冗長な符号拡張命令の除去手法

川人基弘[†] 小松秀昭[†] 中谷登志男[†]

近年，64 ビットアーキテクチャが使われ始め，32 ビットアーキテクチャから移行が行われようとしている．しかし，32 ビットアーキテクチャを前提として設計されたプログラムは，多くのデータサイズが 32 ビットとして扱われている．たとえば Java 言語は，最も多く使われる int 型を 32 ビットと規定している．このようなプログラムを 64 ビットアーキテクチャ上で実行させるには，多くの整数型命令に対して 32 ビットから 64 ビットへの符号拡張を行う命令が必要となり，プログラムの実行速度を下げる．我々は，符号拡張命令をできるだけ除去するために，最初にデータフロー解析を使った実装を行った．しかし，この手法では配列インデックスのアドレス計算に対する符号拡張命令は，ほとんどの場合除去できず，さらにプログラム上でより多く実行される場所から選択的に除去するということができなかった．そこで，我々は効果的に符号拡張命令の除去と移動を行う新しいアルゴリズムを提案する．本稿では，特に Java 言語に対する実装手法について述べているが，他の言語でも実装可能である．我々の手法は，より多く実行される場所から順に，変数の依存関係を利用して除去を行う．また，配列に関する言語仕様を符号拡張命令の除去に利用し，さらに符号拡張命令をより実行頻度の少ない場所に移動させることも行っている．我々はこのアルゴリズムを現在開発中の 64 ビットアーキテクチャ向けの IBM Java Just-in-Time（JIT）compiler に実装して評価を行った．その結果，符号拡張命令の実行回数を平均で 87%削減することができた．我々の知る限り，本稿は符号拡張命令の除去を行う最初のアルゴリズムである．

A Method for Elimination of Redundant Sign Extensions

Motohiro Kawahito,[†] Hideaki Komatsu[†] and Toshio Nakatani[†]

Recently, 32 bit architectures are being shifted to 64 bit architectures. However, a program designed for a 32-bit architecture still uses many 32-bit data. For example, Java specifies “int” (which is the most frequently used type) as 32 bits. If such programs are executed on 64-bit architecture, many 32-bit data are necessary to be sign-extended to 64-bit data for integer type instructions. It causes performance degradation. At first, we implemented sign extension elimination based on data-flow analysis to improve performance. However, this approach could hardly eliminate sign extensions for array indices. Moreover, it could not selectively eliminate sign extensions from frequently executed points. In this paper, we present an effective new algorithm for eliminating and moving sign extensions. The same approach should work for any language and architecture requiring sign extensions. Our approach utilizes the dependencies on a variable to selectively eliminate sign extensions in order of frequently executed points. Additionally, it utilizes a language specification of the array type to eliminate more sign extensions, and moves sign extensions to rarely executed points. We implemented the algorithm in the IBM Java Just-in-Time (JIT) compiler for 64-bit architectures, which is under development. Our experimental results show that the execution count of sign extensions can be eliminated by 87% on average. To the best of our knowledge, this is the first algorithm for sign extension elimination.

1. はじめに

コンパイラでプログラムをコンパイルする際に，プログラム中でアーキテクチャのレジスタサイズよりも小さいサイズとして定義されている値は，レジスタサイズに補正し，それから計算するのが一般的である．たとえば，64 ビットアーキテクチャではレジスタイメージは 64 ビットのため，プログラム中の 32，16，8 ビットの符号付きの値は 64 ビットに符号拡張して計算する必要がある．さらに，まだ世の中のシステムやアプリケーションの多くは 32 ビットアーキテクチャを前提として設計されている．たとえば Java の言語仕様[4)]では，最も多く使われる int 型を 32 ビットと規定している．そのようなプログラムは，ほとんどのデータサイズが 32 ビットのため，多くの整数型命令に対して符号拡張命令が必要となり，プログラムの実行速度を下げる．

† 日本アイ・ビー・エム株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

符号拡張命令の実装方法として，アーキテクチャによってはメモリからのロード・符号拡張を同時に行う命令が利用できる．たとえば PowerPC の Instruction Set Reference [6] を見ると，このような命令が利用できる．以下,「メモリからのロード・符号拡張を同時に行うことができるアーキテクチャ」を **PPC64 型アーキテクチャ**と呼ぶことにする．しかし，PPC64 型アーキテクチャでも，プログラム内で仮定されているデータサイズとアーキテクチャのレジスタサイズが異なる場合は，除去処理を行わなければほとんどの演算結果に対して符号拡張を生成しなければならない．メモリからのロード・符号拡張を同時に行う命令を持っていない 64 ビットアーキテクチャでは，より多くの符号拡張命令が生成されるため，特に符号拡張命令の除去が重要となる．たとえば，IA64 の Instruction Set Reference [7] を見ると，メモリからのロードはゼロ拡張されるが，符号拡張は明示的に行わなければならない．以下,「メモリからのロード・ゼロ拡張は行われるが，符号拡張は明示的に行わなければならないアーキテクチャ」を **IA64 型アーキテクチャ**と呼ぶことにする．

基本的に，符号拡張命令は「入力が必ず符号拡張されている場合」，または「符号拡張命令の出力結果を使用するすべての場所で，符号拡張しようとするデータサイズ以下で使われている場合」のどちらかの条件が満たされたときに除去できる．我々は，最初に符号拡張命令の除去を実装したときには，前者の性質を符号拡張命令の生成時に判断し，後者の性質を後方データフロー解析（backward data-flow analysis）で解析し，除去を行った．しかし，この方法では**図 1** の例に対して適用した場合，(1) と (5) の符号拡張しか除去できなかった．これは，次の 3 点が問題となっているためである．

1 つ目の問題は，ループの誘導変数が配列のインデックスとして使われている場合，ループ内のその変数に対する符号拡張命令（図 1 の例では (3)）は，多くの場合除去できない点である．この理由は配列アクセスのアドレス計算そのものはレジスタサイズで計算されるため，後方データフロー解析では除去できないからである．その結果，符号拡張命令がループ内に残ってしまう．一般的に配列アクセスはループ内にあることが多く，配列のインデックスに対する符号拡張命令を除去することはプログラムの高速化に大きく貢献する．

2 つ目の問題は，ループの外側・内側のどちらか片方しか符号拡張命令を除去できない状況下では，データフロー解析を使った除去では，選択的に有利なほうを除去することができない点である．前方データフロー

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
i = extend(i);            - (1)（除去可能）
do {
   i = i - 1;             - (2)
   i = extend(i);         - (3)
   j = a[i];              - (4)
   j = extend(j);         - (5)（除去可能）
   t += j;                - (6)
   t = extend(t);         - (7)
} while(i > start);
// t に対して符号拡張必要
d = (double)t;            - (8)
```

（extend( )は 32 ビットから 64 ビットへの符号拡張を意味する）

図 1　データフロー解析を使った符号拡張除去の問題点
Fig. 1　Drawbacks of the sign extension elimination using data-flow analysis.

解析を使った除去法では，実行方向に情報を伝えていくために，実行順で先のものが残る傾向にある．後方データフロー解析を使った除去法では，実行順で後のものが残る傾向にある．このため，消す順番を考慮しないとループの内側の符号拡張命令が残る場合がある．たとえば，図 1 内に i に関する符号拡張命令は (1) と (3) の 2 カ所ある．もし (1) か (3) どちらか片方しか除去できない場合は，ループ内の (3) を除去したほうが効果が大きい．しかし，データフロー解析を使った除去では，必ず (3) が除去されるという仮定をすることができない．

3 つ目の問題は，ループ内では符号拡張が必要ないのにもかかわらず，ループ外で符号拡張が必要なため，ループ内に符号拡張命令が残る点である．図 1 の例では，(7) はループ外の (8) が符号拡張を必要とするために除去することができない．

我々は，これらの問題を解く新しいアルゴリズムを提案する．我々の手法の特徴は次の 4 点である．

- 配列に関する言語仕様を符号拡張命令の除去に利用し，単純には除去できない配列アクセスのアドレス計算のための，配列インデックスに対する符号拡張命令を除去する．
- 多く実行されると予想される頻度順に，符号拡張命令の除去を行う．
- 上の 2 つの特徴を持つ除去法を可能にするために，UD（Use-Definition）/DU（Definition-Use）chain（使用–定義連鎖，定義–使用連鎖）[1] を利用し，符号拡張命令の除去を行う．
- 符号拡張命令の除去の前処理として，符号拡張命令の挿入を行う．これと除去の組合せにより，符号拡張命令をより実行頻度の少ない場所に移動させることができる．

我々はこのアルゴリズムを現在開発中の 64 ビットアーキテクチャ向けの IBM Java Just-in-Time（JIT）コンパイラに対して IA64 型アーキテクチャと PPC64 型アーキテクチャを想定して実装を行い，SPECjvm98 のベンチマークを用いて，それぞれのアーキテクチャに対してアルゴリズムの評価を行った．この結果，本アルゴリズムは SPECjvm98 で符号拡張命令の実行回数を平均で IA64 型アーキテクチャでは 88%，PPC64 型アーキテクチャでは 86%削減することができた．また，JIT のコンパイル時間は符号拡張命令の除去だけでは 0.21%，UD,DU-chain の作成時間を合わせても 2.5%というわずかな増加にとどまった．

本稿の以下の構成は次のようになっている．2 章では従来の研究について述べる．3 章では我々の手法の概略について述べる．4 章では我々のアルゴリズムについて述べる．5 章では我々の手法の評価を行う．

2. 従来の手法

符号拡張命令の実装方法として，アーキテクチャによってはメモリからのロード・符号拡張を同時に行う命令が利用できる．我々の IA32 や PowerPC 上の 32 ビットアーキテクチャ向けの JIT コンパイラ[8),14)]では，16，8 ビットの値についてメモリからのロード・符号拡張を同時に行う命令を単純に使っているだけで，符号拡張命令を消す処理は行っていなかった．64 ビットアーキテクチャで，Java などの 32 ビットを演算の中心とする言語をコンパイルしようとすると，ほとんどの整数型命令に対して符号拡張が必要となる．さらに，IA64 型アーキテクチャでは，メモリからのロード・符号拡張を同時に行う命令を持っていないため，特に符号拡張命令の除去が重要となる．我々の調べた限りでは，符号拡張命令を除去するコンパイラの最適化手法は過去に発表されていない．

3. 我々の手法の概略

本章では我々がどのような方法で，符号拡張命令を除去しているかを述べる．3.1 節では，我々が最初に実装した除去手法について簡単に述べる．3.2 節では，最初に実装した除去手法で見つけた問題点をどのように解決しているかを述べる．

3.1 データフロー解析を使った除去法

基本的に，符号拡張命令は「入力が必ず符号拡張されている場合」，または「符号拡張命令の出力結果を使用するすべての場所で，符号拡張しようとするデータサイズ以下で使われている場合」のどちらかの条件が満たされたときに除去できる．我々が，最初に符号拡張命令の除去を実装したときは，次のような流れで行った．

(1) レジスタサイズより小さいデータサイズ（32，16，8 ビット）の変数の値を定義する命令について，直後に符号拡張命令を生成する．ただし，変数の値を定義する命令の出力結果が，必ず符号拡張された状態になっていると分かるような命令に関しては，生成しない．このような命令の例としては（実装に依存するが）arraylength などがあげられる．これにより,「符号拡張命令の入力が必ず符号拡張されている場合」の解析を行っている．

(2) 後方データフロー解析を使って，符号拡張命令の出力結果が，使われるすべての場所で，符号拡張しようとするデータサイズ以下で使われているかどうかの情報を求め，該当するならば除去する．

3.1.1 データフロー解析を使った除去法の問題点

3.1 節で実装した手法で見つけた問題点を次にあげる．

- ループの誘導変数が配列のインデックスに使われている場合，ループ内のその変数に対する符号拡張命令は除去できない．この理由は，配列アクセスのアドレス計算そのものはレジスタサイズで計算されるためである．たとえば，要素のサイズが 4 バイトの a[i] という配列アクセスがあったとき，有効アドレスは $a + (i << 2)$ で計算される．このとき i がレジスタサイズに符号拡張されていなければ，間違ったアドレスを指してしまう．一般的に配列アクセスはループ内にあることが多く，配列のインデックスに対する符号拡張命令を除去できない場合にはパフォーマンスが悪くなる．
- 符号拡張命令の生成時のみ「入力が必ず符号拡張されている場合」を調べるのは不十分である．それは他の最適化の結果により，新たに符号拡張命令の除去機会が生まれる場合があるからである．たとえば，a = b & c というビット演算があり，b か c のどちらかに定数が伝播された場合，その定数値（たとえば 0xff）によっては，a を使用している命令の種類にかかわらず，a に対する符号拡張命令は除去できる．このように符号拡張命令の除去機会を増やす最適化の例として，constant propagation，copy propagation，common sub-expression elimination，scalar replacement[11)]，value range analysis[3),5)]などがあげられる．
- ループの外側・内側のどちらか片方しか符号拡張命令を除去できない状況下では，データフロー解析を使った除去法では，必ず有利なほうを除去できる

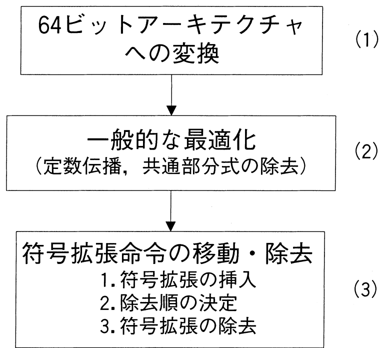

図 2 符号拡張命令の最適化の流れ図
Fig. 2 Flow diagram of sign extension optimization.

(a) 挿入処理前

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
i = extend(i);          - (1)
do {
   i = i - 1;           - (2)
   i = extend(i);       - (3)
   j = a[i];            - (4)
   j = extend(j);       - (5)
   t += j;              - (6)
   t = extend(t);       - (7)
} while(i > start);
// t に対して符号拡張必要
d = (double)t;          - (8)
```

(b) 挿入処理後

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
i = extend(i);          - (1)
do {
   i = i - 1;           - (2)
   i = extend(i);       - (3)
   j = a[i];            - (4)
   j = extend(j);       - (5)
   t += j;              - (6)
   t = extend(t);       - (7)
} while(i > start);
// t に対して符号拡張必要
t = extend(t);          - (9)
d = (double)t;          - (8)
```

図 3 符号拡張命令の挿入により効果があがる例
Fig. 3 An effective example by inserting sign extension.

とは限らない．後方データフロー解析を使った後向きの除去法では，実行順で後のものが残る傾向にあり，前方データフロー解析を使った前向きの除去法では実行順で先のものが残る傾向にある．このため，消す順番を考慮しないとループの内側の符号拡張命令が残る場合がある．

- 符号拡張命令がループ内にしか含まれていない場合，ループ内では符号拡張が必要ないが，ループ外で符号拡張が必要な場合がある．このようなときは，3.1 節の方法ではループ内に符号拡張命令が残ってしまう．

3.2 我々の最終的な除去法

本節では，前節で述べた問題点をどのように解決しているかを述べる．

3.2.1 符号拡張命令の最適化の流れ

図 2 は符号拡張命令の最適化の流れ図である．64 ビットアーキテクチャへの変換 (1) の処理は，バイトコードを 64 ビットアーキテクチャで最適化を行いやすくするために，中間コード上の変形を行う．このときに，3.1 節で述べた生成方法で符号拡張命令を生成する．一般的な最適化処理 (2) では，符号拡張命令も最適化対象とする．たとえば，定数伝播によって符号拡張命令の入力変数に定数が流れてきた場合には，符号拡張命令を通常のコピー命令に変更する．共通部分式の除去処理では，符号拡張命令についても最適化対象とする．我々は Lazy Code Motion [9),10)] のアルゴリズムを共通部分式の除去処理に適用している．この処理により，ループ不変の符号拡張命令を実行とは逆方向に移動させることにより，ループ外にも移動させることができる．

符号拡張命令の移動・除去処理 (3) は，3 つの処理に分かれる．最初に符号拡張命令を実行方向にどこまで遅らせることができるかを解析し，境界点に符号拡張命令を挿入する．本稿では，境界点とは実行方向へ移動可能な領域と移動不可能な領域の境界を意味する．我々の手法では，解析コストをできるだけ減らすために，ベーシックブロック内の厳密な境界点は求めず，ベーシックブロックの入口か出口に境界点を設定する．この処理と除去の組合せで，実行頻度が低い場所に符号拡張命令を移動させることができる．次に除去順を決定する．この理由は，実行頻度の高い場所から符号拡張命令を優先的に除去するためである．最後に決定された除去順に従って，UD,DU-chain を使った符号拡張命令の除去を行う．この 3 つの組合せにより，3.1.1 項で述べた問題点を解決することができる．

3.2.2 項では符号拡張命令の挿入処理，3.2.3 項では除去順の決定処理，3.2.4 項では我々の手法の最大の特徴である配列のインデックスに対する符号拡張命令の除去についての概略を述べる．

3.2.2 符号拡張命令の挿入処理

図 3 は，符号拡張命令の挿入により効果があがる例である．図 3 (8) はプログラム中の t という変数に対して，唯一符号拡張しなければならない命令である．しかし，このまま除去処理を行うと，(7) の符号拡張命令は必要と判断され残ってしまう．このため，ループ内では t に対して符号拡張は必要ないにもかかわらず，ループが回るたびに符号拡張する結果となってしまう．この挿入処理は前方データフロー解析を使って，符号拡張命令をどこまで遅らせられるかを求め，境界部分に符号拡張命令を挿入する．図 3 に対して適用すると，(b) のように (9) に符号拡張命令が挿入される．この後，3.2.4 節の除去処理を行うと**図 4** (b) のようになる．この結果，図 4 (a) と比較すると (7) の符号拡張命令はループの外 (9) に移動し，挿入処理を行わない場合と比べて性能が向上することが分かる．

(a) 挿入処理を行わずに除去した結果

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
do {
   i = i - 1;              - (2)
   j = a[i];               - (4)
   t += j;                 - (6)
   t = extend(t);          - (7)
} while(i > start);
// t に対して符号拡張必要
d = (double)t;             - (8)
```

(b) 挿入処理を行って除去した結果

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
do {
   i = i - 1;              - (2)
   j = a[i];               - (4)
   t += j;                 - (6)

} while(i > start);
// t に対して符号拡張必要
t = extend(t);             - (9)
d = (double)t;             - (8)
```

図 4 図 3 に対する除去結果
Fig. 4 An elimination result to Fig. 3.

3.2.3 除去順の決定処理

除去順を決定する理由は，実行頻度の高い場所から符号拡張命令を優先的に除去するためである．たとえば，配列アクセスのアドレス計算に対する符号拡張命令は配列インデックスにかかわる計算の 1 カ所で行えばよい場合が多い．そのため，ある場所で除去した符号拡張命令の影響で，他の符号拡張命令が除去できなくなる場合がある．そのため，配列インデックスにかかわる定義に関する符号拡張命令がループの外側・内側にあるとき，順番によってはループの外の符号拡張命令が除去され，その結果ループ内の符号拡張命令が除去できない場合がある．たとえば IA64 型アーキテクチャでは，図 3 (b) に対して，(1) → (3) の順番で除去を行うと (2) の式によっては，(1) が除去される影響で (3) が除去されない場合がある．

前項で述べた挿入処理は除去できる候補を複数作り出すための処理である．そのため，除去順により効果が大きく異なる．いい換えると，挿入処理は本項の除去順の決定処理を仮定しているともいえる．たとえば，図 3 (b) に対して，(9) → (7) の順番で除去を行うと (9) が除去される影響で (7) が除去されなくなってしまう．

そのため，実行頻度が高いと予想される部分から順番に除去を行うことが必要となる．実行頻度を決める方法として，次の 3 つが考えられる．これらの情報を組み合わせてプログラム内の各符号拡張命令の実行頻度の見積りを決定し，除去順序を決める．実行頻度が同じときには実行順で後から除去順を設定する．

- ループの中：ループのネストが深くなればなるほど，多く実行される可能性が大きい．
- 条件によって実行されるベーシックブロック：一般的にプログラム内の条件分岐が行われるごとに，分岐先のブロックが実行される確率は低くなり，コントロールフローの合流が行われるごとにそれ以降のブロックの実行確率は高まる．各分岐確立をもとに，各ブロックの実行される頻度を予測することができる．
- 実行時のプロファイル情報の利用：上の 2 つはプログラムを静的に解析しているが，実行時のプロファイル情報[2),12)]を使うことができれば，頻繁に実行されるベーシックブロックをより正確に見つけることができる．また，ほとんど実行されないブロックという情報も重要で，このようなブロックがループ内に存在する場合もある[15)]．

3.2.4 符号拡張命令の除去処理

3.1 節の除去法の最大の問題点は，配列のアドレス計算のための配列インデックスに対する符号拡張命令が除去できない点である．しかし，言語仕様により負のインデックスを使って配列をアクセスしてはいけないと定められている言語では，負の配列インデックスを使ってメモリアクセスが行われないと仮定することで除去を行うことができる場合がある．

たとえば，Java 言語では負のインデックスを使って配列をアクセスすると，配列境界チェックによって例外をおこす．このため，メモリアクセスの時点では配列インデックスに必ず負の値がこないと仮定できる．なお，この配列境界チェックを実装するために符号拡張を必要とする可能性があるが，レジスタの下 32 ビットどうしを比較する命令を持っている 64 ビットアーキテクチャでは，符号拡張することなく配列境界チェックを実現することができる．PPC64 や IA64 などはこのような命令を持っているため[6),7)]，この仮定はそれほど特別なものではないと思われる．

言語仕様により，負の配列インデックスを使ってメモリアクセスが行われないと仮定することで次の定理 1〜4 が導き出せる．なお，定理内の前提条件で値のとりうる範囲を制限しているものがあるが，これは値が定数の場合はもちろん，変数の場合であっても値の範囲分析（value range analysis）[3),5)]を行うことにより，条件を満たすかどうかをコンパイル時に検出することができる．

定理 1 変数 i について次の 2 つの条件がともに満たされるならば，i を配列インデックスとしてアクセスする配列のアドレス計算には，i に対して 32 ビットから 64 ビットの符号拡張は必要ない．

- i の上 32 ビットが 0 で初期化されている．
- 言語仕様により，配列インデックス i の下 32 ビットについて，$0 \le$ i の下 32 ビット $\le$ 0x7fffffff が成り立つ．

証明 1，2 番目の条件により，$0 \le i \le$ 0x7fffffff となり，i は符号付 32 ビット表現で負の値にはならないため，すでに符号拡張された状態になっている．そのため，符号拡張は必要ない．

□

定理 2 変数 i，j について次の 3 つの条件がすべて満たされるならば，i+j を配列インデックスとしてアクセスする配列のアドレス計算には，i+j に対して 32 ビットから 64 ビットの符号拡張は必要ない．

- i，j は両方ともすでに符号拡張されている．
- i，j どちらかの値が 0 以上 0x7fffffff 以下の範囲内にある．
- 言語仕様により，配列インデックス i+j の下 32 ビットについて，$0 \le$ (i+j) の下 32 ビット $\le$ 0x7fffffff が成り立つ．

証明 i+j について i と j は交換可能なため，i が 2 番目の条件に合うときのみ証明できれば，j が 2 番目の条件に合うときも同様に証明できる．

$0 \le i \le$ 0x7fffffff のとき

- **$0 \le j \le$ 0x7fffffff のとき** i+j の上 32 ビットについて 0 が保証されているので，定理 1 より証明可能．
- 0xffffffff80000000 $\le$ **j** $\le$ 0xffffffffffffffff **のとき** 0xffffffff80000000 $\le$ i+j $\le$ 0xffffffffffffffff または $0 \le$ i+j $\le$ 0x7ffffffe が成り立つ．3 番目の条件により $0 \le$ i+j $\le$ 0x7ffffffe となり，符号付 32 ビット表現で i+j は負の値にはならないため，すでに符号拡張された状態になっている．そのため，符号拡張は必要ない．

□

また，配列の大きさの上限が，ある値に制限されている（または制限できる）場合や配列の大きさがコンパイル時に分かる場合は，定理 2 を拡張した次の定理 3 が導き出せる．

定理 3 次の 4 つの条件がすべて満たされるならば，i+j を配列インデックスとしてアクセスする配列のアドレス計算には，i+j に対して 32 ビットから 64 ビットの符号拡張は必要ない．

- 配列の大きさの上限が maxlen で与えられ，$0 \le$ maxlen $\le$ 0x7fffffff が成り立つ．
- i，j は両方ともすでに符号拡張されている．
- i，j どちらかの値が (maxlen−1)−0x7fffffff 以上 0x7fffffff 以下の範囲内にある．
- 言語仕様により，配列インデックス i+j の下 32 ビットについて，$0 \le$ (i+j) の下 32 ビット $<$ maxlen $\le$ 0x7fffffff が成り立つ．

証明 i+j について i と j は交換可能なため，i が 3 番目の条件に合うときのみ証明できれば，j が 3 番目の条件に合うときも同様に証明できる．

$0 \le i \le$ 0x7fffffff のとき 定理 2 より，符号拡張不要なことは証明可能．

(maxlen−1)−0x7fffffff $\le$ i $\le$ 0xffffffffffffffff のとき

- **$0 \le j \le$ 0x7fffffff のとき** 定理 2 より，符号拡張不要なことは証明可能．
- 0xffffffff80000000 $\le$ **j** $\le$ 0xffffffffffffffff **のとき** 0xffffffff00000000+maxlen $\le$ i+j $\le$ 0xfffffffffffffffe が成り立ち，maxlen $\le$ (i+j) の下 32 ビット $\le$ 0xfffffffe となる．ところが，この場合は 4 番目の条件を満たすことはないため条件外となる．

□

なお，定理 2，3 は i−k のような引き算の場合についても，j を (−k) と置き換えることで，定理の前提条件を満たす k の範囲を求めることにより，適用することができる．

定理 3 を利用すると，片方が負の値の足し算式であっても，符号拡張命令の除去を行うことができる．たとえば，Java では，言語仕様上の配列の大きさの最大値は 0x7fffffff となるので，i または j のどちらかの値が −1〜0x7fffffff の範囲内にあり，i，j の両方が符号拡張されていれば，配列インデックス i+j でアクセスされる配列のアドレス計算には，i+j に対して符号拡張は不要となる．定理 2 の前提条件と比べると，−1 を許すか許さないかの差だけのように思えるが，増分が −1 の count down ループなどはよく使われることを考えると，実用上この差は大きい．たとえば，図 3 に対して適用すると，定理 3 より (3) は除去することができる．

また，別の例をあげると，言語処理システムの実装上の制限などにより，配列の大きさの最大値がたとえば 0x3fffffff に制限できる場合は，i または j のどちらかの値が 0xffffffffbfffffff(−1073741825)〜0x7fffffff の範囲内にあり，i，j の両方が符号拡張されていれば，配列インデックス i+j でアクセスされる配列のアドレス計算には，i+j に対して符号拡張は不要となる．実際には，これほど絶対値の大きい負の値は配列アクセスの際にはほとんど使われることはない．実用上は −65536 程度，すなわち配列の大きさの最大値が 0x7fff0001 以下に制限できれば十分であると思われる．

図 5 に配列の大きさによっては符号拡張命令を除去できる例を示す．図 5 は図 3 (a) (2) を i = i − **2**; と変更した例である．この例で，mem が符号付 32 ビッ

```
int j; // j は 32bit 変数
int t = 0; // t は 32bit 変数
int i = mem; // i は 32bit 変数
i = extend(i);          - (1)
// ここで i = 0xffffffff80000000 と仮定する．
do {
   i = i - 2;           - (2)
   // i == 0xffffffff7ffffffe となる．
   i = extend(i);       - (3)
   // i == 0x7ffffffe となり配列の大きさによってはアクセス可能
   j = a[i];            - (4)
   j = extend(j);       - (5)
   t += j;              - (6)
   t = extend(t);       - (7)
} while(i > start);
// t に対して符号拡張必要
d = (double)t;          - (8)
```

図 5 配列の大きさによっては符号拡張命令を除去できる例
Fig. 5 A removable sign extension depending on array size.

トの最小値 0x80000000 だったと仮定すると，最初の配列アクセス時には i の下 32 ビットは 0x7ffffffe となり，配列 a の大きさが 0x7fffffff のときには a[i] はアクセス可能である．この場合には i の上 32 ビットを補正しなければならないために，(3) の符号拡張命令は除去できない．配列 a の大きさが 0x7ffffffe 以下の場合は a [i] は不正なアクセスとなり，プログラム上の間違いとなる．これは「i の下 32 ビット (0x7ffffffe) < 配列 a の大きさ」が成り立たないためである．そのため，配列 a の大きさが 0x7ffffffe 以下ということがコンパイル時に判断できる場合は，a [i] のアドレス計算のための符号拡張 (3) は不要となる．

また，次の定理 4 を使えば，引き算式についてさらに拡張が行える．IA64 型アーキテクチャはメモリロードの際にゼロ拡張が行われるため，定理 4 を使うことができる機会が多い．図 3 に対して定理 4 を適用すると IA64 型アーキテクチャの場合でも (1) を除去することができる．

定理 4 次の 3 つの条件がすべて満たされるならば，i−j を配列インデックスとしてアクセスする配列のアドレス計算には，i−j に対して 32 ビットから 64 ビットの符号拡張は必要ない．

- i の上 32 ビットが 0 で初期化されている．
- j の値が 0 以上 0x7fffffff 以下の範囲内にある．
- 言語仕様により，配列インデックス i−j の下 32 ビットについて，0 ≤ (i−j) の下 32 ビット ≤ 0x7fffffff が成り立つ．

証明 i のとりうる範囲は，0 ≤ i ≤ 0xffffffff. i−j のとりうる範囲は 0xffffffff80000001 ≤ i−j ≤ 0xffffffffffffffff または 0 ≤ i−j ≤ 0xffffffff である．3 番目の条件から，0 ≤ i−j ≤ 0x7fffffff が成り立ち，符号付 32 ビット表現で i−j は負の値にはならないため，すでに符号拡張された状態になっている．そのため，符号拡張は必要ない．

□

これらの定理を利用して，配列のアドレス計算に対して符号拡張が必要かどうかの解析および除去を行う．

4. アルゴリズム

本章では我々のアルゴリズムについて説明する．4.1 節では，符号拡張命令の挿入処理について述べる．4.2 節では，符号拡張命令の除去処理について述べる．

4.1 符号拡張命令の挿入処理

この処理は，コード内に存在する符号拡張命令をできるだけ実行方向に遅らせ，境界部分に符号拡張命令を挿入する．この処理と 4.2 節で述べる符号拡張命令の除去処理を組み合わせることにより，部分的冗長性（partial redundancy [11]）を持つ符号拡張命令の除去やループ内の符号拡張命令をループ外に移動させることができる．

In(n)，Out(n) はメソッド内に含まれる符号拡張命令を実行方向に移動させた場合に，ベーシックブロック n の先頭，最後へ移動可能な符号拡張命令の集合である．これらの集合は次の前方データフロー解析を使うことによって求められる．

$$\mathrm{In}(n) = \bigcap_{m \in Pred(n)} \mathrm{Out}(m)$$

$$\mathrm{Out}(n) = (\mathrm{In}(n) - \mathrm{Kill}(n)) \cup \mathrm{Gen}(n)$$

ここで Gen と Kill は次のような集合を意味する．

Gen(n)：ベーシックブロック n 内の符号拡張命令について，ベーシックブロック n の最後に移動できる符号拡張命令の集合．

Kill(n)：ベーシックブロック n を実行方向に超えることができない符号拡張命令の集合．

Gen(n) および Kill(n) は次のアルゴリズムにより求める．

```
Kill(n) = Gen(n) = φ
for(n 内の初めから終わりまでの命令 I について){
  for(v ∈ I で使われる変数){
    if(I は変数 v の上 32 ビットに影響を受ける){
      E = 変数 v に対する符号拡張命令
      Kill(n) = Kill(n) ∪ E
      Gen(n) = Gen(n) - E
    }
  }
  if (I には出力変数 v がある){
    E = 変数 v に対する符号拡張命令
```

```
EliminateOneExtend(EXT) {
  全命令の USE,DEF,ARRAY の解析済みフラグの初期化
  required = FALSE
  /* DU-chain を使う */
  for (I ∈ EXT の出力変数を使用する全ての命令){
    required = AnalyzeUSE(EXT, I, TRUE)
    if (required) break;
  }
  if (required){
    /* UD-chain を使う */
    for (I ∈ EXT の入力変数を定義する全ての命令){
      required = AnalyzeDEF(I)
      if (required) break;
    }
  }
  if (!required) EXT を除去
}
```

図 6　1 つの符号拡張命令の除去アルゴリズム
Fig. 6　An elimination algorithm of one sign extension.

```
    Kill(n) = Kill(n) ∪ E
    if (I は符号拡張命令)
      Gen(n) = Gen(n) ∪ E
    else
      Gen(n) = Gen(n) - E
  }
}
```

次に，ベーシックブロック n の先頭，最後に挿入する符号拡張命令の集合 InIns(n)，OutIns(n) を次の式を使って求める．

$$\mathrm{InIns}(n) = \mathrm{In}(n) \cap \mathrm{Kill}(n)$$
$$\mathrm{OutIns}(n) = \mathrm{Out}(n) - \mathrm{Gen}(n) - \bigcap_{m \in Succ(n)} \mathrm{In}(m)$$

求まった InIns(n) に含まれる符号拡張命令をベーシックブロック n の先頭に挿入し，OutIns(n) に含まれる符号拡張命令をベーシックブロック n の最後に挿入する．

4.2　符号拡張命令の除去処理

ここでの処理は，図 2 (5) で決めた順番に，1 つずつ UD,DU-chain を使って符号拡張命令の解析および除去を行う．**図 6** の EliminateOneExtend は 1 つの符号拡張命令 EXT の除去処理のアルゴリズムである．なお，以下のアルゴリズム内では，1 命令ごとに USE, DEF，配列のアドレス計算をすでに解析したかどうかの 3 つの情報を持っていることを仮定している．これを「解析済みフラグ」と呼ぶことにする．

基本的に，符号拡張命令は「入力が必ず符号拡張されている場合」，または「符号拡張命令の出力結果を使用するすべての場所で，符号拡張しようとするデータサイズ以下で使われている場合」のどちらかの条件が満たされたときに除去できる．**図 7** の AnalyzeDEF は前者の条件を解析する処理である．図 7 の中で,「I の出力結果が必ず符号拡張された状態であると分かる命令」の代表的な例としては，AND をとる命令で片方のオペランドが負ではないことが保証されている場合などがあげられる．また,「I の入力変数が符号拡張されていれば I の出力結果も符号拡張されていると判断できる命令」の代表的な例としてはコピー命令があげられる．この処理で，3.1.1 項で述べた 2 番目の問題点を解決できる．

```
/* FALSE を返せば I について符号拡張不要
   TRUE を返せば I について符号拡張必要 */
AnalyzeDEF(I) {
  if (I は DEF について解析済み) return FALSE;
  I を DEF について解析済みとして記録
  switch(I){
  case I の出力結果が必ず符号拡張された状態である
       とわかる命令：
    return FALSE;

  case I の入力変数が符号拡張されていれば I の出力結果
       も符号拡張されていると判断できる命令：
    /* UD-chain を使う */
    for (J ∈ I の入力変数を定義する全ての命令){
      if ( AnalyzeDEF(J) ) return TRUE;
    }
    return FALSE;
  }
  return TRUE;
}
```

図 7　符号拡張命令の入力変数に対する解析
Fig. 7　Analysis for an input variable of a sign extension.

図 8 の AnalyzeUSE は符号拡張命令を除去できる条件の後者の条件を解析する処理である．図 8 の中で,「I は使用される変数の上 32 ビットに影響を受けない命令」の代表的な例としては，32 ビット以下のサイズのメモリへの書き込み命令があげられる．また,「I の出力結果について符号拡張必要なしと判断できれば I の入力変数も必要無しと判断できる命令」の代表的な例としては，コピー命令や足し算・引き算などがあげられる．

図 9 の AnalyzeARRAY は配列のアドレス計算について解析する処理である．この処理は，EXT の入力変数を定義するすべての命令の出力変数について，定理 1，2，3，4 のどれかが成り立てば，配列のアドレス計算命令 AOP について符号拡張命令は不要と判断している．この処理で，3.1.1 項で述べた 1 番目の問題点を解決できる．

5. 実 験 結 果

我々は SPECjvm98 [13] のベンチマークを測定し，本

```
/* FALSEを返せばIについて符号拡張不要
   TRUEを返せばIについて符号拡張必要 */
AnalyzeUSE(EXT, I, DO_ARRAY) {
  if (IはUSEについて解析済み) return FALSE;
  IをUSEについて解析済みとして記録
  switch(I){
  case Iは使用される変数の上32ビットに
       影響を受けない命令：
    return FALSE;

  case 配列のアドレス計算：
    if (DO_ARRAY){
      return AnalyzeARRAY(EXT, I);
    }
    break;

  case Iの出力結果について「符号拡張必要なし」と判断
      できればIの入力変数も必要無しと判断できる命令：
    if (Iを経由すると配列のアドレス計算に対する
        解析が不可能){
      DO_ARRAY = FALSE;
    }
    /* DU-chainを使う */
    for (J ∈ Iの出力変数を使用する全ての命令){
      if ( AnalyzeUSE(EXT, J, DO_ARRAY) ){
        return TRUE;
      }
    }
    return FALSE;
  }
  return TRUE;
}
```

図 8　符号拡張命令の出力変数に対する解析

Fig. 8　Analysis for an output variable of a sign extension.

アルゴリズムの評価を行った．すべての測定結果を同じ環境で行うために，SPECjvm98の測定方法はコマンドラインから個々のベンチマークを問題の大きさを100にして実行させた．すべての実験結果は，現在開発中の64ビットアーキテクチャ向けのIBM Java Just-in-Time（JIT）compilerに，PPC型アーキテクチャとIA64型アーキテクチャを想定して実装および評価を行った．それぞれのアーキテクチャが持つ符号拡張に関連した特長は次のとおりである[6),7)]．

- 共通の特徴
  - レジスタの下32ビットどうしを比較する命令を持っている．そのため，符号拡張することなく配列境界チェックを行うことができる．
- IA64型アーキテクチャ
  - メモリからのロード・符号拡張を同時に行う命令を持っていない．そのため，符号拡張の最適化の効果が大きい．
  - メモリからのロード・ゼロ拡張を同時に行う命令を持っている．そのため，3.2.4項で述べた定理1，定理4を使うことができる機会が多い．
- PPC64型アーキテクチャ
  - メモリからのロード・符号拡張を同時に行う命令を持っている．そのため，符号拡張の最適化の効果が小さい．

```
/* FALSEを返せばAOPについて符号拡張不要
   TRUEを返せばAOPについて符号拡張必要 */
AnalyzeARRAY(EXT, AOP) {
  if (配列のアドレス計算命令AOPの出力変数を
      使用する命令が全て配列アクセスの時){
    for (D ∈ EXTの入力変数を定義する命令){
      if (DはARRAYについてまだ解析していない){
        DをARRAYについて解析済みとして記録
        required = TRUE;
        switch(D){
        case Dの出力変数の上32ビットは必ず
             0で初期化されている: /* 定理1 */
          required = FALSE;
          break;

        case 引き算命令:
        case 足し算命令:
          if (Dのオペランドについて定理2,3,4
              のどれかの前提条件が成り立つ){
            required = FALSE;
          }
          break;

        case コピー命令:
          /* UD-chainを使う */
          for (J ∈ Dの入力変数を定義する
                    全ての命令){
            if ( AnalyzeARRAY(J, AOP) )
              return TRUE;
          }
          required = FALSE;
          break;
        }
        if (required) return TRUE;
      }
    }
    return FALSE;
  }
  return TRUE;
}
```

図 9　配列のアドレス計算に対する解析

Fig. 9　Analysis for an address computation of an array.

5.1　本アルゴリズムの効果

表 1，**表 2** にIA64型，PPC64型アーキテクチャを想定した場合の，SPECjvm98内の7つのベンチマークに対する符号拡張命令の実行回数を示す．これは，コンパイルされたコード内に符号拡張命令を生成する際に，回数を数えるコードも同時に生成させて得たものである．表1，表2内の**bold体**は効果があった項目，*italic*体は悪い結果となった項目を意味する．除去順序の決定を行っていない版はすべて深さ優先探

表 1 IA64 型アーキテクチャの符号拡張命令の実行回数
（bold 体：効果があった項目，italic 体：悪い結果となった項目）
Table 1 Dynamic counts of sign extension for IA64 type architecture.

	mtrt	jess	compress	db	mpegaudio	jack	javac	average
no opt	19250992	179631874	1868243993	426564035	798778870	99985664	262613889	
	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)	(100.00%)
bwd flow	**9000995**	**70530009**	**612671881**	**239128908**	**337814900**	**47605052**	**120233896**	
	(46.76%)	**(39.26%)**	**(32.79%)**	**(56.06%)**	**(42.29%)**	**(47.61%)**	**(45.78%)**	**(44.37%)**
basic ud/du	**7969878**	**63483505**	**573217891**	239120133	**251217797**	**41580830**	**108608624**	
	(41.40%)	**(35.34%)**	**(30.68%)**	(56.06%)	**(31.45%)**	**(41.59%)**	**(41.36%)**	**(39.70%)**
insert	7969516	63486640	573217977	239120301	251217875	*41775843*	**108408170**	
	(41.40%)	(35.34%)	(30.68%)	(56.06%)	(31.45%)	*(41.78%)*	**(41.28%)**	*(39.71%)*
sort	7969944	63483515	573217883	239120142	251217797	41580830	**104748269**	
	(41.40%)	(35.34%)	(30.68%)	(56.06%)	(31.45%)	(41.59%)	**(39.89%)**	**(39.49%)**
insert, sort	7969003	**61583292**	**572322509**	239118911	251124104	**41459594**	**104358917**	
	(41.40%)	**(34.28%)**	**(30.63%)**	(56.06%)	(31.44%)	**(41.47%)**	**(39.74%)**	**(39.29%)**
array	**2143804**	**17874189**	**253160692**	**39160707**	**69832670**	**25187404**	**52252046**	
	(11.14%)	**(9.95%)**	**(13.55%)**	**(9.18%)**	**(8.74%)**	**(25.19%)**	**(19.90%)**	**(13.95%)**
array, insert	**2117381**	**17603047**	**243279665**	**15041791**	69479431	**23603592**	**51859996**	
	(11.00%)	**(9.80%)**	**(13.02%)**	**(3.53%)**	(8.70%)	**(23.61%)**	**(19.75%)**	**(12.77%)**
array, sort	**2089200**	**17771713**	**243277265**	**27399025**	**68204510**	**22887294**	**44567388**	
	(10.85%)	**(9.89%)**	**(13.02%)**	**(6.42%)**	**(8.54%)**	**(22.89%)**	**(16.97%)**	**(12.66%)**
all	2087591	**15689329**	**242381415**	**14001494**	**67170924**	**22671298**	**42136962**	
	(10.84%)	**(8.73%)**	**(12.97%)**	**(3.28%)**	**(8.41%)**	**(22.67%)**	**(16.05%)**	**(11.85%)**

no opt: 符号拡張命令の最適化を止めた版．
bwd flow: 我々の最初の手法．後方データフロー解析を使って除去を行う．
basic ud/du: insert，sort，array の 3 つの最適化を止めた版．変数の定義・使用を解析することにより除去を行う．
insert: basic ud/du に加えて，符号拡張命令の挿入を行った版．
sort: basic ud/du に加えて，除去順序の決定を行った版．
insert，sort: insert，sort の組合せ．
array: basic ud/du に加えて，配列のアドレス計算に関する除去処理を行った版．
array，insert: array，insert の組合せ．
array，sort: array，sort の組合せ．
all: basic ud/du に加えて，insert，sort，array の 3 つの最適化を行った版．

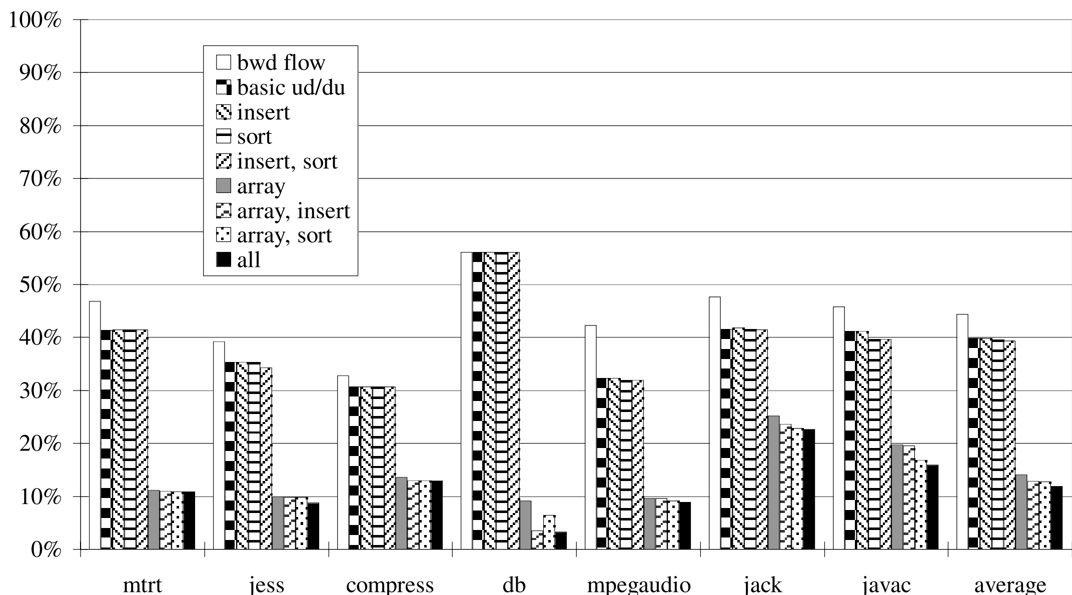

図 10 IA64 型アーキテクチャの符号拡張命令の実行回数（no opt を 100%とする）
Fig. 10 Dynamic counts of sign extension for IA64 type architecture (no opt=100%).

表 2 PPC64 型アーキテクチャの符号拡張命令の実行回数
（bold 体：効果があった項目，italic 体：悪い結果となった項目）
Table 2 Dynamic counts of sign extension for PPC64 type architecture.

	mtrt	jess	compress	db	mpegaudio	jack	javac	average
no opt	10184366 (100.00%)	61680302 (100.00%)	1129361408 (100.00%)	298096782 (100.00%)	596966915 (100.00%)	51039079 (100.00%)	164627134 (100.00%)	(100.00%)
bwd flow	**6999825** **(68.73%)**	**43525896** **(70.57%)**	**374167659** **(33.13%)**	**184965495** **(62.05%)**	**281212081** **(47.11%)**	**34455413** **(67.51%)**	**82930355** **(50.37%)**	**(57.07%)**
basic ud/du	**6958445** **(68.32%)**	**36463524** **(59.12%)**	**334712555** **(29.64%)**	184954438 (62.05%)	**194891006** **(32.65%)**	**29521432** **(57.84%)**	**73134631** **(44.42%)**	**(50.58%)**
insert	6958369 (68.32%)	36466662 (59.12%)	334712641 (29.64%)	184954597 (62.05%)	194891092 (32.65%)	*29840358* *(58.47%)*	73088437 (44.40%)	*(50.66%)*
sort	6957530 (68.32%)	36463533 (59.12%)	334712562 (29.64%)	184954438 (62.05%)	194890992 (32.65%)	29521432 (57.84%)	**73091713** **(44.40%)**	**(50.57%)**
insert, sort	6957037 (68.31%)	**34564422** **(56.04%)**	**333817355** **(29.56%)**	184954278 (62.05%)	194882706 (32.65%)	**29501422** **(57.80%)**	73074060 (44.39%)	**(50.11%)**
array	**1733140** **(17.02%)**	**9732235** **(15.78%)**	**232511044** **(20.59%)**	**38957089** **(13.07%)**	**41620474** **(6.97%)**	**14068685** **(27.56%)**	**27435057** **(16.66%)**	**(16.81%)**
array, insert	**1709668** **(16.79%)**	**9468276** **(15.35%)**	**222632017** **(19.71%)**	**14849434** **(4.98%)**	41419770 (6.94%)	*14252864* *(27.93%)*	**27033074** **(16.42%)**	**(15.45%)**
array, sort	**1707518** **(16.77%)**	**9635530** **(15.62%)**	**222629317** **(19.71%)**	**27206733** **(9.13%)**	**39993422** **(6.70%)**	**14024407** **(27.48%)**	**23603420** **(14.34%)**	**(15.68%)**
all	1706261 (16.75%)	**7557208** **(12.25%)**	**221733992** **(19.63%)**	**13810486** **(4.63%)**	**39196650** **(6.57%)**	**13910407** **(27.25%)**	**23369001** **(14.20%)**	**(14.47%)**

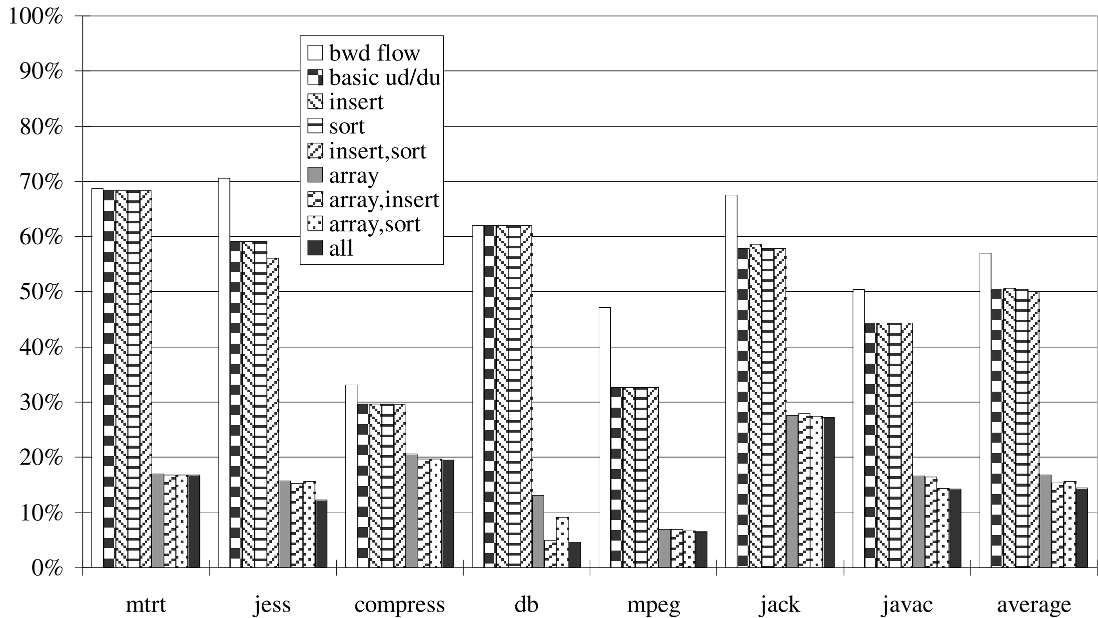

図 11 PPC64 型アーキテクチャの符号拡張命令の実行回数（no opt を 100%とする）
Fig. 11 Dynamic counts of sign extension for PPC64 type architecture (no opt=100%).

索（Depth-first search）の逆順に除去を行った．これは，後方データフロー解析を使って除去を行った場合を想定して，この順番にした．**図 10**，**図 11** はそれぞれ表 1，表 2 に対応し，最適化を行っていない場合を 100%としたときの削除効果のグラフを示したものである．

bwd flow と basic ud/du の差は 3.1.1 項の 2 番目の問題点の解決，すなわち他の最適化が除去機会を増やした効果と考えられる．配列のアドレス計算に関する除去効果は，すべてのベンチマークで大きく表れた．

表 3 IA64 型アーキテクチャの符号拡張命令の除去個数，および 100MHz のプロセッサ上で期待される速度向上

Table 3 Dynamic counts of eliminated sign extension and estimated performance improvements for IA64 type architecture.

		mtrt	jess	compress	db	mpegaudio	jack	javac
bwd flow	除去数	10249997	109101865	1255572112	187435127	460963970	52380612	142379993
	予想秒数	0.102	1.091	12.556	1.874	4.610	0.524	1.424
all	除去数	17163401	163942545	1625862578	412562541	731607946	77314366	220476927
	予想秒数	0.172	1.639	16.259	4.126	7.316	0.773	2.205

表 4 PPC64 型アーキテクチャの符号拡張命令の除去個数，および 100MHz のプロセッサ上で期待される速度向上

Table 4 Dynamic counts of eliminated sign extension and estimated performance improvements for PPC64 type architecture.

		mtrt	jess	compress	db	mpegaudio	jack	javac
bwd flow	除去数	3184541	18154406	755193749	113131287	315754834	16583666	81696779
	予想秒数	0.032	0.182	7.552	1.131	3.158	0.166	0.817
all	除去数	8478105	54123094	907627416	284286296	557770265	37128672	141258133
	予想秒数	0.085	0.541	9.076	2.843	5.578	0.371	1.413

除去順の決定と符号拡張命令の挿入に関しては，次のような興味深い結果が分かった．

- 「除去順の決定」を行う場合は,「配列のアドレス計算に関する除去」または「符号拡張命令の挿入」と組み合わせることにより，除去効果をより高めることができる．
- 「符号拡張命令の挿入」を行う場合には,「除去順の決定処理」との組合せは必須である．

前者に関しては,「配列のアドレス計算に関する除去」または「符号拡張命令の挿入」を適用した場合には，どこか 1 カ所で符号拡張を行っていればよいという状況が頻繁に発生する．そのため，選択的に有利な順に除去を行う効果が大きく表れると考えられる．実験結果を見ても,「除去順の決定処理」単独で効果が出ているのは javac だけである．さらにほとんどのベンチマークプログラムで，array，insert，sort の 3 つの処理すべてがそろわないと除去できない多くの符号拡張命令が観測された．

後者に関しては,「除去順の決定処理」を行わずに「符号拡張命令の挿入」だけを行うと，不利なほうが残ってしまう可能性が増えるためと考えられる．実験結果を見ても，jack については除去順の決定処理と組み合わせずに挿入処理を行うと逆に悪い結果となり，insert と sort との組合せが必須であることが分かる．

表 3，**表 4** に表 1，表 2 から計算した IA64 型，PPC64 型アーキテクチャを想定した場合に除去できる符号拡張命令の数，および 1 命令 1 サイクルで実行されたと仮定したときの 100 MHz のプロセッサ上で期待される速度向上（秒）を示す．bwd flow は我々が最初に実装した後方データフロー解析を使った除去法，all は本稿で述べたすべての最適化を適用した版の結果を示している．予想どおり，IA64 型アーキテクチャのほうが全体的に符号拡張の除去の効果が大きい．しかし，PPC64 型アーキテクチャでも符号拡張の除去の効果は無視できるものではないことが分かる．特に compress，db，mpegaudio に関しては大きなパフォーマンスの向上が期待できる．

5.2 JIT のコンパイル時間

本節では，符号拡張命令の最適化がどの程度 JIT のコンパイル時間に影響を与えているかについて述べる．我々は，開発マシン上で利用可能なトレースツールを用いて JIT のコンパイル時間の内訳を測定した．なお，非常に小さい割合を比較する都合上，より処理量が多い IA64 型アーキテクチャを想定した場合のコンパイル時間を測定した．**表 5**，**図 12** に IA64 型アーキテクチャを想定した場合の JIT のコンパイル時間の内訳を示す．本稿で述べたすべての符号拡張命令の最適化と UD,DU-chain の作成時間を合わせても平均 2.5%程度の増加にとどまっている．

さらに，我々の実装では元々UD,DU-chain を別の最適化のために使っていることから，符号拡張命令の最適化を行わなくても UD,DU-chain の作成は必要である．なお，表 5 の UD,DU-chain の作成時間には，4.1 節の符号拡張命令の挿入処理によって新たに挿入された命令に対する作成時間も含まれている．そこで，挿入処理による命令数の増加を調べた結果，全体の命令

表 5 IA64 型アーキテクチャの JIT コンパイル時間の内訳
Table 5 Breakdown of JIT compilation times for IA64 type architecture.

	符号拡張命令の最適化 (all)	UD/DU 作成	その他
mtrt	0.20%	2.11%	97.68%
jess	0.22%	1.67%	98.11%
compress	0.17%	2.40%	97.42%
db	0.15%	3.32%	96.53%
mpegaudio	0.15%	1.73%	98.12%
jack	0.11%	2.83%	97.06%
javac	0.21%	1.68%	98.12%
相加平均	0.17%	2.25%	97.58%

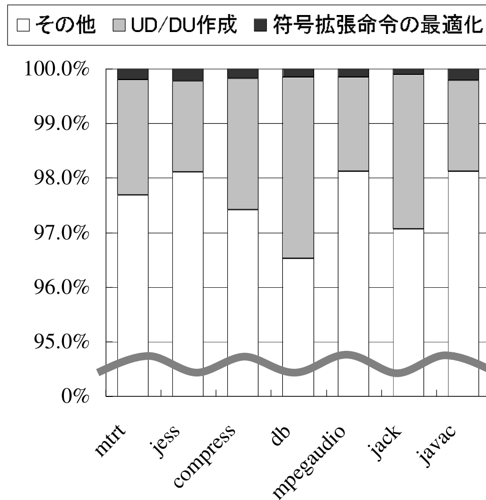

図 12 IA64 型アーキテクチャのコンパイル時間の内訳
Fig. 12 Breakdown of JIT compilation times for IA64 type architecture.

数から比べて平均約 1.6%程度命令数が増加することが分かった．これを考慮すると，我々の実装における符号拡張命令の最適化では，平均約 0.21%（0.17+0.04）という非常にわずかなコンパイル時間の増加で，表 3, 表 4 で示したような大きなパフォーマンスの向上を達成できる．

6. 終わりに

本稿では，符号拡張命令の除去についての新しいアルゴリズムを述べた．符号拡張命令の挿入処理は，符号拡張命令を実行方向に移動させたと仮定し，移動できない境界部分に符号拡張命令を挿入する．これによって，ループの中の符号拡張命令をループの外に移動することができる．配列に関する言語仕様を利用することによって，単純には除去できない配列アクセスのアドレス計算のための配列インデックスに対する符号拡張命令を除去する．符号拡張命令の除去順の決定を行うことにより，多く実行されると予想される頻度順に，符号拡張命令の除去を行う．これらの最適化を適用した結果，わずかなコンパイル時間の増加で多くの符号拡張命令を除去することができた．我々のアルゴリズムは，Java 言語だけではなく，符号拡張命令を必要とするほかの言語・アーキテクチャなどに対しても応用可能である．我々は，本稿で述べた手法の重要性が，将来高まることを期待している．

謝辞 本研究を進めるにあたり，貴重なご意見をいただいた IBM 東京基礎研究所 JIT compiler グループの皆様に深く感謝します．

参 考 文 献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: コンパイラ II 原理・技法・ツール，サイエンス社 (1990).
2) Ball, T. and Larus, J.R.: Optimally Profiling and Tracing Programs, *Principles of Programming Languages* (1992).
3) Blume, W. and Eigenmann, R.: Symbolic range propagation, *9th International Parallel Processing Symposium*, Santa Barbara, CA, pp.357–363 (1995).
4) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley Publishing Co., Reading (1996).
5) Harrison, W.: Compiler Analysis of the Value Ranges for Variables, *IEEE Trans. Softw. Eng.*, Vol.3, No.3, pp.243–250 (1977).
6) IBM Corp.: PowerPC homepage. http://www.chips.ibm.com/products/powerpc/
7) Intel Corp.: Itanium Architecture — Manuals. http://www.intel.com/design/itanium/manuals/
8) Ishizaki, K., Kawahito, M., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Design, implementation, and evaluation of optimizations in a just-in-time compiler, *Proc. ACM SIGPLAN Java Grande Conference* (1999).
9) Knoop, J., Rüthing, O. and Steffen, B.: Lazy code motion, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.224–234 (1992).
10) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *ACM Trans. Prog. Lang. Syst.*, Vol.17, No.5, pp.777–802 (1995).
11) Muchnick, S.S.: *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc. (1997).
12) Sarkar, V.: Determining Average Program Ex-

ecution Times and Their Variance, *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp.298–312 (1989).

13) Standard Performance Evaluation Corp.: SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/

14) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol.39, No.1, pp.175–193 (2000).

15) 古関 聡，小松秀昭：非常に偏った条件分岐が存在するプログラムのデータフロー最適化，情報処理学会論文誌：プログラミング，Vol.42, No.2, pp.26–36 (2001).

付　録

A.1 本アルゴリズムの適用例

図 13 に本アルゴリズムを適用した場合の例をいくつか紹介する．(a) は一般的なプログラムで比較的よく使われるループの例である．この場合は，ループ内の符号拡張命令は 3.2.4 項の定理 2 より除去可能である．ループ外の符号拡張命令は IA64 の場合は定理 1 より，PPC64 の場合はメモリからのロードに対する符号拡張命令のため，どちらのアーキテクチャでも除去を行うことができる．

(b) は (a) から配列アクセスの位置を変えた場合の例である．この場合は，ループ内の符号拡張命令については定理 2 より除去可能だが，ループ外の符号拡張命令は IA64 型アーキテクチャの場合は除去できないという点が重要である．除去を行った場合を仮定すると，mem の値が 0xffffffff(−1) のときは最初の配列アクセスの際に，インデックス i の値が 0x100000000 となってしまう．本来インデックス i は 0 としてアクセスされるべきであり，除去を行った場合は正しい最適化結果とならない．これは，IA64 型アーキテクチャではメモリからのロードはゼロ拡張されることが影響している．

(c) は "a[i+j-1] = 0; a[i+j] = 0; a[i+j+1] = 0;" というプログラムをコンパイルした場合の結果を示している．まず，共通部分式の除去処理によって，このプログラムは「最適化前」のように変形される．この中間コードに対して符号拡張命令の最適化を行うと，配列アクセスの直前にある 2 つの符号拡張命令は，ともに 3.2.4 節の定理 3 より除去することができる．

(d) は部分的冗長性（partial redundancy）を持つ符号拡張命令が最適化される例である．この最適化結果は符号拡張命令の挿入と除去順の決定により達成できる．まず，符号拡張命令の挿入によって if 文の中に t に対する符号拡張命令が挿入される．次に，除去順の決定では ① if 文の前，② if 文の中，の順で除去が行われ，「最適化後」の結果のようになる．

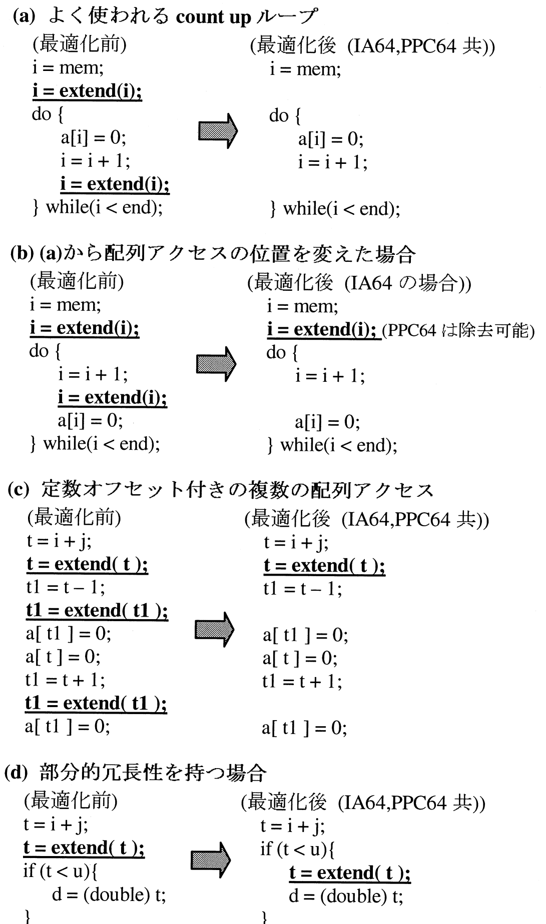

図 13 本アルゴリズムの適用例
Fig. 13 Examples of our sign extension elimination.

(平成 13 年 5 月 31 日受付)
(平成 13 年 8 月 31 日採録)

川人 基弘（正会員）

1968 年生．1991 年早稲田大学理工学部電子通信学科卒業．同年日本 IBM に入社．現在，同社東京基礎研究所に所属．コンパイラの研究に従事．

小松　秀昭（正会員）
1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM 東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

中谷登志男（正会員）
1975 年早稲田大学理工学部数学科卒業．同年，日本 IBM（株）野洲工場入社．1983 年から米国プリンストン大学大学院（コンピュータ・サイエンス学科）．1985 年同大学から M.S.E. および M.A.，1987 年同大学から Ph.D.．同年より，日本 IBM（株）東京基礎研究所に移り，VLIW コンパイラ，HPF コンパイラ，JIT コンパイラなどのプロジェクトを担当．一貫して，プログラムを最適化して高速に実行させるための新しいソフトウェア技術について研究開発している．現在，IBM Distinguished Engineer，ネットワーク・コンピューティング・プラットフォーム担当．