Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

オブジェクトの型を利用したキャッシュラインの重複除去によるハードウェアメモリ圧縮の高効率化

李袁忠[1,a)]　Li Zihan[1]　鵜川 始陽[1]　塩谷 亮太[1]

概要：現代では様々な機器や装置が組み込みシステムと呼ばれる小型のコンピュータシステムによって制御されている．組み込みシステムではC言語などの低級な言語による開発が現代でも主流であり，その開発効率が問題となっている．その理由の1つとして，組み込みシステムでは搭載されるプロセッサやメモリ量などの制約により，より高機能なスクリプト言語などの導入が困難なことがある．これに対し，本研究ではハードウェアによるメモリ圧縮により，スクリプト言語ランタイムによるメモリ使用量の削減を目指す．従来からキャッシュライン間のデータ重複を除去することによりデータを圧縮する手法が提案されているが，それらの手法ではランタイム上のデータ構造を把握できないため，効率的な重複の除去が行えなかった．これに対し，本論文ではランタイムとハードウェアを協調させることで，効率的なメモリの圧縮を行うAligned Objects Based De-duplication (AOBD)を提案する．AOBDでは，ランタイムがメモリ上の構造体やオブジェクトをキャッシュ・ライン単位に揃えて配置し，その上でそれらの型の情報をハードウェアに明示的に指示する．これにより，ハードウェアは同じ型のデータが格納されたライン間のデータの重複の効率的な除去が可能となり，高い圧縮率を実現する．組み込み向けのJavaScriptランタイムを使用した評価の結果，提案手法は既存手法と比べ平均22.3%圧縮率が向上した．

1. はじめに

現代では様々な機器や装置が組み込みシステムと呼ばれる小型のコンピュータシステムによって制御されている．特にインターネットと繋がった小型のInternet of Things (IoT)デバイスは急速に普及してきており，それらを制御する組み込みシステムの重要性は増す一方である．組み込みシステムの課題として開発効率がある．組み込みシステムは一般に機械に組み込まれるため，計算資源の制約が大きく，現在でもC言語などの低機能な言語を用いた開発が主流となっている．その一方で，開発効率の高いスクリプト言語などの採用は難しい．その主な理由の1つがメモリの使用量である．一般にスクリプト言語のランタイムが使用するメモリは，組み込みシステムが備えるメモリよりも大幅に大きい場合が多い．たとえば多くの組み込みシステムが数百KB程度のメモリを備えるのに対し，PCやサーバー向けのスクリプト言語ランタイムでは起動直後でさえも数MB程度のメモリを消費してしまう．一部，組み込みシステム向けにメモリ使用量を大きく減らしたランタイムも提案されているが，それらにおいてもメモリ使用量は依然として重要な課題である．この問題に対し，本研究ではハードウェアによる**メモリ圧縮**により，スクリプト言語のランタイムのメモリ使用量の削減を目指す．ハードウェアによるメモリ圧縮では，ソフトウェアから見て透過的にデータを圧縮してメモリに格納する．メモリ圧縮には様々な手法 [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11] が提案されているが，本研究ではその中でも特に圧縮率の高いBase and Compressed Difference de-duplication (BCD) [12] に着目する．BCDでは比較的単純なハードウェア構造にも関わらず高い圧縮率を達成する．しかし，ハードウェアに閉じた手法であるため，ソフトウェアからみたデータ構造を考慮できず，圧縮が効率的でない場合がある．たとえば，BCDでは「同じ型の構造体同士は中身が似ている」といった知識の利用はできず，非効率な圧縮を行っている場合がある．

そこで，本研究ではランタイムとハードの協調により，データ間の類似性をよりよく考慮することで効率的な圧縮を行うAligned Objects Based De-duplication (AOBD)を提案する．我々はオブジェクト指向言語における一連のデータをまとめたデータ構造である**オブジェクト**に着目する．このオブジェクトとその型に基づき，以下の2点から成る手法を提案する．

(1) ランタイムにより各オブジェクトをキャッシュのライン境界に合わせて配置する．これによりラインとオブ

[1] 東京大学 大学院情報理工学系研究科
[a)] yukitada@rsg.ci.i.u-tokyo.ac.jp

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

ジェクトを 1 対 1 で対応付けることが可能となる．以降ではこの配置を行うことをアラインメントと呼ぶ．

(2) 各ラインのオブジェクトの型をランタイムがハードウェアに明示的に伝える．これにより，ハードウェアから各ライン上にあるオブジェクトの型を認識することが可能となる．

これらの結果，ハードウェアより同一の型のオブジェクトがあるライン同士で重複を除去することが可能となる．一般に，同じ型のオブジェクトでは内容が似ていることが多く，これにより効率的な圧縮を実現する．組み込み向けの JavaScript ランタイムを使用した評価の結果，提案手 AOBD は既存手法 BCD と比べ平均 22.3%，最大 52.1%圧縮率が向上した．

2. Base and Compressed Difference deduplication (BCD)

本章では，提案手法の背景としてハードウェアによりメモリ圧縮を行う BCD を説明する．BCD は似た値を格納したライン間の重複を除去する圧縮手法である．また，signature と呼ばれるラインの特徴を表した値の一致に基づいて，似ているラインを検出する．BCD の圧縮は以下の 3 つのステップにより行われる．

(1) signature の一致に基づく似ているラインの検出

(2) 似ているライン間の差分の圧縮

(3) 圧縮データの格納

本章では，上記の 3 つステップについてそれぞれ説明する．BCD では SPEC2017[13]，DaCapo[14]，TPCDS[15]，TPC-H[16] をベンチマークとして使用し，既存の圧縮手法 Compresso[1]，Compresso deduplication と比べ，それぞれ平均 32.3%，14.9%高い圧縮率を達成した．

2.1 signature の一致に基づく似ているラインの検出

似た値のデータの上位部分は同じ値になることが多いため，BCD では値の上位部分を signature として使用する．BCD ではキャッシュラインが 64 バイトである場合に，これを 8 つのワードに分割する．各ワードは 8 バイトであり，それらのワードの上位 2 バイトを結合した 16 バイトを signature として使用する．

図 1 はオブジェクトと signature の生成の例を示す．上段はランタイム上のオブジェクトを定義したソースコードを表す．ここでは sold_time, ... の 4 つの要素を持つ Object という型を定義し，その型を用いて A と B の 2 つの値を定義している．下段は定義したオブジェクトがライン上に配置された様子を示す．ここでは 4 バイトの各ワードの上位 1 バイトを signature としている．A と B はお互いに似た値を保持しているが，このような場合，ワードの上位バイトの部分が一致することが多い．signature はこの赤文字で示した上位バイト部分に対応し，各ワードごとの上位バイトの一致によりラインの類似を検出できる．

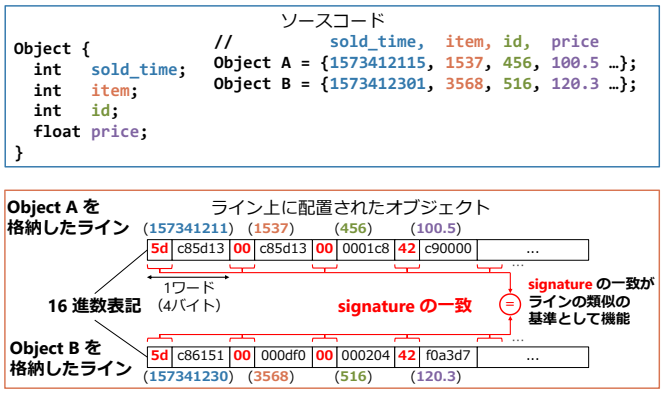

図 1: オブジェクトと signature の生成例

2.2 似ているライン間の差分の圧縮

本節では 2.1 節で述べた方法に従ってラインの signature を求めた後，signature が一致したライン間で XOR 演算を行い差分を求める．XOR 演算ではお互いの入力の同じ部分が 0 に，違う部分が 1 となるので，一般的に似ているデータ間の XOR の結果はゼロが連続する．差分内の各ビットにおいて連続するゼロをその連続個数にエンコードし圧縮する．このような連続したゼロの，個数へのエンコーディングは Leading Zero Count (LZC) 圧縮と呼ばれる．

詳細な動作を図 2 を用いて説明する．この図は 2 つのラインの間で差分をとる場合の例である．後述するように BCD では基準となるラインはそのままの形でメモリ上に格納し，それに似たラインは基準となるラインからの差分を圧縮して格納する．ただし，後述する圧縮アルゴリズムによる差分の圧縮では，圧縮後に元の差分のサイズを超えることが稀にあるため，その場合は圧縮を行わずに差分を格納する．この図では上側が基準となるライン，下側が新しく圧縮するラインである．基準となるラインを Base Line，圧縮された差分（または圧縮後元のサイズを超えたため圧縮されなかった差分）を Diff Line と呼ぶ．BCD ではキャッシュラインが 64 バイトである場合に，これを 8 つのワードに分割する．ラインの各ワードの signature を除いた 6 バイト（図の 6 B と書かれた青ブロック）間の差分として XOR（図の 6 B XOR と書かれた緑ブロック）を求める．そして求めた各ワードの差分を，LZC 圧縮によって先頭のゼロをその個数による表現でエンコードする．最後に各ワードの圧縮された差分を結合し，それをメモリに格納する．ただし，上述したように，LZC 圧縮による圧縮は圧縮後の差分が元の差分のサイズを超えることがあるため，その場合は圧縮を行わずに差分を格納する．これらは

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

それぞれ 2.3 節で説明する Base Array，Diff Array に格納されている．

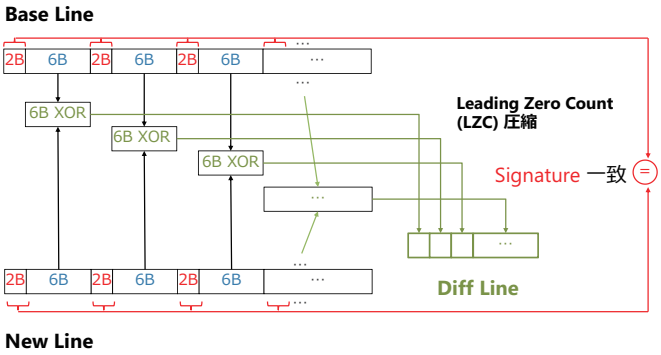

図 2: BCD での signature の一致に基づく重複除去

2.3 圧縮データの格納

BCD では，圧縮後のデータは以下に示すメモリ上の 3 つのテーブルを用いて格納される．

- **Translation Table**：ラインのアドレスをインデックスとして，Base Line と Diff Line へのポインタを格納するテーブル．
- **Base Array**：Base Line を格納するテーブル．
- **Diff Array**：Diff Line を格納するテーブル．

本節では上述の 3 つのテーブルについて説明する．補足として，メモリ上には他にも Overflow Region と呼ばれる (2)，(3) に入り切らなかったデータを格納する領域がある．

Translation Table はラインのアドレスをインデックスとして，エントリに Base Line と Diff Line へのポインタ及び圧縮と展開に必要なメタデータを格納する．エントリの構造を図 3 に示す．エントリでは Base Pointer，Diff Pointer 以外に Diff Valid と Diff Compressed と呼ばれるメタデータが格納される．Diff Valid は Diff Line への参照の有無を表す．参照がない場合は参照した Base Line そのものがラインの値をとなる．Diff Compressed は重複除去が行われた後の差分が圧縮されたかを示す．

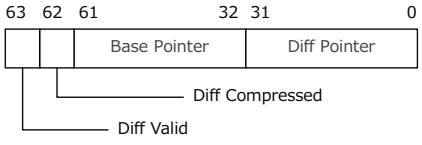

図 3: BCD Translation Table Entry

BCD ではライン 1 つに対して Translation Table の各エントリが対応する．各エントリのサイズは 8 バイトであり，ラインサイズが 64 バイトであることから，有効なメモリ空間の 1/8 の大きさが Translation Table に必要となる．BCD では圧縮の効果がこの大きなオーバーヘッドを上回るため，これを許容している．

Base Array は Base Line を格納するテーブルである．BCD ではラインの signature からハッシュ値を生成し，そのハッシュ値を Base Array のエントリを参照するインデックスとして使用する．Base Array のエントリは Bucket と呼ばれるデータ構造をもち，その中に入る各 Base Line に対応するデータを Way と呼ぶ．Way の数は，Base Array に割り当てるメモリの大きさによって設定される．

図 4 は BCD が論文内で仮定している 32 Way の Base Array を示す．この図のように，各 Bucket は signature のハッシュ値が同一となる Base Line 群を各 Way に格納する．また，それぞれの Way は格納された Base Line の参照カウントを保持する．図では Ref-Line がこれに対応する．この参照カウントは，対応する Base Line を参照して圧縮を行っているラインの数を記録している．参照カウントがゼロの場合，対応する Base Line を使用しているラインは存在しないため，安全に Bucket から追い出すことが可能である．

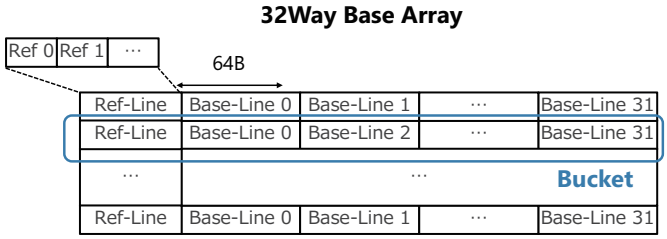

図 4: BCD Base Array

Diff Array は，Diff Line を格納するテーブルである．このハッシュテーブルは Base Array と似たよな構造を持ち，Bucket と Way によってデータを格納する．BCD では Diff Line は圧縮結果のサイズに基づき，16 バイト，32 バイトの圧縮ありと 48 バイトの圧縮なしの 3 種類のサイズのいずれかで格納される．Diff Array では，Diff Line の値そのものからハッシュ値を算出し，そのハッシュ値をテーブル内の Bucket へのインデックスとして使用する．Base Array と同様に各 Way には参照カウントがあり，これは格納されている Diff Line がいくつの非圧縮ラインから参照されているかを保持している．また，Base Array 同様，参照カウントがゼロになると対応する Diff Line を参照している非圧縮ラインは存在しないことを意味するため，安全にその Diff Line を追い出すことができる

一方，Base Array と異なり，Diff Array では 3 種類のサイズの Diff Line を扱うため，Block と呼ばれる 16 バイトの単位で Diff Line を管理する．図 5 は BCD が論文内で仮定している 51 Way，120 Block の Diff Array を示す．

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

51 Way，120 Block は最大で Diff Line を 51 個かつ，その合計のサイズは 120 Block 以下を格納できる Bucket を意味する．また参照カウント以外に，Diff Line のサイズの Block 数もメタデータとして格納されている．図の Size がそれに対応する．

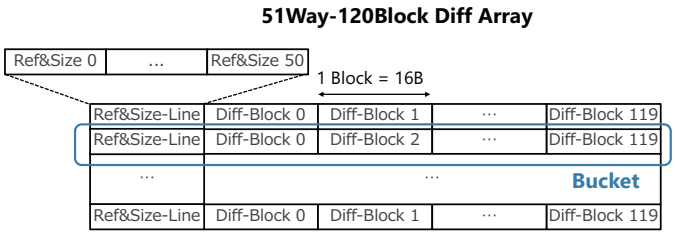

図 5: BCD Diff Array

以上で説明した 3 つのテーブルにおける圧縮データの格納の様子を図 6 で示す．この図では 3 つのライン A，B，C は同じ signature を持ち，同じ Base Line を共有する．その Base Line との差分が圧縮されて Diff Array に Diff Line として格納される．また，図 7 は簡略化したメモリの構成における図 6 にある 3 つのライン A，B，C の特にライン A における圧縮の手順を示す．この図ではラインは 16 ビット，signature はラインの上位 4 ビットと仮定している．また Base Array では，Bukect には 1 つの Way（Base Line）しかないとする．圧縮の手順として，まずはライン A の上位 4 ビットの signature のハッシュ値をインデックスとして Base Array にある Bukect を参照する．次にライン A と signature が一致する Base Line との差分として XOR 値を計算し，LZC 圧縮による圧縮を行う．最後に圧縮した差分を Diff Array に格納する．

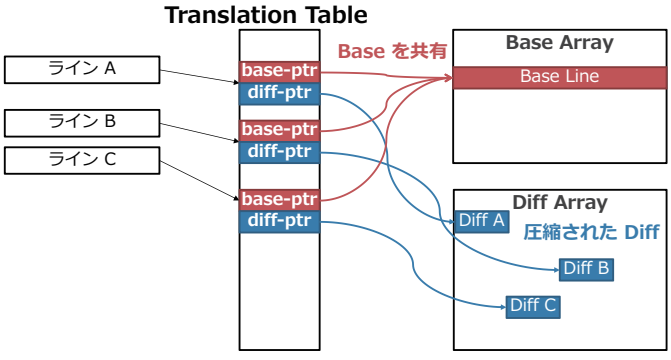

図 6: BCD 圧縮データの格納

3. Base and Compressed Difference deduplication (BCD) の問題

この章では第 2 章で説明した BCD の問題を述べる．BCD ではオブジェクト間にあるデータの類似性を捉えられないことによる問題がある．1 章で述べたように，オブジェクトは関連するデータをまとめて 1 つのデータの塊にしたものを指す．オブジェクトの型は，そのオブジェクトがどのようなデータをまとめているかを示す属性となり，型が同じオブジェクトは中身が似ている事が多い．

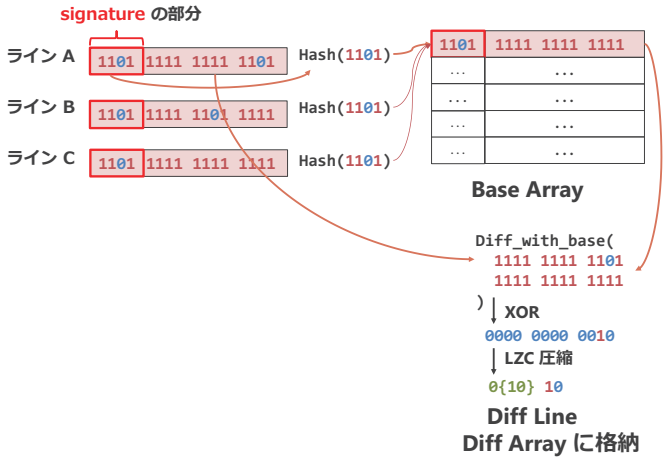

図 7: BCD 圧縮の手順

具体的な問題として以下の 2 つがある．3.1 節，3.2 節で 2 つの問題についてそれぞれ述べる．

- 同じ型のオブジェクトの配置がライン上でずれる．
- 異なる型のオブジェクト間で差分が取られる．

3.1 同じ型のオブジェクトの配置がライン上でずれる

多くの場合，同じ型のオブジェクトはラインの異なる位置に配置される．図 8 はオブジェクトの配置とオブジェクトの値を示す．上段がメモリ上のオブジェクトの配置の一部分を示しており，型 A，B のオブジェクトが配置されている．この図は eJSVM [17] と呼ばれる組み込み向けの JavaScript ランタイムでのオブジェクトの配置に基づいている．下段が上段の赤枠で囲んだ部分を拡大したもので，各オブジェクトに格納されている値を示す．

図の下段の 2 つの型 A のオブジェクトに格納されている値に着目する．赤い文字の各ワードの上位バイトが signature を表す．この時，似ている値のワードの位置がライン間で異なるため，ライン間で signature が一致しない．この場合，同じ型 A のオブジェクト同士で中身が似ているが，BCD では重複を除去のラインとして選択されない．そのため，BCD の圧縮ではこのような重複は除去されず，圧縮効率が低くなる．

3.2 異なる型のオブジェクト間で差分が取られる

BCD では異なる型のオブジェクトが格納されたライン間でも signature が一致した場合は圧縮が行われる．

図 9 を用いてそのような例を説明する．この図は 2 つの異なる型 Type A，B のオブジェクトが格納されたラインを示す．図のように Type A，B と異なる型のオブジェク

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

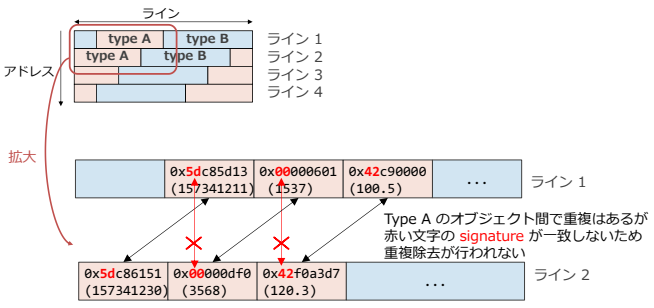

図 8: オブジェクトの配置と signature の不一致

トが格納されたライン間でも赤文字で示した signature が一致する場合がある．特に整数データにおいて，絶対値が小さい値の上位バイトはすべて 0 また 1 であるため，ライン上の各ワードの上位がすべて 0 または 1 であることは多い．図のように Type A，B のオブジェクトの各ワードの上位バイトが全て 0 である時，それらが格納されたライン間では赤い文字の signature は一致する．しかし，異なる型のオブジェクトが格納されているため，ライン間では signature 以外のデータが大きく異なる場合がある．このように signature が一致しても，その他のデータが大きく異なる場合，重複は少なく圧縮が効率的ではない．

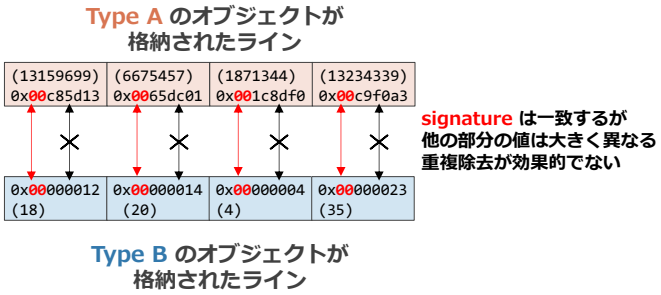

図 9: 異なる型のオブジェクトが格納されたライン間の signature の一致

4. 提案手法：Aligned Objects Based De-duplication (AOBD)

本研究ではオブジェクト間の類似性を考慮することで，効率的な重複除去を行うメモリ圧縮手法 Aligned Objects Based De-duplication (AOBD) を提案する． ランタイムとハードの協調によって提案手法 AOBD を実現する．ランタイムでは以下の 2 つを行う．

- メモリ割り当て時に，オブジェクトをラインの先頭に揃えて配置する．このことをアラインメントと呼ぶ．
- ハードウェアにライン（＝オブジェクト）ごとの型情報を指示する．

オブジェクトのアラインメントによって，1 つのラインに対して 1 つのオブジェクトが格納されることとなり，オブジェクトをライン単位で扱うことが可能となる．このことを上述のようにライン（＝オブジェクト）と表記する．またハードでは以下の 2 つを行う．

- ランタイムから指示された同じ型のライン（＝オブジェクト）間で重複除去を行う．
- オブジェクトのアラインメントによって生じる未使用の隙間領域を別途圧縮する．

本章では，まず 4.1 節でオブジェクトのアラインメントを説明し，4.2 節でライン（＝オブジェクト）ごとの型情報を指示することを説明する．そして，4.3 節でオブジェクトのアラインメントよって生じる未使用の隙間領域の圧縮を説明する．ランタイムから指示された同じ型のライン（＝オブジェクト）間での重複除去については，ラインの選択基準以外は基本的に BCD と同様である．

4.1 オブジェクトのアラインメント

オブジェクトのアラインメントによって，1 つのラインに対して 1 つのオブジェクトが格納されることとなり，オブジェクトをライン単位で扱うことが可能となる．図 10 はオブジェクトのアラインメントの様子を示す．上が通常のメモリ配置で，下がオブジェクトのアラインメントによるメモリ配置となる．これによって，オブジェクトの型情報もラインに対応するものとなる．

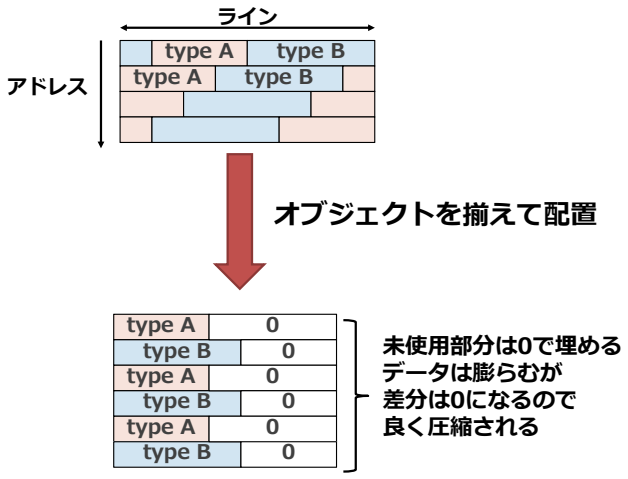

図 10: オブジェクトのアラインメント

また，オブジェクトのアラインメントによってラインの後方には隙間領域が生じ，データが膨らむ．そこで，後方の隙間領域は，図で示すようにゼロ埋めを行う．これによって，同じ型のライン（＝オブジェクト）間で比較した時に，この部分の差分はゼロとなり，非常に良く圧縮される．よって，データが膨らむことによるオーバーヘッドはほとんどない．また後述の 4.3 節で説明するように，この領域自体を圧縮して格納することも可能である．

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

4.2 ハードウェアにラインの型情報を指示

本節ではランタイムからライン（＝オブジェクト）ごとの型情報を指示する方法を説明する．BCD では重複除去対象となるラインの選択は，ラインの一部を取り出した signature に基づいて行う．一方，AOBD では，オブジェクトごとの型情報を指示して，それに基づいて重複除去対象のラインを選択する．その方式として以下の 2 つを提案する．

(1) **間接指定方式**: ランタイムから signature として取り出す部分のラインの範囲を指定する．そして，ハードウェアではその指示に基づき，指定された範囲の値を signature として使用する．

(2) **直接指定方式**：ランタイムからラインごとの型情報の ID を signature として直接指定する．そして，ハードウェアではその直接指定された signature を使用する．

それぞれの方式について 4.2.1 節，4.2.2 節で詳しく説明する．

4.2.1 間接指定方式

一般的に，ランタイムは，各オブジェクトが自身の型情報の ID を内部に含む実装になっていることが多い．AOBD ではオブジェクトのアラインメントにより，ライン内の ID の位置は固定される．間接指定方式では，ランタイムからハードウェアへとライン上にある ID 部分の範囲の指定を行い，固定された位置にあるライン内の ID を取得する．

例えば，組み込みシステム向けの JavaScript ランタイムである eJSVM[17] では，オブジェクトの先頭の 8 から 15 バイト部分が型 ID として利用できる．この部分は，オブジェクトの型を表す構造体である Hidden Class へのポインタを格納している．図 11 はその様子を示す．図の上段のラインは間接指定方での signature の指定を示し，下段のラインは BCD の signature の指定を示す．このように赤文字の type ID=2 の位置をランタイムからハードに指定し，ハードで signature として使用する．一方，BCD では各ワードの上位 2 バイトを signature として使用するため，オブジェクトの型による判定が行えない．

間接指定方式の利点は，ランタイムにオブジェクトのアラインメントの実装をすることで AOBD が実現可能となり，ランタイムの修正が少ない．一方，問題としては以下の 2 つがある．

(1) ライン上にオブジェクトの型の ID が簡単にわかる形で埋め込まれている必要がある．

(2) 2 ライン以上に渡るオブジェクトでは 2 ライン目以降は型情報の取得が不可能である．

(1) については，間接指定方式ではハードウェアがランタイムから指示された範囲から ID の取得を行うため，ID が

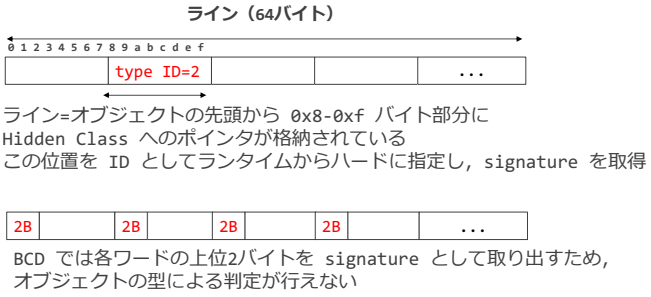

図 11: 間接指定方式による signature の指定

複雑に埋め込まれた場合ハードウェアの負担が大きくなる．(2) については，図 12 が示す様に，2 ライン目以降には型の ID が含まれない．よって，ハードウェアで取得する指定された位置のデータは ID のデータではない．また，ハードウェアでは取得した指定された範囲にあるデータが ID であるかの判断はできない．そのため，ID ではないデータを signature として使用したとき，効率的な圧縮とならない可能性がある．

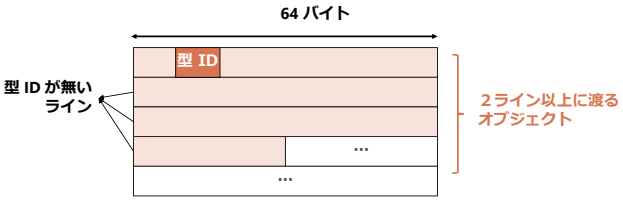

図 12: 2 ライン以上に渡るオブジェクト

4.2.2 直接指定方式

直接指定方式ではランタイムで以下の 3 つを行うことで，明示的にハードウェアに型情報の ID を signature として指定する．

- オブジェクトの型情報の ID と，オブジェクトの何ライン目にあるかの情報を用いて sigature を計算し，明示的にラインに割り当てる．
- ラインに signature を割り当てる際は，ラインのアドレスとランタイムで計算した signature を引数とする API を使用する．API 内では，専用命令によってハードウェアにラインの signature を伝える．
- ハードウェアは受け取った signature をラインごとに設定する．signature の設定の具体的な動作として，ハードウェアはラインに対応する signature を Translation Table のエントリのメタデータとして格納する．

直接指定方式の利点は 2 つある．1 つはオブジェクト内に型 ID を明示的に格納しないランタイムであっても，signature を計算する関数の実装を行うことで，ラインごとの signature の生成と割り当てが可能となる．もう 1 つ

Vol.2022-ARC-248 No.20
Vol.2022-SLDM-198 No.20
Vol.2022-EMB-59 No.20
2022/3/10

は，ランタイム側で型以外の特徴を signature に埋め込むことで，圧縮効率を向上させることができる．図 13 を例として用いて説明する．この図は 2 ライン以上に渡るオブジェクトの各ラインに対して signature の割り当てを行う様子を示す．左側の赤文字が各ラインの signature を表す．この様に，型 ID である type_id とオブジェクトの何ライン目かを示す line_offset を引数として signature を算出する make_sig 関数を設定することで，2 ライン目以降も圧縮に効果的な signature を割り当てることができる．問題として，直接指定方式では signature をメタデータと

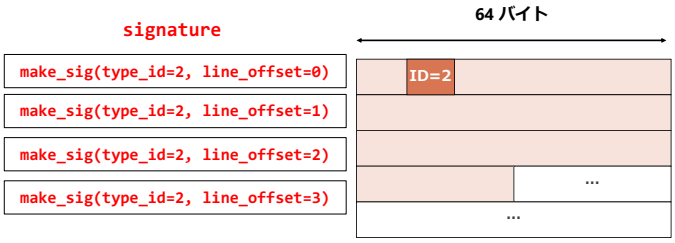

図 13: 2 ライン以上に渡るオブジェクトへの signature の割り当て

して格納するための容量が追加で必要になる．BCD と間接指定方式は直接指定方式と異なり，signature はラインのデータそのものから計算される．そのため，ライン上の signautre の範囲を指定することで signature を使用でき，追加で signature を格納する容量が必要ない．一方，直接指定方式はランタイムによって計算された signature を使用する．これはライン上のデータによって求まるものではないので signature そのものを格納する必要がある．

4.3 アラインメントよる未使用の隙間領域の圧縮

本節でオブジェクトのアラインメントよって生じる未使用の隙間領域の圧縮を説明する．具体的には Base Line に対する圧縮を行う．図 14 を用いて説明する．図の左下の様に，オブジェクトのアラインメントによってラインの後方には未使用のゼロ埋めされた隙間領域が生じる．Base Line でも後方はゼロ埋めされた領域となるため，ここの部分に対して圧縮を行う．ゼロ埋めされた領域の連続したゼロをその個数によって表現してエンコードする単純な圧縮を Base Line に適用する．Base Line の圧縮後のサイズは可変のため，Base Array は Diff Array と同様な仕組みで圧縮後の Base Line を格納する．

Base Line の圧縮は後方の未使用のゼロ埋めされた隙間領域に対して行うものであるため，図の右上にある様な通常のメモリ配置ではこの圧縮は有効ではない．よって本研究では Base Line の圧縮を，オブジェクトのアラインメントを行うメモリ配置の圧縮手法のみのオプションとして追加する．

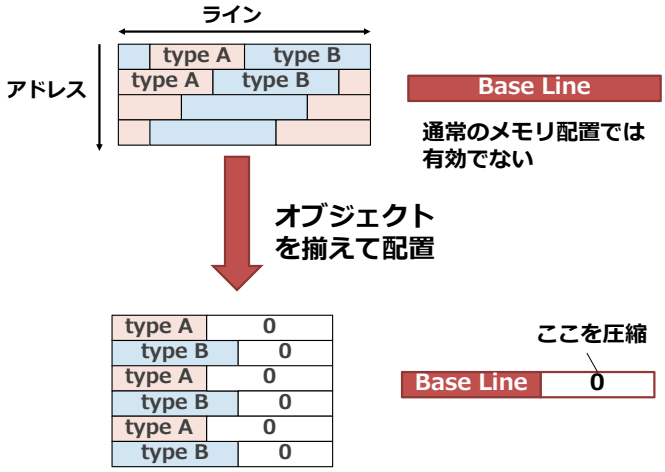

図 14: アラインメントのメモリ配置での Base Line の圧縮

5. 評価

本章では，4 章で説明した提案手法 AOBD と既存手法 BCD の評価を行う．本章ではまず 5.1 節で評価環境を説明し，次に 5.2 節でメモリ使用量の評価方法を説明する．そして最後に 5.3 節で評価結果について述べる．

5.1 評価環境

本研究では，オブジェクトの型を動的に判別して最適化する機能を持つ，組込みシステム向け JavaScript ランタイムの eJSVM に提案手法を実装した．そして，圧縮率をシミュレーションによって評価した．ベンチマークは Cross-Language Compiler Benchmarking[18] から 9 つのプログラムを選択し使用した．プログラムの振舞を解析ツール Pin[19] を使用し，ヒープ領域への Read/Write アクセスをトレースとして記録したトレースを用いて評価を行った．その生成したトレースを入力として動的にプログラムを実行したときの圧縮をシミュレーションした．圧縮の手法として表 1 で示した 7 つに対して評価を行なった．

表 1: 評価する圧縮手法

AL: オブジェクトのアラインメント
BL: Base Line に対して圧縮を行う

手法	メモリ配置	型の指定	Base Line 圧縮
BCD	通常	N/A	N/A
BCD-AL	アラインメント	N/A	×
BCD-AL-BL	アラインメント	N/A	○
AOBD-INDIRECT	アラインメント	間接	×
AOBD-INDIRECT-BL	アラインメント	間接	○
AOBD-DIRECT	アラインメント	直接	×
AOBD-DIRECT-BL	アラインメント	直接	○

5.3.2 節での評価を除き，本節では典型的な汎用コンピュー

タのメモリサイズを想定して Base Array，Diff Array にそれぞれ 2 GB のメモリを割り当てた．また Way の数は 32，Block の数は 256 と設定した．本研究では既存手法 BCD，提案手法 AOBD のどちらも Block の大きさは 8 バイトとする．これは 8 バイトと粒度を小さくすることで，圧縮後のラインのサイズの分布をより詳細に把握するためである．Base Line を圧縮する場合の Base Array や Diff Array のように可変サイズのラインを格納する場合は，Way の数は Block の数と等しくする．これよって，小さいサイズのラインが格納されて Way の数の上限に達することによる Bucket に使用されてない空き領域が生じるのを防ぐ．

5.2 メモリ使用量の評価方法

BCD の論文の圧縮率の評価方法では，基準とするメモリ使用量に，メモリ圧縮の効果を純粋に測るには不適切なメモリ領域を含む．それを説明するためにまずはメモリ空間の領域がガベージコレクション (GC) を行うランタイムからはどのような種類に分類されるかを説明する．メモリ上にある空間は Reachable，Unreachable，Free の 3 種類に分類される．

- **Reachable**: ランタイムから到達可能な生きているオブジェクトを格納．到達可能とはプログラムで使用されていることを意味し，到達可能なオブジェクトは Garbage Collection (GC) によって回収されない．
- **Unreachable**: ランタイムから到達不可能なオブジェクトを格納．GC を行うことでそのオブジェクトが回収される．
- **Free**: ランタイムのプログラムの実行では使用されてはいないが，使用可能な空き領域として登録されている．一般的に初期化によるゼロ埋めなどが行われている．

以降，Reachable，Unreachable，Free space のそれぞれの総サイズを R，U，F で表す．またプログラムの実行によって使用されたメモリ領域の総サイズを O で表す．O はメモリ上で Free な空間以外の総サイズ，つまり R と U の和に等しい．メモリ全体の総サイズは H で表す．各ラベルの意味を表 2 にまとめる．

表 2: ラベルが示す値

ラベル	値
R	Reachable なメモリ領域の総サイズ
U	Unreachable なメモリ領域の総サイズ
F	Free なメモリ領域の総サイズ
O(=R+U)	プログラムの実行で使用されたメモリ領域の総サイズ
H(=O+F)	メモリ全体の総サイズ

本研究と BCD の論文ではメモリ使用量の基準が異なっている．既存研究は Reachable な領域と Unreacheble なメモリ領域の合計である O を用いて評価を行っていたの対し，本研究では Reachable なメモリ領域の使用量 R を用いる．Unreacheble なメモリ領域は GC の挙動（タイミング）に依存してメモリ使用量が変わってしまい，メモリ圧縮の効果を純粋に測ることができない．以上より本研究ではメモリ使用量の基準は R とし，ラベルを用いて圧縮率の式を示すと以下の式 1 となる．

$$\text{圧縮率} = \frac{\text{Compress(R)}}{\text{R}} \tag{1}$$

5.3 評価結果

本節では評価結果について述べる．まず 5.3.1 節で提案手法 AOBD と既存手法 BCD に対して，Base Array と Diff Array に十分なメモリ（2 GB）を割り当てた条件での圧縮率の比較結果を示す．そして，5.3.2 節ではメモリのサイズを小さくしていったときの圧縮率を示す．

5.3.1 BCD との圧縮率の比較

図 15 はベンチマークに対する，各手法の圧縮率を示す．x 軸がベンチマークと各手法を表し，y 軸が圧縮率を表す．このように，全てのベンチマークで赤い AOBD の方が優れていることがわかる．AOBD の間接，直接指定方式の平均圧縮率はそれぞれ 75.3%,76.9%，最高圧縮率は 44.9%，45.0%を達成した．

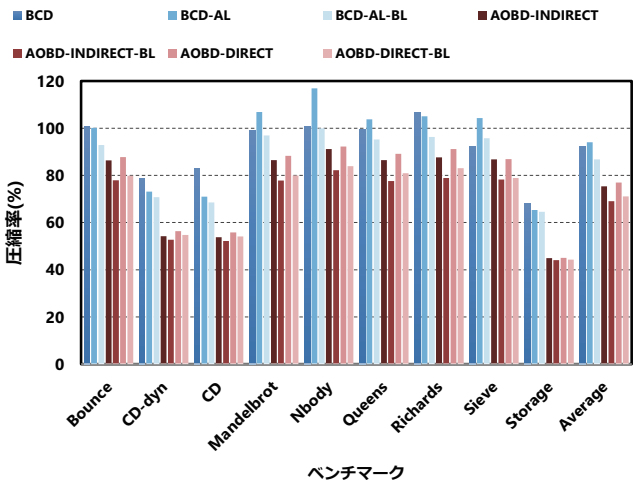

図 15: 圧縮率 (Base Array: 2GB，Diff Array: 2GB)

図 16 は各手法の平均圧縮率の内訳を示す．x 軸が手法，y 軸が圧縮率を表す．青が Translation Table，赤が Base Array，緑が Diff Array を示す．このグラフより，AOBD では Diff Array の占める割合が小さくなっており，重複除去が効果的に行われていることがわかる．

17 は，圧縮が効果的だったベンチマーク CD.js のラインの圧縮後サイズの分布を示す．このグラフは Base Line と Diff Line の合計を集計したものである．x 軸が圧縮後のラ

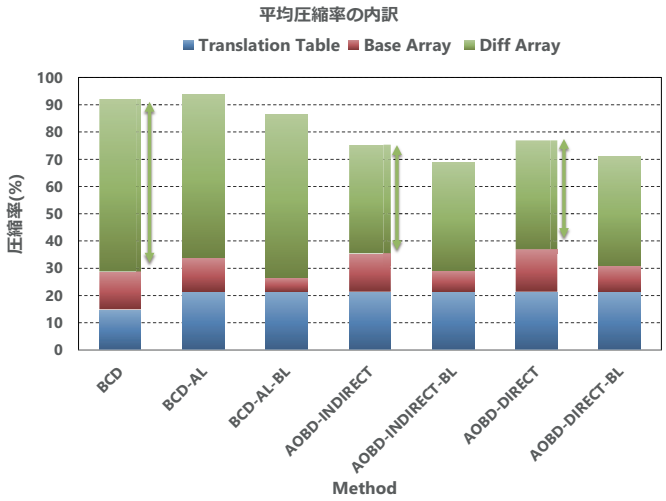

図 16: 平均圧縮率の内訳 (Base Array: 2GB，Diff Array: 2GB)

インのサイズで，y 軸がラインの個数を表す．上段にある 2 つのグラフ，BCD と BCD-AL で比較すると，BCD-AL の方がグラフが左によっていることから，アラインメントが圧縮の効率を高めていることがわかる．また，AOBD の手法と BCD の手法を比較すると，明らかに，AOBD の方が小さく圧縮されたラインの数が多く，AOBD の方が圧縮が効率的であることがわかる．

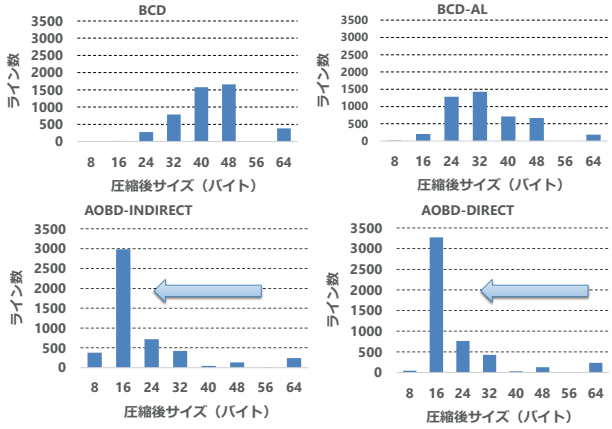

図 17: CD.js ラインの圧縮後サイズの分布 (Base Array: 2GB, Diff Array: 2GB)

5.3.2 小さいメモリへの圧縮の適用

既存手法 BCD や提案手法 AOBD では Base Array と Diff Array に割り当てるメモリのサイズによって圧縮の効率が変わるため，それらのサイズを変化させて評価を行った．ここでは，典型的な汎用コンピュータのメモリサイズから組み込みシステムのメモリサイズまで，幅広いサイズのメモリに対して圧縮手法を適用し，評価した．これによりメモリのサイズを変化させた時の影響と圧縮手法が効果的なメモリのサイズを確認する．

また，提案手法や BCD ではメモリサイズごとに，適切な Base Array や Diff Array のサイズ，それらのテーブル内の適切な Way 数や Block 数などのパラーメータは異なる．そこで本研究では，以下の表 3 に示すメモリサイズや，そのメモリサイズごとの Way 数や Block 数の構成を評価した．これらは本来はさまざまなパラメータを試行し適切なものを使用するべきであるが，それらについてはまだ結果が得られていない．そのため，ここでは BCD の研究で用いられているパラメータを元に，メモリサイズを用いてスケーリングした値を用いた．なお，BCD の既存研究では Base Array と Diff Array に割り当てるメモリのサイズを 2GB と固定し，そのメモリのサイズで圧縮が効果的となる Way の数，Block の数を設定して評価を行なっていた．

表 3: メモリのサイズと Way の数，Block の数の関係

サイズ	Way	Block
64 MB 以上	32	256
1 MB 以上 64 MB 未満	16	128
32 KB 以上 1 MB 未満	8	64
32KB 未満	4	32

本節ではメモリを小さくして評価を行なっていくため，Overflow Region へのラインの格納が生じる．そこで，圧縮が有効に行われているかどうかを判定するために圧縮率以外に Overflow Region に格納されたラインのデータ量に着目する．Base Array，Diff Array と異なり Overflow Region で格納されたラインは共有されない．そのため，Overflow Region にあるラインが多いほど，圧縮の効率は低下する．

Base Array に割り当てるメモリのサイズを小さくしていき，圧縮が効率的に機能する Base Array のサイズの探索を行なった．順に実験の過程と結果を以下に述べる．

- 実験では Base Array を 32 KB まで縮小しても，全ベンチマークでほとんど Overflow Region への格納が発生しなかった．
- そこで, Base Array を 32 KB と設定し, 今度は Diff Array のサイズを変化させて評価を行なった．
- 結果 Storage.js を除いたベンチマークプログラムでは Diff Array のサイズを 256 KB まで縮小しても AOBD ではほとんど Overflow Region への格納が起きなかった．
- そこで，Storage.js の評価結果を見ていき，メモリのサイズの変化による影響を確認した．
- Storage.js はベンチマークプログラムの中で，5.2 節で説明した，ランタイムが到達可能（Reachable）なオブジェクトの総サイズ R が最も大きいベンチマークプログラムであった．Storage.js では R は約 450 KB で

ある．

図 18 は Storage.js で Diff Array を 1 MB と設定した時の圧縮結果を示す．図の紫色は Diff Array から溢れたラインの Overflow Region へ格納されたデータ量の割合を示す．図では Diff Overflow とラベルをつけている．図より以下のことがわかる．

- BCD の手法では Overflow Region へのデータの格納が多く起きている．
- AOBD の手法では Overflow Region へのデータの格納がほとんんど起きてない．
- AOBD の手法は BCD の手法と比べ効果的に圧縮が行えている．

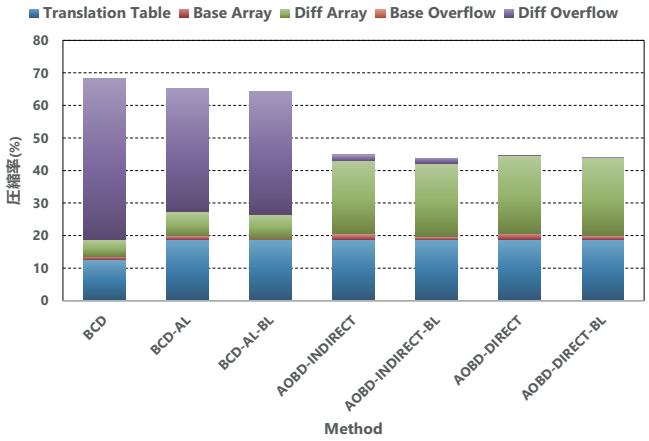

図 18: Storage.js 圧縮結果 (Base Array: 32 KB, Diff Array: 1 MB)

Diff Array のサイズが R より大きいのにもかかわず，Overflow が発生するのは，プログラムの実行時には R の役 2〜3 倍のメモリ空間がランタイムで使用されているためである．プログラムの実行時に Overflow Region に格納された Reachable なオブジェクトは，GC 後も Overflow Region に残る．

さらに Diff Array に割り当てるメモリを小さくすると AOBD でも Overflow が発生する．以上より，組み込みシステムで実行するプログラムに依存するが，Base Array に 32KB，Diff Array に 256KB 以上割り当てた時 AOBD による圧縮は効率的だと判断する．

6. まとめ

本研究ではランタイムとハードウェアの協調によって (1) オブジェクトのアラインメント行うことと，(2) 型情報をランタイムからハードウェアへ指示を行う圧縮効率の高いメモリ圧縮手法 Aligned Objects Based Deduplication (AOBD) を提案した．AOBD ではオブジェクト間の類似性を捉えて重複除去を行う．圧縮率の評価を AOBD と既存のメモリ圧縮手法 BCD に対して行なった．AOBD の間接指定方式，直接指定方式は BCD と比較してそれぞれ平均 22.3%，20.0%高い圧縮率を示した．また，組み込みシステム向けのサイズが小さいメモリを想定した評価も行った．現状 AOBD は Base Array，Diff Array に割り当てるメモリのサイズをそれぞれ 32 KB，256 KB ぐらいまで小さくしても効率的にメモリ圧縮を実施できることがわかった．

今後の課題として，パフォーマンスの測定がある．ラインの圧縮と展開には時間的なオーバーヘッドがあり，その影響は以下で述べるように小さいものと予想されるが，実際に提案手法の時間的なオーバーヘッドが小さいことを定量的に確かめる必要がある．提案と同様の仕組みをもつ BCD では，この圧縮と展開に必要な時間のオーバーヘッドは小さい [12]．これはラインの圧縮はラインの更新ごとではなく，キャッシュからメモリへとラインがフィルされる時に行われるためである．AOBD では BCD と同様にラインの圧縮はキャッシュからメモリへとラインがフィルされる時に行われる．同様の手順で圧縮や展開を行うため，その時間的なオーバーヘッドは BCD と同様に小さいものと予想される．

また，圧縮や展開に必要な時間とは別に，提案手法ではオブジェクトのアライメントによる空間的なオーバーヘッドがある．このアライメントにより生じた隙間はメモリ上では効率的に圧縮できるため大きな問題とはならないものの，展開された後のキャッシュ上では実際に容量を消費してしまう．このため，キャッシュ上においてもこの隙間を高速かつ効果的に圧縮する手法の考案や既存手法 [20], [21] の適用を今後検討する．

謝辞 本研究の一部は，JSPS 科研費 20H00578 および 20H04153 の助成を受けたものである．

参考文献

[1] Choukse, E., Erez, M. and R. Alameldeen, A.: Compresso: Pragmatic Main Memory Compression, *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2018).
[2] Ekman, M. and Stenstrom, P.: ARobust Main-memory Compression Scheme, *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2005).
[3] Pekhimenko, G., Seshadri, V., Kim, Y., Xin, H., Mutlu, O., B. Gibbons, P., A. Kozuch, M. and C. Mowry, T.: Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework, *Proceedings of the International Symposium on Microarchitecture (MICRO)* (2013).
[4] Tsai, P.-A. and Sanchez, D.: Compress Objects, Not Cache Lines: An Object-Based Compressed Memory Hierarchy, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2019).

[5] Alameldeen, A. and Wood, D.: Frequent Pattern Compression: A Significance-based Compression Scheme for L2 Caches, *Technical report, University of Wisconsin-Madison* (2004).
[6] Pekhimenko, G., Seshadri, V., Mutlu, O., B. Gibbons, P., Kozuch, M. A. and C. Mowry, T.: Base-delta-immediate Compression: Practical Data Compression for On-chip Caches, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2012).
[7] Kim, J., Sullivan, M., Choukse, E. and Erez, M.: Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures, *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2016).
[8] R Alameldeen, A. and A Wood, D.: Adaptive Cache Compression for Highperformance Processors, *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2004).
[9] Xie, Y. and H Loh, G.: Thread-aware Dynamic Shared Cache Compression in Multi-core Processors, *IEEE International Conference on Computer Design*.
[10] Young, V., J Nair, P. and K Qureshi, M.: Dice: Compressing dram caches for bandwidth and capacity, *Proceedings of the International Symposium on Computer Architecture (ISCA)* (2017).
[11] Zhang, Y., Yang, J. and Gupta, R.: Frequent Value Locality and Valuecentric Data Cache Design, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2000).
[12] Park, S., Kang, I., Moon, Y., Ho Ahn, J. and Suh, G. E.: BCD Deduplication: Effective Memory Compression using Partial Cache-Line Deduplication, *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).
[13] SPEC: SPEC CPU2017, `https://www.spec.org/cpu2017`.
[14] M Blackburn, S., Garner, R., Hoffmann, C., M Khang, A., S McKinley, K., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Z Guyer, S., Hirzel, M., Hosking, A., Jump, M., Maria, H., B.Moss, J. E., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D. and Wiedermann, B.: The DaCapo Benchmarks: Java Benchmarking Development and Analysis, *Proceedings of ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)* (2006).
[15] Singh, S. and Awasthi, M.: TMemory Centric Characterization and Analysis of SPEC CPU2017 Suite, *Proceedings of the International Conference on Performance Engineering* (2019).
[16] Poess, M., Smith, B., Kollar, L. and Larson, P.: TPC-DS, Taking Decision Support Benchmarking to the Next Level, *ACM SIGMOD International Conference on Management of Data* (2002).
[17] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Computer Languages*, No. 51, pp. 261–279 (2019).
[18] Marr, S., Daloze, B. and Mossenbock, H.: Cross-Language Compiler Benchmarking: are we fast yet?, *Proceedings of the International Symposium on Dynamic Languages (DLS)* (2016).
[19] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Janapa Reddi, V. and Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, *Programming Language Design and Implementation* (2015).
[20] Dusser, J., Piquet, T. and Seznec, A.: Zero-content Augmented Caches, *Proceedings of the International Conference on Supercomputing (ICS)* (2009).
[21] Goeman, B., Vandierendonck, H. and De Bosschere, K.: Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency, *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)* (2001).