実行時再コンパイルを用いたHPFプログラムの最適化

荒木拓也[†] 村井均[†]
蒲池恒彦[†] 妹尾義樹[†]

最適化コンパイラは計算機の性能をできる限り引き出すため，さまざまな最適化を行う．しかし，ある種の最適化は変数値やシステムパラメータがコンパイル時に決定していないとできない場合がある．たとえば，並列化や強さの低減，ループ交換などは，それらが可能であるかを判定するための変数値がコンパイル時に決定していないと行うことができない．また，メモリ容量などのシステムパラメータがコンパイル時の最適化手法の選択に影響を与えることがある．本研究では，実行時にこれらの情報を収集し，それを元にプログラムの一部を実行時に再コンパイル，最適化することによりプログラム実行を高速化するシステムの試作を行った．本研究ではHPF（High Performance Fortran）で記述されたプログラムを対象にする．我々が実装したシステムでは，再コンパイルや実行時情報の管理などをプログラムを実行するプロセッサとは別プロセッサで行う．これにより，従来の同様の研究と異なり，再コンパイルのコストが実行時間に上乗せされないという利点がある．本研究では，このシステムを用いた実行時再コンパイルの予備評価を行い，1.3倍から数十倍の速度向上を確認した．これにより，実行時再コンパイル手法の有効性が確かめられた．

Optimization of HPF Programs with a Dynamic Recompilation Technique

Takuya Araki,[†] Hitoshi Murai,[†] Tsunehiko Kamachi[†] and Yoshiki Seo[†]

Optimizing compilers do various optimizations in order to exploit best performance from computer systems. However, some kinds of optimizations cannot be applied if values of variables or system parameters are not known at compilation time. For example, parallelization, strength reduction, or loop interchange are not allowed if values of variables which are used to examine correctness and effectiveness of the optimization are not known at compilation time. In addition, system parameters like memory size may affect selections of optimization methods. In this study, we implemented a system which collects such information at run time, and recompile and reoptimize parts of the program based on such information at run time. The language used in this system is HPF (High Performance Fortran). In our system, recompilation and management of runtime information are executed on a processor other than the processors which execute user programs. Therefore, recompilation cost is not added to execution time, unlike other similar systems. In this study, we preliminarily evaluated the dynamic recompilaiton technique using this system. The evaluation result shows that 1.3 to 80 times speedups can be attained. With this result, we confirmed the usefulness of the dynamic recompilation technique.

1. はじめに

最適化コンパイラは計算機の性能をできる限り引き出すため，さまざまな最適化を行う．しかし，ある種の最適化は変数値やシステムパラメータがコンパイル時に決定していないとできない場合がある．

たとえば，並列化や強さの低減，ループ交換などは，それらが可能であるかを判定するための変数値がコンパイル時に決定していないと行うことができない．また，通信速度やキャッシュ，メモリ容量などのシステムパラメータがコンパイル時の最適化手法の選択に影響を与えることがある．

本研究では，実行時にこれらの情報を収集し，その値を元にプログラムの一部を実行時に再コンパイル，最適化することによりプログラム実行を高速化するシステムの試作を行った．

本研究ではHPF（High Performance Fortran）で

† 新情報処理開発機構並列分散システムNEC研究室
Parallel and Distributed Systems NEC Laboratory, RWCP

記述されたプログラムを対象にするシステムを構築した．しかし，他のプログラミング言語を対象にした場合でも，システムの実装や評価結果の一般性は失われないと考えられる．

今回実装したシステムでは，サブルーチンを単位に再コンパイルを行う．また，対象にした実行時情報は，スカラ変数の値と残りメモリ容量である．スカラ変数の値を用いることにより，コンパイル時の解析では不可能であった並列化，強さの低減，ループ交換などが可能になる．また，利用可能なメモリ容量を実行時に取得することにより，メモリの使用量と実行時間にトレードオフがある場合に，利用可能なメモリの範囲で最も実行時間が短くなるような最適化を行うことができる．他の実行時情報を用いた最適化も今回実装した枠組み上に容易に実現することが可能である．

実行時の情報を用いた最適化を行うシステムとしては，他にマルチバージョン化と呼ばれる手法もある．この手法では，複数の版のコードを作成しておき，実行時の変数値に基づいて実行するコードを切り替えるが，コードサイズが爆発する危険性がある．

また，実行時の情報を用いてプログラムの再コンパイル，最適化を行うシステムも過去に提案されている．Java の JIT コンパイラにおける研究がさかんに行われているが，実行時情報として実行頻度情報だけを用い，コストのかかる最適化を行うかどうかを判断するという程度のものが多い．

変数情報を用いるものとしては，C 言語などを拡張したシステムがある．これらはほとんどが逐次計算機を対象とし，低レベルでのコード操作を行う．このため，コンパイラの実装は計算機に依存した複雑なものになる．また，コード生成はプログラムを実行しているプロセッサと同じプロセッサで行うため，コード生成時間が計算時間に影響し，複雑な最適化はできない．また，C 言語などを拡張した言語を用いることで，元の言語との互換性を失っているものも多い．

これに対し，我々のシステムは並列計算機を対象とし，コンパイルはプログラムを実行するプロセッサと別のプロセッサで並列に行う．これにより，コンパイル時間は実行時間に上乗せされない．また，我々のシステムでは実行時情報に基づいた最適化をソースコードレベルで行っているため，後段のコンパイラには任意のものを利用できる．さらに，再コンパイルの対象を指示文で指定することで，元の言語との互換性を保ちながらシステムを利用することができる．

本論文の構成は以下のとおりである．まず 2 章で本研究で対象にする最適化について説明する．次に 3 章で本研究で構築した実行時再コンパイルシステムについて説明する．次に 4 章で予備評価について述べた後，5 章で関連研究との比較を行い，最後に 6 章でまとめと今後の課題について述べる．

2. 実行時再コンパイルによる最適化

実行時再コンパイルによる最適化の対象になるのは，時間発展ループ内のルーチンのように，プログラム中で複数回実行されるルーチンである．繰返しの初期で実行時情報を取得し，その情報を用いて再コンパイルを行う．後の繰返しでは最適化されたルーチンを実行する．

本実装で対象にした実行時情報は，スカラ変数の値と残りメモリ容量である．以下にこれらの情報を利用してどのような最適化が可能になるかについて述べる．

2.1 スカラ変数値

実行時にスカラ変数値が変化しない場合（以後，このような変数を実行時定数と呼ぶ），その変数値に特化した最適化を施すことが可能になる．

たとえば，

```
do i = 1, 100
   a(i) = b(i) * c
enddo
```

というコードを考える．このコードを含むサブルーチンが再コンパイルの対象とされ，変数 c が実行時定数であると指定されると，コンパイル時に変数 c の情報を収集するサブルーチンが挿入される．

このようにして得られた変数 c の値が実行時に 0 であることが確認されれば，その値を用い，コードを以下のように変形，コンパイルすることが可能である：

```
do i = 1, 100
   a(i) = b(i) * 0
enddo
```

ここで，`a(i) = b(i) * 0` はさらに `a(i) = 0` と最適化される．

実行中のプログラムは最適化されたオブジェクトコードをロードし，以後同じサブルーチンの呼び出しがあった場合，再コンパイル後のサブルーチンを実行する．これにより，配列 `b(i)` のロードや乗算が削除され，プログラムの実行速度を飛躍的に向上させることができる．

また，次のような例を考える：

```
!HPF$ distribute a(block)
      do i = 1, 50
         a(i*2) = a(i*2-k)
      enddo
```

このプログラムの場合，k の値が奇数であればループをまたぐ依存関係がないため並列化可能である．しかし，コンパイル時に k の値が分からない場合，たとえ実行時に k の値がつねにある奇数であっても並列化できない．このような場合も先ほどと同様に実行時に k の値を取得し，サブルーチンを変形，再コンパイルすることにより，並列化したサブルーチンを実行できるようになる．

また，このプログラムのように HPF で記述されていた場合は，並列化にともない，通信に関しても最適化が行われる．すなわち，再コンパイル前は逐次実行を行うため，ループ前で配列 a を全プロセッサで複製して持つ必要があったが，再コンパイル後は並列化されているため，幅 k のシフト通信を行うだけでよい．

また，次のような例を考える：

```
do j = 1, n
  do i = 1, m
     ... a(i,j) ...
  enddo
enddo
```

ベクトル計算機においては，できるだけ長いループでベクトル化することが，ベクトル長を長くするためには望ましい．このため，外側のループの方がループ長が長ければ，(依存関係が許す限り) 内側と外側のループを入れ換える☆．

しかし，この例の場合，ループ長が変数で与えられているため，コンパイラにはどちらのループ長が長いか判断できず，もし実行時に m が非常に小さい値になるとしても，ループ交換を行わない．

このような場合，実行時に n，m の値を定数に置き換えて再コンパイルすることで，適切にループ交換を行うことが可能になる．

このほかにも実行時定数を用いた最適化として，

- ループ融合，ループ一重化，メモリアクセスの連続化など，他のループ変形，
- 定数伝搬，畳み込みや IF 文の除去，
- 通信生成の最適化（配列サイズやアクセスパターンが不明で，コンパイル時に整列関係が不明な場合など），

などがあげられる．

☆ ループの入れ換えによってメモリアクセスが不連続となる場合があるが，ベクトル計算機は多数のバンクからなるメモリシステムを持つため，(アクセスするメモリアドレスが 2 の冪乗飛びでない限り) メモリからベクトルレジスタへのロードは高速に行うことができる．

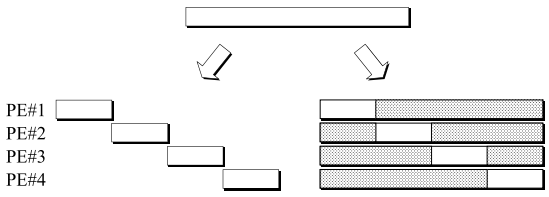

図 1 shrunk 方式と noshrunk 方式
Fig. 1 Shrunk method and noshrunk method.

2.2 残りメモリ容量

HPF コンパイラは，通常 HPF プログラムを SPMD プログラムに変換してコンパイルするように実装される．その際，ソースプログラム中で宣言された配列は，出力する SPMD プログラムでは各プロセッサの担当範囲分だけ割り付けられる．たとえば，

```
      real a(100)
!HPF$ processors pe(10)
!HPF$ distribute a(block) onto pe
```

のような宣言があると，各プロセッサでは配列 a は 10 要素分だけ割り付けられる．これを以後 shrunk 方式と呼ぶ．

shrunk 方式では，配列アクセスを行う際，アドレス変換という作業が必要になる．たとえば，元のプログラムで a(15) = 1 という文があった場合，出力された SPMD プログラムでアクセスすべき要素は，2 番目のプロセッサの 5 番目の要素である．すなわちグローバルアドレスである 15 から，プロセッサ番号 2 のローカルアドレスである 5 を求める作業が必要となる．これをアドレス変換と呼ぶ．アドレス変換はコンパイラが可能な限り最適化するが，完全にそのコストをゼロにすることは難しい．

一方，SPMD プログラムで自プロセッサの担当範囲外のメモリ領域も割り付けてしまう方法も考えられる．この場合，先ほどの例ではすべてのプロセッサが 100 要素分 a を割り付ける．担当範囲以外のメモリ領域は基本的には使用されない（ただし，袖領域として利用できる）．この場合，メモリ使用量は増加するが，アドレス変換のコストがかからないという利点がある．先ほどの例では，担当範囲を持つプロセッサが，a(15) = 1 を実行すればよい．これを以後 noshrunk 方式と呼ぶ．

shrunk 方式と noshrunk 方式のメモリ割付けの様子を**図 1** に示す．noshrunk 方式では，白で表した部分が担当領域で，灰色で表した部分は基本的に使用されない．noshrunk 方式は，HPF/JA 仕様[9]で定義された「フル SHADOW」指定を配列に対して行うこと

に相当する．

ここで，配列ごとにshrunk方式，noshrunk方式を選択可能であるとする．計算機を占有して利用できるような場合，利用できるメモリ容量の範囲で可能な限り多くの配列をnoshrunk方式で割り付けることにより，できるだけ実行を高速化することが望ましい．しかし，利用可能なメモリ容量は実行する環境によって異なるうえ，shrunk方式で割り付けられた配列があればプロセッサ台数によっても変化する．したがって，どの配列をnoshrunk方式で割り付ければよいか，コンパイル時に最適な判断を下すことは不可能である．

ここで，実行時に残りメモリ容量がどれだけあるかの情報が取得できれば，それを元に可能な限り配列をnoshrunk方式で割り付け，再コンパイルすることにより，実行速度を高速化することが可能となる．この方法の詳細については，文献7)に我々の初期の提案が示されている．

3. 実行時再コンパイルシステム

本章では，本研究で実装した実行時再コンパイルシステムにおいて，指示文，システム構成について詳細に述べる．

3.1 指 示 文

どの部分を再コンパイルの対象にするか，どの最適化を利用するかは，現在指示文で指定する方式をとっている．これにより，ユーザが明示的に再コンパイルを制御することが可能になっている．また，将来的には指示文を自動的に挿入することにより，指示文挿入の手間を省くことも考えられる．

また，今回の実装ではサブルーチンを再コンパイルの単位としている．文で構成されるブロックを再コンパイルの対象とする場合，それを内部的にサブルーチン呼び出しに変形することで対応可能である．

指示文は，再コンパイルの対象となるサブルーチンを指定するもの（呼び出し側）と，再コンパイルによる最適化の指定を行うもの（呼び出される側）の2種類ある．

3.1.1 呼び出し側

呼び出し側の指示文では，再コンパイルの対象となるサブルーチンの指定と，再コンパイルを起動するタイミングの指定を行う．たとえば，

```
      do iter = 1, 100
!DYN$ recompile,trigger(iter .eq. 10)
        call foo(a,b,c)
      enddo
```

のように指定する．ここで，!DYN$で始まる行は再コンパイル用指示文であることを表す．!DYN$ recompileを指定することで，その下のサブルーチン呼び出し（この場合はfoo）が再コンパイルの対象であること，また，trigger(iter .eq. 10)を指定することで，変数iterが10のときに再コンパイルを起動することを表す．trigger節の中には任意の論理式が記述可能である．

現在は再コンパイルによって置き換えられるのはプログラムの実行を通じて1回のみである．複数回の置き換えにも容易に対応できるが，その場合は指示文の拡張と実行時システムの改造が必要になる．

再コンパイルの対象となったサブルーチン呼び出しは，コンパイル時にライブラリルーチンに置き換えられる．このライブラリルーチン内で，再コンパイル要求や，再コンパイルしたオブジェクトのロード，実行などを行う．

3.1.2 呼び出される側

呼び出される側の指示文では，再コンパイルによってどのように最適化を行うかを指定する．まず，実行時定数を利用する場合は以下のように指定する．

```
      subroutine foo(a,b,c)
      real a(100),b(100),c,d
      common /bar/ d
!DYN$ runtime_constant(c) begin
      do i = 1, 100
         a(i) = b(i) * c * d
      enddo
!DYN$ end runtime_constant
      end
```

このように指定することで，再コンパイル時にbeginからendまでの変数cの出現が定数に置き換えられる．この指示文は入れ子にすることもでき，また1つの指示文内に複数の変数を指定することも可能である．ただし，指示文で囲まれた範囲内で対象となる変数が書き換えられないことは，ユーザが保証しなければならない．

変数の値が再コンパイル後に変更されることも考えられる．そのような場合でも正しく動作することを保証するため，再コンパイル時にはcの値が想定した値であるかどうかをチェックする条件分岐が挿入される．想定した以外の値が来た場合は，変形前のコードが実行される☆．

☆ ただし，runtime_constantの部分をruntime_constant_noguardと指定することにより，この条件分岐によるガードを削除することができる．すなわち，この場合はプログラマにより，対象の変数は一度決定されると二度と変更されないと表明されたことになる．

次に，メモリ割付けの最適化を行う場合は，以下のように指定する．

```
      subroutine foo(a)
      real a(100), b(100)
!HPF$ distribute (block) :: a,b
!DYN$ candidates_of_fullshadow(b,a)
      ... a(i) ...
      end
```

この指示文はサブルーチンの変数宣言部に挿入する．この場合は配列 b，a がこの順で noshrunk 方式でのメモリ割付けの候補となる（ただし，再コンパイル前は，明示的に指定されたもの以外はすべての配列を shrunk 方式で割り付けるものとする）．実行時には残りメモリ容量と，a，b のサイズを取得する．再コンパイル時には，配列を指定された順に noshrunk 方式で割り付けていく．メモリ容量が不足して noshrunk 方式にすることができなければ，次の順位の配列を noshrunk 方式で割付け可能か調べる．これをすべての候補配列について調べる．

候補配列が局所配列の場合，単純にその配列を noshrunk 方式で割り付ける．候補配列が引数や COMMON で宣言された配列の場合，別に noshrunk 方式で配列を割り付け，その配列にコピーして計算を行い，サブルーチン終了時に書き戻す．この場合は，コピーのオーバヘッドがかかるため，最適化に寄与するかどうかはユーザが判断して指示文に記述する必要がある．

また，メモリ割付けの最適化を行う場合，再コンパイル対象のサブルーチンからのサブルーチン呼び出しは基本的に禁止する．これは，その中から呼び出されるサブルーチン内でさらにメモリ割付けが行われると，メモリが不足してしまう可能性があるためである．

3.2 システム構成

次に，以上に述べたような指示文を含むプログラムを解釈，実行するシステムについて説明する．これは実行前のコンパイルシステムと，実行時に情報収集，再コンパイルするシステムに分けられる．

3.2.1 実行前コンパイルシステム

実行前のコンパイルでは，指示文を解釈し，必要な変形を行って実行可能プログラムを作成する．本研究では，実行時再コンパイル用の指示文を解釈する部分は，通常の HPF コンパイラに対するプリプロセッサとして実装している．構成を**図 2** に示す．

指示文解釈部の動作を例を用いて説明する．

まず，呼び出し側の指示文に対する処理について説明する．たとえば，3.1.1 項であげた例の場合，指示文解釈部は`!DYN$ recompile,...` を見つけると，そ

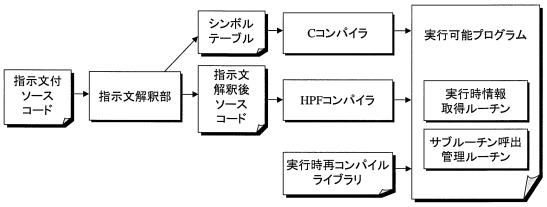

図 2 実行前コンパイル
Fig. 2 Pre-execution compilation.

の下のサブルーチン呼び出し foo を以下のようなライブラリルーチン呼び出しに置換する．

```
      trg = i .eq. 10
      call dcaller(foo,foo_info,foo_new,
     &             trg,state,3,a,b,c)
```

ライブラリルーチンである dcaller の引数には，サブルーチン foo のほか，実行時情報を取得するサブルーチン`foo_info`，再コンパイル後のサブルーチンへのポインタを保持する`foo_new`，trigger 部の論理値，dcaller 中で利用する再コンパイルの状態 state，サブルーチンの引数の数，サブルーチンへの引数が渡される．ここで，状態を保存するため，`foo_new`，state は SAVE 属性を持った変数として宣言されている．内部の動作は 3.2.2 項で説明する．

次に呼び出される側の指示文について説明する．

まず実行時定数を用いる指示文について，3.1.2 項の最初の例を用いて説明する．指示文解釈部は，このようなプログラムを見つけると，以下のように変形した`foo_info` という名前の別のサブルーチン定義を生成する．

```
      subroutine foo_info(a,b,c)
      real a(100),b(100),c,d
      common /bar/ d
      call send_info(c,id_of_c,...)
      do i = 1, 100
        a(i) = b(i) * c * d
      enddo
      end
```

このサブルーチンでは，begin 指示文があったところに`send_info` というライブラリルーチンを挿入し，指定された変数の値を送出する．`send_info` の引数には，変数の値と，その変数および出現場所を一意に決定するための ID のセットを渡す（これらの通信には，システムを通して TCP/IP を利用している）．

これとは別に，指示文を削除した元のサブルーチン`foo` も作成する．

また，再コンパイル対象のサブルーチン中に COM-

MONブロックやサブルーチン呼び出し，入出力文などが存在した場合，オブジェクトコードのロード時に，そのアドレスへの参照を実行中のプログラム中にあるアドレスに書き換える必要がある．このため，再コンパイル対象のサブルーチン中にこのようなシンボル参照があった場合，シンボル名とそのアドレスをテーブルに書き出しておき，実行可能プログラムとリンクすることで，ロード時にアドレスを検索できるようにする．

たとえば，このサブルーチンでは bar というCOMMONブロックを用いている．この場合，

```
extern char bar;
struct symtab_ {char *name; char *ptr;};
struct symtab_ symtab[] = {{"bar",&bar}};
```

のようなテーブルを作成する．すなわち，"bar"というシンボル名から bar のアドレスが検索できるようなテーブルを作成する．

残りメモリ容量を取得する場合も同様である．こちらはサブルーチンに入ったところで残りメモリ容量と，noshrunk 候補の配列の宣言サイズと，各プロセッサで割り付けられているサイズを通知するルーチンを挿入する．その他は実行時定数の場合と同じである．

実行可能プログラムは，シンボルテーブル，指示文解釈後の HPF プログラム，その内部から呼ばれる dcaller，send_info などのライブラリルーチンをリンクして作成する．

3.2.2 実行時システム

実行時システムの構成を**図 3** に示す．ここで，実行可能プログラム中の「実行時情報取得ルーチン」は前項の send_info などに相当し，「サブルーチン呼び出し管理ルーチン」は前項の dcaller に相当する．

実行可能プログラムは並列計算機上で実行される．また，実行時再コンパイル管理部と実行時再コンパイル部は別計算機（または並列計算機上の別プロセッサ）で実行される．これにより，再コンパイルにかかわるオーバヘッドが実行時間に上乗せされないという利点がある．

再コンパイルのためにプロセッサを 1 台割り当てることにより，計算に使えるプロセッサが減るという欠点があるが，プロセッサ台数が十分大きい場合は無視できる．たとえば，プロセッサ数 100 台の並列計算機を用いるのであれば，そのうちの 1 台を再コンパイル用に使っても 1%の速度低下でしかない．

以下，各部の動作について説明する．

まず，サブルーチン呼び出し管理ルーチンは以下のように動作する：

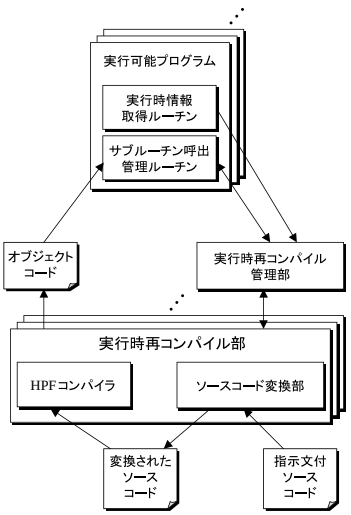

図 3 実行時システム
Fig. 3 Runtime system.

(1) 現在の再コンパイル状態（dcaller の引数では state に相当，以下単に「状態」）を調べる（初期値は「再コンパイル前」）．

(2) 「再コンパイル前」の場合，trigger 部の論理値を調べる．

(a) 偽なら実行時情報を送出するサブルーチン（先の例ではfoo_info）を実行して終了する．

(b) 真なら実行時情報を送出するサブルーチンを実行後，再コンパイル要求を実行時再コンパイル管理部に送る．また，状態を「再コンパイル中」に変更する．

(3) 「再コンパイル中」の場合，実行時再コンパイル管理部に再コンパイルが終了したかを尋ねる．

(a) 再コンパイル成功の場合，状態を「再コンパイル済」に変更し，オブジェクトコードをロードする．この際，前項で述べたシンボル名の解決を各 PE で行い，ロードしたサブルーチンへのポインタを（先の例ではfoo_new に）保存し，実行，終了する．

(b) 再コンパイル失敗の場合，状態を「再コンパイル失敗」に変更し，通常のサブルーチン（先の例では foo）を実行し終了する．

(c) まだ再コンパイル中の場合，通常のサブルーチンを実行し終了する．

(4) 「再コンパイル済」の場合，以前ロードした再コンパイル済みのサブルーチン（foo_new）を実行して終了する．

(5) 「再コンパイル失敗」の場合，通常のサブルーチンを実行し終了する．

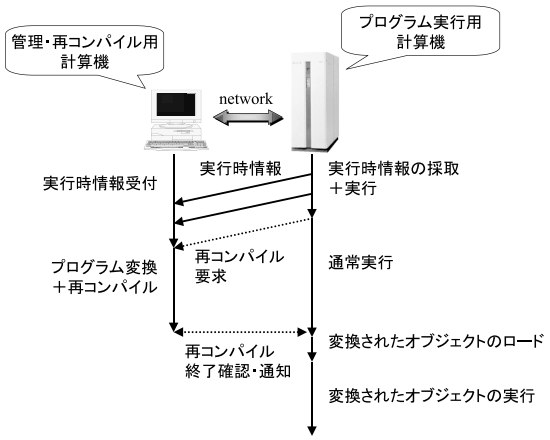

図 4 実行時再コンパイルの様子
Fig. 4 Execution using dynamic recompilation technique.

この実行の様子を**図 4** に示す．図の「再コンパイル要求」は上記 (2)(b) で,「再コンパイル終了確認・通知」は上記 (3) で行われる．このように動作することで，再コンパイルとサブルーチンの実行を並列に行うことができる．

ただし，再コンパイルによって通信にかかわる変更が行われる可能性があるため，オブジェクトのロードは全プロセッサで同時に行う必要がある．このため，上述の条件判定は全プロセッサで同じになるよう，trigger 部の論理値は 1 プロセッサの値を代表値とし，実行時再コンパイル管理部との通信は 1 プロセッサが代表して行う．これらの値は全プロセッサにブロードキャストされ，それにより全プロセッサで再コンパイル状態の更新やオブジェクトのロードが同期して行われる．

実行時再コンパイル管理部は，実行時情報取得ルーチンから実行時情報を受け取り，その情報を保存する．再コンパイルの要求があった場合，その情報を元に再コンパイルする価値があるかどうかを判断する．再コンパイルしても速度向上が見込めないようであれば，「再コンパイル失敗」と判断する．

再コンパイルを行う場合，実行時再コンパイル部を起動する（複数のサブルーチンを再コンパイルする場合は，同時に複数の実行時再コンパイル部が起動される場合もありうる）．この部分は実行時情報に基づきソースコードを変形する部分と，変形したソースコードをコンパイルする部分に分けられる．

実行時定数を利用する場合は，実行時情報に基づき，該当変数の最頻値が渡される☆．その値を元に，指示文で指定された部分の変数出現を定数に置き換える．またここで,（指示文が `noguard` 付きでなければ）条件分岐によるガードを挿入する．たとえば，3.1.2 項の例の場合，`c` が 0 であるという条件で再コンパイルを行うと

```
if(c .eq. 0) then
  do i = 1, 100
     a(i) = b(i) * 0 * d
  enddo
else
  do i = 1, 100
     a(i) = b(i) * c * d
  enddo
endif
```

のようになる．`a(i) = b(i) * 0 * d` は後段のコンパイラで `a(i) = 0` に最適化される．

メモリ割付けの最適化を行う場合は，残りメモリ量と候補配列のサイズを元に，どの配列を noshrunk 方式にするか決定する．引数や COMMON ブロックの配列を noshrunk 方式にすると判断した場合，noshrunk 方式で配列を割り付けた後，その配列に内容をコピーして計算を行い，サブルーチンから抜ける前に計算結果を書き戻す．たとえば，3.1.2 項の 2 番目の例の場合，`b`, `a` ともに noshrunk 方式で割り付けるとすると，

```
      subroutine foo(a)
      real a(100), b(100), new_a(100)
!HPF$ distribute (block) :: a,b,new_a
!HPX$ noshrunk :: new_a,b
      copy_to_noshrunk(new_a,a)
      ... new_a(i) ...
      copy_to_shrunk(a,new_a)
      end
```

のように変形される．`!HPX$ noshrunk` は noshrunk 方式でメモリを割り付ける指示文である．また，`copy_to_noshrunk`，`copy_to_shrunk` で，配列の内容をコピーする．

以上のように変形されたあと，HPF コンパイラによってコンパイルされ，オブジェクトコードが生成される．サブルーチン呼び出し管理ルーチンはこのオブジェクトコードをロードする．オブジェクトのロードには，基本的に GNU BFD ライブラリを利用し，直接オブジェクトファイルを操作することで実現した．BFD ライブラリを用いることで，さまざまなオブジェクト

☆ 現在の実装では最頻値を用いているが，複数の値に対応したコードを生成する，あるいは値がある範囲にあるかどうかを確かめるコードを生成するといった拡張も考えられる．

ファイルフォーマットに対して高い移植性を持つ☆．ただし，後述するSX-4システム上での実装では，BFDライブラリが利用できなかったため，COFF形式のオブジェクトファイルを直接操作することで実現した．

オブジェクトファイル中でCOMMONブロックなどが用いられている場合，その名前がリロケーションエントリ中に未解決のシンボルとして現れる．この場合，コンパイル時に作成したテーブルを利用して，オブジェクトファイル中のアドレス参照を，プログラム中のアドレスに書き換える．

4. 評　　価

本章では，我々が実装を行ったシステムのうち，並列コンピュータCenju-4[4]，およびベクトルスーパコンピュータSX-4[10]上での評価結果について述べる．

4.1 Cenju-4上での評価

4.1.1 評価環境

実行可能プログラムはCenju-4の各PEで，実行時再コンパイル管理部および実行時再コンパイル部に関しては，Cenju-4のフロントエンドプロセッサである，ワークステーションEWS4800上で実行した．

HPFコンパイラは我々がRWCプロジェクトで開発中のものを，Cコンパイラおよび HPFコンパイラのバックエンドである Fortranコンパイラは，システムに標準の Cコンパイラおよび f 90コンパイラを用いた．

評価対象にしたプログラムは以下の3つである．

mul 配列を定数倍するプログラム．再コンパイルにより強さの低減が行われる．2.1節の最初の例に相当する．300回サブルーチンが呼び出される．

cpy 配列内でコピーを行うプログラム．静的には並列化不可能だが，再コンパイルにより並列化可能になる．2.1節の2番目の例に相当する．1,000回サブルーチンが呼び出される．

sp NAS Parallel Benchmarksに収められたプログラムの1つ（第1版）．サイズはCLASS Aを用いた．残りメモリ容量情報を用いた最適化の評価に用いた．再コンパイル対象のサブルーチンは，xisweep，etasweep，ztasweep，rhsであり，内部で一時変数として使われる配列すべてと，引数に渡される配列のうち頻繁にアクセスされる配列をnoshrunk方式での割付け候補として指示文に指定した．これらのサブルーチンは400回呼び出される．

4.1.2 基本性能

まずmulを用い，実行時再コンパイルの基本性能として，オブジェクトをロードするためにかかる時間と実行時情報を取得するためのオーバヘッドを計測した．これらは対象とするルーチンの前後に`MPI_Wtime()`を挿入することで計測した．ただし，実行時情報を取得するルーチンに関しては1回の実行時間が短いため，複数回での実行時間の総和を実行回数で割ることで計測した．mulではロードしたオブジェクトのサイズは約4 KByteである．

オブジェクトをロードするのにかかった時間は，1台の場合は70 msecであったが，台数が増えるごとに増加し，32台では981 msecであった．これは，オブジェクトのロードが各PEで同時に行われるため，フロントエンドプロセッサとのネットワークやディスクが同時に使われることが理由であると考えられる．ファイルのロードを1台のプロセッサで行い，その後Cenju-4の内部ネットワーク経由でコピーするような実装にすれば高速化は可能であると考えられる．

また，1つの変数値を1回送出するのにかかるコストは約0.7 msecであった．プログラム実行中での実行時情報送出コストを抑えるためには，実行時情報送出回数を減らす必要がある．このためには，`!DYN$ recompile,trigger (...)`指示文の`trigger`節で，適切な論理値を指定する．たとえば，3.1.1項の例では，`iter .eq. 10`となっている．これにより，変数`iter`が`10`のときに再コンパイルが起動されるため，実行時情報が送出される回数は，サブルーチン内で送出される実行時情報の数×10回となる．このように，実行時情報の送出が十分少ない回数の場合，このオーバヘッドが実行時間に占める割合は十分小さいと考えられる．

4.1.3 実行速度

mul，cpy，spの評価をそれぞれ**表1**，**2**，**3**に示す．「再コンパイルなし」，「再コンパイルあり」，「理想値」はそれぞれの場合の実行時間を秒で表す．ただし，「理想値」とは，mul，cpyでは，元のプログラムでは実行時にしか得られない情報をソースコード上に定数として指定したプログラムの実行時間である．spでは，すべての配列をnoshrunk方式で割り付けた場合の実行時間を示す．「再コンパイルなし」および「理想値」では，実行時再コンパイルシステムを用いず，通常のHPFプログラムとしてコンパイル，実行した．

「ロードタイミング」とは，何度目の繰返しですべ

☆ このほかにもOSに用意されている動的リンク機構（`dlopen`, `dlsym`）を用いる方法もある．しかし，今回実装したプラットフォームでは提供されていなかったことなどから利用しなかった．

表 1 mul の評価
Table 1 Evaluation of mul.

PE 台数	1	2	4	8	16	32
再コンパイルなし	248	123	62.2	31.0	15.5	7.8
再コンパイルあり	88.9	49.1	22.9	12.0	7.3	4.5
理想値	87.2	43.6	21.7	10.9	5.3	2.2
ロードタイミング	3	6	9	15	26	38
速度向上率	2.8	2.5	2.7	2.6	2.1	1.7
理想値との比	0.98	0.89	0.95	0.91	0.78	0.49

表 2 cpy の評価
Table 2 Evaluation of cpy.

PE 台数	1	2	4	8	16	32
再コンパイルなし	29.7	85.1	123	167	211	272
再コンパイルあり	29.5	15.0	6.8	5.5	3.3	3.4
理想値	29.3	13.8	5.8	2.4	1.1	0.4
ロードタイミング	45	18	12	9	8	7
速度向上率	1.0	5.7	18.1	30.0	63.9	80.0
理想値との比	0.99	0.92	0.85	0.44	0.33	0.12

表 3 sp の評価
Table 3 Evaluation of sp.

PE 台数	1	2	4	8	16	32
再コンパイルなし	8821	4584	2438	1282	701	432
再コンパイルあり	5476	2928	1586	861	484	333
理想値	2521	1439	807	441	252	185
ロードタイミング	6	9	15	28	48	74
速度向上率	1.6	1.6	1.5	1.5	1.4	1.3
理想値との比	0.46	0.49	0.51	0.51	0.52	0.56

てのサブルーチンについて再コンパイル後のオブジェクトがロードされたかを表す．

「速度向上率」は再コンパイルありの場合の再コンパイルなしの場合に対する速度向上率を,「理想値との比」は，再コンパイルありの場合の理想値との比をそれぞれ表す．

まず，mul は実行時再コンパイルにより，1.7〜2.8 倍の速度向上率を示す．理想値に対する比率はプロセッサ台数が増えると低下し，1〜0.5 程度となっている．この理由は，プロセッサ台数が増えることにより実行速度が向上し，再コンパイル中に多くの計算が進行することで，再コンパイル後のオブジェクトを実行する機会が減少したことによる．

cpy は 32 台で 80 倍ときわめて高い速度向上率を示す．これは，元のプログラムでは並列化が不可能であったため分散配置された配列が全プロセッサにコピーされ，プロセッサ台数が増えるごとに速度が低下していたためである．

ロードタイミングはプロセッサ台数が増えるごとに早まっている．これは，オブジェクトをロードするまでの時間は，再コンパイルにかかる時間に依存するためほとんど変化しないのに対し，この期間での実行速度は，再コンパイル前のため，台数が増えるごとに低下していたためである．台数が増えると同じ時間で実行できる繰返し数が減るため，ロードタイミングの値が小さくなる．

オブジェクトをロードするまでの時間は台数が増えても変わらないのに対し，理想値の実行時間は全体が台数が増えるごとに短くなる．これにより，台数が増えた場合，理想値に対する比率は，mul と同様プロセッサ台数が増えるごとに低下する．

sp ではメモリ容量に余裕があったため，再コンパイル時に，すべてのプロセッサ台数で，すべての候補配列が noshrunk 方式で割り付けられた．sp は 1.3 から 1.6 倍程度の速度向上を示している．理想値に対する比率は 0.5 程度と低いが，これは，引数の配列に関してはコピーのオーバヘッドがあること，また，そのためすべての配列を noshrunk 方式での割付け候補にしたわけではないことがあげられる．

また，元のプログラムには，再コンパイル対象のサブルーチン以外にも，計算を行っている部分が残されている．これも理想値に対する比率を下げる原因の 1 つであると考えられる．

4.2 SX-4上での評価

4.2.1 評価環境

実行可能プログラムはSX-4上で，実行時再コンパイル管理部および実行時再コンパイル部に関しては，Linuxが動作しているPC上で実行した．評価に用いたSX-4は2CPUの共有メモリモデルであり，ベクトルレジスタ長は256である．PCのCPUはPentium III 550MHzであり，Red Hat Linux 6.1が動作している．

用いたHPFコンパイラはCenju-4での評価で用いたものと同じものである．Cコンパイラおよび HPFコンパイラのバックエンドであるFortranコンパイラは，システムに標準のものを用いた．ただし，実行時再コンパイル用にはPC上で動作するクロスコンパイラを用いた．

評価対象にしたプログラムは，APR社が公開しているベンチマークプログラムであるshallowを用いた．このプログラムは，2次元のShallow Water方程式を数値計算で解くものである．

本評価ではプログラムを変更し，配列を分散しないようにした．すなわちCPU1台のみを用いた逐次実行での評価である．また，元のプログラムでは問題サイズをプログラム中で固定していたが，これをファイルから読み込むことで，実行時に変更できるようにした．元のプログラムでは静的に解析が可能であったが，問題サイズを実行時に可変にすることで，プログラムの有用性が増す．評価に用いた問題サイズは64×4096であり，ループの繰返しを1,000回とした．ここで，問題サイズはシミュレーションにおける場のサイズとなる．

問題サイズを64×4096とすることで，プログラム中の多くのループが，内側64回，外側4096回繰り返す2重ループとなる．実行時再コンパイルでは，このループ長を実行時定数として指定した．これにより，この2重ループが交換され，ベクトル長が長くなることが期待される．

再コンパイル対象のサブルーチンはcalc1，calc2，calc3とした．

4.2.2 実行速度

評価結果を**表4**に示す．それぞれの項目の意味は，Cenju-4での評価の場合と同じである．「理想値」のプログラムでは，問題サイズをparameter文で指定した．

実行時再コンパイルにより，1.7倍の速度向上が見られた．これは，ループ交換によって，ベクトル長が長くなったためであると考えられる．

表4 shallowの評価
Table 4 Evaluation of shallow.

再コンパイルなし	35.4
再コンパイルあり	20.8
理想値	18.9
ロードタイミング	99
速度向上率	1.7
理想値との比	0.91

SX-4では，実行時に環境変数を設定することで，プログラムの実行におけるさまざまなプロファイルデータを得ることができる．この設定により，実行されたベクトル命令の平均ベクトル長を調べたところ，再コンパイルなしの場合の平均ベクトル長は64.1，再コンパイルありの場合は198.7，理想値の場合は255.9であった．これからも，再コンパイルにより平均ベクトル長が長くなっていることが確認できる．

理想値との比は0.91である．理想値と再コンパイルを行ったときの実行時間の差は，ロードタイミングに依存すると考えられる．本評価では，ロードタイミングが99であるため，99回目の繰返しまでは再コンパイル前の状態で実行されている．再コンパイルなしの実行時間35.4秒で，理想値の実行時間が18.9秒であるから，これが99回目までの繰返しで切り替わると考えると，$35.4 \times 99/1000 + 18.9 \times (1000 - 99)/1000 = 20.5$秒となる．これにオブジェクトのロードなどのオーバヘッドを考えると，測定値の20.8秒に近い値となる．

本評価では配列を分散しなかった．これは，現在の実行時再コンパイルの実装では，HPFプログラムレベルで定数への置換を行っており，並列化によって変形されるループの開始値，終了値を定数に置換できないためである．たとえば，

```
do i = n, m
   ...
enddo
```

のようなループがあった場合，並列化が行われると，たとえば以下のように変形される☆：

```
call calc_loop_bound(n,m,lb,ub,...)
do i = lb, ub
   ...
enddo
```

lb，ubはそのプロセッサでの計算範囲を表す．これらはn，m，プロセッサ数と実行するプロセッサ番号，および配列の分散状況などから実行時に計算される．こ

☆ ここで示した変形は概念的なもので，実際の変形とは異なる．

のようにプログラムが変換されたあと，後段の Fortran コンパイラへ渡され，コンパイルされる．

したがって，HPF プログラムレベルでループの開始値 n，終了値 m を定数に置換しても，後段の Fortran コンパイラへの入力の段階では，変数 lb，ub に変換されている．このため，Fortran コンパイラレベルでループ交換のような最適化を行うことができない．

これを可能にすることは，今後の課題である．これには，たとえばループ交換のような最適化を上述のようなプログラムの変換前に行うなどの方法が考えられる．

5. 関 連 研 究

実行時の情報を用いた最適化には，実行時再コンパイルのほか，マルチバージョン化と呼ばれる方法もある．これは，実行時の値によって最適化方法が変わるとき，特定の値専用のコードを作成しておき，その値が来た場合には条件分岐して専用のコードを実行するものである．この手法は実行時再コンパイルとは異なり，実行時システムや実行時のコンパイルが不要であるという利点がある．しかし，最適化方法が複数ある場合など組合せが爆発する場合には適用できない．さらに，コンパイラがすべての最適化手法に対して個別にマルチバージョン化に対応する必要があるため，コンパイラの実装コストが高い．実行時再コンパイルの場合は，後段のコンパイラは任意のものを用いることができるため，実装は容易である．

実行時にコード生成を行う研究は多く行われている．Java 言語の各種 JIT コンパイラが著名であり，たとえば文献 1) のようなものがある．しかし，これらの多くは実行時情報として実行頻度情報だけを用い，コストのかかる最適化を JIT コンパイラで行う価値があるかどうかを判断するにとどまる．

実行時の変数値情報を利用して最適化を行うシステムとしては，他に ‘C[5]，DyC[2]，Tempo[3] などがある．これらの研究では，低レベルのコード操作を行うため，コンパイラの実装は計算機に依存した複雑なものになる．また，これらは逐次計算機での実行を仮定し，プログラムの実行と再コンパイルを同一のプロセッサ上で実行する．したがって，いかにコード生成時間を短くし，最適化後の実行の高速化でコード生成時間を埋め合わせるかが研究のポイントになっており，コストの高い最適化は対象にされていない．我々の研究では，再コンパイルは別プロセッサで行われるため，再コンパイル時間の実行時間に対する影響は小さい．また，これら従来の研究で行われているコード生成を，本研究と同様に並列に行うことも考えられる．

我々は指示文を用いて再コンパイルを行う部分を指定したが，‘C では，実行時に生成するコードをプログラマが明示的にプログラム中で作成する．DyC では C 言語にアノテーションを付加することでコード生成を行う部分を指定している．Tempo では，対象とする部分をコンフィギュレーションファイルで指定する．

文献 6) では，我々と同様，別プロセッサで再コンパイルを行う手法が提案されているが，手動で変形したソースを使って評価されており，システムとしてはまだ実装されていない．また，計算を並列に行うこともまだ考慮されていない．

6. お わ り に

本研究では，実行時に得られる情報を元に，プログラムの再コンパイル，最適化を行うシステムを実装し，予備評価を行った．実行時情報としてはスカラ変数値，残りメモリ容量を用いた．予備評価により，再コンパイルのオーバヘッドは十分小さく，再コンパイルにより大きな速度向上が得られることを確認した．

本研究では，合計 4 本のプログラムを用いて評価を行った．今後の課題として，より多くの実アプリケーションでの評価を行うことにより，実行時再コンパイルの有用性の大きさを示すことがあげられる．これには，ほかに実行時再コンパイルによる最適化が適用可能な場合を調査する，あるいは新たな実行時情報を用いた最適化をシステムに組み込むといったことも含まれる．

さらに，本論文では，Cenju-4 および SX-4 を用いて評価を行った．このほかにも，現在 Linux が動作する PC クラスタでの実装を完了している．このような他の計算機システム上での評価や，HPF 以外の他のプログラミング言語における実装，評価も今後の課題にあげられる．

特に，我々は複数の種類の計算機を結合して 1 つの並列計算機システムとして用いる異機種並列分散システムについても研究を行っている[8]．現在，Cenju-4 と Linux が動作する PC クラスタを結合した異機種並列分散システムにおいても，実行時再コンパイルシステムの実装を完了している．このような異機種並列分散システムにおける実行時再コンパイルの評価や，新たな最適化手法の研究を行うことも，今後の課題にあげられる．

参 考 文 献

1) Arnold, M., Fink, S., Grove, D., Hind, M. and Sweeney, P.F.: Adaptive Optimization in the Jalapeño JVM, *OOPSLA '00*, pp.47–65, ACM (2000).
2) Grant, B., Mock, M., Chambers, C. and Eggers, S.J.: An Evaluation of Staged Run-time Optimizations in DyC, *Programming Language Design and Implementation*, pp.293–304, ACM (1999).
3) Marlet, R.: Tempo Specializer — A Partial Evaluator for C. http://www.irisa.fr/compose/tempo
4) Nakata, T., Kanoh, Y., Tatsukawa, K., Yanagida, S., Nishi, N. and Takayama, H.: Architecture and the Software Environment of Parallel Computer Cenju-4, *NEC RESEARCH & DEVELOPMENT*, Vol.39, No.4, pp.385–390 (1998).
5) Poletto, M., Hsieh, W.C., Engler, D.R. and Kaashoek, M.F.: 'C and tcc: A Language and Compiler for Dynamic Code Generation, *ACM Trans. Programming Languages and Systems*, Vol.21, No.2, pp.324–369 (1999).
6) Voss, M.J. and Eigenmann, R.: A Framework for Remote Dynamic Program Optimization, *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pp.32–40, ACM (2000).
7) 村井 均，荒木拓也，松浦健一郎，末広謙二，妹尾義樹：実行時再コンパイルによる並列プログラムのメモリ割り付け最適化，情報処理学会研究報告，99-HPC-79, Vol.99, No.103, pp.61–66 (1999).
8) 松浦健一郎，荒木拓也，蒲池恒彦，妹尾義樹：Webインタフェースを備えた異機種並列分散プログラミング環境，情報処理学会研究報告，2001-HPC-87, Vol.2001, No.77, pp.141–146 (2001).
9) 財団法人高度情報科学技術研究機構：High Performance Fortran 2.0 公式マニュアル，シュプリンガー・フェアラーク東京 (1999).
10) 井上政信，大和田刻明，古井利幸，片桐 勝：SX-4 シリーズのハードウェア構成，NEC 技報，Vol.48, No.11, pp.13–22 (1995).

(平成13年8月31日受付)
(平成14年2月13日採録)

荒木 拓也（正会員）

1971年生．1994年東京大学工学部電気工学科卒業．1999年同大学院情報工学専攻博士課程修了．工学博士．同年，NEC入社．現在インターネットシステム研究所勤務．プログラミング言語の実装，特に最適化，並列化等に興味を持つ．

村井 均（正会員）

1971年生．1996年京都大学大学院工学研究科情報工学専攻修士課程修了．同年NEC入社．現在，地球シミュレータセンター技術員．並列スーパーコンピュータ向けの言語環境の研究開発に従事．

蒲池 恒彦（正会員）

1964年生．1988年九州大学工学部情報工学科卒業．1990年同大学院総合理工学研究課情報システム学専攻修了．同年NEC入社．現在，インターネットシステム研究所主任．並列計算機アーキテクチャ，自動並列化コンパイラ，並列化支援システム等に興味を持つ．

妹尾 義樹（正会員）

1961年生．1984年京都大学工学部情報工学科卒業．1986年同大学院修士課程修了．同年NEC入社．インターネットシステム研究所にてスーパーコンピュータの研究開発に従事．並列処理アーキテクチャ，分散メモリマシンのための並列化言語環境，並列アルゴリズムに興味を持つ．現在インターネットシステム研究所主任研究員．工学博士．1988年本会論文賞受賞．