再帰的データ構造のためのキャッシュ内でのフィールド配列圧縮

高木将通[†] 平木敬[†]

再帰的構造体によるキャッシュミスを減らす手法，Field Array Compression Technique (FACT) を提案する．FACT はハードウェアとソフトウェアを用いた手法である．まずデータレイアウト変換によって，temporal affinity を持つデータを連続領域に配置する．次に再帰的ポインタと整数フィールドを 1/8 に圧縮する．これらによりキャッシュブロックのプリフェッチの効果を増す．さらに，実効的なキャッシュ容量を増す．再帰的構造体を用いる 8 個のプログラムにおいて，FACT は平均 41.6% のメモリ待ちサイクルの削減，平均 37.4% の速度向上を示した．

Field Array Compression in Data Caches for Recursive Data Structures

Masamichi Takagi[†] and Kei Hiraki[†]

We introduce a software/hardware scheme called Field Array Compression Technique (FACT) which reduces cache misses caused by recursive data structures. By the data layout transformation, FACT gathers data with temporal affinity in contiguous memory, and compresses the recursive pointer and the integer field there into 1/8. As a resut, FACT enhances the prefetching effect of a cache-block. In addition, the compression enlarges effective cache capacity. On a suite of pointer-intensive programs, FACT achieves 41.6% reduction of memory stall time and 37.4% speedup on average.

1. はじめに

非数値計算プログラムでは再帰的構造体 (Recursive Data Structures, **RDS**) は広く用いられている．RDS は，可変個の物体のリスト，space cell をあらわしたり探索に用いたりする木構造などのグラフを表現するために使われる．RDS を用いるプログラムはグラフを作ってからそれを辿る．この辿るコードはキャッシュミスを起こしやすい．主な原因は，グラフのノードが多くキャッシュに収まりきらないこと，ノードのキャッシュ上での配置が効率的でないことである．この問題に対処する方法として，プリフェッチが挙げられる[4]．もう一つの方法として，データレイアウト変換が挙げられる．この方法は temporal affinity のあるデータを連続領域に配置して，キャッシュブロックのプリフェッチの効果を増す[2]．さらに他の方法として，キャッシュ容量の増大が挙げられる．この方法はアクセス速度低下のために限界がある．一方格納するデータのサイズを縮小してキャッシュの実効容量を増大させる手法として，キャッシュにおけるデータ圧縮が提案されている[5)6)7]．この手法は実効容量を増大させるだけでなく，キャッシュブロックの実効サイズを増大させることができる．このため圧縮をデータレイアウト変換と組み合わせることによって，キャッシュブロックのプリフェッチの効果をレイアウト変換単体の適用時より更に増加させることができる．この組み合わせはプリフェッチと補い合う方法として利用できる．しかし従来のデータ圧縮手法では，キャッシュの構造が複雑になる問題により，圧縮率は $\frac{1}{2}$ に制限されている．本稿では，従来方法の限界 $\frac{1}{2}$ を超える圧縮率を実現し，データレイアウト変換の効果をより多く引き出す手法を提案する．

本稿では，Field Array Compression Technique (FACT) と名付ける手法を提案する．この手法では，RDS のデータレイアウト変換を，再帰的ポインタ・整数フィールドの圧縮と組み合わせることにより，キャッシュブロックのプリフェッチの効果を増す．また，キャッシュの実効容量を増す．これらにより，RDS によるキャッシュミスを減らす．また，キャッシュの構造を複雑にしないために，データレイアウト変換，メモリ上での圧縮データ配置の工夫，圧縮データをキャッシュ上で指定する方法の工夫，の三つを行う．これらにより，従来のデータ圧縮手法の限界 $\frac{1}{2}$ を超える $\frac{1}{8}$ の圧縮率を達成する．本稿の構成は以下の通りである．第 2 章で関連研究を述べ，第 3 章で FACT を説明し，第 4 章で評価方法を述べ，第 5 章で評価結果を論じ，第 6 章で結論を述べる．

2. 関連研究

いくつかの研究がキャッシュにおけるデータ圧縮を提案している．Yang らはハードウェアを用いる手法を提

† 東京大学大学院 情報理工学系研究科 コンピュータ科学専攻
Dept. of Computer Science, Graduate School of Information Science and Technology, University of Tokyo

案した[5]．我々の手法と異なりプログラムの変更を必要としない．1 キャッシュブロックを 32-bit word 単位で圧縮し，一次キャッシュに格納する．彼らの手法は，実行の全体を通じて，メモリアクセスにおいて頻繁に現れる少数の別々の値を探し，辞書を用いて圧縮する．我々の手法は整数フィールドの圧縮にこの方法を用いている．Larin らはハードウェアを用いる Yang らと同様の手法を提案した[6]．この方法は huffman code を用いて圧縮する．これらの手法では，圧縮後のデータをキャッシュ上で参照する際，圧縮前のデータのアドレスを用いるので，圧縮率が $\frac{1}{R}$ の場合，R 個のタグをチェックせねばならない．このため圧縮率を $\frac{1}{2}$ に制限している．FACT は，圧縮データをキャッシュ上で指定する方法の工夫によりこの問題を解決する．

Zhang らはソフトウェアとハードウェアを用いた，動的に割り当てられる構造体を圧縮する手法を提案した[7]．この手法では，構造体内に 1-word の slot を設ける．そして構造体のポインタと整数のフィールドの内，$\frac{1}{2}$ に圧縮できるもの 2 つをセットにしてその slot に収める．slot は構造体内部にあるため，圧縮しないフィールドのアライン要請により，slot は 1-word より小さくならない．また，slot に収めるデータは一つのインスタンスから集めねばならない．この二つの問題が圧縮率を制限する．FACT はデータレイアウト変換によりこの問題を解決する．また，彼らの手法では，プログラム実行時に，圧縮後データ用の領域のみを最初に割り当て，圧縮不可能データを見つけた時点で圧縮前データ用の領域を割り当てる．一方で我々の方法は圧縮前後両方の領域を最初に割り当てる．

Truong らは Instance Interleaving と名付ける，RDS のデータレイアウト変換法を提案した[2]．この変換は構造体の複数のインスタンスから同一フィールドを取り出し一列に並べる．これらのフィールドに temporal affinity がある際は，この変換はキャッシュブロックのプリフェッチの効果を高める．キャッシュにおけるデータ圧縮はキャッシュブロックの実効サイズを増大させるため，この変換の後に圧縮をかければ，プリフェッチの効果を更に高めることができる．このため我々は圧縮の前処理としてこの手法を用いている．

3. Field Array Compression Technique

FACT はデータレイアウト変換と構造体フィールドの圧縮により，RDS によるキャッシュミスを減らすことを目的とする．具体的な効果は以下の通り．まずデータレイアウト変換によって，temporal affinity を持つフィールドをメモリ上の連続領域に配置する．この変換により，キャッシュブロックのプリフェッチの効果を増大させる．本手法は更に構造体のフィールドを圧縮する．圧縮によるキャッシュブロックの実効サイズの増大が，temporal affinity を持つデータの連続領域への配置とあいまって，キャッシュブロックのプリフェッチの効果を更に増大させる．また，キャッシュの実効容量を増大させる．手順は以下の通り．

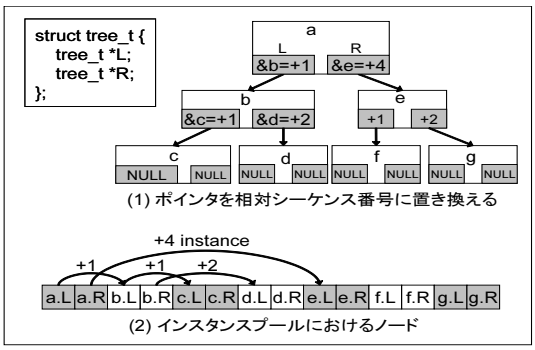

図 1 ポインタフィールドの圧縮.

(1) プロファイル実行を用いて，プログラム中の再帰的構造体の再帰的ポインタフィールドと整数フィールドが実行時にとる値を調べ，実行時に圧縮可能な値を多くとるフィールド (圧縮可能なフィールドと呼ぶ) を見つける．このフィールドが圧縮対象となる．

(2) ソースコードの変更により，圧縮対象構造体のデータレイアウトの変換を行う．この変換によって，複数のインスタンスから圧縮可能なフィールドを分離収集して，フィールドの配列を作る．

(3) 圧縮対象フィールドを参照する load/store 命令を圧縮復号作業を行うものに置き換える (それぞれ `cld`/`cst` と名付ける)．

(4) 実行時に `cld`/`cst` が通常の load/store 命令の動作に加えて，ハードウェアを用いた圧縮・復号の動作を行う．

フィールドの配列を用いるので，この圧縮法を Field Array Compression Technique (FACT) と名付ける．圧縮前後のデータ，圧縮に関係しないデータは全て同様の扱いを受け，同じキャッシュに格納される．本手法を実装するシステムは一次データキャッシュ，二次ユニファイドキャッシュをチップ内に持つと想定するので，圧縮後のデータはそれらに格納される．以下では FACT の各手順に関する課題について述べる．

3.1 構造体のフィールドの圧縮

FACT は，クリティカルパスで参照されることの多い再帰的ポインタフィールドを圧縮する．また，冗長性を持つことの多い整数フィールドを圧縮する．

3.1.1 シーケンス番号を用いたポインタの圧縮

RDS の再帰的ポインタは，同じ構造体のインスタンスしか指さない．また，RDS はしばしばまとめて割り当てられるため，ポインタでつながれる二つのインスタンス間のアドレス差は多くの場合小さい．このため，再帰的ポインタは，絶対アドレスよりも使用ビット幅の小さい，構造体単位の相対アドレスで置き換えることができる．相対アドレス計算のために，まず専用メモリアロケータを作成する．このアロケータの割り当てのステップは文献[1] と同様である．割り当て要求時

stdata	格納データ
base	ベースアドレス
1/R	圧縮率
incmp	圧縮不能を表す符号語(-2^(64/R-1))
nullcode	NULLを表す符号語(-2^(64/R-1)+1)

```
/* ポインタフィールド圧縮のアルゴリズム */
compress_ptr(stdata, base) {
   if(stdata == 0) { return nullcode }
   diff = (stdata – base)/8
   n = 64/R
   if(diff != nullcode &&  diff != incmp &&
      -2^(n-1) <= diff && diff <= 2^(n-1)-1) {
      return diff
   } else {
      return incmp
   }
}
```

図 2 ポインタフィールドの圧縮アルゴリズム.

に，もし利用可能なインスタンスがない場合は，インスタンスのプールを割り当て，そこから一つのインスタンスをプログラムに渡す．次にプログラムのソースコードを変更し，このアロケータを使用させる．圧縮作業は実行時に `cst` 命令が行う．

図 1 に圧縮の様子を示す．RDS を使って，深さ優先の順で完全二分木を作るとする (1)．インスタンスプールを用いるアロケータを使う際は，インスタンスはメモリ上で一列に並ぶ (2)．それゆえポインタを相対シーケンス番号で置き換えることができる (1)．

図 2 にアルゴリズムを示す．圧縮は `cst` 命令がポインタの格納の際に行う．格納先の構造体の先頭アドレスを `base`，格納するポインタを `stdata`，圧縮率を `1/R` とする．3.3.1 章に述べるデータレイアウト変換により，隣り合うインスタンスの先頭アドレスの差は 8-byte になるため，`(stdata-base)/8` を計算し，64/R-bit の符号語とする．`NULL` ポインタには特別な符号語を割り当てる．計算結果が圧縮表現が許す範囲を超えている際は，`cld` によって特別扱いされる，圧縮不能を示す符号語を用いる．`base` は `cst` 命令のベースアドレスから得られる．復号は `cld` 命令がポインタの読み出しの際に行う．読み出す構造体の先頭アドレス `base`，圧縮データ `lddata` として，`base+lddata`×8 を計算する．`base` は `cld` 命令のベースアドレスから得られる．

3.1.2 整数フィールドの圧縮

整数フィールドはしばしば冗長性を持つ．例えば，取る値に偏りのあるフィールドがある[5]．また使用しているビット幅が使用できる最大幅より狭いフィールドがある．FACT ではそれぞれの性質を使い，二つの手法を用いて圧縮する．

一番目の圧縮法は，実行の全体を通じて，頻繁に取る値の種類が少ない 32-bit の整数型のフィールドを探し，静的な辞書を用いて圧縮する[5]．エントリ数 N の辞書を作成する際は，プロファイル実行によって，全ての load/store 命令が参照する値の統計をとる．頻繁に参照される順に取った N 個の値を辞書とする．辞書の値は memory-mapped インターフェイスあるいは特殊レジスタを経てハードウェアの辞書に送る． 図

stdata	格納データ
base	ベースアドレス
1/R	圧縮率
incmp	圧縮不能を表す符号語(-2^(32/R-1))

```
/* 整数フィールド圧縮のアルゴリズム(辞書使用) */
compress_int_dict(stdata) {
   if(stdata in 辞書) {
      return 辞書のエントリ番号
   } else {
      return incmp
   }
}

/* 整数フィールド圧縮のアルゴリズム(狭ビット幅使用) */
compress_int_narrow(stdata) {
   n = 32/R
   if(stdata != incmp &&
     -2^(n-1) <= stdata && stdata <= 2^(n-1)-1) {
      return stdata
   } else {
      return incmp
   }
}
```

図 3 整数フィールドの圧縮アルゴリズム.

3 上部にアルゴリズムを示す．圧縮は `cst` 命令が整数フィールドの格納時に行う．圧縮率を $1/R$ とする．格納するデータを辞書内で探し，見つかった場合はエントリ番号を 32/R-bit の符号語とし，見つからない場合は圧縮不能を表す符号語を用いる．復号は `cld` 命令が整数フィールドの読み出し時に行う．

二番目の圧縮法は，使用しているビット幅が狭い整数フィールドを探して，より狭いビット幅の整数に置き換える．図 3 下部にアルゴリズムを示す．復号の際は，圧縮データを符号拡張する．

一番目の方法は復号の際にハードウェアの辞書を引く必要があり，辞書のエントリ数が大きいと復号に時間がかかる．このためエントリ数により一番目と二番目の方法を使い分ける．つまり，プログラム全体の圧縮率について，$\frac{1}{8}$ 及びそれより高い圧縮率を選んだ際には，プログラム全体で一番目の方法を用いる．それより低い圧縮率を選んだ際には，プログラム全体で二番目の方法を用いる．

3.2 圧縮対象構造体フィールドの選択

FACT ではまず，プログラム中の RDS のフィールドの内，圧縮可能なものを見つける．圧縮可能性は，フィールドの，動的に決定される値に依存するため，実行時の情報を得るために，本手法では以下のようなプロファイルの手法を用いる: プロファイル実行により，load/store 命令の実行時の統計を取る．プロファイル実行時の入力パラメタは実際の実行時のものとは異なるものを用いる．集められる情報は，アクセス回数，参照したデータが圧縮可能であった回数である．そして，アクセス回数の全体に対する割合が A より大きく，圧縮可能である率が B より大きいものをマークする．マークされた命令が参照するフィールドが圧縮対象となる．本稿における評価では経験的に $A = 0.1\%$, $B = 90\%$ と設定している．プロファイルは複数の圧縮率，$\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{16}$ について取っている．最後に，プロ

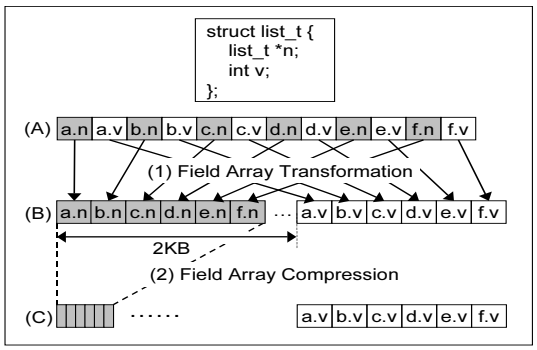

図 4 FAT は圧縮可能なフィールドを分離収集する．

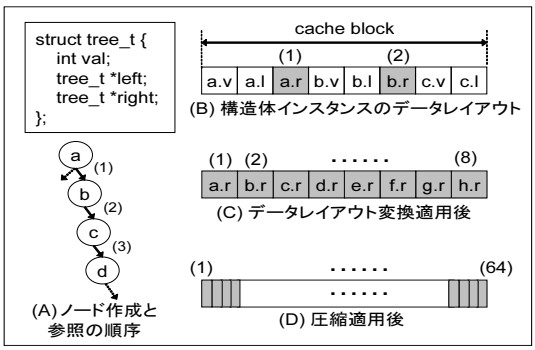

図 5 FAT は temporal affinity を持つフィールドを集め，FACT は圧縮により 1 キャッシュブロックにより多くのフィールドが収まるようにする．

ファイルに示される圧縮可能率によって一つの圧縮率を選ぶ．現時点では経験的に選んでいる．

3.3 圧縮に適したデータレイアウトへの変換

FACT では圧縮に適した形になるように RDS のデータレイアウト変換を行う．この変換は Truong らの提案した Instance Interleaving[2] と同じである．変換作業は現時点では手で行われている．変換によって，同一フィールドの配列ができるため，この変換を便宜的に Field Array Transformation (FAT) と名付ける．

3.3.1 FAT による圧縮可能フィールドの分離収集

FACT では，RDS の再帰的ポインタと整数フィールドを圧縮する．圧縮後のデータは圧縮により位置がずれてしまうため，キャッシュ上で参照する際は，圧縮前のアドレスを，キャッシュ上で圧縮後のデータを指せるアドレスへ変換する必要がある．キャッシュ上の圧縮データに対して専用のアドレス空間を用意すれば，アドレスもデータと同じ率で縮小することでこの変換が行える．ここで圧縮対象構造体のインスタンスのアドレスを I，圧縮率を $1/R$ とすると，変換後のアドレスは I/R となる．ところがこの計算に必要な R は構造体ごとに変わる． このため RDS のデータレイアウトの変換を行い，圧縮可能なフィールドを分離収集する．例として，圧縮可能なポインタ `n` と，圧縮不能な整数 `v` を持つ構造体を圧縮するとする．図 4 に分離の様子を示す．この例ではフィールド `n` が圧縮可能なので，複数のインスタンスから `n` だけを取り出し，メモリ上で一列に並べる．このようにして，一つのセグメントを `n` で埋め，次のセグメントを `v` で埋め，`n` を配列として分離する (B)．多くの場合，全ての要素を $\frac{1}{8}$ に圧縮できる．このとき配列全体を圧縮できる (C)．さらに，n に対するアドレス変換はシフト演算だけですむ $\times\frac{1}{8}$ になる．

3.3.2 FAT による temporal affinity の利用

データレイアウト変換により，フィールド間の temporal affinity が利用できる．図 5 に例を示す．評価に用いているプログラム `treeadd` の木構造を考える．構造体の宣言は図中に示されている．`treeadd` は二分木を，`right` を先にした深さ優先順で作る (A)．それゆえメモリ上でも木のノードは深さ優先順に並んでいる (B)．`treeadd` はその後同じ順序で木を辿る．このため，メモリ上で連続する二つの `right` ポインタには temporal affinity がある．キャッシュのブロックサイズを 64-byte とすると，FAT なしでは，インスタンスのサイズは 24-byte なので，1 キャッシュブロックに temporal affinity を有する 2 つの `right` ポインタが入る (B)．FAT のみの場合，これは 8 になり (C)，更に $\frac{1}{8}$ の圧縮を加えた場合，64 に増大する (D)．この変換は，キャッシュブロックのプリフェッチの効果を高める．

3.3.3 FAT の実装

まず，構造体の宣言部を変更して，フィールド間に pad を挿入する．図 6 に例を示す．load/store 命令のアドレッシングは 64-bit レジスタと符号付 16-bit 即値オフセットを用いるとする．コンパイラは，オフセットが 32KB までのフィールドの参照の際は，構造体の先頭アドレスをベースアドレスにしたアドレッシングを採用する．FACT ではポインタの圧縮にこの性質を利用している．pad の挿入によって，構造体内の n 番目のフィールドのアドレスは (構造体の先頭アドレス)+(pad のサイズ)×$(n-1)$ となる．このため pad のサイズを 2KB に制限して，16 番目までのフィールドの参照の際はベースアドレスが構造体の先頭アドレスになるようにする．図 6 にこの pad を挿入した構造体を利用した割当の様子を示す．ある構造体に対して初めて割り当てる際，変更された構造体を割り当て，そのアドレス (A とする) を返す (1)．次の割り当て要求時には，$A+8$ を返して，二番目のインスタンスのフィールドを使って pad を再利用する (2)．

次に，この割り当てを実現する専用アロケータを用意し，それを使うようプログラムを変更する．このアロケータはメモリブロック (アリーナと呼ぶ) を割り当て，インスタンスプールに使う[1]．引数として構造体の型の ID を取り，ID ごとにアリーナを管理する．アリーナは数百インスタンス程度しか格納できないため，必要に応じて追加で割り当てていく．圧縮を行う際は，圧縮率を $1/R$ とすると，アリーナを $1:R$ で圧縮後の領域，圧縮前の領域に分けて使う．

評価時の baseline となる構成にも文献[1] と同様のインスタンスプールを用いたアロケータを用いる．これ

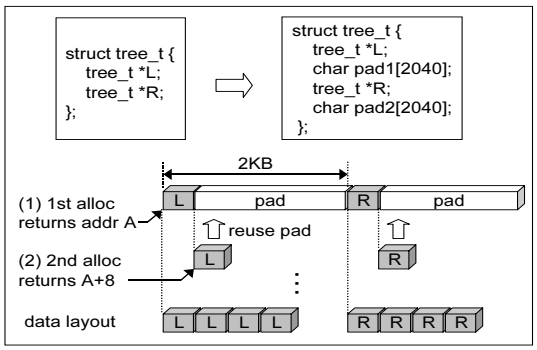

図 6 pad を挿入した構造体による FAT の様子.

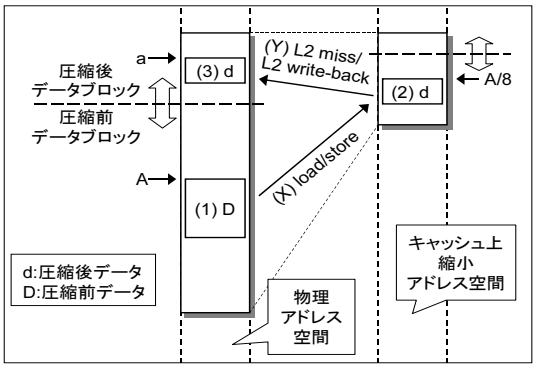

図 7 FACT におけるアドレス変換の様子.

stdata	格納データ
base	ベースアドレス
offset	オフセット
1/R	圧縮率
incmp	圧縮不能を表す符号語

```
/* cst の動作 */
cst(stdata, offset, base) {
    pa = (base + offset)を物理アドレスに変換したもの
    ca = pa/R
    cdata = cstの種類に応じて
            compress_ptr(stdata, base)等を呼ぶ
    cache_write(ca, cdata, C)
    if(cdata == incmp) { /* stdataが圧縮不可能 */
        cache_write(pa, stdata, N)
    }
}
```

図 8 cst 命令の動作.

は，この変更だけで速度向上が得られたためである．

3.4 圧縮データの配置とキャッシュ上での指定

圧縮を試みるデータは，動的に変わるため，常に圧縮可能とは限らない．圧縮が不可能な際は，圧縮前のデータ用の場所が必要となる．これに対処する方法には大きく分けて二つある．一つは，圧縮後データ用の領域のみを最初に割り当て，圧縮不可能な際に新たに割り当てを行うもの[5]，もう一つは最初から圧縮前と圧縮後両方の領域を確保するものである．一つ目の方法は圧縮前と圧縮後のデータのアドレスの関係を複雑にするので，FACT では二つ目の方法を用いる．この方法では，主記憶上の圧縮データを参照する際に，圧縮前のアドレスから，圧縮後のアドレスへの変換作業が必要になる．FACT では，以下のような方法を用い，変換を線型変換で済ます．圧縮率を $1/R$ とする．圧縮対象構造体のために専用アロケータを用い，一定サイズのブロックを割り当て，そのブロックを 1:R の割合で圧縮後・圧縮前データ用の領域に分ける．この配置においては，圧縮後データのブロックは圧縮前データのブロックを縮小した形を持つ．また圧縮前データを符号語で置き換えて表現する．このため密集させた圧縮後データのみを頻繁に参照させることができる．

圧縮後のデータを d，その物理アドレスを a，圧縮前のデータを D，その物理アドレスを A とし，圧縮率を $1/R$ とする．キャッシュ上で a を使って D を指すと，キャッシュの $1/(R+1)$ しか利用できなくなる．一方 A を使って D を指すと，$R/(R+1)$ を使うことができる．そこで FACT ではキャッシュ上の圧縮後データに新たなアドレス空間を用意し，その空間のアドレス A/R を使って d を指す．この新たなアドレス空間を縮小アドレス空間と呼ぶ．A/R を得るには A をシフトするだけでよい．各キャッシュには，縮小アドレス空間と通常のアドレス空間を区別するための 1-bit のタグを追加する．圧縮後データの主記憶への write-back，または主記憶からの fetch の際は，a を使って d を指す必要があるため，A/R を a に変換する．これは計算で行える．この計算による遅延の実行時間への影響は評価に使用した全プログラムで 1%以下である．

図 7 を用いて，具体的なアドレス変換を説明する．物理アドレス A から始まるデータ (1) を 1/8 に圧縮するとする．圧縮後のデータは主記憶の物理アドレス a(3) に格納される．`cld`/`cst` 命令は，$A/8$ を使ってキャッシュ内の圧縮後のデータにアクセスする (2)(X)．圧縮後のデータの主記憶への write-back，または主記憶からのフェッチの際は，$A/8$ を a に変換する (Y)．

3.5 `cld`/`cst` 命令の適用と動作

圧縮対象フィールドを参照する load/store 命令は圧縮作業を行う `cld`/`cst` 命令に置き換えられ，プログラム動作中に，通常の動作に加えて圧縮・復号に必要な動作をする．圧縮対象は再帰的ポインタフィールドと整数フィールドであり，整数フィールドの圧縮方法には二種類ある．これに対応して `cld`/`cst` 命令はそれぞれ三種類存在し，置き換えの際に，圧縮対象のフィールドの型と圧縮方法に応じて一種類を選ぶ．整数圧縮の方法は二種類あるが，一種類のみをプログラム全体で使う．具体的には，プログラム全体で用いる圧縮率が 1/4 より低い際は，狭ビット幅を利用する整数圧縮方法を用い，それ以外の圧縮率の際は辞書を利用する整数圧縮方法を用いる．

`cst` 命令の動作を図 8 に，`cst` 命令のキャッシュに対する動作を図 9 に示す．簡単のために単一レベルのキャッシュ階層を想定した動作を示してある．また，物理インデックス・物理タグを用いるキャッシュを想定している．`cst` 命令は，格納するデータが圧縮可能であるか調べ，可能なら圧縮する．圧縮後のデータはアドレス変換を経てキャッシュに格納される．この変換は，アドレスを圧縮率と同率で縮める．圧縮不能なデータ

addr	キャッシュ上でのアドレス
data	格納データ
flag	C:圧縮後データ、N:圧縮前データを示すフラグ

```
/* cstのキャッシュ書き込み時の動作 */
cache_write(addr, data, flag) {
   if({addr, flag}でミス) {
      if(flag == C) {
         pa = addrから主記憶上の圧縮後データの
              アドレスを算出
         アドレスpaで主記憶参照、キャッシュフィル
      } else {
         アドレスaddrで主記憶参照、キャッシュフィル
      }
   }
   アドレスaddrにdata格納
}
```

図 9 cst 命令のキャッシュに対する動作.

を扱う際は，圧縮不能を表す符号語を格納し，その後圧縮前のデータを変換前のアドレスに格納する．圧縮後のデータがキャッシュミスした際には，アドレス変換を経て主記憶をアクセスする．このアドレス変換は，圧縮後のデータのキャッシュ上のアドレスを圧縮前のデータの主記憶上のアドレスに変換する．

`cld` 命令の動作を図 10 に示す．圧縮後のデータを参照する際は，アドレス変換を経てキャッシュを参照し，復号する．圧縮後のデータが圧縮不能を示す符号語である際は，圧縮前のデータに，変換前のアドレスを使ってアクセスする．圧縮後のデータがキャッシュミスした際には，`cst` 命令と同様，アドレス変換を経て主記憶をアクセスする．

4. 評価方法

FACT はスーパースケーラの 64-bit プロセッサを用いるとする．我々はイベント駆動ソフトウェアシミュレータを作成し評価に用いた．表 1 にそのパラメタを示す．命令の latency と issue rate は Compaq Alpha 21264 と同じとする．パイプラインは 7 段構成とし，load-to-use latency は 3 サイクルとする．`cst` 命令の圧縮作業，`cld`/`cst` 命令が圧縮データを参照する際のアドレス縮小作業は命令に遅延を追加しないとする．`cld` 命令の復号作業は load-to-use レイテンシに 1 サイクルを加えるとする．`cld`/`cst` 命令が圧縮不可能データを扱う際のペナルティは，最低で，それぞれ 6，4 サイクルとする．また，整数圧縮用のハードウェア辞書へのデータ転送のオーバーヘッドは考慮しない．

評価には，RDS を用い，メモリデータ待ち時間の実行時間に対する率が高いプログラム 8 個を用いる．それらは olden benchmark[3] のプログラム `health`, `treeadd`, `perimeter`, `tsp`, `em3d`, `bh`, `mst`, `bisort` である．プロファイル実行用のパラメタ，評価用のパラメタ，プログラムの特徴を表 2 に示す．全プログラムは Compaq C Compiler version 6.2-504 で Alpha の Linux 上でコンパイルされた．最適化オプションは-O4 を用いた．

☆ `perimeter` は 4K×4K でなく 16K×16K の画像を用いるように変更された.

☆☆ `bh` のタイムステップは 10 から 4 に変更された.

base	ベースアドレス
offset	オフセット
1/R	圧縮率
incmp	圧縮不能を表す符号語

```
/* cld の動作 */
cld(offset, base) {
   pa = (base + offset)を物理アドレスに変換したもの
   ca = pa/R
   memdata = cache_read(ca, C)
   if(memdata != incmp) {
      dst = cldの種類に応じてmemdataを復号
      return dst
   } else {
      memdata  = cache_read(pa, N)
      return memdata
   }
}
```

図 10 cld 命令の動作.

5. 結果と考察

5.1 再帰的構造体のフィールドの圧縮可能性

まず，圧縮対象フィールドへの動的メモリアクセス回数 (A_{target}) の，全動的メモリアクセス回数に対する割合 (アクセス割合と呼ぶ) と，圧縮可能データの出現回数の A_{target} に対する割合 (圧縮成功率と呼ぶ) を見る．圧縮率は $\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{16}$ と変える． 表 3 に結果を示す．これらのプログラムの主なデータ構造はグラフ構造である．ポインタフィールドについては，`treeadd`，`perimeter`，`em3d`，`tsp` は圧縮成功率が高い．これは構造体が作るグラフのノードの，メモリ上の順序と辿る順序がほぼ同じであるからである．これらのプログラムでは多くの再帰的ポインタが 8-bit に圧縮できている．`health`，`bh`，`mst`，`bisort` は圧縮成功率が低い．これはノードのメモリ上の順序と辿る順序が異なるからである．整数フィールドについては，`treeadd`，`tsp`，`em3d`，`bh`，`bisort` は使用ビット幅の狭い整数フィールドを持ち，圧縮成功率が高い．その中で `treeadd`，`bisort` ではアクセス割合も高い．`perimeter` は列挙型フィールドを持ち，アクセス割合，圧縮成功率共に高い．`perimeter`，`em3d` では圧縮率が高いときに，辞書を用いた圧縮方法が狭ビット幅を用いた方法より圧縮成功率が高い．これは前者が少ないビット幅をより効率的に用いるからである．`health`，`tsp` では狭ビット幅の方が成功率が高い．これはプロファイル実行時に現れなかった値が実行時に現れたためである．以降では，`treeadd`，`perimeter`，`tsp`，`em3d` には $\frac{1}{8}$，`health`，`bh`，`mst`，`bisort` には $\frac{1}{4}$ の圧縮率を用いる．

5.2 FAT，FACT のプログラム実行時間の比較

図 11 に FAT，FACT を適用した場合の結果を示す．各棒グラフは，下から，キャッシュミスによるメモリデータ待ちサイクル以外の busy cycle (busy)，二次キャッシュアクセスによる stall cycle(upto L2)，主記憶アクセスによる stall cycle(upto mem) である．全グラフは baseline を 100%とした相対グラフである．グラフからわかる通り，FACT はメモリデータ待ち

表 1 シミュレーションのパラメタ: baseline 構成の，プロセッサとメモリ階層のパラメタ.

フェッチ	fetch upto 8 insts, 24-entry request queue, 32-entry inst. queue
分岐予測	16K-entry GSHARE, 256-entry 4-way BTB, 16-entry RAS
Decode/Issue	decode upto 8 insts, 128-entry inst. window, issue upto 8 insts
実行ユニット/リタイア	4 INT, 4 LD/ST, 2 other INT, 2 FADD, 2 FMUL, 2 other FLOAT, 64-entry load/store queue, 16-entry write buffer, 8 MSHRs, retire upto 8 insts, 256-entry reorder buffer
L1 cache	inst.: 32KB, 2-way, 64B block size | data: 32KB, 2-way, 64B block size, 3-cycle load-to-use latency
L2 cache	256KB, 4-way, 64B block size, 13-cycle load-to-use latency, on-die
Main Memory	200-cycle load-to-use latency, off-chip memory controller, 128-bit address/data muxed bus to L2
TLB	256-entry, 4-way inst. TLB, 256-entry, 4-way data TLB, pipelined, 50-cycle latency h/w miss handler

表 2 評価に使われたプログラム: 名称，プロファイル実行用入力パラメタ，評価用入力パラメタ，動的に割り当てられたメモリの最大値，実行命令数，キャッシュミスによるメモリデータ待ち時間の割合， データ構造，使用する圧縮率，圧縮が行われた構造体のフィールドの数 (ポインタ，整数)．第 3〜5 列は baseline 構成のもの.

名称	プロファイル実行用入力パラメタ	評価用入力パラメタ	最大メモリ割当量	命令数	メモリ待ち率	データ構造	使用圧縮率	圧縮対象
health	lev. 5, time 50	lev. 5, time 300	2.58MB	69.5M	95.0%	四分木, 双連結リスト	1/4	2, 3
treeadd	4K nodes	1M nodes	25.3MB	89.2M	74.7%	二分木	1/8	2, 1
perim.	128×128 img.	16K×16K img.☆	19.0MB	159M	57.5%	四分木	1/8	5, 2
tsp	256 cities	100K cities	7.43MB	504M	47.6%	二分木, 双連結リスト	1/8	4, 1
em3d	1K nodes, 3D	32K nodes, 3D	12.4MB	213M	71.9%	単連結リスト	1/8	1, 2
bh	256 bodies	4K bodies☆☆	.909MB	565M	30.2%	八分木, 単連結リスト	1/4	10, 6
mst	256 nodes	1024 nodes	27.5MB	312M	47.1%	単連結リストの配列	1/4	1, 0
bisort	4K integers	256K integers	6.35MB	736M	48.2%	二分木	1/4	2, 1

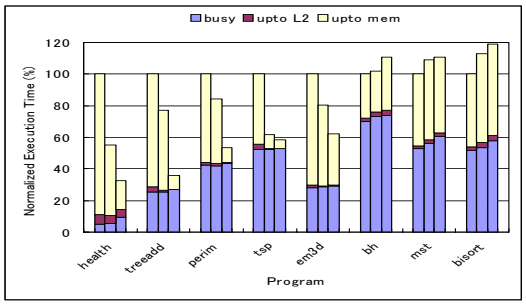

図 11 プログラムの実行時間比較．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，FACT を適用した場合の実行時間.

サイクルを平均 41.6%削減する．FAT 単体では平均 23.0%削減する．これらのプログラムの主なデータ構造はグラフ構造である．グラフのノードのメモリ上の順序と辿る順序がほぼ同じ際は，FAT がキャッシュブロックのプリフェッチの効果を増し，FACT が圧縮によりその効果を更に高める．`health`，`treeadd`，`perimeter`，`em3d` がこの場合に当たる．特に `treeadd`，`perimeter` では構造体全体が圧縮されるので FACT の効果が高い．`tsp` に関しては，FAT はメモリ待ちサイクルの多くを削減しており，FACT の更なる削減効果は小さい．

FAT は構造体内の複数のフィールドを複数のキャッシュブロックに分散させる．この点ではキャッシュの利用率を下げるが，temporal affinity を有するフィールドを複数インスタンスから集め，一つのキャッシュブロックに収めることで，全体として利用率を上げる．このためグラフのノードのメモリ上の順序と辿る順序が異なる際は，FAT は temporal afiinity を有するフィールドを集めることができず，キャッシュの利用率を下げ，性能を下げる．`bh` においては，グラフの作成の順序と辿る順序が異なる．`mst` においては，複数のリストが交代で少量の要素を割り当てて挿入する．`bisort` においては，動的に頻繁にグラフの構造が変わる．このため，これらのプログラムにおいては，グラフのメモリ上の順序と辿る順序が異なり，FAT の効果が低く，それに伴って FACT の効果も低い．

FACT において，圧縮対象が圧縮不能であった際は，圧縮後のデータ，圧縮前のデータの両方を参照する．この際，参照を二回行うためのオーバーヘッド，両者でのキャッシュミスにより速度が低下する．`bisort` は動的に頻繁にグラフの構造を変えるためポインタの圧縮不能率が高くなり，FACT により速度が低下する．

5.3 FACT 対ブロックサイズ二倍のキャッシュ

FACT は圧縮によりキャッシュブロックの実効サイズを増大させる．次にこの効果を見る．比較のため，キャッシュブロックの数は同じで，キャッシュブロックのサイズが倍になった一次・二次キャッシュ(倍キャッシュと呼ぶ) を考える．そして FACT と変更前のキャッシュの組み合わせを，FAT と倍キャッシュの組み合わせ (FATx2 と名付ける) と比較する．図 12 に結果を示す．倍キャッシュにすることで，キャッシュブロックのプリフェッチの効果が増大し，容量も二倍になる．mst 以外では，この効果によりメモリ待ちサイクルが減少している．`tsp` に関しては，FAT がすでに FACT と同程度メモリ待ちサイクルを削減しているため，FATx2 の方が効果が高い．`tsp`，`bh`，`bisort` 以外に対しては，FATx2 より FACT の効果の方が高い．このことから，FACT の圧縮が，倍キャッシュ以上の効果を持つ場合があることがわかる．

表 3 圧縮対象フィールドへの動的メモリアクセス回数 A_{target} の，全動的メモリアクセス回数に対する割合と，圧縮可能データの出現回数の，A_{target} に対する割合．上：ポインタフィールドの圧縮．中：狭ビット幅を用いる方法による整数フィールドの圧縮．下：辞書を用いる方法による整数フィールドの圧縮．

プログラム	圧縮対象参照割合 (%)			圧縮成功率 (%)		
圧縮法→	ポインタフィールド，相対シーケンス番号					
圧縮率 →	1/4	1/8	1/16	1/4	1/8	1/16
health	31.1	1.45	1.45	94.6	76.8	76.5
treeadd	11.6	11.6	11.5	100	98.9	96.5
perim.	17.6	17.5	17.6	99.8	95.9	85.6
tsp	10.2	10.2	10.2	100	96.0	67.1
em3d	.487	.487	.487	100	99.6	99.6
bh	1.56	1.56	.320	88.2	51.3	52.2
mst	5.32	5.32	0	100	28.7	0
bisort	43.0	41.2	41.0	90.8	65.6	59.2

プログラム	圧縮対象参照割合 (%)			圧縮成功率 (%)		
圧縮法→	整数フィールド，狭ビット幅					
圧縮率→	1/4	1/8	1/16	1/4	1/8	1/16
health	24.2	1.51	.677	83.7	88.6	93.7
treeadd	5.80	5.78	5.77	100	100	100
perim.	12.4	12.4	4.33	100	100	83.1
tsp	.107	.107	.107	99.2	87.5	50.0
em3d	1.54	1.54	.650	100	99.1	68.8
bh	.0111	.0111	.0111	100	100	100
mst	0	0	0	0	0	0
bisort	27.8	0	0	100	0	0

プログラム	圧縮対象参照割合 (%)			圧縮成功率 (%)		
圧縮法→	整数フィールド，辞書					
圧縮率→	1/4	1/8	1/16	1/4	1/8	1/16
health	24.3	24.1	.827	46.9	18.9	90.3
treeadd	5.80	5.78	5.77	100	100	100
perim.	12.4	12.4	12.4	100	100	91.0
tsp	.107	.107	.107	50.0	50.0	50.0
em3d	1.54	1.54	1.14	100	100	72.6
bh	.0176	.0111	.0111	100	100	100
mst	6.08	0	0	29.8	0	0
bisort	27.8	0	0	100	0	0

6. ま と め

再帰的構造体によるキャッシュミスを減らす手法 Field Array Compression Technique (FACT) を提案した．データレイアウト変換と再帰的ポインタ・整数フィールドの圧縮により，キャッシュブロックのプリフェッチの効果を増す．またキャッシュの実効容量を増す．特にポインタに対するプリフェッチ効果が増し，グラフを辿るコードに有効である．シミュレーションを通じて FACT が，平均 41.6%のメモリデータ待ちサイクルの削減，平均 37.6%の速度向上を示すことを確かめた．本稿の主な貢献は以下の三点である．

(1) FACT が従来の圧縮方法の限界 $\frac{1}{2}$ を超える，$\frac{1}{8}$ の圧縮率を達成した．また多くの再帰的ポインタフィールドが 8-bit に圧縮できることを示した．

(2) メモリ領域を圧縮前のデータ用，圧縮後のデータ用の二種に分けるという概念を示した．8-byte の圧縮前データに対し 1-byte の圧縮後データを対応させ，圧縮前データを圧縮後データの領域で符号語に置き換えて表現する．このレイアウトにより，密集させた圧縮後データを頻繁に参照させることができる．また，圧縮前データのアドレスは線型変換により圧縮後データのアドレスに変換できる．

(3) キャッシュ上の圧縮後データのために専用アドレス空間を用いる手法を示した．圧縮前データのアドレス空間を縮小したものを圧縮後データのアドレス空間として使う．これにより圧縮前データのアドレスから，キャッシュ上の圧縮後データを指すアドレスが簡単に得られる．またキャッシュコンフリクトが減少する．

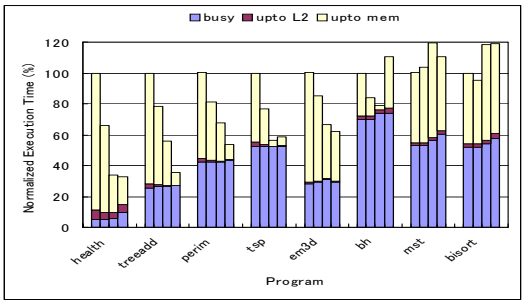

図 12 プログラムの実行時間比較．棒グラフは各グループについて，左から順に，baseline の場合，一次・二次キャッシュについてブロック数は変えずにブロックサイズを倍にした場合，変更したキャッシュに FAT を適用した場合，キャッシュはそのままで FACT を適用した場合の実行時間．

参 考 文 献

1) D. A. Barret et al. Using lifetime prediction to improve memory allocation performance. *In Proc. of PLDI '93*, 28(6):187–196, Jun. 1993.
2) D. N. Truong et al. Improving cache behavior of dynamically allocated data structures. *In Proc. of PACT '98*, pp. 322–329, Oct. 1998.
3) A. Rogers et al. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.
4) A. Roth et al. Dependence based prefetching for linked data structures. *In Proc. of ASPLOS VIII*, pp. 115–126, Oct. 1998.
5) J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. *In Proc. of MICRO-33*, pp. 258–265, Dec. 2000.
6) S. Y. Larin. Exploiting program redundancy to improve performance, cost and power consumption in embedded systems. Ph. D. Thesis, ECE Dept., North Carolina State Univ., 2000.
7) Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. *Int. Conf. on Compiler Construction*, LNCS 2304, pp. 14–28, Apr. 2002.