Vol.2009-ARC-184 No.7
2009/8/4

自動メモ化プロセッサを支援するプログラム変換手法の提案と実装

加藤　拡[†1]　津邑公暁[†1]　松尾啓志[†1]

専用のハードウェアを用いることにより，過去の関数の実行結果の再利用を実現するモデルとして、自動メモ化プロセッサがある．この自動メモ化プロセッサを対象とした，3 種類のプログラム最適化手法を提案する．1 つ目は，関数の処理の一部を新たに別の関数として定義することにより，より細かい処理単位での計算再利用を可能とする手法である．2 つ目は，関数の入力から不要なアドレス情報を削除し，計算再利用が適用可能な関数を増やす手法である．3 つ目は、入力の値が取る範囲が関数の動作を決定する場合に，値が完全に一致しなかったとしても再利用が可能となるようにする手法である．

上述のようなプログラム変換を行うトランスレータを実装し，汎用 GA ソフトウェアである GENEsYs をベンチマークとして，変換前後の実行サイクル数を自動メモ化プロセッサを用いて比較を行った．その結果，提案手法によって実行サイクル数が平均 4.4%，最大 53.6%削減できた．

A Program Translation Method for Utilizing an Auto-Memoization Processor

Hiromu Kato,[†1] Tomoaki Tsumura[†1] and Hiroshi Matsuo [†1]

We have proposed an auto-memoization processor. This processor automatically and dynamically memoizes functions and skips their execution. This paper proposes three methods to translating programs for utilizing our auto-memoization processor. The methods are cutting out smaller reusable blocks, eliminating unnecessary pointer arguments, and decomposing if-else blocks. We implemented a translator and automated these translation. Through an evaluation with GENEsYs a general purpose GA software, we found that the translation methods increase the performance gain of an auto-memoization processor. Translated GENEsYs programs reduces up to 53.6% execution cycles and 4.4% on average.

1. はじめに

プログラムの実行を高速化する手法として，スーパースケーラのように命令レベル並列性 (Instruction-Level Parallelism：ILP) に着目したものが研究されてきた．しかしながら ILP による高速化には限界があり，命令レベルの並列化を行うだけではプロセッサの性能向上が頭打ちになりつつある．

一方現在では，1 つの CPU に複数のコアを搭載したマルチコア CPU が一般的となってきている．これは，消費電力や発熱量の問題を解決しつつプロセッサあたりの処理能力の向上を実現しているが，一方で複数のコアを用いたプログラムの高速化が問題となる．複数コアを用いたシングルスレッドプログラムの高速化[1] を行うためには，スレッドレベルの並列性に着目してプログラムを複数スレッドに分割し，各スレッドをコアの一つ一つに割り当てる必要がある．しかし，スレッドレベルの並列性には限界があるため，割り当てるコアの数にも限界がある．

これらはいずれも命令実行の並列化という方法で高速化を図るものであるが，一方でそれとはまったく異なる高速化手法として，過去の計算結果を再利用する計算再利用という手法が存在する．計算再利用には，ハードウェアによるものとソフトウェアによるもの，またその双方によるものなど，様々なものが提案されている．このうち，専用のハードウェアを用いることにより計算再利用を実現するモデルとして、我々は自動メモ化プロセッサ[2] を提案している．本稿ではこの自動メモ化プロセッサを対象とした，プログラムの最適化手法を提案する．自動メモ化プロセッサは実行した関数の入出力を表に記録する．そして，再び同関数が呼び出されたときに現在の入力と表に記録された入力とを比較し，一致すれば過去の出力を利用することでプログラムの高速化が実現される．本稿では，この高速化をより多くの場合に適用するために，プログラム変換による高速化を図る手法を提案する．

2. 背　景

自動メモ化プロセッサについて，その高速化手法の方針と動作原理，そしてその問題点を概説する．

†1 名古屋工業大学
Nagoya Institute of Technology

```
1:  int f(int a, int b, int c){
2:      a = a + b;
3:      b = a - b;
4:      a = a * b * c;
5:      return(a);
6:  }
```

```
7:  int main(){
8:      f(4,2,2);
9:      f(4,2,3);
10:     return(0);
11: }
```

図 1 一部のみが再利用可能な関数の例

```
1: int g(int array[], int size){
2:     int i;
3:     int sum = 0;
4:         for(i = 0; i < size; i++)
5:             sum += array[i];
6:     return(sum);
7: }
```

```
8: int main(){
9:     int x[3] = {1,2,3};
10:    int y[3] = {1,2,3};
11:    g(x,3);
12:    g(y,3);
13:    return(0);
14: }
```

図 2 配列の先頭アドレスを入力に取る関数の例

2.1 自動メモ化プロセッサ

メモ化[3] とは，関数等の命令区間に対してその入出力を計算再利用可能な状態で記憶しておく処理である．これにより，次回以降の同一入力による当該関数の実行を省略し，プログラム全体の実行を高速化できる．このメモ化を，ハードウェアを用いて動的に行うプロセッサとして考案されたのが，自動メモ化プロセッサである．

自動メモ化プロセッサがプログラムを実行する際，実行された各関数について入力と出力を表に登録する．ここで入力とは，関数の引数，および局所変数以外への変数参照に伴う主記憶参照を指す．また出力とは，関数の返り値，および大域変数への書き込みを指す．この関数の入出力が登録される表を再利用表と呼ぶ．

プログラム中で関数が呼び出されるとき，自動メモ化プロセッサは再利用表を検索し，呼び出された関数の開始アドレスおよび入力が，過去に実行された関数のものと一致するかを比較する．一致した場合，関数の実行を省略し，記録された出力を書き戻す．一致しなかった場合は関数を通常通り実行し，関数の開始アドレス，入力，出力を再利用表に登録する．計算再利用が成功した場合は関数の実行そのものを省略できる．このような省略を重ねることで，プログラム実行の高速化を図る．

2.2 問　題　点

自動メモ化プロセッサが抱える問題点として，本来再利用可能な区間を再利用不可能と判断してしまう可能性を持つことが挙げられる．以下では，いくつかの再利用不可能となるパターンについて，具体的なプログラム例を用いつつ説明する．

2.2.1 一部のみが再利用可能な関数

1 つ目の問題点は，関数の一部の計算のみを再利用するということができない点である．**図 1** において，関数 `f` は 3 つの整数を引数に取り，第 1 引数と第 2 引数の和，第 1 引数と第 2 引数の差，第 3 引数の 3 つの値の積を返す関数である．関数 `f` が呼び出されるのは，図 1 の 8 行目と 9 行目の 2ヵ所である．最初に 8 行目で `f(4,2,2)` が実行され，次の 9 行目で `f(4,2,3)` が呼び出される．このとき 8 行目とは入力値が異なるため，自動メモ化プロセッサは再利用不能と判断し，関数を通常実行する．ここで，関数内部においてどのような計算が行われているかを考える．`f(4,2,2)` と `f(4,2,3)` において，2 行目と 3 行目では同じ計算が行われる．両者の間で異なる計算が行われるのは 4 行目のみであり，2 行目と 3 行目は同じであるが，関数を単位とするメモ化では，この部分にだけ再利用を適用することはできない．

2.2.2 配列の先頭アドレスを入力に取る関数

2 つ目の問題点は，入力が異なっていたとしても動作に変化が生じず本質的には再利用できる関数を，再利用不可能と判断してしまう点である．この例を**図 2** に示す．ここで関数 `g` は整数型の配列へのポインタ，および配列の大きさを引数に取り，配列のすべての値の合計を返り値として返すものである．`main` 関数では 2 つの配列が宣言されており，順に配列へのポインタを引数として関数 `g` を呼び出している．最初に 11 行目で関数 `g(x,3)` が通常実行され，このとき再利用表には `g(x,3)` の入力が記録される．その後，12 行目で `g(y,3)` が実行される．このとき再利用表を検索しても現在の入力と一致する過去の入力は見付からず，計算再利用は行われない．

関数 `g` の内部では，配列の値のみを参照しており，配列 `x` と `y` に格納されている値はそれぞれ同じであるため，`g(x,3)` と `g(y,3)` で実行される計算は同じである．一方で，関数の引数として与えられているのは配列へのポインタであり，再利用表に登録されている値は異なる．関数内部において配列に格納されている値がメモリ上から読み出されるため，これらもまた再利用表には入力として登録されており，それらの配列の値は登録されている入力と一致はするものの，配列の先頭アドレスが一致しないため，自動メモ化プロセッサは，

```
1:  int h(int a, int b){
2:      if(b > 0)
3:          a = -a;
4:      else
5:          a = a / 3;
6:      a = a * 2;
7:      return(a);
8:  }
```

```
 9:  int main(){
10:      int b = 1;
11:      h(4,b);
12:      b = 2;
13:      h(4,b);
14:  }
```

図 3 入力の取る範囲によって動作が変わる関数の例

再利用可能な関数でないと判断してしまう．

2.2.3 入力の取る範囲によって動作が変わる関数

3 つ目の問題点は，入力の厳密な値ではなくその値が取る範囲によって関数の動作が決定される場合にも入力値の厳密な比較をするために，本来再利用可能な関数を再利用不可能と判断してしまう点である．この例を，**図 3** に示す．関数 h は，第 2 引数 b の値が取る範囲によって動作が変化する関数である．この関数は 11 行目と 13 行目で呼び出されており，それぞれ第 2 引数が異なっている。それぞれ入力値が異なるため自動メモ化プロセッサは再利用不可能と判断する．ここで，各関数呼び出しにおける実際の関数の動作を考える．入力 b の値は 11 行目では 1，13 行目では 2 となっており，いずれも 2 行目の条件式を真とする。このとき，2 行目以外で入力 b は使用されておらず，h(4,1) と h(4,2) とでは関数の動作結果に違いが無いことがわかる．よって，これらの入力に対する関数 h の出力は本質的には再利用可能であったと言える．

3. 提　　案

自動メモ化プロセッサは，計算再利用の単位として関数を利用している．そのため 2 章で述べたように，プログラマが関数をどのように定義するかによって，再利用の成否や効率が大きく変化する．そこで本稿では，プログラムに変換を施すことにより，計算再利用が可能な部分を増加させる手法を提案する．本章では，どのような変換が可能かを示し，それぞれの場合に期待される効果とオーバーヘッドについて述べる．

3.1 異なる有効範囲を持つ入力の分離

図 1 に示したプログラムの関数 f を考える．2 行目と 3 行目の計算には第 3 引数 c は使用されておらず，これらの計算が同じ結果となるには，第 1 引数 a と第 2 引数 b が一致すればよい．図 1 のプログラムは，**図 4** のように変換することで，入力 c に関係しない処理を関数として分離することが出来る．新たに分離生成された関数 f_b は，引数として入力 a および b の 2 つを取り、それらの和と差をメンバ変数として持つ構造体を返す．図 1 の関数 f 全体でしか再利用表に登録出来ていなかったものが，以上の変換を施すことによって 2 行目と 3 行目のみを新たに命令区間として再利用表に登録することが可能となる．

一方で，プログラム内部で呼ばれる関数の数が増えるため，関数の呼び出しコストと，再利用が可能かの判定のためのコストの増加が考えられる．加えて，返り値のメンバ変数の代入のためのコストも発生する．また，関数の種類が増えるため，再利用表に登録される関数の数も増えることになり，有限な再利用表の領域が圧迫される恐れがある．

```
 1:  typedef struct{
 2:      int a_0;    int b_0;
 3:  }output;
 4:
 5:  output f_b(int a, int b){
 6:      int a_0 = a + b;
 7:      int b_0 = a - b;
 8:      output tmp;
 9:      tmp.a_0 = a_0;
10:      tmp.b_0 = b_0;
11:      return(tmp);
12:  }
```

```
12:  int f(int a, int b, int c){
13:      output tmp = f_b(a,b);
14:      int a_0 = tmp.a_0;
15:      int b_0 = tmp.b_0;
16:      int a_1 = a_0 * b_0 * c;
17:      return(a_1);
18:  }
```

図 4 異なる有効範囲を持つ入力の分離の例（変換後）

```
1: int *__buf_g = NULL;
2:
3: int gMain(int array[],int size){
4:     int i,sum = 0;
5:     for(i=0;i<size;i++)
6:         sum += array[i];
7:     return(sum);
8: }
```

```
 9: int g(int array[], int size){
10:     if(__buf_g == NULL)
11:         __buf_g = malloc(size*sizeof(int));
12:     memcpy(__buf_g, array,size*sizeof(int));
13:     return(gMain(__buf_g, size));
14: }
```

図 5 不要なアドレス情報の削除の例（変換後）

```
 1:  int h_T(int a){
 2:      a = -a;
 3:      int a_0 = a * 2;
 4:      return(a_0);
 5:  }
 6:
 7:  int h_F(int a){
 8:      a = a / 3;
 9:      int a_0 = a * 2;
10:      return(a_0);
11:  }
12:  int main(){
13:      int b = 1;
14:      if(b > 0)
15:          h_T(4);
16:      else
17:          h_F(4);
18:      b = 2;
19:      if(b > 0)
20:          h_T(4);
21:      else
22:          h_F(4);
23:  }
```

図 6 条件分岐の分離の例（変換後）

3.2 関数入力からの不要なアドレス情報の削除

2.2.2 項の図 2 に示すプログラムの関数 g を考える．この関数を，配列に格納された値のみで一致比較が可能なようにするためには，**図 5** のように変換すればよい．大域変数として，関数の入力にある配列の型と同じ型のポインタ変数__buf_g を宣言する．対象となる関数の内部で，入力として用いられた配列と同じ大きさの領域をメモリ上に確保する．この領域を本稿では一時領域と呼ぶことにする．この一時領域のアドレスは大域変数に格納される．領域の確保はプログラム実行中 1 度しか行われないため，プログラムの実行中は一時領域のアドレスは変わらない．一旦その一時領域に，ポインタ変数__buf_g を介して入力の配列 array の値を格納し，そのあと一時領域の先頭アドレスを，本来の関数と同じ働きを持つ関数に入力として与える．以上の結果，入力の配列の先頭アドレスは常に一定となる．そのため，配列の先頭アドレスが再利用の可否に影響を与えなくなり，配列の値のみの一致比較が可能となる．

一方でこの変換では，不要なアドレス情報を削除するために，メモリ領域の確保や値のコピーなどの新たな処理をプログラムに追加している．当然ながら実行する命令数は増加しており，実行速度の低下に繋がる可能性がある．

3.3 条件分岐の分離

2.2 節の図 3 のプログラムの関数 h は，第 2 引数 b の取る範囲によって動作が変化する関数である．この関数の呼び出しは 11 行目では h(4,1)，13 行目では h(4,2) となっており，両者の実行結果に違いは無い．そこで，条件分岐を関数呼び出し元に移動させ，条件分岐の結果に応じて呼び出す関数を変更するようにする．これによって関数の引数が 1 つ減り，計算再利用が適用できるケースを増やすことが可能であると考えられる．このような変換を図 3 のプログラムに施した結果を**図 6** に示す．図 3 のプログラムの条件式の真偽それぞれの場合に対応した 2 つの関数 h_T，h_F を新たに定義する．また，元々の関数 h 内部にあった条件分岐を関数 h の呼び出し元に移動させ，条件式の真偽によって呼び出す関数を変化させるようにする．この変換の結果，変換後は第 1 引数の値だけを登録すればよいようになり，再利用表に登録されるパターンが減少する．これにより，再利用表の領域の節約が期待できる．

一方で，条件分岐を関数の呼び出し元で行うようにするため，関数の計算再利用が成功したとしても，条件分岐の比較命令は省略できない．そのため，計算再利用によって省略できるサイクル数は減少する．

4. 実　　装

3 章に示したコード変換は，専用のトランスレータを用いることで実現する．トランスレータには，プログラムのソースを入力として与える．トランスレータは与えられたソースに対して変換を施し，変換されたソースファイルを出力する．本実装では，対象となるプログラミング言語を C 言語としている．

トランスレータは，大きく分けてソース解析とコード変換の 2 つの工程から構成される．以下の節では，各工程の作業内容について述べる．

4.1 ソース解析

4.1.1 ソース解析の概略

ソースを変換するために最初に行う必要があるのは，ソースの解析である．入力されたソースがどのような構造を取り，どのような変換が可能かを調べる必要がある．ここで調べるのは，各変数の有効範囲，始祖変数，用途である．変数の有効範囲を（v_{Lstart}，v_{Lend}）の形式で表すこととする。ここで、v_{Lstart} は変数が最初に宣言される行番号であり，v_{Lend} は変数が最後に使用される行番号である．例えば，図 1 の関数 f における変数の有効範囲は，a は (1,5)，b は (1,4)，c は (1,4) と表せる．また本稿では，ある変数が持つ値もしくはその値を用いた計算結果が他の変数に代入される場合，後者は前者に依存していると考え，前者を後者の親変数と呼ぶことにする．多くの場合において親変数もまた親変数を持つが，この依存関係をたどることにより，各変数の値に影響を与え得る入力変数や局所変数を調べることができる．この入力変数および局所変数のことを始祖変数と呼ぶ．例えば，図 4 の関

```
1:  int f(int a, int b, int c){
2:      int a_0 = a + b;
3:      int b_0 = a_0 - b;
4:      int a_1 = a_0 * b_0 * c;
5:      return(a_1);
6:  }
```

図 7 図 1 の関数 f 変数リネーミング結果

数 f において，a_0 の始祖変数は a,b であり，a_1 の始祖変数は a,b,c である．

ここで，ある変数に対して代入が起こった場合を考える．値の代入とは，つまり値の上書きである．上書きをされているということは，代入の前後では全く異なる値を持つ．このことは，たとえ同じ名前を持つ変数でも，それが持つ値が依存している変数が常に同一とは限らないことを示している．このような変数を把握するために，ソース解析の下準備として変数の名前を値の依存関係が分かりやすい形に変換していく工程を挟む．この工程を**変数リネーミング**と呼ぶことにする．これは，各変数のソース解析全体の流れとしては，最初に変数リネーミングを行い，そのあとソースから変数の有効範囲や始祖変数，用途といった情報を解析する．

4.1.2 変数リネーミング

変数リネーミングとは，変数に対して代入が起こる際に，代入先となる変数を逐次新たに宣言し，同名の変数を継続して使用しないようにすることである．例として，3.1 項の図 1 の関数 f に，変数リネーミングを施す場合を考える．まず，関数 f が実行され，2 行目で変数 a に対して値の代入が行われる．このとき，a の値は新しい値に上書きされる．3 行目における変数 b への代入にも同様のことが言える．新しい値であることを考えると，代入先を他の変数と置換したとしてもプログラムに影響はない．そこで代入先の変数を新たに宣言し，それ以降に現れる古い変数名を新しい名前に変更することにする．図 1 の関数 f の場合，2 行目の左辺の a と 3 行目の左辺の b の名前を変え，更に 4 行目の右辺の a と b をその変更した変数名と同じ名前に変更する．これと同様の変換を繰り返すことで，変数名から値の有効な範囲が明確にわかる．図 1 の関数 f に変数リネーミングを施した結果は，**図 7** のようになる．

ただし変数リネーミングは，適用可能な場面が限られる．適用できない場合を以下に示す．

- ポインタ変数やポインタ変数との関連付けがされた変数は，他の変数から値を参照される可能性があるため，変数リネーミングできない．
- 配列や構造体の要素について変数リネーミングを行う場合，値の一貫性を保持するためには大きなコピーコストがかかるため，配列や構造体への代入については変数リネーミングを施すべきでない．
- 代入が起こる際のスコープが，代入先変数のスコープと異なる場合，新たに宣言した変数が無効になるおそれがあるため，変数リネーミングは適用できない．

以上で示した変数リネーミングを施したソースコードから，コード変換に必要なデータを抽出する．抽出するデータは「変数の有効範囲」,「各変数の始祖変数」,「変数の用途」である．ここで言う変数の有効範囲および始祖変数は，4.1.1 項で説明したものである．変数の用途とは，変数がどのような目的で使用されるかである．本実装では，条件分岐の条件式で使用されるかとループ文のイタレータとして使用されるかを調べる．

以上で述べたデータはコード変換ルーチンへと引き渡される．

4.2 コード変換

4.2.1 コード変換の流れ

コード変換ルーチンは，ソース解析ルーチンで抽出されたデータを基にソースコードの変換を行う．

変換の概要を説明する．コード変換は，大きく分けて関数内部の変換と関数呼び出し元の変換から成る．ここで言う呼び出し元とは，プログラム内の，変換の対象となる関数を呼び出すコードが書かれている箇所を指す．

まずはじめに，関数内部の変換を行う．変換ルーチンは関数定義の部分を探し，その関数の内部のコードが 3 章で示したパターンに当てはまるかどうかを調べていく．コードがパターンに当てはまるとき，コード変換ルーチンは，そのパターンに対応した変換を施す．対応した変換が関数の呼び出し元の変換を必要とする場合，トランスレータはその関数名を記録する．この記録は関数呼び出し元の変換の際に使用される．パターンに当てはまらない場合は，変換を施さないまま出力される．以上の工程をすべての関数に対して繰り返すことで，関数内部は変換される．

関数呼び出し元の変換では，関数変換履歴をもとに，それに対応した変換を施す．内部変換されたソースコードを読み込み，呼び出し元の変換を要する関数について，逐次変換を行う．以下の項では，プログラム変換の条件を，3 章で示した変換の種類ごとに説明する．

4.2.2 異なる有効範囲を持つ入力の分離

3.1 節に示した，入力変数が異なる有効範囲を持つ場合に，関数の一部を新たな関数とし

変数名	a	b	c	a_0	b_0	a_1
有効範囲 (v_{Lstart}, v_{Lend})	(1,2)	(1,3)	(1,4)	(2,4)	(3,4)	(4,5)

表 1 図 7 の変数の有効範囲

て定義するような変換を施す手順を，例を用いて説明する．まず，図 1 のプログラムに対して変数リネーミングを施す．その結果は，前節の図 7 に示されている．このプログラムの変数の有効範囲は，**表 1** のようになっている．まずはじめに，新しい関数の生成を行う．全ての入力変数が使用されるまでのコードを読み込み，それらを新たな関数の命令コードとする．この例では，使用されない入力変数がなくなる最初の行は 4 行目である．このことから，この直前 3 行目までが入力 c に関係なく結果が得られる命令区間であると判別できる．次に，生成した新しい関数の出力に用いる構造体を宣言するコードを生成する．この構造体はメンバ変数として，関数として切り離したコードとそうでないコードの両方に有効範囲が及んでいる変数と同じ変数を持つ．この例では，有効範囲に 3 行目と 4 行目の両方が含まれている変数は a_0,b_0,c の 3 つである．ここで例外として，使用されていないことが保証される入力 c をメンバ変数の候補から除外する．以上の工程から，構造体のメンバ変数が決定される．最後に，生成された関数の出力を元の変数に代入するコードを生成する．以上の工程を経ることで，一部の入力変数のみを使用するコードを新たな関数として切り出すことができる。

4.2.3 関数入力からの不要なアドレス情報の削除

3.2 節に示した，配列の先頭アドレスを入力とする際に配列の値を一時領域に一旦格納することで，配列の先頭アドレスが常に一定となるようにする変換を施す手順を，例を用いて説明する．2.2.2 項の図 2 のプログラムは，引数としてポインタ変数 array を取る．引数にポインタ変数が含まれていることを確認した場合，このポインタが配列の先頭アドレスとして使用されることを可能性に含めてソースコードの読み込みを行う．次に，for 文に着目する．関数 g の内部には for 文によるループが存在し，その中で配列は使われている．どの変数がイタレータであるかの判定には，以下の条件を用いることにする．

- for 文の構文中にて，定数による初期化が行われている．
- 条件式に含まれ，尚且つその条件式が単純な大小比較である．
- 1 イタレーションごとに，定数が加算される．

上記の条件は極めて厳しく，これに当てはまらないケースも存在するが，上記の条件を満たす変数の記述は，イタレータ変数の記述として最も一般的な形式のうちの 1 つであるため，これらの条件を満たす場合は多いと考えられる．図 2 に示されている関数 g の for 文はこれらの条件を満たし，イタレータ変数である i の値の取りうる範囲が 0 から size-1 までであることがわかる．

また，配列の添え字にはイタレータ i のみが使用されている．これより，この関数の内部において配列 array は size 番目の要素までしかアクセスされないことが保証できる．

以上を満たすとき 3.2 節に示した変換を施すことが出来る．

```
1:  int h(int a, int b){
2:      if(b > 0)
3:          a = -a;
4:      else
5:          a = a / 3;
6:      int a_0 = a * 2;
7:      return(a_0);
8:  }
```

図 8 条件分岐の分離の例（変数リネーム後）

```
1:  int h_T(int a, int b){
2:      a = -a;
3:      int a_0 = a * 2;
4:      return(a_0);
5:  }

1:  int h_F(int a, int b){
2:      a = a / 3;
3:      int a_0 = a * 2;
4:      return(a_0);
5:  }
```

図 9 条件分岐の分離の経過

4.2.4 条件分岐の分離

3.3 節に示した，条件分岐の分岐ごとに関数を用意するように変換する手順を，例を用いて説明する．2.2.3 項の図 3 の関数 h に変数リネームを施した結果は，**図 8** のプログラムのようになる．ここで，変数 b が 2 行目の条件式に使用されていることに着目する．入力 b が変数リネーム後に条件式に使用されていることは，b の入力値が条件分岐の成否に直接影響することを意味する．このことを引き金として，条件分岐ごとに関数の分離が可能かどうかを調べ始める．調べる手順は以下の通りである．

まず，元から存在する関数 h とは別に，2 行目の条件分岐で分離された場合の関数を実際に生成する．すると**図 9** のようになる．本実装では条件分岐を分離して関数を生成する際，新たに生成する名前は元の関数の名前の後ろに，条件式が真の場合のものは¨_T¨を，偽の場合のものは¨_F¨を付けることとしている．入力 b の有効範囲は，関数 h_T および h_F のいずれにおいても (1,1) である．これは，関数内部には変数 b は存在しないことを意味するため，関数の引数から b を消去可能であり，これらの関数 h_T と関数 h_F を生成することにより，関数を再利用するために一致が必要となる値の数が減少する．条件文が入れ子構造と

実行環境のパラメータ

D1 Cache 容量	32	KBytes
ラインサイズ	32	Bytes
ウェイ数	4	
レイテンシ	2	cycles
ミス ペナルティ	10	cycles
D2 Cache 容量	32	KBytes
ラインサイズ	32	Bytes
ウェイ数	4	
レイテンシ	10	cycles
ミス ペナルティ	100	cycles
Register Window 数	4	sets
Window ミス ペナルティ	20	cycles/set
ロード レイテンシ	2	cycles
整数乗算 〃	8	cycles
整数除算 〃	70	cycles
浮動小数点加減乗算 〃	4	cycles
単精度浮動小数点除算 〃	16	cycles
倍精度浮動小数点除算 〃	19	cycles
再利用表のエントリ数	64K	エントリ

GENEsYs のパラメータ

交叉率	60.0	%
突然変異率	0.1	%
個体数	50	体
世代数	2000	世代
その他	default	値

表 2 シミュレーション時のパラメータ

なっている場合は，上述の処理を再帰的に施すことによって変換可能である．以上の変換を施すと，3.3 節の図 6 に示したように，変数 b の値の範囲によって異なる 2 種類の関数を呼び出すプログラムになる．

図 6 では省略しているが，変換を施したあとも，変換前に存在した関数 h の定義はそのまま残すこととする．これは，関数ポインタなどによる検出出来ない関数呼び出しを持つプログラムに対して本手法を適用した場合にも，プログラムを正常に動作させるためである．

5. 評　　価

前章に示したトランスレータを実装し，それを用いて変換したプログラムの実行サイクル数を，適用していないプログラムのものと比較した．

5.1 評 価 環 境

実行環境は，計算再利用のための機構を実装した単命令発行の SPARC-V8 シミュレータを用いた．また，評価対象プログラムには汎用 GA ソフトウェアである GENEsYs[4] を用いた．GA は生殖，適合度計算，個体選択の 3 ステージを繰り返して行うアルゴリズムであり，適合度計算の処理が最も負荷が高い．この適合度計算のための評価関数が，GENEsYs

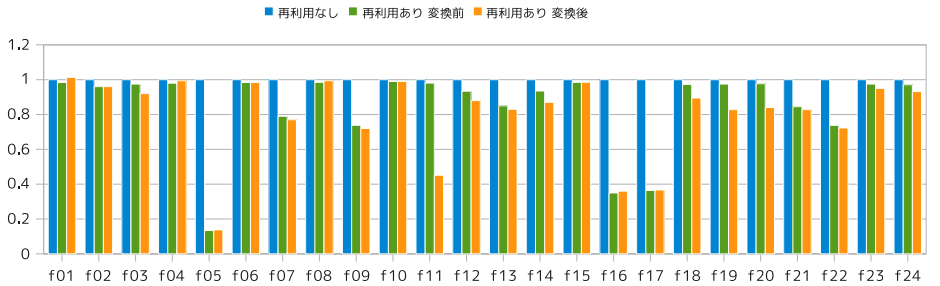

図 10 GENEsYs の実行時間

には 24 種類用意されており，今回はこれらの評価関数に対して提案手法を適用し評価を行った．評価時の各パラメータを**表 2** に示す．

評価プログラムは，gcc3.0.2 にてコンパイルを行い，スタティックリンクによって生成したものを用いた．コンパイルオプションには，変換の対象とならないソースファイルについては-O2 を用いた．一方，変換の対象となったソースファイルについては-O1 を用いた．コンパイルオプションに-O1 を用いた理由は，コンパイラによるインライン展開などの最適化によって，施した変換が無効化されるのを防ぐためである．

5.2 結果と考察

GENEsYs による評価結果を**図 10** に示す．横軸は実行した評価関数を示している．各評価関数について，左から順に再利用なし，再利用ありで変換前のもの，再利用ありで変換後のものを表す．縦軸は実行時間を示しており，各評価関数について，再利用なしのプログラムの実行時間を 1 として正規化している．

提案手法により，13 個の評価関数の実行サイクル数が短縮された．提案手法による変換を行ったプログラムの実行サイクル数は，変換を行っていないプログラムと比べ，平均 4.6%，評価関数 f11 において最大 53.6%削減できた．

提案手法により，プログラム実行の高速化が実現していることが確認出来る．本提案手法により，計算再利用による実行時間の削減が，配列の一時領域へのコピーや関数呼び出しのオーバーヘッドを上回ったことを示している．一方，7 個の評価関数に関しては実行サイクル数は増加しており，評価関数 f01 で再利用なしの場合と比べて最大 1.3%，再利用ありで

変換なしの場合と比べて 3.5%増加してしまっている．これは，配列の一時領域へのコピーや関数呼び出しのオーバーヘッドが表れていると考えられる．しかし，最大削減率と比較すると増加率は軽微である．これは，本提案が対象とするプログラムがメモリアクセスを大量に行うものであったため、メモリコピーのオーバーヘッドが目立たなくなったためだと考えられる．実行サイクル数に大きな変化が見られない関数では，計算再利用が適用されるケースが増加しているにも関わらず，削減された実行サイクル数はオーバーヘッドによって打ち消されてしまっていると考えられる．

6. まとめと今後の課題

本稿では，自動メモ化プロセッサで計算再利用が効率的に行われるようにするために，プログラムに事前に変換を施すことを提案した．提案手法をトランスレータとして実装し，GENEsYs をベンチマークに用いて評価を行ったところ，計算再利用の効率が向上し実際に高速化が行われたことが確認できた．

今後の課題として，再利用を行っても利得が見込めないような関数の生成を制限する手法や，より広い範囲を計算再利用の単位区間として登録可能となるようにすることで，1 回の計算再利用でより多くの処理を省略可能なようにする変換手法の提案と実装が考えられる．また，今回は GA を対象として評価を行ったが，今後は SPEC CPU などの一般的なプログラムを対象としての評価も行いたい．

参 考 文 献

1) 大津金光，小野喬史，横田隆史，馬場敬信：バイナリレベルマルチスレッド化コード生成方法とその評価，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム， Vol.42, No.SIG 1(HPS 6), pp.70–80 (2003).
2) Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. Parallel and Distributed Computing and Networks*, pp.245–250 (2007).
3) Norvig, P.: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992).
4) Bäck, T.: GENEsYs 1.0. Software distribution and installation notes (1992).