Groupe de travail Réseau
Request for Comments : 3284
Catégorie : En cours de normalisation
Traduction Claude Brière de L'Isle
juin 2002

D. Korn, AT&T Labs
J. MacDonald, UC Berkeley
J. Mogul, Hewlett-Packard Company
K. Vo, AT&T Labs

Format VCDIFF de différenciation et compression générique

Résumé
Le présent mémoire décrit VCDIFF, un format général, efficace et portable de données convenables pour le codage de données compressées et/ou différenciées de telle sorte qu'elles puissent être aisément transportées d'un ordinateur à l'autre.

Table des matières

1. Résumé

Les techniques de compression et de différenciation peuvent notablement améliorer le stockage et la transmission des fichiers et des versions de fichiers. Comme les fichiers sont souvent transportés à travers des machines qui ont des architectures et des caractéristiques de performances différentes, de telles données devraient être codées sous une forme portable et pouvoir être décodées avec peu ou pas du tout de connaissance des codages. Le présent document décrit Vcdiff,

un format compact et portable de codage conçu à cette fin.

La différenciation des données est le processus de calcul d'un codage compact et réversible d'un "fichier cible" étant donné un "fichier source". La compression des données est similaire, mais sans l'utilisation des données source. Les utilitaires UNIX diff, compress, et gzip sont des exemples bien connus d'outils de différenciation et de compression de données. Pour la différenciation des données, le codage calculé est appelé un "fichier delta", et pour la compression des données, il est appelé un "fichier compressé". Les fichiers delta et les fichiers compressés sont bons pour le stockage et pour la transmission car ils sont souvent plus petits que les originaux.

La différenciation et la compression des données sont traditionnellement traitées comme des types distincts de traitement de données. Cependant, comme l'ont montré Korn et Vo [1] pour la technique Vdelta, la compression peut être vue comme un cas particulier de différenciation dans laquelle les données de source sont vides. L'idée de base est d'unifier le schéma d'analyse de chaîne utilisé dans le compresseur de style Lempel- Ziv'77 (LZ'77) [2] et la technique de déplacement de bloc de Tichy [3]. En gros, cela fonctionne de la façon suivante :

a. Enchaîner les données de source et de cible.
b. Analyser les données de gauche à droite comme dans LZ'77 mais s'assurer qu'un segment analysé commence les données cible.
c. Commencer la sortie des résultats lorsque on atteint les données cible.

L'analyse se fonde sur des algorithmes de correspondance de chaîne, comme des arborescences de suffixes [4] ou de hachage avec des caractéristiques de performances de temps et d'espace différentes. Vdelta utilise un algorithme de correspondance de chaîne rapide qui exige moins de mémoire que les autres techniques [5], [6]. Cependant, même avec cet algorithme, les exigences de mémoire peuvent encore être prohibitives pour les gros fichiers. Une façon courante de pallier les limitations de mémoire est de partager un fichier d'entrée en tronçons appelés "fenêtres" et de les traiter séparément. Ici, sauf pour le travail non publié de Vo, il n'y a eu que peu de travaux de conception sur des schémas efficaces de fenêtrage. Les techniques actuelles, y compris Vdelta, utilisent simplement les fenêtres source et cibles avec les adresses correspondantes à travers les fichiers source et cible.

Les algorithmes de correspondance de chaîne et de fenêtrage ont une grande influence sur le taux de compression des deltas et des fichiers compressés. Cependant, il est souhaitable d'avoir un format de codage portable indépendant de ces algorithmes. Cela permet la construction d'applications client-serveur dans lesquelles un serveur peut traiter des clients qui ont des caractéristiques de calcul non connues. Malheureusement, tous les outils actuels de différenciation et de compression, y compris Vdelta, manquent cet objectif. Leurs formats de mémorisation sont étroitement liés aux algorithmes de correspondance de chaîne et/ou fenêtrage mis en œuvre.

Le format de codage Vcdiff proposé ici s'occupe des aspects mentionnés ci dessus. Vcdiff réalise les caractéristiques suivantes :

Résultat compact :
Le format de codage de base représente de façon compacte les fichiers compressés ou delta. Les applications peuvent étendre le format de codage de base avec des "codeurs secondaires" pour obtenir plus de compression.

Portabilité des données :
Le format de codage de base n'est pas soumis aux questions d'ordre des octets de la machine et de taille de mot. Cela permet de coder les données sur une machine et de les décoder sur une machine différente avec une architecture différente.

Algorithme générique :
L'algorithme de décodage est indépendant des algorithmes de correspondance de chaîne et de fenêtrage. Cela permet la compétition entre les mises en œuvre de codeur tout en gardant le même décodeur.

Efficacité du décodage :
Sauf pour les questions de codeur secondaire, l'algorithme de décodage fonctionne sur une durée proportionnelle à la taille du fichier cible et utilise un espace proportionnel à la taille de fenêtre maximale. Vcdiff diffère des compresseurs plus conventionnels en ce qu'il n'utilise que des données alignées sur l'octet, évitant par là les opérations au niveau du bit, ce qui améliore la vitesse de décodage au prix d'une légère perte d'efficacité de compression.

Cette méthode combinée de différenciation et de compression est appelée "compression des deltas" [14]. Comme cette façon de traiter les données fait de la compression un cas particulier de différenciation, on utilisera le terme de "fichier delta" pour indiquer le résultat compressé dans les deux cas.

2. Conventions

L'unité de données de base est l'octet. Pour la portabilité, Vcdiff devra limiter un octet à ses huit bits de moindre poids même sur les machines qui ont des octets plus grands. Les bits d'un octet sont ordonnés de droite à gauche de sorte que le bit de moindre poids (le LSB) a la valeur 1, et le bit de poids fort (le MSB) a la valeur 128.

Pour les besoins de l'exposé dans le présent document, on adopte la convention que le LSB est numéroté 0, et le MSB est numéroté 7. Les numéros de bits n'apparaissent jamais dans le format codé lui-même.

Vcdiff code les valeurs d'entier non signés en utilisant un format portable, de taille variable (originellement introduit dans la bibliothèque Sfio [7]). Ce codage traite un entier comme un nombre en base 128. Ensuite, chaque chiffre dans cette représentation est codé dans les sept bits de moindre poids d'un octet. Sauf pour l'octet de moindre poids, les autres octets ont leur bit de poids fort à un pour indiquer qu'il y a encore d'autres chiffres dans le codage. Les deux propriétés clés de ce codage d'entiers qui sont avantageuses pour un format de compression de données sont :

a. que le codage est portable parmi les systèmes qui utilisent des octets de 8 bits, et
b. que les petites valeurs ont un codage compact.

Par exemple, considérons la valeur 123 456 789, qui peut être représentée par quatre chiffres de 7 bits dont les valeurs sont 58, 111, 26, 21 rangées du plus fort poids au moindre poids. En dessous est le codage d'octets de 8 bits de ces nombres. Noter que les MSB de 58, 111 et 26 sont à 1.

```
+-----------------------------------------+
| 10111010 | 11101111 | 10011010 | 00010101 |
+-----------------------------------------+
  MSB+58     MSB+111    MSB+26     0+21
```

Dorénavant, les termes "octet" et "entier" se réfèreront à un octet et à un entier non signés comme décrit plus haut.

Les algorithmes en langage C sont affichés occasionnellement pour préciser les descriptions. Ce code C n'est destiné qu'à des fins de précision et ne fait pas partie de la spécification réelle du format Vcdiff.

Les mots clés "DOIT", "NE DOIT PAS", "EXIGE", "DEVRA", "NE DEVRA PAS", "DEVRAIT", "NE DEVRAIT PAS", "RECOMMANDE", "PEUT", et "FACULTATIF" dans le présent document sont à interpréter comme décrit dans le BCP 14, RFC2119 [12].

3. Instructions Delta

Un grand fichier cible est partitionné en sections non chevauchantes appelées "fenêtres cibles". Ces fenêtres cibles sont traitées séparément et en séquence sur la base de leur ordre dans le fichier cible.

Une fenêtre cible T, de longueur t, peut être comparée à un segment de données source S, de longueur s. Par construction, ce segment de données source S vient soit du fichier source, s'il en est utilisé un, soit d'une partie du fichier cible antérieure à T. De cette façon, durant le décodage, S est complètement connu lorsque T est décodé.

Le choix de T, t, S et s est fait par un algorithme de choix de fenêtre, qui peut affecter de façon notable la taille du codage. Cependant, comme on le verra plus loin, ces choix sont codés de sorte qu'aucune connaissance de l'algorithme de choix de fenêtre n'est nécessaire durant le décodage.

Supposons que S[j] représente le $j^{ème}$ octet dans S, et que T[k] représente le $k^{ème}$ octet dans T. Alors, pour les instructions de delta, on traitera les fenêtres de données S et T comme des sous-chaînes d'une super chaîne U, formée par des enchaînements comme celui-ci :

S[0]S[1]...S[s-1]T[0]T[1]...T[t-1]

L'"adresse" d'un octet dans S ou T est référencée par sa situation dans U. Par exemple, l'adresse de T[k] est s+k.

Les instructions pour coder et diriger la reconstruction d'une fenêtre cible sont appelées les instructions delta. Elles sont de trois types :

ADD : Cette instruction a deux arguments, une taille x et une séquence de x octets à copier.

COPY : Cette instruction a deux arguments, une taille x et une adresse p dans la chaîne U. Les arguments spécifient la sous chaîne de U qui doit être copiée. On dira qu'une telle sous chaîne doit être entièrement contenue dans S ou T.

RUN : Cette instruction a deux arguments, une taille x et un octet b, qui sera répété x fois.

Voici un exemple de fenêtres source et cible et les instructions delta qui codent la fenêtre cible en termes de fenêtre source.

```
a b c d e f g h i j k l m n o p
a b c d w x y z e f g h e f g h e f g h z z z z

COPY  4, 0
ADD   4, w x y z
COPY  4, 4
COPY 12, 24
RUN   4, z
```

Donc, la première lettre 'a' dans la fenêtre cible est à la position 16 dans la super chaîne. Noter que la quatrième instruction, "COPY 12, 24", copie les données de T lui-même car l'adresse 24 est en position 8 dans T. Cette instruction montre aussi que c'est bien de faire chevaucher les données à copier avec les données d'où elles sont copiées, pour autant que celles-là commencent avant. Cela permet un codage efficace des séquences périodiques, c'est-à-dire, des séquences avec des sous séquences qui se répètent régulièrement. L'instruction RUN est une façon compacte de coder une séquence qui répète le même octet même si une telle séquence peut être vue comme une séquence périodique de période 1.

Pour reconstruire la fenêtre cible, on traite simplement une instruction delta à la fois et on copie les données, soit de la fenêtre source, soit de la fenêtre cible qui est reconstruite, sur la base du type de l'instruction et de l'adresse associée, si il en est une.

4. Organisation de fichier Delta

Un fichier delta Vcdiff commence par une section d'en-tête suivie par une séquence de sections de fenêtres. La section d'en-tête comporte des octets magiques pour identifier le type de fichier, et des informations concernant le traitement des données au delà du format de codage de base. Les sections de fenêtre codent les fenêtres cibles.

Ci-dessous figure l'organisation globale d'un fichier delta. Les éléments en retrait précisent ceux qui sont immédiatement au-dessus d'eux. Un élément entre crochets peut être présent ou non dans le fichier selon les informations codées dans l'octet Indicateur au-dessus de lui.

```
En-tête
        En-tête1                                - octet
        En-tête2                                - octet
        En-tête3                                - octet
        En-tête4                                - octet
        Hdr_Indicator                           - octet
        [ID de compresseur secondaire]          - octet
        [Longueur des données de tableau de code] - entier
        [Données de tableau de code]
           Taille d'antémémoire proche          - octet
           Taille d'antémémoire same            - octet
           Données de tableau codé compressé
    Fenêtre1
        Win_Indicator                           - octet
        [Taille de segment source]              - entier
        [Position de segment source]            - entier
        Codage delta de la fenêtre cible
           Longueur du codage delta             - entier
           Codage delta
              Taille de la fenêtre cible        - entier
              Delta_Indicator                   - octet
              Longueur des données pour ADD et RUN - entier
              Longueur des instructions et tailles - entier
              Longueur des adresses pour COPY   - entier
              Section de données pour ADD et RUN - collection d'octets
              Section instructions et tailles   - collection d'octets
              Section adresses pour les COPY    - collection d'octets
```

Fenêtre2
...

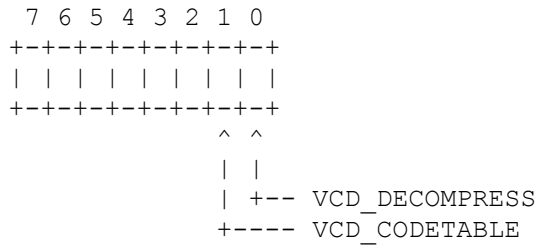
4.1 Section en-tête

Chaque fichier delta commence par une section d'en-tête organisée comme ci-dessous. Noter la convention que les crochets entourent des éléments facultatifs.

En-tête1	- octet = 0xD6
En-tête2	- octet = 0xC3
En-tête3	- octet = 0xC4
En-tête4	- octet
Hdr_Indicator	- octet
[ID de compresseur secondaire]	- octet
[Longueur des données du tableau de code]	- entier
[Données du tableau de code]	

Les trois premiers octets En-tête sont les caractères ASCII 'V', 'C' et 'D' avec leurs bits de poids fort établis (en hexadécimal, les valeurs sont 0xD6, 0xC3, et 0xC4). Le quatrième octet En-tête est actuellement réglé à zéro. À l'avenir, il pourrait être utilisé pour indiquer la version de Vcdiff.

L'octet Hdr_Indicator indique si des données d'initialisation sont requises pour aider à la reconstruction des données dans les sections de fenêtre. Cet octet PEUT avoir des valeurs différentes de zéro pour l'un des deux bits VCD_DECOMPRESS et VCD_CODETABLE ci-dessous, ou les deux ou ni l'un ni l'autre.



Si le bit 0 (VCD_DECOMPRESS) est différent de zéro, cela indique qu'un compresseur secondaire peut avoir été utilisé pour compresser encore plus certaines parties des données de codage de delta comme décrit au paragraphe 4.3 et à la section 6. Dans ce cas, l'identifiant du compresseur secondaire est donné ensuite. Si ce bit est zéro, l'octet d'identifiant de compresseur n'est pas inclus.

Si le bit 1 (VCD_CODETABLE) est différent de zéro, cela indique qu'un tableau de code défini par l'application est à utiliser pour décoder les instructions delta. Ce tableau est lui-même compressé. La longueur des données comprenant ce tableau de code compressé et les données suivent. La Section 7 expose les tableaux de code définis par l'application. Si ce bit est zéro, la longueur des données du tableau de code et les données du tableau de code ne sont pas incluses.

Si les deux bits sont établis, alors l'octet d'identifiant de compresseur est inclus avant la longueur des données du tableau de code et les données du tableau de code.

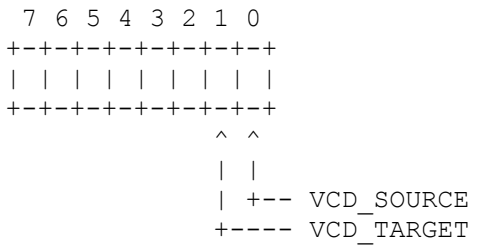
4.2 Format d'une section fenêtre

Chaque section fenêtre est organisée comme suit :

Win_Indicator	- octet
[Longueur du segment source]	- entier
[Position du segment source]	- entier
Le codage de delta de la fenêtre cible	

Ci-dessous figurent les détails des divers éléments :

Win_Indicator:
Cet octet est un ensemble de bits, comme suit :



Si le bit 0 (VCD_SOURCE) est différent de zéro, cela indique qu'un segment des données provenant du fichier "source" a été utilisé comme fenêtre de données source correspondante pour coder la fenêtre cible. Le décodeur va utiliser ce même segment de données source pour décoder la fenêtre cible.

Si le bit 1 (VCD_TARGET) est différent de zéro, cela indique qu'un segment de données provenant du fichier "cible" a été utilisé comme fenêtre de données source correspondante pour coder la fenêtre cible. Comme ci-dessus, ce même segment de données source est utilisé pour décoder la fenêtre cible.

L'octet Win_Indicator NE DOIT PAS avoir plus d'un des bits établis (non zéro). Il PEUT n'avoir aucun de ces bits établi.

Si un de ces bits est établi, l'octet est suivi par deux entiers pour indiquer, respectivement, la longueur et la position du segment de données source dans le fichier pertinent. Si l'octet indicateur est à zéro, la fenêtre cible a été compressée par elle-même sans se comparer à un autre segment de données, et ces deux entiers ne sont pas inclus.

Le codage de delta de la fenêtre cible :
Il contient le codage de delta de la fenêtre cible, soit en termes de segment de données source (c'est-à-dire, VCD_SOURCE ou VCD_TARGET a été établi) soit par lui-même si aucune fenêtre source n'est spécifiée. Ce format de données est exposé plus loin.

4.3 Codage de delta d'une fenêtre cible

Le codage de delta d'une fenêtre cible est organisé comme suit :

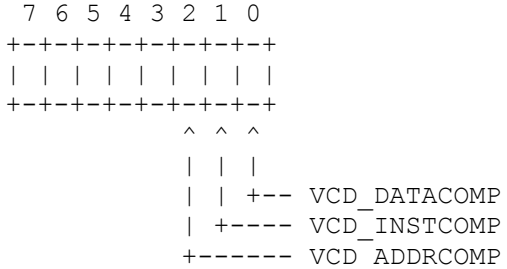
Longueur du codage de delta	- entier
Le codage de delta	
Longueur de la fenêtre cible	- entier
Delta_Indicator	- octet
Longueur des données pour les ADD et RUN	- entier
Longueur de la section instructions	- entier
Longueur des adresses pour COPY	- entier
Section de données pour ADD et RUN	- collection d'octets
Section instructions et tailles	- collection d'octets
Section adresses pour COPY	- collection d'octets

Longueur du codage de delta :
Cet entier donne le nombre total d'octets restants qui comprennent les données du codage de delta pour cette fenêtre cible.

Le codage de delta : Il contient les données qui représentent le codage de delta qui est décrit ensuite.

Longueur de la fenêtre cible :
Cet entier indique la taille réelle de la fenêtre cible après décompression. Un décodeur peut utiliser cette valeur pour allouer de la mémoire pour stocker les données non compressées.

Delta_Indicator : Cet octet est un ensemble de bits, comme indiqué ci-dessous :



VCD_DATACOMP : bit de valeur 1.
VCD_INSTCOMP : bit de valeur 2.
VCD_ADDRCOMP : bit de valeur 4.

Comme on l'a exposé plus haut, le codage de delta consiste en instructions COPY, ADD et RUN. Les instructions ADD et RUN ont des données d'accompagnement non confrontées (c'est-à-dire, des données qui ne correspondent spécifiquement à aucune donnée dans la fenêtre source ou dans quelque partie antérieure de la fenêtre cible) et les instructions COPY ont les adresses de l'endroit d'où provient la correspondance. FACULTATIVEMENT, ces types de données PEUVENT être encore compressés en utilisant un compresseur secondaire. Donc, Vcdiff sépare le codage des instructions delta en trois parties :

a. les données sans correspondance dans les instructions ADD et RUN,
b. les instructions delta et les tailles qui les accompagnent,
c. les adresses des instructions COPY.

Si le bit VCD_DECOMPRESS (paragraphe 4.1) était établi, chacune de ces sections peut avoir été compressée en utilisant le compresseur secondaire spécifié. Les positions de bit 0 (VCD_DATACOMP), 1 (VCD_INSTCOMP), et 2 (VCD_ADDRCOMP) indiquent respectivement, si elles sont différentes de zéro, que les parties correspondantes sont compressées. Ensuite, ces parties DOIVENT être décompressées avant de décoder les instructions delta.

Longueur des données pour les ADD et RUN :
C'est la longueur (en octets) de la section de données mémorisant les données sans correspondance qui accompagnent les instructions ADD et RUN.

Longueur de la section instructions :
C'est la longueur (en octets) des instructions delta et des tailles qui les accompagnent.

Longueur des adresses pour les COPY :
C'est la longueur (en octets) de la section qui mémorise les adresses des instructions COPY.

Section de données pour les ADD et RUN :
Cette séquence d'octets code les données sans correspondance pour les instructions ADD et RUN.

Section instructions et tailles :
Cette séquence d'octets code les instructions et leurs tailles.

Section Adresses pour les COPY :
Cette séquence d'octets code les adresses des instructions COPY.

5. Codages des instructions Delta

Les instructions delta décrites à la Section 3 représentent le résultat d'une confrontation de chaînes. Pour de nombreuses applications de différenciation de données dans lesquelles les changements entre les données de source et de cible sont faibles, toute représentation directe de ces instructions sera adéquate. Cependant, pour les applications qui comportent la différenciation de fichiers binaires ou de compression de données, il est important de bien coder ces instructions pour réaliser de bons taux de compression. La clé de cette réalisation est l'efficacité du codage des adresses des instructions COPY et la taille de toutes les instructions delta.

5.1 Modes de codage d'adresse des instructions COPY

Les adresses des instructions COPY sont des localisations de correspondances et surviennent souvent très proche ou même exactement égales les unes des autres. C'est parce que les données dans les régions locales sont souvent dupliquées avec des changements mineurs. À son tour, cela veut dire que coder une adresse qui vient de trouver sa correspondance par rapport à des adresses récemment confrontées peut être bénéfique. Pour tirer parti de ce phénomène et coder plus efficacement les adresses des instructions COPY, le format de données Vcdiff prend en charge l'utilisation de deux différents types d'antémémoires d'adresse. Le codeur et le décodeur tiennent tous deux ces antémémoires, de sorte que les antémémoires du décodeur restent synchronisées avec celles du codeur.

a. Une antémémoire "proche" est une collection d'intervalles "s_near", contenant chacun une adresse utilisée pour coder les adresses à proximité des adresses précédemment codées (seulement dans la direction positive). L'antémémoire proche tient aussi un indice "next_slot" sur l'antémémoire proche. Les nouvelles entrées dans l'antémémoire proche sont toujours insérées dans l'indice next_slot, qui entretient une mémoire tampon circulaire des s_near plus récentes adresses.

b. Une antémémoire "same" est une collection de "s_same", avec un multiple de 256 intervalles, contenant chacun une adresse. L'antémémoire same tient un tableau de hachage des adresses récentes utilisé pour répéter le codage de l'exacte même adresse.

Par défaut, les paramètres s_near et s_same sont réglés respectivement à 4 et 3. Un codeur PEUT modifier ces valeurs, mais il DOIT alors coder les nouvelles valeurs dans le codage lui-même, comme exposé à la Section 7, de sorte que le décodeur puisse établir correctement ses propres antémémoires.

Au début du traitement d'une fenêtre cible, une mise en œuvre (codeur ou décodeur) initialise à zéro tous les intervalles dans les deux antémémoires. Le pointeur next_slot de la prochaine antémémoire est réglé de façon à pointer sur l'intervalle zéro.

Chaque fois qu'une instruction COPY est traitée par le codeur ou décodeur, les antémémoires de la mise en œuvre sont mises à jour comme suit, où "addr" est l'adresse dans l'instruction COPY.

a. l'intervalle dans l'antémémoire proche référencée par l'indice next_slot est réglé à addr. L'indice next_slot est alors incrémenté modulo s_near.

b. L'intervalle dans l'antémémoire same dont l'indice est addr%(s_same*256) est réglé à addr. [On utilise les notations C de % pour modulo et * pour la multiplication.]

5.2 Exemple de code pour l'entretien des antémémoires

Pour rendre claire la description ci-dessus, voici des exemples de structures et algorithmes de données d'antémémoire pour les initialiser et les mettre à jour :

```
typedef struct _cache_s
{
   int*  near;                          /* collection de taille s_near       */
   int   s_near;
   int   next_slot;                     /* indice circulaire pour near */
   int*  same;                          /* collection de taille s_same*256    */
   int   s_same;
} Cache_t;

cache_init(Cache_t* ka)
{
   int   i;

   ka->next_slot = 0;
   for(i = 0; i < ka->s_near; ++i)
       ka->near[i] = 0;

   for(i = 0; i < ka->s_same*256; ++i)
       ka->same[i] = 0;
}

cache_update(Cache_t* ka, int addr)
{
   if(ka->s_near > 0)
   {   ka->near[ka->next_slot] = addr;
      ka->next_slot = (ka->next_slot + 1) % ka->s_near;
   }

   if(ka->s_same > 0)
      ka->same[addr % (ka->s_same*256)] = addr;
}
```

5.3 Codage des adresses d'instruction COPY

L'adresse d'une instruction COPY est codée en utilisant différents modes, selon le type d'adresse en antémémoire utilisé,

s'il en est.

Soit "addr" l'adresse d'une instruction COPY à décoder et soit "here" la localisation actuelle dans les données cible (c'est-à-dire, le début des données à coder ou décoder). Soit near[j] le $j^{ème}$ élément dans l'antémémoire proche, et soit same[k] le $k^{ème}$ élément dans l'antémémoire same. Voici les modes d'adresse possibles :

VCD_SELF : Ce mode a la valeur 0. L'adresse était codée par elle-même comme un entier.

VCD_HERE : Ce mode a la valeur 1. L'adresse était codée comme valeur d'entier "here - addr".

Modes proches : Les "modes proches" sont dans la gamme [2,s_near+1]. Soit m le mode du codage d'adresse. L'adresse a été codée comme la valeur d'entier "addr - near[m-2]".

Modes same : Les "modes same" sont dans la gamme [s_near+2,s_near+s_same+1]. Soit m le mode de codage. L'adresse a été codée comme un seul octet b tel que "addr == same[(m - (s_near+2))*256 + b]".

5.4 Exemple de code pour coder et décoder les adresses d'instruction COPY

On donne des exemples d'algorithmes pour montrer plus clairement l'utilisation des modes d'adresse. Le codeur a toute liberté pour choisir les modes d'adresse ; l'exemple d'algorithme addr_encode() montre une façon de prendre le mode d'adresse. L'algorithme de décodage addr_decode() va décoder de façon univoque les adresses, sans considération du choix de l'algorithme par le codeur.

Noter que les antémémoires d'adresse sont mise à jour immédiatement après qu'une adresse est codée ou décodée. De cette façon, le décodeur est toujours synchronisé avec le codeur.

```
int addr_encode(Cache_t* ka, int addr, int here, int* mode)
{
   int  i, d, bestd, bestm;

/* Les tentatives de découverte du mode d'adresse qui donnent la plus petite valeur d'entier pour "d", la valeur d'adresse
codée, minimisant ainsi la taille codée de l'adresse. */

   bestd = addr; bestm = VCD_SELF;                  /* VCD_SELF == 0 */

   if((d = here-addr) < bestd)
      { bestd = d; bestm = VCD_HERE; }              /* VCD_HERE == 1 */

   for(i = 0; i < ka->s_near; ++i)
      if((d = addr - ka->near[i])≥ 0 && d < bestd)
         { bestd = d; bestm = i+2; }

   if(ka->s_same > 0 && ka->same[d = addr%(ka->s_same*256)] == addr)
      { bestd = d%256; bestm = ka->s_near + 2 + d/256; }

   cache_update(ka,addr);

   *mode = bestm;                                   /* cela retourne le mode de codage d'adresse */
   return  bestd;                                   /* cela retourne l'adresse codée  */
}
```

Noter que l'algorithme addr_encode() choisit le meilleur mode d'adresse en utilisant une optimisation locale, mais cela peut ne pas conduire à la meilleure efficacité de codage parce que des modes différents conduisent à des codages d'instruction différents, comme on le décrit ci-dessous.

Les fonctions addrint() et addrbyte() utilisées dans addr_decode(), obtiennent de la "section d'adresses pour les COPY" (paragraphe 4.3) respectivement un entier ou un octet. Ces utilitaires ne seront pas décrits ici. On rappelle simplement qu'un entier est représenté par une chaîne d'octets compacte de taille variable, comme décrit à la Section 2 (c'est-à-dire, en base 128).

```
int addr_decode(Cache_t* ka, int here, int mode)
{   int  addr, m;
```

```
    if(mode == VCD_SELF)
        addr = addrint();
    else if(mode == VCD_HERE)
        addr = here - addrint();
    else if((m = mode - 2) >= 0 && m < ka->s_near)      /* antémémoire proche */
        addr = ka->near[m] + addrint();
    else /* same cache */
    {    m = mode - (2 + ka->s_near);
        addr = ka->same[m*256 + addrbyte()];
    }

    cache_update(ka, addr);

    return addr;
}
```

5.4 Codes d'instructions

Les correspondances sont souvent de courte longueur et séparées par de petites quantités de données sans correspondance. C'est-à-dire que les longueurs des instructions COPY et ADD sont souvent courtes. C'est particulièrement vrai des données binaires comme les fichiers exécutables ou les données structurées, telles que HTML ou XML. Dans de tels cas, la compression peut être améliorée en combinant le codage des tailles et les types d'instruction, ainsi qu'en combinant le codage des instructions delta adjacentes avec des tailles de données suffisamment petites. Les choix effectifs des moments où effectuer de telles combinaisons dépendent de nombreux facteurs incluant les données à traiter et l'algorithme de correspondance de chaîne utilisé. Par exemple, si de nombreuses instructions COPY ont la même taille de données, il peut valoir la peine de coder ces instructions de façon plus compacte que les autres.

Le format de données Vcdiff est conçu de telle sorte qu'un décodeur n'ait pas besoin de connaître les choix faits dans les algorithmes de codage. Cela se fait grâce à la notion d'un "tableau de code d'instruction", qui contient 256 entrées. Chaque entrée définit soit une seule instruction delta soit une paire d'instructions qui ont été combinées. Noter que le tableau de code lui-même n'existe que dans la mémoire principale, et non dans le fichier delta (sauf en utilisant un tableau de code défini par l'application, décrit à la Section 7). Les données codées incluent simplement l'indice de chaque instruction et, comme il n'y a que 256 indices, chaque indice peut être représenté par un seul octet.

Chaque entrée de code d'instruction contient six champs, chacun d'un seul octet avec une valeur non signée :

```
+-----------------------------------------------+
| inst1 |taille1| mode1 | inst2 |taille2| mode2 |
+-----------------------------------------------+
```

Chaque triplet (inst,taille,mode) définit une instruction delta. La signification de ces champs est la suivante :

inst : un champ "inst" peut avoir une de ces quatre valeurs : NOOP (0), ADD (1), RUN (2) ou COPY (3) pour indiquer les types d'instruction. NOOP signifie qu'aucune instruction n'est spécifiée. Dans ce cas, les champs de taille et de mode correspondants seront tous deux à zéro.

taille : un champ "taille" est zéro ou positif. Une valeur de zéro signifie que la taille associée à l'instruction est codée séparément comme un entier dans la "section instructions et tailles" (Section 6). Une valeur positive pour "taille" définit la taille réelle des données. Noter que comme la taille est restreinte à un octet, la valeur maximum pour toute instruction avec taille définie implicitement dans le tableau de code est 255.

mode : un champ "mode" n'est significatif que lorsque l'instruction delta associée est une COPY. Il définit le mode utilisé pour coder les adresses associées. Pour les autres instructions, c'est toujours zéro.

5.6 Tableau de code

À la suite de l'exposé sur les tableaux de code des modes et instruction d'adresses, on définit un "Tableau de code" comme ayant les données ci-dessous :

s_near : taille de l'antémémoire proche,
s_same : taille de l'antémémoire same,
i_code : tableau de code d'instruction à 256 entrées.

Vcdiff lui-même définit un "tableau de code par défaut" dans lequel s_near est 4 et s_same est 3. Donc, il y a 9 modes d'adresse pour une instruction COPY. Les deux premiers sont VCD_SELF (0) et VCD_HERE (1). Les modes 2, 3, 4 et 5 sont pour les adresses codées dans l'antémémoire proche. Et les modes 6, 7 et 8, sont pour les adresses codées dans l'antémémoire same.

	Type	Taille	Mode	Type	Taille	Mode	Indice
1.	RUN	0	0	NOOP	0	0	0
2.	ADD	0, [1,17]	0	NOOP	0	0	[1,18]
3.	COPY	0, [4,18]	0	NOOP	0	0	[19,34]
4.	COPY	0, [4,18]	1	NOOP	0	0	[35,50]
5.	COPY	0, [4,18]	2	NOOP	0	0	[51,66]
6.	COPY	0, [4,18]	3	NOOP	0	0	[67,82]
7.	COPY	0, [4,18]	4	NOOP	0	0	[83,98]
8.	COPY	0, [4,18]	5	NOOP	0	0	[99,114]
9.	COPY	0, [4,18]	6	NOOP	0	0	[115,130]
10.	COPY	0, [4,18]	7	NOOP	0	0	[131,146]
11.	COPY	0, [4,18]	8	NOOP	0	0	[147,162]
12.	ADD	[1,4]	0	COPY	[4,6]	0	[163,174]
13.	ADD	[1,4]	0	COPY	[4,6]	1	[175,186]
14.	ADD	[1,4]	0	COPY	[4,6]	2	[187,198]
15.	ADD	[1,4]	0	COPY	[4,6]	3	[199,210]
16.	ADD	[1,4]	0	COPY	[4,6]	4	[211,222]
17.	ADD	[1,4]	0	COPY	[4,6]	5	[223,234]
18.	ADD	[1,4]	0	COPY	4	6	[235,238]
19.	ADD	[1,4]	0	COPY	4	7	[239,242]
20.	ADD	[1,4]	0	COPY	4	8	[243,246]
21.	COPY	4	[0,8]	ADD	1	0	[247,255]

Le tableau de code d'instruction par défaut est décrit ci-dessus, dans une représentation compacte qui n'est utilisée que pour les besoins de la description. Voir à la section 7 la spécification de la façon de représenter un tableau de code d'instruction dans le format de codage Vcdiff. Dans la description, une valeur de zéro pour la taille indique qu'elle est codée séparément. Le mode des non instructions COPY est représenté par 0, bien qu'elles ne soient pas utilisées.

Dans la description, chaque ligne numérotée représente une ou plusieurs entrées dans le tableau réel de code d'instruction (on se rappelle qu'une entrée dans le tableau de code d'instruction peut représenter jusqu'à deux instructions delta combinées.) La dernière colonne ("Indice") montre quelle valeur d'indice, ou gamme de valeurs d'indice, des entrées est couverte par cette ligne. (La notation [i,j] signifie les valeurs de i à j, inclus.) Les six premières colonnes d'une ligne de la description décrivent les paires d'instructions utilisées pour la ou les valeurs d'indice correspondantes.

Si une ligne de la description inclut une entrée de colonne en utilisant la notation [i,j], cela signifie que la ligne est instanciée pour chaque valeur dans la gamme de i à j, inclus. La notation "0, [i,j]" signifie que la ligne est instanciée pour la valeur 0 et pour chaque valeur dans la gamme de i à j, inclus.

Si une ligne dans la description comporte plus d'une entrée utilisant la notation [i,j], impliquant une "boucle incorporée" pour convertir la ligne en une gamme d'entrées du tableau, la première de ces gammes [i,j] spécifie la boucle externe, et la seconde spécifie la boucle interne.

Les exemples ci-dessous devraient éclairer la description ci-dessus :

La ligne 1 montre la seule instruction RUN avec l'indice 0. Comme le champ de taille est 0, cette instruction RUN a toujours sa taille réelle codée séparément.

La ligne 2 montre les 18 instructions ADD seules. L'instruction ADD avec le champ de taille 0 (c'est-à-dire, la taille réelle est codée séparément) a l'indice 1. Les instructions ADD avec les tailles de 1 à 17 utilisent les indices de code 2 à 18 et leurs tailles sont celles données (donc elles ne seront pas codées séparément.)

À la suite des instructions ADD seules sont les instructions COPY seules ordonnées par mode de codage d'adresse. Par exemple, la ligne 11 montre les instructions COPY avec le mode 8, c'est-à-dire, le dernier de l'antémémoire same. Dans ce cas, l'instruction COPY avec le champ de taille 0 a l'indice 147. Là encore, la taille réelle de cette instruction sera codée séparément.

Les lignes 12 à 21 montrent les paires d'instructions qui sont combinées ensemble. Par exemple, la ligne 12 décrit l'entrée

12 dans laquelle une instruction ADD est combinée avec une instruction COPY qui suit immédiatement. Les entrées avec les indices 163, 164, 165 représentent les paires dans lesquelles les instructions ADD ont toutes la taille 1, alors que les instructions COPY ont le mode 0 (VCD_SELF) et respectivement les tailles 4, 5 et 6 .

La dernière ligne, la ligne 21, montre les huit paires d'instructions, où la première instruction est une COPY et la seconde est une ADD. Dans ce cas, toutes les instructions COPY ont une taille de 4 avec un mode allant de 0 à 8 et toutes les instructions ADD ont la taille 1. Donc, l'entrée avec le plus grand indice 255 combine une instruction COPY de taille 4 et mode 8 avec une instruction ADD de taille 1.

Le choix de la taille minimum de 4 pour les instructions COPY dans le tableau de code par défaut a été fait à partir d'expériences qui ont montré que l'exclusion des petites correspondances (de moins de quatre octets de long) améliorait les taux de compression.

6. Décodage d'une fenêtre cible

Le paragraphe 4.3 expose que les instructions delta et les données associées sont codées en trois collections d'octets :
section de données pour les ADD et les RUN,
section instructions et tailles,
section adresses pour les COPY.

De plus, ces sections de données peuvent avoir été encore plus compressées par un compresseur secondaire. En supposant que de telles données compressées aient été décompressées de sorte que nous ayons maintenant trois ensembles :
inst : octets codant les instructions et tailles,
data : données sans correspondance associées aux ADD et RUN,
addr : octets codant les adresses des COPY.

Ces ensembles sont organisés comme suit :
inst : une séquence de tuplets (indice, [taille1], [taille2]) où "indice" est un indice dans le tableau de code d'instruction, et taille1 et taille2 sont des entiers qui PEUVENT ou non être inclus dans le tuplet comme suit. L'entrée avec cet "indice" dans le tableau de code d'instruction définit potentiellement deux instructions delta. Si la première instruction delta n'est pas une VCD_NOOP et si sa taille est zéro, alors taille1 DOIT être présent. Autrement, taille1 DOIT être omis et la taille de l'instruction (si elle n'est pas VCD_NOOP) est ce qui est défini dans le tableau. La présence ou l'absence de taille2 est définie de façon similaire par rapport à la seconde instruction delta.
data : une séquence de valeurs de données, codées comme octets.
addr : une séquence de valeurs d'adresses. Les adresses sont normalement codées comme entiers comme décrit à la Section 2 (c'est-à-dire, en base 128). Cependant, comme l'antémémoire same émet des adresses dans la gamme [0, 255], les adresses de l'antémémoire same sont toujours codées comme un seul octet.

Pour résumer, chaque tuplet dans la collection "inst" comporte un indice d'une entrée dans le tableau de code d'instruction qui détermine :
a. si une ou deux instructions étaient codées et leurs types,
b. si les instructions avaient leur taille codée séparément, ces tailles vont suivre, dans l'ordre, dans le tuplet,
c. si les instructions ont des données d'accompagnement, c'est-à-dire, des ADD ou des RUN, leurs données seront dans la collection "données",
d. de même, si les instructions sont des COPY, les adresses codées se trouvent dans la collection "addr".

La procédure de décodage traite simplement les collections en lisant un indice de code à la fois, en recherchant l'entrée de code d'instruction correspondante, puis en "consommant" les tailles, données et adresses respectives suivant les directives de cette entrée. En d'autres termes, le décodeur entretient un pointeur implicite sur le prochain élément pour chaque collection ; "consommer" un tuplet d'instruction, données, ou valeur d'adresse implique d'incrémenter le pointeur associé.

Par exemple, si durant le traitement de la fenêtre cible, le prochain tuplet non consommé dans la collection inst a une valeur d'indice de 19, alors la première instruction est une COPY dont la taille est trouvée comme étant l'entier suivant immédiatement dans la collection inst. Comme le mode de cette instruction COPY est VCD_SELF, l'adresse correspondante est trouvée en consommant le prochain entier dans la collection addr. La collection données est laissée intacte. Comme la seconde instruction pour l'indice de code 19 est une NOOP, ce tuplet est terminé.

7. Tableaux de code défini par l'application

Bien que le tableau de code par défaut utilisé dans Vcdiff soit bon pour les codeurs d'utilisation générale, il y a des fois où

d'autres tableaux de code ont de meilleures performances. Par exemple, pour coder un fichier avec de nombreux segments de données identiques, il peut être avantageux d'avoir une instruction COPY avec la taille spécifique de ces segments de données, afin que l'instruction puisse être codée sur un seul octet. Un tel tableau de code particulier DOIT alors être codé dans le fichier delta afin que le décodeur puisse le reconstruire avant de décoder les données.

Vcdiff permet de spécifier un tableau de code défini par l'application dans un fichier de delta avec les données suivantes :

Taille de l'antémémoire proche - octet
Taille de l'antémémoire same - octet

Données compressées de tableau de code
Les "données compressées de tableau de code" codent le delta entre le tableau de code par défaut (source) et le nouveau tableau de code (cible) de la même manière que décrit au paragraphe 4.3 pour le codage d'une fenêtre cible en termes de fenêtre source. Ce delta est calculé en utilisant les étapes suivantes :

a. Convertir le nouveau tableau de code d'instruction en une chaîne "code" de 1536 octets en utilisant dans l'ordre les étapes suivantes :
   i. Ajouter dans l'ordre les 256 octets représentant les types des premières instructions dans les paires d'instructions.
   ii. Ajouter dans l'ordre les 256 octets représentant les types des secondes instructions dans les paires d'instructions.
   iii. Ajouter dans l'ordre les 256 octets représentant les tailles des premières instructions dans les paires d'instructions.
   iv. Ajouter dans l'ordre les 256 octets représentant les tailles des secondes instructions dans les paires d'instructions.
   v. Ajouter dans l'ordre les 256 octets représentant les modes des premières instructions dans les paires d'instructions.
   vi. Ajouter dans l'ordre les 256 octets représentant les modes des secondes instructions dans les paires d'instructions.
b. De façon similaire, convertir le tableau de code par défaut en une chaîne "dflt".
c. Traiter la chaîne "code" comme une fenêtre cible et "dflt" comme les données de source correspondantes et applique un algorithme de codage pour calculer le codage de delta de "code" en termes de "dflt". Ce calcul DOIT utiliser le tableau de code par défaut pour coder les instructions delta.

Le décodeur peut alors inverser les étapes ci-dessus pour décoder les données compressées du tableau en utilisant la méthode de la Section 6, en employant le tableau de code par défaut, pour générer le nouveau tableau de code. Noter que le décodeur n'a pas besoin de connaître les détails de l'algorithme de codage utilisé à l'étape (c). Il est capable de décoder le nouveau tableau de code parce que le format Vcdiff est indépendant du choix de l'algorithme de codage, et parce que le codeur à l'étape (c) utilise le tableau de code par défaut qui est connu.

8. Performances

Le format de codage est compact. Pour la seule compression, l'utilisation de la stratégie d'analyse de chaîne LZ-77 et sans aucun compresseur secondaire, le taux de compression typique est meilleur que la compression Unix et proche de gzip. Pour la différenciation, le format de données est meilleur que toutes les méthodes connues en terme d'objectif déclaré, qui est principalement la vitesse de décodage et l'efficacité du codage.

On compare les performances de compress, gzip et Vcdiff en utilisant les archives de trois versions du compilateur Gnu C, gcc-2.95.1.tar, gcc-2.95.2.tar et gcc-2.95.3.tar. Gzip a été utilisé à son niveau de compression par défaut. Les données Vcdiff ont été obtenues en utilisant le logiciel Vcodex/Vcdiff (Section 13).

Voici les différentes sessions de Vcdiff :

Vcdiff : vcdiff est utilisé seulement comme compresseur.

Vcdiff-d : vcdiff est utilisée seulement comme différenciateur. C'est-à-dire qu'il compare seulement les données cible aux données de source. Comme les fichiers impliqués sont gros, ils sont partagés en fenêtres. Dans ce cas, chaque fenêtre cible, commençant à un certain décalage de fichier dans le fichier cible, est comparé à une fenêtre source avec le même décalage de fichier (dans le fichier source). La fenêtre source est aussi légèrement plus grande que la fenêtre cible pour augmenter les opportunités de correspondance.

Vcdiff-dc : C'est similaire à Vcdiff-d, mais vcdiff peut aussi comparer des données cible à des données cible selon le cas applicable. Donc, vcdiff calcule à la fois les différences et compresse les données. L'algorithme de fenêtrage est le même que ci-dessus. Cependant, le conseil ci-dessus est annulé dans ce cas.

Vcdiff-dcw : C'est similaire à Vcdiff-dc mais l'algorithme de fenêtrage utilise une heuristique fondée sur le contenu pour choisir une fenêtre source qui aura une plus forte probabilité de correspondre à une certaine fenêtre cible. Donc, le segment de données source choisi pour une fenêtre cible sera souvent non aligné avec les décalages de fichier de cette fenêtre cible.

	gcc-2.95.1	gcc-2.95.2	gcc-2.95.3
1. taille brute	55 746 560	55 797 760	55 787 520
2. compressé	-	19 939 390	19 939 453
3. gzip	-	12 973 443	12 998 097
4. Vcdiff	-	15 358 786	15 371 737
5. Vcdiff-d	-	100 971	26 383 849
6. Vcdiff-dc	-	97 246	14 461 203
7. Vcdiff-dcw	-	256 445	1 248 543

Le tableau ci-dessus montre les tailles brutes des fichiers tar et les tailles des résultats compressés. Les résultats différenciés entre les colonnes gcc-2.95.2 ont été obtenus en compressant gcc-2.95.2 en partant de gcc-2.95.1. Les mêmes résultats pour la colonne gcc-2.95.3 ont été obtenus en compressant gcc-2.95.3 à partir de gcc-2.95.2.

Les rangées 2, 3 et 4 montrent que, pour la seule compression, le taux de compression à partir de Vcdiff est plus mauvais que gzip et meilleur que compress.

Les trois dernières rangées dans la colonne gcc-2.95.2 montrent que lorsque deux versions de fichier sont très similaires, la différenciation peut donner des taux de compression extrêmement bons. Vcdiff-d et Vcdiff-dc utilisent la même méthode simple de choix de fenêtre qui consiste à l'aligner les décalages de fichier, mais Vcdiff-dc fait aussi la compression de sorte que son résultat est légèrement plus petit. Vcdiff-dcw utilise un algorithme fondé sur le contenu pour chercher des données source qui vont vraisemblablement correspondre à une certaine fenêtre cible. Bien que cela donne de bons résultats, l'algorithme ne trouve pas toujours les meilleures correspondances qui, dans ce cas, sont données par le simple algorithme de Vcdiff-d. Il en résulte que la taille résultante pour Vcdiff-dcw est légèrement plus grande.

La situation est inversée dans la colonne gcc-2.95.3. Ici, les fichiers et leur contenu ont été suffisamment réarrangés ou changés entre la constitution de l'archive gcc-2.95.3.tar et l'archive gcc-2.95.2 de sorte que la méthode simple d'alignement des fenêtres par décalage de fichier ne fonctionne plus. Il en résulte que Vcdiff-d et Vcdiff-dc n'ont pas de bonnes performances. En permettant la compression, ainsi que la différenciation, Vcdiff-dc s'arrange pour battre Vcdiff-c, qui ne fait que la compression. L'algorithme de correspondance de fenêtre fondé sur le contenu dans Vcdiff-dcw est efficace pour faire correspondre les bonnes fenêtres source et cible de sorte que Vcdiff-dcw est globalement gagnant.

9. Questions en suspens

Quelques problèmes ne sont pas abordés par le présent document :

Compresseurs secondaires :
Comme exposé au paragraphe 4.3, certaines sections du codage de delta d'une fenêtre peuvent être encore compressées par un compresseur secondaire. À notre connaissance, le format Vcdiff de base est adéquat pour la plupart des objets de sorte que des compresseurs secondaires sont rarement nécessaires. En particulier, pour une utilisation normale de différenciation de données, où les fichiers à comparer ont de longues étendues de correspondance, une grande partie du gain en taux de compression est déjà obtenue par la correspondance de chaîne normale. Et donc, l'utilisation de compresseurs secondaires est rarement nécessaire dans ce cas. Cependant, pour des applications qui vont au delà de la différenciation de tels fichiers presque identiques, les compresseurs secondaires peuvent être nécessaires pour obtenir un résultat compressé maximal.

Donc, on recommande de laisser le format de données Vcdiff comme défini dans le présent document afin que l'utilisation de compresseurs secondaires puisse être mise en œuvre lorsque elle deviendra nécessaire à l'avenir. Les formats des données compressées via de tels compresseurs ou tout compresseur qui pourrait être défini à l'avenir sont laissés au choix de leur mise en œuvre. Cela pourrait inclure un codage de Huffman, un codage arithmétique, et un codage d'arborescence à évasement *(splay tree)* [8], [9].

Système de grands fichiers contre système de petite fichiers :
Comme exposé à la Section 4, une fenêtre cible dans un grand fichier peut être comparée à une fenêtre source dans un autre fichier ou dans le même fichier (à partir d'une partie antérieure). Dans ce cas, le décalage du fichier de la fenêtre source est spécifié comme un entier de taille variable dans le codage de delta. Il y a une possibilité que le codage ait été calculé sur un système qui prend en charge des fichiers beaucoup plus grands que dans un système où les données peuvent être décodées (par exemple, des systèmes de fichiers à 64 bits contre des systèmes de fichiers à 32 bits). Dans ce cas, certaines données cibles peuvent n'être pas récupérables. Ce problème pourrait affecter tout format de compression, et devrait être résolu par un mécanisme de négociation générique dans le ou les protocoles appropriés.

10. Résumé

Le présent document décrit Vcdiff, un format général et portable de codage pour les données compressées et différenciées. L'avantage de ce format est qu'il permet de mettre en œuvre un décodeur sans connaissance du codeur. De plus, ignorant l'utilisation de compresseurs secondaires non définis dans le format, les algorithmes de décodage fonctionnent en temps linéaire et exigent un espace de fonctionnement proportionnel à la taille de fenêtre.

11. Remerciements

Des remerciements sont dus à Balachander Krishnamurthy, Jeff Mogul et Arthur Van Hoff qui nous ont encouragé à publier Vcdiff. En particulier, Jeff a aidé à préciser la description du format de données présenté ici.

12. Considérations pour la sécurité

Vcdiff ne fait que fournir un format de codage de données compressées et différenciées. Il ne traite aucune question concernant la façon dont ces données sont, en fait, mémorisées dans un système de fichiers ou dans la mémoire de démarrage d'un système d'ordinateur. On ne prévoit donc aucun problème de sécurité à l'égard de Vcdiff.

13. Disponibilité du code source

Vcdiff est mis en œuvre comme méthode de transformation des données dans la bibliothèque Vcodes de Phong Vo. AT&T Corp. a rendu le code source de Vcodex disponible à tous pour l'utiliser à transmettre des données via le codage de données HTTP/1.1 [10], [11]. Le code source et l'accord de licence sont accessibles à l'URL suivant :

http://www.research.att.com/sw/tools

14. Considérations sur la propriété intellectuelle

Une revendication de droits de propriété intellectuelle a été notifiée à l'IETF à l'égard de tout ou partie de la spécification contenue dans le présent document. Pour d'autres informations, consulter la liste en ligne des revendications de droits à <http://www.ietf.org/ipr.html>.

L'IETF ne prend position sur la validité ou la portée d'aucun droit de propriété intellectuelle ou d'autres droits qui pourraient être revendiqués au titre de la mise en œuvre ou l'utilisation de la technologie décrite dans le présent document ou sur la mesure dans laquelle toute licence sur de tels droits pourrait être ou non disponible ; pas plus qu'elle ne prétend qu'elle ait fait aucun effort pour identifier de tels droits. Des informations sur les procédures de l'IETF au sujet des droits dans la documentation en cours de normalisation et en rapport avec les normes peuvent être trouvées dans le BCP-11. Des copies des revendications de droits peuvent être disponibles à la publication et toutes les assurances de licences peuvent être rendues disponibles, ou le résultat de tentatives d'obtention d'une licence ou permission générale pour l'utilisation de tels droits de propriété par les mises en œuvre ou utilisateurs de la présente spécification peuvent être obtenus auprès du secrétariat de l'IETF.

15. Considérations relatives à l'IANA

L'autorité d'allocation des numéros de l'Internet (IANA, *Internet Assigned Numbers Authority*) administre l'espace de nombres pour les valeurs d'identifiant de compresseur secondaire. Les valeurs et leur signification doivent être documentées dans une RFC ou autre référence à révision par les pairs, à disponibilité permanente et directe, avec un niveau de détail suffisant pour que l'interopérabilité entre des mises en œuvre indépendantes soit possible. Sous réserve de ces contraintes, l'allocation des noms est faite au premier qui les demande – voir la RFC2434 [13]. Les valeurs d'identifiant légales sont da,ns la gamme 1 à 255.

Le présent document ne définit aucune valeur dans cet espace de nombres.

16. Références

[1] D.G. Korn et K.P. Vo, "Vdelta: Differencing et Compression", Practical Reusable Unix Software, Editor B. Krishnamurthy, John Wiley & Sons, Inc., 1995.

[2] J. Ziv et A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Trans. on Information Theory, 23(3):337-343, 1977.

[3] W. Tichy, "The String-to-String Correction Problem with Block Moves", ACM Transactions on Computer Systems, 2(4):309-321, novembre 1984.

[4] E.M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm", Journal of the ACM, 23:262-272, 1976.

[5] J.J. Hunt, K.P. Vo, W. Tichy, "An Empirical Study of Delta Algorithms", IEEE Software Configuration and Maintenance Workshop, 1996.

[6] J.J. Hunt, K.P. Vo, W. Tichy, "Delta Algorithms: An Empirical Analysis", ACM Trans. on Software Engineering and Methodology, 7:192-214, 1998.

[7] D.G. Korn, K.P. Vo, "Sfio: A buffered I/O Library", Proc. of the Summer '91 Usenix Conference, 1991.

[8] D. W. Jones, "Application of Splay Trees to Data Compression", CACM, 31(8):996:1007.

[9] M. Nelson, J. Gailly, "The Data Compression Book", ISBN 1-55851- 434-1, M&T Books, New York, NY, 1995.

[10] J.C. Mogul, F. Douglis, A. Feldmann, et B. Krishnamurthy, "Potential benefits of codage de delta and data compression for HTTP", SIGCOMM '97, Cannes, France, 1997.

[11] Mogul, J., Krishnamurthy, B., Douglis, F., Feldmann, A., Goland, Y. et A. Van Hoff, "Codage de delta dans HTTP", RFC3229, janvier 2002.

[12] S. Bradner, "Mots clés à utiliser dans les RFC pour indiquer les niveaux d'exigence", RFC2119, BCP 14, mars 1997.

[13] T. Narten et H. Alvestrand, "Lignes directrices pour la rédaction d'une section Considérations relatives à l'IANA dans les RFC", RFC2434, BCP 26, octobre, 1998. *(Rendue obsolète par la RFC 5226)*

[14] D.G. Korn et K.P. Vo, "Engineering a Differencing and Compression Data Format", soumis à Usenix'2002, 2001.

17. Adresse des auteurs

Kiem-Phong Vo
AT&T Labs, Room D223
180 Park Avenue
Florham Park, NJ 07932
téléphone : 1 973 360 8630
mél : kpv@research.att.com

David G. Korn
AT&T Labs, Room D237
180 Park Avenue
Florham Park, NJ 07932
téléphone : 1 973 360 8602
mél : dgk@research.att.com

Jeffrey C. Mogul
Western Research Laboratory
Hewlett-Packard Company
1501 Page Mill Road, MS 1251
Palo Alto, California, 94304, U.S.A.
mél : JeffMogul@acm.org

Joshua P. MacDonald
Computer Science Division
University of California, Berkeley
345 Soda Hall
Berkeley, CA 94720
mél : jmacd@cs.berkeley.edu

18. Déclaration complète de droits de reproduction

Remerciement

Le financement de la fonction d'édition des RFC est actuellement fourni par l'Internet Society.