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，キャッシュの実効的な容量を増す．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，平均 13.4%の off-chip bus traffic の削減を示した．

Field Array Compression in Data Caches for Recursive Data Structures

Masamichi Takagi[†] and Kei Hiraki[†]

We introduce a software and hardware scheme called the Filed Array Compression Techique (FACT) which reduces cache misses caused by recursive data structures. Using a data layout transformation, FACT gathers data with temporal affinity in congtiguous memory, and then it compresses recursive pointer and integer fields there. As a result, FACT improves the prefetching effect of a cache-block. In addition, the compression enlarges effective cache capacity. On a suite of pointer-intensive programs, FACT achieves a 41.6% reduction in memory stall time, a 37.4% speedup, and a 13.4% reduction in off-chip bus traffic 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 を表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．プログラムの例としては，可変個のオブジェクトを扱うレイトレーシング法のプログラム，空間を木構造で表す多体問題を解くプログラムがあげられる．RDS を用いるプログラムはグラフを作ってからそれをたどる．このたど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，プリフェッチがあげられる．ソフトウェア・ハードウェアによるプリフェッチ[1),2)]が代表的な方法である．もう 1 つの方法として，データレイアウト変換があげられる．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持つプリフェッチ効果を高める[3)]．さらに他の方法として，キャッシュ容量の増大があげられる．この方法はアクセス速度低下を招くために限界がある．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する手法として，キャッシュにおけるデータ圧縮が提案されている[4)~7)]．この手法はキャッシュの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さらに高めることができる．この組合せはプリフェッチと補い合う方法として利用できる．ところが変数単位で見れば 1/8 以上にできる圧縮率が，従来のデータ圧縮方法では，キャッシュの構造が複雑になる問題，圧縮可能・不可能フィールドの混在による問題によって，1/2 に制限されている．このため，データレイアウト変換との相乗効果は限られる．そこで本稿では，従来方法の限界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

† 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
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo

（FACT）と名づ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高める．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，圧縮前データ・圧縮後データのメモリ上での配置を工夫し，さらに圧縮データをキャッシュ上で指定する方法を工夫する．これらによって，圧縮率を制限する問題を解決し，従来方法の限界 1/2 を超えた 1/8 以上の圧縮率を実現する．

本稿の構成は以下のとおりである．2 章で関連研究について述べ，3 章で，Field Array Compression について説明し，4 章で評価方法を述べ，5 章で評価結果について述べ，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の各 32-bit word を可能なものは 3-bit 符号語へ変換し，1 次キャッシュのキャッシュブロックの半分の大きさのスロットに格納する．残りの半分は他の圧縮データに使われる．データの圧縮方法について，Yang 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，固定長の符号語と静的な辞書を用いて圧縮する．我々の手法では，整数フィールドの圧縮にこの方法を採用している．

Larin と Conte はハードウェアを用いる手法を提案した[6]．この方法もプログラムの変更の必要はない．この方法は，N 個のキャッシュブロックをバイト単位で huffman code を用いて圧縮し，1 次キャッシュに格納する．Lee らはハードウェアを用いる手法を提案した[4]．この手法もプログラムの変更を必要としない．この方法は 2 個のキャッシュブロックを X-RL アルゴリズム[8] を用いて圧縮し，2 次キャッシュのキャッシュブロックの大きさのスロットに納める．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，圧縮前のデータと同じアドレスとタグを用いるので，圧縮率が $1/R$ の場合，圧縮データの参照の際に R 個のタグをチェックせねばならないので，ハードウェアが複雑になる．このため圧縮率を 1/2 に制限している．FACT は，圧縮前データ・圧縮後データのメモリ上での配置の工夫，圧縮データをキャッシュ上で指定する方法の工夫によりこの問題を回避する．

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，構造体内にワードの大きさのスロットを設ける．そして構造体のポインタと整数のフィールドのうち，1/2 に圧縮できるもの 2 つをセットにしてスロットに納める．スロットは構造体内にあるため，圧縮しないフィールドのワードアライン要請により，スロットの大きさを 1 ワードより小さくできない．また，スロットに納めるデータは 1 つのインスタンスから集めねばならない．この 2 つの問題が圧縮率を制限する．FACT は圧縮可能なフィールドを分離収集してから圧縮することでこの問題を解決する．また，圧縮データのメモリ上での格納場所について，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，圧縮が不可能な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．一方で我々の手法は，圧縮前後両方の領域を最初から割り当てる．

Truong らは instance interleaving と名づける，動的に割り当てられる再帰的構造体のデータレイアウトの変換方法を提案した[3]．この変換は構造体の複数のインスタンスから同一のフィールドを取り出してきて一列に並べる．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．また，キャッシュブロックの利用率を上げることで，キャッシュブロックの利用数を減らし，ミスを減らす．この手法は，プログラムのソースコードに手を入れ，構造体宣言と，メモリ割当ての部分を変更する必要がある．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，この変換とともに用いれば相乗効果を生む．それだけでなく，この変換によって，圧縮可能なフィールドを分離収集できる．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

FACT は RDS のデータレイアウト変換と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とおり．

本手法では，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．このデータレイアウト変換により，キャッシュブロックのプリフェッチの効果が増大する．また，キャッシュブロックの利用率が上がる．利用率増大は一定時間あたりのキャッシュブロックの利用数を減少させ，ミスを減らす．本手法ではさらに構造体の

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増大させる．また，キャッシュの実効容量増大により，ミスが減る．

手順は以下のとおり．

(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・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．現時点では変更作業は手で行われている．

(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` と名づける）．`cld`，`cst` は圧縮対象，圧縮方法に応じて複数種類用意し，置き換えの際にはそれぞれがアクセスするフィールドの種類，圧縮方法に応じたものを用いる．

(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）と名づける．本手法は配列を用いることによって，圧縮率を制限する問題を解決し，従来方法の限界 1/2 を超える，1/8 以上の圧縮率を実現する．圧縮されたデータは圧縮されないデータと同様の扱いを受けキャッシュに格納される．またキャッシュは圧縮されないデータと共有する．本手法を実装するシステムは 1 次データキャッシュ，2 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，圧縮された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．具体的には以下のとおり．

(1) 構造体フィールドの圧縮方法

(2) 圧縮対象フィールドの選択方法

(3) 構造体のデータレイアウト変換による圧縮可能フィールドの分離

(4) 圧縮データのキャッシュ上でのアドレッシング

(5) `cld`/`cst` への置き換えとその動作

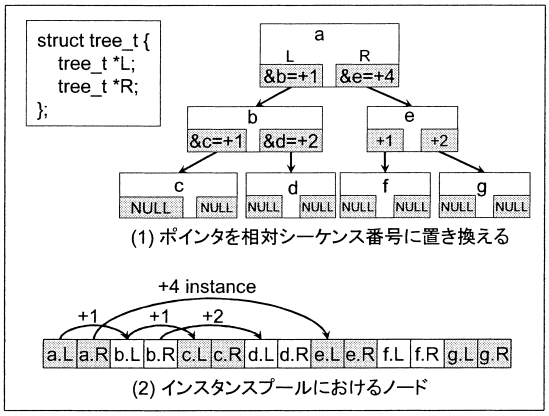

図 1 ポインタフィールドの圧縮：絶対アドレスをインスタンスプール内の相対シーケンス番号に置き換える

Fig. 1 Pointer compression: we replace absolute addresses with relative sequence numbers in the instance pool.

3.1 構造体のフィールドの圧縮

提案する手法は，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のフィールドを圧縮する．また，圧縮可能な冗長性を持つことの多い整数のフィールドを圧縮する．提案する手法は，1 次キャッシュに圧縮データを格納するため，圧縮復号が短時間で行え，追加ハードウェアが少ない圧縮アルゴリズムを必要とする．そこで構造体の圧縮にはフィールドを固定長の符号語に置き換える方法を用いる．

3.1.1 シーケンス番号を用いたポインタの圧縮

再帰的構造体のポインタは，同じ構造体のインスタンスしか指さない．また，再帰的構造体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 2 つのインスタンス間のアドレスの差は多くの場合小さい．このため，再帰的構造体のポインタは，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．この方法を用いた圧縮のために，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当てのステップは文献 9) と同様である．割当て要求時に，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 1 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，このアロケータを用いるようにする．実行時に `cst` 命令がポインタをインスタンスプール内の相対シーケンス番号に置き換える．

図 1 に圧縮の様子を示す．再帰的構造体を使って，深さ優先の順で完全二分木を作ることを考える (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

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：ポインタを相対シーケンス番号に置き換える．狭いビット幅で表せない範囲には圧縮不能を表す符号語を用いる．NULL には NULL を表す符号語を用いる
Fig. 2 Compression algorithm of pointer fields.

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 にポインタフィールドの圧縮アルゴリズムを示す．圧縮は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 節に述べるデータレイアウトの変更により，隣り合うインスタンスの先頭アドレスの差は通常 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．結果として，圧縮の計算は加算とシフトで実現できる．

復号はポインタの読出しの際に行う．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．圧縮率 1/8 の際は，復号には，キャッシュから word データを取得した後，8 対 1 の MUX を経た後，8-bit の加算とシフトが必要になる．

3.1.2 整数フィールドの圧縮

整数フィールドを圧縮する手法について述べる．整数型変数のとる値には偏りがあり，とる値の種類が少ない変数がある[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変数がある．FACT ではそれぞれの性質を使い，2 つの手法を用いて圧縮する．

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：`cst` 命令の種類によって辞書を使う方法，狭いビット幅を用いる方法を分ける．辞書にないデータ，狭いビット幅で表せないデータには圧縮不能を表す符号語を用いる
Fig. 3 Compression algorithm for integer fields.

1 番目は，実行の全体を通じて，少数の別々の値しかとらない 32-bit の整数型のフィールドを探し，固定長の符号語と静的な辞書を用いて圧縮する方法である[5]．2 番目は，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方法である．1 番目の方法は 2 番目の方法を含んでいるが，復号の際にハードウェアの辞書を引く必要があるため，辞書のエントリ数を大きくすると復号に時間がかかる．このため，エントリ数が 16 以下の際には 1 番目の方法を用い，それ以外には 2 番目の方法を用いる．つまり，プログラム全体の圧縮率について，1/8 およ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 1 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 2 番目の方法を用いる．

1 番目の手法の，エントリ数 N の辞書作成の方法は以下のとおり：まず，プロファイル実行によって，すべての 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とった N 個の値を静的な辞書とする．

図 3 上部に辞書を用いた整数フィールド圧縮のアルゴリズムを示す．圧縮は整数フィールドの格納の際に行う．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．実際には，辞書は CAM を用いる．復号は整数フィールドの読出しの際に行い，読み出したデータで辞書を引く．実際には，辞書はレジスタファイルを用いる．圧縮率 1/8 の際は，キャッシュから word データを得た後，8 対 1 の MUX を経た後，16 エントリのレジスタファイルを引くことで実現できる．

図 3 下部に狭ビット幅を用いた整数フィールド圧縮のアルゴリズムを示す．圧縮の際は，格納データの使用ビット幅を調べ，圧縮可能ならば上位ビットを省く．復号の際は，圧縮データを符号拡張する．想定するプロセッサモデルにおける圧縮復号のタイミングは 1 番目の方法と同じとする．

3.2 圧縮対象構造体フィールドの選択

FACT ではまず，プログラム中の再帰的構造体のフィールドのうち，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とる．プロファイル実行時の入力パラメー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，アクセス数，参照されたデータが圧縮可能であった数である．そして，アクセス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構造体が圧縮対象となる．現時点では経験的に $A = 0.1\%$，$B = 90\%$ と設定している．プロファイルは複数の圧縮率についてとる．64-bit ポインタの実際の情報量は 32-bit 以下であると考え，ポインタが 32-bit より小さくなる圧縮率を考える．また，32-bit 整数は 1/16 に圧縮すると 2-bit になるので，これ以上の bit 数になる圧縮率を考える．つまり，1/4，1/8，1/16 についてとっている．最後に，プロファイルに示される圧縮可能率によって 1 つの圧縮率を選ぶ．現時点では経験的に選んでいる．

3.3 構造体のデータレイアウト変換による圧縮可能フィールドの分離

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．変換はソースコードの変更によって行う．この変換は Truong らの提案した Instance Interleaving [3] と同じである．変更作業は現時点では手で行われている．レイアウト変換によって，同一の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）と名づける．

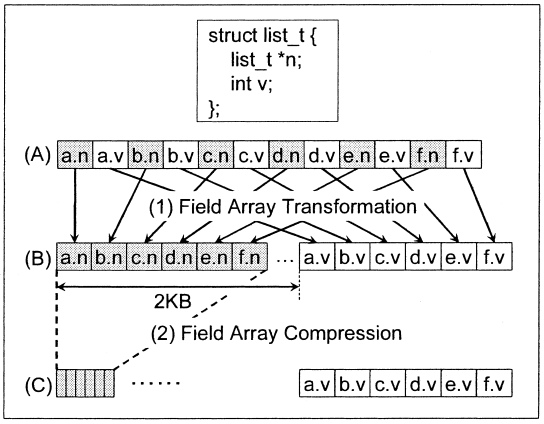

図 4 Field Array Transformation（FAT）は圧縮可能なフィールドを集め，圧縮不能なフィールドから分離する
Fig. 4 Fields Array Transformation (FAT) isolates and groups the compressible fields.

3.3.1 圧縮に適したデータレイアウトへの変換

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後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．また R が 2 のべき乗でない際は複数サイクルかかる除算が必要となる．このため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，1 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)．すべてのポインタを 1/8 に圧縮できたとすると，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 $I/8$ になる．つまり配列の形にすることによって圧縮可能・不能なものを分け，可能なもののみ圧縮できる．また可能なものは多くの場合全要素が圧縮可能なので，同率で圧縮すれば，アドレス変換が単純になる．

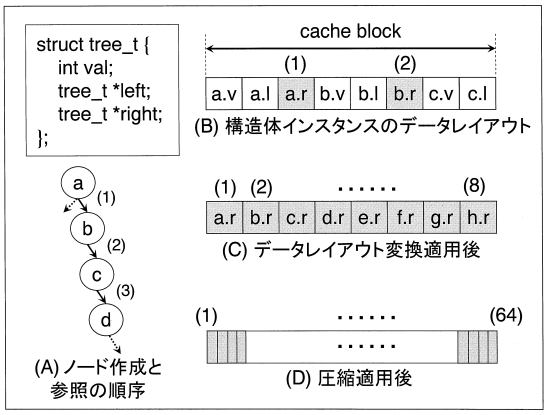

図 5 FAT は temporal affinity を持つフィールドを集め，FACT は圧縮により 1 キャッシュブロックにより多くのフィールドが収まるようにする

Fig. 5 FAT groups fields with temporal affinty; FACT allows one cache-block to contain more fields by compression.

3.3.2 FAT による temporal affinity の利用

フィールドの配列を作ることにより，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．`val` の後ろには 8-byte アラインのため 4-byte の pad がある．`treeadd` は二分木を，`right` を先にした depth-first order で作る (A)．それゆえメモリ上でも木のノードは depth-first order に並んでいる (B)．`treeadd` はその後同じ順序で木を再帰呼び出しを用いてたどる．このため，メモリ上で連続する 2 つの `right` ポインタには temporal affinity がある．FAT なしでは，インスタンスのサイズは 24-byte なので，キャッシュのブロックサイズを 64-byte とすると，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)，さらに 1/8 の圧縮を加えた場合，64 に増大する (D)．この変換により，1 つ目にキャッシュブロックのプリフェッチの効果が高まる．2 つ目にキャッシュブロックの利用率が上がる．利用率増大は一定時間あたりのキャッシュブロックの利用数を減少させ，ミスを減らす効果がある．

3.3.3 専用メモリアロケータ

データレイアウト変換は，ソースコードの構造体宣言部分，メモリ割当て部分の変更によって行う．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** にプログラム `treeadd` の構造体 `tree_t` の宣言を示す．(1) が FAT 適用前，(2) が FAT 適用後のものである．load/store 命令のアドレッシングは

```
typedef struct tree {
    int val;
    struct tree *left, *right;
} tree_t;
```
(1)

```
typedef struct tree {
    int val;
    char pad1[2044];
    tree *left;
    char pad2[2040];
    tree *right;
    char pad3[2040];
} tree_t;
```
(2)

図 6 プログラム `treeadd` の構造体 `tree_t` の宣言．FAT 適用前（1）と FAT 適用後（2）

Fig. 6 Declaration of `tree_t` in `treeadd`, original (1) and after FAT (2).

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 32 KB 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．pad の挿入によって，フィールドの間隔は pad のサイズになるため，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)$\times(n-1)$ となる．このため pad を大きくするとフィールド参照時に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が不可能になる．そこで pad のサイズを 2 KB 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．こうすることによってポインタの圧縮時に相対アドレスを算出する際に必要な，ポイント元のアドレスが得られるようにする．

次に，専用アロケータを使うようプログラムを変更する．このアロケータは引数として ID をとり，ID ごとにインスタンスプールを管理する[9]．**図 7** に割当ての様子を示す．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)．その後，その構造体をインスタンスプールとして用いるので，次の割当て要求時には，$A+8$ を返す (2)．オブジェクトの free list は文献 10) と同様の方法で管理する．圧縮を行う際は，圧縮前の領域のほかに圧縮後の領域も必要とするため，3.4 節で述べるように，先頭の一部を圧縮後のデータの領域とする．

評価時の baseline となる構成にも文献 10) 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．というのは，この変更だけで速度向上が得られたためである．

3.4 圧縮データ用アドレス空間

圧縮を試みるデータは動的に変わるため，つねに圧

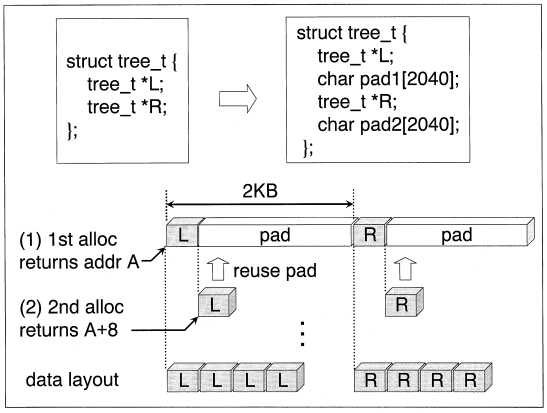

図 7 専用アロケータによる FAT の様子．構造体宣言に pad を入れることにより，連続領域に隙間を設け，その隙間に次々と同一フィールドを割り当てていくことにより，フィールドの配列を作る

Fig. 7 FAT using a custom allocator. By inserting pads between fields, it reserves the contiguous area, and places identical fields from other instances there.

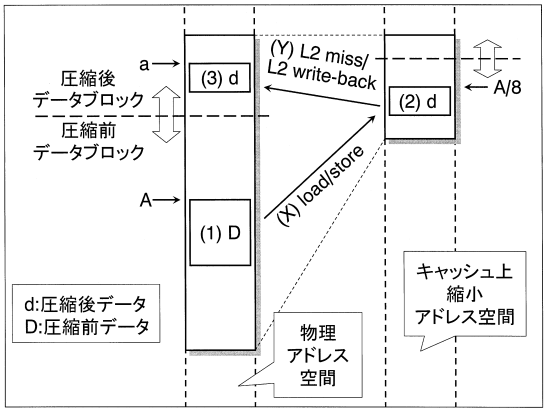

図 8 FACT におけるアドレス変換の様子

Fig. 8 Address translation in FACT.

縮可能とは限らない．圧縮が不可能な際は，圧縮しない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 2 つある．1 つは，圧縮データ用の領域のみを最初に割り当てて，圧縮不可能な際には新たに割当てを行うものである[7]．この方法では，圧縮データの位置に非圧縮データのアドレスを格納し，圧縮データの参照の際にリダイレクトを行う．もう 1 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．1 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 2 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．この変換に，page-table および TLB を用いる方法では，OS のインタフェースの追加と TLB の変更が必要となる．これを避けて 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．また圧縮後データの領域では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，キャッシュの $R/(R+1)$ 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名づける．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．一定サイズのブロックを一定の割合で二分して圧縮データと非圧縮データの領域に当てているので，変換は計算で行える．主記憶からの fetch の場合，この変換は 2 次キャッシュのアクセスと並行して行い，隠蔽する．主記憶への write-back の場合は追加の latency が必要となるが，実行時間への影響は評価に使用したプログラムでいずれも 1%以下である．

図 8 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 から始まる配列 (1) を 1/8 に圧縮することを考える．圧縮後のデータはあらかじめ割り当てておいた主記憶の領域 (3) に格納される．その物理アドレスを a とする．キャッシュ内の圧縮後データにアクセスする `cld/cst` 命令は，通常の load/store 命令と同様，A というアドレス情報を持っている．`cld/cst` 命令は A を使ってアドレス変換を行う．この例では，圧縮率は 1/8 であるため，$A/8$ を使ってキャッシュ上の圧縮後データにアクセスする (2)(X)．つまり，物理アドレス空間のアドレス A を縮小アドレス空間のアドレス $A/8$ に変換して用いる．圧縮ブロックの主記憶への write-back，または主記憶からの fetch の際

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 9 cst 命令の動作：圧縮可能であれば圧縮後のデータを，圧縮不能であれば圧縮不能を表す符号語と圧縮前のデータを格納する
Fig. 9 Operation of cst instruction.

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 10 cst 命令のキャッシュに対する動作：圧縮データに対しキャッシュミスした際は，主記憶上の圧縮データのアドレスを算出した後主記憶から圧縮データを取得しフィルする
Fig. 10 Operation of cst instruction in the cache.

は，縮小アドレス空間のアドレス $A/8$ を物理アドレス 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 2 種類ある．これに対応して cld/cst 命令はそれぞれ 3 種類存在し，置き換えの際に 1 種類を選ぶ．圧縮対象のポインタフィールドを参照する load/store 命令はポインタ圧縮を行う種類の cld/cst 命令に置き換えられ，圧縮対象の整数フィールドを参照する load/store 命令は，整数圧縮を行う種類の cld/cst 命令に置き換えられる．整数圧縮を行う cld/cst 命令は 2 種類あるが，1 種類のみをプログラム全体で使う．どちらを使うかは，プロファイル情報を基に判断する，プログラム全体を通じての圧縮率に応じて決める．具体的には，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 9** に，cst 命令のキャッシュに対する動作を**図 10** 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．圧縮されたデータはアドレス変換を経て 1 次・2 次キャッシュに格納される．このアドレス変換は，アドレスを圧縮率と同率

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 11 cld 命令の動作：キャッシュから取得したデータが圧縮不能を表す符号語である際は圧縮前のデータを取得する
Fig. 11 Operation of cld instruction.

で縮める．圧縮不能なデータを扱う際は，圧縮不能を表す符号語を格納し，その後圧縮されていないデータを変換前のアドレスに格納する．圧縮データのキャッシュブロックがすべてのキャッシュでミスした際には，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，圧縮データのキャッシュ上のアドレスを圧縮データの主記憶上のアドレスに変換する．

cld の動作を**図 11** に，cld 命令のキャッシュに対する動作を**図 12** に示す．cld 命令は，圧縮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．このアドレス変換は，アドレスを圧縮率と同率で縮める．圧縮データのキャッシュブロック

addr	キャッシュ上でのアドレス
flag	C:圧縮データ、N:非圧縮データ

```
/* cldのキャッシュ参照時の動作 */
cache_read(addr, flag) {
   if(addr、flagでキャッシュミス) {
      if(flag == C) {
         pa = addrから主記憶上の圧縮データの
              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
      }
   }
   return 取得したメモリデータ
}
```

図 12 cld 命令のキャッシュに対する動作：圧縮データに対しキャッシュミスした際は，主記憶上の圧縮データのアドレスを算出した後主記憶から圧縮データを取得しフィルする
Fig. 12 Operation of `cld` instruction in the cache.

から圧縮不能を示す符号語を取り出した際は，圧縮されていないデータに，変換前のアドレスを使ってアクセスする．圧縮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. Evaluation Methodology

FACT を実装するアーキテクチャはスーパスケーラの 64-bit プロセッサを用いると仮定する．パイプラインは整数命令に対しては，命令キャッシュアクセス 1，同アクセス 2，decode/rename，schedule，レジスタ読込み，実行，write-back/commit の 7 段構成とする．load/store 命令に対しては，レジスタ読込みの後に，アドレス生成，TLB アクセス/データキャッシュアクセス 1，同アクセス 2，write-back/commit とする．したがって load-to-use latency は 3 サイクルである．`cst` 命令がポインタフィールド，整数フィールドを圧縮する計算は `cst` 命令のアドレス生成，TLB 参照のサイクルで行え，latency を追加しないとする．`cld` 命令がポインタフィールド，整数フィールドを復号する作業は，load-to-use latency に 1 サイクルを加えるとする．`cst` 命令が圧縮不能なデータを扱う際は，圧縮後・圧縮前のデータの両方を書き込まねばならない．このときのペナルティは最低 4 サイクルとなる．`cld` 命令が圧縮不能なデータを扱う際は，圧縮後・圧縮前のデータの両方を読まねばならない．このときのペナルティは最低 6 サイクルとなる．また，`cld`/`cst` がキャッシュ上で圧縮データをアクセスする際にアドレスを縮小する演算は，各命令のアドレス生成，TLB 参照ステージで行え，latency を追加しないとする．

我々は，スーパスケーラプロセッサの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．このシミュレータは cycle-accurate である．**表 1** にそのパラメータを示す．命令の latency および issue rate は Compaq Alpha 21264 [11] と同じである．base-line モデルでは，メモリ階層は，64-byte block，2-way set-associative の 32-KB の 1 次データキャッシュ，同じ構成の 1 次命令キャッシュ，64-byte block，4-way set-associative の 256-KB の 2 次キャッシュとなっている．キャッシュおよびバスでの contention もシミュレートする．

評価には，動的に割り当てられる再帰的データ構造を用い，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 [12] のプログラム `health`，`treeadd`，`perimeter`，`tsp`，`em3d`，`bh`，`mst`，`bisort` である．プログラムの入力パラメータ，特徴，評価時の使用圧縮率を**表 2** に示す．すべてのプログラムは 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．

5. 結果と考察

5.1 再帰的ポインタフィールドと整数フィールドの圧縮可能性

まず，プロファイル実行用入力パラメータを用いてプロファイル実行で圧縮対象構造体を選んだ後，評価用のパラメータを用いてプログラムを走らせた際の，圧縮を試みたフィールドのメモリアクセス割合と，圧縮可能率を見る．圧縮率は 1/4，1/8，1/16 と変える．64-bit ポインタフィールドはそれぞれ 16-bit，8-bit，4-bit に圧縮される．32-bit 整数フィールドはそれぞれ 8-bit，4-bit，2-bit に圧縮される．**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` のような，構造体が作るグラフのノードの，メモリ上の順序とたどる順序がほぼ同じものは圧縮成功率が高いが，`health`，`bh`，`mst`，`bisort` のような，それらの順序が異なるものは成功率は低い．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．そのなかで `treeadd`，`bisort` では圧縮を試みたアクセスの全アクセスに対する割合も高い．`perimeter` は列挙型フィールドを持ち，アクセス割合，圧縮成功率ともに高い．`bh`，`mst` では圧縮の可能なアクセスが少ない．`perimeter`，`em3d`

表 1 シミュレーションのパラメータ：baseline 構成の，プロセッサとメモリ階層のパラメータ
Table 1 Simulation parameters: parameters of processor and memory hierarchy for baseline configuration.

プロセッサ		
パイプライン	7 stages, 6-cycle misprediction penalty	
フェッチ	fetch upto 8 insts, regardless of cache-block boundary, one request per cache-block, second taken branch terminates fetch, 24-entry request queue, 32-entry inst. queue	
分岐予測	16 K-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	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, oracle resolution of load–store addr. dependency	
リタイア	retire upto 8 insts, 256-entry reorder buffer	
メモリ階層		
L1 cache	inst.: 32 KB, 2-way, 64 B block size	data: 32 KB, 2-way, 64 B block size, 3-cycle load-to-use latency
L2 cache	256 KB, 4-way, 64 B block size, 13-cycle load-to-use latency, on-die	
Main memory	200-cycle load-to-use latency, off-chip memory controler, 128-bit address/data muxed bus to L2, which is clocked at 1/4 frequency of processor chip	
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，プロファイル実行時入力パラメータ，入力パラメータ，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，選択された圧縮率，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
Table 2 Program used in the evaluation: input parameters for profile-run, input parameters for evaluation, max. memory dynamically allocated, instruction count, ratio of stall cycles waiting for memory data due to cache-miss against the total execution cycles, data structures, compression ratio used, numbers and kinds of compression target fields in data structures (integer, pointer). The 4th to 6th column show the numbers in the baseline configuration.

名称	プロファイル実行時入力パラメータ	入力パラメータ	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	2.58 MB	69.5 M	95.0%	四分木，双連結リスト	1/4	2, 3
treeadd	4 K nodes	1 M nodes	25.3 MB	89.2 M	74.7%	二分木	1/8	2, 1
perim.	128×128 img.	16 K×16 K img.☆	19.0 MB	159 M	57.5%	四分木	1/8	5, 2
tsp	256 cities	100 K cities	7.43 MB	504 M	47.6%	二分木，双連結リスト	1/8	4, 1
em3d	1 K nodes, 3 D	32 K nodes, 3 D	12.4 MB	213 M	71.9%	単連結リスト	1/8	1, 2
bh	256 bodies	4 K bodies☆☆	.909 MB	565 M	30.2%	八分木，単連結リスト	1/4	10, 6
mst	256 nodes	1,024 nodes	27.5 MB	312 M	47.1%	単連結リストの配列	1/4	1, 0
bisort	4 K integers	256 K integers	6.35 MB	736 M	48.2%	二分木	1/4	2, 1

☆ `perimeter` は 4 K×4 K ではなく 16 K×16 K 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．

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．これは辞書を用いた方法が少ないビット幅をより効率的に用いるからである．一方で，`health`，`tsp` では狭ビット幅を用いた方法が辞書を用いた方法より圧縮成功率が高い．これはプロファイル実行時に用いられていない値が実行時に用いられたためである．

以降では，`treeadd`，`perimeter`，`tsp`，`em3d` には 1/8，`health`，`bh`，`mst`，`bisort` には 1/4 の圧縮率を用いる．

5.2 整数フィールド圧縮法，ポインタフィールド圧縮法それぞれの効果

FACT では，ポインタフィールドの圧縮と，整数フィールドの圧縮を組み合わせて使う．さらに整数フィールドの圧縮に関しては辞書を用いる方法と狭ビット幅を用いる方法を選択して用いる．そこで，これら 3 種類の圧縮方法の効果を個別に見てみる．**図 13** にそれぞれの方法を単体で適用した場合の結果を示す．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，狭ビット幅を使用した整数圧縮を適用した場合，辞書を使用した整数圧縮を適用した場合，ポインタ圧縮を適用した場合，FACT

表 3 圧縮を試みたメモリアクセスの，全メモリアクセスに対する割合と圧縮成功率．プロファイル実行時入力パラメータを用いてプロファイル実行を行って，圧縮対象構造体を選択した後，評価用パラメータを用いてプログラムを実行した．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

Table 3 Dynamic memory accesses of compression target fields (A_{target}) normalized to the total dynamic memory accesses, and dynamic accesses of compressible data normalized to A_{target}. Upper: compression of recursive pointer fields. Middle: compression of integer fields using narrow bit-width. Bottom: compression of integer fields using dictionary.

プログラム名	圧縮を試みたポインタのアクセス割合（%）			圧縮可能であったポインタの率（%）		
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

プログラム名	圧縮を試みた整数のアクセス割合（%）			圧縮可能であった整数の率（%）		
圧縮法→	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

プログラム名	圧縮を試みた整数のアクセス割合（%）			圧縮可能であった整数の率（%）		
圧縮法→	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

を適用した場合の実行時間である．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），2 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．

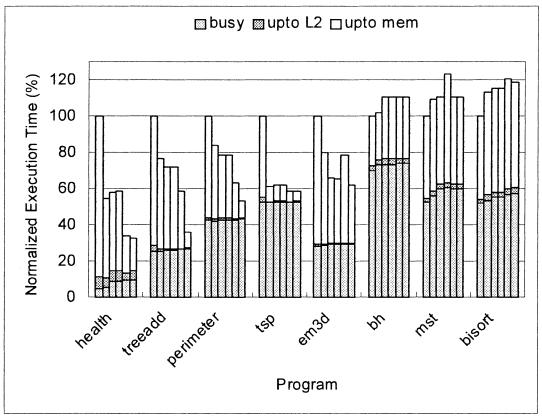

図 13 プログラムの実行時間比較．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，狭ビット幅を使用した整数圧縮を適用した場合，辞書を使用した整数圧縮を適用した場合，ポインタ圧縮を適用した場合，FACT を適用した場合の実行時間

Fig. 13 Execution time of the programs. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, with integer compression using narrow bit-width, with integer compression using the dictionary, with pointer compression, and with FACT, respectively.

まず整数フィールド圧縮とポインタフィールド圧縮を比べる．`health`，`treeadd`，`perimeter`，`tsp` においては整数圧縮よりポインタ圧縮の効果が大きい．これは，これらのプログラムではポインタをたどるクリティカルパスにおいて整数フィールドをあまり読まないためである．一方 `em3d` では整数圧縮のほうがメモリ待ち時間を削減する．これはクリティカルパスが，整数フィールドを読む部分に依存しているためである．また，圧縮可能な整数フィールドのほうがポインタフィールドより数が多いためである．

次に，2 種類の整数フィールド圧縮法を比べる．`health`，`mst` においては狭ビット幅を用いたほうが実行時間が短い．これは辞書を用いる方法について，プロファイル時に用いられなかった値が用いられたことによる．その他のプログラムでは 2 種類の方法はほぼ同じである．

`bh`，`mst`，`bisort` においてはどの整数圧縮方法もメモリ待ち時間を削減しない．また，ポインタ圧縮も効果が低い．これはノードをたどる順序がメモリ上の順序と異なるために，FAT が temporal affinity のあるデータを連続領域に配置できないためである．

図 14 にポインタ圧縮と整数圧縮を組み合わせて適用した場合の結果を示す．`health`，`mst` 以外については 2 つの整数圧縮法の間の差は小さい．`health`，`mst`

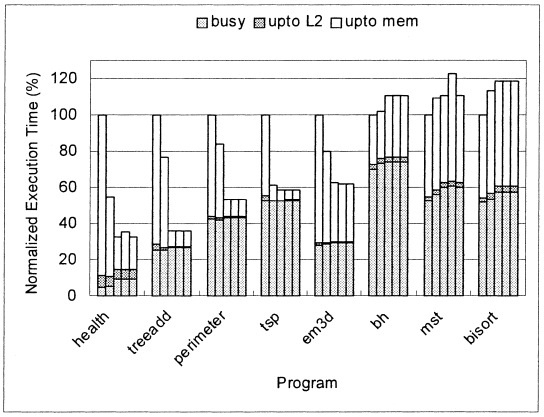

図 14 プログラムの実行時間比較．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，狭ビット幅を使用した整数圧縮とポインタ圧縮を適用した場合，辞書を使用した整数圧縮とポインタ圧縮を適用した場合，FACT を適用した場合の実行時間

Fig. 14 Execution time of the programs. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, with integer field compression using narrow bit-width and pointer field compression, with integer field compression using the dictionary and pointer field compression, with FACT, respectively.

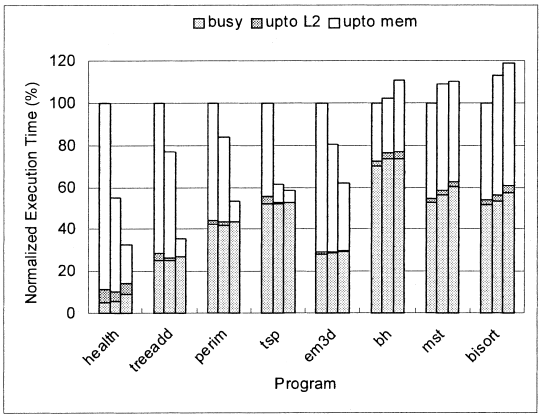

図 15 プログラムの実行時間比較．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，FACT を適用した場合の実行時間

Fig. 15 Execution time of the programs with FAT and FACT. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, and with FACT, respectively.

においては狭ビット幅を用いたほうがよりメモリ待ち時間を削減する．図 13，図 14 を比較すると，FAT の効果の低い `bh`，`mst`，`bisort` を除くと，ポインタと整数の圧縮法を組み合わせたほうが単体よりメモリ待ち時間を削減できることが分かる．FACT はポインタと整数の圧縮法を組み合わせたうえで，整数の圧縮方法については 2 つの方法から選択している．両方の図のそれぞれにおいて，右端のグラフが FACT の実行時間を表す．他の組合せとの比較から分かるとおり，FAT の効果の低い `bh`，`mst`，`bisort` を除くと，FACT が最もメモリ時間を削減する組合せと同じ効果を示している．

5.3 FAT，FACT のプログラム実行時間の比較

図 15 に FAT，FACT を適用した場合の結果を示す．棒グラフは各グループについて，左から，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である．

グラフから分かるとおり，FACT はメモリデータ待ちサイクルを平均 41.6%削減する．FAT 単体では平均 23.0%削減する．FACT は，圧縮を行うことによって FAT からさらにメモリデータ待ちサイクルを削減する．

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．`health`，`treeadd`，`perimeter`，`em3d` に関しては，グラフのノードのメモリ上の順序とたどる順序がほぼ同じなので，FACT による効果がある．FAT もこれらの速度を向上させるが，いずれも FACT のほうが速度向上が大きい．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 の主な効果は頻繁にアクセスされるフィールドとそうでないフィールドの分離である．この分離によって，メモリ待ちサイクルのほとんどを削減しており，FACT のさらなる削減効果は小さい．

FAT において，グラフのノードのメモリ上の順序とたどる順序が異なる際は，構造体内の複数のフィールドが複数のキャッシュブロックに分散していて，かつ各ブロックは temporal afiinity を有するフィールドを収めていない状態になる．このとき構造体内の複数のフィールド間に temporal affinity があると，複数ブロックに対する連続ミスを起こす．FAT なしではこれは同一ブロックへの連続ミスとなる．複数ブロックに対する連続ミスはメモリバスのコンフリクトによって同一ブロックに対する連続ミスより遅延が大きく，速度低下につながる．FACT において，圧縮を試みたものの圧縮が不可能であった際は，圧縮データと，非圧縮データの両方を参照する．この際，参照を 2 回行うためのオーバヘッド，両者でのキャッシュミス，圧縮データと非圧縮データ間のコンフリクトにより，速度が低下することがある．

`bh` においては，グラフの作成の順序とたどる順序が異なる．`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たどる順序が異なる．また，これらのプログラムには構造体内の複数フィールド間に temporal affinity がある．このため FAT の効果が低く，それにともなって FACT の効果も低い．また bisort においては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．なお，health，bh，mst，bisort においては busy cycle が増大している．これは，構造体内の複数のフィールドが複数のキャッシュブロックに分散し TLB ミスが増加するため，また，圧縮を試みたが不可能であったデータが多く，これらが 2 回の参照を必要とするためである．

5.4 FAT，FACT のミス削減効果の内訳

FAT，FACT は複数の効果によってキャッシュミスを減らすため，それぞれがどの程度貢献しているかを見る．効果は主に，キャッシュブロックのプリフェッチの効果と，キャッシュブロックの再利用数増加の効果に分かれる．そこで，キャッシュアクセスについて，それぞれに対応する spatial-hit，temporal-hit という尺度を用いる．これらは以下のようにして計測される：ソフトウェアシミュレータ上で，キャッシュアクセスのたびに，キャッシュ上にあるキャッシュブロックについて，どのワードが参照されたかを記録しておく．この記録はキャッシュフィルの際に，フィルをおこしたワードのみ参照されているとしてリセットする．また，1 次・2 次キャッシュ両方にブロックがある際は，1 つのキャッシュアクセスで両方に記録される．メモリアクセス命令がキャッシュにヒットし，フィル時から一度も参照していないワードを参照した場合を spatial-hit，参照したことのあるワードを参照した場合を temporal-hit とする．spatial-hit，temporal-hit と FAT，FACT の効果の対応関係は以下のようになる．FAT はデータレイアウトを変換することによって temporal affinity のある要素を同じキャッシュブロックに格納する．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が増す．また，キャッシュブロックの利用率が高まる．プリフェッチの効果増大はキャッシュのミス数を減らし，spatial-hit を増やす（プリフェッチの効果と呼ぶ）．利用率増大は一定時間あたりのキャッシュブロックの利用数を減少させ，temporal-hit 数を増やす（foot-print のコンパクションの効果と呼ぶ）．FACT は，FAT に加えてさらに圧縮を行うため，キャッシュブロックの実効サイズが増大し，temporal affinity のある要素が連続アドレスに配置されている際は，spatial-hit 数が増加する（ブロックサイズ増大の効果と呼ぶ）．また，圧縮によってキャッシュブロックの利用数が減少し，temporal-hit 数が増加する（圧縮の効果と呼ぶ）．この対応関係を使い，FACT，FAT の持つ複数の効果を，spatial-hit，temporal-hit の増加から観察する．FAT はヒープ上のデータに対してレイアウトの変換を行い，FACT はそのデータに対して圧縮を行うので，ヒープへのアクセスについて，キャッシュにおけるアクセス数の内訳を見てみる．

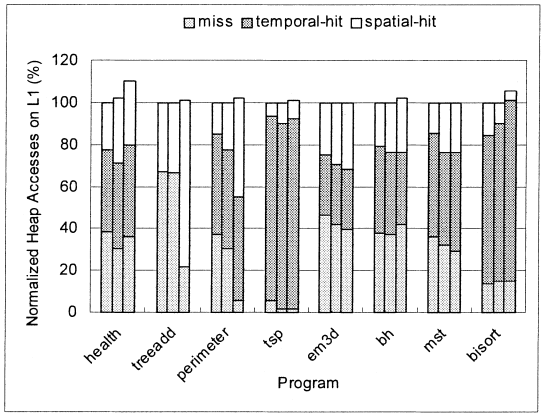

図 16 ヒープへのアクセスについて，1 次キャッシュへのアクセス数の内訳．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，FACT を適用した場合のアクセス数

Fig. 16 Breakdown of accesses to the primary data cache, which refers to heap data. In each group, each bar shows from the left, the breakdown of accesses of the baseline configuration, with FAT, and with FACT, respectively.

ヒープへのアクセスについて，**図 16** に 1 次キャッシュへのアクセス数，**図 17** に 2 次キャッシュへのアクセス数の内訳を示す．まず FAT を baseline と比較する．health，em3d については，1 次・2 次キャッシュにおいてミスが減り，spatial-hit が増えている．perimeter，tsp，mst については，1 次キャッシュにおいて，spatial-hit が増えている．treeadd については，2 次キャッシュにおいて，spatial-hit が増えている．bh については，2 次キャッシュにおいて，spatial-hit，temporal-hit ともに増えている．bisort についてはミスが増えている．以上より，FAT に関しては，主に spatial-hit が増加しており，プリフェッチの効果が大きいといえる．

次に FACT を FAT と比較する．treeadd については，1 次キャッシュにおいてミスが減り，spatial-hit が増えている．mst，tsp については，1 次キャッシュにおいて temporal-hit が増えている．perimeter については，1 次キャッシュにおいて spatial-hit，temporal-hit

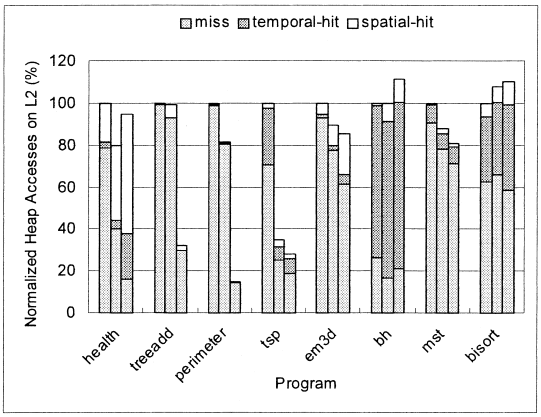

図 17 ヒープへのアクセスについて，2 次キャッシュへのアクセス数の内訳．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，FACT を適用した場合のアクセス数

Fig. 17 Breakdown of accesses to the secondary cache, which refers to heap data. In each group, each bar shows from the left, the breakdown of accesses of the baseline configuration, with FAT, and with FACT, respectively.

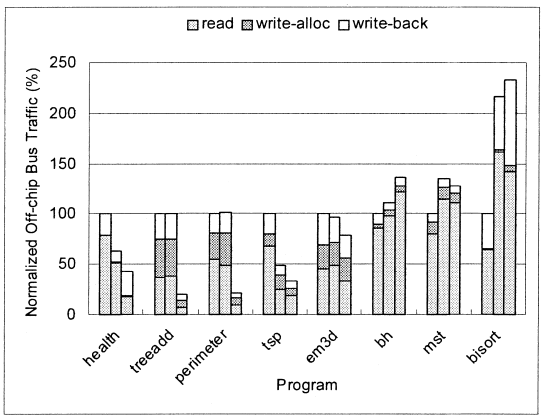

図 18 off-chip bus traffic の比較．棒グラフは各グループについて，左から順に，baseline の場合，FAT を適用した場合，FACT を適用した場合の off-chip bus traffic

Fig. 18 Comparison of off-chip bus traffic. In each group, each bar shows, from the left, the off-chip bus traffic of the baseline configuration, with FAT, and with FACT, respectively.

ともに増えている．`em3d` については，1 次・2 次キャッシュにおいて spatial-hit が増えている．2 次キャッシュにおいては temporal-hit も増えている．`health`，`bisort` については，2 次キャッシュにおいて spatial-hit，temporal-hit ともに増えている．`bh` についてはミスは増加している．以上より，FACT に関しては，プログラムに依存して，ブロックサイズ増大の効果，圧縮の効果の片方あるいは両方がみられる．

なお，1 次キャッシュのアクセス数が baseline より増加しているプログラムがあるが，これは，圧縮を試みたものの圧縮が不可能であったデータに対しては，圧縮データと，非圧縮データの両方を参照するためである．

5.5 off-chip bus traffic 比較

最後に，メモリコントローラと主記憶につながる off-chip bus の traffic の，圧縮による削減効果を見る．**図 18** に示すように，FACT は `bh`，`mst`，`bisort` 以外のプログラムの traffic を減少させ，平均では 13.4%削減する．キャッシュミスが減少すると traffic が減少することが主な理由で，traffic 削減度は速度向上度と対応する．しかし FACT は，圧縮したデータを圧縮したまま off-chip bus 上で授受するので，さらなる削減効果を持つ．この効果は特に `treeadd`，`perimeter` における write-allocate traffic の減少に表れている．`bh`，`mst`，`bisort` に関しては，グラフのノードのメモリ上の順序とたどる順序が異なるために，構造体内の temporal affinity しか利用できないが，FAT がこれを利用できなくするため FAT，FACT で traffic が増大する．

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．再帰的構造体の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，グラフをたど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 37.4%の速度向上，平均 41.6%のメモリデータ待ちサイクルの削減，平均 13.4%の off-chip traffic の削減をもたらすことを確認した．本稿の主な貢献は以下の 4 点である．

(1) FACT が従来の圧縮方法の限界 1/2 を超える，1/8 の圧縮率を達成した．これはデータレイアウト変換とキャッシュ上の圧縮データを指し示す方法の工夫による．

(2) 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．

(3) メモリ領域を圧縮前のデータ用，圧縮後のデータ用の 2 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．

(4) 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．

本手法は，キャッシュミス数，off-chip traffic をともに削減するため，性能向上という方向だけでなく，将来重要視される消費電力削減方法としても応用できる．

謝辞 初期草稿について有用なコメントをいただいた査読者の方々に感謝します．

参 考 文 献

1) Luk, C.-K. and Mowry, T.C.: Compiler based prefetching for recursive data structures, *Proc. Seventh International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.222–233 (1996).
2) Roth, A., Moshovos, A. and Sohi, G.S.: Dependence based prefetching for linked data structures, *Proc. Eighth International Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.115–126 (1998).
3) Truong, D.N., Bodin, F. and Seznec, A.: Improving cache behavior of dynamically allocated data structures, *Proc. 1998 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.322–329 (1998).
4) Lee, J., Hong, W.-K. and Kim, S.D.: An on-chip cache compression technique to reduce decompression overhead and design complexity, *Journal of Systems Architecture*, Vol.46, pp.1365–1382 (2000).
5) Yang, J., Zhang, Y. and Gupta, R.: Frequent value compression in data caches, *Proc. 33rd annual IEEE/ACM Intl. Symp. on Microarchitecture*, pp.258–265 (2000).
6) Larin, S.Y.: *Exploiting program redundancy to improve performance, cost and power consumption in embedded systems*, Ph.D. Thesis, ECE Dept., North Carolina State Univ., Raleigh, North Carolina (2000).
7) Zhang, Y. and Gupta, R.: Data compression transformations for dynamically allocated data structures, *Proc. Intl. Conf. on Compiler Construction*, LNCS, Vol.2304, pp.14–28, Springer-Verlag (2002).
8) Kjelso, M., Gooch, M. and Jones, S.: Design and performance of a main memory hardware data compressor, *Proc. 22nd EUROMICRO Conference*, pp.423–430 (1996).
9) Bonwick, J.: The slab allocator: An object-caching kernel memory allocator, *Proc. USENIX Conference*, pp.87–98 (1994).
10) Barret, D.A. and Zorn, B.G.: Using lifetime prediction to improve memory allocation performance, *Proc. ACM SIGPLAN' 93 Conf. on Programming Language Design and Implementation*, Vol.28, No.6, pp.187–196 (1993).
11) Compaq Computer Corporation: *Alpha 21264 Microprocessor Hardware Reference Manual* (1999).
12) Rogers, A., Carlisle, M.C., Reppy, J. and Hendren, L.: Supporting dynamic data structures on distributed memory machines, *ACM Transactions on Programming Languages and Systems*, Vol.17, No.2, pp.233–263 (1995).

(平成 15 年 2 月 2 日受付)
(平成 15 年 4 月 30 日採録)

高木 将通

1975 年生．1999 年東京大学理学部情報科学科卒業．2001 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年より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博士課程に在学．プロセッサアーキテクチャ，メモリ階層に興味を持つ．

平木 敬（正会員）

東京大学理学部物理学科，東京大学理学系研究科物理学専門課程博士課程退学，理学博士．工業技術院電子技術総合研究所，米国 IBM 社 T.J.Watson 研究センターを経て現在東京大学大学院情報理工学系研究科勤務．数式処理計算機 FLATS，データフロースーパーコンピュータ SIGMA-1，大規模共有メモリ計算機 JUMP-1 等多くのコンピュータシステムの研究開発に従事，現在は超高速ネットワークを用いる遠隔データ共有システム Data Reservoir システムの研究を行っている．