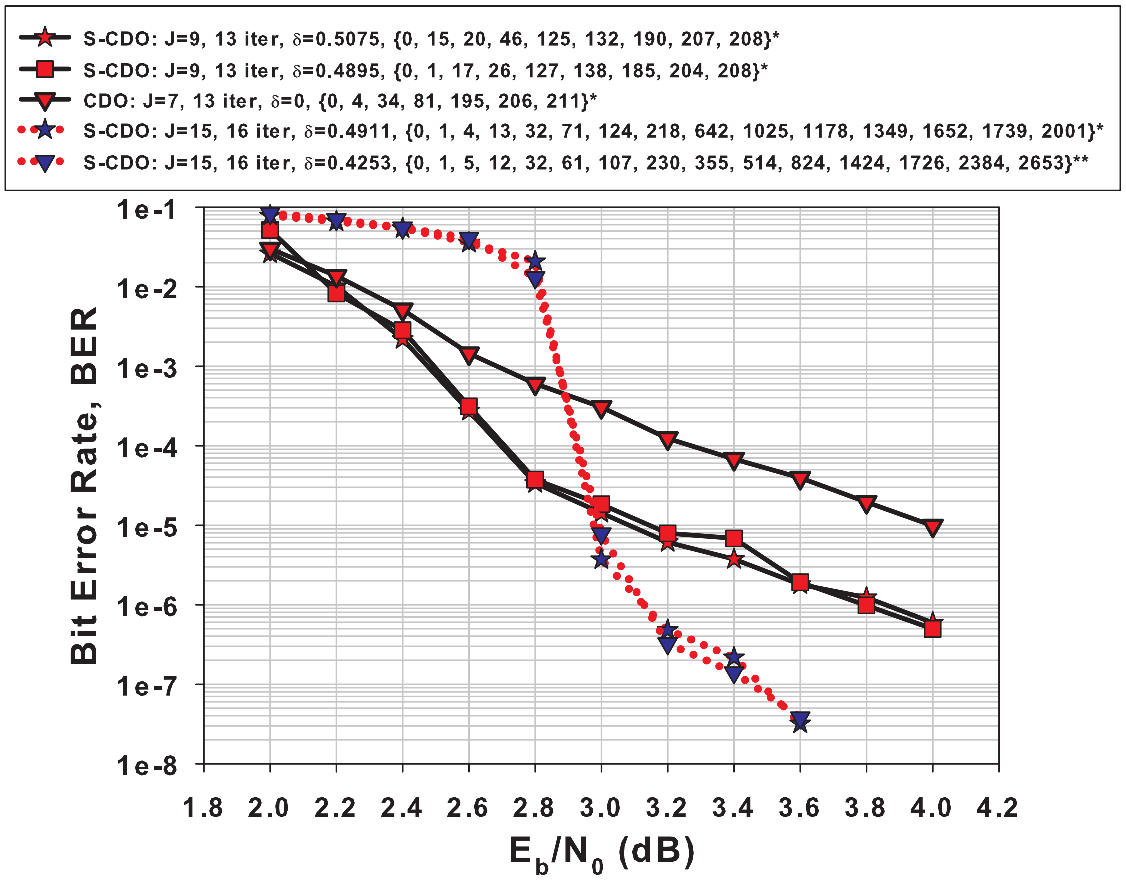


Figure 3.6 Rate $\frac{1}{2}$ systematic CDO and S-CDO code error correction performance for $J \in \{7, 9\}$, $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) and $J = 15$, $\frac{E_b}{N_0} \in [2.0; 3.6]$ (dB), after 13 and 16 iterations respectively. Novel codes presented are marked with a single asterisk ('*'): the S-CDO codes of order $J = 9$ and the CDO code of order $J = 7$ have an optimal span. The S-CDO code presented in [21] is marked with a double asterisk ('**').

3.2.5 Conclusions

This paper has presented an efficient implicitly-exhaustive search algorithm for finding rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes with shortest possible spans. The drastic speedup it offers over previous exhaustive-search and pseudo-random search algorithms is achieved through a search-space reduction technique. The proposed method is a type of *branch and bound algorithm*, thus performing an exhaustive search: as a consequence, we have been able to prove the optimality of the optimal-span codes found, which was previously not possible except for *very small* values of J (i.e. codes with a very small number of generator connections).

In addition to being able to yield codes of shorter span than what was possible with the best pseudo-random algorithms available, the proposed algorithm has allowed us to find new rate $R = \frac{1}{2}$ systematic codes that are optimal for $J \in \{6, 7, 8\}$ (CDO codes) and $J \in \{9\}$ (S-CDO codes). The span improvement of 25% for S-CDO codes ($J \in \{14, 15\}$) and $13\% - 26\%$ for CDO codes ($J \in \{10, 11\}$) will directly translate into a latency reduction of the same magnitude in the novel error-correcting encoding/iterative threshold decoding systems for which they are intended. The obtained results pave the way for a new generation of even faster *Convolutional Self-Doubly Orthogonal* code searching techniques, and hence more powerful and optimal new codes.

CHAPTER 4

AN EFFICIENT PARALLEL AND IMPLICITLY-EXHAUSTIVE SEARCH ALGORITHM

4.1 Overview and discussion

In Chapter 3, we presented an improved implicitly-exhaustive search algorithm for finding systematic rate $R = \frac{1}{2}$ optimal-span (S-)CDO codes. The algorithm [15] uses three techniques, defined in Theorems 3.1, 3.2 and 3.3 (see Section 3.2.3.3), to reduce the size of the (S-)CDO code search space, thus yielding new optimal-span codes as well as codes with shorter spans. Nevertheless, its operation being sequential in nature, the algorithm does not fully take advantage of the performance offered by modern multi-core computer systems. Furthermore, it is plagued by some of the limitations that are present in the reference exhaustive-search algorithm [5].

In this chapter, the second article of this thesis [14] is included verbatim in Section 4.2: it provides a high-level description of a completely novel *efficient and parallel implicitly-exhaustive search algorithm* that greatly reduces the computational time required for finding systematic rate $R = \frac{1}{2}$ optimal-span (S-)CDO codes. In order to increase the speed and efficiency of the search process, the algorithm exploits significant algorithmic improvements. First, it uses a *stricter set of constraints* to identify and concentrate the search on only potentially valid codes. Indeed, Theorems 4.1 and 4.2 presented in this chapter replace Theorems 3.1 and 3.2 from [15], since they improve the efficiency of the tree-traversal by further reducing the size of the search space. Next, the algorithm reduces the computation time by performing a simultaneous exploration of independent regions in the search-tree: this *parallel search* allows it to leverage the additional computing cores in modern multi-core microprocessors [19]. Finally, the algorithm performs an *incremental computation* with *data-reuse*, allowing it to reuse the results of previous computations to increase the efficiency of the (S-)CDO code validation process. Using this technique, the validation function is able to greatly reduce the number of computations that are necessary for validating a code:

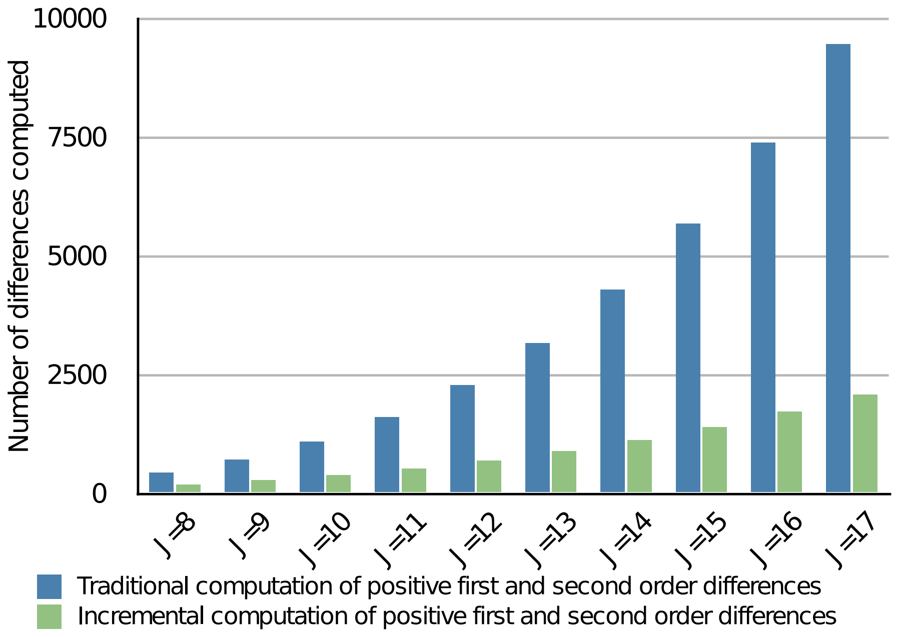


Figure 4.1 Number of first and second order differences required to validate a (S-)CDO code - traditional vs. incremental computation.

although the number of differences required is still expressed by polynomial equations, their degree has been effectively decreased by one when compared to (2.5), (2.6), and (2.7), thus resulting in significant computational savings that become larger as J increases (see Fig. 4.1). For example, when validating a code of order $J = 17$, the proposed method allows for a reduction factor of 4.5 in the total number of computed first and second order differences. We recall that this algorithm is *implicitly-exhaustive*, because no nodes that could potentially yield valid codes with a span shorter than the shortest known span at the end of the process are discarded, thereby ensuring that the search remains exhaustive in nature, in an implicit manner, and that it yields optimal-span codes.

Using the *efficient and parallel implicitly-exhaustive search algorithm* presented in this chapter, we were able to obtain optimal-span $J = 9$ CDO codes and $J \in \{10, 11\}$ S-CDO codes, offering a span reduction of 16%, 9% and 24% respectively. Furthermore, we were also able to obtain, within an acceptable amount of computation time, new $J \in \{10, 12, ..., 17\}$ CDO codes and new $J \in \{12, ..., 20\}$ S-CDO codes having the shortest spans published to date for these higher values of J. Figure 4.2 shows, for $J \in [9; 20]$, the (S-)CDO code span

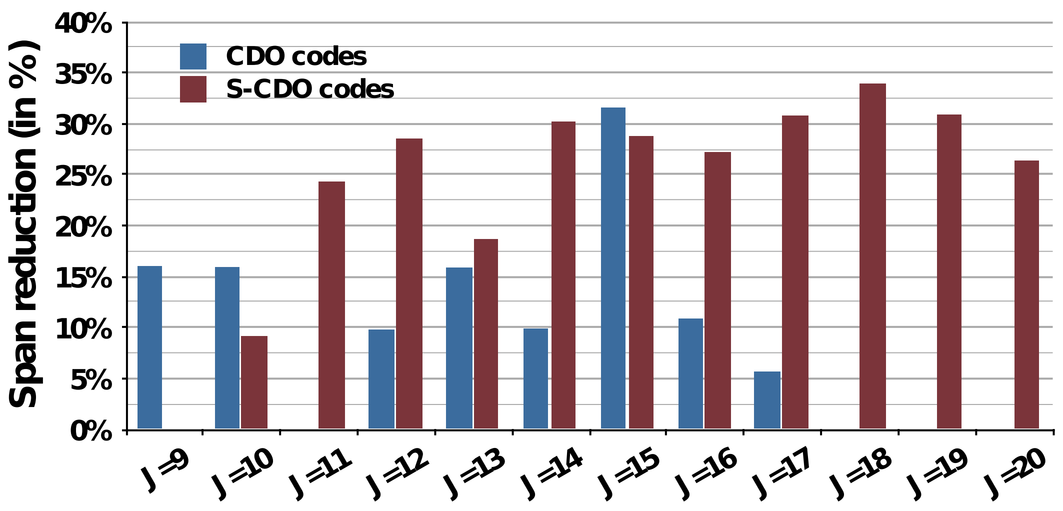


Figure 4.2 *Span Improvement* in percentage [14] for (S-)CDO codes and $J \in [9; 20]$. Note that for $J = 9$ (CDO codes) and $J \in \{10, 11\}$ (S-CDO codes) the codes obtained have the shortest possible spans for those orders (i.e. optimal-span codes).

reduction achieved by using the novel algorithm presented in Section 4.2: it was possible to reduce the span by an average of 14% for CDO codes and 26% for S-CDO codes, which directly translates into a latency reduction of the same magnitude in the error-correcting systems for which they are intended.

We have previously observed that the ease of implementation of high-performance (S-)CDO code decoders may vary as a function of the encoder generator vector chosen [77]. Therefore, for each order J, a choice of two codes having the shortest spans obtained is presented in Section 4.2.4.1 of this chapter. Furthermore, their spans are compared to known theoretical bounds and the error-correction performance for some of these codes is provided. As noted in Chapter 3, the simulation of each error-performance curve required several weeks of computation time to deliver the $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) bit error rates obtained, and thus several instances of the (S-)CDO code simulation software were run in parallel and on multiple computers. The error-correction performance for all of the codes provided in Chapter 4 is presented in Appendix A, and their density maps are presented in Appendix B. The complete list of improved codes obtained through the use of this novel algorithm is presented in Appendix C.

Finally, Section 4.2 presents the evolution of the error-correction performance for (S-)CDO codes as a function of the order J: we show that although the bit error rate is lowered as J increases, the "*waterfall*" region of the error-performance curve migrates to higher values of E_b/N_0, a fact that may require consideration depending on the application of interest.

4.2 Article #2: Efficient Parallel Search Algorithm for Determining Optimal $R = 1/2$ Systematic Convolutional Self-Doubly Orthogonal Codes

G. Kowarzyk, N. Bélanger, D. Haccoun, Y. Savaria
École Polytechnique de Montréal
{*gilbert.kowarzyk, normand.belanger, david.haccoun, yvon.savaria*}*@polymtl.ca*

Publication source: IEEE Transactions on Communications, vol. 61, no. 3, March 2013, pp. 865-876.

Abstract

A novel parallel and implicitly-exhaustive search algorithm for finding, in systematic form, rate $R = \frac{1}{2}$ optimal-span Convolutional Self-Doubly Orthogonal (CDO) codes and Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes is presented. In order to obtain high-performance low-latency codecs with these codes, it is important to minimize their constraint length (or "span") for a given J number of generator connections. The proposed exhaustive algorithm uses algorithmic enhancements over the best previously published searching techniques, yielding new and improved codes: we were able to obtain new optimal-span CDO/S-CDO codes (having order $J \in \{9\}$ and $J \in \{10, 11\}$ respectively), as well as new codes having the shortest spans published to date for higher values of J ($J \in \{10, 12, ..., 17\}$ and $J \in \{12, ..., 20\}$ for CDO and S-CDO codes respectively). The new codes and their error performance are provided. An analysis of the evolution of the CDO/S-CDO code error performance as J increases is presented, and the shortest CDO/S-CDO code span values for each given J are compared.

Index Terms: *Convolutional codes, self-doubly orthogonal codes, systematic codes, threshold decoding.*

4.2.1 Introduction

An iterative error-control coding scheme presented in [4, 6, 7] differs from the classical Turbo code procedure invented in 1993 [8, 9], as it uses no interleaver, neither at the encoding nor at the decoding process. The iterative threshold decoding algorithm it uses [4, 9, 11] employs a new class of convolutional codes expressed in systematic form and that must satisfy double orthogonality properties, beyond those of the well-known orthogonal codes used in the conventional non-iterative threshold decoding [10]. Throughout this paper, the widely used terminology *systematic codes* is used to represent *convolutional codes expressed in systematic form*, that is, codes whose encoders are systematic. We only consider rate $R = \frac{1}{2}$ codes, hence only one encoder generator vector is provided. The additional so-called double orthogonality properties required from the codes ensure a quasi-independence of the observables over the first two decoding iterations, thereby allowing the use of an iterative decoding procedure and hence a good error performance while allowing attractive trade-offs between complexity, latency, and error performance [6].

As the error-correcting capability of the new codes depends essentially on the dimension J of the vector generator of the $R = \frac{1}{2}$ code [20], and because the code constraint length (or span of the code) has a direct impact on the latency of the system, it is of great importance to search for rate $R = \frac{1}{2}$ systematic Convolutional Self-Doubly Orthogonal (CDO) codes (and their variants) having the shortest possible spans for any given J number of connections. Since no systematic deterministic method for solving this problem is currently known, the code searching is usually conducted using heuristic search algorithms [20, 28]. However, although finding a CDO code is relatively easy, determining the shortest span codes for a given J has eluded analysis and is still an open problem. In fact, the search for optimal CDO codes (and their variants) is far more computationally challenging than the problem of finding "optimal" simply orthogonal codes, also known as the *Golomb ruler* problem, which has an *NP-hard* complexity [13, 46]. Indeed, CDO codes may be viewed as second-order Golomb rulers.

Pseudo-random and exhaustive search algorithms have been previously developed to obtain good, i.e. short span, CDO codes [5, 20, 28]. However, using these algorithms to find ever shorter span or eventually optimal span (i.e. *shortest* span) codes requires a computational time that becomes rapidly excessive, especially as the number J of generator connections increases beyond $J = 5$. Preliminary work on reducing the computational time required by the reference exhaustive search algorithm [5] consisted in adapting it to perform a basic parallel search [19]. Although the resulting technique showed that the search for optimal-span codes lends itself well to parallel processing, it is also plagued by many of the inefficiencies that are present in the reference algorithm. In fact, a more efficient technique for reducing the search time through an improved implicitly-exhaustive search process was described in [15], but its operation being sequential in nature, it does not fully take advantage of the performance offered by modern multi-core computer systems.

In this paper, we present a completely novel parallel and implicitly-exhaustive search algorithm that greatly reduces the computational time required for finding optimal-span CDO codes and their variants. We show that the drastic increase in speed and efficiency is achieved through very significant algorithmic improvements over the algorithm presented in [15], in particular: a parallel search, an incremental computation with data-reuse, and an enhanced dynamic search-space reduction based on a stricter set of constraints to identify and concentrate the search on only potentially valid codes. This faster technique has allowed finding several new optimal-span codes having larger J values. Moreover, short of obtaining optimal codes for some yet higher J values, we have also been able to obtain, within an acceptable amount of computation time, new codes with significantly shorter spans than the ones previously published in [15, 20, 21, 28].

The paper is organized as follows: in Section 4.2.2, CDO and S-CDO codes (a CDO code variant) are defined to establish the notation used in this paper. A novel and more efficient parallel implicitly-exhaustive search algorithm is described in Section 4.2.3. In Section 4.2.4, new optimal-span codes, as well as novel codes with a shorter span than previously obtained in [15, 20, 21, 28] are presented. The shortest S-CDO code spans and their CDO counterparts are compared. Finally, their error performance, and an analysis of the evolution of their error-performance as J increases are provided.

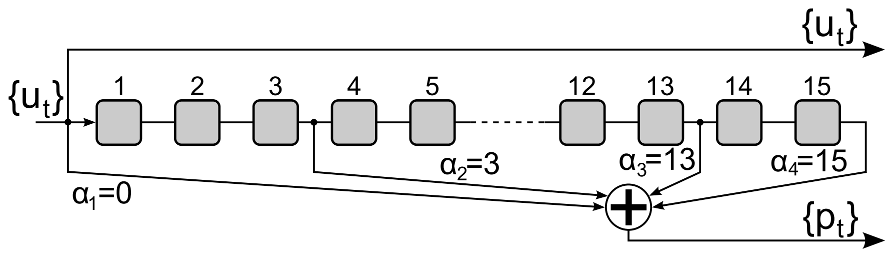


Figure 4.3 Example of systematic CDO code encoder: $R = \frac{1}{2}$, $J = 4$, $M = 15$, $\Omega = \{0, 3, 13, 15\}$.

4.2.2 Definitions

In this section, we provide the necessary definitions on the vector generator of the systematic CDO and S-CDO codes of coding rate $R = \frac{1}{2}$ [4, 49], which, as previously stated, may be viewed as second-order Golomb rulers [43].

4.2.2.1 Convolutional Self-Doubly Orthogonal (CDO) codes

A systematic Convolutional Self-Doubly Orthogonal (CDO) code of coding rate $R = \frac{1}{2}$ and order J is defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers $(\alpha_1 < \alpha_2 < ... < \alpha_J)$ such that the following conditions are satisfied [6, 20, 21, 49]:

1. The elements in S, the set of first-order differences between these integers, are all distinct:
$$S = \{s_{k,l} = (\alpha_k - \alpha_l) \ : \ k \neq l\};$$

2. The elements in D, the set of second-order differences (the differences between the differences), are all distinct from one another, with the exception of the unavoidable differences caused by the permutations of indices (l, m) or (k, n):
$$\begin{aligned} D = \ & \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \ : \\ & k \neq l, m \neq n, k \neq m, l \neq n\}; \end{aligned}$$

3. The elements in sets S and D are distinct from one another ($D \cap S = \emptyset$).

The α_i elements ($i \in [1; J]$) represent the connections between the encoder shift register and the modulo-2 adder. By convention α_1, the first integer in our set, is always equal to zero

($\alpha_1 = 0$). The span M of a CDO code is equal to α_J, the largest integer in Ω, and corresponds to the length of the encoder shift register (see Fig. 4.3), that is, α_J is the encoder memory length [20]. The number J of elements in Ω is equal to the number of generator connections of the code and is called the order of the CDO code. An optimal CDO code of a given order J is defined as a CDO code whose span M_{opt} is the *smallest* span that exists for that order. However, an optimal CDO may not be unique and hence there may be more than one optimal CDO code for any given order J.

Since the validity of a code as a CDO code depends only on the relationship between the J elements composing it, any subset of L consecutive elements from the set defining a CDO code also forms a valid CDO code, albeit one of smaller order L, $L < J$. For example:

$$\begin{aligned} \mathrm{CDO}_{J=5} &= \{0, 1, 24, 37, 53\} \\ \mathrm{CDO}_{J=4} &= \{0, 1, 24, 37\} \\ \mathrm{CDO}_{J=3} &= \{0, 1, 24\} \end{aligned} \tag{4.1}$$

are all valid, although not optimal, CDO codes. This property will be leveraged to speed-up the proposed algorithm.

When calculated directly as per the definition, first-order and second-order differences come in pairs of *equal magnitude* but *opposite sign.* The number of *positive* first-order differences (N_S) and second-order differences (N_D) that exist for a code are a function of J, given by [20]:

$$N_S^J = \frac{J(J-1)}{2} \tag{4.2}$$

$$N_D^J = \frac{J(J^3 - 2J^2 + 3J - 2)}{8}. \tag{4.3}$$

Directly computing the exact span of optimal codes, whether simply or doubly orthogonal, is still an unsolved problem [13, 46]. However, a loose lower bound for the span of a CDO code has been developed in [20, 50] and can be expressed as a function of J, the order of the code, using (4.2) and (4.3) as follows:

$$\alpha_J \geq \alpha_J^* = \left\lceil \frac{N_S^J + N_D^J}{2} \right\rceil . \tag{4.4}$$

Furthermore, any CDO code has always a symmetrical (mirror) equivalent composed of integers with the same differences but in the reverse order, a property shared with the so-called Golomb ruler problem they are related to [20]: the symmetrical equivalent of {0, 2, 12, 15} would therefore be {0, 3, 13, 15}. This property will also be used to speed-up the proposed search algorithm.

4.2.2.2 Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes

Simplified Convolutional Self-Doubly Orthogonal (S-CDO) codes are obtained by relaxing the second CDO condition, yielding codes with shorter spans than regular CDO codes. The latency of the decoding process is in direct proportion to the span of the code and the number of iterations used for reaching a given error performance. Thus, using S-CDO instead of CDO codes substantially reduces the decoding latency at the cost of only a very small degradation of the error-correction performance [5, 21, 51].

A systematic S-CDO code of coding rate $R = \frac{1}{2}$ and order J is thus defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers such that it satisfies the *first* and *third* CDO conditions, and a *modified version of the second* condition, as follows [5, 51]:

2b) The set D of second-order differences between the integers in Ω, defined as:

$$\begin{aligned} D = \ & \{d_{m,n}^{k,l} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \ : \\ & k \neq l, m \neq n, k \neq m, l \neq n\} \end{aligned}$$

is composed of $2N_D$ second-order differences (of which $2N_D^e$ have an equal value in the set D), computed by excluding the unavoidable second-order differences caused by the permutations of indices (l, m) or (k, n). We define δ, the simplification coefficient, as $\delta = \frac{N_D^e}{N_D}$, where $N_D^e < N_D$ and $0 \leq \delta \leq 1 - \frac{N_S}{N_D}$ [5].

Clearly, a CDO code may be viewed as a S-CDO code for which $\delta = 0$, and thus S-CDO and CDO codes share most of their properties. A loose lower bound on the span of a S-CDO code has been derived as a function of J and δ, and is expressed, using (4.2) and (4.3), as [5, 21]:

$$\alpha_J^* = \left\lceil \frac{N_S + (1-\delta) \cdot N_D}{2} \right\rceil . \tag{4.5}$$

An optimal S-CDO code of order J and simplification coefficient δ is thus a S-CDO code having the smallest span, M_{opt}, which exists for that order and δ. Again, there may be more than one optimal S-CDO code for a given order J and simplification coefficient δ. In this paper, the notation "(S-)CDO" will be used when referring to both CDO and S-CDO codes.

We now present an efficient search procedure for obtaining $R = \frac{1}{2}$ systematic (S-)CDO codes with a span shorter than codes previously published.

4.2.3 Novel Efficient Parallel Implicitly-Exhaustive Search Algorithm

In this section, we present a brief overview of previous (S-)CDO code searching techniques and the key concepts behind the implicitly-exhaustive search algorithm developed in [15]. Then, we describe a novel and efficient parallel implicitly-exhaustive search algorithm. The several and significant enhancements it introduces to the search algorithm in [15] led to a drastic increase in searching speed for finding new optimal-span codes and codes with shorter spans than those previously published in [15, 20, 21, 28] for the same orders.

4.2.3.1 Overview of previous search algorithms

The first technique for finding CDO codes used a projective geometry approach to yield valid codes [50, 71]. However, this approach yielded codes with excessively large spans, leading to the development of new code-searching methods [20, 50].

Recent code-searching algorithms can be divided into two categories, exhaustive and pseudo-random. *Exhaustive* search algorithms guarantee that the optimal span for a given J number of connections is found, provided that a sufficiently large initial M_{curr}, the smallest known span, is used. This is achieved by testing all the potentially valid codes in the search space. However, as J becomes larger, the very rapidly increasing computational effort required to obtain optimal (S-)CDO codes led to dropping this type of algorithm in favor of more practical pseudo-random search algorithms [5, 50]. *Pseudo-random* search algorithms are based on the use of a pseudo-random rejection criterion that can be adjusted in order to shorten the spans of the codes obtained [20]. Note that this type of algorithm cannot guarantee that minimal-span codes have been found. However, with the use of a span reduction method based on modulo operations [50] and a carefully chosen lower bound, the technique did provide some of the codes with the shortest known spans, albeit possibly not optimal codes [5, 20].

In order to find new optimal-span codes for larger values of J, an attempt at improving the reference exhaustive-search algorithm [5] is described in [19]: the computation time is reduced by means of a very basic simultaneous exploration of different regions of the search space. Although this preliminary brute-force parallel approach showed that the problem lends itself well to parallel computing, it quickly became clear that a more capable algorithm would be required to address the exploding size of the search space as J and M_{curr} increase. As a consequence, the search algorithm in [15] was developed: it uses a more effective *implicitly-exhaustive* searching technique for efficiently reducing the size of the search space while still performing an exhaustive search, and thus, was able to yield new optimal-span codes having larger values of J.

In this paper, we have focused on developing new methods to significantly improve the computational performance of the search algorithm in [15] by further reducing the size of the search space and by making a more efficient use of the resources offered by modern multi-core computer systems.

4.2.3.2 An implicitly-exhaustive search

The (S-)CDO code exhaustive-search algorithm described below uses a tree-like structure to perform the search (see Fig. 4.4 for $J = 3$). The tree is composed of nodes which must have a value larger than their parent node and their sibling nodes to the left. Sibling nodes are nodes with the same parent. The tree depth represents the total number of connections J, while the values of the nodes on a path from the root to a leaf represent the elements of $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$, the set of positive integers defining the code. All nodes at depth $J - 1$ are leaf-nodes. A *valid path* in this search-tree starts at the root node and ends at a leaf-node that has a value not larger than M_{curr}, the currently smallest known span value.

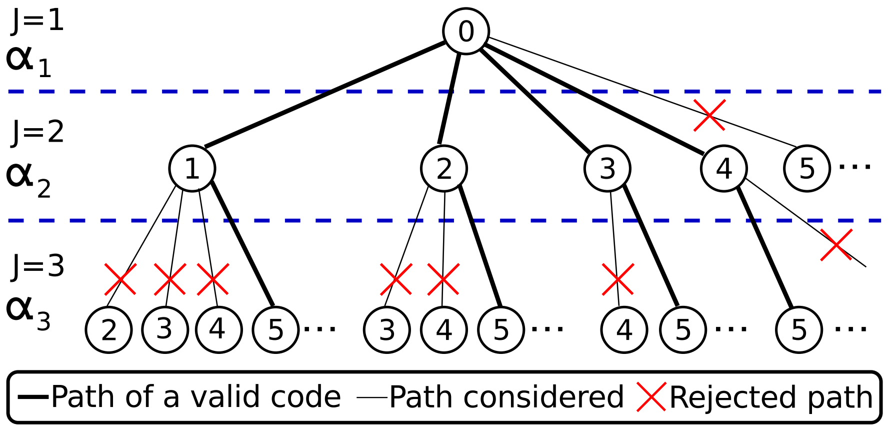


Figure 4.4 (S-)CDO search-tree: searching for a CDO with order $J = 3$.

The (S-)CDO code search-tree size depends on the current span, M_{curr}, and on J. Since there is only one path leading from the root node to a leaf-node, the number of leaf-nodes in the search-tree, N_L, also represents the total number of possible paths (or (S-)CDO code candidates). Since the set Ω defining a code always starts with zero (the root node), the number of possible combinations of $J-1$ nodes with integer values ranging from 1 to M_{curr} may be expressed as:

$$N_L = \begin{pmatrix} M_{curr} \\ J-1 \end{pmatrix} = \frac{(M_{curr})!}{(J-1)!\,(M_{curr}-J+1)!}.$$

It can be seen that the number of leaves quickly explodes as M_{curr} and J increase. Clearly, tree-pruning techniques should be used as much as possible.

The novel algorithm presented in this section is *implicitly*-exhaustive and belongs to the “branch and bound” class of algorithms [73]: it does not need to test all the nodes and paths, but still performs an exhaustive search while reducing the search complexity by several orders of magnitude. Since an implicitly-exhaustive algorithm still performs an exhaustive search, the codes that are obtained are *proven to be optimal* [15].

4.2.3.3 Improving the implicitly-exhaustive search: a more aggressive dynamic search-space reduction

The novel algorithm described in this section proposes an exhaustive searching technique that is faster at finding rate $R = \frac{1}{2}$ systematic (S-)CDO codes with a shorter span than the best pseudo-random and exhaustive search algorithms presently known, yielding optimal-span codes for higher J order values than previously known ($J \leq 9$ for CDO, $J \leq 11$ for S-CDO). For codes with a large order ($J \geq 10$), where an exhaustive-search can be prohibitive, this new search-method also allows to gradually obtain codes with significantly shorter spans than previously known, thus allowing to find “better” codes within the same amount of computation time.

One way that the *efficient parallel implicitly-exhaustive* search algorithm reduces the computational time required for searching for optimal (S-)CDO codes is by using more aggressive search-tree pruning techniques to further reduce the number of branches that are explored. Indeed, children nodes of nodes that have been discarded do not need to be traversed or validated, thus allowing for substantial computational savings. To that effect, the novel algorithm provides two significant search-tree pruning enhancements over the *implicitly-exhaustive* search algorithm in [15]: Theorems 1 and 2 from [15] are replaced by Theorems 4.1 and 4.2 provided below.

The first search-tree pruning technique consists in realizing that Theorem 3 in [15] may also be applied in the direction going from the leaf-nodes to the root node. From (4.1), any subset of a (S-)CDO code must also be a valid (S-)CDO code, and thus it is possible to establish a tighter upper bound when generating the list of children nodes to be evaluated. Indeed, the theorem below states that it is possible to reduce the search space by choosing an upper bound with a smaller value than the one used in Theorem 1 of [15], which stated that $V^d_{max} = M_{curr} - ((J-1) - d)$. Improvements on this upper bound value are obtained through the use of the lower bound (4.4) (for CDO codes only), or, if such a value is known, by using the optimal span value for a code of order $J - d$ when generating the maximum value of nodes at depth d. A separate list of known optimal spans is used for CDO and S-CDO codes in order to generate the correct maximum value during the search.

Theorem 4.1. *Let V_{max}^{d} be an upper bound for node values having $d < J - 1$. Then, V_{max}^{d} can be defined as the smallest of the following three values:*

- *$M_{curr} - M_{opt}^{J-d-1}$, where M_{curr} is the current shortest known span for a code of order J, and M_{opt}^{J-d-1} is the optimal-span of a code of order $J - d$, if it is known;*
- *$M_{curr} - \alpha_J^*$, where M_{curr} is the current shortest known span and α_J^* is the lower bound calculated as per (4.4) (for CDO codes only);*
- *$V_{max}^{d+1} - 1$ for $d < J - 1$, where V_{max}^{d+1} is the V_{max}^{d} value for children nodes of the current node, and the current node is not a leaf-node.*

Proof. By definition, the minimum value of the *difference* between the current leaf-node value M_{curr} and the value of a node at depth d is equal to V_{min}^{J-d-1}, as per Theorem 3 in [15]. This is true because the difference would correspond to the lower bound of a symmetrical code starting at the leaf-node and ending at the root node. *QED* □

Theorem 4.1 above is equivalent to the *Maximum Position Reduction* technique used in the search for optimal Golomb rulers [17].

The second search-tree pruning improvement complements Theorem 4.1 by limiting the maximum value of the α_{mid} nodes, where $mid = \left\lfloor \frac{J-1}{2} \right\rfloor + 1$. Indeed, this theorem also exploits the symmetry property of (S-)CDO codes mentioned above in Section 4.2.2 in order to discard some additional codes whose symmetrical has already been encountered during the tree exploration, as stated in Theorem 4.2 below.

Theorem 4.2. *Let $V_{max}^{\alpha_{mid}}$ be defined as:*

$$V_{max}^{\alpha_{mid}} = \left\lceil \frac{M_{curr} + 1}{2} \right\rceil - 1 \tag{4.6}$$

where $mid = \left\lfloor \frac{J-1}{2} \right\rfloor + 1$. Then, any code with an α_{mid} node having a value larger than $V_{max}^{\alpha_{mid}}$ would have a symmetrical *equivalent within the codes in the search-tree having $\alpha_{mid} \le V_{max}^{\alpha_{mid}}$.*

Proof. Assuming that the tree is traversed in the same order as in the reference exhaustive search algorithm, $V_{max}^{\alpha_{mid}}$ can be seen as a symmetry center for all the possible connection

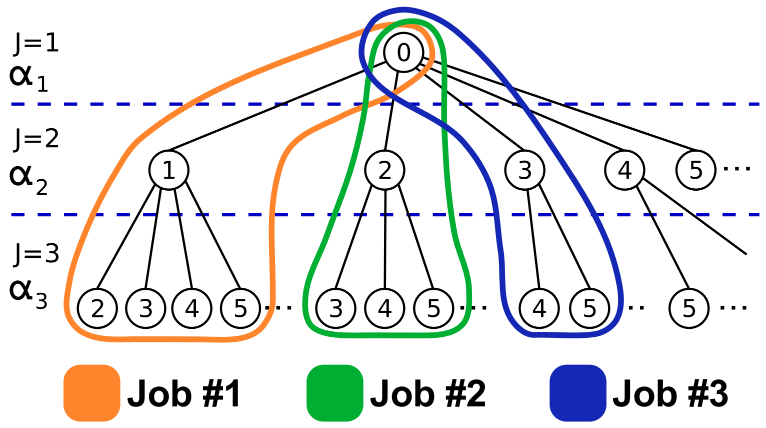


Figure 4.5 The parallel implicitly-exhaustive search algorithm divides the search-tree into a set of sub-trees (or "*jobs*") that are searched in parallel by the scout ants (here for a code of order $J = 3$).

patterns to an encoder shift register with a span M_{curr} (see Fig. 4.3). As soon as the middle connection, α_{mid}, crosses this symmetry center, all connection patterns thereafter will be symmetrical to the patterns before this threshold is crossed. Thus, all search-tree branches with an α_{mid} node value larger than $V_{max}^{\alpha_{mid}}$ may be safely skipped because if they were to lead to an optimal-span code, that code's symmetrical equivalent would have already been encountered earlier in the tree exploration. *QED* □

Theorem 4.2 is equivalent to the *Midpoint Reduction* technique used in the search for optimal Golomb rulers, and thus a search-space reduction of also approximately 50% can be achieved [17].

We recall that the use of these theorems only allows to discard either the nodes that cannot yield valid codes with a shorter span than the shortest span currently known, or one of the two codes in the pair of symmetrical (mirror) equivalent codes in the search space (as defined in Section 4.2.2). Therefore, no nodes that could potentially yield valid codes with a span shorter than the shortest known span at the end of the process are discarded, thereby ensuring that the search remains exhaustive in nature, in an implicit manner, and that it yields optimal-span codes.

4.2.3.4 Data reuse and parallel computation

Another way that the *efficient parallel implicitly-exhaustive* search algorithm reduces the computational time required for searching for optimal (S-)CDO codes is through data reuse and parallel computation. Previous (S-)CDO code searching algorithms [5, 15, 19, 20, 21] would compute *all* the first and second order differences of a code candidate in order to evaluate whether it is a valid (S-)CDO code or not. The novel algorithm presented in this paper is more efficient at validating a code by only computing the first and second order differences *generated by the last node addition*, and *reusing* the differences that were previously computed for the parent nodes in the same branch. For example, when searching for a code of order $J = 4$ and having computed the branch $\{0, 1, 5\}$, which is a valid CDO code, the node $\alpha_3 = 5$ is kept and candidate nodes α_4 are evaluated one after another. The positive first-order differences that exist for $\{0, 1, 5\}$ are "1", "5" and "4", and since they will not change as we evaluate different α_4 node values, they are *kept in memory* and *reused* for each code validation (the same applies to second-order differences). Thus, for each different α_4 node value, only the differences contributed by this last node addition need to be computed for evaluating the code candidate. It can be observed that the substantial computational savings obtained when employing *data reuse with incremental computation* increase as the order of the codes increases. Indeed, the number of first (N_S^J) and second (N_D^J) order differences (see (4.2) and (4.3)) that need to be computed for each code validation drop to $N_{S,incr}^J$ and $N_{D,incr}^J$ respectively, as given below:

$$N_{S,incr}^J = N_S^J - N_S^{J-1} = J - 1 \tag{4.7}$$

$$N_{D,incr}^J = N_D^J - N_D^{J-1} = \frac{J^3 - 3J^2 + 4J - 2}{2}. \tag{4.8}$$

Comparing (4.2) and (4.3) with (4.7) and (4.8), we can see that the degrees of the polynomials have all been reduced by one, and thus the algorithmic complexity of a validation goes from $O(J^4)$ to $O(J^3)$.

With multi-core computer systems becoming a commodity, it is important to have an algorithm that scales well and harnesses the computational power they offer by parallelizing the processing of data. The search algorithm presented in [15] focused on serial performance

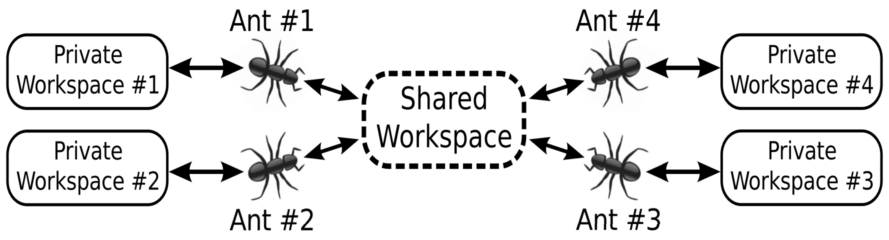


Figure 4.6 Ant colony with four ants: each scout ant has its own private workspace for keeping track of data pertaining to the current job, and all ants have access to a shared workspace for storing results and for communicating with each other.

and thus did not take advantage of modern microprocessors. This observation led to the development of the efficient *parallel* and implicitly-exhaustive search algorithm described in this paper, where we define a "*job*" as a sub-tree exploration.

To improve the efficiency of the search, the novel algorithm bases its behavior on that of an ant colony: conceptually, the search-tree is divided into a set of sub-trees that are explored in parallel, each by a different scout ant. For example, in Fig. 4.5, three *jobs* are depicted: the search-tree is thus divided into three independent sub-trees that have a fixed trunk going from the root node, α_1, to the sub-tree's base-node, here at α_2. The novel algorithm improves on [19] by recognizing that if the tree is traversed in the same order as in the reference exhaustive search algorithm, earlier branches in the tree will carry more nodes than later branches. Therefore, instead of assigning the sub-tree's base-nodes to α_2 nodes, it allows their depth to be configurable: by increasing the depth at which the base-nodes are located, the size of their corresponding sub-trees can be reduced, thus improving the algorithm's load balancing. Note that there is a single node per search-tree depth between the root node and the base-node of each sub-tree, and that their respective values do not change during the processing of a job: in fact, this unique node-value sequence is used as the job's id, and is employed for tracking *completed*, *active* and *pending* jobs to be processed. By dividing the search-tree into independent sub-trees, the scouts are able to work with very little resource contention: each scout has its own private workspace for keeping track of data pertaining to the current job, and all ants have access to a shared workspace for storing results and for communicating with each other (see Fig. 4.6). By partitioning the search space into independent sub-trees that are only accessed by one scout ant at a time, the ants can do most of the processing within their private workspace, foregoing the need of complex resource sharing mechanisms and thus reducing the overall synchronization overhead. When a scout has finished processing a job, another job is assigned to it until no more sub-trees are available, at which point the exploration of the search-tree is completed.

Using the shared workspace, the ant colony executes a *cooperative search*: when a scout discovers a valid code with a shorter span than the current shortest span known, its span value is shared with all other ants to collectively apply *all known* tree-pruning techniques to the current and future jobs being processed. This is another enhancement over the algorithm in [19], which would only set the maximum value for the leaf nodes to be equal to the new shortest recorded span and forego applying any other tree-pruning techniques. Sharing this information instantly benefits *all scout ants*, thus significantly decreasing the overall computation time by allowing the search space to converge to a smaller search-tree in less time. For example, in Fig. 4.7, the first ant to find a shorter span is *Ant #2*, with a span value of 55 during the processing of *Job #2*. Since this "*best known span value*" is written to the shared workspace and thus available to all ants, its updated value allows for immediately reducing the size of the search space for all scout ants, even if they had not yet themselves found an improved span value. In this example, if the computation had been executed with only one scout ant (i.e. serially), *Job #1* would have had to be completed before the first search space reduction could happen at *Job #2*, and thus, *Job #1* would not have been able to benefit from the search space reduction obtained during the processing of *Jobs #2* and *#3*. Figure 4.7 shows three ants working on different jobs and communicating with each other through the shared workspace. Since *Jobs #1*, *#2*, and *#3* are running in parallel, they can all benefit from each other's "*best known span value*" updates and thus apply search-tree pruning techniques earlier and at a faster rate than it would be otherwise possible with a non-collaborative approach. Through the collective span updates and the more aggressive tree-pruning techniques, the overall search-tree size converges more quickly to a smaller tree, potentially offering a better than linear performance with respect to the parallelism employed [19].

The novel *efficient parallel implicitly-exhaustive search algorithm* allowed us to find many new codes with shorter spans than the ones previously published for those orders [15, 20, 21,

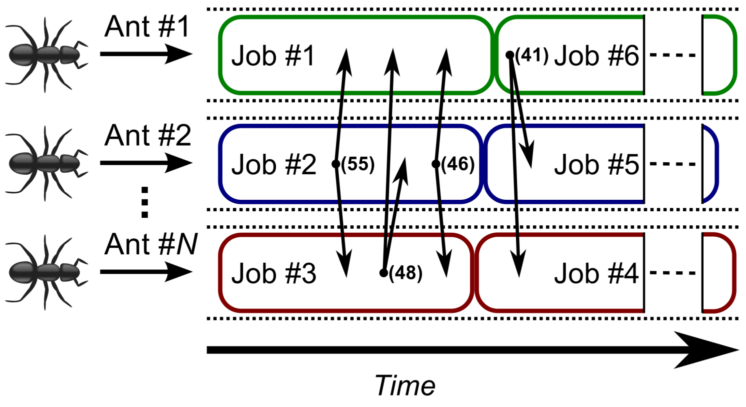


Figure 4.7 The ant colony executes a *cooperative search*: upon discovering a valid code with a shorter span than the shortest currently known, the improved span value is shared with all ants such as to collectively apply tree-pruning techniques to the jobs being processed.

28], and to find new optimal-span codes for $J = 9$ (CDO codes) and $J \in \{10, 11\}$ (S-CDO codes). Although the implementation details of this algorithm are beyond the scope of this paper and will be published elsewhere, suffice it to say that it was written in C and uses POSIX pthreads and mutex-protected shared variables to implement a cooperative multithreading model, thereby achieving a speedup of more than two orders of magnitude compared to the best algorithms in [5, 15, 19]. It is worth noting once again that although the algorithm presented in this section is much faster than previous exhaustive and pseudo-random search algorithms, its improvements and parallel execution do not compromise the exhaustive nature of the search.

The new codes obtained with this much-improved algorithm are presented in the next section.

4.2.4 New CDO and S-CDO Code Results

In this section, new optimal-span rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes are provided, as well as codes with a span shorter than any comparable previously reported codes. Their spans are compared to known theoretical lower-bounds [5, 20, 50], and the bit error-correction performance for some of these codes is presented. The span reduction obtained when using S-CDO codes instead of CDO codes is shown, and the evolution of their error performance as a function of their order J is described.

4.2.4.1 New rate $R = \frac{1}{2}$ systematic (S-)CDO codes

The novel algorithm presented in this paper allowed us to determine new optimal-span rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes, for orders $J = 9$ and $J \in \{10, 11\}$ respectively. Moreover, for $J \in \{9, 10, [12; 17]\}$ (CDO codes) and $J \in [10; 20]$ (S-CDO codes), we have been able to obtain, within a reasonable amount of computation time, new codes with significantly shorter spans than the ones previously published in [15, 20, 21, 28]. Note that the optimal-span S-CDO code of order $J = 9$ and the shortest known CDO code of order $J = 11$ were presented in [15], and as a consequence, they are not included in these results. Likewise, for values of $J \in [17; 20]$, span improvements were only attempted on S-CDO codes, and thus no CDO codes are provided. Although many codes were found, we chose to only present the two "best" codes for each order J: we have previously observed that the ease of implementation of high-performance (S-)CDO code decoders may vary somewhat as a function of the encoder generator vector chosen [77], and thus a choice of two codes is provided. It can be observed that with this novel technique it was possible to reduce the span by an average of 14% for CDO codes and by 26% for S-CDO codes. This span improvement directly translates into a latency reduction of the same magnitude in the error-correcting encoding/iterative threshold decoding systems for which they are intended. The codes obtained, along with their lower bound and the span reduction achieved are shown in Tables 4.1, 4.2 and 4.3.

Table 4.1 presents rate $R = \frac{1}{2}$ systematic (S-)CDO codes of order $9 \leq J \leq 13$. The optimal-span CDO code of order $J = 9$ has a span of $M_{J=9} = 766$, that is 16% shorter than 912, the previous shortest span known for that order [21]. The optimal-span S-CDO codes of order $J \in \{10, 11\}$ have a span of $M_{J=10} = 309$ and $M_{J=11} = 445$ respectively, 9% and 24% shorter than the previously shortest known spans. Their *simplification coefficients* δ have values $\delta = 0.5256$ and $\delta = 0.5279$ respectively. When observing all the known optimal-span S-CDO codes [15, 21], we can notice that the value of δ increases as J increases. This could possibly be used, for larger values of J, as a very approximative way of gaining insight on how close we are to finding the optimal span for those orders. The CDO codes of orders $J \in \{10, 12, 13\}$, have spans 10% to 16% shorter, and the S-CDO codes of order $J = 12$ and $J = 13$ have respectively spans that are 29% and 19% shorter than previously known codes [5, 20, 21]. Please note that CDO codes of order $J = 11$ were not included because no span improvements over [15] were obtained.

Table 4.2 presents rate $R = \frac{1}{2}$ systematic (S-)CDO codes of orders $J = 14$, $J = 15$ and $J = 16$. For CDO codes, the new shortest known spans have values of $M_{J=14} = 12416$, $M_{J=15} = 20219$ and $M_{J=16} = 31120$, with span improvements over the best known of 10%, 32% and 11% respectively. For S-CDO codes, the new shortest known spans have values of $M_{J=14} = 1373$, $M_{J=15} = 1890$ and $M_{J=16} = 2571$, with span improvements over the best known of 30%, 29%, and 27%, and δ values of 0.4881, 0.4956 and 0.4793 respectively.

Table 4.1 Summary of new rate $R = \frac{1}{2}$ systematic (S-)CDO codes obtained with the novel technique for $J \in \{9, 10, 11, 12, 13\}$

J	Code Type	Optimal-span?	New (S-)CDO codes	Lower Bound	Prev. best span (span reduction ††)	δ
9	CDO	NO	{0, 2, 24, 100, 428, 585, 667, 777, 792}	351 †	912 (13.16%)	0
9*	CDO	YES	{0, 2, 30, 108, 238, 537, 722, 763, 766}	351 †	912 (16.01%)	0
10	CDO	NO	{0, 1, 5, 96, 885, 1061, 1094, 1401, 1422, 1473}	540 †	1698 (13.25%)	0
10	CDO	?	{0, 1, 5, 99, 388, 789, 1128, 1359, 1401, 1428}	540 †	1698 (15.90%)	0
10	S-CDO	NO	{0, 7, 9, 83, 86, 118, 260, 296, 309, 317}	279 ‡	340 (6.76%)	0.5053
10*	S-CDO	YES	{0, 6, 10, 34, 111, 130, 234, 267, 298, 309}	268 ‡	340 (9.12%)	0.5256
11	S-CDO	NO	{0, 5, 8, 50, 123, 184, 303, 385, 399, 428, 448}	391 ‡	588 (23.81%)	0.5286
11*	S-CDO	YES	{0, 2, 10, 17, 52, 108, 187, 323, 398, 434, 445} △	391 ‡	588 (24.32%)	0.5279
12	CDO	NO	{0, 2, 5, 19, 63, 161, 1637, 2659, 3550, 3936, 4489, 4737}	1139 †	5173 (8.43%)	0
12	CDO	?	{0, 2, 5, 19, 63, 161, 1641, 2646, 3454, 3889, 4376, 4668}	1139 †	5173 (9.76%)	0
12	S-CDO	NO	{0, 6, 7, 16, 144, 270, 361, 470, 553, 583, 610, 648}	553 ‡	894 (27.52%)	0.5296
12	S-CDO	?	{0, 8, 9, 32, 160, 300, 438, 530, 551, 605, 633, 639} ▽	560 ‡	894 (28.52%)	0.5237
13	CDO	NO	{0, 2, 5, 19, 63, 161, 365, 1298, 4368, 4978, 5737, 7344, 7840}	1580 †	9252 (15.26%)	0
13	CDO	?	{0, 2, 5, 19, 63, 161, 365, 1553, 4016, 4553, 5658, 6789, 7785}	1580 †	9252 (15.86%)	0
13	S-CDO	NO	{0, 1, 4, 13, 32, 168, 532, 584, 725, 795, 872, 926, 998}	816 ‡	1217 (18.00%)	0.4963
13	S-CDO	?	{0, 12, 13, 16, 34, 83, 164, 374, 564, 685, 791, 949, 990}	792 ‡	1217 (18.65%)	0.5112

*	novel optimal-span codes of order J
△	completing the search for this optimal-span code required approximately 3 months of computation time
▽	the search was involuntarily interrupted after 7 months of computation time on a 12-core machine: the size of the search space is approximately 5000 times larger than that for S-CDO codes of order $J = 11$
†	lower bound calculated as per (4.4)
‡	lower bound calculated as per (4.5)
††	compared to the shortest span obtained in [21]

Table 4.2 Summary of new rate $R = \frac{1}{2}$ systematic (S-)CDO codes obtained with the novel technique for $J \in \{14, 15, 16\}$

J	Code Type	Optimal-span?	New (S-)CDO codes	Lower Bound	Prev. best span (span reduction)	δ
14	CDO	NO	{0, 1, 5, 21, 55, 153, 368, 856, 2912, 7031, 8493, 10825, 11937, 12505}	2139 †	13774 (9.21%) ††	0
14	CDO	?	{0, 1, 5, 21, 55, 153, 368, 856, 2919, 6512, 7772, 10032, 11480, 12416}	2139 †	13774 (9.86%) ††	0
14	S-CDO	NO	{0, 4, 5, 16, 30, 63, 172, 308, 746, 865, 952, 1212, 1312, 1377}	1096 ‡	1967 (29.99%) ††	0.4986
14	S-CDO	?	{0, 8, 9, 14, 35, 59, 248, 756, 855, 967, 1137, 1218, 1310, 1373}	1117 ‡	1967 (30.20%) ††	0.4881
15	CDO	NO	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3857, 8784, 13537, 16082, 18927, 20241}	2835 †	29532 (31.46%) ‡‡	0
15	CDO	?	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 4346, 10689, 13652, 16851, 19098, 20219}	2835 †	29532 (31.54%) ‡‡	0
15	S-CDO	NO	{0, 3, 4, 13, 28, 64, 108, 235, 609, 782, 1142, 1430, 1635, 1785, 1942}	1461 ‡	2653 (26.80%) ††	0.4940
15	S-CDO	?	{0, 10, 11, 14, 37, 69, 108, 254, 636, 1040, 1181, 1379, 1631, 1801, 1890}	1456 ‡	2653 (28.76%) ††	0.4956
16	CDO	NO	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 5566, 17437, 21413, 24642, 30654, 32618}	3690 †	34908 (6.56%) ††	0
16	CDO	?	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 5929, 13480, 20893, 22857, 29325, 31120}	3690 †	34908 (10.85%) ††	0
16	S-CDO	NO	{0, 10, 11, 14, 37, 69, 108, 223, 481, 1078, 1256, 1659, 1866, 2247, 2409, 2580}	1909 ‡	3532 (26.95%) ††	0.4908
16	S-CDO	?	{0, 11, 12, 15, 32, 71, 117, 228, 812, 1128, 1707, 1846, 2001, 2187, 2438, 2571}	1951 ‡	3532 (27.21%) ††	0.4793

†	lower bound calculated as per (4.4)
‡	lower bound calculated as per (4.5)
††	compared to the shortest span obtained in [21]
‡‡	compared to the shortest span obtained in [50]

Table 4.3 Summary of new rate $R = \frac{1}{2}$ systematic (S-)CDO codes obtained with the novel technique for $J \in \{17, 18, 19, 20\}$

J	Code Type	Optimal-span?	New (S-)CDO codes	Lower Bound	Prev. best span (span reduction ††)	δ
17	CDO	NO	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 8306, 15910, 27374, 36920, 45696, 48361}	4726 †	50071 (3.42%)	0
17	CDO	?	{0, 17, 18, 22, 64, 177, 409, 739, 1605, 2597, 5277, 8375, 20438, 30617, 37767, 44012, 47231}	4726 †	50071 (5.67%)	0
17	S-CDO	NO	{0, 1, 4, 13, 32, 71, 124, 218, 375, 671, 1294, 1563, 2290, 2497, 3022, 3281, 3452}	2479 ‡	4978 (30.65%)	0.4824
17	S-CDO	?	{0, 1, 4, 13, 32, 71, 124, 218, 375, 862, 1584, 2162, 2311, 2763, 2935, 3347, 3447}	2520 ‡	4978 (30.76%)	0.4737
18	S-CDO	NO	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 1033, 2444, 2759, 3084, 3589, 3902, 4462, 4589}	3244 ‡	6905 (33.54%)	0.4624
18	S-CDO	?	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 1027, 1419, 2193, 3112, 3565, 3824, 4299, 4565}	3185 ‡	6905 (33.89%)	0.4723
19	S-CDO	NO	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1763, 2429, 3620, 4137, 5419, 5690, 6390}	4036 ‡	8748 (26.95%)	0.4627
19	S-CDO	?	{0, 4, 5, 16, 30, 63, 128, 206, 358, 542, 787, 1163, 1878, 3260, 3532, 4524, 4811, 5731, 6046}	4007 ‡	8748 (30.89%)	0.4667
20	S-CDO	NO	{0, 8, 9, 14, 35, 59, 122, 213, 337, 484, 743, 1032, 1519, 2786, 3654, 5263, 5818, 6942, 7465, 7609}	5040 ‡	9749 (21.95%)	0.4549
20	S-CDO	?	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1162, 1682, 3065, 3517, 5250, 5602, 6167, 6861, 7177}	4831 ‡	9749 (26.38%)	0.4780

† lower bound calculated as per (4.4)

‡ lower bound calculated as per (4.5)

†† compared to the shortest span obtained in [21]

Table 4.3 presents rate $R = \frac{1}{2}$ systematic (S-)CDO codes of orders $J = 17$, $J = 18$, $J = 19$ and $J = 20$. For $J = 17$ CDO codes, the improved span is 6% shorter than the best known, with a value of $M_{J=17} = 47231$. For S-CDO codes, the new shortest known spans have values of $M_{J=17} = 3447$, $M_{J=18} = 4565$, $M_{J=19} = 6046$ and $M_{J=20} = 7177$ with span improvements over the best known of 31%, 34%, 31% and 26%, having δ values of 0.4737, 0.4723, 0.4667 and 0.4780 respectively.

Finally, we can observe that when using the shortest known spans, it is clearly more advantageous, latency-wise, to use a S-CDO code over a CDO code as J increases. Therefore a S-CDO code should be chosen over a CDO code of the same order as the span is substantially reduced (from 55% for $J = 6$ to 93% for $J = 17$).

4.2.4.2 Error performance simulation results for (S-)CDO codes

The error-correcting performance for some of the novel codes presented in this paper are shown in Figs. 4.8, 4.9, 4.10, and 4.11. These codes are meant to operate at moderate values of $E_b/N_0 \geq 3$ dB as will be seen below. With the exception of Fig. 4.9, the decoding was iterated until no significant performance improvement was observed.

Figure 4.8 shows the error performance for two CDO codes and two S-CDO codes of order $J = 14$, after 12 decoding iterations. As was described before in [15, 21], the error performance for these codes is more sensitive to the type of code (CDO vs. S-CDO) and its order J, than to its span. It can be seen that these four codes experience a "*waterfall*" region between 2.6 and 3.0 dB. This region starts earlier and is steeper for the CDO codes, since it ends at 2.8 dB instead of 3.0 dB, thus offering a coding gain of about 0.2 dB. Nevertheless, although the CDO codes offer a slightly better performance, starting at 3.0 dB, the error performance of the S-CDO codes and the CDO codes is roughly equal. However, the CDO codes have a span (and thus a decoding latency), that is approximatively *nine times* that of the S-CDO codes: despite the fact that from a theoretical perspective the CDO codes perform slightly better, from an engineering point of view the S-CDO codes offer a much reduced latency, and thus can be far more advantageous.

Figure 4.9 consists of four sets of two curves. The first set comprises the two *Maximum Free Distance* (MFD) nonsystematic Viterbi codes ($K = 7$ and $K = 9$). The second set

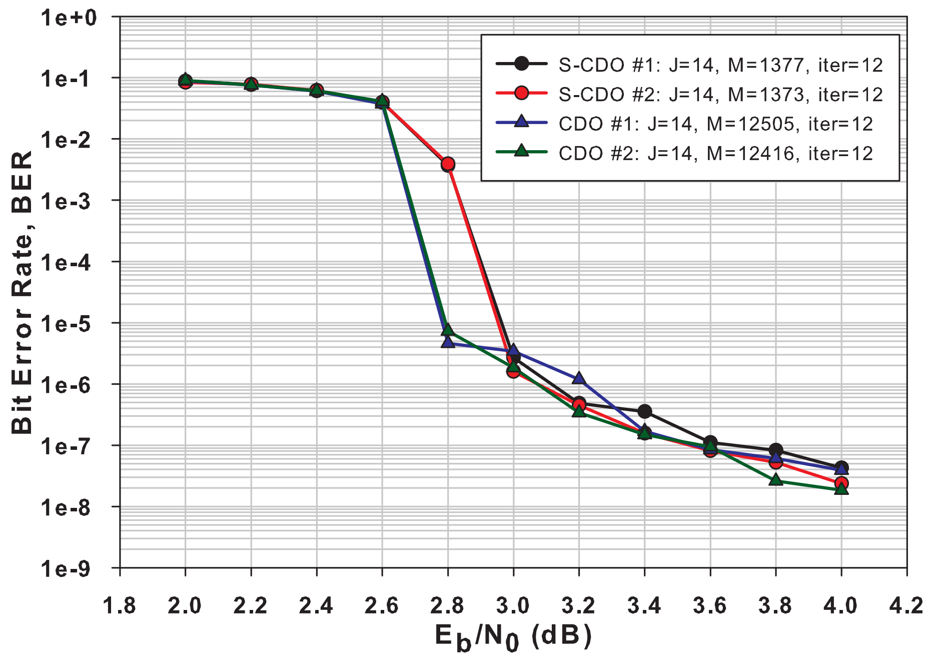


Figure 4.8 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) for $J = 14$, after 12 iterations.

comprises the *simulated* floor regions of two modern punctured rate-1/2 Turbo codes [22] employing an interleaver size of 1000 bits, and after 8 decoding iterations. The specific codes used were a pseudo-randomly punctured Turbo code (PRP-PCCC) and a punctured systematic Turbo code (S-PCCC), both having a rate-1/3 PCCC(1,5/7,5/7) parent code. Whereas the continuous line segments illustrate their simulated error performance as per [22], the shorter dashed-line segments represent a reasonable extrapolation of their respective Turbo code error floor tendencies. The third set comprises two CDO codes of order $J = 17$, after 20 decoding iterations: the number of iterations was increased until no significant performance improvement was observed. The fourth set comprises two S-CDO codes of order $J = 17$, after *only* 4 *decoding iterations*: the number of iterations was kept to 4 so that their decoding latency of about 13800 clock cycles would approximate the depicted Turbo codes' decoding latency of around 16000 clock cycles. Just as on Fig. 4.8, the two CDO codes have a roughly equal error performance, as do the two S-CDO codes. Please recall that for a same number of decoding iterations, when CDO and S-CDO codes have the same order,

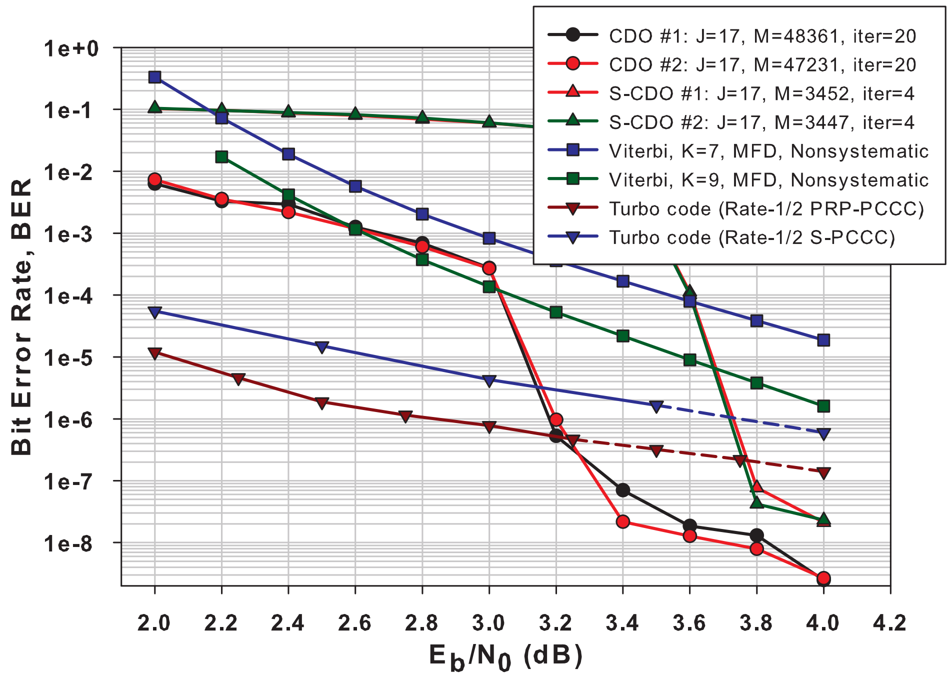


Figure 4.9 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) for $J = 17$. Included are performances of two CDO codes after 20 decoding iterations, together with those of two S-CDO codes after 4 decoding iterations (approximately matching the decoding latency of the included Turbo codes), $K = 7$ and $K = 9$ Viterbi codes (MFD, nonsystematic - *Odenwalder, 1970*), and the *simulated* floor regions of two modern punctured rate-1/2 Turbo codes [22] employing an interleaver size of 1000 bits, after 8 decoding iterations (specifically, we used a pseudo-randomly punctured Turbo code (PRP-PCCC) and a punctured systematic Turbo code (S-PCCC), both having a rate-1/3 PCCC(1,5/7,5/7) parent code). For the two Turbo codes, the continuous line segments illustrate their simulated error performance as per [22], whereas the shorter dashed-line segments represent a reasonable extrapolation of their respective error floor tendencies.

their error performance is very similar [21]. Nevertheless, since the S-CDO codes presented in Fig. 4.9 only use 4 decoding iterations, their "*waterfall*" region appears at slightly higher E_b/N_0 values of 3.4 dB to 3.8 dB, versus 3.0 dB to 3.4 dB for CDO codes. For moderate values of $E_b/N_0 \geq 3.0$ dB, and with this set of codes, we can conclude that if 20 decoding iterations are used, the (S-)CDO codes outperform the Viterbi codes for $E_b/N_0 \geq 3.1$ dB, and that they also outperform the Turbo codes for $E_b/N_0 \geq 3.2$ dB. Alternatively, if only 4 decoding iterations are used, the (S-)CDO codes outperform the Viterbi codes for $E_b/N_0 \geq 3.7$ dB, and the depicted Turbo codes for $E_b/N_0 \geq 3.8$ dB, despite having a 14% lower latency than the Turbo codes and a lower implementation complexity. For E_b/N_0 values smaller than 3.0 dB, the (S-)CDO codes cannot provide an interesting error performance: below 3.2 dB, Turbo decoding clearly outperforms both (S-)CDO and Viterbi decoding. However, beyond $E_b/N_0 = 3.0$ dB, Turbo decoding has already reached its error floor performance, and thus can be outperformed by (S-)CDO codes after their "*waterfall*" region. Furthermore, one can observe that the (S-)CDO code performance curves intersect the Turbo code curves at a much lower E_b/N_0 value than the Viterbi codes do. It should also be noted that for a same number of decoding iterations, these CDO codes have a span (and thus a decoding latency) that is almost over *fourteen times* that of the S-CDO codes, making the S-CDO codes, from an engineering perspective, far more interesting than their CDO code counterparts.

We can conclude that from a practical point of view, S-CDO codes are more advantageous than CDO codes, as they offer a lower decoding latency for a similar error-correcting performance. For moderate to high E_b/N_0 values (starting at 3.8 dB for the codes on Fig. 4.9), the S-CDO codes offer a compelling alternative to Turbo codes, as they provide a better error performance, at a lower latency and reduced implementation complexity.

4.2.4.3 Error performance evolution as J increases

Figures 4.10 and 4.11 describe the evolution of the bit error performance for CDO and S-CDO codes respectively, as the order J increases.

Figure 4.10 illustrates the performance of CDO codes for $J \in \{9, 10, [12; 17]\}$, at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), and after 15 iterations. All of our simulation results have shown that as J grows, the slope of the "*waterfall*" region slightly increases, eventually stabilizing. Furthermore, they have shown that this slope starts at higher values of E_b/N_0, but that the ensuing error floor is also lowered. An analysis explaining this behavior is still an open problem and beyond the scope of this paper. However, it can be observed that while in the iterative decoding of (S-)CDO codes the reliability of information bits improves with each decoding iteration, the reliability of parity symbols does not improve beyond the second decoding iteration, thus resulting in a performance degradation at lower E_b/N_0 values. As the E_b/N_0 values increase and the parity bits become less corrupted, the codes with larger values of J are able to show their true potential, thus reaching lower bit error rates. In other words, the

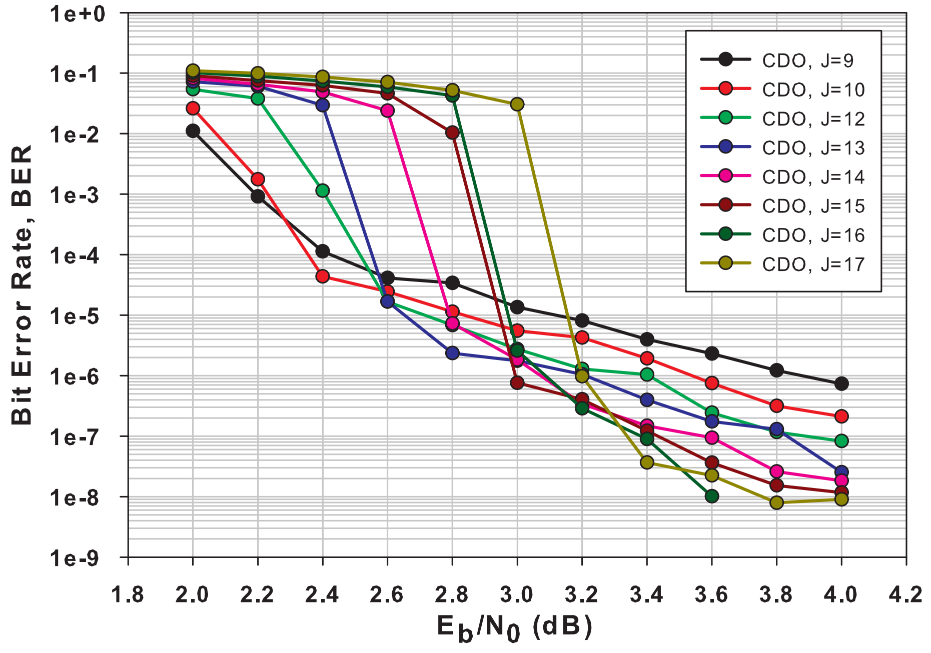


Figure 4.10 Evolution of CDO code error-correction performance as a function of J, for $J \in \{9, 10, [12; 17]\}$, at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), and after 15 iterations.

higher the value of J, the better the bit error rate, but at the expense of having a "*waterfall*" region that moves to higher values of E_b/N_0.

Figure 4.11 illustrates the performance of S-CDO codes for $J \in [10; 20]$, at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), and after 15 iterations. It can be clearly seen that as the value of J increases, the slope of the "*waterfall*" region slightly increases, eventually stabilizing, and that the bit error rate is lowered, but at the expense of having a "*waterfall*" region that migrates to higher values of E_b/N_0. It can also be observed, although less evident, that the coding gain offered by CDO codes over S-CDO codes in the "*waterfall*" region diminishes as J increases.

Finally, we can conclude that S-CDO codes are a more attractive alternative to CDO codes, due to the much reduced latency they offer. We can also conclude that depending of the application, it may not be of much use to employ S-CDO codes with higher values of J than 20, since the "*waterfall*" region may occur at E_b/N_0 values that are too high to be acceptable for the application of interest.

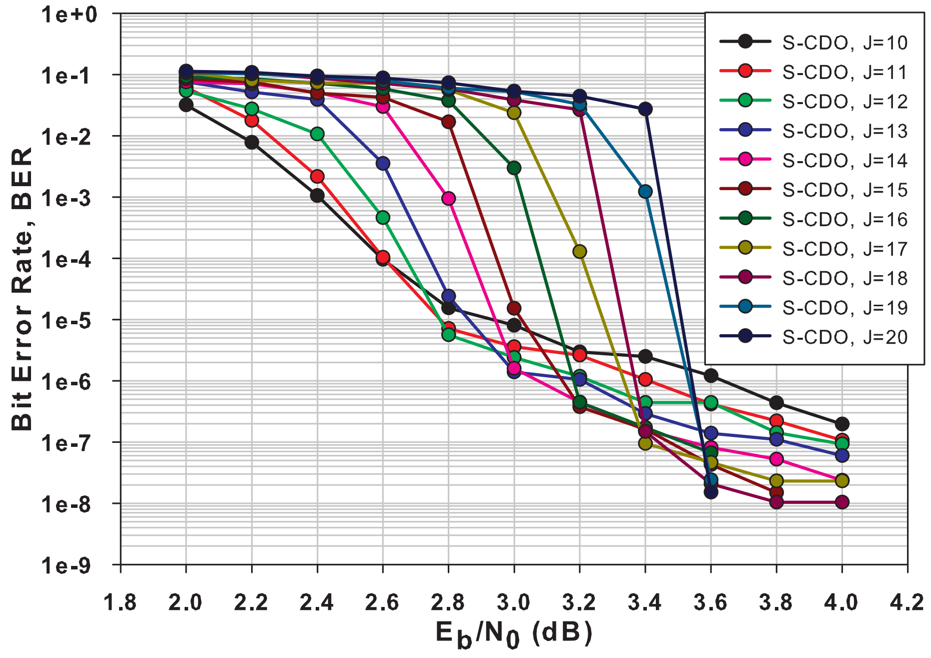


Figure 4.11 Evolution of S-CDO code error-correction performance as a function of J, for $J \in [10; 20]$, at $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), and after 15 iterations.

4.2.5 Conclusion

We have presented a high-performance efficient and parallel implicitly-exhaustive search algorithm for finding rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes with the shortest possible spans. The algorithm is faster at finding codes with shorter spans than previous exhaustive-search and pseudo-random search algorithms. The increase in speed and efficiency is achieved through significant algorithmic improvements: an incremental computation with data-reuse for validating codes in less time, an enhanced dynamic search-space reduction based on a stricter set of constraints to identify and concentrate the search on only potentially valid codes, and a parallel cooperative search in order to compute more search-tree branches at the same time, and to converge to a smaller tree at a quicker rate than what would otherwise be possible.

It is worth noting that although the algorithm presented in this section is much faster than the best previous exhaustive and pseudo-random search algorithms, its algorithmic improvements and parallel execution do not compromise the exhaustive nature of the search. We were thus able to prove the optimality of the optimal-span codes found, which was previously not possible except for very small values of J (i.e. codes with a very small number of generator connections). In addition to being able to yield codes of shorter span than what was previously possible, the proposed algorithm has allowed us to find new rate $R = \frac{1}{2}$ systematic codes that are optimal for $J = 9$ (CDO codes) and $J \in \{10, 11\}$ (S-CDO codes), and with spans that are respectively 16%, 9% and 24% shorter than the shortest previously known. Although many codes with shorter spans were found, we provide the two best codes for each order J, since we have observed in previous work that the ease of implementation of high-performance CDO/S-CDO code decoders may vary somewhat as a function of the encoder generator vector chosen. Through this technique, it was possible to reduce the span by an average of 14% for CDO codes and 26% for S-CDO codes, resulting in a latency reduction of the same magnitude in the error-correcting encoding/iterative threshold decoding systems for which they are intended.

The span of the provided codes is compared to known theoretical lower-bounds, and the error-correction performance for some of these codes is presented, along with the span improvements obtained when using S-CDO codes instead of CDO codes of the same order. Furthermore, the evolution of the error performance of CDO/S-CDO codes as a function of their order J is shown. For medium E_b/N_0 values (i.e. $\frac{E_b}{N_0} > 3$ dB), CDO/S-CDO codes offer a competitive error performance and a compelling alternative to Turbo codes, since their error performance curves may go below the "*floor*" region of Turbo codes, thus providing a better error performance along with a lower latency and reduced implementation complexity. The $J = 17$ CDO/S-CDO code error performances presented in this paper intersect the rate-1/2 PRP-PCCC and S-PCCC Turbo code curves at much lower E_b/N_0 values than the Viterbi $K = 9$ MFD code.

We also show that the greater the value of J, the better the error performance, but this is obtained at the expense of the "*waterfall*" region of the error performance moving to higher values of E_b/N_0. Thus, depending on the application, it may or not be advantageous to employ S-CDO codes with an order J greater than $J > 20$, since the "*waterfall*" region may occur at E_b/N_0 values that are too high to be acceptable for the specific application. Even though CDO codes perform slightly better than S-CDO codes for medium E_b/N_0 values, from an engineering point of view, S-CDO codes clearly offer a much lower decoding latency for a similar error performance, and thus may be better alternatives to CDO codes.

4.3 Further S-CDO code tree-traversal improvements over theorems presented in Chapters 3 and 4

It is worth mentioning that for S-CDO codes, it is possible to obtain slightly improved *lower* and *upper* bounds than the ones respectively provided by Theorem 3.3 in Section 3.2.3 and Theorem 4.1 in Section 4.2.3.3. These improvements leverage the list of known optimal spans to obtain tighter bounds, as we describe below.

Let G be the largest S-CDO code order for which an optimal span value is known, such that all optimal-span codes having order $J \in [1; G]$ are also known. For all $d^* \in \{0, ..., G-1\}$, let $M_{opt}^{d^*}$ represent the optimal span value of a code having order $J^* = (d^* + 1)$, where $J^* \leq G$. Therefore, $M_{opt}^{d^*}$ also represents the minimum possible value for an α_{d^*+1} element in $\Omega = \{\alpha_1, \alpha_2, ..., \alpha_G\}$ where $d^* \in \{0, ..., G-1\}$. We recall that any subset of consecutive elements in a valid S-CDO code must also form a valid S-CDO code (see (2.4) in Chapter 2). Therefore, for any pair of elements (α_i, α_j) from $\Omega = \{\alpha_1, \alpha_2, ..., \alpha_J\}$, where $j > i$ and $G > (j-i)$, the value of their difference must be such that $(\alpha_j - \alpha_i) \geq M_{opt}^{d^*=j-i}$. This minimum offset value between the (α_i, α_j) pair elements has to be respected past the last known optimal span $M_{opt}^{d^*=G-1}$, and therefore it is possible to compute a list M_{min}^{g} of minimum span values for nodes having depth g, where $(G-1) < g < (2*G-1)$, such that $M_{min}^{g=G-1+d^*} = M_{opt}^{G-1} + M_{opt}^{d^*}$, for $0 < d^* < G$. For S-CDO codes, Theorem 4.3 below improves Theorem 3.3 as follows:

Theorem 4.3. *Let V_{min}^{d} be a lower bound for S-CDO code node values having depth $d \leq (J-1)$, and let G be the largest S-CDO code order for which an optimal span value is known, such that all optimal spans having order $J \in [1; G]$ are also known. Let $M_{opt}^{d^*}$ be the optimal span corresponding to a code having order $J = d^* + 1$ and leaf nodes at depth d^* in the search-tree, where $0 < d^* < G$. Let $M_{min}^{g=G-1+d^*} = M_{opt}^{G-1} + M_{opt}^{d^*}$, such that $(G-1) < g < (2*G-1)$. Then, V_{min}^{d} can be defined as the largest* of the following three values*: the optimal span corresponding to the node's depth d, $M_{opt}^{d^*=d}$, for $0 < d < G$; the value $M_{min}^{g=d}$ for nodes having* *depth d, where $(G-1) < d < (2*G-1)$; the node value of the parent node plus one.*

Proof. By definition, the smallest value that a node at depth $d = N$ can have while being part of a valid (S-)CDO code is equal to M^{d}_{opt}, the optimal-span for a (S-)CDO code of order $J = N+1$. Furthermore, we recall that the validity of an S-CDO code depends only on the relationship between the J elements composing it, and that any subset of L consecutive elements from the code also forms a valid S-CDO code of order L, $L < J$. Thus, if $\Omega^{*} = \{\alpha_1, \alpha_2, ..., \alpha_J\}$ is a valid S-CDO code and b is an integer value, then $\Omega^{**} = \{\alpha_1+b, \alpha_2+b, ..., \alpha_J+b\}$ is a code that is equivalent to Ω^{*}, albeit starting at a non-conventional root node value of b. Therefore, the minimum offset values between a node at depth $d = G-1$ and a node at depth $g = d + d^{*}$ is equal to $M^{d^{*}}_{opt}$, where $d^{*} = g - d$, and thus $M^{g=d}_{min}$ is a lower bound for nodes at depth d, where $(G-1) < d < (2*G-1)$. Finally, during the search, the value of the parent node plus one is obviously a lower bound as well. Thus, the largest of these three lower bounds is the tightest lower bound among them. *QED* □

Similarly, for S-CDO codes, Theorem 4.4 improves on Theorem 4.1 (see Section 4.2) as follows:

Theorem 4.4. *Let V^{d}_{max} be an upper bound for node values having depth $d < (J-1)$, and let G be the largest S-CDO code order for which an optimal span value is known, such that all optimal spans having order $J \in [1; G]$ are also known. Let $M^{d^{*}}_{opt}$ be the optimal span corresponding to a code having order $J = d^{*}+1$ and leaf nodes at depth d^{*} in the search-tree, where $0 < d^{*} < G$. Let $M^{g=G-1+d^{*}}_{min} = M^{G-1}_{opt} + M^{d^{*}}_{opt}$, such that $(G-1) < g < (2*G-1)$. Then, V^{d}_{max} can be defined as the* smallest of the following three values*: $M_{curr} - M^{d^{*}=J-d-1}_{opt}$, where M_{curr} is the current shortest known span for a code of order J, and $M^{d^{*}=J-d-1}_{opt}$ is the optimal span of a code of order $J-d$, and where depth $d > (J-G-1)$; the value $M_{curr} - M^{g=J-d-1}_{min}$, where $(J-2*G) < d < (J-G)$; $V^{d+1}_{max} - 1$ for $d < (J-1)$, where V^{d+1}_{max} is the V^{d}_{max} value for children nodes of the current node, and the current node is not a leaf-node.*

Proof. By definition, the minimum value of the *difference* between the current leaf-node value M_{curr} and the value of a node at depth d is equal to V^{J-d-1}_{min}, as per Theorem 3 in [15]. This is true because the difference would correspond to the lower bound of a symmetrical code starting at the leaf-node and ending at the root node. *QED* □

Although these improvements were implemented in later versions of the algorithm, the resulting reduction in the size of the search space has not been characterized. Nevertheless, we expect them to provide a significant reduction in the size of the search space when searching for optimal-span S-CDO codes having order $J > G+1$. Please note that the two improvements for Theorems 4.3 and 4.4 described above only apply to S-CDO codes: for CDO codes, the use of the mathematical lower bound[1] α_J^* provides tighter upper and lower bounds than using M_{min}^g for that purpose, and thus Theorems 3.3 and 4.1 should be used instead.

[1]See (2.8) in Section 2.1.3.1.

CHAPTER 5

A HIGH-PERFORMANCE PARALLEL TREE-SEARCH FOR FINDING SHORTEST-SPAN ERROR-CORRECTING CDO CODES

5.1 Overview and discussion

In Chapter 4, we presented a high-level overview of the novel *efficient and parallel implicitly-exhaustive search algorithm* [14] that we developed for determining new optimal/short-span systematic rate $R = \frac{1}{2}$ (S-)CDO codes. Indeed, finding these codes is computationally very challenging (see Chapter 2), and is a problem that is exacerbated by the fact that finding optimal-span codes having order $J + 1$ is exponentially more complex.

In this chapter, the third article of this thesis [16] is included verbatim in Section 5.2: it focuses on describing the underlying techniques that were employed in the *efficient and parallel implicitly-exhaustive search algorithm* for leveraging the high performance offered by modern multi-core computer systems, thus achieving the speedup required for obtaining the new and improved $J \in [9; 20]$ (S-)CDO codes in [14]. Indeed, when compared to the reference exhaustive search algorithm [5], the resulting optimizations provide an impressive speedup factor that is greater than 16300 when searching for optimal-span CDO codes of order $J = 7$, and greater than 6300 when searching for optimal-span S-CDO codes of order $J = 8$. Moreover, the novel validation function exhibits an even more remarkable speedup factor: when compared to the reference (S-)CDO code validation function in [5], it is greater than 190000 for $J = 17$ CDO codes and greater than 60000 for $J = 17$ S-CDO codes, and when compared to the fastest known CDO code validation function used in high-performance pseudo-random search algorithms [20], it is greater than 2000 for $J = 17$ CDO codes. Figure 5.1 compares these validation functions: the speedup obtained as a function of J approximates a polynomial growth of order 5, for $J \in [8; 16]$.

The speedup is achieved through the use of a *vastly improved* code validation function that employs a *novel data structure* for enabling *data reuse* and *incremental computations*, and a *parallel dynamic search-space reduction technique* that substantially reduces the size

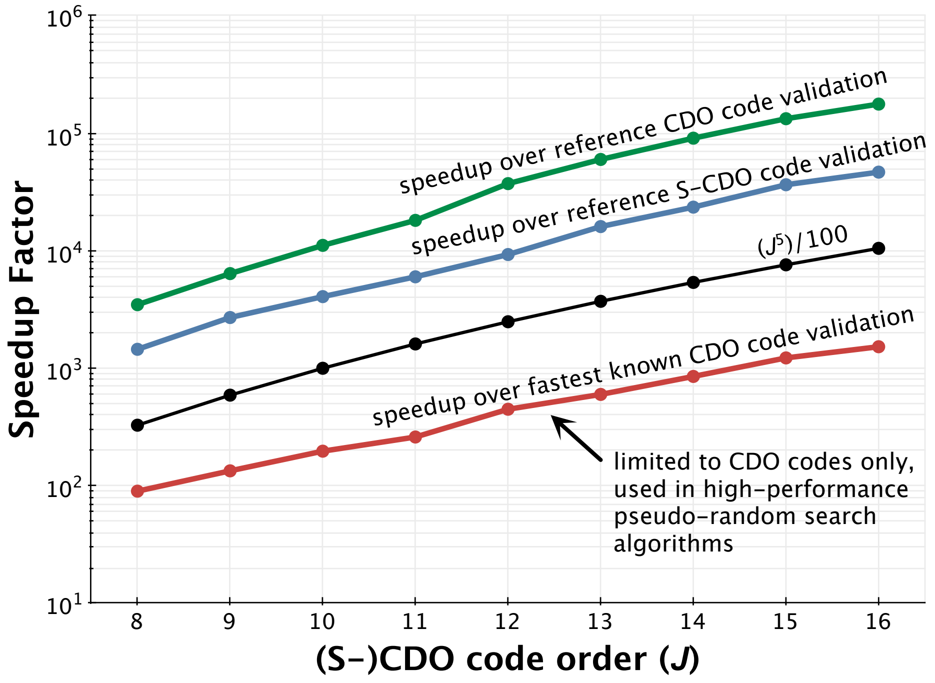


Figure 5.1 Scaling of (S-)CDO code validation speedup as a function of J, for $J \in [8; 16]$.

of the search-space without compromising the exhaustive nature of the search. In particular, Theorems 5.3, 5.2 and 5.1 are used in the search for optimal-span CDO codes, and Theorems 4.4, 5.2 and 4.3 are used in the search for optimal-span S-CDO codes (see Chapter 4), acting respectively as *lower*, *midpoint* and *upper* bound values for nodes in the search-tree. We also describe the *optimizations* and *load-balancing techniques* that allowed us to leverage hundreds of processor cores in order to complete an exhaustive search over a search-space that is some 10^{14} times larger than what was previously possible.

We now present verbatim the article submitted to *IEEE Transactions on Parallel and Distributed Systems* for review [16]: the various optimization techniques leading to the novel high-performance efficient and parallel implicitly-exhaustive (S-)CDO search algorithm are described, and the speedup achieved is provided.

5.2 Article #3: Optimizing the Parallel Tree-Search for Finding Shortest-Span Error-Correcting CDO Codes

G. Kowarzyk, N. Bélanger, D. Haccoun, Y. Savaria
École Polytechnique de Montréal
{*gilbert.kowarzyk, normand.belanger, david.haccoun, yvon.savaria*}*@polymtl.ca*

Publication source: IEEE Transactions on Parallel and Distributed Systems - submitted August 18, 2013.

Abstract

Finding optimal/short-span Convolutional Self-Doubly Orthogonal (CDO) codes and Simplified-CDO (S-CDO) codes for a specified order J is computationally very challenging. This paper describes several optimizations that were applied to an implicitly-exhaustive search algorithm in order to reduce the time required for finding these types of codes. The resulting high-performance parallel implementation provides an impressive speedup that is greater than 16300 (CDO, J=7) and 6300 (S-CDO, J=8) over the reference implicitly-exhaustive search algorithm, and greater than 2000 (J=17) over the fastest published CDO validation function used in high-performance pseudo-random search algorithms. These speedups are achieved through enhancements in the deterministic search-space reduction, and a vastly improved validation function that makes use of a novel data structure for enabling data-reuse and incremental computations. The resulting validation function speedup is greater than 60000 (S-CDO, J=17) and 190000 (CDO, J=17) when compared to its reference implementation. The combination of optimizations and load-balancing techniques allowed us to leverage hundreds of processor cores in order to complete an exhaustive search over a search space that is some 10^{14} times larger than what was previously possible.

Index Terms: *convolutional code, self-doubly orthogonal code, parallel tree-traversal, data-reuse, systematic encoder, threshold decoder.*

5.2.1 Introduction

In recent years, the rise of mobile devices has been accompanied by an ever increasing need for reliable high-bandwidth wireless communications. To mitigate or eliminate the errors that are introduced due to noise and interference in the communication channels, error-correcting coding schemes may be employed. These coding schemes are based on *codes* used to preserve the error performance while allowing the data-rate of digital communications to be increased and the transmission power at lower signal-to-noise ratios to be reduced, thus improving the overall power efficiency of these devices.

The error-control coding system presented in [4, 6, 7] offers very interesting improvements in latency and implementation complexity over the classical turbo code architecture used in reliable digital communication systems [8, 9]. Its iterative threshold decoding algorithm uses a new class of convolutional codes that must satisfy "double-orthogonality" properties, while not requiring interleavers, neither at the encoding nor at the decoding.

Their error-correcting capability depends essentially on the number J of generator vectors in the code, whereas the constraint length (or "span" of the code) has a direct impact on the latency of the system [20, 78]. As a consequence, it is important to use Convolutional Self-Doubly Orthogonal (CDO) and Simplified-CDO (S-CDO) codes with the shortest possible spans for a given J. Nevertheless, finding CDO/S-CDO codes having the shortest span has eluded analysis and is still an open problem. In fact, the search for optimal CDO codes (and their variants) is far more computationally challenging than the problem of finding optimal simply orthogonal codes (a.k.a. the *Golomb ruler problem*), which is believed to have an *NP-hard* complexity [13, 46]. Indeed, CDO codes may be viewed as second-order Golomb rulers [43].

While pseudo-random [20] and exhaustive search [5] algorithms exist and have been used to find codes of shorter spans than previously known (albeit not guaranteed to be optimal), the computational time required to find new shorter-span codes is very high as J increases. As high-performance computer systems are becoming a commodity, it is important to have an algorithm that scales well and that harnesses their computing power by using smarter data structures for the processing of data. This paper presents a series of optimizations that result in a very efficient parallel and implicitly-exhaustive tree-search implementation that greatly reduces the time required for finding optimal-span or shorter-span CDO/S-CDO codes by exploiting significant algorithmic improvements at all conceptual levels of the search. At the *application level*, unnecessary computations are avoided in the most often executed sections of the program. Focus is set on quickly invalidating rather than validating potentially good code candidates, that is, code candidates that cannot be optimal are eliminated as early as possible in the search. At the *algorithmic level*, a dynamic search-space reduction and a stricter set of constraints identifying potential code candidates allow for a significant decrease in the number of search-tree branches explored. Furthermore, an efficient validation function generating only half of the second-order differences to determine whether the validity conditions are met is employed. At the *implementation level*, a form of look-up table of differences is used to allow for constant-time complexity data access, and to eliminate the need for explicitly sorting and comparing all the differences with each other. This sophisticated low-maintenance data structure does not need to be initialized or cleared for each code validation, and is able to keep track of the relevant differences to facilitate data reuse: only the new differences produced by the next potential code candidate need to be computed. At the *hardware level*, redundant branch tests in the validation function are eliminated in order to reduce the performance penalty associated with branch mispredictions on modern microprocessors. The data structures and program are designed to be small enough to fit in state-of-the-art microprocessor caches, and the algorithm takes advantage of the increased parallelism offered by modern multi-core systems. Furthermore, implementing a dedicated load-balancing algorithm enabled us to efficiently leverage hundreds of processor cores. These techniques allowed us to obtain, within a reasonable amount of time, new optimal-span CDO and S-CDO codes as well as codes with significantly shorter spans than previously possible for a given number of J connections. The new (S-)CDO codes, their error-performance and a high-level overview of the *high-performance parallel and efficient implicitly-exhaustive search algorithm* that we have developed are presented in [14]. In this paper, we focus on describing the *optimization techniques* that were applied to the search algorithm used in [14] to reduce the time required for finding optimal-span CDO/S-CDO codes. We *characterize the speedup* obtained and show that using the novel algorithm and its efficient implementation, a very substantial *speedup of more than four orders of magnitude* is achieved.

The paper is organized as follows: in Section 5.2.2, we recall the definitions of CDO and S-CDO codes, and provide the notation used. Section 5.2.3 describes the CDO/S-CDO code search space and a brief overview of the reference exhaustive search algorithm. The proposed parallel implicitly-exhaustive tree-search optimizations are presented in Section 5.2.4, and the resulting *speedup* and *multi-core scaling* are presented in Section 5.2.5.

5.2.2 Definitions: CDO and S-CDO Codes

In this section, we provide the definitions and notations that are useful for the remainder of the paper. In order to keep this paper self-contained, these are repeated from [14].

A systematic Convolutional Self-Doubly Orthogonal (CDO) code of coding rate $R = \frac{1}{2}$ and order J is defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers ($\alpha_1 < \alpha_2 < ... < \alpha_J$) such that the following conditions are satisfied [6, 20, 21, 49]:

1. The elements in S, the set of first-order differences between these integers, are all distinct:
$$S = \{s_{k,l} = (\alpha_k - \alpha_l) \;:\; k \neq l\} \tag{5.1}$$
2. The elements in D, the set of second-order differences (the differences between the differences) are all distinct from one another, with the exception of the unavoidable differences caused by the permutations of indices (l, m) or (k, n):
$$\begin{aligned} D = \{d^{k,l}_{m,n} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \;: \\ k \neq l, m \neq n, k \neq m, l \neq n\} \end{aligned} \tag{5.2}$$
3. The elements in sets S and D are distinct from one another ($D \cap S = \emptyset$).

Simplified-CDO (S-CDO) codes are obtained by relaxing the second CDO condition, yielding codes with shorter spans than regular CDO codes. The latency of the decoding process is in direct proportion to the span of the code and the number of iterations used for reaching a given error performance. It has been shown that using S-CDO instead of CDO codes substantially reduces the decoding latency at the cost of only a very small degradation of the error-correction performance [5, 21, 51].

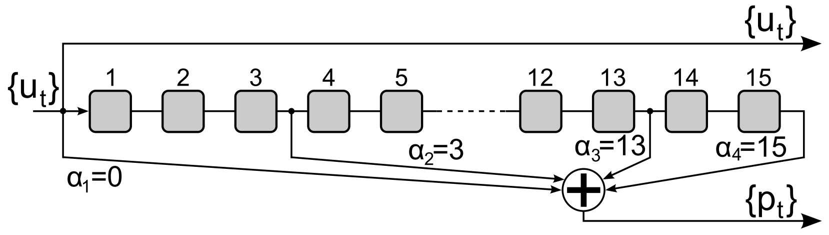


Figure 5.2 Example of systematic (S-)CDO code encoder: $R = \frac{1}{2}$, $J = 4$, $M = 15$, $\Omega = \{0, 3, 13, 15\}$.

A systematic S-CDO code of coding rate $R = \frac{1}{2}$ and order J is thus defined as the set $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$ of J ascendingly ordered positive integers such that it satisfies the *first* and *third* CDO conditions, and a *modified version of the second* condition, as follows [5, 51]:

2b) The set D of second-order differences between the integers in Ω, defined as:

$$\begin{aligned} D = \; & \{d_{m,n}^{k,l} = (\alpha_k - \alpha_l) - (\alpha_m - \alpha_n) \; : \\ & k \neq l, m \neq n, k \neq m, l \neq n\} \end{aligned}$$

is composed of $2N_D$ second-order differences, computed by excluding the unavoidable second-order differences caused by the permutations of indices (l, m) or (k, n), and of which $2N_D^e$ have a value that is not unique in D. Indeed, second-order difference values may repeat one or more times.

The notation "(S-)CDO" will be used when referring to both CDO and S-CDO codes, and hence clearly, a CDO code may be viewed as a S-CDO code for which $N_D^e = 0$.

The α_i elements ($i \in [1; J]$) represent the connections between the encoder shift register and the modulo-2 adder (see Fig. 5.2). By convention α_1, the first integer in our set, is always equal to zero ($\alpha_1 = 0$). The span M of a (S-)CDO code is equal to α_J, the largest integer in Ω, and corresponds to the length of the encoder shift register, that is, α_J is the encoder memory length [20]. The number J of elements in Ω is equal to the number of generator connections of the code and is called the order of the (S-)CDO code. An optimal (S-)CDO code of a given order J is defined as a (S-)CDO code whose span M_{opt} is the *smallest* span that exists for that order. However, an optimal (S-)CDO may not be unique and hence there may be more than one optimal (S-)CDO code for any given order J.

The validity of a code as a S-CDO/CDO code depends only on the relationship between the J elements composing it, thus any subset of L consecutive elements from the set defining the (S-)CDO code also forms a valid (S-)CDO code, albeit one of smaller order L, $L < J$ [14].

We recall that when calculated directly as per the definition, first-order and second-order differences come in pairs of *equal magnitude* but *opposite sign.* The number of *positive* first (N_S) and second (N_D) order differences are a function of J and can be expressed as follows [20]:

$$N_S^J = \frac{J(J-1)}{2} \tag{5.3}$$

$$N_D^J = \frac{J(J^3 - 2J^2 + 3J - 2)}{8}. \tag{5.4}$$

Directly computing the exact span of optimal codes, whether simply or doubly orthogonal, is still an unsolved problem [13, 46]. However, a loose lower bound for the span of a CDO code has been developed in [20, 50] and can be expressed as follows:

$$\alpha_J \geq \alpha_J^* = \left\lceil \frac{N_S^J + N_D^J}{2} \right\rceil. \tag{5.5}$$

Finally, any (S-)CDO code has always a symmetrical equivalent composed of integers with the same differences but in the reverse order, a property shared with the so-called Golomb ruler problem they are related to [20]: the symmetrical equivalent of {0, 2, 12, 15} would therefore be {0, 3, 13, 15}.

We now describe the (S-)CDO code search-space and present a brief overview of the key concepts behind the reference algorithm's tree traversal.

5.2.3 Problem Size and Tree Traversal

The algorithm described in this section performs the search for (S-)CDO codes using a tree-like structure (see Fig. 5.3 for $J = 3$). The tree is composed of nodes which must have a value larger than their parent node and their sibling nodes to the left. The tree depth represents the total number of connections J, and all nodes at depth $J - 1$ are leaf-nodes. The values of the nodes on a path from the root to a leaf represent the elements in $\Omega = \{\alpha_1, \alpha_2, \ldots, \alpha_J\}$. A *valid path* in this search-tree starts at the root node and ends at a leaf-node that has a value not larger than M_{curr}, the smallest known span value.

The search-tree size depends on the current span, M_{curr}, and on J. The number of leaf-nodes in the search-tree, N_L, represents the total number of possible paths (or (S-)CDO code candidates). Since the set Ω defining a code always starts with zero (the root node), the number of possible combinations of $J-1$ nodes with integer values ranging from 1 to M_{curr} may be expressed as:

$$N_L = \left(\begin{array}{c} M_{curr} \\ J-1 \end{array} \right) = \frac{(M_{curr})!}{(J-1)!\,(M_{curr}-J+1)!}. \tag{5.6}$$

Note that the number of leaves explodes as M_{curr} and J increase, thus making the search for optimal-span codes of order $J+1$ exponentially more complex.

In order to find new optimal-span codes for larger values of J, an attempt at improving the reference exhaustive-search algorithm [5] is described in [19]: the computation time is reduced by means of a very basic simultaneous exploration of different regions of the search space. Although this preliminary brute-force parallel approach showed that the problem lends itself well to parallel computing, it quickly became clear that a more capable algorithm would be required to address the exploding size of the search space as J and M_{curr} increase. Thus, the search algorithm in [15] was developed by the authors: it uses a more effective *implicitly-exhaustive* searching technique for efficiently reducing the size of the search space while still performing an exhaustive search. The proposed algorithm [14], whose very efficient implementation is described in this paper, combines and further improves these techniques [15, 19], thus yielding new optimal-span codes having larger values of J than any of the ones previously published in [20, 21, 28].

To our knowledge, the reference exhaustive-search algorithm [5] is the fastest algorithm for finding optimal-span (S-)CDO codes prior to our work in [14, 15, 19]. Therefore, its implementation will be used for comparison with the proposed search algorithm's implementation presented in this paper.

Conceptually, in these algorithms an (S-)CDO code with $N+1$ connections is built by

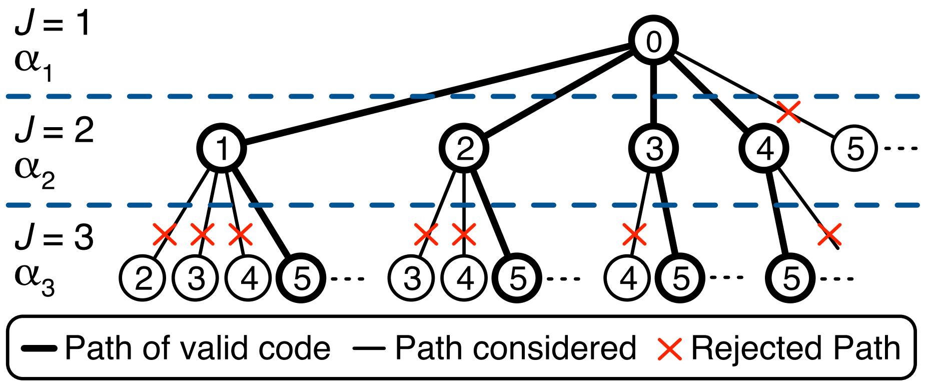


Figure 5.3 (S-)CDO search-tree: searching for a J=3 CDO.

taking an (S-)CDO code with N connections and adding another element α_{N+1} at the end of it, in such a way that the newly formed code is still valid and has a span that is no longer than M_{curr}, the best known span for that J. This tree traversal is performed in a depth-first sequence in order to allow optimizations as described later.

Since the reference algorithm [5] is used as the basis for the algorithm described in this paper, its pseudo-code for searching for $J = 3$ CDO codes is now briefly described. The algorithm begins by initializing M_{curr} to some high value, since no $J = 3$ CDO codes are considered currently known. Starting at the root node $\alpha_1 = 0$, the first available child node $\alpha_2 = 1$ is added (see Fig. 5.3). The validation routine is applied to the $\{0, 1\}$ code, and because it satisfies the CDO code conditions, the node is kept. The current number of connections is smaller than J, so the next available node $\alpha_3 = \alpha_2 + 1 = 2$ is added. Since $\{0, 1, 2\}$ fails the validation test, the node is therefore discarded. Nodes can be discarded either because the validation test fails or because their span is larger than M_{curr}. The process of adding, testing for validity, and discarding a node is repeated for all sibling nodes on a path until either the added node forms a valid CDO code, in which case the node is kept and its children are evaluated, or no more such siblings exist, in which case the next parent is evaluated. The best known span is updated when the current valid code has J connections and its span value is smaller than M_{curr}, leading to a very substantial tree pruning from that point on. When all valid paths of depth J with spans no larger than M_{curr} have been explored, the list of all optimal CDO codes will be all the codes with a span equal to M_{curr}.

The proposed algorithm, whose implementation is presented in the next section, is *im-*

plicitly-exhaustive and belongs to the "branch and bound" class of algorithms [73, 79]: using tree-pruning techniques, it performs an exhaustive search while reducing the search complexity by several orders of magnitude. Since this algorithm still performs an exhaustive search, the codes that are obtained are *proven to be optimal* [15].

5.2.4 An Efficient Parallel Implicitly-Exhaustive Search – Implementation

The algorithmic improvements that led to finding the new (S-)CDO codes presented in [14] are detailed in this section. The algorithm implements an exhaustive searching technique that is faster at finding rate $R = \frac{1}{2}$ systematic (S-)CDO codes with a span shorter than the best previously published pseudo-random and exhaustive search algorithms [5, 20, 21].

Since the size of the search-space is a combination of "M_{curr} choose $J-1$" (see (5.6)), linear improvements in performance are not sufficient to provide new results. Indeed, finding optimal-span (S-)CDO codes of order $J+1$ is exponentially more complex than finding optimal-span (S-)CDO codes of order J. The following sections elaborate on the implementation techniques which have allowed us to greatly accelerate the search for (S-)CDO codes with short spans.

5.2.4.1 A Validation Function Leveraging Data Reuse

The proposed validation function offers several key algorithmic enhancements resulting in a validation speedup of several orders of magnitude. First, as mentioned earlier, rather than attempting to *validate* a potential code candidate, it focuses on *invalidating* a code as quickly as possible. The function will return *false* and abort as soon as a first or second order difference breaks the validity conditions, thereby eliminating the need for generating the remaining differences. Other improvements include the elimination of the need for explicitly sorting and comparing all the differences with each other, and also a data structure allowing for data reuse and incremental computation of differences. Furthermore, a reduction in the number of differences computed is achieved through the use of improved loops requiring less branching tests and generating directly only half of the second-order differences.

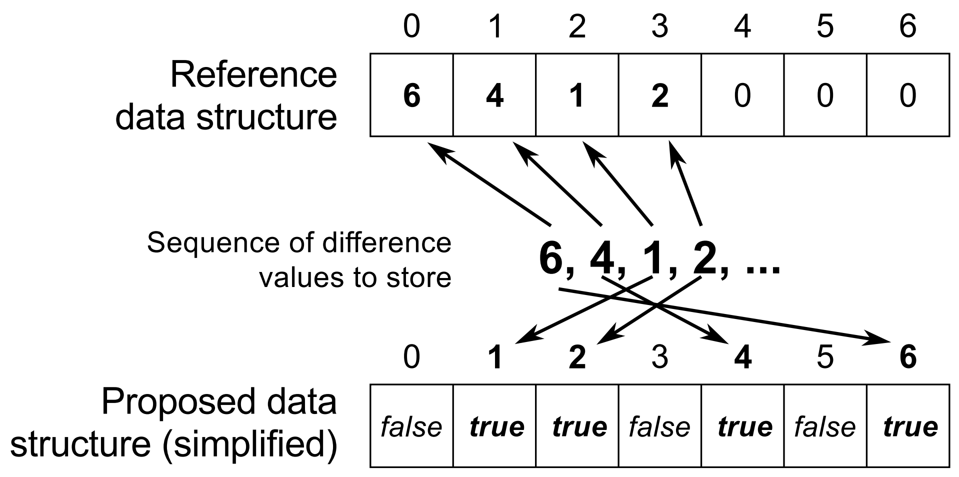

0 1 2 3 4 5 6
Reference data structure
6 4 1 2 0 0 0
Sequence of difference values to store
6, 4, 1, 2, ...
0 1 2 3 4 5 6
Proposed data structure (simplified)
false true true false true false true

Figure 5.4 Proposed data structure for storing differences (simplified).

5.2.4.1.1 Data Access Time / Sort-Compare Improvements In order to determine whether or not an (S-)CDO condition has been met, it is important to be able to recognize if a resulting first or second order difference has been encountered before. The term *collision* will be used to describe the situation where one or more differences result in the same value.

Rather than storing the data by value in an array of integers [5, 15, 20, 21], a look-up table using the size of the differences as its index is employed. The booleans in the table are initialized to *false*. When a difference is computed, the boolean at the index given by this difference is set to *true* to indicate that it was found. If the boolean was previously set to *true*, then a collision is detected. This is illustrated in Fig. 5.4.

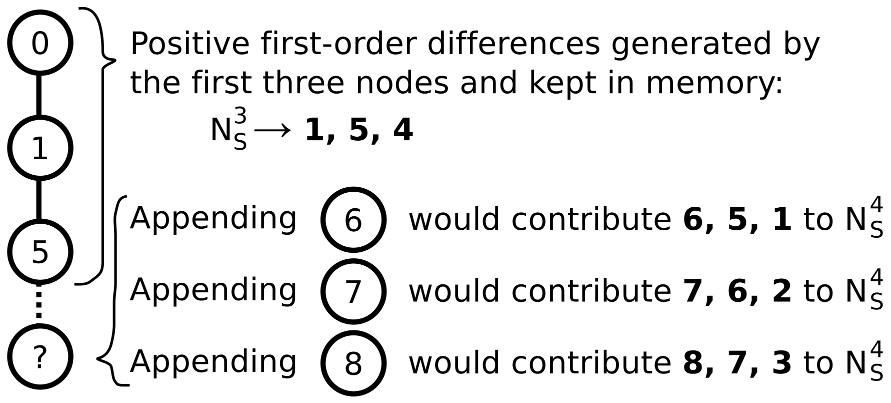
Storing difference values by tagging indexed array locations offers several advantages. First, the dataset is *implicitly sorted* as it is being stored in the data structure, and only one comparison per saved value is necessary to ensure that it is unique. This fine granularity for detecting a collision means that the validation function can abort and return *false* as soon as the first unwanted data repetition occurs: there is no need to evaluate the remaining differences. Furthermore, since the data structure acts as a lookup table for the validation function, access times are greatly reduced: saving a value or checking that it does not generate a collision can be done with a *constant* time complexity as a function of J.

5.2.4.1.2 Support for Data Reuse and Incremental Computation of Differences

A significant contribution in this paper is the method used to reduce the number of differences required for validating a potential (S-)CDO code in order to speed up the search. This is achieved by means of a deterministic tree-traversal, a *reuse* of previously computed data and an *incremental computation* of only *half* of the first and second order differences for each new code validation.

Indeed, rather than computing all of the differences for each code that is being tested as in [5, 15, 20, 21], the proposed algorithm only computes the differences *contributed by the last node addition*, reusing the rest of the previously generated differences that are stored in memory. Figure 5.5 shows a valid $J = 3$ CDO code $\Omega_J = \{0, 1, 5\}$ to which an attempt is made to append a node with the smallest possible value, such that the resulting $J = 4$ code is also a valid CDO code. As each potential $J = 4$ CDO code is tested, the differences that were computed to determine that $\Omega_{J=3}$ is a valid CDO code are kept in memory, and only the differences contributed by each appended node have to be generated, thus drastically reducing the total number of differences required to validate each $J = 4$ code candidate. In this example, appending a node $\alpha_4 = 6$ would generate two first-order difference collisions, since values 5 and 1 are already in N_S^3. However, neither node $\alpha_4 = 7$ nor node $\alpha_4 = 8$ would generate first-order difference collisions.

In order to support *data reuse* and *incremental computation of differences*, it is necessary to track which nodes are *active*, that is, which nodes are part of the current code candidate being evaluated. Furthermore, it is also necessary to be able to determine which differences were contributed by each node, such that they can be discarded when that particular node becomes *inactive*. Therefore, the data structure presented in the previous section was extended to associate each saved difference with the specific node that generated it: instead of using an array of booleans, an array of *tuples* is used to associate a *tuple* tag to an index representing the values of the positive difference results (see Fig. 5.6). Each *tuple* (D, I) in this $difference\_store$ array is composed of an integer D indicating the *depth* of the node having generated that difference, and an integer I indicating the *id* of the node having generated that difference. There is an independent set of *ids* for each node *depth*, and these increase from left to right on the search-tree. Each *tuple* in the $difference\_store$ array is initialized



0
1
5
?
Positive first-order differences generated by the first three nodes and kept in memory:
$N_S^3 \longrightarrow$ **1, 5, 4**
Appending 6 would contribute **6, 5, 1** to N_S^4
Appending 7 would contribute **7, 6, 2** to N_S^4
Appending 8 would contribute **8, 7, 3** to N_S^4

Figure 5.5 Node addition and incremental computation of differences (only first-order differences are shown).

to $(0, 0)$, which signifies that it is not associated with any node of the search-tree. Indeed, the combination of a node's *depth* and its *id* allows it to be uniquely identified within the search-tree.

To track which nodes are currently active, *active_ids*, an array of integers of size J is used as a supporting data structure: it is essentially an array of counters, one per search-tree node depth. Each array element holds the id of the last added node at that depth, and thus represents the *active id* at that depth. In other words, *active_ids* represents the per-level ids that are *currently in use*. For example, Fig. 5.6 shows that the active first-order differences for the current tree path $\Omega = \{0, 2, 5\}$ are 2, 3 and 5: for each of these differences, the corresponding tuple (D, I) in the $difference\_store$ array has an *id* I that is equal to the value stored in *active_ids* for that *depth* D. In particular, the difference value 2 in the $difference\_store$ array was produced by a node at depth D=1 with an id I=2, and the *active_ids* array indicates that a node with id I=2 at depth D=1 is part of the current path being evaluated (i.e. an *active* node).

The *active_ids* array counters are all initialized with a value of 1, corresponding to the ids of the leftmost nodes at each depth (because the value zero is used to indicate a difference that has not yet been encountered). Both arrays, *active_ids* and $difference\_store$ are created once and reused throughout the search. The search-tree is traversed as described in Section 5.2.3. For each new node addition test, the counter value in the *active_ids* array corresponding to that node's depth is incremented by 1, in such a way that it matches the

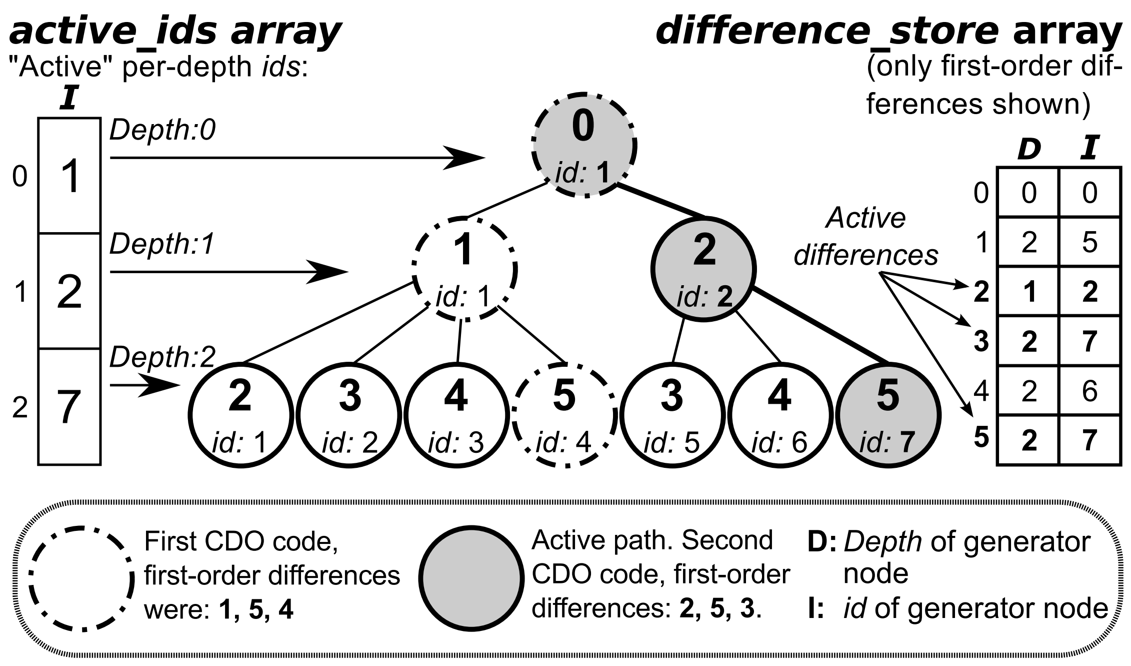


Figure 5.6 Example illustrating how to store the positive differences and detect collisions during a tree-traversal.

node's id. This causes all differences stored in the *difference_store* array and tagged with an equal depth to *expire* implicitly (i.e. they cease to be considered *active* and are effectively ignored).

Figure 5.7 shows the contents of the *difference_store* and *active_ids* arrays for each step in the following example. In Fig. 5.6, after the first valid CDO$_{J=3}$ code $\Omega = \{0, 1, 5\}$ is found, the node value 2 (with id I=2) in the next branch at depth D=1 is explored, and the corresponding id in *active_ids* is incremented from I=1 to I=2 (STEP #1 in Fig. 5.7). The code $\{0, 2\}$ is validated and generates the first-order difference value 2. Next, *active_ids* is incremented at depth D=2 from I=4 to I=5, and a node with value 3 is appended to form $\{0, 2, 3\}$. As this code undergoes validation, it generates first-order differences 3 and 1, which do not generate a collision (STEP #2 in Fig. 5.7), but the node is discarded because of second-order difference collisions (not shown in Figs. 5.6 and 5.7). The next sibling is appended and the corresponding *active_ids* value at depth D=2 is incremented from I=5 to I=6: this *expires* the differences 3 and 1, which are owned by node value 3 (D=2, I=5) such that they can safely be ignored. The next sibling with value 4 (D=2, I=6) is appended,

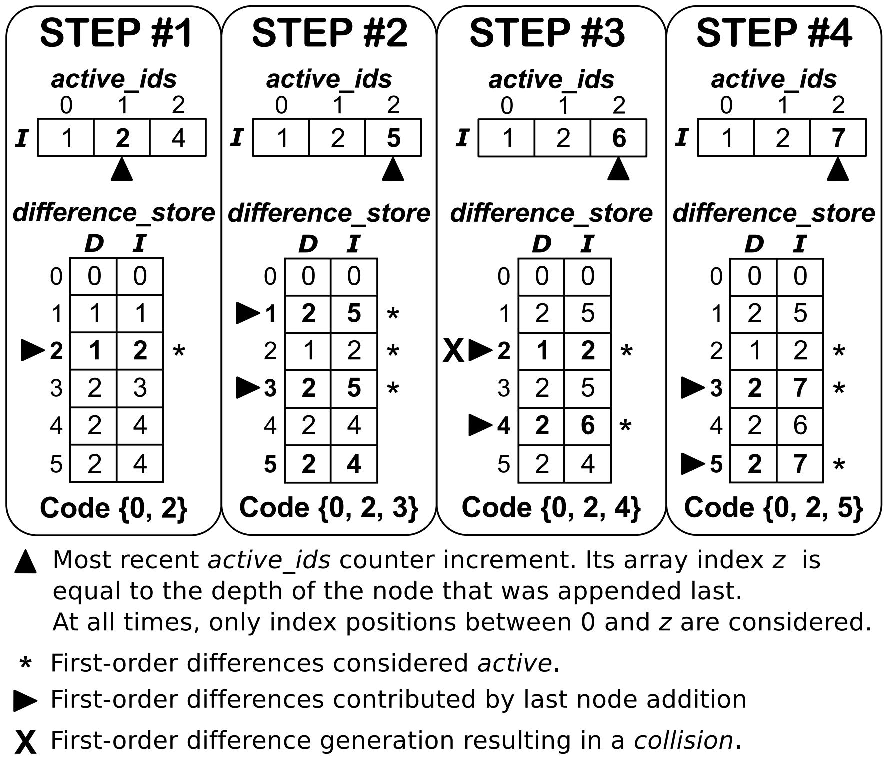

STEP #1
STEP #2
STEP #3
STEP #4
active_ids
difference_store
Code {0, 2}
Code {0, 2, 3}
Code {0, 2, 4}
Code {0, 2, 5}
▲ Most recent active_ids counter increment. Its array index z is equal to the depth of the node that was appended last. At all times, only index positions between 0 and z are considered.
* First-order differences considered active.
▶ First-order differences contributed by last node addition
X First-order difference generation resulting in a collision.

Figure 5.7 Using the *difference_store* array and the *active_ids* array to perform *incremental computation of differences* (only first-order differences shown).

forming $\{0, 2, 4\}$, and generating differences 4 and 2 (STEP #3 in Fig. 5.7).

These create a collision, since 2 in *difference_store* is currently owned by the active node (D=1, I=2). The node 4 is thus discarded, *active_ids* at depth D=2 is incremented from I=6 to I=7, and $\{0, 2, 5\}$ is tested, generating first-order differences 5 and 3 (STEP #4 in Fig. 5.7). Note that the tuple at index 3 in *difference_store* can safely be overwritten (i.e. no collision is encountered), as the corresponding array element is owned by the *inactive* node (D=2, I=5): since this code does not generate any first or second order difference collisions, $\Omega = \{0, 2, 5\}$ is the second valid CDO$_{J=3}$ code in the search.

The more sophisticated *collision detection algorithm* with data reuse support is shown in Fig. 5.8. For each difference value checked, detecting if a collision occurred requires at most three comparisons, and is independent of J: no collision is encountered if either the *difference_store* array location representing the generated positive difference value has just been *reset* (i.e. condition *temp_1*), or, if the location is not owned by any *active* node in the current path being validated. In particular, condition *temp_2* indicates that the difference

```
Require: $node\_diff$, $node\_depth$ are positive integers.
 1: function COLLISION($node\_diff$, $node\_depth$)
 2:     global $difference\_store$, $active\_ids$
 3:     $depth \leftarrow difference\_store[node\_diff].depth$
 4:     $id \leftarrow difference\_store[node\_diff].id$
 5:     $temp\_1 \leftarrow (depth <= 0)$
 6:     $temp\_2 \leftarrow (depth > node\_depth)$
 7:     $temp\_3 \leftarrow (id \neq active\_ids[abs(depth)])$
 8:     if ($temp\_1$ or $temp\_2$ or $temp\_3$) then
 9:         return $False$
10:     else
11:         return $True$
12:     end if
13: end function
```

Figure 5.8 Pseudo-code of the collision-detection algorithm.

was generated by a node located at a greater depth in the tree, thus, given our deterministic tree-traversal, a past node that is not in the current path. Similarly, condition *temp_3* states that the difference was produced by a node with a different id than the *active id* for this depth, and thus it is not an *active* node on the current path. As a consequence, the collision test is done with a *constant* time complexity as a function of J. Note that for first-order S-CDO code differences, a variation of this algorithm must be used, as is described in the next section.

The combination of techniques described above allows us to quickly distinguish the stored differences that are relevant to the code being validated from the ones that are not relevant anymore. Before a new validation, the differences contributed by the previous node can be efficiently discarded by means of a single counter increment and without the need of iterating through the large array. The process is efficient because differences are reused when appropriate, thus allowing for a drastic reduction in the number of differences computed for a node-addition validation.

5.2.4.1.3 Testing the Validity Conditions The data structure presented in Section 5.2.4.1.2 is used for efficiently detecting collisions between differences generated by a sequence of integers. The proposed routine for validation uses a single $difference\_store$ and

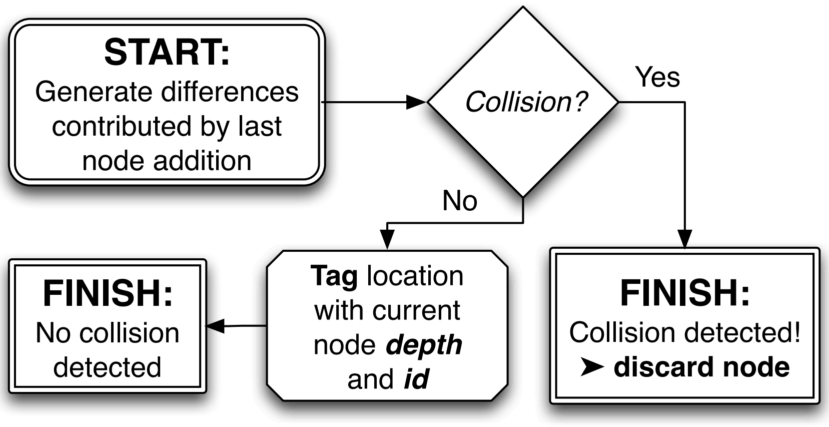

START:
Generate differences contributed by last node addition
Collision?
Yes
No
Tag location with current node depth and id
FINISH: No collision detected
FINISH: Collision detected! ➤ discard node

Figure 5.9 Flowchart for validating a CDO code.

active_ids array combination for both, first and second order differences to further reduce the time required for rejecting an invalid code, as is described below. Please note that we will assume, for this section, that only positive first and second-order differences are tracked and that differences resulting in negative values are automatically discarded.

For CDO codes, all three CDO conditions require that the computed differences be unique in value. The required size for the $difference\_store$ array is $S_{CDO} = 2 * M_{curr} + 1$: it depends on the largest value a second-order difference can have, which itself is dependent on the shortest known span. The validation function starts by testing the first condition, which consists in computing the first-order differences generated by the last node addition and checking if each difference would result in a collision (see Fig. 5.9). If no collision is encountered, the difference is stored in our data structure and the next difference is computed. If a collision is detected, the current node and the differences it produced are immediately discarded and the next node addition is validated. Once all first-order differences have been generated and added to the $difference\_store$ array, the same process is repeated for second-order differences: this tests the second and third conditions at the same time since, for each value, collisions will be checked against the first and second order differences already stored in our array. A code candidate is considered a valid CDO code if none of its first and second order differences generates a collision.

For S-CDO codes, the validation function starts off in the same manner, by computing the first-order differences generated by the last node addition (see Fig. 5.11). However, for each difference, it uses a variation of the collision-detection algorithm shown in Fig. 5.8 to determine whether the difference would result in a collision or not (see Fig. 5.12): instead of using the raw depth value D stored in the $difference\_store$ array, it uses its absolute value. This is done to detect collisions with second-order differences, since, for reasons explained below, their depths are stored in the data structure as negative values. If no collision is encountered, the first-order difference is stored in the $difference\_store$ array and the next difference is computed. If a collision is detected, the current node and the differences it produced are immediately discarded and the next node addition is validated. Once all first-order differences have been generated and added to the data structure, second-order differences are computed.

Since the second S-CDO code condition allows for second-order difference repetitions to occur, collisions between these do not need to be verified or tracked. Nevertheless, the third condition requires that second-order differences be distinct from first-order differences. Therefore, it is necessary to verify that none of the computed second-order differences generates a collision with the first-order differences already stored in our data structure and vice-versa. The regular collision detection algorithm in Fig. 5.8 is used to determine whether or not a second-order difference would result in a collision. If a collision is encountered, the node and the differences it produced are discarded. Otherwise, the second-order difference is stored in the $difference\_store$ array, but with a *negative* depth value (as opposed to the *positive* depth values used for first-order differences), and the next difference computation and collision-testing can proceed. By encoding the second-order difference depths as negative values and using slightly different collision-detection algorithms for testing first and second order differences, we are able to efficiently ensure that all three S-CDO code conditions are properly verified: the collision-detection algorithm in Fig. 5.12 ensures that tested first-order differences are distinct from active first *and second* order differences that were previously stored; the collision-detection algorithm in Fig. 5.8 ensures that tested second-order differences are distinct from active first-order differences that were previously stored, but ignores previously computed second-order differences because the sign of the recorded depth causes the collision detection to fail, thus effectively allowing for second-order difference repetitions to occur.

```
Ensure: node's $depth$ and $id$ are positive integers.
function GEN_DIFFERENCES_INCR($depth$, $id$)
    global $code$
                                          ▷ Generate contributed first-order differences:
    for ($i = 0$; $i < depth$; $i$++) do
        $d \leftarrow code[depth] - code[i]$
        if $collision(d, depth)$ then
            return $False$
        else
            $tag\_difference\_store(depth, id)$
        end if
    end for
                                          ▷ Generate contributed second-order differences:
    for ($j = 0$; $j < depth$; $j$++) do                              ▷ main loop
        $term\_1 \leftarrow code[depth] - code[j]$
        for ($k = 0$; $k < j$; $k$++) do                             ▷ first k loop
            for ($n = 0$; $n < k$; $n$++) do                         ▷ first n loop
                $d \leftarrow term\_1 + (code[k] - code[n])$
                if $collision(d, depth)$ then
                    return $False$
                else if validating CDO codes then
                    $tag\_difference\_store(depth, id)$
                end if
            end for
                                                                     ▷ second n loop
            for ($n = k + 1$; $n \leq j$; $n$++) do
                $d \leftarrow term\_1 + (code[k] - code[n])$
                if $collision(d, depth)$ then
                    return $False$
                else if validating CDO codes then
                    $tag\_difference\_store(depth, id)$
                end if
            end for
        end for                                                      ▷ end first k loop
                                                                     ▷ second k loop
        for ($k = j + 1$; $k \leq depth$; $k$++) do
            for ($n = 0$; $n \leq j$; $n$++) do
                $d \leftarrow term\_1 + (code[k] - code[n])$
                if $collision(d, depth)$ then
                    return $False$
                else if validating CDO codes then
                    $tag\_difference\_store(depth, id)$
                end if
            end for
        end for                                                      ▷ end second k loop
    end for                                                          ▷ end main loop
    return $True$
end function
```

Figure 5.10 Pseudo-code of novel incremental first-order and second-order difference generation, CDO code collision test, and tagging of values in the *difference_store* array. Returns *True* if code is valid, *False* otherwise.

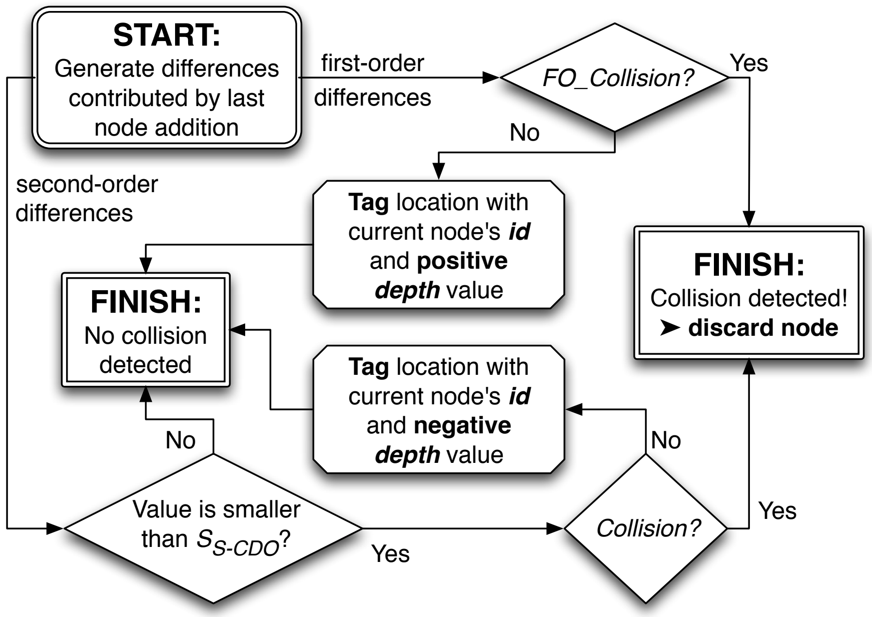

START:
Generate differences contributed by last node addition
first-order differences
FO_Collision?
Yes
No
Tag location with current node's id and positive depth value
second-order differences
FINISH:
No collision detected
FINISH:
Collision detected!
➤ discard node
Tag location with current node's id and negative depth value
No
No
Yes
Value is smaller than $S_{S\text{-}CDO}$?
Collision?
Yes

Figure 5.11 Flowchart for validating a S-CDO code.

Finally, for S-CDO codes, only second-order differences that have a value *equal to* or *lower* than the maximum possible first-order difference value can generate a collision with first-order differences: second-order differences greater than this value do not need to be tested, since the third S-CDO code condition does not apply to them. Therefore, we can define S_{S-CDO}, the size of the $difference\_store$ array, as $S_{S-CDO} = M_{curr} + 1$, since M_{curr}, the shortest known span, is the largest value a positive first-order difference can have. A code candidate is considered a valid S-CDO code if no collision is encountered during the difference generation. Note that for S-CDO codes, if a maximum for an *active_ids* counter is reached, it is reset to 1, and all *tuples* in the $difference\_store$ array with an *absolute value* of D equal to that counter's depth are *reset* to $(0, 0)$.

We have assumed that a known span exists for the code order being searched for. However, if such a value were not known or available, a very large value may be temporarily used: a first run of the search would allow us to find a few codes whose span may then be used in later runs. Alternatively, another search algorithm may be used to obtain this initial span value.

```
Require: $node\_diff$ and $node\_depth$ are positive integers.
function FO_COLLISION($node\_diff$, $node\_depth$)
    global $difference\_store$, $active\_ids$
    $depth \leftarrow abs(difference\_store[node\_diff].depth)$
    $id \leftarrow difference\_store[node\_diff].id$
    $temp\_1 \leftarrow (depth == 0)$
    $temp\_2 \leftarrow (depth > node\_depth)$
    $temp\_3 \leftarrow (id \neq active\_ids[depth])$
    if ($temp\_1$ or $temp\_2$ or $temp\_3$) then
        return $False$
    else
        return $True$
    end if
end function
```

Figure 5.12 Pseudo-code of modified collision-detection algorithm for S-CDO code first-order differences.

5.2.4.1.4 Efficient Incremental Computation of Differences and Data Reuse

The proposed validation algorithm will only represent and save positive first and second order differences in the data structure: this decreases the array size by half and improves the data structure's *spatial locality*, thus also reducing the *cache miss-rate* [80]. Furthermore, since first and second order differences exist in pairs of equal magnitude but opposite sign, considering only one of the difference pair elements eliminates the ambiguity that may result if two equal pairs are generated with their pair elements interleaved in sign and order.

Rather than generating all of the second-order differences and discarding the *negative* ones [5, 20], only one of the difference pair elements (irrespective of its sign) is computed, and then its absolute value is evaluated. This results in a simpler second-order difference generation routine than generating *positive* second-order differences.

The novel first and second order difference generation algorithm is shown in Fig. 5.10: it is based on a simple set of smaller *for-loops* and improves on previous techniques by directly computing only *half* of the total first and second order differences, and only those *contributed* by the last node addition (see Section 5.2.4.1.2).

Figure 5.13 shows the differences computed when validating a sample CDO code: differences highlighted in grey are not computed by the more efficient proposed difference generation. Since only differences contributed by the last added node are generated, for $\Omega = \{0, 1, 5\}$,

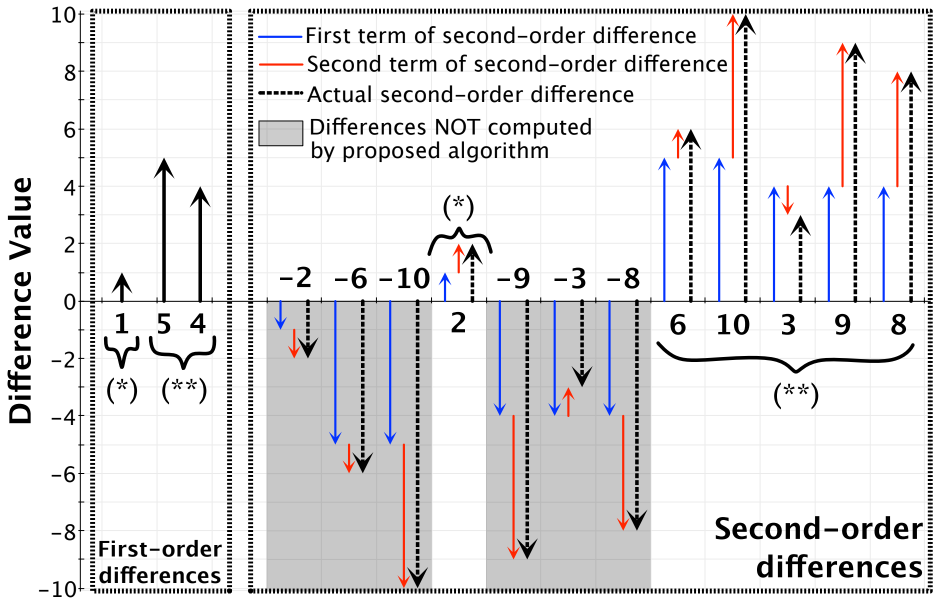


Figure 5.13 First and second order differences computed when validating the CDO code $\Omega = \{0, 1, 5\}$: (*) denotes the differences added to the set by $\alpha_2 = 1$, and (**) denotes the differences added by $\alpha_3 = 5$.

the generation routine is called once for $\alpha_2 = 1$ and once for $\alpha_3 = 5$. Finally, since the new *for-loops* only depend on the *depth* of the node that is producing the contributed differences, they were fully unrolled prior to compile time.

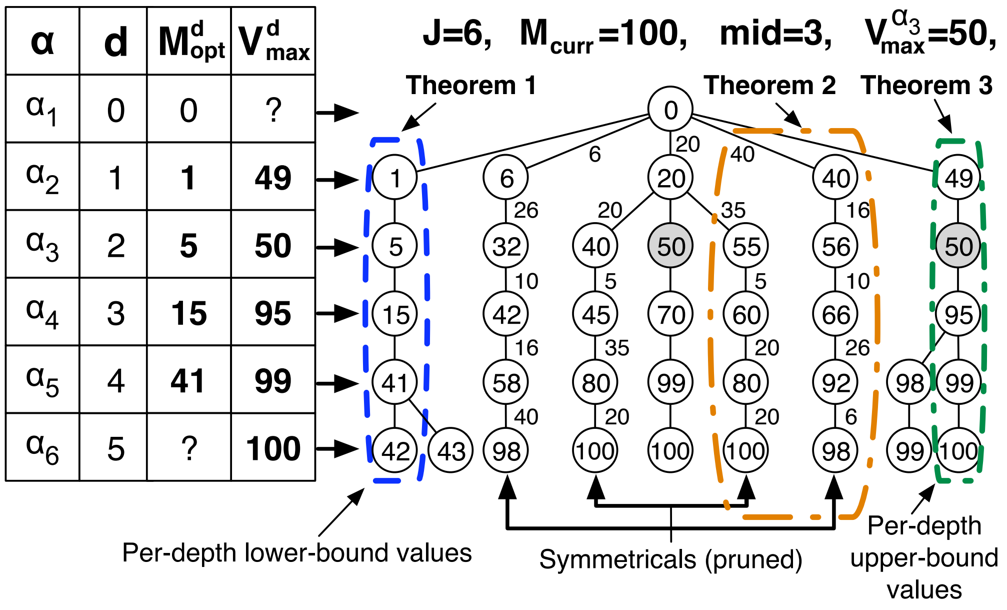
The number of positive first ($N^J_{S,incr}$) and second ($N^J_{D,incr}$) order differences generated by the proposed incremental difference computation are expressed by [14]:

$$N^J_{S,incr} = J - 1 \tag{5.7}$$

$$N^J_{D,incr} = \frac{J^3 - 3J^2 + 4J - 2}{2} \tag{5.8}$$

Although still represented by polynomial equations, their degree has been effectively decreased by 1 when compared to the earlier expressions (5.3) and (5.4), thus allowing for significant computational savings, which become even larger as J increases. For example, when validating a code of order $J = 17$, the proposed *incremental computation with data reuse* method allows for a 4.5x reduction in the total number of computed first and second order differences that are required to validate *each code.*

We now describe the parallel tree-traversal improvements and load balancing.



α	d	M^d_{opt}	V^d_{max}
α_1	0	0	?
α_2	1	**1**	**49**
α_3	2	**5**	**50**
α_4	3	**15**	**95**
α_5	4	**41**	**99**
α_6	5	?	**100**

J=6, M_curr=100, mid=3, V^α3_max=50,
Theorem 1
Theorem 2
Theorem 3
Per-depth lower-bound values
Symmetricals (pruned)
Per-depth upper-bound values

Figure 5.14 Summary of the most important tree-pruning techniques used: Theorems 1, 2 and 3 allow for a considerable reduction in the size of the search space [14].

5.2.4.2 Parallel Dynamic Search-space Reduction - an Implicitly-exhaustive Search

The following sections briefly explain the proposed algorithm's tree-traversal improvements and the parallel searching techniques developed to efficiently use the performance offered by modern multi-core systems.

5.2.4.2.1 Tree-Traversal Improvements The *efficient parallel implicitly-exhaustive* search algorithm reduces the computational time required for searching for optimal-span (S-)CDO codes by using more aggressive search-tree pruning techniques in order to further reduce the number of branches that are explored.

Three significant search-tree pruning enhancements over the reference exhaustive-search algorithm [5] are presented in [14, 15] and are employed to obtain a considerable reduction in the size of the search space (see Fig. 5.14). In order to keep this paper self-contained, and since the three theorems are used to speed up the proposed implementation, they are repeated below. The proofs for these theorems are available in [14, 15].

Theorem 5.1 defines a higher lower-bound node value for each search-tree depth, as explained below.

Theorem 5.1. *Let V_{min}^{d} be a lower bound for node values having $d < J - 1$. Then, V_{min}^{d} can be defined as the largest of the following three values:* the optimal-span corresponding to the node's depth, M_{opt}^{d}, if it is known; the lower bound calculated as per (5.5) (for CDO codes only); the node value of the parent node plus one.

For example, in Fig. 5.14, the nodes at depth $d = 4$ have a lower bound of $\alpha_5 = 41$.

Theorem 5.2 exploits the symmetry property of (S-)CDO codes mentioned in Section 5.2.2 to discard some additional codes whose symmetrical has already been encountered during the tree exploration. Indeed, as stated below, it complements Theorem 5.3 by further limiting the maximum value of the α_{mid} nodes, where $mid = \left\lfloor \frac{J-1}{2} \right\rfloor + 1$.

Theorem 5.2. *Let $V_{max}^{\alpha_{mid}}$ be defined as:*

$$V_{max}^{\alpha_{mid}} = \left\lceil \frac{M_{curr} + 1}{2} \right\rceil - 1 \tag{5.9}$$

where $mid = \left\lfloor \frac{J-1}{2} \right\rfloor + 1$. Then, any code with an α_{mid} node having a value larger than $V_{max}^{\alpha_{mid}}$ would have a symmetrical *equivalent within the codes in the search-tree having $\alpha_{mid} \leq V_{max}^{\alpha_{mid}}$.*

For example, in Fig. 5.14, $mid = 3$ and thus nodes at depth $d = 2$ have an upper bound value of $V_{max}^{\alpha_3} = 50$. Therefore, any code with an α_3 node value larger than $V_{max}^{\alpha_3}$ is a symmetrical of a code encountered earlier in the search and can be discarded. Note that having this upper-bound value for α_3 nodes also implies having a lower upper-bound value for nodes that are closer to the root node, since these must have a smaller value than α_3. Theorem 5.2 is equivalent to the *Midpoint Reduction* technique used in the search for optimal Golomb rulers, and thus a search-space reduction of approximately 50% can also be achieved [17].

Theorem 5.3 defines a lower upper-bound node value for each search-tree depth, as explained below.

Theorem 5.3. *Let V_{max}^{d} be an upper bound for node values having $d < J - 1$. Then, V_{max}^{d} can be defined as the smallest of the following four values:*

- *$M_{curr} - M_{opt}^{J-d-1}$, where M_{curr} is the current shortest known span for a code of order J, and M_{opt}^{J-d-1} is the optimal-span of a code of order $J - d$, if it is known;*
- *$V_{max}^{\alpha_{mid}}$ as per Theorem 5.2 (for nodes at $d = mid - 1$ only);*
- *$M_{curr} - \alpha_J^*$, where M_{curr} is the current shortest known span and α_J^* is the lower bound calculated as per (5.5) (for CDO codes only);*
- *$V_{max}^{d+1} - 1$ for $d < J - 1$, where V_{max}^{d+1} is the V_{max}^{d} value for children nodes of the current node, and the current node is not a leaf-node.*

For example, in Fig. 5.14, nodes at depth $d = 3$ have an upper bound of $\alpha_4 = M_{curr} - M_{opt}^{d=2} = 95$.

5.2.4.2.2 Parallel Search and Load-Balancing With multi-core computer systems becoming a commodity, it is important to have an algorithm that scales well and harnesses the computational power they offer by parallelizing the processing of data. This observation led to the development of the efficient parallel and implicitly-exhaustive search algorithm used for finding the new (S-)CDO codes presented in [14], and whose high-performance implementation and algorithmic improvements are detailed in this paper.

Conceptually, the search-tree is divided into a set of sub-trees that are explored in parallel. Each sub-tree exploration corresponds to a *task* that needs to be completed. The proposed implicitly-exhaustive search algorithm's implementation instantiates a pool of *Linux POSIX Threads* (or *pthreads*) to perform a deterministic parallel search-tree traversal that fully exploits the available computing cores. Indeed, each thread is assigned an independent sub-tree to work on (see Fig. 5.15). By partitioning the search space into independent sub-trees that are only accessed by one thread at a time, threads can do most of the processing within their private workspace, foregoing the need of complex resource-sharing mechanisms and thus reducing the overall synchronization overhead.

The pool of threads executes a *cooperative search* by using a mutex-protected [81] shared workspace: when a thread discovers a valid code with a shorter span than the current shortest span known, its span value is shared with all other threads to collectively apply all known tree-pruning techniques to the current and future jobs being processed. Sharing this information

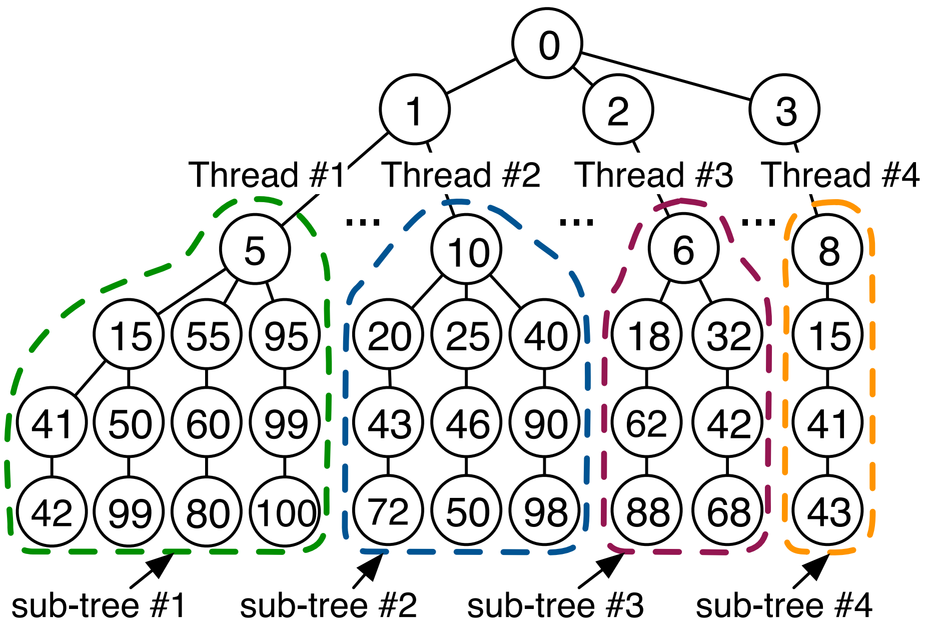


Figure 5.15 Task partitioning and load-balancing.

instantly benefits all threads, thus significantly decreasing the overall computation time by allowing the search space to converge to a smaller search-tree in less time, potentially even offering a better-than-linear performance with respect to the parallelism employed [19]. When a thread has finished processing a job, another job is assigned to it until no more sub-trees are available, at which point the exploration of the search-tree is complete.

Because earlier branches in the search-tree will carry more nodes than later branches (see (5.6)), having a static mapping of sub-tree base-nodes to α_2 nodes [19] would result in the first tasks taking exponentially longer to complete, thus becoming the bottleneck of the parallel search (Amdahl's law [82]). Instead, we employ a load-balancing technique consisting in varying the depth to which sub-tree base-nodes are mapped to. In fact, sub-tree base-nodes may be mapped to any node having depth $d \in [1; J-2]$: an increase in this depth reduces the size of their corresponding sub-tree, at the expense of increasing the total number of tasks to compute. For example, in Fig. 5.15, one can easily see that mapping sub-tree base-nodes to α_3 nodes reduces the size of each sub-tree but increases the number of sub-trees that need to be explored. Since the gap between the longest running task and the shorter running tasks is reduced, threads can operate concurrently for a longer period of time, thus enabling load-balancing of work. The depth to which sub-tree base-nodes are mapped to is chosen

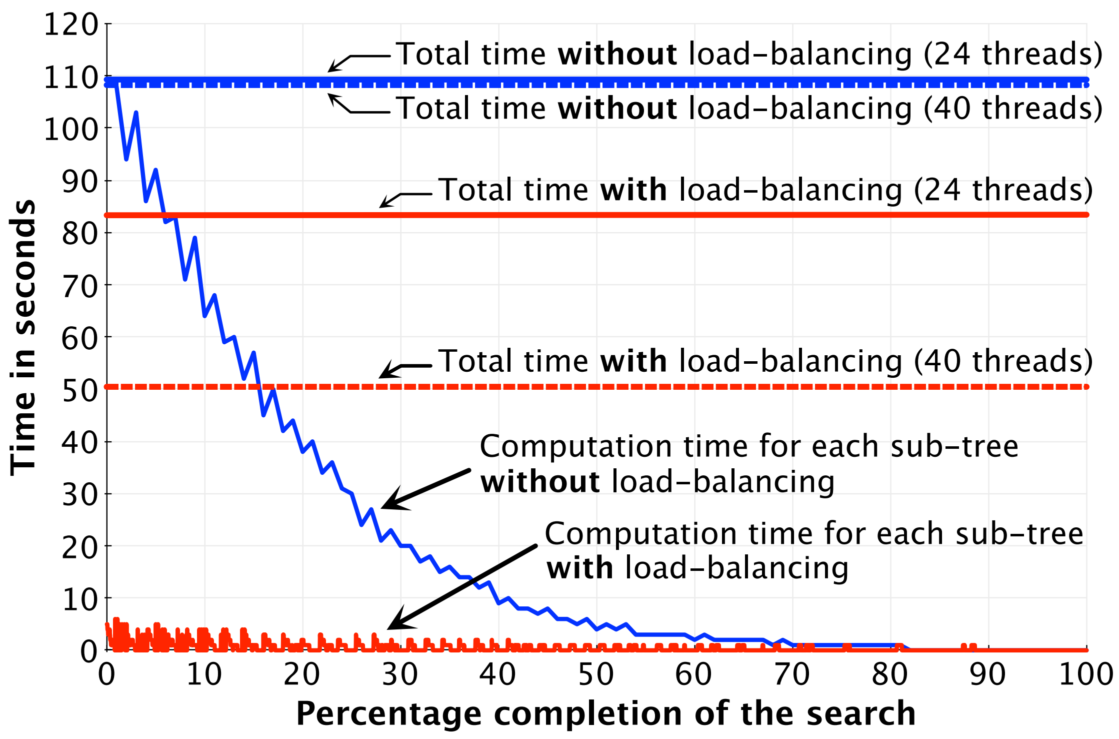


Figure 5.16 Computation time for each sub-tree *with* and *without* load-balancing ($J = 9$, S-CDO codes).

by profiling the average time required for completing a sub-tree traversal and ensuring that it remains small, in the order of a few minutes. Empirical tests show that best results are obtained with an average task-completion time of less than 10 minutes. Figure 5.16 shows the computation time required for each sub-tree ($J = 9$, S-CDO codes) when using a pool of 24 threads and a pool of 40 threads. As can be seen, without load-balancing, i.e. with a static mapping of sub-tree base-nodes to α_2 nodes, the first sub-tree computation time is almost entirely responsible for the total computation time of the search, which is thus not significantly reduced when the number of threads is increased from 24 to 40. On the other hand, with load-balancing enabled, the gap between the longest and shortest running task is much smaller: not only is the total computation time improved when using 24 threads, but increasing the number of threads to 40 further reduces the total computation time as the algorithm is able to better exploit the additional parallelism offered by the system. Finally, note that there is a single node per search-tree depth between the root node and the base-node of each sub-tree, and that their respective values do not change during the processing of a job: in fact, this unique node-value sequence is used as the job's id, and is employed for tracking *completed*, *active* and *pending* jobs to be processed.

The speedup achieved with the proposed algorithm and its implementation is presented in the next section.

5.2.5 Results

The test systems employed to produce the reported results were configured to use a *Scientific Linux 6.2* distribution, running on a *64-bit GNU/Linux 2.6.32 kernel*. All source code was compiled using GCC-4.4.6 with the -O3 flag enabled. Since the original C/C++ source code for the previous algorithms was available, it was used with little-to-no modification for comparison with the novel algorithm's implementation.

In this section, we will show the drastic performance improvements achieved through the use of the proposed algorithm and its implementation. The breakthrough in *code validation speed*, *multi-core/multithreaded scaling*, *search-space/complexity reduction* and *overall speedup* will be shown for CDO and S-CDO codes.

5.2.5.1 (S-)CDO Code Validation Speed

The proposed validation function *eie_validate()* is compared with *ie_validate()*, the (S-)CDO code validation function used in the reference exhaustive-search algorithm [5], as well as with *prs_validate()*, the higher-performance CDO-code-only validation function used in the pseudo-random search algorithm presented in [20].

In order to measure the speed of the three validation functions, a supporting framework for exploring the search-tree in a similar way as the reference exhaustive-search algorithm was devised. This ensured that the three validation functions were provided with the same realistic set of codes, with an order varying from 2 to J, as they would have had to validate in the exhaustive-search tree-traversal described above.

Twenty samples of five hundred thousand validations were taken, and the cumulative average number of *validations per second* was recorded. For a $J = 6$ S-CDO code search, there weren't enough codes to sample twenty times / five hundred thousand codes, so sampling was done for fifty thousand codes instead. The performance of the validation functions was

Table 5.1 CDO validation function comparison - *ie_validate()* [5]

J	*ie_validate()* (validations/sec)	*eie_validate()* (validations/sec)	Speedup Factor
6	53856.00	2.78721E+07	518
8	6286.36	2.18509E+07	3476
10	1233.19	1.37055E+07	11114
12	254.68	9.49252E+06	37272
14	66.73	6.05981E+06	90809
16	21.90	3.87740E+06	177055
17	13.30	2.57027E+06	193261

Table 5.2 CDO validation function comparison - *prs_validate()* [20]

J	*prs_validate()* (validations/sec)	*eie_validate()* (validations/sec)	Speedup Factor
6	803523.00	2.78721E+07	35
8	241968.00	2.18509E+07	90
10	69677.90	1.37055E+07	197
12	21203.20	9.49252E+06	448
14	7110.04	6.05981E+06	852
16	2542.93	3.87740E+06	1525
17	1243.60	2.57027E+06	2067

measured for CDO and S-CDO codes, with orders $J \in [6; 17]$. The test system used was equipped with an *Intel Core i7-960* Central Processing Unit (CPU) clocked at 3.2GHz, with 8MB of cache and 12GB of RAM. The reported values are all *single-threaded validation speeds* (i.e. using a single CPU core), thus ensuring a fair comparison between validation functions.

Tables 5.1 and 5.2 show that for CDO codes, the novel validation function offers an impressive speedup factor ranging from 518 to 193261 when compared to the reference exhaustive search validation function [5], and from 35 to 2067 when compared to the fastest known CDO validation function [20]. Likewise, Table 5.3 (S-CDO codes) shows that the novel validation function offers a very significant speedup factor over the validation function in [5], ranging

Table 5.3 S-CDO validation function comparison - *ie_validate()* [5]

J	*ie_validate()* (validations/sec)	*eie_validate()* (validations/sec)	Speedup Factor
6 $*$	66071.50	1.75168E+07	265
8	8432.96	1.22286E+07	1450
10	1808.25	7.36185E+06	4071
12	487.17	4.52461E+06	9288
14	111.32	2.61650E+06	23504
16	29.04	1.35555E+06	46673
17	16.50	1.00354E+06	60832

$*$ average over 20 samples of $5 * 10^4$ validated codes only

from 265 to 60832. Note that due to space constraints, results for most odd values of J are not included.

For CDO and S-CDO validations, the speedup also becomes greater as J (and thus the complexity) increases, since the novel validation function has fewer differences to compute and comparisons to make in order to apply the validation test to a potential (S-)CDO code.

5.2.5.2 Multi-core / Multithreaded Scaling

In order to test the scalability of the proposed parallel algorithm, a system consisting of two *Intel QuickPath Interconnect* enabled server blades was used. Each blade is equipped with 40 Hyper-Threading Technology enabled CPU cores (4x 10-core *Intel Xeon E7-8870* CPUs clocked at 2.4GHz, with 30MB of cache) and 512GB of RAM.

The amount of parallelism used was gradually increased from 1 thread to 160 threads (since the system has 80 real cores and 80 virtual cores), and the speedup compared to a single-threaded operation was recorded.

Figure 5.17 compares the overall speedup obtained when increasing the number of scouting threads from 1 to 160. Two curves can be observed: the solid line represents an ideal (linear) scaling with the number of threads; the dashed line represents the actual scaling obtained, with the circles illustrating our actual sample points. As can be seen, when increasing the

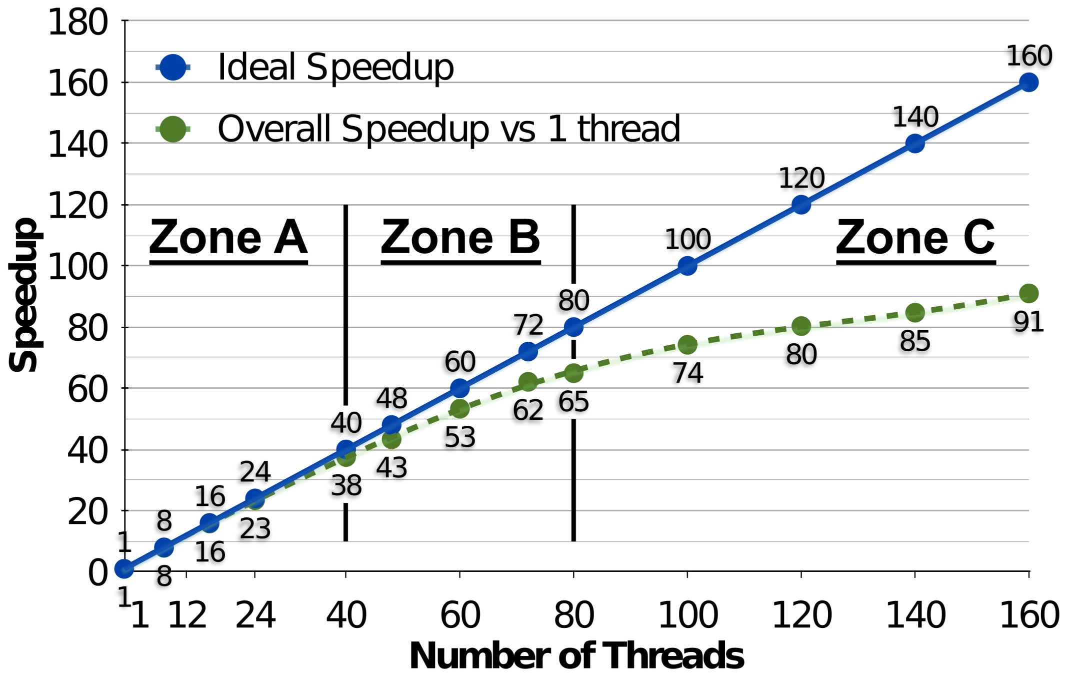


Figure 5.17 Scaling of the novel algorithm as a function of the number of threads used (S-CDO codes, $J = 9$).

number of threads from 1 to 40 (Zone A), the scaling is very close to being ideal. As the second blade starts being used (Zone B - number of threads ranging from 40 to 80), due to the communication overhead between blades, the distance between the obtained and the ideal scaling becomes wider. Finally, as the number of threads increases from 80 to 160 (Zone C), virtual cores are used, further distancing the obtained speedup from the ideal scaling. Nevertheless, employing virtual cores on this system is still advantageous as it allows us to improve the speedup factor from 65 (for 80 real cores) to 91 (for 80 real + 80 virtual cores).

5.2.5.3 Overall speedup

The overall speedup, when using the proposed algorithm instead of the reference algorithm, was recorded on a test system with an *Intel Core i7-2600* CPU clocked at 3.4GHz, with 8MB of cache and 16GB of RAM. Whereas the reference algorithm executed a serial (single-threaded) search [5], the novel algorithm performed a parallel search using 8 threads (for 4 real cores, and 4 virtual cores).

The search was conducted for (S-)CDO codes and $J \in \{6, 7, 8\}$. The *CPU Time* (*CT*)

Table 5.4 CDO Code Exhaustive Search *Overall Speedup*

J	CPU Time Reference Algo. (in seconds)	CPU Time Proposed Algo. (in seconds)	Overall Speedup Factor
6	102.05	0.02	6280
7	73596.20	4.49	16377
8	*** ‡	6848.34	N/A

‡ computation time took too long to complete (over 2 weeks)

Table 5.5 S-CDO Code Exhaustive Search *Overall Speedup*

J	CPU Time Reference Algo. (in seconds)	CPU Time Proposed Algo. (in seconds)	Overall Speedup Factor
6	1.02	*** †	N/A
7	205.55	0.04	5873
8	23358.62	3.70	6322

† the value was below our measurement resolution

was measured for each implementation and J value. The *Overall Speedup* was calculated as the ratio of the *CPU Time* required for the reference implementation to complete the search, over the one required for the proposed algorithm to complete the same task.

Tables 5.4 and 5.5, for CDO codes and S-CDO codes respectively, show that the proposed algorithm offers a dramatic speedup over the previous reference algorithm, ranging from *three* to *four* orders of magnitude. It can also be observed that this speedup increases as J increases.

5.2.5.4 (S-)CDO Code Span Improvements Obtained

Figure 5.18 shows the total number of leaf-nodes in the search-tree as a function of the shortest known (S-)CDO code spans and their order J. When searching for optimal-span (S-)CDO codes, the number of leaf-nodes is indicative of the problem size. The highlighted shaded area in Fig. 5.18 represents the search-space that was conquered using the novel search algorithm and its very efficient implementation. Using these techniques, it was possible to

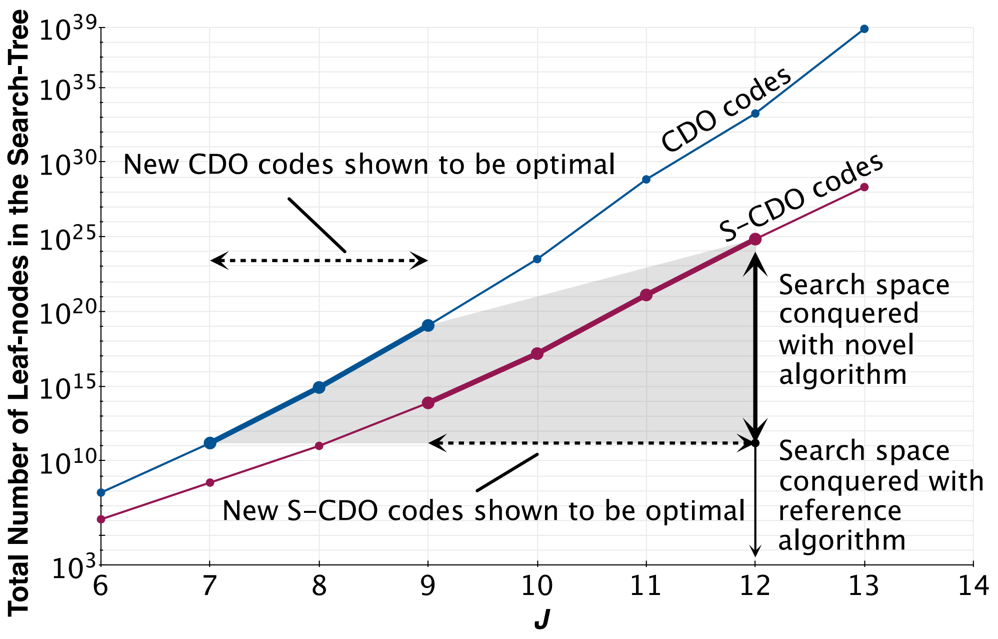


Figure 5.18 Total number of leaf-nodes in the search-tree as a function of the best known (S-)CDO code spans and their order J.

complete an exhaustive search over a search space that is 10^{14} time larger than what was previously possible.

We were able to find and/or verify optimal-span codes for $J \in \{7, 8, 9\}$ (CDO codes) and $J \in \{9, 10, 11, 12\}$ (S-CDO codes) [14], and for larger values of J, to obtain codes with spans that are between 9.12% and 33.89% shorter than the ones published in [15, 20, 21, 28], with an average span improvements of 14.43% for CDO codes and 26.25% for S-CDO codes.

5.2.6 Conclusions

This paper presented a parallel, high-performance, efficient and implicitly-exhaustive search algorithm implementation for finding CDO and S-CDO codes with short spans, offering a drastic speedup over previous exhaustive-search and pseudo-random search algorithms. The speedup is achieved through the use of a vastly improved code validation function that employs a novel data structure for enabling *data-reuse* and *incremental computation of differences*, and a parallel dynamic search-space reduction technique that substantially reduces the size of the search-space without compromising the exhaustive nature of the search.

When searching for optimal (shortest span) J=7 CDO codes and J=8 S-CDO codes, a remarkable speedup factor over the reference algorithm was observed (>16300 and >6300 respectively). Furthermore, the algorithm scales well as the number of microprocessors used increases. The proposed validation function exhibits an impressive speedup factor of >190000 (CDO code, J=17) and >60000 (S-CDO code, J=17) over the reference implicitly-exhaustive search algorithm, and >2000 (J=17) over the fastest published CDO code validation function used in high-performance pseudo-random search algorithms. In addition to being able to yield codes with shorter spans than previously possible with the best pseudo-random search algorithms then available, the proposed algorithm remains exhaustive in nature and is able to process a search space that is some 10^{14} times larger than the largest search space that could be viably explored by earlier exhaustive algorithms. As a result, we were able to find and/or verify optimal-span codes for $J \in \{7, 8, 9\}$ (CDO codes) and $J \in \{9, 10, 11, 12\}$ (S-CDO codes). Finally, for code orders of $J \in [9; 20]$, we have been able to obtain an average span reduction of 14.43% for CDO codes, and 26.25% for S-CDO codes.

5.3 Notes on the novel data structure

As described in Section 5.2.4, the novel validation function uses a form of *Look-Up Table* (LUT) to store the difference values along with a *Unique Identifier* (UID) that associates them with the particular node addition that generated them. The UID, (d, id), is a tuple composed of the node's *depth* and an *identifier* integer value: it is used to differentiate the relevant differences from the differences that can be ignored, thus enabling data-reuse and incremental computations. A set of simple counters act as per-depth *identifier generators*: there is one counter for each node depth in the search-tree, and each counter is initialized to a value of 1. The number of nodes in the search-tree is very large, and as a consequence, the identifier generator counters may overflow after reaching the maximal value represented by their data type. To preserve the "uniqueness" of the UIDs, they are checked after each increment: if a maximum value for the identifier generator associated to a particular node depth d is reached, it is *reset* to 1, and all *tuples* in the LUT corresponding to that depth are *reset* to $(0, 0)$. Subsequent differences contributed by a node at depth d will use the updated identifier value, now equal to 1, and the corresponding tuple will then be stored in the data structure as per Section 5.2.4.

The use of a separate counter for each search-tree depth greatly reduces the total number of *reset* operations that occur, thereby ensuring a low-latency and implicit difference-discarding operation for most cases: the array is only scanned when a counter reaches its maximal value, thus triggering a selective clearing of the array for *tuples* marked with that specific depth. Furthermore, the amount of memory required for storing this data structure is small, when compared to today's microprocessor cache sizes, and its use is well worth the advantages offered during the proposed deterministic search-tree traversal: first, a *latency* reduction for invalidating non-qualifying codes, and second, by reusing the differences that do not change when going from one search-tree branch to the next, a drastic reduction in the number of differences that need to be computed for a node-addition validation.

5.4 Notes on the proposed first and second order difference generation

5.4.1 Reducing the overhead due to branch tests

The novel second-order difference generation described in Section 5.2.4 eliminates the five *if-statements* found in the traditional embedded *for-loops* [4, 20]. Furthermore, in order to eliminate most of the branch tests present in the proposed validation function, a compile-time pre-processing script is used to generate a set of *depth-specific* functions for generating first and second order differences: for each, the depth is *constant* and *known*, therefore allowing for the their embedded *for-loops* to be fully unrolled (see Fig. 5.10).

The resulting loop-free and depth-specific functions are accessed at runtime using function pointers, and can more easily be optimized by modern compilers. Furthermore, by decreasing the number of branching statements, the performance penalty associated with branch mispredictions on modern microprocessors is also reduced: preliminary results show that the depth-specific difference generation using fully unrolled loops offers a *total computation time* reduction of approximately 53%.

5.4.2 Postponing the computation of δ during the search for S-CDO codes

When searching for optimal-span S-CDO codes, several approaches for dealing with the simplification coefficient δ are available: one could search for the shortest-span codes for a specific

δ value; or one could use a “binning” technique, and store the K shortest-span codes for each of the one or more “bins” representing a δ interval of values; or, one could search for the K shortest-span codes, irrespective of their δ value.

The main advantage of using S-CDO codes instead of CDO codes is that the former offer a substantially reduced span (and thus much shorter latency in the decoding system). Therefore, we focus on obtaining the S-CDO codes with the shortest possible spans by allowing any repetition within the set of second-order differences to occur. As a side-effect, the computational overhead is reduced, since δ does not need to be evaluated for each valid code. The δ values are thus independently computed at a later time for the resulting codes of interest.

Note that although the value of the simplification coefficient δ is not taken into consideration during the S-CDO code search, the algorithm will store the last K obtained codes in memory: this allows for a later comparison of the error-correcting performance of codes having a similar span but different δ value.

5.4.3 Eliminating one element in the second-order difference pair

In this chapter, we define a “*collision*” as a difference value repetition. For a non-qualifying code, we are especially interested in identifying collisions as early as possible during the validation process to reduce the number of wasteful computations.

The novel validation function described in Section 5.2.4 improves on previous techniques by directly computing only *half* of the total first and second order differences. This leads to a significant computational speedup: during a code validation, rather than generating all second-order differences and discarding the *negative* ones [5], the proposed algorithm only computes one of the difference pair elements (irrespective of its sign), and then takes its absolute value (see Fig. 5.10). This results in a simpler second-order difference generation routine than what would otherwise be possible by directly generating the *positive* second-order differences, and reduces the number of difference elements that need to be computed before a collision is detected. Furthermore, computing only one element of the first and second order difference pairs also eliminates a possible delay in detecting a collision: indeed, as differences are produced, pair-elements of opposite sign but pertaining to two identical difference pairs may appear as a difference pair of their own, thus delaying the collision

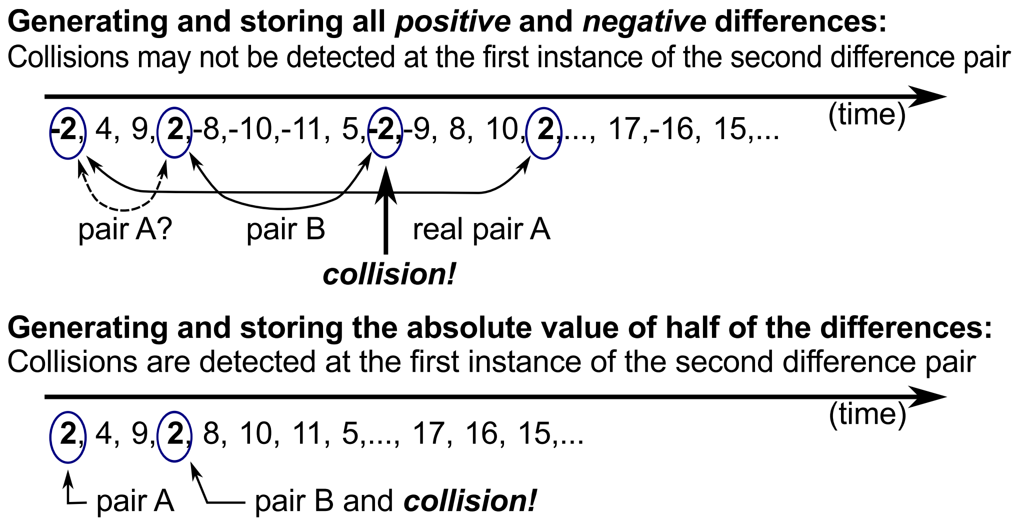

Generating and storing all *positive* and *negative* differences:
Collisions may not be detected at the first instance of the second difference pair
(time)
-2, 4, 9, 2, -8, -10, -11, 5, -2, -9, 8, 10, 2, ..., 17, -16, 15, ...
pair A?
pair B
real pair A
collision!
Generating and storing the absolute value of half of the differences:
Collisions are detected at the first instance of the second difference pair
(time)
2, 4, 9, 2, 8, 10, 11, 5, ..., 17, 16, 15, ...
pair A
pair B and *collision!*

Figure 5.19 First and second order differences exist in pairs of equal magnitude but opposite sign: computing only one of the difference pairs eliminates the ambiguity that would otherwise delay the detection of a collision.

detection until a third element from the difference pairs is encountered.

Figure 5.19 depicts such a situation, when generating a stream of differences that includes two equal pairs of $\{-2, 2\}$ difference values. In the first stream of computed differences, both positive and negative differences are generated and stored. The first value is '-2', followed by '4' and then '9'. When the value '2' is generated, it is not possible to determine whether it completes a pair with the previously computed '-2' value (i.e. "*pair A?*"), or if it is part of a new $\{-2, 2\}$ pair (i.e. "*pair B*"), which would result in a collision. In this example, it is part of a new pair, but since the ambiguity exists, the collision detection has to be delayed, and "wasteful" differences have to be computed and stored. Only when the next '-2' value is generated can the existence of the two equal pairs be confirmed. In the second stream of computed differences, only half of the difference pairs are computed, and their absolute value is considered: the ambiguity is eliminated and a collision can be detected at the first and only instance of the second difference pair. The proposed validation algorithm will thus only represent and save *positive* first and second order differences in the data structure: this decreases the size of the LUT by half and improves the data structure's *spatial locality*, thus also reducing the *cache miss-rate* [80].

Table 5.6 Comparison of the total number of differences computed for a CDO code search and $J \in \{5, 6, 7, 8\}$

J	Nb. of differences (Ref. Algorithm)	Nb. of differences (Novel Algorithm)	Computational Improvement Rate
5	4,194,900	365,173	11.49
6	1,930,920,996	99,818,539	19.34
7	897,371,638,134	43,859,726,611	20.46
8	798,134,489,599,614	74,487,064,702,488	10.72

5.5 (S-)CDO Computational Improvement Rate

As described in Chapter 4 and Section 5.2.4 below, the proposed search algorithm uses tree-pruning techniques to discard nodes whose addition cannot yield a valid code having an improved span. Indeed, children nodes of nodes that have been discarded do not need to be traversed or validated, thus resulting in a significant decrease in the number of examined nodes, especially leaf-nodes, that will have to be added and then validated for (S-)CDO code compliance. Since these leaf-node additions/validations have the highest computational cost, the total search time will be substantially reduced. Furthermore, reducing the number of search-tree branches explored allows for a faster tree-traversal, thus increasing the likelihood of finding new valid codes with shorter spans, and eventually optimal-span codes. As improved spans are found, the bounds used for the tree-pruning are updated (see Section 5.2.4.2.1), thereby allowing for a faster convergence to a search-tree of smaller size.

In order to obtain an estimate of the reduction in the number of search-tree branches explored, and thus an estimate of the reduction in the number of differences computed, counters were placed at each search-tree depth and were incremented whenever a node at that depth underwent a validation. The test system used was equipped with an *Intel Core i7-2600* CPU clocked at 3.4GHz, with 8MB of cache and 16GB of RAM.

The search was performed with the reference [5] and the proposed exhaustive-search tree-traversal algorithms using small values of J for which the optimal span M_{opt}^J is known. An initial span value of $M_{curr}^J = 150\% * M_{opt}^J$ was chosen as the “shortest known span”. Whereas a single thread was used for the reference tree-traversal, four threads were used for the proposed

Table 5.7 Comparison of the total number of differences computed for a S-CDO code search and $J \in \{6, 7, 8, 9\}$

J	Nb. of differences (Ref. Algorithm)	Nb. of differences (Novel Algorithm)	Computational Improvement Rate
6	32,067,849	2,867,122	11.18
7	3,706,653,015	227,635,876	16.28
8	294,664,053,120	28,254,421,251	10.43
9	58,393,631,548,884	2,848,228,256,221	20.50

tree-traversal, such as to simulate a parallel search. Indeed, by executing a cooperative search, each thread in the novel algorithm may benefit from the span improvements achieved by the other threads (see Chapter 4). Since we only needed the resulting counter values, the fastest validation routine was used for both algorithms, as it offered the best performance and thus allowed us to complete the tree-traversals in less time. Then, using equations (5.3), (5.4), (5.7) and (5.8), an estimate of the total number of differences for each algorithm was computed by considering *one* second-order difference to be equivalent to *three* first-order differences.

Tables 5.6 and 5.7 compare the total number of differences computed for the reference and the proposed algorithms. The *Computational Improvement Rate* (*CIR*) is defined in this chapter as the ratio of the number of differences computed with the reference algorithm over the number of differences computed with the novel algorithm. Although it is an interesting metric, it is only an approximation: it does not take into account the fact that the novel validation function may not need to generate all differences (if a *collision* is encountered early-on during the validation process), nor does it account for the overhead in computing the next tree-traversal path or the algorithm's ability to be optimized by the compiler.

One can observe that the *CIR* values vary from one search-tree to another. This could be explained by the fact that some search-trees may have, on average, more valid branches that reach to greater depths but that do not lead to valid leaf-node additions: since the algorithm is not able to predict that they will lead to "dead ends", more nodes are invalidated at a greater depth, thus resulting in an increased computational cost. Indeed, the presence of more valid branches reaching to greater depths may explain why, despite an apparent similar search-space size, the search for optimal-span $J = 10$ CDO codes has resulted in being considerably more challenging than the search for optimal-span $J = 12$ S-CDO codes, and hence has not as of yet been completed (see Fig. 5.18 in Section 5.2.5.4). Nevertheless, these results show that compared to the reference tree-traversal, the proposed algorithm converges more quickly to a smaller tree, therefore further reducing the computational cost of the exhaustive search.

5.6 Dealing with the Mean Time Before Failures (MTBF)

Although the proposed search algorithm offers a drastic performance improvement over the reference algorithm, due to the sheer size of the search space (see Section 2.3.1), the search for optimal-span codes having order $J \geq 10$ still requires several months of computation time to complete. Indeed, earlier versions of the novel algorithm were bound in execution time by the low *Mean Time Before Failure* (MTBF) of computers running the search: power outages, network failures, system crashes, or computer reboots following security updates would all result in a significant data loss that would require the search to be restarted from the beginning.

In order to overcome the computers' low MTBF, basic fault-tolerance was included in the design: regular snapshots of the current state of the search are performed at configurable intervals of time. Furthermore, the algorithm has to comply with several additional requirements. First, snapshots have to scale over thousands of computing cores, and thus proper locking of the resources must be used to ensure that the written *state-files* are always coherent. Nevertheless, excessive locking may also aversely affect the performance of the search, and thus must only be applied when strictly essential. Then, to ensure that the work can be migrated from one computer to another, it must be possible to stop and resume the search without a significant loss in progress. Finally, it is important to ensure that *state-files* are written to disk as fast as possible: during a system *shutdown* or *restart*, the operating system only grants the program a few seconds to terminate gracefully, after which its main process is killed. Therefore, to avoid corrupted or incomplete files, the writing of *state-files* must complete before the end of the graceful termination period.

In order to satisfy the above requirements, the current state of the search is efficiently saved into a *verifiable*, *serialized* and *compressed* XML file (see Appendix D). Furthermore, to provide file redundancy, a set of N *state-files* are kept and written to in a *round-robin* fashion: these files correspond to the N most recent snapshots, taken at a configurable time interval. Using multiple *state-files* greatly increases the probability of being able to recover a valid file when the program is terminated unexpectedly, such as during a power outage or a system crash. Upon resuming the search, all N XML *state-files* undergo basic validation: the most recent and valid *state-file* is automatically selected and its contents are used to reinstate the previous state of the search, thus minimizing the loss of search progress.

The novel algorithm makes extensive use of *POSIX signals*: they are employed to trigger the writing of *state-files*, to display a status update on the search, to detect a system shutdown or restart, and to detect a request for stopping the search. Indeed, two threads are used to enable this functionality: one thread handles the signals and cancels the *worker threads* accordingly; a second thread is used for writing the *state-files* at specified time intervals. If the program receives a POSIX signal signifying that the program should gracefully terminate, all *worker threads* are cancelled and then one last *state-file* is written to disk. Then, all memory allocations are freed and general cleanup is performed prior to the main thread's termination.

To ensure that the content of *state-files* is always coherent, *worker thread* canceling is inhibited during critical sections of the code: instead, their cancelation is postponed until they reach specific "cancellation points". Critical sections of the code are kept as small as possible to ensure that the last *state-file* can be written to disk before the operating system forces the program's termination (i.e. a *SIGKILL*).

Since *state-files* are written in XML format and are easy to parse, they may also be used for sharing span updates across different computers. Furthermore, they provide an alternate method of remotely obtaining search-progress updates: a Python script is used to parse the files and provide an easy-to-read summary of the state of the search across different computers on the network. For example, 488 *worker threads* across 22 networked computers are currently in use for the search of optimal-span $J = 10$ CDO codes.

CHAPTER 6

GENERAL DISCUSSION

We have presented a novel high-performance *efficient and parallel implicitly exhaustive search algorithm* for determining optimal/short-span rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes. Using analytical and computer engineering techniques, the proposed search algorithm is much faster than previous exhaustive-search and pseudo-random search algorithms, and features several significant synergistic improvements that led to finding many new and improved codes.

The search algorithm performs a more efficient *implicitly-exhaustive* search-tree traversal that dynamically applies tree-pruning techniques to identify and focus the search on only potentially valid codes. Indeed, in order to facilitate tree pruning, we provided *lower*, *midpoint* and *upper bound* values for nodes in the search-tree, thereby reducing the search complexity by several orders of magnitude. The proposed search algorithm is a type of *branch and bound* algorithm: although it does not test all the branches on the search-tree, it effectively performs an exhaustive search and thus ensures that optimal-span codes are found at the end of the search, and within a reasonable amount of time. This was previously not possible, except for codes having a very small value of J. Moreover, to further reduce the computation time, the algorithm performs a *parallel cooperative search* that can leverage hundreds of processing cores to compute more search-tree branches at the same time and hence converge to a smaller tree at a much faster rate than would otherwise be possible.

The novel validation function focuses on quickly *invalidating* codes rather than *validating* codes, thus ensuring that a code is discarded as early as possible during the validation process. Furthermore, by using compile-time meta-programming techniques to remove the branches and loops in the validation function, we eliminate many of the associated branch-misprediction penalties that would incur on modern microprocessors.

Finally, the novel search algorithm also implements *basic fault-tolerance measures* to counteract the low mean time between failures of the computers running the search.

CHAPTER 7

GENERAL CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

This chapter presents the general conclusions of this thesis and several recommendations for future work.

7.1 General conclusions

In this manuscript-based thesis, we have presented two articles that were published in *IEEE Transactions on Communications* [14, 15], and one article that was submitted for publication in the *IEEE Transactions on Parallel and Distributed Systems* [16].

We have presented a novel high-performance *efficient and parallel implicitly exhaustive search algorithm* for determining optimal/short-span rate $R = \frac{1}{2}$ systematic CDO and S-CDO codes. The search algorithm we have developed performs a more efficient *implicitly-exhaustive* search-tree traversal that dynamically applies tree-pruning techniques to reduce the size of the search space. Indeed, using *lower*, *midpoint* and *upper bound* values for nodes in the search-tree, the algorithm is able to identify and focus the search on only potentially valid codes.

The proposed search algorithm uses a drastically improved (S-)CDO code validation function that employs a novel low-maintenance data structure to perform an *incremental computation* with *data-reuse*. Indeed, the data structure allows for tracking the relevant differences with $O(1)$ time-complexity, thus facilitating data-reuse. The novel validation function only computes the *new differences* generated by the next code candidate, and *reuses* the differences that were already computed for the previous code validation. Hence, the degree of the polynomial equation describing the total number of differences to compute for each code validation is effectively reduced *by one* (from J^4 to J^3). Moreover, it only computes *one element* of the second-order difference pairs, thereby further reducing the number of computed second-order differences by one half.

In order to further reduce the computation time, the algorithm performs a *parallel cooperative search* to compute more search-tree branches at the same time and hence converge to a smaller tree at a faster rate than would otherwise be possible. Indeed, the search algorithm is able to scale efficiently by using an effective load-balancing technique to leverage hundreds of processing cores. Furthermore, to compensate for the low mean time between failures of the computers running the search, the proposed search algorithm implements *basic fault-tolerance measures*: *regular snapshots* of the current state of the search are performed, thus allowing the search to be stopped and resumed without a significant loss of progress. The snapshots are saved in a *verifiable XML format*, thereby ensuring the possibility of recovery in the case of a file corruption and allowing the search to be resumed after a system crash or a system restart.

We have characterized the dramatic speedup achieved with the novel search algorithm over previously published algorithms. Compared to the reference implicitly-exhaustive search algorithm, the resulting high-performance parallel implementation provides an impressive speedup that is larger than 16300 when searching for optimal-span $J = 7$ CDO codes, and larger than 6300 when searching for optimal-span $J = 8$ S-CDO codes. When compared to the reference exhaustive-search (S-)CDO code validation function, the novel validation function offers a speedup that is larger than 190000 when validating $J = 17$ CDO codes, and larger than 60000 when validating $J = 17$ S-CDO codes. Moreover, when compared to the fastest CDO code validation function used in high-performance pseudo-random search algorithms, the novel validation function achieves a speedup that is larger than 2000 for $J = 17$ CDO code validations.

Most of the speedup is achieved by means of a two-pronged approach. On the one hand, the *time required to process each node on the search-tree* is very significantly reduced through the use of the novel validation function: by themselves, the novel data structure (allowing efficient tracking of relevant differences) and the data-reuse (with incremental computation of differences) provide a speedup of two-to-three orders of magnitude. On the other hand, the *amount of work to be processed* is significantly reduced by means of a more efficient parallel tree-traversal: the total number of nodes to process is considerably decreased through the use of the above-mentioned tree-pruning techniques, allowing for the computational cost of the search to be reduced by a factor of around ten. In addition, by performing a simultaneous exploration of different branches on the search-tree, the overall computation time for completing the tree-traversal is almost linearly decreased with the number of processing cores used. These two search-tree traversal enhancements provide an additional speedup of one-to-two orders of magnitude. We observe a resulting total speedup of three-to-four orders of magnitude with respect to the reference algorithm, which suggests that the various optimization techniques employed are independent in nature, and that they synergistically combine such that their respective speedups are multiplied. Finally, the total allowed runtime for the search has also been appreciably increased with the addition fault-tolerance techniques, thus allowing the search to operate without significant down-time for much longer periods of time than previously possible. Indeed, despite the speedup obtained, the computation time required for completing the search for optimal-span codes having order $J \geq 9$ far exceeds the typical uptime of the systems employed (hardware and operating system). As a consequence, to make certain that these codes are found, fault-tolerance techniques must be used in conjunction with an algorithm providing a drastic speedup, such as to ensure a high probability of completing the search. Using the combination of algorithmic optimizations and load-balancing techniques described in the thesis, we were able to complete the search over a search space that is some 10^{14} times larger than previously possible, thus leading to finding new and improved codes.

We have provided new optimal and short-span rate $R = \frac{1}{2}$ systematic (S-)CDO codes with shorter spans than any previously published codes of the same order. Using the novel *efficient and parallel implicitly exhaustive search algorithm*, we were able to determine new optimal-span CDO codes for $J \in \{6, 7, 8, 9\}$, and new optimal-span S-CDO codes for $J \in \{9, 10, 11, 12\}$. Moreover, the proposed algorithm has also allowed us to find several new short-span CDO codes for $J \in [10; 17]$ and several new short-span S-CDO codes for $J \in [13; 20]$. A maximal span reduction of 32% for CDO codes and 34% for S-CDO codes was achieved, and we were able to obtain an average span reduction of 14% for CDO codes and 26% for S-CDO codes. Naturally, these span improvements directly translate into a latency reduction of the same magnitude in the error-correcting iterative threshold decoding systems for which they are intended.

We have described some of the characteristics of the (S-)CDO codes obtained. The spans of the provided codes are compared to known theoretical lower-bounds, and the error-correction performance for some of these codes is presented. We also present the span improvements obtained when using S-CDO codes instead of CDO codes of the same order. For moderate E_b/N_0 values (i.e. $\frac{E_b}{N_0} > 3$ dB), we show that S-CDO codes offer a competitive error performance and a compelling alternative to Turbo codes, since their error performance curves may go below the "*floor*" region of Turbo codes, thus providing for these values of E_b/N_0 a better error performance along with a lower latency and reduced implementation complexity.

We have presented the evolution of the (S-)CDO code error-performance as their order J increases: although the error floor seems to be lowered as J becomes larger, the "*waterfall*" region progressively moves to higher values of E_b/N_0, a fact that will need to be considered when selecting one of these codes for a given application of interest. Indeed, depending on the application, it may or may not be advantageous to employ S-CDO codes with an order J larger than $J > 20$, since the "*waterfall*" region may occur at E_b/N_0 values that are too high to be acceptable for the intended use. Finally, we also conclude that even though CDO codes perform slightly better than S-CDO codes at moderate E_b/N_0 values, from an engineering point of view, S-CDO codes clearly offer a much lower decoding latency for a similar error performance, and hence may be a better alternative to CDO codes.

Our analysis reveals the complexity and challenges of this topic and suggests that significant findings could be expected through further investigation.

7.2 Suggestions for further research

We believe that numerous enhancements to the proposed algorithm are possible, and that several tools may be developed to aid in better understanding CDO codes and their variants.

7.2.1 Improving current-generation (S-)CDO code searching algorithms

Preliminary results have shown that when using the proposed algorithm, a modest gain in CDO code searching performance may be achieved by only computing second-order difference collisions. Indeed, satisfying the second CDO code condition also satisfies the first and the third CDO code conditions: this technique is currently used in the CDO code validation function described in [20]. Nevertheless, the potential speedup has not been properly characterized and requires a further analysis.

Preliminary results have also shown that a reduction in computation time of around 18% may be achieved by reordering the difference generation such that the first and second order differences having the smallest values are generated first. Indeed, as is shown in Appendix B, most of the first and second order (S-)CDO code differences tend to be small in value. Therefore, by computing smaller difference values first, the validation function is more likely to generate differences that will result in a collision at an earlier stage of the validation process, thereby reducing the time required for detecting an invalid node addition. However, the potential speedup obtained with this technique has not been properly characterized, requiring further analysis.

Another technique that may reduce the time required for finding optimal-span (S-)CDO codes is to use the LUT of differences as a means for determining the difference values that are currently not in use. Indeed, these difference values may be used as offsets between the children node values being tested and their parent node value, thus allowing for some nodes values that cannot yield valid codes to be efficiently skipped.

A very significant speedup may be achieved by developing a (S-)CDO code searching algorithm based on the *Shift Algorithm* [17]. Indeed, using a simple and efficient *FLEGE*-like algorithm [66] for finding new optimal-span (S-)CDO codes would clearly allow for a high-performance implementation to be devised. Nevertheless, correctly expressing and computing the first and second order differences with bitmaps and SHIFT/OR operations has so far eluded analysis. Exploring the various possible representations for (S-)CDO codes and their difference values may give some insight into the development of such an algorithm. Alternatively, a Golomb-ruler *pre-selection filter* based on the *Shift Algorithm* may be used for validating potential code candidates as CSO codes first, prior to applying the more computationally expensive (S-)CDO code validation function. Although this technique is less efficient than using bitmaps for all computations, preliminary results have shown that using a Golomb ruler *pre-selection filter* significantly reduces the number of branches that need to undergo (S-)CDO code validation: a very efficient *pre-selection filter* implementation would thus reduce the total time required for determining new optimal-span (S-)CDO codes.

7.2.2 A next-generation error-correction performance simulator

Simulating the error-correction performance of (S-)CDO codes using the current error-performance simulator is a very time-consuming task. Indeed, its very slow operation limits our ability to determine the error performance of (S-)CDO codes for SNR values larger than $\frac{E_b}{N_0} > 4.0$ dB.

Based on a very old 32-bit design, the inefficiencies and serial operation of the simulator do not leverage the performance offered by current multi-core microprocessors. Indeed, to deliver each $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB) bit error rate curve presented in this thesis, several weeks of computation time were required. Furthermore, its current *pseudo-random information bit generation* and *channel noise induced error generation* do not support the fast and accurate error-performance simulation that is required for low bit error rates at higher SNR values.

We propose the development of a next-generation error-correction performance simulator that is based on a modern 64-bit multi-threaded design, leveraging the parallelism offered by modern multi-core microprocessors and employing General-Purpose computing on Graphics Processing Units (GPGPU) to accelerate the simulation time. Indeed, modern graphics processing units (GPUs) have an architecture that is designed for parallel data-processing, offering hundreds or even thousands of parallel streams of computation. Since their architecture is less efficient at decision-making, they are generally used to assist generic microprocessors (CPUs) in completing the work: whereas CPUs handle the overall task management and workflow, GPUs are used as accelerators to perform the complex computations.

Using such a much faster error-correction performance simulator, we would be able to gain insight into the error-performance of (S-)CDO codes at SNR values larger than $\frac{E_b}{N_0} > 4.0$ dB. Furthermore, we would be able to observe where the error-performance floor lies for different values of J. Finally, by performing an automated error-correction simulation for all known short-span (S-)CDO codes and storing the results in a database, we would be able to improve our understanding of the error-correcting performance characteristics of these codes. This information will be useful when designing and implementing high-performance hardware-based error-correcting systems using these codes.

7.2.3 RCDO codes and next-generation search and error-performance simulation algorithms

In 2001, *C. Cardinal* defined *Recursive Convolutional Self-Doubly Orthogonal* (RCDO) codes [4], a more powerful variant of CDO codes. Later, in 2010, *E. Roy* introduced *Simplified RCDO* (S-RCDO) codes [18], which relax some of the RCDO code double-orthogonality conditions, thus potentially offering a reduced decoding latency and encoder/decoder implementation flexibility. In fact, these channel capacity approaching codes were shown to be a special case of *Low-Density Parity-Check* (LDPC) codes [18, 83]. However, compared to LDPC codes, the double orthogonality conditions defining these codes allow for a simplified RCDO/S-RCDO code determination process and the development of error-correcting encoding/decoding systems having a reduced latency and implementation complexity [83].

At low SNR values, RCDO and S-RCDO codes offer a significantly much better error-correction performance than regular (S-)CDO codes [4]. Indeed, whereas (S-)CDO codes only correct *information bits* during the iterative threshold decoding, RCDO and S-RCDO codes are able to correct both, the *information bits* and the *redundant parity check symbols* obtained from the noisy communication channel [4, 83]. We will henceforth refer to RCDO and S-RCDO codes as *(S-)RCDO* codes.

The error-correcting performance of (S-)RCDO codes is controlled by the position and the number of forward and feedback connections that constitute the encoder [83]. However, the precise set of parameters required to extract the best compromise between the *error-correcting performance* and the *implementation flexibility* of these novel codes is still to be defined, and is considered an active topic of research [83]. Therefore, in order to fine-tune the set of parameters resulting in powerful (S-)RCDO codes, an efficient (S-)RCDO code searching algorithm and a very fast (S-)RCDO code error performance simulator need to be devised.

To this end, the (S-)CDO code searching algorithm developed in this thesis may be adapted such as to find (S-)RCDO codes with a specific set of characteristics. Furthermore, the next-generation (S-)CDO code error-correction performance simulator may be modified to provide a GPGPU-accelerated error-performance simulation for (S-)RCDO codes. Indeed, by using the information obtained from the fast error-performance simulator, we will be able to acquire a better understanding of (S-)RCDO codes and their characteristics, and thus define the set of (S-)RCDO code parameters allowing us to determine the most powerful and efficient (S-)RCDO codes.

These high-performance (S-)RCDO codes will be used in the design and development of more powerful and efficient error-correcting systems.

REFERENCES

[1] R. G. Gallager, *Information theory and reliable communication.* John Wiley & Sons Inc, Jan. 1968.

[2] S. Lin and D. J. Costello, *Error control coding*, ser. fundamentals and applications. Prentice Hall, 2004.

[3] J. C. Moreira and P. G. Farrell, *Essentials of error-control coding.* Wiley, 2006.

[4] C. Cardinal, "Décodage à seuil itératif sans entrelacement des codes convolutionnels doublement orthogonaux," Ph.D. dissertation, École Polytechnique de Montréal, Jun. 2001. [Online]. Available: http://en.scientificcommons.org/50599283

[5] E. Roy, "Recherche et Analyse de Codes Convolutionnels Doublement Orthogonaux Simplifiés au Sens Large," Master's thesis, École Polytechnique de Montréal, École Polytechnique de Montréal, Aug. 2006.

[6] C. Cardinal, D. Haccoun, and F. Gagnon, "Iterative threshold decoding without interleaving for convolutional self-doubly orthogonal codes," *IEEE Transactions on Communications*, vol. 51, no. 8, pp. 1274–1282, 2003.

[7] Harris Corporation / F.G., D.H., C.C., "Apparatus for convolutional self-doubly orthogonal encoding and decoding," Patent 6,167,552, Dec., 2000.

[8] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *IEEE Transactions on Communications*, vol. 44, no. 10, pp. 1261–1271, 1996.

[9] Y. V. Svirid and S. Riedel, "Threshold decoding of turbo-codes," in *Proceedings of the IEEE International Symposium on Information Theory, ISIT 1995.* IEEE, 1995, p. 39.

[10] W. Wu, "New Convolutional Codes–Part I," *IEEE Transactions on Communications*, vol. 23, no. 9, pp. 942–956, Sep. 1975. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1092903

[11] J. L. Massey, “Threshold Decoding,” Massachusetts Institute of Technology, Research Laboratory of Electronics, Tech. Rep. 410, Apr. 1963.

[12] W. T. Rankin, “Optimal Golomb Rulers: An Exhaustive Parallel Search Implementation,” Master’s thesis, Duke University, Electrical Engineering Department, Dec. 1993.

[13] Wikipedia. (2012, Apr.) Golomb ruler. [Online]. Available: http://en.wikipedia.org/wiki/Golomb_Ruler

[14] G. Kowarzyk, N. Bélanger, D. Haccoun, and Y. Savaria, “Efficient Parallel Search Algorithm for Determining Optimal R=1/2 Systematic Convolutional Self-Doubly Orthogonal Codes,” *IEEE Transactions on Communications*, vol. 61, no. 3, pp. 865–876, 2013.

[15] ——, “Efficient Search Algorithm for Determining Optimal R=1/2 Systematic Convolutional Self-Doubly Orthogonal Codes,” *IEEE Transactions on Communications*, vol. 60, no. 1, pp. 3–8, 2012.

[16] ——, “Optimizing the Parallel Tree-Search for Finding Shortest-Span CDO Codes of Order J,” *IEEE Transactions on Parallel and Distributed Systems - submitted August 18, 2013*, pp. 1–14.

[17] A. Dollas, W. Rankin, and D. McCracken, “A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler,” *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 379–382, 1998.

[18] E. Roy, C. Cardinal, and D. Haccoun, “Recursive convolutional codes for time-invariant LDPC convolutional codes,” in *Proceedings of the IEEE International Symposium on Information Theory, ISIT 2010.* IEEE, 2010, pp. 834–838.

[19] G. Kowarzyk, Y. Savaria, and D. Haccoun, “Searching for short-span Convolutional Doubly Self-Orthogonal Codes: A parallel implicitly-exhaustive -search algorithm,” in *Canadian Conference on Electrical and Computer Engineering, CCECE 2008.*, 2008, pp. 001 659–001 662. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4564824

[20] D. Haccoun, C. Cardinal, and F. Gagnon, “Search and Determination of Convolutional Self-Doubly Orthogonal Codes for Iterative Threshold Decoding,” *IEEE Transactions on Communications*, vol. 53, no. 5, pp. 802–809, May 2005.

[21] C. Cardinal, E. Roy, and D. Haccoun, “Simplified Convolutional Self-Doubly Orthogonal Codes: Search Algorithms and Codes Determination,” *IEEE Transactions on Communications*, vol. 57, no. 6, pp. 1674–1682, 2009.

[22] I. Chatzigeorgiou, M. Rodrigues, I. Wassell, and R. Carrasco, “Analysis and design of punctured rate-1/2 Turbo codes exhibiting low error floors,” *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 6, pp. 944–953, 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5174523

[23] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.

[24] A. Viterbi, “Principles of Digital Communication and Coding,” McGraw-Hill, 1979.

[25] L. Hanzo, T. H. Liew, and B. L. Yeap, *Turbo coding, turbo equalisation and space-time coding*. Department of Electronics and Computer Science, University of Southampton, UK: John Wiley & Sons, 2002.

[26] V. K. Bhargava, D. Haccoun, R. Matyas, and P. P. Nuspl, *Digital Communications by Satellite: Modulation, Multiple Access and Coding*. John Wiley & Sons Inc, Jan. 1982.

[27] L. Yang, H. Liu, and C. J. R. Shi, “Code construction and FPGA implementation of a low-error-floor multi-rate low-density Parity-check code decoder,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 4, pp. 892–904.

[28] D. Haccoun, C. Cardinal, and F. Gagnon, “New results on the search for convolutional self-doubly orthogonal codes suitable for iterative threshold decoding,” in *IEEE 58th Vehicular Technology Conference, VTC-Fall 2003*. IEEE, 2003, pp. 304–307 Vol. 1.

[29] A. Dimitromanolakis, “Analysis Of The Golomb Ruler And The Sidon Set Problems And Determination Of Large Near-Optimal Golomb Rulers,” Department of Electronic and Computer Engineering, Technical University of Crete, Tech. Rep., Jun. 2002.

[30] American Mathematical Society. (2012, Jan.) Feature Column from the AMS: Weird Rulers. [Online]. Available: http://www.ams.org/samplings/feature-column/fc-2012-01

[31] Out of the Norm. (2012, Jan.) From rope-cutting to Golomb rulers, via magic. [Online]. Available: http://outofthenormmaths.wordpress.com/2012/01/25/golomb-rulers/

[32] J. Jedwab and M. Strange, "Wavelength Isolation Sequence Design," *IEEE Transactions on Information Theory*, vol. 59, no. 5, pp. 3210–3214.

[33] J. Wodlinger, "Costas arrays, Golomb rulers and wavelength isolation sequence pairs," Master's thesis, Simon Fraser University, Simon Fraser University, Apr. 2012.

[34] B. Beavers, "Golomb Rulers and Graceful Graphs," 2001.

[35] G. S. Bloom and S. W. Golomb, "Applications of numbered undirected graphs," *Proceedings of the IEEE*, vol. 65, no. 4, pp. 562–570, Apr. 1977.

[36] K. Klonowska, L. Lundberg, and H. Lennerstad, "Using Golomb rulers for optimal recovery schemes in fault tolerant distributed computing," in *IEEE International Parallel & Distributed Processing Symposium, IPDPS 2003.* IEEE Comput. Soc, p. 9.

[37] K. Klonowska, L. Lundberg, H. Lennerstad, and C. Svahnberg, "Extended Golomb Rulers as the New Recovery Schemes in Distributed Dependable Computing," in *IEEE International Parallel & Distributed Processing Symposium, IPDPS 2005.* IEEE, 2005, pp. 279a–279a.

[38] M. K. Suaidi, L. Yong, and A. A. R. B. P. H. Yusof, "Third and Fifth Order IMDs at Carrier Position," in *2006 International RF and Microwave Conference.* IEEE, pp. 455–459.

[39] M. K. Suaidi, L. Yong, and A. Yusof, "Carrier position composite triple beat reduction using the Golomb ruler," in *13th IEEE International Conference on Networks, 2005.* IEEE, 2005, p. 6 pp.

[40] L. Yong, A. A. Rahman, B. Yusof, and M. K. Suaidi, “Fiber optic microcellular system composite triple beat reduction using the Golomb ruler,” in *Proceedings of the 12th IEEE International Conference on Networks, 2004.* IEEE, 2004, pp. 590–594.

[41] S. R. Blackburn, T. Etzion, K. M. Martin, and M. B. Paterson, “Two-Dimensional Patterns With Distinct Differences-Constructions, Bounds, and Maximal Anticodes,” *IEEE Transactions on Information Theory*, vol. 56, no. 3, pp. 1216–1229, 2010.

[42] M. Schroeder, *Number Theory in Science and Communication*, ser. With Applications in Cryptography, Physics, Digital Information, Computing, and Self-Similarity. Springer Verlag, Dec. 2008.

[43] J. B. Shearer, I. Center, and Y. Heights, “Some new optimum Golomb rulers,” *IEEE Transactions on Information Theory*, vol. 36, no. 1, pp. 183–184, 1990.

[44] S. W. Soliday, A. Homaifar, and G. L. Lebby, “Genetic algorithm approach to the search for Golomb rulers,” in *Proceedings of the International Conference on Genetic Algorithms.* Citeseer, 1995, pp. 528–535.

[45] J. Tavares, F. B. Pereira, and E. Costa, “Understanding the role of insertion and correction in the evolution of Golomb rulers,” in *Proceedings of the 2004 Congress on Evolutionary Computation.* IEEE, 2004, pp. 69–76.

[46] C. Meyer and P. A. Papakonstantinou, “On the complexity of constructing Golomb Rulers,” *Discrete applied mathematics*, vol. 157, no. 4, pp. 738–748, Feb. 2009.

[47] M. Sorge, “Algorithmic Aspects of Golomb Ruler Construction,” *arXiv.org*, May 2010.

[48] C. Berrou, A. Glavieux, and P. Thitmajshima, “Near Shannon limit error-correcting coding and decoding: Turbo-codes,” in *IEEE International Conference on Communications, ICC 1993.*, 1993, pp. 1064–1070.

[49] Y. He and D. Haccoun, “An analysis of the orthogonality structures of convolutional codes for iterative decoding,” *IEEE Transactions on Information Theory*, vol. 51, no. 9, pp. 3247–3261, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1499055

[50] B. Baechler, “Analyse et détermination de codes doublement orthogonaux pour décodage itératif,” Master’s thesis, École Polytechnique de Montréal, École Polytechnique de Montréal, Jun. 2000.

[51] C. Cardinal, D. Haccoun, and Y. He, “Reduced-complexity convolutional self-doubly orthogonal codes for efficient iterative decoding,” in *IEEE 63rd Vehicular Technology Conference, VTC-Spring 2006.* IEEE, 2006, pp. 1372–1376.

[52] K. Drakakis, “A Review of the Available Construction Methods for Golomb Rulers,” *Advances in Mathematics of Communications*, vol. 3, no. 3, pp. 235–250, 2009.

[53] P. Galinier and B. Jaumard, “A tabu search algorithm for difference triangle sets and Golomb rulers,” *Computers & Operations Research*, vol. 33, no. 4, pp. 955–970, 2006.

[54] J. P. Robinson, “Genetic search for Golomb arrays,” *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 1170–1173, May 2000.

[55] T. Leitão, “Evolving the Maximum Segment Length of a Golomb Ruler,” in *Genetic and Evolutionary Computation Conference*, 2004.

[56] J. Tavares, F. B. Pereira, and E. Costa, “Golomb Rulers: A fitness landscape analysis,” in *2008 IEEE Congress on Evolutionary Computation (CEC).* IEEE, 2008, pp. 3695–3701.

[57] S. Bansal, S. Kumar, H. Sharma, and P. Bhalla, “Generation of Golomb Ruler Sequences and Optimization Using Biogeography Based Optimization,” in *5th International Multi Conference on Intelligent Systems, Sustainable, New and Renewable Energy Technology and Nanotechnology (IISN-2011)*, Feb. 2011.

[58] C. Cotta and A. Fernández, “A Hybrid GRASP – Evolutionary Algorithm Approach to Golomb Ruler Search,” in *Parallel Problem Solving from Nature - PPSN VIII*, X. Yao, E. Burke, J. Lozano, J. Smith, J. Merelo-Guervós, J. Bullinaria, J. Rowe, P. Tiňo, A. Kabán, and H.-P. Schwefel, Eds. Springer Berlin / Heidelberg, 2004, pp. 481–490.

[59] I. Dotú and P. Van Hentenryck, “A simple hybrid evolutionary algorithm for finding golomb rulers,” in *IEEE Congress on Evolutionary Computation, CEC 2005.* IEEE, 2005, pp. 2018–2023 Vol. 3.

[60] C. Cotta, I. Dotu, A. J. Fernández, and P. Van Hentenryck, "A memetic approach to Golomb rulers," in *Proceedings of the Parallel Problem Solving From Nature, PPSN IX.* Univ Malaga, Dpto Lenguajes & Ciencias Computac, E-29071 Malaga, Spain, 2006, pp. 252–261.

[61] C. Cotta, I. Dotu, A. Fernández, and P. Van Hentenryck, "Local Search-based Hybrid Algorithms for Finding Golomb Rulers," *Constraints*, vol. 12, no. 3, pp. 263–291, 2007.

[62] N. Ayari, T. V. Luong, and A. Jemai, "A hybrid genetic algorithm for Golomb ruler problem," in *2010 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA).* IEEE, 2010, pp. 1–4.

[63] R. Morales Espina, "Recherche de règles de Golomb optimales selon la programmation par contraintes," Master's thesis, Université de Montréal, École Polytechnique de Montréal, Jan. 2002.

[64] C. Jaillet and M. Krajecki, "Constructing optimal Golomb rulers in parallel," *Proc 6th European Workshop on OpenMP*, 2004.

[65] J. B. Shearer, "Improved LP lower bounds for difference triangle sets," *Electronic Journal of Combinatorics*, vol. 6, no. R31, p. 2, 1999.

[66] Distributed.net. (2013, Jun.) distributed.net: public source code. [Online]. Available: http://www.distributed.net/Source

[67] A. Dollas, W. T. Rankin, and D. McCracken, "A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler," Tech. Rep., 1995.

[68] A. Dollas, E. Sotiriades, and A. Emmanouelides, "Architecture and design of GE1, an FCCM for Golomb ruler derivation," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1998.* IEEE, 1998, pp. 48–56.

[69] E. Sotiriades, A. Dollas, and P. Athanas, "Hardware-software codesign and parallel implementation of a Golomb ruler derivation engine," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.* IEEE, 2000, pp. 227–235.

[70] P. Malakonakis, E. Sotiriades, and A. Dollas, "GE3: a single FPGA client-server architecture for Golomb ruler derivation," in *International Conference on Field-Programmable Technology (FPT), 2010.* IEEE, 2010, pp. 470–473.

[71] B. Baechler, D. Haccoun, and F. Gagnon, "On the search for self-doubly orthogonal codes," in *Proceedings of the IEEE International Symposium on Information Theory, ISIT 2000.* IEEE, 2000, p. 292.

[72] Wikipedia. (2013, Jun.) Depth-first search. [Online]. Available: http://en.wikipedia.org/wiki/Depth_first_search

[73] ——. (2012, Apr.) Branch and bound. [Online]. Available: http://en.wikipedia.org/wiki/Branch_and_bound

[74] ——. (2013, Jun.) Bubble sort. [Online]. Available: http://en.wikipedia.org/wiki/Bubble_sort

[75] ——. (2013, Jun.) Speedup. [Online]. Available: http://en.wikipedia.org/wiki/Speedup

[76] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 215–222, Mar. 1981.

[77] A. Nemr, C. Cardinal, M. Sawan, and D. Haccoun, "Very high throughput iterative threshold decoder for convolutional self-doubly orthogonal codes," in *Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference, NEWCAS-TAISA 2008.* IEEE, 2008, pp. 257–260. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4606370

[78] D. Haccoun and C. Cardinal, "High-rate punctured convolutional self-doubly orthogonal codes for iterative threshold decoding," *IEEE Transactions on Communications*, vol. 53, no. 1, pp. 55–63, 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1391190

[79] A. H. Land and A. G. Doig, "An automatic method of solving discrete programming problems," *Econometrica: Journal of the Econometric Society*, pp. 497–520, 1960.

[80] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Revised Fourth Edition*, ser. The Hardware/Software Interface. Morgan Kaufmann, Nov. 2011. [Online]. Available: http://books.google.ca/books?id=DMxe9AI4-9gC&pg=PP4&dq=intitle:Computer+organization+and+design+the+hardware+software+interface+2009&cd=1&source=gbs_api

[81] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, Sep. 1965. [Online]. Available: http://portal.acm.org/citation.cfm?doid=365559.365617

[82] D. P. Rodgers, "Improvements in multiprocessor system design," *ACM SIGARCH Computer Architecture News*, vol. 13, no. 3, pp. 225–231, Jun. 1985. [Online]. Available: http://portal.acm.org/citation.cfm?doid=327070.327215

[83] E. Roy, "*Étude des Propriétés des Codes Convolutionnels Récursifs Doublement-Orthogonaux*," Ph.D. dissertation, École Polytechnique de Montréal, École Polytechnique de Montréal, Sep. 2011.

APPENDIX A

ERROR-CORRECTING PERFORMANCE FOR SOME CDO/S-CDO CODES

In this appendix, we present the error-correcting performance of several $R = \frac{1}{2}$ systematic (S-)CDO codes for orders $J \in [9; 20]$.

For each code of span M, the number of decoding iterations, *iter*, was increased until no appreciable improvement in error-performance was observed: the total decoding latency for each code is proportional to "M x *iter*".

The following figures show that for medium signal-to-noise ratios of $\frac{E_b}{N_0} \geq 3$ (dB), S-CDO codes offer a much lower decoding latency than CDO codes of the same order J, but at the cost of a small degradation in error-correcting performance. Nevertheless, the error-correcting capability of these codes depends essentially on the dimension J of the vector generator [20]. Therefore, the small degradation in error-performance can potentially be compensated for by selecting, within a given "latency budget", an S-CDO code having a larger J value than the order of a CDO code of equivalent latency. Finally, we can also observe that CDO codes exhibit their "waterfall" region at lower $\frac{E_b}{N_0}$ values than their S-CDO code counterparts.

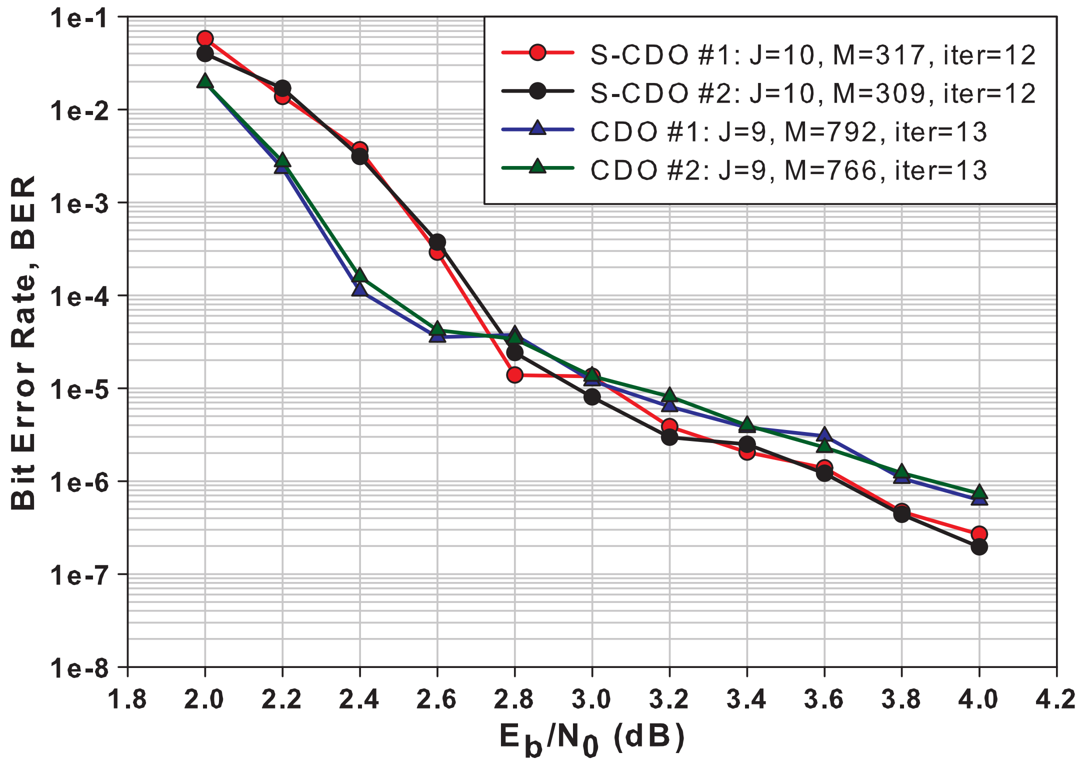


Figure A.1 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 12 decoding iterations for two S-CDO codes of order $J = 10$, and after 13 decoding iterations for two CDO codes of order $J = 9$.

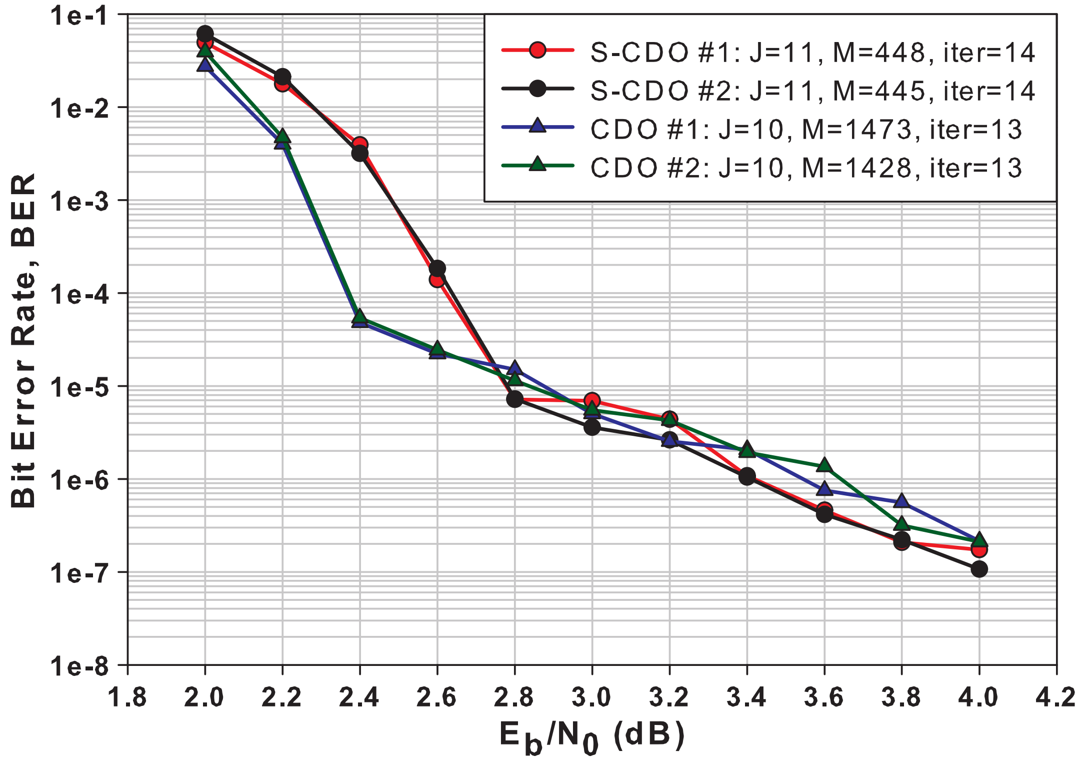


Figure A.2 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 14 decoding iterations for two S-CDO codes of order $J = 11$, and after 13 decoding iterations for two CDO codes of order $J = 10$.

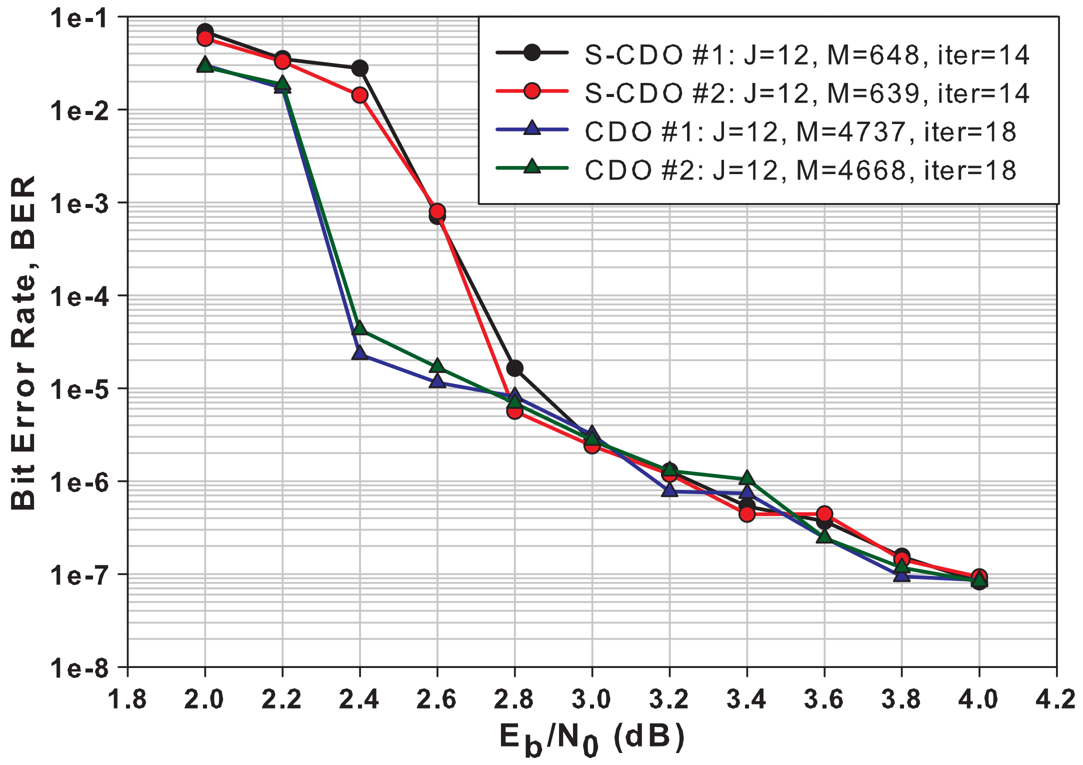


Figure A.3 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 14 decoding iterations for two S-CDO codes of order $J = 12$, and after 18 decoding iterations for two CDO codes of order $J = 12$.

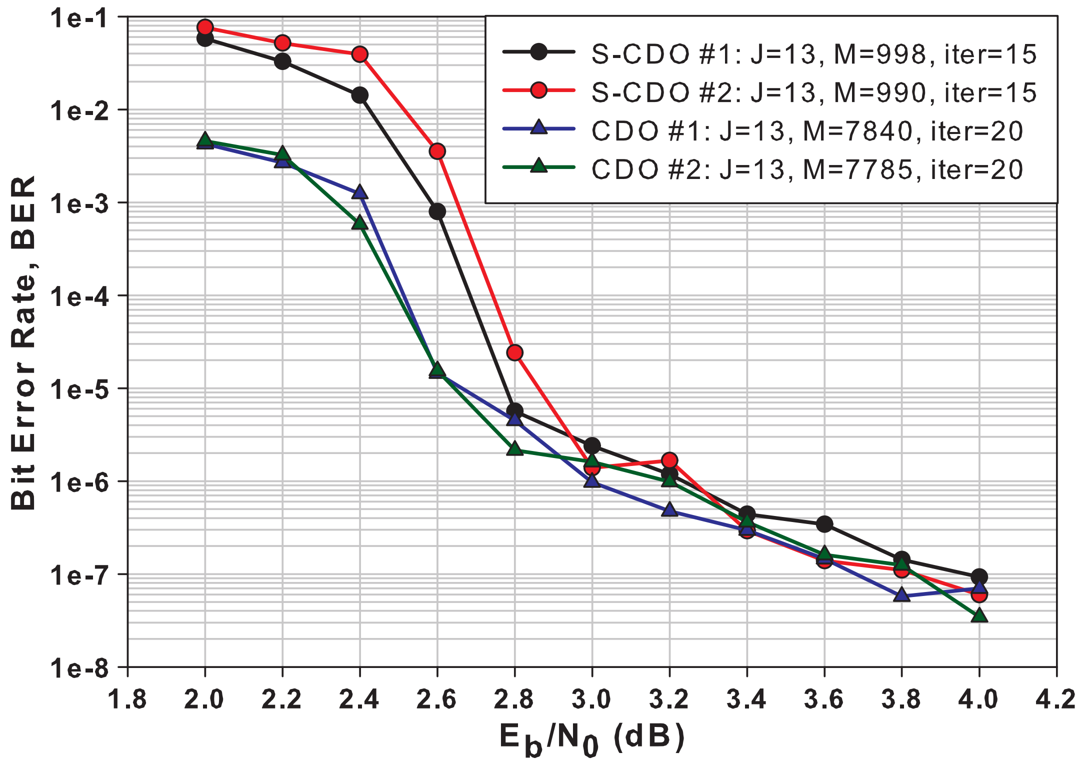


Figure A.4 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 15 decoding iterations for two S-CDO codes of order $J = 13$, and after 20 decoding iterations for two CDO codes of order $J = 13$.

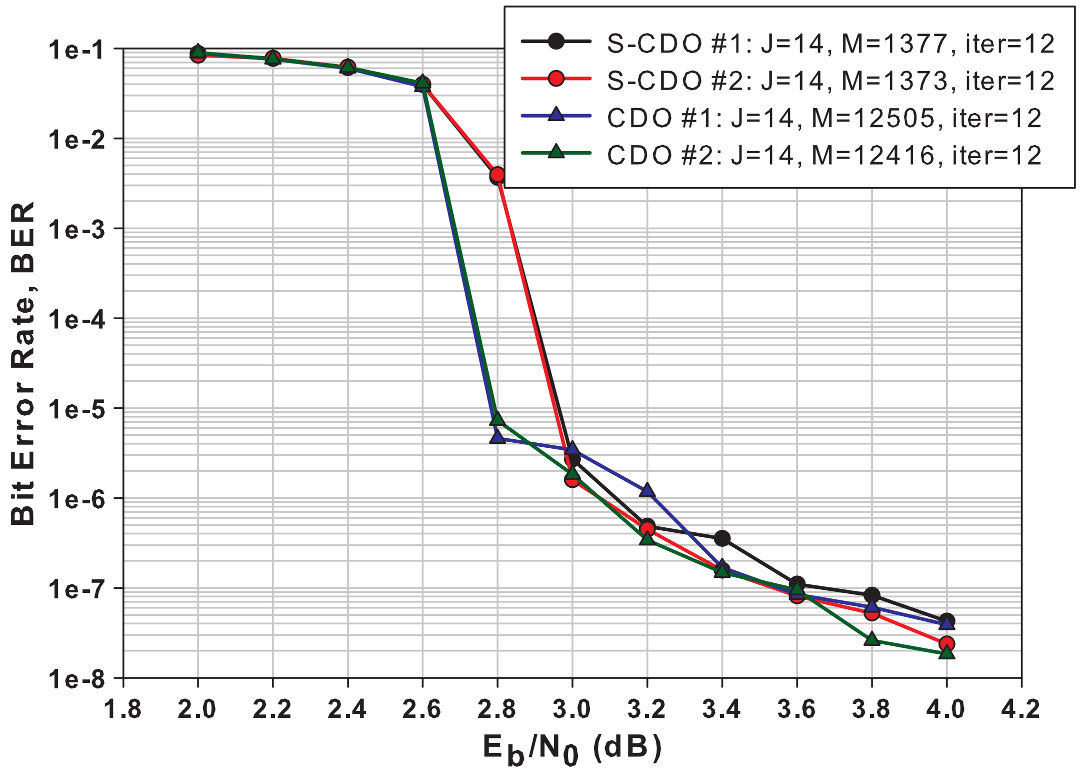


Figure A.5 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 12 decoding iterations for two S-CDO codes and two CDO codes of order $J = 14$.

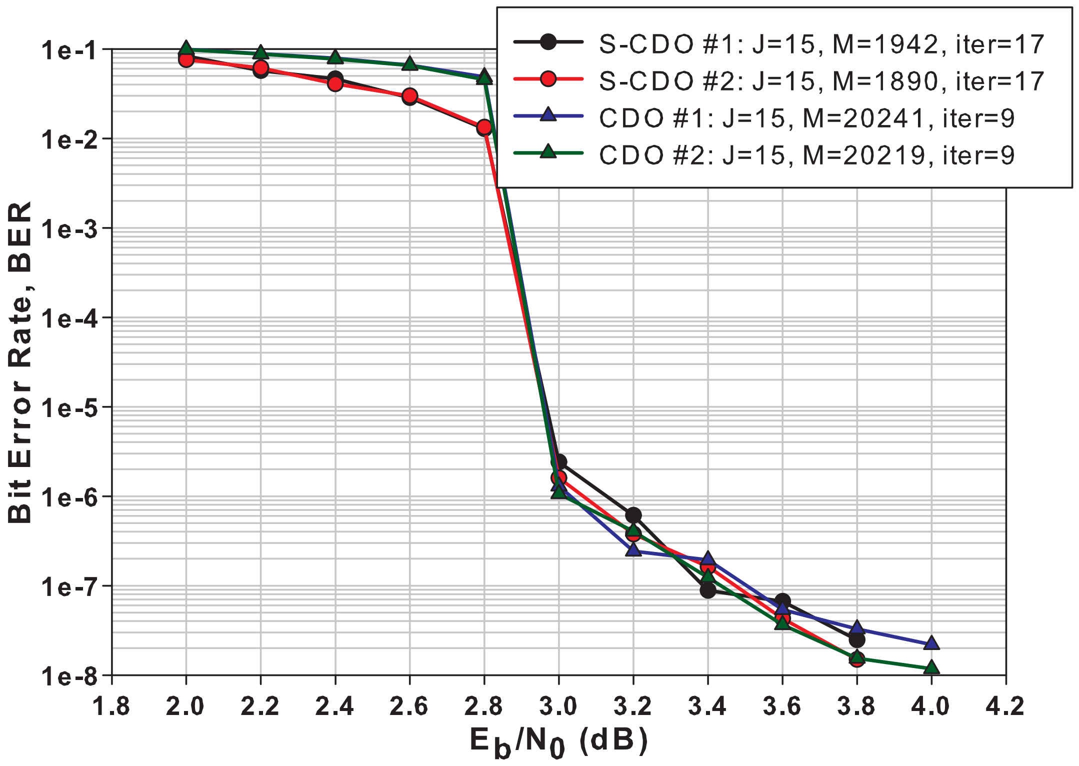


Figure A.6 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance after 17 decoding iterations for two S-CDO codes of order $J = 15$ ($\frac{E_b}{N_0} \in [2.0; 3.8]$ dB), and after 9 decoding iterations for two CDO codes of order $J = 15$ ($\frac{E_b}{N_0} \in [2.0; 4.0]$ dB).

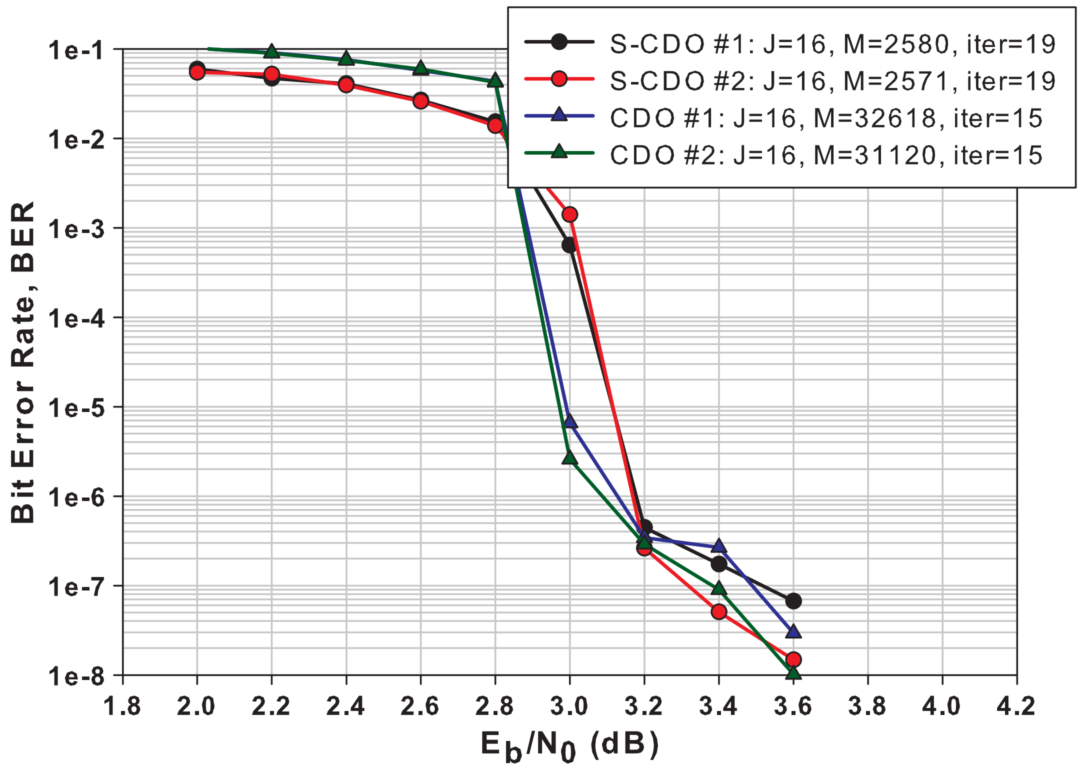


Figure A.7 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 3.6]$ (dB), after 19 decoding iterations for two S-CDO codes of order $J = 16$, and after 15 decoding iterations for two CDO codes of order $J = 16$.

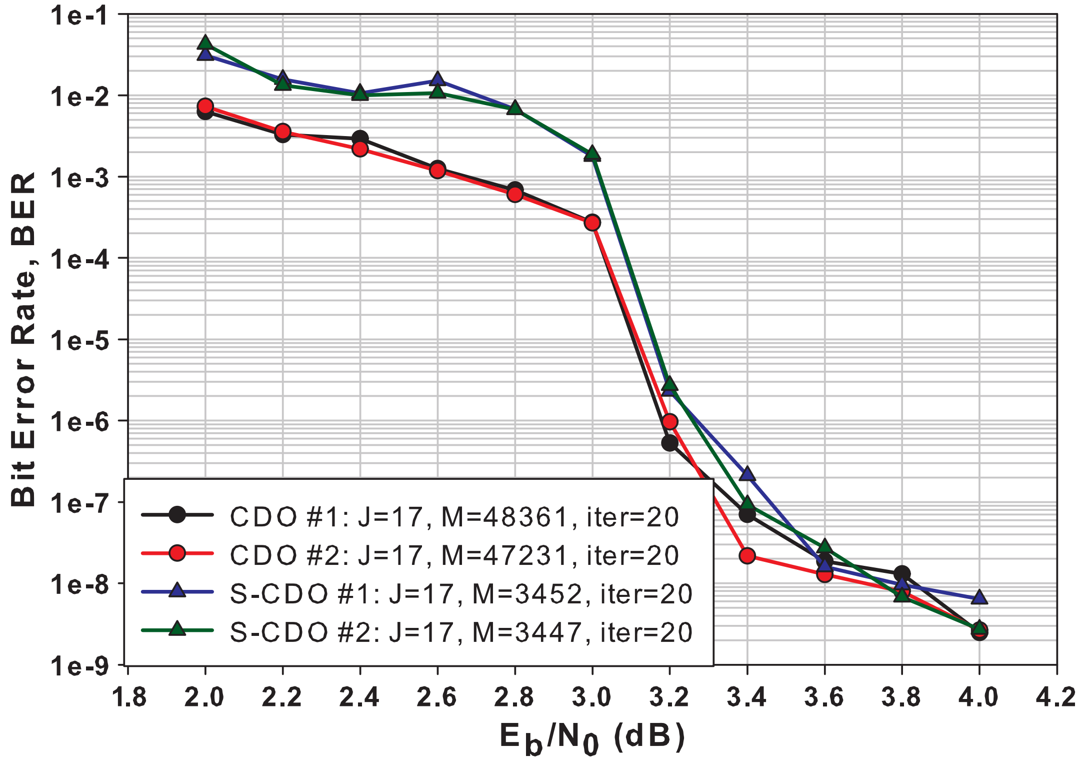


Figure A.8 Rate $R = \frac{1}{2}$ systematic (S-)CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 4.0]$ (dB), after 20 decoding iterations for two CDO codes and two S-CDO codes of order $J = 17$.

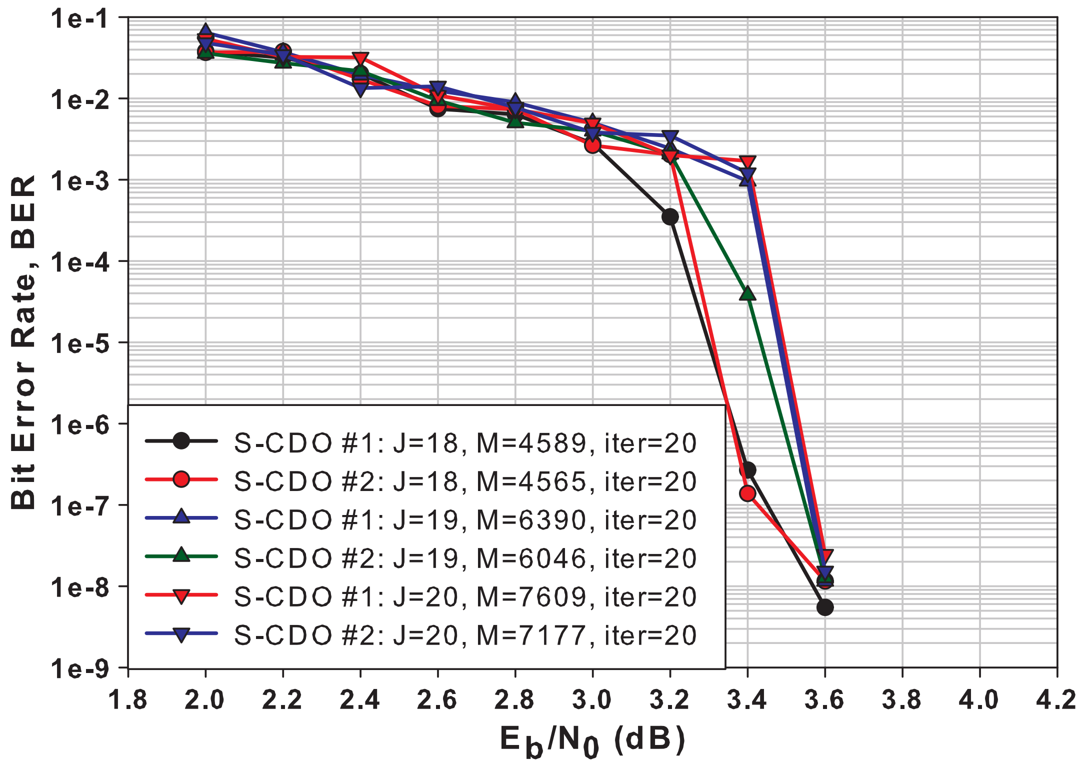


Figure A.9 Rate $R = \frac{1}{2}$ systematic S-CDO code error-correction performance for $\frac{E_b}{N_0} \in [2.0; 3.6]$ (dB) after 20 decoding iterations, for two codes of orders $J = 18$, $J = 19$ and $J = 20$ respectively.

APPENDIX B

DENSITY MAPS FOR SOME CDO/S-CDO CODES

In this appendix, we present density maps for the rate $R = \frac{1}{2}$ systematic (S-)CDO codes provided in [14, 15]. Density maps depict, along an axis of natural numbers $\mathbb{N}$ used as indices, the "density" of difference values generated during a (S-)CDO code validation. Indeed, for each figure, these natural numbers are spaced evenly and increase monotonically in value from left to right: each index (or "slot") holds a colored thin vertical bar that indicates the *presence* or *absence* of a given difference value, its *type*, and for S-CDO code second-order differences, the *number of times the difference was generated*.

For each figure below, a set of codes from [14, 15] were chosen: the density maps were scaled such that the *distance* between "index value zero" and the index representing the "largest possible difference value in that set" occupies the *available width*. First-order difference values are represented by a red vertical bar. Second-order difference values are represented by a blue vertical bar. Since second-order difference values may repeat for S-CDO codes, the shade of the blue bar was used to indicate the number of times a same difference value was generated: up to six repetitions or shades of blue, from lighter to darker blue, were accounted for. Second-order differences that repeat more than six times were capped to a value of six repetitions. Index values not tagged with a first or second order difference value hold a white vertical bar: they represent "empty slots" (i.e. the difference value is absent for that code). The codes employed for generating the density maps in this appendix are provided in Table B.1 below. We will define the *density of a code* as the ratio of the red and blue areas over the white area comprised between index zero (leftmost vertical bar) and the rightmost blue bar (maximal second-order difference value) on their density map.

Figure B.1 shows the density map for two optimal-span CDO codes ($J \in \{6, 7\}$) and two optimal-span S-CDO codes ($J = 9$). One can observe that there are many more second-order differences than first-order differences, and that S-CDO codes seem to be denser than CDO codes (this will be easier to see in the following figures). One can also see that although

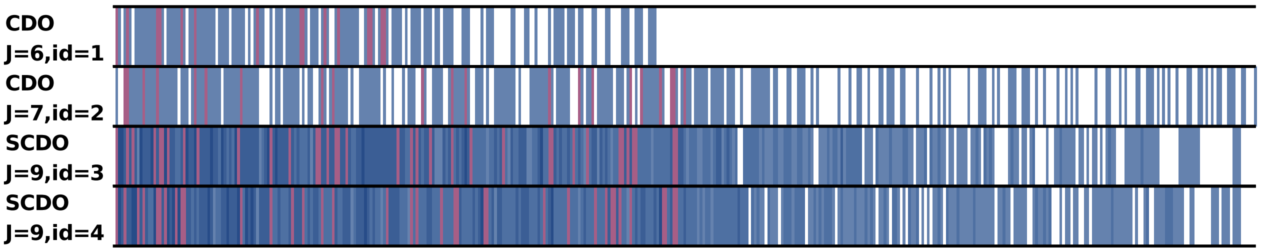


Figure B.1 Density map for two optimal-span CDO codes ($J \in \{6, 7\}$) and two optimal-span S-CDO codes ($J = 9$).

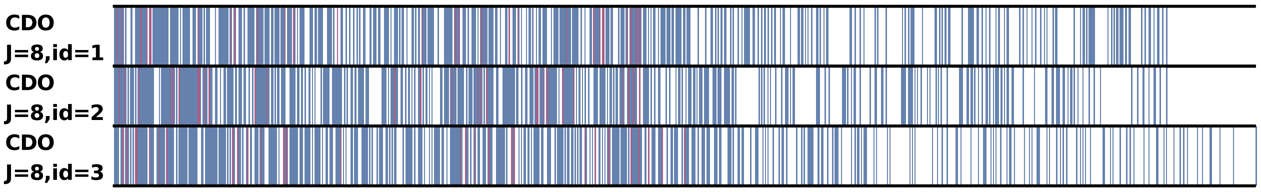


Figure B.2 Density map for two $J = 8$ optimal-span CDO codes ($id \in \{1, 2\}$) and one $J = 8$ short-span CDO code ($id = 3$).

both $J = 9$ S-CDO codes have an equal span, the code with $id = 3$ has darker blue areas and more white space than the code with $id = 4$: this is due to the fact that this code has a larger δ value, which signifies that a higher percentage of second-order difference values are repetitions of previously computed second-order differences. Note that the total number of differences generated for these two codes is equal, as it just depends on J (see (3.2) and (3.3)).

Figure B.2 shows the density map for two $J = 8$ optimal-span CDO codes ($id \in \{1, 2\}$) and one $J = 8$ short-span CDO code ($id = 3$). One can observe that, for the two optimal-span codes, the generated set of difference values is very different, and that therefore these kinds of patterns could be used as a type of "fingerprint" for each code. One can also see that the optimal span codes are denser than the short-span code, as the same number of differences is represented in a smaller horizontal distance.

Figure B.3 clearly shows that for a similar span, an S-CDO code is much denser than a CDO code, as the S-CDO code is able to "pack" many more differences in the same horizontal distance ($J = 14$ for the S-CDO code vs. $J = 10$ for the CDO code).

Figure B.4 also clearly shows how S-CDO codes are denser than a CDO code of similar

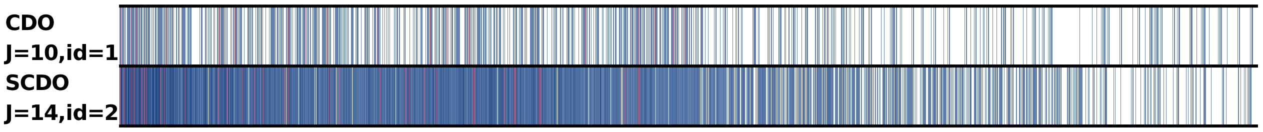


Figure B.3 Density map for a short-span CDO code ($J = 10$) and a short-span S-CDO code ($J = 14$).

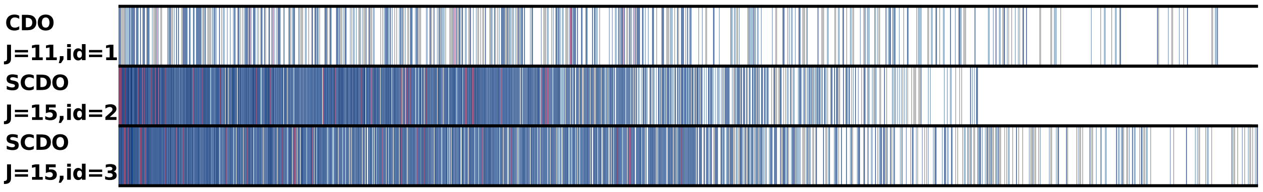


Figure B.4 Density map for a short-span CDO code ($J = 11$) and two short-span S-CDO codes ($J = 15$).

span, but also how as the span for an S-CDO code is reduced ($id = 3$ vs. $id = 4$), the density of the S-CDO codes is increased.

Figure B.5 illustrates the density maps for two $J \in \{10, 11\}$ optimal-span S-CDO codes ($id \in \{1, 3\}$) and two $J \in \{10, 11\}$ short-span S-CDO codes ($id \in \{2, 4\}$). Note how the location of the second-order difference value repetitions is different for each code.

Figure B.6 shows how even the density map of an optimal-span $J = 9$ CDO code is less dense than the ones for the short-span $J = 12$ S-CDO codes. Figure B.7 also shows how the short-span $J = 10$ CDO codes have a density map that is less dense than the density-map of the short-span $J = 12$ S-CDO codes.

Figure B.8 shows how as J increases, the density map for CDO codes becomes less dense.

Figures B.9, B.10, B.11 and B.12 illustrate again the difference in density for S-CDO codes

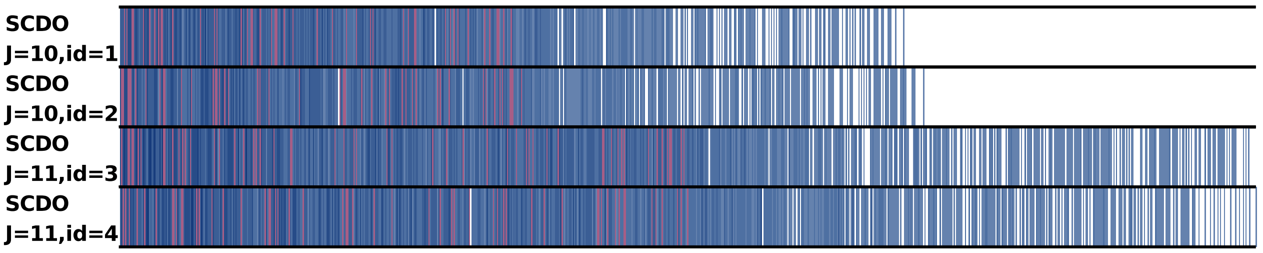


Figure B.5 Density map for two $J \in \{10, 11\}$ optimal-span S-CDO codes ($id \in \{1, 3\}$) and two $J \in \{10, 11\}$ short-span S-CDO codes ($id \in \{2, 4\}$).

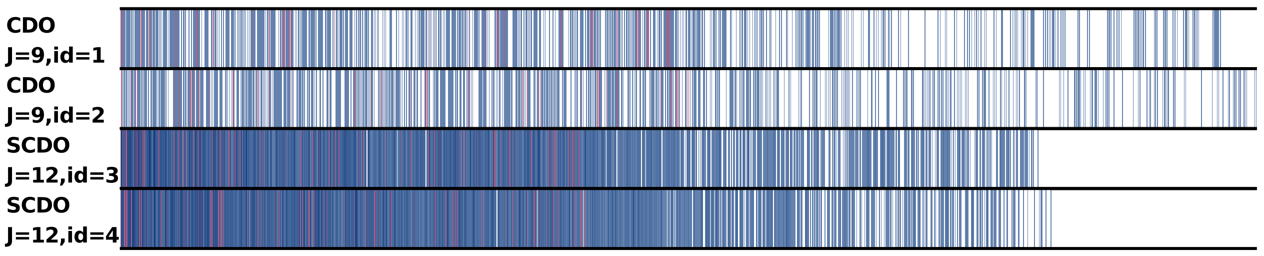


Figure B.6 Density map for one $J = 9$ optimal-span CDO code ($id = 1$), one $J = 9$ short-span CDO code ($id = 2$), and two $J = 12$ short-span S-CDO codes ($id \in \{3, 4\}$).

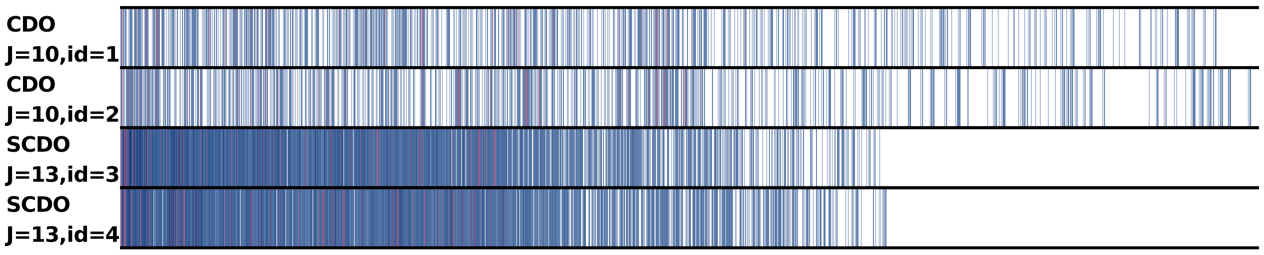


Figure B.7 Density map for two short-span CDO codes ($J = 10$) and two short-span S-CDO codes ($J = 13$).

and CDO codes for orders $J \in \{14, 15, 16, 17\}$ respectively. One can see that as J increases, the density is reduced more rapidly for CDO codes than for S-CDO codes, as the span of CDO codes increases more rapidly with J than for S-CDO codes. One can also observe that for both, S-CDO and CDO codes, the tendency is for most generated differences values to be smaller than half the maximal second-order difference value.

Finally, Figure B.13 shows the density map for six short-span S-CDO codes ($J \in \{18, 19, 20\}$). One can clearly see that most of the generated difference values are smaller than half of the maximal second-order difference value.

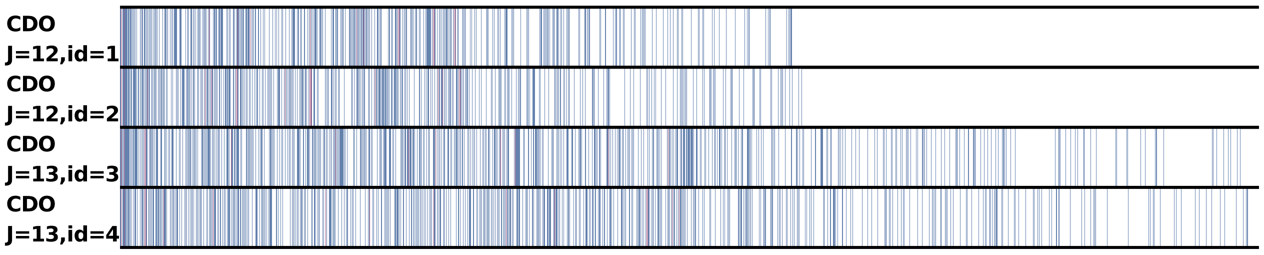


Figure B.8 Density map for four short-span CDO codes ($J \in \{12, 13\}$).

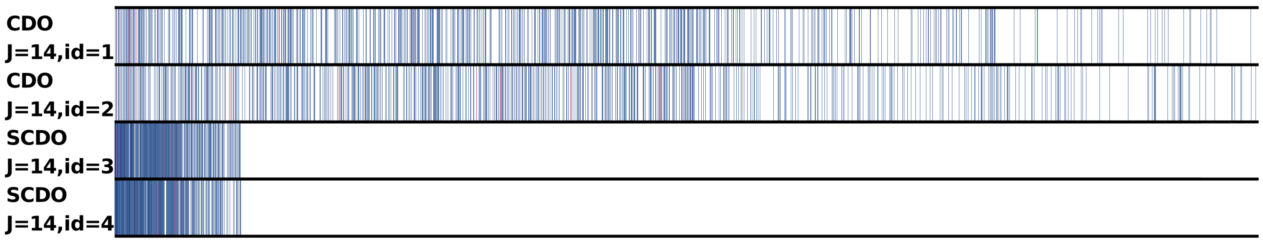


Figure B.9 Density map for two short-span CDO codes ($J = 14$) and two short-span S-CDO codes ($J = 14$).

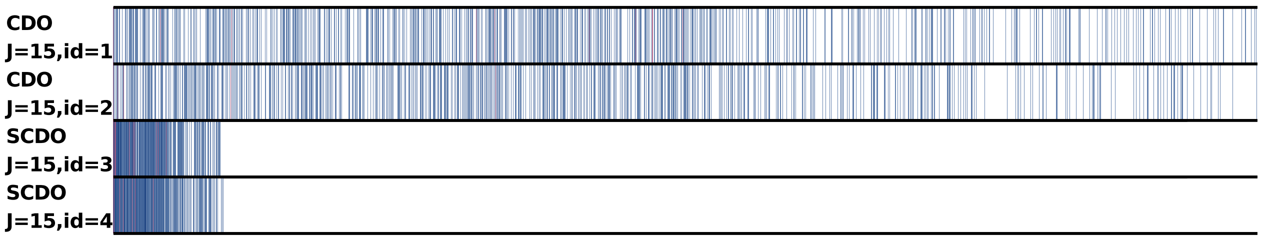


Figure B.10 Density map for two short-span CDO codes ($J = 15$) and two short-span S-CDO codes ($J = 15$).

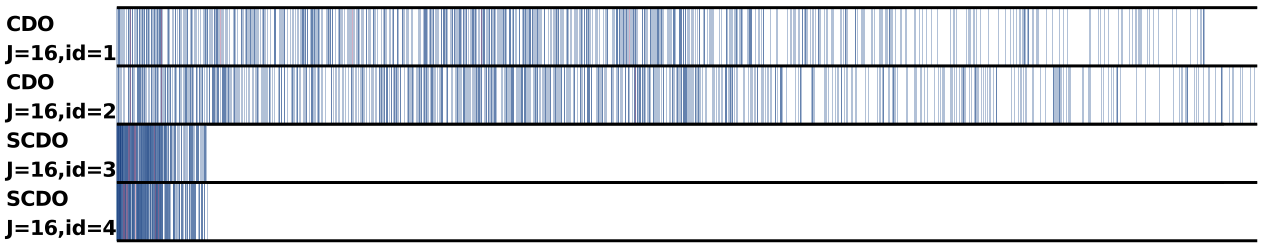


Figure B.11 Density map for two short-span CDO codes ($J = 16$) and two short-span S-CDO codes ($J = 16$).

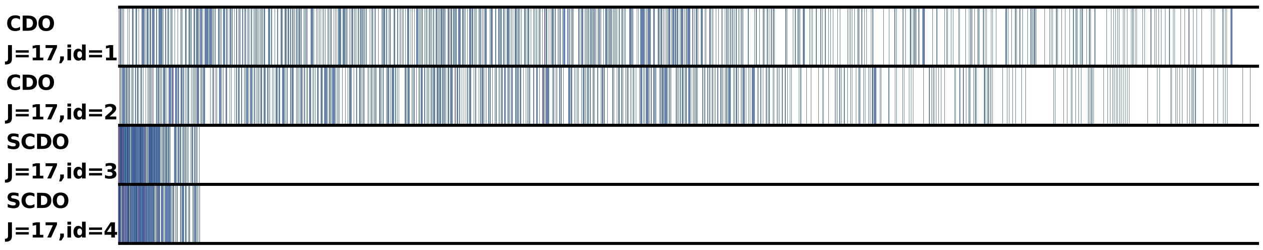


Figure B.12 Density map for two short-span CDO codes ($J = 17$) and two short-span S-CDO codes ($J = 17$).

Table B.1 Code to Density Map Mapping (1 of 2)

Figure	Code Type	J	id	(S-)CDO Code
B.1	CDO	6	1	{0, 1, 17, 70, 95, 100} [15]
B.1	CDO	7	2	{0, 4, 34, 81, 195, 206, 211} [15]
B.1	S-CDO	9	3	{0, 15, 20, 46, 125, 132, 190, 207, 208} [15]
B.1	S-CDO	9	4	{0, 1, 17, 26, 127, 138, 185, 204, 208} [15]
B.2	CDO	8	1	{0, 3, 30, 98, 278, 394, 416, 423} [15]
B.2	CDO	8	2	{0, 5, 53, 74, 300, 346, 414, 423} [15]
B.2	CDO	8	3	{0, 43, 139, 322, 422, 430, 441, 459} [5]
B.3	CDO	10	1	{0, 1, 5, 33, 543, 913, 1216, 1354, 1398, 1477} [15]
B.3	S-CDO	14	2	{0, 1, 4, 13, 32, 71, 156, 353, 827, 927, 1034, 1099, 1357, 1475} [15]
B.4	CDO	11	1	{0, 1, 5, 21, 72, 1388, 1569, 1809, 2109, 2423, 2559} [15]
B.4	S-CDO	15	2	{0, 1, 4, 13, 32, 71, 124, 218, 642, 1025, 1178, 1349, 1652, 1739, 2001} [15]
B.4	S-CDO	15	3	{0, 1, 5, 12, 32, 61, 107, 230, 355, 514, 824, 1424, 1726, 2384, 2653} [5]
B.5	S-CDO	10	1	{0, 6, 10, 34, 111, 130, 234, 267, 298, 309} [14]
B.5	S-CDO	10	2	{0, 7, 9, 83, 86, 118, 260, 296, 309, 317} [14]
B.5	S-CDO	11	3	{0, 2, 10, 17, 52, 108, 187, 323, 398, 434, 445} [14]
B.5	S-CDO	11	4	{0, 5, 8, 50, 123, 184, 303, 385, 399, 428, 448} [14]
B.6	CDO	9	1	{0, 2, 30, 108, 238, 537, 722, 763, 766} [14]
B.6	CDO	9	2	{0, 2, 24, 100, 428, 585, 667, 777, 792} [14]
B.6	S-CDO	12	3	{0, 8, 9, 32, 160, 300, 438, 530, 551, 605, 633, 639} [14]
B.6	S-CDO	12	4	{0, 6, 7, 16, 144, 270, 361, 470, 553, 583, 610, 648} [14]
B.7	CDO	10	1	{0, 1, 5, 99, 388, 789, 1128, 1359, 1401, 1428} [14]
B.7	CDO	10	2	{0, 1, 5, 96, 885, 1061, 1094, 1401, 1422, 1473} [14]
B.7	S-CDO	13	3	{0, 12, 13, 16, 34, 83, 164, 374, 564, 685, 791, 949, 990} [14]
B.7	S-CDO	13	4	{0, 1, 4, 13, 32, 168, 532, 584, 725, 795, 872, 926, 998} [14]
B.8	CDO	12	1	{0, 2, 5, 19, 63, 161, 1641, 2646, 3454, 3889, 4376, 4668} [14]
B.8	CDO	12	2	{0, 2, 5, 19, 63, 161, 1637, 2659, 3550, 3936, 4489, 4737} [14]
B.8	CDO	13	3	{0, 2, 5, 19, 63, 161, 365, 1553, 4016, 4553, 5658, 6789, 7785} [14]
B.8	CDO	13	4	{0, 2, 5, 19, 63, 161, 365, 1298, 4368, 4978, 5737, 7344, 7840} [14]

Table B.2 Code to Density Map Mapping (2 of 2)

Figure	Code Type	J	id	(S-)CDO Code
B.9	CDO	14	1	{0, 1, 5, 21, 55, 153, 368, 856, 2919, 6512, 7772, 10032, 11480, 12416} [14]
B.9	CDO	14	2	{0, 1, 5, 21, 55, 153, 368, 856, 2912, 7031, 8493, 10825, 11937, 12505} [14]
B.9	S-CDO	14	3	{0, 8, 9, 14, 35, 59, 248, 756, 855, 967, 1137, 1218, 1310, 1373} [14]
B.9	S-CDO	14	4	{0, 4, 5, 16, 30, 63, 172, 308, 746, 865, 952, 1212, 1312, 1377} [14]
B.10	CDO	15	1	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 4346, 10689, 13652, 16851, 19098, 20219} [14]
B.10	CDO	15	2	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3857, 8784, 13537, 16082, 18927, 20241} [14]
B.10	S-CDO	15	3	{0, 10, 11, 14, 37, 69, 108, 254, 636, 1040, 1181, 1379, 1631, 1801, 1890} [14]
B.10	S-CDO	15	4	{0, 3, 4, 13, 28, 64, 108, 235, 609, 782, 1142, 1430, 1635, 1785, 1942} [14]
B.11	CDO	16	1	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 5929, 13480, 20893, 22857, 29325, 31120} [14]
B.11	CDO	16	2	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 5566, 17437, 21413, 24642, 30654, 32618} [14]
B.11	S-CDO	16	3	{0, 11, 12, 15, 32, 71, 117, 228, 812, 1128, 1707, 1846, 2001, 2187, 2438, 2571} [14]
B.11	S-CDO	16	4	{0, 10, 11, 14, 37, 69, 108, 223, 481, 1078, 1256, 1659, 1866, 2247, 2409, 2580} [14]
B.12	CDO	17	1	{0, 17, 18, 22, 64, 177, 409, 739, 1605, 2597, 5277, 8375, 20438, 30617, 37767, 44012, 47231} [14]
B.12	CDO	17	2	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 8306, 15910, 27374, 36920, 45696, 48361} [14]
B.12	S-CDO	17	3	{0, 1, 4, 13, 32, 71, 124, 218, 375, 862, 1584, 2162, 2311, 2763, 2935, 3347, 3447} [14]
B.12	S-CDO	17	4	{0, 1, 4, 13, 32, 71, 124, 218, 375, 671, 1294, 1563, 2290, 2497, 3022, 3281, 3452} [14]
B.13	S-CDO	18	1	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 1027, 1419, 2193, 3112, 3565, 3824, 4299, 4565} [14]
B.13	S-CDO	18	2	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 1033, 2444, 2759, 3084, 3589, 3902, 4462, 4589} [14]
B.13	S-CDO	19	3	{0, 4, 5, 16, 30, 63, 128, 206, 358, 542, 787, 1163, 1878, 3260, 3532, 4524, 4811, 5731, 6046} [14]
B.13	S-CDO	19	4	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1763, 2429, 3620, 4137, 5419, 5690, 6390} [14]
B.13	S-CDO	20	5	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1162, 1682, 3065, 3517, 5250, 5602, 6167, 6861, 7177} [14]
B.13	S-CDO	20	6	{0, 8, 9, 14, 35, 59, 122, 213, 337, 484, 743, 1032, 1519, 2786, 3654, 5263, 5818, 6942, 7465, 7609} [14]

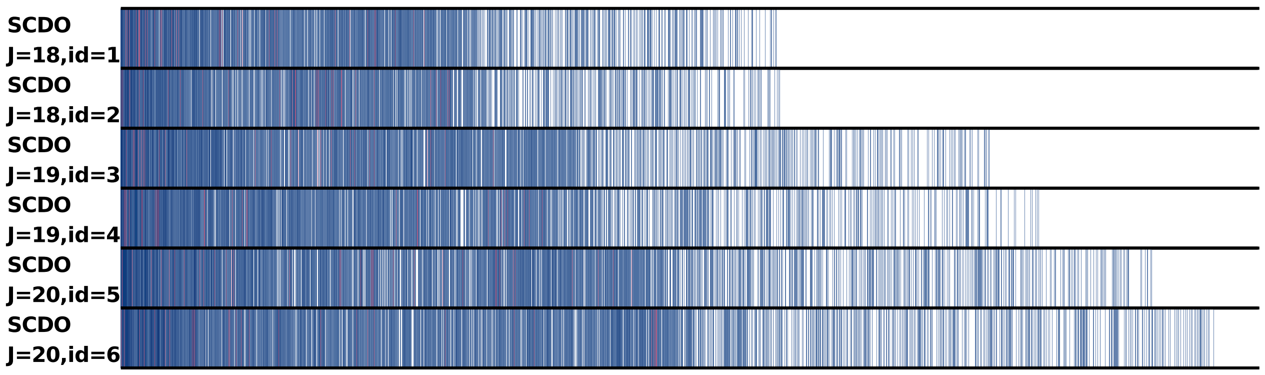


Figure B.13 Density map for four short-span S-CDO codes ($J \in \{18, 19, 20\}$).

APPENDIX C

SOME SHORT-SPAN CDO AND S-CDO CODES OF ORDER $J \leq 20$

In this appendix, we present novel short-span rate $R = \frac{1}{2}$ systematic (S-)CDO codes of order $J \in [7; 20]$. These codes were obtained while using the high-performance parallel algorithm described in this thesis to search for optimal-span (S-)CDO codes of corresponding order J. Therefore, their span is larger than the span of the codes of the same order published in [14, 15].

Tables C.1, C.2 and C.3 provide new short-span rate $R = \frac{1}{2}$ systematic CDO codes.

Tables C.4, C.5, C.6, C.7 and C.8 provide new short-span rate $R = \frac{1}{2}$ systematic S-CDO codes, as well as their respective simplification coefficient δ.

Table C.1 Short-span CDO codes of order $J \in \{7, 10, 11, 12, 13\}$

J	CDO codes
7	{0, 2, 22, 87, 188, 193, 221}
10	{0, 8, 9, 21, 124, 325, 929, 1297, 1444, 1492}
10	{0, 2, 5, 23, 207, 877, 976, 1328, 1458, 1490}
10	{0, 2, 5, 36, 152, 247, 782, 1105, 1455, 1475}
11	{0, 3, 5, 19, 58, 262, 1491, 1905, 2045, 2485, 2674}
11	{0, 2, 5, 19, 65, 894, 1580, 2035, 2322, 2562, 2662}
11	{0, 5, 6, 25, 91, 570, 1438, 1908, 2289, 2505, 2647}
11	{0, 5, 6, 25, 102, 1020, 1875, 2220, 2358, 2616, 2644}
11	{0, 6, 7, 23, 107, 542, 1556, 1803, 2392, 2472, 2614}
12	{0, 1, 5, 21, 55, 153, 391, 1766, 2889, 4236, 4917, 5132}
12	{0, 2, 5, 19, 63, 161, 414, 1936, 2846, 3642, 4420, 5107}
12	{0, 2, 5, 19, 63, 161, 463, 2026, 3209, 3822, 4673, 5023}
12	{0, 2, 5, 19, 63, 161, 526, 1773, 2613, 3945, 4650, 4882}
12	{0, 1, 5, 21, 55, 153, 624, 2054, 3072, 3574, 4433, 4839}
12	{0, 1, 5, 21, 55, 153, 631, 2209, 3142, 3573, 4578, 4816}
12	{0, 1, 5, 21, 55, 153, 853, 2676, 3186, 3859, 4452, 4797}
12	{0, 2, 5, 19, 63, 161, 1043, 2817, 3289, 4126, 4524, 4770}
12	{0, 1, 5, 21, 55, 153, 1266, 1985, 3313, 3710, 4284, 4762}
13	{0, 1, 5, 21, 55, 153, 368, 856, 1952, 5204, 5827, 8214, 9249}
13	{0, 1, 5, 21, 55, 153, 368, 856, 2192, 5381, 6855, 8651, 9231}
13	{0, 2, 5, 19, 63, 161, 365, 801, 2278, 5207, 6116, 7791, 9217}
13	{0, 1, 5, 21, 55, 153, 368, 856, 2555, 5700, 7080, 8016, 9128}
13	{0, 1, 5, 21, 55, 153, 368, 856, 2747, 5191, 7162, 8530, 9110}
13	{0, 1, 5, 21, 55, 153, 368, 856, 2767, 5138, 6074, 7387, 9020}
13	{0, 1, 5, 21, 55, 153, 368, 856, 2775, 4269, 6541, 7576, 8726}
13	{0, 2, 5, 19, 63, 161, 365, 801, 3709, 6008, 6898, 8169, 8723}
13	{0, 2, 5, 19, 63, 161, 365, 801, 3836, 5649, 6784, 8112, 8666}

Table C.2 Short-span CDO codes of order $J \in \{13, 14, 15\}$

J	CDO codes
13	{0, 2, 5, 19, 63, 161, 365, 815, 3427, 4802, 6159, 7999, 8640}
13	{0, 1, 5, 21, 55, 153, 368, 857, 4209, 5492, 6566, 7880, 8504}
13	{0, 1, 5, 21, 55, 153, 368, 898, 4558, 5173, 6902, 7404, 8472}
13	{0, 1, 5, 21, 55, 153, 368, 915, 2207, 4983, 6430, 7694, 8364}
13	{0, 2, 5, 19, 63, 161, 365, 991, 3671, 5824, 6523, 7725, 8259}
13	{0, 1, 5, 21, 55, 153, 368, 1038, 3651, 5435, 5965, 7558, 8149}
14	{0, 2, 5, 19, 63, 161, 365, 801, 1732, 5609, 8696, 10086, 12284, 13486}
14	{0, 1, 5, 21, 55, 153, 368, 856, 1935, 3248, 5883, 10655, 12055, 13483}
14	{0, 2, 5, 19, 63, 161, 365, 801, 1985, 7498, 8625, 10820, 11895, 13206}
14	{0, 1, 5, 21, 55, 153, 368, 856, 2035, 4484, 8218, 10008, 12484, 13107}
14	{0, 2, 5, 19, 63, 161, 365, 801, 2156, 6888, 8072, 9610, 12040, 13086}
14	{0, 1, 5, 21, 55, 153, 368, 856, 2382, 5910, 8898, 10377, 12041, 13026}
14	{0, 1, 5, 21, 55, 153, 368, 856, 2696, 6329, 8197, 10265, 11647, 12743}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 2603, 4967, 8194, 13663, 22432, 28169}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 7925, 13034, 18620, 27850}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 7925, 13416, 22252, 27506}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 2603, 4967, 8194, 15070, 22504, 26453}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 2603, 4967, 8194, 15773, 21891, 25840}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 8306, 14827, 22587, 25797}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 8306, 15529, 21546, 24756}
15	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3014, 4504, 9555, 15061, 22039, 24381}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 11168, 16961, 21130, 24340}
15	{0, 3, 5, 19, 58, 142, 364, 840, 1378, 3008, 5080, 9653, 17419, 19911, 23940}
15	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3014, 5255, 12744, 17078, 21031, 23533}

Table C.3 Short-span CDO codes of order $J \in \{15, 16, 17\}$

J	CDO codes
15	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3014, 5777, 12690, 16275, 20967, 23469}
15	{0, 3, 5, 19, 58, 142, 364, 840, 1378, 3008, 5637, 12212, 15861, 21297, 23195}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 2603, 5780, 10451, 15835, 18943, 22701}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 2603, 5830, 12718, 15082, 19265, 22407}
15	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 6514, 11178, 14659, 19454, 21826}
15	{0, 4, 5, 21, 61, 165, 393, 871, 1605, 3014, 7898, 12691, 15144, 19424, 21388}
15	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 3489, 9167, 13630, 16652, 18936, 21223}
15	{0, 8, 9, 21, 61, 160, 383, 860, 1787, 2944, 10140, 13251, 15468, 19874, 21081}
15	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 3765, 10039, 13046, 15809, 18749, 21030}
15	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 3885, 10403, 13112, 17316, 19793, 20977}
15	{0, 8, 9, 21, 61, 160, 383, 860, 1787, 3832, 11286, 13572, 16580, 19685, 20842}
15	{0, 1, 5, 21, 55, 153, 368, 856, 1424, 3252, 8937, 12628, 14429, 18837, 20805}
15	{0, 8, 9, 21, 61, 160, 383, 860, 1787, 4136, 10964, 13469, 16208, 19629, 20776}
15	{0, 6, 7, 23, 65, 151, 357, 805, 1729, 4281, 11520, 14596, 17343, 19468, 20573}
16	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 9824, 17604, 27296, 30506, 34675}
16	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 10039, 17107, 24858, 31699, 34364}
16	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 10902, 17518, 21186, 27355, 32935}
16	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 11273, 18260, 23646, 29663, 32873}
16	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 5566, 17024, 20384, 23613, 30818, 32782}
17	{0, 2, 5, 19, 63, 161, 365, 801, 1355, 2920, 4715, 7925, 22812, 31827, 37279, 42533, 49071}

Table C.4 Short-span S-CDO codes of order $J \in \{9, 10, 11, 12, 13, 14\}$

J	δ	S-CDO codes
9	0.4880	{0, 2, 5, 43, 94, 160, 182, 194, 212}
10	0.4957	{0, 16, 17, 22, 80, 200, 243, 278, 309, 333}
10	0.5014	{0, 12, 13, 20, 94, 153, 217, 272, 322, 325}
10	0.4908	{0, 8, 9, 30, 67, 99, 186, 300, 319, 325}
10	0.5063	{0, 1, 4, 30, 44, 94, 176, 253, 315, 324}
10	0.4986	{0, 5, 6, 45, 143, 208, 260, 290, 308, 322}
10	0.5043	{0, 3, 5, 16, 106, 137, 239, 262, 303, 320}
10	0.4928	{0, 2, 7, 41, 96, 200, 242, 293, 313, 319}
11	0.5195	{0, 4, 9, 37, 73, 252, 314, 365, 400, 448, 454}
12	0.4971	{0, 4, 5, 16, 123, 270, 423, 561, 594, 636, 650, 686}
12	0.5007	{0, 2, 5, 14, 172, 220, 386, 537, 574, 645, 665, 678}
12	0.5075	{0, 2, 5, 14, 191, 303, 407, 518, 577, 640, 653, 673}
12	0.5246	{0, 23, 24, 30, 110, 121, 228, 366, 521, 603, 647, 673}
12	0.5179	{0, 9, 10, 13, 114, 207, 363, 500, 546, 585, 619, 672}
12	0.5206	{0, 56, 57, 89, 99, 288, 292, 545, 593, 652, 664, 671}
12	0.5179	{0, 6, 7, 16, 120, 277, 423, 487, 523, 604, 625, 666}
12	0.5237	{0, 15, 16, 19, 136, 207, 365, 475, 533, 616, 625, 661}
13	0.4421	{0, 1, 4, 13, 32, 71, 163, 228, 336, 584, 697, 1126, 1428}
13	0.4979	{0, 1, 4, 13, 32, 104, 233, 407, 521, 717, 875, 964, 1012}
13	0.5024	{0, 1, 4, 13, 32, 150, 220, 316, 550, 733, 905, 960, 1009}
14	0.4343	{0, 1, 4, 13, 32, 71, 124, 224, 342, 510, 707, 1034, 1507, 2145}
14	0.4376	{0, 1, 4, 13, 32, 71, 124, 416, 503, 652, 929, 1228, 1547, 2035}
14	0.4362	{0, 1, 4, 13, 32, 71, 124, 363, 450, 617, 802, 1021, 1722, 1936}
14	0.4866	{0, 2, 5, 14, 27, 60, 146, 286, 395, 824, 1032, 1257, 1347, 1472}
14	0.4706	{0, 2, 5, 14, 27, 60, 146, 468, 825, 1039, 1129, 1225, 1395, 1470}
14	0.4796	{0, 9, 10, 13, 34, 66, 105, 509, 669, 883, 1038, 1245, 1419, 1470}
14	0.4747	{0, 8, 9, 14, 35, 59, 140, 612, 834, 978, 1081, 1286, 1374, 1466}

Table C.5 Short-span S-CDO codes of order $J \in \{14, 15, 16, 17\}$

J	δ	S-CDO codes
14	0.4804	{0, 10, 11, 14, 37, 69, 146, 521, 782, 958, 1139, 1241, 1371, 1460}
14	0.4861	{0, 3, 4, 13, 28, 64, 144, 486, 802, 896, 1069, 1183, 1362, 1444}
14	0.4838	{0, 7, 8, 18, 31, 58, 142, 367, 791, 863, 1105, 1259, 1315, 1435}
14	0.4900	{0, 10, 11, 14, 37, 69, 157, 456, 597, 976, 1083, 1246, 1322, 1434}
14	0.4902	{0, 3, 4, 13, 28, 64, 158, 400, 663, 893, 1118, 1162, 1355, 1433}
14	0.4833	{0, 1, 4, 13, 32, 71, 164, 549, 747, 927, 1127, 1215, 1368, 1421}
14	0.4897	{0, 3, 4, 13, 28, 64, 168, 514, 640, 966, 1059, 1197, 1330, 1411}
14	0.5069	{0, 9, 10, 13, 34, 66, 171, 482, 625, 893, 1014, 1149, 1221, 1410}
14	0.4974	{0, 10, 11, 14, 37, 69, 206, 381, 673, 877, 1088, 1182, 1301, 1408}
14	0.5026	{0, 5, 6, 14, 35, 67, 151, 542, 642, 935, 1053, 1130, 1303, 1406}
14	0.4945	{0, 2, 5, 14, 27, 60, 159, 424, 724, 805, 1081, 1209, 1309, 1398}
15	0.4433	{0, 1, 4, 13, 32, 71, 124, 304, 391, 561, 767, 1183, 1636, 2148, 2746}
15	0.4401	{0, 1, 4, 13, 32, 71, 124, 237, 324, 575, 834, 1201, 1586, 2205, 2686}
15	0.4451	{0, 1, 4, 13, 32, 71, 124, 349, 436, 549, 920, 1248, 1718, 2006, 2661}
15	0.4500	{0, 1, 4, 13, 32, 71, 124, 484, 571, 684, 833, 1469, 1686, 1892, 2627}
15	0.4816	{0, 6, 7, 16, 37, 70, 120, 222, 664, 835, 1197, 1510, 1659, 1920, 1998}
15	0.4863	{0, 5, 6, 14, 35, 67, 144, 249, 744, 833, 1180, 1525, 1700, 1787, 1989}
15	0.4744	{0, 11, 12, 15, 32, 71, 117, 228, 823, 1218, 1373, 1556, 1689, 1836, 1975}
15	0.4898	{0, 2, 5, 14, 27, 60, 135, 363, 563, 835, 1216, 1345, 1737, 1813, 1972}
16	0.4500	{0, 1, 4, 13, 32, 71, 124, 218, 425, 525, 872, 1100, 1569, 2111, 2781, 3638}
16	0.4421	{0, 1, 4, 13, 32, 71, 124, 218, 390, 477, 889, 1163, 1526, 2218, 2807, 3635}
16	0.4472	{0, 1, 4, 13, 32, 71, 124, 218, 398, 516, 688, 1263, 1517, 2256, 2683, 3603}
16	0.4727	{0, 1, 4, 13, 32, 71, 124, 218, 375, 770, 1249, 1704, 2055, 2262, 2548, 2719}
16	0.4691	{0, 1, 4, 13, 32, 71, 124, 218, 470, 1020, 1504, 1766, 1994, 2311, 2613, 2700}
16	0.5004	{0, 1, 4, 13, 32, 71, 124, 218, 486, 855, 1314, 1628, 1931, 2080, 2308, 2582}
17	0.4432	{0, 1, 4, 13, 32, 71, 124, 218, 386, 473, 881, 1220, 1741, 2278, 2892, 3612, 4715}
17	0.4502	{0, 1, 4, 13, 32, 71, 124, 218, 398, 516, 688, 1263, 1517, 2256, 2683, 3603, 4700}

Table C.6 Short-span S-CDO codes of order $J \in \{17, 18, 19\}$

J	δ	S-CDO codes
17	0.4526	{0, 1, 4, 13, 32, 71, 124, 218, 456, 605, 777, 864, 1738, 2077, 3101, 3781, 4365}
17	0.4587	{0, 1, 4, 13, 32, 71, 124, 218, 913, 1000, 1168, 1321, 1491, 1836, 3344, 3750, 4151}
17	0.4567	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 1333, 1939, 2442, 2965, 3226, 3495, 3666}
17	0.4758	{0, 3, 4, 13, 28, 64, 108, 234, 312, 589, 1271, 1715, 2429, 2576, 2855, 3264, 3625}
17	0.4734	{0, 12, 13, 16, 34, 61, 120, 198, 282, 511, 1311, 1733, 2221, 2629, 2974, 3282, 3621}
17	0.4675	{0, 3, 4, 13, 28, 64, 108, 234, 312, 589, 1454, 1859, 2208, 2636, 3047, 3444, 3591}
17	0.4705	{0, 7, 8, 18, 31, 58, 114, 206, 332, 585, 999, 1442, 2324, 2779, 3038, 3400, 3528}
18	0.4163	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4047, 5297, 6703}
18	0.4396	{0, 1, 4, 13, 32, 71, 124, 218, 375, 634, 805, 1159, 1461, 2201, 2753, 3793, 4774, 6469}
18	0.4439	{0, 1, 4, 13, 32, 71, 124, 218, 375, 694, 862, 1164, 1491, 2143, 2851, 4063, 4516, 6228}
18	0.4345	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1162, 1478, 2030, 2683, 3660, 5435, 6140}
18	0.4461	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 2878, 3773, 4958, 5920}
18	0.4492	{0, 1, 4, 13, 32, 71, 124, 218, 375, 1027, 1127, 1396, 1657, 1854, 3362, 3826, 4897, 5765}
18	0.4446	{0, 1, 4, 13, 32, 71, 124, 218, 375, 744, 912, 1208, 1556, 2285, 2899, 3883, 5280, 5729}
18	0.4307	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4690, 5214, 5640}
18	0.4279	{0, 8, 9, 14, 35, 59, 122, 213, 337, 484, 743, 1032, 1461, 2302, 3660, 4205, 5465, 5609}
18	0.4678	{0, 4, 5, 16, 30, 63, 128, 206, 358, 542, 787, 1163, 1781, 2656, 2971, 4000, 4287, 5207}
18	0.4725	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1705, 2475, 3199, 3897, 4327, 5077}
18	0.4753	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1897, 2250, 2904, 3746, 4449, 5029}
18	0.4534	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1278, 2693, 3046, 3401, 4335, 4606, 4946}
18	0.4626	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1343, 1671, 2753, 3510, 4274, 4626, 4871}
18	0.4540	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1494, 2521, 3209, 3628, 4193, 4545, 4846}
18	0.4663	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1705, 2110, 2754, 3620, 4005, 4383, 4802}
18	0.4650	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1631, 2269, 3032, 3451, 4113, 4429, 4703}
18	0.4789	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 899, 1630, 2088, 2480, 3492, 3663, 4309, 4657}
18	0.4584	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 947, 2225, 2759, 3087, 3661, 4184, 4500, 4627}
19	0.4336	{0, 1, 4, 13, 32, 71, 124, 218, 375, 634, 805, 1159, 1461, 2201, 2753, 3793, 4774, 6469, 8197}

Table C.7 Short-span S-CDO codes of order $J \in \{19, 20\}$

J	δ	S-CDO codes
19	0.4293	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4047, 5297, 6703, 7928}
19	0.4487	{0, 1, 4, 13, 32, 71, 124, 218, 375, 1027, 1127, 1396, 1657, 1854, 3362, 3826, 4897, 5765, 7904}
19	0.4289	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4047, 5297, 6703, 7838}
19	0.4455	{0, 1, 4, 13, 32, 71, 124, 218, 375, 694, 862, 1164, 1491, 2143, 2851, 4063, 4516, 6228, 7825}
19	0.4202	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4047, 5297, 7355, 7781}
19	0.4314	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3039, 4035, 5322, 6911, 7625}
19	0.4362	{0, 1, 4, 13, 32, 71, 124, 218, 375, 661, 832, 1344, 1739, 2041, 2969, 4130, 5306, 6802, 7584}
19	0.4321	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3039, 4035, 5612, 6695, 7409}
19	0.4369	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 809, 1134, 1524, 2391, 3031, 4342, 5513, 6479, 7201}
19	0.4561	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 2878, 4258, 4951, 5939, 7101}
19	0.4331	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 809, 1134, 1524, 2391, 3113, 4939, 5980, 6668, 6983}
19	0.4451	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3349, 4197, 5332, 6371, 6846}
19	0.4704	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1162, 1478, 2030, 3083, 3736, 5034, 5858, 6601}
19	0.4616	{0, 2, 5, 14, 27, 60, 135, 211, 372, 486, 888, 1162, 1478, 2486, 3145, 4404, 5001, 6056, 6408}
20	0.4224	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4047, 5297, 6703, 7838, 10986}
20	0.4506	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 1580, 1751, 2044, 2436, 3059, 4597, 5271, 6032, 6663, 10030}
20	0.4409	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 833, 1102, 1722, 1981, 2919, 3781, 5074, 5558, 8404, 9718}
20	0.4242	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 809, 1134, 1524, 2391, 3031, 4255, 5304, 6369, 9131, 9705}
20	0.4480	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 1088, 1188, 1614, 2290, 2904, 3607, 4983, 6121, 7687, 9686}
20	0.4266	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3039, 4035, 5322, 6405, 8917, 9631}
20	0.4371	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 2878, 3773, 4958, 6831, 8544, 9532}
20	0.4414	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 846, 1272, 1736, 2128, 2927, 4016, 4968, 6781, 8117, 9374}
20	0.4348	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3039, 4035, 5322, 6676, 8680, 9346}
20	0.4440	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 2878, 3773, 4958, 7113, 8275, 8973}
20	0.4400	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3039, 4035, 5805, 6519, 8302, 8968}
20	0.4371	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4472, 6307, 6634, 8365, 8791}
20	0.4297	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2441, 3097, 4499, 6116, 7709, 8233, 8659}

Table C.8 Short-span S-CDO codes of order $J \in \{20\}$

J	δ	S-CDO codes
20	0.4398	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 2878, 4834, 5557, 7314, 8315, 8452}
20	0.4457	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3052, 4603, 5610, 6850, 8095, 8448}
20	0.4505	{0, 3, 4, 13, 28, 64, 108, 234, 312, 565, 916, 1281, 1560, 2031, 3349, 4749, 6114, 7115, 7252, 8293}
20	0.4483	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2142, 3382, 4614, 5690, 7241, 7854, 8207}
20	0.4504	{0, 1, 4, 13, 32, 71, 124, 218, 375, 572, 744, 1208, 1556, 2445, 3446, 4924, 5963, 6577, 7748, 8174}
20	0.4534	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1442, 2178, 3746, 4742, 5714, 6797, 7473, 8127}
20	0.4577	{0, 5, 6, 14, 35, 67, 144, 228, 370, 629, 809, 1134, 1524, 2501, 4175, 4892, 5975, 6530, 7447, 7762}
20	0.4689	{0, 4, 5, 16, 30, 63, 128, 206, 358, 542, 787, 1163, 1805, 2423, 3373, 4255, 5611, 6238, 7330, 7757}
20	0.4601	{0, 6, 7, 16, 37, 70, 120, 222, 300, 519, 831, 1171, 1751, 3108, 3379, 4910, 5759, 6485, 7382, 7735}
20	0.4650	{0, 8, 9, 14, 35, 59, 122, 213, 337, 484, 743, 1032, 1519, 2121, 3287, 4868, 5297, 7033, 7177, 7700}

APPENDIX D

SAMPLE XML STATE-FILE

In this appendix, we present a sample *XML state-file* that was generated during the search for optimal-span $J = 10$ CDO codes by a binary that was configured to use two *worker threads*. The *decompressed*, *deserialized* and indented file contents are shown in Fig. D.1:

- Lines 3 to 8 hold information on the *binary* having created the state-file.
- Lines 10 to 14 hold information pertaining to the current *configuration* of the search.
- Lines 16 to 22 show the current *root stub* being used. The *root stub* is used to generate *tasks* for the *worker threads*.
- Lines 24 to 41 hold the *current tasks* being worked on by the two *worker threads*.

```xml
<?xml version=”1.0” encoding=”UTF-8”?>
<cpu_scdo>
    <version>1.0</version>
    <binary_date>Jun 24 2013</binary_date>
    <binary_time>21:33:47</binary_time>
    <binary_compiler>4.4.7 20120313 (Red Hat 4.4.7-3)</binary_compiler>
    <epoch>1372549478</epoch>
    <date_time>Sat Jun 29 19:44:38 2013</date_time>

    <type_of_code>cdo</type_of_code>
    <order_of_code>10</order_of_code>
    <main_rs_size>4</main_rs_size>
    <gen_rs_size>10</gen_rs_size>
    <best_span>1383</best_span>

    <main_rs_has_finished>0</main_rs_has_finished>
    <main_root_stub>
        <a0>0</a0>
        <a1>280</a1>
        <a2>430</a2>
        <a3>546</a3>
    </main_root_stub>

    <generators numb_gen=”2”>
        <gen id=”0” gen_finished=”0”>
            <gen_curr_mrs>
                <a0>0</a0>
                <a1>280</a1>
                <a2>430</a2>
                <a3>545</a3>
            </gen_curr_mrs>
        </gen>
        <gen id=”1” gen_finished=”0”>
            <gen_curr_mrs>
                <a0>0</a0>
                <a1>280</a1>
                <a2>430</a2>
                <a3>543</a3>
            </gen_curr_mrs>
        </gen>
    </generators>
</cpu_scdo>
```

Figure D.1 Sample *XML state-file* generated during the search for optimal-span $J = 10$ CDO codes by a binary configured to use two *worker threads*.