卒業研究論文

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

指導教員　　松尾 啓志 教授
　　津邑 公暁 准教授

名古屋工業大学 工学部 情報工学科
平成 17 年度入学　17115051 番

加藤 拡

平成 21 年 2 月 10 日

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

内容梗概

今日までに，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) に着目したものやスレッドレベル並列性 (Thread-Level Parallelism：TLP) に着目したものが提案されてきた．しかしながら，これらの手法による更なる高速化は難しい状況になりつつある．

一方で，これらとは全く異なる高速化手法として，計算再利用というものが存在する．本研究ではこの計算再利用に着目する．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．専用のハードウェアを用いることにより，関数を単位区間として計算再利用を実現するモデルとして、自動メモ化プロセッサというものが存在する．

本論文では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、関数の動作が，入力の値が持つ範囲によって変化する場合に，値が異なったとしても範囲が一致すれば再利用が可能となるようにする手法である．

上記の提案手法をトランスレータによるソース変換として実装し，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．

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

目次

1 はじめに **1**

2 背景 **1**

2.1 自動メモ化プロセッサ 2

2.2 問題点 2

2.2.1 一部のみが再利用可能な関数 2

2.2.2 配列の先頭アドレスを入力に取る関数 3

2.2.3 入力の取る範囲によって動作が変わる関数 4

3 提案 **5**

3.1 異なる有効範囲を持つ入力の分離 5

3.2 関数入力からの不要なアドレス情報の削除 6

3.3 条件分岐の分離 8

4 実装 **10**

4.1 実装の概略 10

4.2 ソース解析 10

4.2.1 ソース解析の流れ 10

4.2.2 変数リネーミング 11

4.2.3 データの抽出 13

4.3 コード変換 13

4.3.1 コード変換の流れ 13

4.3.2 異なる有効範囲を持つ入力の分離 15

4.3.3 関数入力からの不要なアドレス情報の削除 15

4.3.4 条件分岐の分離 17

4.3.5 関数の呼び出し元が不明な場合への対応 19

5 評価 **21**

5.1 評価環境 21

5.2 結果 21

5.3 考察 21

6 まとめと今後の課題 23
謝辞 23
参考文献 23

1　はじめに

今日までに，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．また，並列性を高めるための命令のスケジューリング制御部分はハードウェアに負担をかけている．この流れを受け，CPU がマルチコア化されるとともに，命令レベル並列性より荒い並列性であるスレッドレベル並列性 (Thread-Level Parallelism：TLP) が注目されることとなった．TLP に着目した手法として、マルチスレッドライブラリや高度なコンパイラが存在する．ライブラリを用いた場合，プログラム内の並列化できる部分を明示的に指定する必要があり，プログラマに負担がかかる．一方，コンパイラによる TLP の抽出はその精度に問題がある．よって TLP によるプログラム実行の高速化もまた難しい状況であるといえる．

これらはいずれもプログラム実行の並列化という方法で高速化を図るものであるが，本研究はそれとはまったく異なる計算再利用という手法に着目する．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．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、自動メモ化プロセッサというものが存在する．本論文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．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．本研究ではこの一致比較をより効率的に行うために，プログラム変換による高速化を図る．

以下，2 章では本研究が扱う自動メモ化プロセッサの動作モデルを述べる．また，それが持つ問題点について説明する．3 章ではどのようにプログラムを変換するかを提案し，4 章で変換の手順と実装を説明する．5 章で本提案の評価を行い，6 章で結論を述べる．

2　背景

本研究で取り扱う自動メモ化プロセッサについて，その高速化の方針と原理，そしてその問題点を概説する．

2.1 自動メモ化プロセッサ

メモ化とは，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させることができる．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 [1] 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，自動メモ化プロセッサは入力と出力を表に登録する．ここで入力とは，関数の引数，および関数の内部から参照される大域変数を指す．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，現在呼び出されている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．そのため，その分だけプログラムの実行時間が短縮出来る．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，再利用可能な区間を見逃してしまう可能性を持つことが挙げられる．そのような関数の例を以下に３つ示し，それらを用いて問題点を説明する．

2.2.1 一部のみが再利用可能な関数

１つ目の問題点は，関数の一部のみが再利用可能である場合は再利用を適用できない点である．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引数の和を第３引数で割った値を返す関数である．関数 f が呼び出されるのは，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(3,3,2) 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可能でないと判断し，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) の場合，２行目で４＋２の計算が，３行目で６÷２の計算が行われる．f(3,3,2) の場合，２行目で３＋３の計算が，３行目で６÷２の計算が行われる．また，３行目の計算結果がそのまま返り値であることから，

```
 1:  int f(int a, int b, int c){
 2:      a = a + b;
 3:      a = a / c;
 4:      return(a);
 5:  }
 6:
 7:  int main(){
 8:      f(4,2,2);
 9:      f(3,3,2);
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:      }
 7:      return(sum);
 8:  }
 9:
10:  int main(){
11:      int x[3] = {1,2,3};
12:      int y[3] = {1,2,3};
13:      g(x,3);
14:      g(x,3);
15:      g(y,3);
16:      return(0);
17:  }
```

図 2: 配列の先頭アドレスを入力に取る関数の例

`f(3,3,2)` の実行の際，3 行目から 5 行目は再利用可能であったと言うことが出来る．

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．図 2

において，関数 f 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に関数 g を呼び出している．関数 g が呼び出されるのは，13，14，15 行目の 3 箇所である．最初に 13 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．そのあと 14 行目で再び g(x,3) が呼び出されるとき，再利用表の検索により現在の関数の入力が過去の入力と一致することがわかる．これにより計算再利用が可能と判断され，14 行目の g(x,3) の実行は省略される．そのあと，今度は 15 行目で g(y,3) が実行される．このときは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ない．

関数 f 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ことから，g(y,3) は，本質的には g(x,3) の結果を用いて計算再利用を行うことができる．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，再利用表には配列の先頭アドレスが記録されている．それらの先頭アドレスは絶対アドレスとして記録されており，当然ながら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，引数として与えられている先頭アドレスが一致しない．そのため，自動メモ化プロセッサは，再利用可能な関数では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

同様に，自動メモ化プロセッサでは再利用可能であることを見逃してしまう例を，図 3 に示す．図 3 の関数 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 14 行目と 16 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は出来ないと判断する．ここで，それぞれの場合における実際の関数の動作を考える．入力 b の値はそれぞれ 1 と 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，ここ以外で入力 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，この関数 h の出力は本質的には再利用可能であったと言える．

```
 1:  int h(int a, int b){
 2:      if(b > 0){
 3:          a = -a;
 4:      }
 5:      else if(b < 0){
 6:          a = a / 3;
 7:      }
 8:      int a = a * 2;
 9:      return(a);
10:  }
11:
12:  int main(){
13:      int b = 1;
14:      h(4,b);
15:      b = 2;
16:      h(4,b);
17:  }
```

図 3: 入力の取る範囲によって動作が変わる関数の例

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，再利用の可否や効率が大きく変化する．そこで本研究では，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，それぞれの場合の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

関数の中には入力が複数必要となるものが多く存在するが，それら入力のすべてが，その関数内部で最初から使用されるわけではなく，また最後まで使用されるわけでもない．例えば，2.2.1 項で示した例の図 1 では，入力 c は 3 行目からのみ使用されている．そこに着目し，複数ある入力が使用される範囲が異なるとき，関数の一部を別の関数として分離することで，関数の部分的な再利用を疑似的に実現する．

変換

図 4 に示すような C 言語によるプログラムを考える．これは，2.2 節の図 1 の抜粋である．関数 f は整数型の引数を 3 つ取る関数であり，第 1 引数 a と第 2 引数 b との和を

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

図4: 異なる有効範囲を持つ入力の分離の例（変換前）

```
1:  int f_b(int a,  int c){
2:      a = a / c;
3:      return(a);
4:  }
5:  int f(int a, int b, int c){
6:      int a_0;
7:      a_0 = a + b;
8:      return(f_b(a_0, c));
9:  }
```

図5: 異なる有効範囲を持つ入力の分離の例（変換後）

第3引数cで割った値を返す．3行目の計算では，cの値と，2行目の計算の結果の値が使用されている．これら2つの値が一致すれば，この関数は結果が同じであったと考えられる．図4のプログラムは，図5のように変換することで，入力cに関わる処理を関数として分離すること出来る．新たに分離生成された関数f_bは，引数としてa+bの結果とcの2つを取るようになる．

期待される効果とオーバーヘッド

変換を行う前は，図4の1行目から5行目までの区間でしか再利用表に登録出来ていなかった．しかし，変換を施すことによって3行目以降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．これにより，より細かい単位での計算再利用が可能となる．

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，関数呼び出しのオーバーヘッドが増えることが考えられる．具体的には，関数を呼び出すCALL命令やRETURN命令のコスト，再利用が可能かの判定のためのコストなどが挙げられる．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，これにより有限な再利用表の領域が圧迫される恐れがある．そのため，場合によっては計算再利用の効率が下がることも考えられる．

3.2 関数入力からの不要なアドレス情報の削除

関数の入力として配列の先頭アドレスが渡されながらも，関数内部では配列の値しか参照されない場合がある．この場合，配列のアドレスは関数の結果に直接影響を与えない．しかし，入力としては配列の先頭の絶対アドレスを持つため，計算再利用が適

```
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:      }
7:      return(sum);
8:  }
```

図 6: 不要なアドレス情報の削除の例（変換前）

用できない．例えば，2.2.2 項で示した図 2 では，15 行目の関数呼び出しの入力が，再利用表に登録されているアドレスと一致しないため，計算再利用は適用されない．そこで，常に同じメモリ空間に一旦値を格納し，その先頭アドレスを渡すことで，関数内では常に一定のアドレスで配列の要素が参照されるようにする．これにより，アドレスの違いによって計算再利用ができなくなる問題を回避できる．

変換

図 6 に示すような C 言語によるプログラムを考える．これは，2.2.2 項の図 2 から抜粋したものである．配列の先頭ポインタとその要素数を引数に取り，その配列の要素の値の合計を計算して返す関数である．この関数では，引数として配列の先頭アドレスを必要としている．しかし，関数の内部で使用されるのは配列の要素の値のみである．このことから，この関数は入力の配列の先頭アドレスに変化があっても，それが持つ値に変化が無ければプログラムの動作は同じであると言える．しかし実際には，自動メモ化プロセッサは配列の先頭アドレスを再利用表に登録し，入力の一致比較時には先頭アドレスに対して比較を行う．メモリ上のどこに値が格納されているかが完全に一致することで初めて計算再利用が可能となるため，本来計算再利用が適用可能であったとしても適用されない場合があると言える．

この図 6 に示すような関数を，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 7 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．この領域を本論文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，また領域の確保はプログラム実行中 1 度しか行われない．そのため，プログラムの実行中はアドレスが変わらない．一旦その一時領域に，ポインタ変数__buf_g を介して入力の

```
 1:  int *__buf_g = NULL;
 2:
 3:  int g_main(int array[], int size){
 4:      int i;
 5:      int sum = 0;
 6:
 7:      for(i=0;i<size;i++){
 8:          sum +=array[i];
 9:      }
10:      return(sum);
11:  }
12:
13:  int g(int array[], int size){
14:      if(__buf_g == NULL)
15:          __buf_g = (int *)malloc(size*sizeof(int));
16:      memcpy(__buf_g, array, size*sizeof(int));
17:      return(g_main(__buf_g, size));
18:  }
```

図 7: 不要なアドレス情報の削除の例（変換後）

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．

期待される効果とオーバーヘッド

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疑似的に可能となる．これにより，配列のアドレスは異なるが値は一致するという場合の再利用が可能となる．

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

例えば 2.2.3 項の図 3 に示されるように，関数の中には入力値の範囲によって関数の挙動が変化するようなものが存在する．このような関数は，入力が完全に一致せずとも一定の範囲を取っているならば再利用が可能であると考えることが出来る．そのような関数を，条件分岐ごとに関数をそれぞれ用意することで，計算再利用が適用でき

```
 1:  int h_T(int a){
 2:      a = -a;
 3:      int a_0;
 4:      a_0 = a * 2;
 5:      return(a_0);
 6:  }
 7:
 8:  int h_F_T(int a){
 9:      a = a / 3;
10:      int a_0;
11:      a_0 = a * 2;
12:      return(a_0);
13:  }
14:
15:  int h_F_F(int a){
16:      int a_0;
17:      a_0 = a * 2;
18:      return(a_0);
19:  }
```

```
20:  int main{
21:      int b = 1;
22:      if(b > 0)
23:          h_T(4,b);
24:      else if(b < 0)
25:          h_F_T(4,b);
26:      else
27:          h_F_F(4,b);
28:      b = 2;
29:      if(b > 0)
30:          h_T(4,b);
31:      else if(b < 0)
32:          h_F_T(4,b);
33:      else
34:          h_F_F(4,b);
35:  }
```

図 8: 条件分岐の分離の例（変換後）

る可能性を向上させる．

変換

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 14 行目では h(4,1)，16 行目では h(4,2) となっており，h(4,1) と h(4,2) とでは関数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植し，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．そ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 8 に示す．図 3 のプログラムの条件式の真偽に応じた 3 つの関数 h_T，h_F_T，h_F_F が新たに定義される．また，元々の関数 h 内部にあった条件分岐を関数 h の呼び出し元に移植し，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．

期待される効果とオーバーヘッド

変換前には関数の入力によって動作が変化していたが，変換後は条件分岐によって呼び出される関数が変わるようになる．そのため，関数が必要とする入力の数が減少する．その結果，一致を必要とする入力数が減るため，計算再利用が効率的に行える

と考えられる．

また，再利用表に登録される入力の種類が減ると考えることもできる．上記の例では，変換後は第 1 引数の値だけの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が減少すると考えられる．

4 実装

4.1 実装の概略

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 つの工程から成る．いずれも字句解析を必要とする工程である．本実装では flex を用いて字句解析ライブラリを生成し，それをトランスレータ内部に組み込んでいる．

以下の節では，各工程の作業内容について述べる．

4.2 ソース解析

4.2.1 ソース解析の流れ

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，各変数の有効範囲と用途である．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 4 における変数の有効範囲は，a は (1,4)，b は (1,2)，c は (1,4) である．有効範囲を調べる上で注意しなくてはいけないのが，変数が持つ値の有効範囲である．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，その時点でそれ以前の値は失われることになり，代入の前後では変数の名前が同じなだけで，全く異なる値を持つと言える．以上から，変数の名前から判断される有効範囲と，変数の値を考慮した有効範囲は必ずしも一致しないと考えることが出来る．この

```
1:  int f(int a, int b, int c){
2:      int a_0;
3:      a_0 = a + b;
4:      int a_1;
5:      a_1 = a_0 / c;
6:      return(a_1);
7:  }
```

図 9: 図 4 の変数リネーミング結果

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有効範囲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．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用途の情報を解析する．

4.2.2 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 4 のプログラムに，変数リネーミングを施す場合を示す．まず，関数 f が実行され，2 行目で変数 a に対して a と b の和が代入される．このとき，a の値は元々の値から新しい値に上書きされる．新しい値であることを考えると，代入先が他の変数であったとしても，それ以降で参照されているのでない限りプログラムに影響はないと言える．しかし，この例では 3 行目で a の値は参照されており，2 行目の代入先の名前を単純に変えてしまうのは正しいとは言えない．しかし言い替えれば，名前を変更した変数を参照している他の箇所も同じ変数名に変更すれば，プログラムの動作に支障は無いということになる．これを利用し，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．この例では，2 行目の左辺の a の名前を変えても，3 行目の右辺の a をその変更した変数名と同じ名前にするならば，プログラムの動きは同じであるので問題はないと言える．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ようにする．図 4 のプログラムに変数リネーミングを施した結果は，図 9 のようになる．

変数リネーミングを施せるコードには条件がある．そこで，変数リネーミングが出来ないプログラムの例を図 10 に示し，それを用いて詳しく説明する．変数リネーミングを施すことが出来るのは，その変換がプログラムの動作に影響を与えないことを保障できる場合に限られる．図 10 の 8 行目では，変数 a とポインタ変数 p との関連付け

```
 1: int v[4] = {1, 2, 3, 4};
 2: int *p;
 3: int a = 0;
 4:
 5: if(a == 0){
 6:     a = 2;
 7: }
 8: p = &a;
 9: *p = 1;
10: v[a] = 0;
11: func(&a);
12: a = 2;
13: v[*p] = v[3];
```

図 10: 変数リネーミング出来ない例

が行われている．9 行目では，p の参照先に対して値の代入がされている．8 行目の関連付けから，*p と a は同じものを指す．そのため，この代入は変数 a に対しても影響を与える．ここで仮に，9 行目に変数リネーミングを施したとする．そのとき，新しい代入先と変数 a との関連付けはされない．そのため，変数 a に対して 9 行目の代入の影響が及ばなくなり，プログラムの動作そのものが変わってしまう．この例では，10 行目の代入先が変化する．このため，ポインタの参照先へ値を代入するコードに対しては，変数リネーミングを施してはならないと言える．

次に，12 行目の代入を考える．a と p が関連付けられているため，変数リネーミングによって a ではない別の変数に対し代入が行われるようになった場合，その代入の影響が p に及ばなくなる．これもまたプログラムの動作に支障をきたす．この例では，13 行目の代入先が変化する．そのため，他の変数に関連付けられた変数は変数リネーミングを施すべきではない．また，11 行目では変数 a が他の関数に対して入力値として参照渡しされている．関数の内部でどのような処理がされているかは不明であり，その内部でもまた他の変数に関連付けされている可能性もある．以上の理由から，アドレスを参照された変数はリネーミングの対象から外すべきであると考えられる．

次に，10 行目の代入を考える．代入先は配列の要素である．この代入に変数リネーミングを施そうとする場合，新たに配列を確保する必要がある．このとき，値の一貫性を保持するために，代入が行われなかった他の要素に関して値をコピーする必要がある．しかし，変数リネーミングを施すたびに各要素をコピーすると，プログラムの

処理時間は大幅に長くなってしまう．また，要素ごとに値の依存関係を調べようとしても，配列の添え字は変数であることが多く，その場合は配列のどの要素に対して代入が起こるかはソースコードからは読み取れない．以上の理由から，配列の要素への代入には変数リネーミングを行えないと言える．

配列と同様に，構造体もまた値の一貫性を保持するためのコストが非常に大きい．そのため，構造体の要素への代入も変数リネーミングを施す対象から除外した．

まとめると，変数リネーミングを施してよいコードは通常の変数同士，またはポインタ変数同士の代入であり，代入先の変数が他の変数に関連付けられていないことが保証される場合に限られる．

また変数リネーミングは，新たな局所変数を宣言しているという点から，変数のスコープに注意しなくてはならない．6 行目の代入に変数リネーミングを施した場合を考える．6 行目で新たに変数を宣言した場合，8 行目での代入はその新しい変数を参照するようになるはずである．しかし変数のスコープを考えると，その変数を参照出来る範囲は 7 行目までであり，8 行目からは参照出来ない．これはプログラムとしては誤った記述となる．代入が起こる際のスコープが，代入先となる変数が宣言されたのときのスコープと異なる場合，変数リネーミングは施すべきでないと言える．

4.2.3　データの抽出

前項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」と「変数の用途」である．

ここで言う変数の有効範囲は，4.2.1 節で説明したものである．ある変数に関して関数内の何行目に宣言されたかと，何行目で使用されなくなるかの 2 つの値の組み合わせで表される．例えば，図 9 のプログラムにおいて，変数 a の有効範囲は (1,3) であり，変数 a_0 の有効範囲は (2,5) 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．具体的には，条件分岐の条件式で使用されるか，ループ文のイタレータとして使用されるか，である．

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．

4.3　コード変換

4.3.1　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．

次に，変換の大きな流れを説明する．図 11 に，コード変換ルーチンのプログラムの

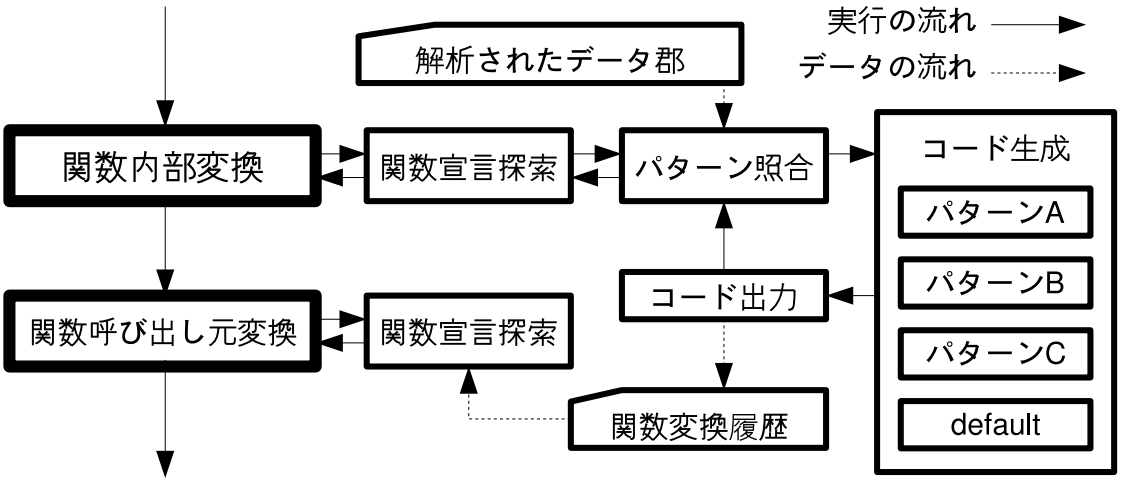

図 11: コード変換ルーチンの構成図

流れを表す．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．

変換のはじめに，変換ルーチンは変換対象となるソースを読み込み，関数定義の部分を探す．関数定義が見つかったら，関数の内部のコードがあらかじめ決められたパターンに当てはまるかどうかを調べていく．このパターンは，トランスレータがソースコードに対してどのような変換を施せばよいのかを調べるためのものである．パターンの詳細については後述する．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されたコードを生成する．当てはまったパターンが，変換対象となる関数の呼び出し元の変換を必要とする場合，トランスレータは変換前の関数名と変換パターンを関数変換履歴として記録する．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．変換を施されたコードは一旦バッファに出力される．そのあと再び関数定義を探し出し，変換が行われる．以上の工程を繰り返し，関数内部は変換される．

関数呼び出し元の変換では，関数内部でどのような変換が施されたかの履歴をもとに，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．

以下の節では，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．

変数名	開始（行目）	終了（行目）
a	1	3
b	1	3
c	1	5
a_0	2	5
a_1	4	6

表 1: 図 9 の変数の有効範囲

4.3.2 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 4 のプログラムに対して変数リネームを施す．その結果は，前項の図 9 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．表 1 からわかる情報として，入力 a，b が働きを失ったあとも，入力 c は働きを持つということが挙げられる．入力 a，b の有効範囲は 3 行目までであるのに対し，入力 c の有効範囲は 7 行目までである．このことから，4 行目以降を関数として切り出すことで，入力 a，b を入力として持たない関数を作ることが出来る．この場合，切り出す境界上に有効範囲が重なっている変数は，引数として渡される．これらの規則に従って関数を切り出すと，図 5 のプログラムが生成される．

4.3.3 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．3.2 節の図 6 のプログラムは，引数としてポインタ変数 array を取る．そのポインタ変数が値のみを参照されるとき，ポインタが持つアドレス情報はプログラムの動作に関係がないと言える．例のプログラムではポインタは配列として扱われており，その値のみが参照されている．配列に対する代入がある場合は，配列の要素の変化が関数外部に反映される必要がある．しかしこの関数では配列への代入は起こらないため，配列の値の変化の影響を考慮するは必要ない．

まず初めに，図 6 のプログラム中の関数 g の引数に着目する．引数にはポインタ変数 array が含まれている．ここで，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ようになる．

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．for 文の変数どの変数がイタレータであるかは，以下の条件から判断できる．

- for 文の構文中にて，0 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，1 ずつ加算が行われる．

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多く存在する．しかし，プログラムの動作に変化が起きてはならないため，条件は厳しく設定した．しかし一方で，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 6 に示されている例の for 文はこれらの条件を満足するため，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．

ここで配列の添え字に着目する．配列の添え字はイタレータ `i` のみが使用されている．このことから，この関数の内部においては，配列 `array` は `size` 番目の要素までのアクセスしか行われないことが保証できる．

以上のように，引数に取るポインタが配列の先頭アドレスであり，且つそれのアクセスされる範囲が特定可能である場合，別の配列に同じ値が格納された状態でその配列の先頭アドレスが渡されたとしても関数の出力には影響を与えないと言え，3.2 節に示した変換を施すことが出来る．

変換の第一段階として，元の関数と全く同じコードを持つ関数を別途生成する．図 7 では，`g_main` としている．次に，図 7 の 1 行目のように，`NULL` で初期化されたポインタ変数 `__buf_g` を大域変数として確保する．ここに格納された値は，プログラムの実行中は保持され続ける．関数 g 内の 15 行目では，`malloc` 関数で動的に確保された領域を 1 行目で宣言したポインタ変数に対して関連付けている．`malloc` 関数は，引数として渡された数値に等しいバイト数の領域をメモリ上に確保し，その領域へのポインタを返す関数である．この領域確保および関連付けは，14 行目の条件式によって，プログラム実行中は 1 度しか実行されないようになっている．この確保された領域は，プログラムの終了まで再度確保されることはない．そのため，この領域へのポインタはプログラム実行中は変化することはない．この領域に，関数の入力として与えられた配列の要素を一旦コピーすることにより，配列の先頭アドレスを常に一定に保つことが可能となる．配列同士のコピーは，16 行目の `memcpy` 関数にて行われる．この関数は，第 1 引数と第 2 引数にポインタを，第 3 引数に整数を取る関数である．第 1 引数

```
 1:  int h(int a, int b){
 2:      if(b > 0){
 3:          a = -a;
 4:      }
 5:      else if(b < 0){
 6:         a = a / 3;
 7:      }
 8:      int a_0;
 9:      a_0 = a * 2;
10:      return(a_0);
11:  }
```

図 12: 条件分岐の分離の例（変数リネーム後）

の指す領域に対して，第 2 引数が指す領域から第 3 引数の値の分のバイト数だけ値をコピーする働きを持つ．これにより，特定の領域への値のコピーが完了する．そして 17 行目で改めて，関数 g_main として宣言しなおされている本来の働きを持つ関数を，新たに確保した配列を入力として呼び出す．この新しい関数の返り値は，そのまま関数 g の返り値として呼び出し元に返される．

4.3.4 条件分岐の分離

3.3 節に示した，条件分岐を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．3.3 節の図 3 の関数 h に変数リネームを施した結果は，図 12 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が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 13 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．図 13 では，h_T と h_F となっている．関数 h_T は，関数 h の 2 行目の条件式が真のときに実行される 3 行目と 8 から 10 行目で構成される．関数 h_F は，関数 h の 2 行目の条件式が偽のときに実行される，5 行目から 10 行目までで構成される．ここで，これらの新しい 2 つの関数について，変数の有効範囲を再度調査する．調査結果は表 2 のようになる．

図 13 のプログラムでは，関数 h_T の変数 b の有効範囲は (1,1) である．これは，関

```
1:  int h_T(int a, int b){
2:      a = -a;
3:      int a_0;
4:      a_0 = a * 2;
5:      return(a_0);
6:  }

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

図 13: 条件分岐の分離の経過

関数 h_T

変数名	開始行	終了行
a	1	4
a_0	3	5
b	1	1

関数 h_F

変数名	開始行	終了行
a	1	6
a_0	5	7
b	1	2

表 2: 図 13 の変数の有効範囲

```
1:  int h(int a, int b){
2:      if(b > 0)
3:          return(h_T(a,b));
4:      else if( b < 0)
5:          return(h_F_T(a,b));
6:      else
7:          return(h_F_F);
8:  }
```

図 14: 条件分岐の分離の元の関数の変換例

```
1:  int h(int a, int b);
2:
3:  int main()
4:  {
4:    int (* func_p)() = h;
5:    func_p(4,2);
6:  }
```

図 15: 関数呼び出し元を見つけることが出来ない例

数内部には変数 b は存在しなくなり，入力 b が関数の動作に全く影響を与えないことを意味する．よって，関数の引数から b を消去可能であり，この関数 h_T を生成することにより効率的な再利用が可能になると判断される．一方，関数 h_F の変数 b は有効範囲は (1,2) となっており，2 行目では条件式に使用されている．関数 h_F に対して上述の分離を再帰的に適用すると，関数内部で変数 b を使用しない 2 つの関数を生成できる．上記の命名規則から，それらの関数の名前は h_F_T および h_F_F となる．これらの関数もまた，入力から引数 b を消去できる．そのあと，関数内部から消去された条件分岐と同じ文を呼び出し元に移植する．以上の変換を施すと，3.3 節の図 8 に示したように，変数 b の値の範囲によって異なる 3 種類の関数を呼び出すプログラムになる．

図 8 では省略しているが，変換を施したあとも h という名前の関数は存在するようにしている．変換完了後の関数 h のコードを，図 14 に示す．この関数は，第 2 引数 b の値の範囲によって，異なる 3 種類の関数を呼び出すプログラムであり，条件分岐や呼び出す関数は，関数 h の呼び出し元に施した変換と対応している．このような形で関数 h を存在させる理由を，次項で説明する．

4.3.5 関数の呼び出し元が不明な場合への対応

基本的に関数の呼び出し元には，呼び出している関数の名前が書いてあると考えられる．しかし，それに当てはまらないケースが存在する．関数ポインタを用いた関数呼び出しがそれに当たる．図 15 にプログラム例を示す．図 15 のプログラムでは，main 関数で関数ポインタ func_p が宣言されており，その func_p に関数 h が関連付けられている．この関数 h に，前項で示したような呼び出し元の変換を必要とする変換が行われたとする．プログラム中には関数 h を呼び出しているような記述は見られないため，呼び出し元の変換は行われない．しかし実際には 5 行目で関数ポインタを介するこ

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	
世代数	2000	
その他	default	値

表 3: シミュレーション時のパラメータ

とで関数 h は呼び出されており，もし仮に関数 h の定義が変更されたり消去されたりした場合，プログラムの動作やコンパイル結果に支障をきたす．これを防ぐため，本実装では図 14 に示す関数 h のような，条件分岐によって呼び出す関数を変更するだけの関数を生成している．これにより，関数呼び出し元を解析することが不可能であったとしても，関数呼び出し元の変換を必要とする変換を，プログラムの動作に支障をきたさず行うことが出来る．

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．今回は，3.2 節で提案し 4.3.3 で実装概要を示した，関数入力からの不要なアドレス情報の削除のみ実装を行い評価した．

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[2] 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 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 3 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．これ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の措置である．

5.2 結果

GENEsYs による評価結果を図 16 に示す．縦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，上は変換前のものを，下は変換後のものを表す．横軸は実行時間を示しており，各評価関数について，変換前のプログラムの実行時間を 1 として正規化している．また，いずれも実行時には計算再利用が行われている．

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．変換を行っていないプログラムと比べ，提案手法による変換を行ったプログラムでは，実行サイクル数が平均 4.6%，評価関数 F_11 において最大 53.6%削減できた．

5.3 考察

提案手法により，プログラムの高速化が実現していることが確認出来る．これは，この高速化は，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を，計算再

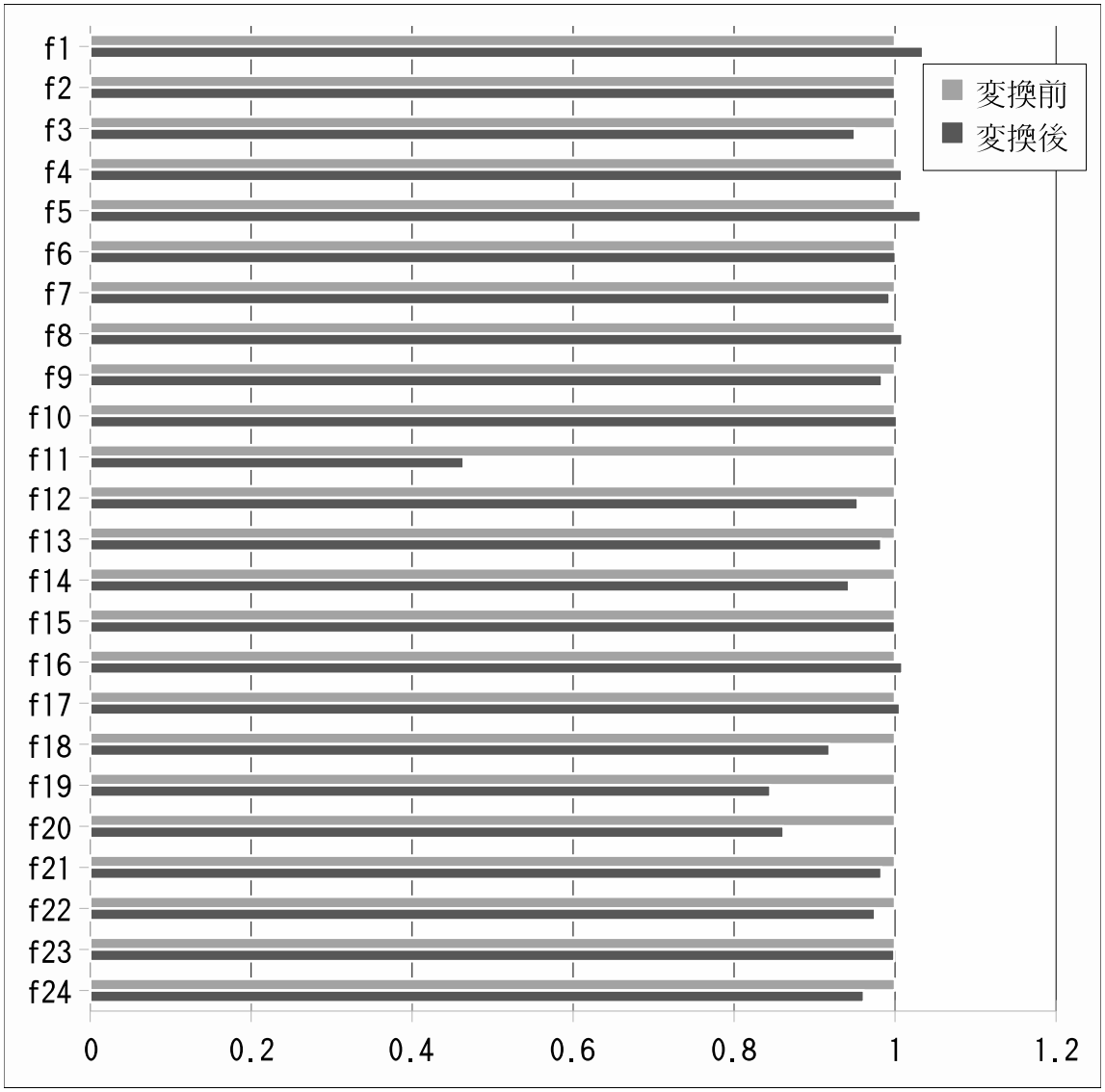

図 16: GENEsYs の実行時間

利用による実行時間の削減が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，平均で 1.4%，評価関数 F_01 で最大 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．また，

実行サイクル数に大きな変化は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が提案手法により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

本論文では，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 をベンチマークを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，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．

謝辞

本研究のために，多大なご尽力，ご協力を頂き，熱心なご指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授，齋藤彰一准教授，松井俊浩助教に深く感謝します．また，本研究の関して多くの助言，協力をくださった松尾・津邑研究室の方々に深く感謝します．中でも，自動メモ化プロセッサに関する研究を行っている同研究室所属の島崎裕介氏，高木伴彰氏には本研究に関する多大な助言を頂き，また相談に乗って頂きました．ここに，深く感謝の意を表します．

参考文献

[1] Tsumura, T., Suzuki, I., Ikeuchi, Y., Matsuo, H., Nakashima, H. and Nakashima, Y.: Design and Evaluation of an Auto-Memoization Processor, *Proc. of Parallel and Distributed Computing and Networks*, pp. 245–250 (2007).

[2] Bäck, T.: GENEsYs 1.0. Software distribution and installation notes (1992).

[3] 鈴木郁真, 池内康樹, 津邑公暁, 中島康彦, 中島浩: 再利用による GA の高速化手法, 情報処理学会論文誌：コンピューティングシステム, Vol. 46, No. SIG 16(ACS 12), pp. 129–143 (2005).