C++言語における機器組み込み向け最適化手法

春名修介[†] 坂田俊幸[†] 富永宣輝[†]
漆原誠一[†] 田中裕久[††] 湯川博司[††]

近年，民生機器に搭載される組み込みソフトウェア量の増加は著しく，C++言語による組み込みソフトウェア開発が要望されている．しかし，C++言語はメモリ増加，実行速度低下などの問題があり，ハードウェア制約が厳しい組み込みソフトウェアの開発に適用することが困難である．本稿では，これらの課題を解決する最適化手法を提案する．これらの手法をコンパイラに実装することにより，(1) 仮想関数呼び出しのオーバヘッドの削除，(2) 定数クラスオブジェクトのROM配置，(3) ディジタル信号処理（飽和演算，積和演算など）のオブジェクト指向言語による記述と高速実行の両立，が可能となる．これらの手法は，コンパイル・リンク時間の増加を抑えた手法であり，ハードウェアの制約などを満たすために修正・コンパイル・リンク・テストを頻繁に繰り返す組み込みソフトウェアの開発に適している．

Optimization of C++ Programs for Embedded Systems

SHUSUKE HARUNA,[†] TOSHIYUKI SAKATA,[†] NOBUKI TOMINAGA,[†]
SEIICHI URUSHIBARA,[†] HIROHISA TANAKA[††] and HIROSHI YUKAWA[††]

Recent remarkable increase of the size of program codes for embedded systems requires software development by C++ language. However, a couple of problems of C++ language that memory size and CPU time are wasted prevent applying to the development for embedded software that the hardware resources are limited. In this paper, some optimization techniques are proposed to address the problems. By installing these techniques in a compiler, 1) the overhead of virtual function calls can be deleted, 2) the constant class objects can be located in ROM area, and 3) Both description of multimedia processing in the high-level language and high-speed execution are achieved. They are also suitable for development of the embedded software that correction, compilation, link, and test are frequently repeated until satisfying hardware constraints, because they do not increase the compiling and linking time.

1. はじめに

近年，民生機器のマルチメディア化に伴い，組み込みソフトウェアには，量的な変化と質的な変化が起こっている．

量的な変化としては，ユーザインターフェースの充実やインターネット接続などの機能向上による，組み込みソフトウェア量の著しい増加が挙げられる．例えば，ディジタル放送用のセットトップボックスでは，2メガバイトのソフトウェアが搭載されている[1]．このため，従来から組み込みソフトウェアの開発言語として広く普及しているC言語より生産性の高い言語が必要となってきている．

質的な変化としては，従来は専用ハードウエアで処理されていたディジタル映像の圧縮，伸長，重ね合わせなどのメディア処理を，機器組み込みマイクロプロセッサ（以下，マイコンと記す）と組み込みソフトウェアで実現するようになったことが挙げられる．これは，低価格要求の厳しいディジタルカメラなどの小型民生機器で顕著である．そのため，マイコンにこれらのメディア処理を支援する積和演算命令や飽和演算命令などのメディア処理命令が搭載されるようになっている[2]．しかし，メディア処理は，C言語の持つ文法（データ形式や演算子）の範囲では扱えないので，C言語ではメディア処理の記述ができない[3]．その結果，メディア処理命令を使うソフトウェアの開発は，生産性の低いアセンブラ言語で行なわなければならないという問題がある．

† 松下電器産業株式会社 マルチメディア開発センター
Multimedia development Center, Matsushita Electric Industrial Co.,Ltd.
†† 松下電器産業株式会社 半導体開発本部
Corporate Semiconductor Development Division, Matsushita Electric Industrial Co.,Ltd.

量的な変化への対応策として，保守性・再利用性の高いオブジェクト指向言語による生産性向上が挙げられる．特に，C++言語[4]は，C言語のソフトウェア資産と開発手法を引き継げる優位性がある．

質的な変化への対応策として，オブジェクト指向言語によるメディア処理記述が挙げられる．オブジェクト指向言語を用いると，メディア処理をクラス定義可能となるので，従来のアセンブラ言語に代わって高級言語による記述が可能となり，メディア処理を含むプログラムの生産性を向上させることが可能となる[5]．

これらの2つの観点から，次期組み込みソフトウェア開発言語として，C++言語が適していると考えられる．

しかし，C++言語は，本来コンピュータ上のアプリケーションを開発する目的で作られた言語であり，組み込みソフトウェア開発にそのまま適用することは困難である．

本稿では，組み込みソフトウェア開発に適した，C++言語における最適化手法を提案する．また，それらを当社の32ビット機器組み込みマイコン[2]用C++言語コンパイラに適用した結果について報告する．

2章では，組み込みソフトウェアの特性を明らかにし，C++言語を組み込みソフトウェア開発に適用する場合の課題について述べる．3，4，5章では，それらの課題を解決するための組み込み向け最適化手法について説明する．6章では，それらの最適化手法の有効性についての評価を行なう．7章では，まとめと今後の展望について述べる．

2. 組み込みソフトウェアの特性とC++言語適用時の課題

組み込みソフトウェアが搭載される民生機器には，以下のようなコンピュータとは異なる制約が存在する．

- コストセンシティブである．コストを可能な限り削減するため，必要最小限の性能のマイコンと小容量のメモリで機能を実現する必要がある[6]．
- 低消費電力要求がある．携帯電話などの携帯機器では，消費電力の低減は必須であり，マイコンの動作周波数は可能な限り低くする必要がある．
- メモリに読み出し専用のROMと読み書き可能なRAMの明確な区別がある．実行コードはROMに，作業用のデータはRAMに置かれる．民生機器では，ROM，RAM，マイコンを一体に集積した1チップマイコンが使われることが多い．1チップマイコンではROM，RAMが面積のほとんどを占め，また，RAMはROMに比べ面積が4～5倍程度大きいため，極力RAMの使用量を削減することが求められる．

このように，組み込みソフトウェアはハードウェアの制約が大きく，これらの制約を満たすために，EC++言語[7]のような組み込みソフトウェアの開発向けにC++言語の言語仕様を制限する試みがなされているが，言語仕様の制限だけでは，以下のような問題を解決することができない．

(1) C++言語の仮想関数呼び出しは，直接呼び出しであるC言語の関数呼び出しと異なり，実行時に呼び出す関数の検索が必要な間接呼び出しであるため，実行性能が低下する．また，検索に必要な実行コードも増加する．

(2) C++言語のオブジェクトは，C言語と異なり，実行時に初期化が行なわれるため，定数オブジェクトであってもRAMに配置され，必要RAM量が増加する．例えば，前述したセットトップボックスでは，フォントやグラフィクス用ビットマップデータなどの定数オブジェクトが1メガバイトに達することがあり，それらが全てRAMに配置されることは避けなければならない．

(3) C++言語でメディア処理を記述した場合，処理本体はアセンブラ言語で書かれた外部関数となるので，関数呼び出し時のデータ受渡しや戻り値の受渡しのための冗長コード（以下，階層冗長コードと記す）が発生し，また，コンパイラによる最適化処理[8]が適用されないため，高速実行が要求されるメディア処理の実行速度が低下する．

使用メモリを削減し，低速プロセッサを用いて所定の実行性能を実現するためには，これらの問題を，コンパイラの最適化機能の強化により解決することが必須となっている．

しかし，組み込みソフトウェア開発では，コンピュータのソフトウェア開発とは異なり，**図1**に示すように修正・コンパイル・リンク・テストを繰り返すカットアンドトライの開発要素が多い．

その一つに，開発したプログラムがROM，RAMサイズの制約内に収まることを確認するメモリ制約テストがある．ROM，RAMサイズは，製品の仕様変更などの理由によりしばしば開発中に変更が起こり，その都度，処理内容の変更が発生する．

また，組み込みソフトウェアの開発では，ハードウエア制御を行う関係上，処理に要する時間が規定のタイミングに適合しているかの確認を行うタイミングテ

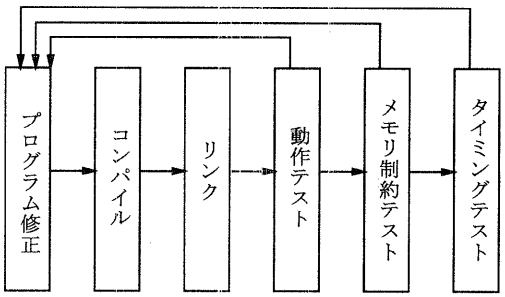


図 1 組み込みソフトウェアの開発形態
Fig. 1 Development proccess of software for embedded system.

ストがある．割り込みタイミングへの追従性の確認，微妙なタイミング調整のためのダミーコードの挿入などが行われる．

また，不要輻射の低減のため，マイコンのクロック周波数を下げる設計変更が発生することもあり，その都度，処理の見直し・改良が行われる．

これらの問題は，モジュール単体のテストの段階では分からず，プログラム全体を結合してテストするシステムテストの段階で発覚することが多く，プログラム全体をコンパイル・リンク・テストする工程を繰り返さなければならない．よって，ソフトウェアの生産性を上げるためには，このテストのターンアラウンドタイムをできるだけ少なくする必要があり，最適化処理のためにコンパイル及びリンクの時間を増加させることは避けなければならない．

これらの組み込みソフトウェア特有ともいえる要件を満たしながら，前述した3つの問題を解決する最適化手法について，以下で述べる．

3. 仮想関数の直接呼び出し手法

仮想関数は，呼び出される関数が参照しているオブジェクトにしたがって動的に決定される機能である．この機能により，呼び出し元では実際のオブジェクトが何であるかを考慮しない記述が可能になり，プログラム間の独立性が高くなる．例えば，オブジェクトの種類を増やしても，呼び出し元プログラムを変更する必要がなくなる．

しかし，仮想関数呼び出しは呼び出す関数を実行時に動的に決定するため，関数の検索処理が必要となる．このため，C言語の関数呼び出しと比べて，コードサイズが大きく，実行性能が低下する．

よって，仮想関数を必要としない場合は，仮想関数を使用しないことが望ましい．しかし，C++言語では，基底クラスの関数が仮想関数である場合は，その

```
//クラス B の定義
class B {
    int b;
  public:
    virtual int vfunc(void);//仮想関数
};
//クラス B を継承した クラス D の定義
class D : public B {
    int d;
  public:
    int vfunc(void);//仮想関数
};
//仮想関数呼び出しを含むプログラム
void func(D& obj)
{
     obj.vfunc();//仮想関数呼び出し
}
```

図 2 仮想関数呼び出しを含むソースプログラム
Fig. 2 source program with virtual function call.

```
func:
  mov  (0,A0),A1 ;;
  mov  (0,A1),A1 ;; 関数検索処理
  call (A1)
  rts
```

図 3 従来手法での図 2 の翻訳結果の機械語命令列
Fig. 3 assembler codes of Fig.2.

派生クラスの関数は仮想関数の指定をしなくても仮想関数として扱われるという問題がある．

Java言語[9]では，仮想関数でないことを指定するキーワード final を言語仕様に導入することによりこの問題の解決を図っているが，言語仕様の標準化が進んでいるC++言語では言語仕様を変更することは困難である．

図2に例を示す．クラス B で関数 vfunc が仮想関数であるため，クラス D でも仮想関数となり，クラス D のオブジェクトを介した関数呼び出し obj.vfunc() は，従来の実装では関数検索処理を含む間接呼び出しとなっていた．

従来手法で図2をコンパイルした結果の機械語命令列を図3に示す．“mov (0,A0),A1”，“mov (0,A1),A1” はオブジェクト obj 内に保持している呼び出される関数のアドレスを検索する処理であり，“call (A1)” は検索した関数のアドレスに分岐する間接呼び出し命令である．

そこで，派生クラスを持たないリーフのクラスの仮

想関数は，再定義されないため，呼び出す関数がコンパイル時に静的に決定できることに着目し，そのような仮想関数を関数検索処理を含まない直接呼び出しに変換する方法を検討した．

ここで問題となるのは，ソフトウェア開発で一般に用いられている分割コンパイルでは，プログラム全体を参照することができないため，クラスがリーフのクラスであるか否かの判定がコンパイル時にできないことである．

この問題を解決するための方法としては，分割コンパイルをせずに，すべてのファイルを1つにまとめてコンパイルする方法[10]が提案されているが，1つのファイルを修正した場合でも，すべのファイルを1つにまとめて再度コンパイルするので，再コンパイルに時間がかかる．また，別の方法としては，プログラムをコンパイラ内部の中間形式である中間コードに翻訳し，すべての中間コードファイルを結合し最適化処理を行なう方法[11]が提案されているが，この方法でも，すべての中間コードファイルをまとめて最適化処理を行なうため，1つのファイルを修正したときの再コンパイルに時間がかかる．このため，頻繁に再コンパイルを繰り返すことを特徴とする組み込みソフトウェア開発に対して，これらの方法を適用することは困難である．

分割コンパイルの利点を活かしながら，プログラム全体を参照するためには，コンパイルされたすべての機械語命令列をリンクする機能を果たすリンカで処理を行なうのが適切である．

そこで，コンパイル時に仮想関数呼び出しを直接呼び出しと関数検索処理を含む間接呼び出しの両方の情報を持つ疑似命令に翻訳し，リンカでどちらかの呼び出しを選択する最適化手法を考案した．この手法を用いれば，1つのファイルの修正によるコンパイル時間の増加を，そのファイルのコンパイル時間だけに抑えることができる．

コンパイラの処理は以下のようになる．

(1) リンク時にクラスがリーフのクラスであるか否かを判定するために，基底クラスとして使用されているすべてのクラスに対して，そのクラス名の疑似命令を出力する．

(2) 仮想関数呼び出しを起動するオブジェクトのクラスがファイル内で基底クラスとして使用されているか否か判定する．

(3) 基底クラスとして使用されていれば，従来と同様に関数検索処理を含む間接呼び出しの機械語命令列に翻訳する．

```
.BASE B   ;;Bは基底クラスとして使用される
func:
  ;; クラス名，直接呼び出し，間接呼び出し
  ;; の情報を含む仮想関数呼び出し疑似命令
  callv D,vfunc,0,0,A1
  rts
```

図4 本手法でのリンク前の機械語命令列
Fig. 4 assembler codes before link.

(4) 基底クラスとして使用されていなければ，そのクラスのクラス名，直接呼び出しの関数のラベル名，及び関数検索処理を含む間接呼び出しの機械語命令列の情報をオペランドとする仮想関数呼び出し疑似命令に翻訳する．

図4に本手法での図2のソースコードのコンパイル結果の機械語命令列を示す．図2では，クラスBがクラスDの基底クラスとして使用されているので，そのことを表す疑似命令“BASE B”が出力される．また，クラスDは基底クラスとして使用されていないので，“obj.vfunc()”は，仮想関数呼び出し疑似命令“callv D,vfunc,0,0,A1”に翻訳されている．Dがクラス名，vfuncが直接呼び出しのラベル名，0,0,A1が関数検索処理を含む間接呼び出しの機械語命令列を生成するために必要な情報である．

また，リンカでの処理は以下のようになる．

(1) すべてのファイルの基底クラスの情報の和集合をとる．

(2) 仮想関数呼び出し疑似命令のオペランドのクラス名が基底クラス情報の和集合に含まれているか否か判定する．

(3) 含まれていない場合は，リーフのクラスであるので，仮想関数呼び出し疑似命令のオペランドの直接呼び出しのラベル名を用いて，仮想関数呼び出し疑似命令を直接呼び出しの機械語命令に変換する．

(4) 含まれている場合は，仮想関数呼び出し疑似命令のオペランドにある関数検索処理を含む間接呼び出しのための情報を用いて，関数検索処理を含む間接呼び出しの機械語命令列に変換する．

(5) 変換された機械語命令列に対してサイズの再計算を行なう．

図2のクラスDがすべてのファイルで基底クラスとして使用されていない場合は，基底クラスの集合には含まれないため，図4の仮想関数呼び出し疑似命令はリンカにより直接呼び出しに変換される（図5）．

この手法を用いることにより，リンカでは基底クラ

```
func:
  call vfunc  ;; 直接呼び出し
  rts
```

図 5 本手法でのリンク後の機械語命令列
Fig. 5 assembler codes after link.

スの集合からクラス名を検索する処理と疑似命令を変換する処理を行なうだけで，リーフクラスの仮想関数の直接呼び出しを実現できる．

なお，ここでは，リーフのクラスの仮想関数だけを直接呼び出しの対象としたが，再定義されない仮想関数すべてを最適化の対象とすることも可能である．その場合，コンパイラでは基底クラスの疑似命令（*.BASE* クラス名）の代わりに再定義される関数の疑似命令（*.BASE* 関数名）を，仮想関数呼び出し疑似命令の第1オペランドにクラス名ではなく，関数名を出力するようにする．この変更により，リンカではリーフクラスを求めたのと同じ処理で再定義されない仮想関数を求めることができる．

4. 定数オブジェクトの ROM 配置手法

C 言語では，構造体のデータを初期化する方法は，図 6 に示すようなメンバの値を列挙する方法である．

コンパイラは，初期値を静的に決定できるため，定数オブジェクトを ROM 領域に配置するデータに翻訳することができる．

しかし，C++言語では，構造体としては記述できないクラスのオブジェクトを初期化する方法は，図 7 に示すような，初期化関数であるコンストラクタ呼び出しによる方法に変更され，列挙による方法は許されなくなった．

この場合，オブジェクトの初期化がコンストラクタの呼び出しにより実行時に動的に行なわれるため，定数オブジェクトは ROM 領域に配置することができず，RAM 領域に配置するデータに翻訳されてしまう．従来の手法での図 7 の定数オブジェクト objp のコンパイル結果の機械語命令列を図 8 に示す．

```
/* 構造体 P の定義 */
struct P {
    int x;
    int y;
};
/* 初期設定文 */
const struct P objp = { 10, 100 };
```

図 6 定数データを含むソースプログラム
Fig. 6 source program with const data.

```
//クラス P の定義
class P {
    int x; //データメンバ
    int y; //データメンバ
  public:
    virtual getPoint(void); //仮想関数
    P(int a, int b)//コンストラクタ
      :x(a),y(b)   //コンストラクタ初期設定子
      {}           //関数本体
};
//初期設定文
const P objp(10, 100); //コンストラクタを呼び出す
```

図 7 定数オブジェクトを含むソースプログラム
Fig. 7 source program with const object.

```
_initialize:      ;; 初期化処理を行なう関数
  mov  &objp,R0   ;;objp のアドレス設定
  mov  10,R1      ;; 第 1 引数設定
  mov  100,R2     ;; 第 2 引数設定
  call P          ;; コンストラクタ呼び出し
  rts
BEGIN RAM_AREA    ;;RAM 領域に配置
objp:
      .area 8     ;; 領域 8 バイト確保
END RAM_AREA
```

図 8 従来手法での定数オブジェクトの機械語命令列
Fig. 8 assembler codes of const object.

RAM は，ROM に比べトランジスタ数が多く，チップ面積が大きくなるため高価であり，組み込み用途では，定数データを ROM に配置する機能は必須である．

EC++言語では，以下のような条件を満たすクラス，すなわち構造体と等価なクラスの定数オブジェクトのみを，C 言語における構造体のデータを初期化するのと同様の方法を用いて ROM に配置することで，この問題に対応している [7]．

- コンストラクタ，デストラクタをユーザ定義しない．
- 他のクラスを継承しない．
- 仮想関数をもたない．
- クラスメンバのアクセス指定はすべて public である．

しかし，これらの制限の下では差分プログラミング，動的バインディング，情報隠蔽というオブジェクト指向の根幹をなす機能を利用することができない．

そこで，コンストラクタ実行後のオブジェクトの初期値をコンパイル時に計算することで，一般のクラスの定数オブジェクトを ROM に配置する手法を考案した．

ここで，計算対象としているクラスをコンストラクタが以下の条件を満たすものに制限している．この最適化処理の目的は，定数オブジェクトの ROM 配置であり，オブジェクト指向本来の機能を有する定数オブジェクトをこの制限内で記述可能である．

- クラス内で定義されている．
- コンストラクタ初期設定子のみからなり，関数本体が空である．
- 設定する初期値に外部変数，関数呼び出しを使用していない．

また，計算対象を限定することには以下の利点もある．

- ユーザに対して，定数オブジェクトを ROM に配置する方法を明確に提示できる．
- コンパイル時間の増加を抑えられる．

オブジェクトの初期値は，図 7 に示すような実際の値を設定する初期設定文から，オブジェクトを初期化する関数であるコンストラクタを呼び出すことにより設定される．そこで，コンストラクタの仮引数と初期設定文の実際の値をコンパイル時に結び付けて，定数畳み込み最適化処理を行なうことにより，初期値を計算する．

コンパイラでのコンストラクタの処理は以下のようになる．

(1) コンストラクタが前述した条件を満たすか否かを判定する．
(2) 満たしていた場合，コンストラクタの仮引数を含むオブジェクトのデータメンバの値が，コンパイラの内部データ（以下，仮初期値式と記す）としてつくられる．

図 7 のコンストラクタの定義では，x(a) から，データメンバ x の値が a で，y(b) から，データメンバ y の値が b である仮初期値式がつくられる．

また，コンパイラでの初期設定文の処理は以下のようになる．

(1) 初期設定文により呼び出されるコンストラクタを求める．
(2) そのコンストラクタの仮初期値式がつくられているか否か判定する．
(3) つくられていた場合は，仮初期値式に含まれるすべての仮引数を実引数に置き換える．
(4) 置き換えた初期値式に対して，定数の畳み込みを行ない，値を計算し，定数か否かを判定する．
(5) 定数であった場合，初期設定されるオブジェクトが定数（const）オブジェクトか否か判定する．
(6) 定数オブジェクトであった場合，初期設定文を ROM に配置されるデータに翻訳する．
(7) 定数オブジェクトでなかった場合，初期設定文を RAM に配置するデータに翻訳する．
(8) それ以外の場合，初期設定文を従来と同様に RAM 領域に配置するデータとコンストラクタ呼び出しのコードに翻訳する．

図 7 の初期設定文では，コンストラクタの仮引数 a が実引数 10 に，仮引数 b が実引数 100 に置き換えら

```
BGEIN ROM_AREA  ;;ROM 領域に配置
objp:
        .int vtbl_P   ;;P の仮想関数情報
        .int 10       ;; メンバ x の初期値
        .int 100      ;; メンバ y の初期値
END ROM_AREA
```

図 9 本手法での定数オブジェクトの機械語命令列
Fig. 9 assembler codes of const object.

```
//クラス P の定義
class P {
    int x;
    int y;
  public:
    virtual getPoint(void); //仮想関数
    //コンストラクタ
    P(int a, int b)
      :x(a),y(b) //コンストラクタ初期設定子
      {}         //関数本体
};
//クラス Q の定義
class Q : public P { //Q は P を継承
    int z;
  public:
    //コンストラクタ
    Q(int l, int m, int n)
      : P(l,m),z(n) //コンストラクタ初期設定子
      {}            //関数本体
};
//初期設定文
const Q objq(10, 100, 100);
```

図 10 派生クラスの定数オブジェクトを含むソースプログラム
Fig. 10 source program with const object of derived class.

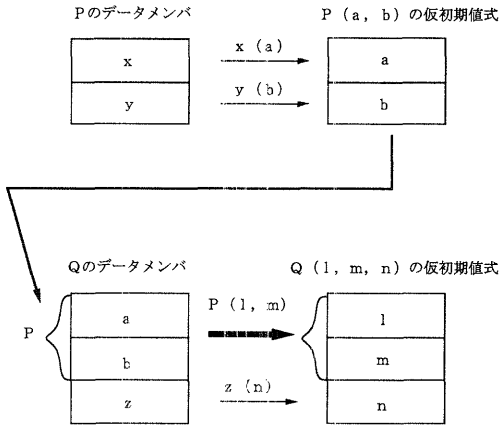


図 11 仮初期値式の例
Fig. 11 example of initialized expression.

れ，定数畳み込みが行なわれる．その結果，データメンバ x の値は 10，y の値は 100 であることが計算され，データメンバがすべて定数であるので，定数オブジェクト objp は，図 9 に示されるように ROM 領域に配置される．なお，"vtbl_P" はオブジェクト内に保持される呼び出される仮想関数の情報であり，コンパイラにより自動的に付加される．

この手法により，C++言語における定数オブジェクトが，C 言語と同様に ROM 領域に配置可能となり，実行時に必要な RAM 容量を削減できる．

本手法は，他のクラスを継承するクラス（派生クラス）のオブジェクトの初期化に対しても適用できる．コンストラクタ初期設定子には，そのクラスが継承しているクラス（基底クラス）のコンストラクタによる初期設定が記述できる．図 **10** にその例を示す．この場合クラス Q のコンストラクタに対応する仮初期値式は，P のコンストラクタの仮初期値式を利用してつくり出す．このとき P の仮初期値式の仮引数を Q の仮引数で置き換える．つまり，a→ l, b→ m の置き換えが行なわれる．後の処理は同様である．生成される仮初期値式を図 **11** に示す．

5. メディア処理命令最適化手法

近年，画像処理等のメディア処理のために SIMD 命令 [3]，積和演算命令，飽和演算命令 [2] のようなメディア処理命令を搭載したマイクロプロセッサが増えている．

例えば，画像の重ね合わせ処理では，重ね合わせた結果，オーバフローにより画像が反転しないように，オーバフローする場合には値を最大値に固定する飽和加算命令が使用される．

しかし，これらのデータ形式や演算子は C 言語では提供されていないので，C 言語の枠組で記述することは困難である．

C++言語では，クラスはユーザ定義型として使用できるので，これを用いることにより，自然な形で上記のデータ形式を記述することができる．また，C++言語では演算子の多重定義を用いることにより，そのデータ形式を使用した演算子も定義できる．16 ビット飽和整数型のクラスの定義の例を図 **12** に示す．

しかし，メディア処理の処理内容自体は，C++言語では記述できないため，アセンブラ言語で記述する必要があった．16 ビット飽和加算のアセンブラ言語による記述を図 **13** に示す．

このような処理はアセンブラ言語で記述されているため，コンパイラの共通部分式最適化，定数伝搬最適化等の中間コード最適化処理 [8] を適用できず，機械語命令レベルのインライン展開のみが行なわれていた．このため，コード生成効率が悪いという問題があった．

図 **14** に飽和加算を使用したプログラムを，図 **15** に図 14 のプログラムをコンパイルした結果の機械語命令列を，図 **16** に図 15 に対して機械語命令レベルで，インライン展開処理を行なった結果を示す．

飽和加算演算が共通部分式最適化処理の対象とならないため，"a+b" の計算を 2 度行なっている．また，レジスタの再割り付けなしには，レジスタの退避・復

```
class SatShort {
    short s;
  public:
    //飽和加算を行なう多重定義演算子
    SatShort operator+(SatShort);
    //飽和減算を行なう多重定義演算子
    SatShort operator-(SatShort);
        :
};
```

図 12 飽和整数型の定義
Fig. 12 definition of saturation integer type.

```
_satadd ;; 飽和加算を行なう関数
  mov   (A0),D0 ;; 引数とりだし
  mov   (A1),D1 ;; 引数とりだし
  add   D1,D0   ;; 加算命令
  sat16 D0      ;; 飽和演算命令
  rts
```

図 13 飽和加算関数の機械語命令列
Fig. 13 assembler codes of saturation addition function.

```
extern SatShort out1, out2;
void func(SatShort a, SatShort b)
{
    //a, b は飽和整数型
    out1 = a + b; //飽和加算関数を呼び出す
        :
    out2 = a + b; //飽和加算関数を呼び出す
}
```

図 14 飽和演算を使用したプログラム
Fig. 14 source program with saturation operation.

```
func:
  mov  A0,A2      ;; レジスタを退避
  mov  A1,A3      ;; レジスタを退避
  call _satadd    ;; 飽和加算関数を呼び出す
  mov  D0,(out1)
       :
  mov  A2,A0      ;; レジスタを復帰
  mov  A3,A1      ;; レジスタを復帰
  call _satadd    ;; 飽和加算関数を呼び出す
  mov  D0,(out2)
  rts
```

図 15 従来手法による図 14 の機械語命令列
Fig. 15 assembler codes of Fig.14.

元のコードを削除することは困難である.

そこで, C++言語のメディア処理の記述とマイコンに搭載されたメディア処理命令を対応付ける組み込みクラス (以下, メディア処理対応クラスと記す) を定義し, メディア処理記述から直接メディア処理命令を生成する手法を考案した.

例えば, マイコンに飽和演算命令が搭載されている場合には, 図12の飽和整数型のようなクラスをメディア処理対応クラスとして定義する.

ユーザはこのクラスを用いて, メディア処理を記述する.

プログラム中のメディア処理記述を機械語命令列に翻訳するためにコンパイラが行なう処理の流れを図 **17** に示す.

コンパイラは中間コード最適化処理を行なう前に, メディア処理記述をその処理内容に対応するメディア処理中間コードへ変換する. これにより, メディア処理記述は関数呼び出しではなく, 加算などの演算子と同様に扱われるため, 中間コード最適化処理により階層冗長コードが削除される.

図 **18** に本手法を用いた場合の図 14 のプログラムをコンパイルした結果の機械語命令列を示す. 図 14 の "a+b" は関数呼び出しとして扱われず, 演算命令

```
func:
  mov  A0,A2      ;; レジスタを退避
  mov  A1,A3      ;; レジスタを退避
  mov  (A0),D0
  mov  (A1),D1
  add  D1,D0      ;; a+b を計算
  sat16 D0        ;;
  mov  D0,(out1)
       :
  mov  A2,A0      ;; レジスタを復帰
  mov  A3,A1      ;; レジスタを復帰
  mov  (A0),D0
  mov  (A1),D1
  add  D1,D0      ;; a+b を計算
  sat16 D0        ;;
  mov  D0,(out2)
  rts
```

図 16 インライン展開後の図 15 の機械語命令列
Fig. 16 assembler codes of Fig.15 after inline expansion.

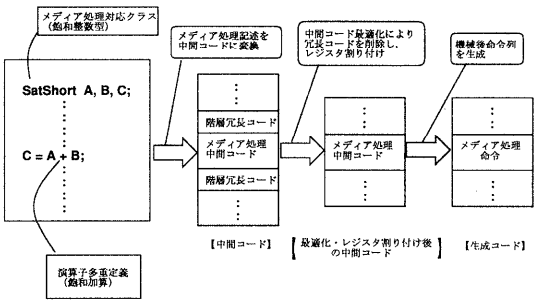


図 17 メディア処理命令直接生成
Fig. 17 Direct generation of media processing instructions.

として扱われるので, レジスタ退避及び復帰のコードが削除される. また, "a + b" に対して共通部分式最適化処理が適用され, 2回目の "a + b" のコードが削除され, コードサイズ, 実行性能が改善される.

なお, メディア処理対応クラスを使用したプログラムを他の処理系でも利用可能にするためには, 多重定義演算子に対する C++言語でのメディア処理命令を使用しない定義を提供すればよい. それらを用いることにより, コード生成効率は悪くなるが同じプログラムが他の処理系でも利用可能となる. 例えば, 飽和加算命令については図 **19** のような関数の定義を提供すればよい.

また, ここでは飽和整数型についてのみ説明したが, SIMD 命令についても, 同様の方法が使用できる. この場合, 図 **20** に示すような SIMD 型のメディア処理

```
func:
  mov  (A0),D0
  mov  (A1),D1
  add   D1,D0      ;; a+b を計算
  sat16 D0         ;;
  mov   D0,(out1)
        :
  mov   D0,(out2)
  rts
```

図 18 本手法による図 14 の機械語命令列
Fig. 18 assembler codes of Fig.14.

```
SatShort SatShort::operator+(SatShort v)
{
    /* int 型で加算を行なう */
    int result = s + v.s;

    /* 飽和処理 */
    if (result > SHORT_MAX) {
      result = SHORT_MAX;
    }
    else if (result < SHORT_MIN) {
      result = SHORT_MIN;
    }
    return SatShort(result);
}
```

図 19 飽和加算関数の定義の例
Fig. 19 example of definition of saturation addition function.

```
class SIMD8x8 {
    char c[8];
  public:
    //パックされたバイトを加算する
    SIMD8x8 operator+(SIMD8x8);
    //パックされたバイトを減算する
    SIMD8x8 operator-(SIMD8x8);
        :
};
```

図 20 SIMD 型の定義
Fig. 20 definition of SIMD type.

対応クラスを定義し，SIMD 型のメディア処理演算子を対応するメディア処理中間コードに変換する処理，その中間コードに対するレジスタ割り付け処理，機械語命令列生成処理をコンパイラに追加することで対応可能である．

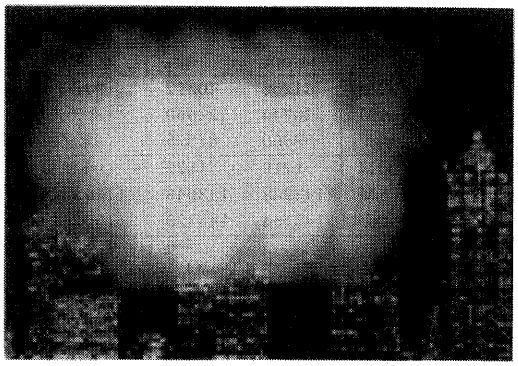


図 21 画像重ね合わせの例
Fig. 21 example of bitmap blending.

6. 評 価

前述した 3 種類の最適化手法を適用しない場合（no-opt）と適用した場合（opt）のコードサイズ，実行時間の比較を，**表 1** に示す 2 種類の C++言語のソースプログラムについて行なった．sample1 は，定数データが多用されている製品プログラムの一部を C++言語で書き換えたものである．sample2 は，複数の画像を重ね合わせて 1 つの画像を生成するプログラムで，画像の計算に飽和加算が使用されている．sample2 のプログラムで計算した画像の表示例を図 **21** に示す．なお，sample2 の最適化処理を適用しない場合の飽和加算処理は，アセンブラ言語で記述した関数を呼び出すことにより行なった．

6.1 コードサイズ

表 2 は，コンパイラにより生成された実行形式のコードサイズを示すものである．

プログラム sample1 では，最適化を適用した場合に，定数オブジェクトの ROM 配置により，RAM が減少しているだけでなく，ROM も減少している．これは，図 **22** に示すように定数オブジェクトの ROM 配置による増加分以上に，定数オブジェクト初期化処理のコードの削除により，ROM が削減されたことによる．このことから，RAM サイズを削減することが目的である定数オブジェクトの ROM 配置手法は，ROM サイズの削減にも貢献することが分かる．

6.2 実行時間

表 3 は，プログラムの実行時間を示すものである．

表 1 評価プログラム
Table 1 source programs for evalution.

program	line	class
sample1	3831	44
sample2	242	3

表 2 コードサイズ (単位 バイト)
Table 2 code size.

program		no-opt	opt	(比率)
sample1	ROM	41256	40312	(97.7%)
	RAM	26744	21560	(80.6%)
	total	68000	61875	(91.0%)
sample2	ROM	1216	1172	(96.3%)
	RAM	143948	143948	(100.0%)
	total	145164	145120	(100.0%)

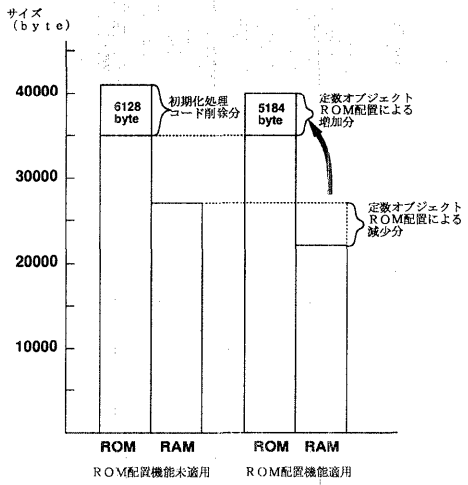


図 22 sample1 のコードサイズ
Fig. 22 code size of sample1.

表 3 実行時間 (単位 ミリ秒)
Table 3 execution time.

program	no-opt	opt	(比率)
sample1	74.59	73.64	(98.7%)
sample2	972.84	679.47	(69.8%)

プログラム sample2 では，最適化を適用することにより，実行性能が 1.4 倍になった．これは，sample2 では，実行頻度が最も高いサブルーチンが仮想関数であり，最適化によりその関数呼び出しが直接呼び出しに変換されているためと，そのサブルーチン内部で使用されている飽和加算命令をコンパイラが直接生成することにより，関数呼び出しのオーバヘッドが削除されているためである．仮想関数の直接呼び出し最適化のみを適用した場合は 1.1 倍，メディア処理命令最適化のみを適用した場合は 1.2 倍，実行性能が向上した．

6.3 コンパイル時間

表 4 は，プログラムのコンパイル及びリンクに要した時間を示すものである．評価は SPARC ワークステーション (SS-UA1, 167MHz) を使用して行なった．

sample1 では，コンパイルに要する時間は，最適化を行なった場合の方が，行なわなかった場合より少なくなっている．これは，定数オブジェクトが ROM に配置され，初期化のコードが削除されたため，その部分の解析に要していた時間が削減されたためである．この結果より，組み込みソフトウェア開発の要件である，最適化によるコンパイル・リンク時間の増加の抑制は達成されたと考えられる．

表 4 コンパイル時間 (単位 秒)
Table 4 Compilation time.

program		no-opt	opt	(比率)
sample1	Compiler	48.945	42.235	(86.3%)
	Linker	2.176	1.536	(70.5%)
	total	51.121	43.773	(85.6%)
sample2	Compiler	0.658	0.648	(98.5%)
	Linker	0.311	0.312	(100.3%)
	total	0.969	0.960	(99.1%)

表 5 最適化機能実装コスト
Table 5 Cost of implement.

最適化	ステップ数
仮想関数直接呼び出し	829
定数オブジェクト ROM 配置	900
メディア処理命令最適化	3575
計	5304

6.4 コンパイラ変更コスト

表 5 は，前述した 3 種類の最適化手法を，アセンブラ，リンカを含むコンパイラシステムに導入する際に要したコストを示すものである．コンパイラシステム全体のステップ数は，約 25 万ステップであり，全体の 2 %程度にあたる 5304 ステップの変更により，これらの最適化機能が実装できた．

また，上記のうちメディア処理命令最適化のみがマイコンに依存しており，他の異なったメディア処理命令を有するマイコン用に本コンパイラシステムを移植する場合には，この部分のみを変更すればよい．

7. おわりに

C++言語を組み込みソフトウェア開発に適用する際の問題を，コンパイル時間の増加を抑えて解決する最適化手法について述べた．仮想関数の直接呼び出し手法により実行性能の向上が，定数オブジェクトの ROM 配置手法により必要メモリ量の削減が可能となる．また，C++言語によるメディア処理記述からメディア処理命令を直接生成する手法では，メディア処理の高級言語による記述と高速実行の両立が達成できる．

今後は，更にハードウェアの制約が厳しい携帯機器などでもオブジェクト指向言語によるソフトウェア開発が適用可能なように，コンパイラの組み込み向け最適化機能の強化について検討していきたい．

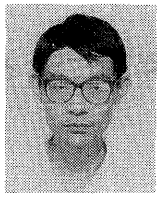
参考文献

1) 米田泰司, 西尾歳朗, 山本創造, 有賀健, 中南聰, 八木行雄, 岡村和男: ディレク・ティービー用ディジタルSTB, *Matsushita Technical Journal Vol.44 No.1* (1998).
2) 松下電子工業株式会社 マイコン事業部: MN10300シリーズ命令説明書.
3) 酒井淳嗣, 枝廣正人, 小長谷明彦: マルチメディア向けSIMD命令の生成手法, 情報処理学会研究報告 96-PRO-6 (1996).
4) Ellis, M. A. and Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison Wesley (1990).
5) Quackenbush, S. R. and Parikh, V. N.: Using C++ For Real-Time Signal Processing, *Proc. of the International Conference on Signal Processing Aplications and Technology* (1995).
6) 中本幸一, 高田広章, 多丸喜一郎: 組込みシステム技術の現状と動向, 情報処理 Vol.38 No.10 (1996).
7) The Embedded C++ Technical Committee: Rationale for the Embedded C++ specification Development, http://www.caravan.net/ec2plus.
8) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers Principles, Techniques, and Tools*, Addison Wesley (1988).
9) Gosling, J., Joy, B. and Steele, G.: *The Java^{TM} Language Specification*, Addison Wesley (1997).
10) Aigner, G. and Hölzle, U.: Eliminating Virtual Function Calls in C++ Programs, *ECOOP'96 Conference Proceedings* (1996).
11) Fernandez, M. F.: Simple and effective link-time optimization of Modula-3 programs, *In Proceedings of the SIGPLAN '95 Conference on Programing Language Design and Implementation* (1995).

(平成10年10月16日受付)
(平成10年12月16日採録)

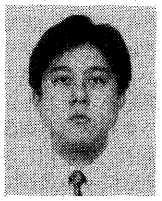
春名 修介（正会員）

昭和29年生. 昭和52年神戸大学工学部電子工学科卒業. 同年松下電器産業(株)入社. 現在, マルチメディア開発センター勤務. マイクロコンピュータアーキテクチャ, コンパイラ・OS等の基本ソフトウエア, ソフトウエア開発環境に関する研究開発に従事. ACM会員.



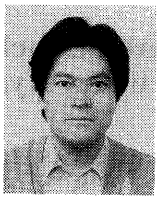
坂田 俊幸（正会員）

昭和44年生. 平成6年東京工業大学大学院電気・電子工学専攻修士課程修了. 同年松下電器産業(株)入社. 現在, マルチメディア開発センター勤務. プログラミング言語処理系の研究開発に従事.



富永 宣輝

昭和38年生. 昭和63年京都大学大学院工学研究科数理工学専攻修士課程修了. 同年松下電器産業(株)入社. 現在, マルチメディア開発センター勤務. コンピュータアーキテクチャ, 基本ソフトウェアおよびソフトウェア開発環境(コンパイラ, OS, VM等)の研究開発に従事.

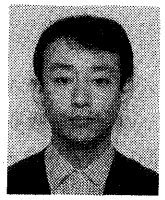


漆原 誠一（正会員）

昭和35年生. 昭和58年大阪大学基礎工学部生物工学科卒業. 昭和60年同大学院生物系研究科修士課程修了. 同年松下電器産業(株)入社. 現在, マルチメディア開発センター勤務. プログラム言語処理系, オブジェクト指向プログラミングなどの研究開発に従事. IEEE会員.

田中 裕久（正会員）

昭和43年生. 平成6年大阪大学大学院工学研究科電子工学専攻修士課程修了. 同年松下電器産業(株)入社. 現在, 半導体開発本部勤務. プログラミング言語処理系, 組み込みマイコンアーキテクチャの設計開発に従事.



湯川 博司（正会員）

昭和34年生. 昭和59年京都大学大学院工学研究科電気工学第二専攻修士課程修了. 同年松下電器産業(株)入社. 現在, 半導体開発本部勤務. プログラミング言語処理系, 組み込みマイコンアーキテクチャの設計開発に従事.