# 数値計算適応型計算機のアーキテクチャ

# AN ADAPTIVE COMPUTER FOR NUMERICAL CALCULATION

## —精度落ち自動防止機構を中心として—

坂村　健　　中野光一　　加藤芳夫　　相磯秀夫
Ken SAKAMURA　Koichi NAKANO　Yoshio KATO　Hideo AISO
慶應義塾大学工学部電気工学科
Department of electrical engineering Keio University

ABSTRACT This paper deals with an automatic recovery mechanism designed to prevent the occurrence of subtract errors at computer architecture level. This mechanism always detects the occurrence of subtract errors whenever addition or subtraction is performed. If a subtract error is detected, backtracking to a certain location is carried out, and from there recomputing is performed in order to realize the required precision. An adaptive algorithm required for this mechanism is described in detail. An experimental adaptive computer was implemented on a Burroughs B1700. The evaluation of the experiments proves that the inherent possibilities of dynamic microprogramming play a very important role in realizing this mechanism, and that the proposed mechanism will be useful for the development of future high-level computer systems.

## 1．はじめに

最初の電子計算機，ENIACの誕生以来，三十年以上の年月が過ぎた。その間計算機技術は，すばらしい進歩を遂げた。たとえば回路のスピードと集積度は最近のLSI技術によって驚くべきほど増進し，新しいアルゴリズムが発明され，より良いソフトウェアの方法論およびプログラミング言語が提案され，ソフトウェアのアプリケーションはより洗練されている。しかし，計算機アーキテクチャレベルで，従来の計算機システムについては，ほとんど重要な進歩がみられない。

現在の計算機システムのほとんどは，フォンノイマンの概念を基にしており，それらは著しく第一世代のものに類似している。計算機での演算は，二進表現が用いられ固定長のデータの上で行なわれている。

固定長データでの二進演算はハードウェアを安価にし，高速なスピードなど多くの利点を持つが丸め誤差，精度落ちなどの重大な問題を生じさせ，それらはしばしばその利点を打消してしまう。特に，精度落ちをプログラムの実行以前に予測することは非常にむづかしい。これは，従来の計算機は精度落ちに対してフォールトトレラントではなく，計算機アーキテクチャレベルでシステムエラーを生じているといえる。それらの現象を既存のシステムで防ごうとするならば，プログラムを書く際に非常に注意を払わなければならない。

この論文では，精度落ちの現象を解析し，フォールトトレラントコンピューティング，すなわち，精度落ちの発生を防止する自動復活アルゴリズムを達成するための方法を論じる。また，この考えを既存の計算機システムに実現することを提案しこの中でダイナミックマイクロプログラミング技術が非常に重要な役割を果すことを示す[(1)]。提案した機構の効果を証明するために，バロースB1700計算機システム[(2)]の上で実験を行なった。この実験の評価およびオーバーヘッドによって，提案した考えが将来の計算機アーキテクチャに根本的に重要であるという結論が導びかれる。

## 2．数値計算におけるフォールトトレラントコンピューティング

プログラマが遭遇する最もやっかいな問題の1つに精度落ちがある。計算機では，演算はたいてい固定長データを取扱うことを基礎としているので，プログラムの実行中に，精度落ちや丸め誤差などの重大な誤りが生成される。特に，精度落ちは突然発生し，予測することは非常にむづかしい。これは，この問題に対してはどんな理論的な解析も適用されないことを意味する。精度落ちによる上位桁の消失はしばしば誤った結果をもたらす。たとえば，ガウスジョルダン法を用いる時，ある行の要素が一せいに有効桁を失うことがある。もし，桁を失った要素をピボットとして掃出しが行なわれると結果は誤ったものになる。このような好ましくない影響を抑えるために，絶対値が最大の要素をピボットとして掃出しを行なうようにプログラムを書かなければならない。もう1つの例として，我々は，二次方程式を解くための有名なアルゴリズムを，そのまま使用すると，まちがった結果を得ることがあるという事を知っている[3]。いいかえれば，ユーザーは，つねに，いくつかの「技法」を用いてプログラムを書くことを要求されている。それらのプログラムは，簡単なアルゴリズムを用いた正直なものではなく，多くの非本質的なステップを含んだ複雑なものである。

計算機が計算分野で基本的な役割を果しているにもかかわらず，数値計算に関して計算機アーキテクチャにおいて目立った進歩がないことは注目すべきである。

精度落ちの発生という問題を計算機アーキテクチャで解決することは，ソフトウェアの生産性の向上に本質的に貢献する。

## 3．精度落ちを防ぐための自動復活機構

### 3－1　可能なアプローチ

精度落ちを復活させるには，3つの方法が考えられる。

(i)アルゴリズムはそのままで実行する計算の順序を変える。(ii)アルゴリズムを改良する。(iii)プログラムの実行中に精度落ちを検出後，より大きい精度で再計算を行なう。

既存の計算機では，演算は，単精度，倍精度のような段階的に指定される有限精度に基づいている。可変精度の計算ができる機構も，精度落ちを検出する機構も持たない。従って，(i)，(ii)の方法がユーザープログラムのレベルでのみ適用されている。これらの2つの方法をシステムレベルで，復活機構を実現するために適用するためには，計算機システムの設計が進んだアルゴリズムに基づく必要がある。さらに，この機構を現在のハードウェアおよびソフトウェア技術で実現した場合にオーバーヘッドは非常に大きくなる。そこで，(iii)の方法が一番良いと思われる。

### 3－2　システムレベルでの問題解決

精度落ちの発生を検出する機構を工夫すること，再計算を開始するための位置の決定，および開始位置へバックトラックするために計算機が準備する方法を見つけることが重要となる。

**①　基　本　原　理**

**(i)　精度落ちの検出**

加減算の結果の桁数が，あらかじめ決められた精度より小さいとき精度落ちが発生したとする。

**(ii)　再計算の開始位置**

開始位置は，乗除算の行なわれた地点とする。それらの結果が，精度落ちした加減算の被演算子として用いられている。

**(iii)　再計算**

再計算の手続きは，要求される精度が得られるまで繰返される。

**(iv)　バックトラックの準備**

再計算を正しく行なうために，精度落ちが発生した加減算と開始位置の間で，変更された変数は，元の値が戻されていなければならない。従って，変更される変数をプログラムの実行中に前もって保存する。

再計算の開始位置をこのように選んだのは以下のような理由である。

- 精度落ちが発生した加減算のオペランドが，それ以前により精度良く定義されていないと精度落ちが復帰できない。
- 計算機の演算の性質から，加減算あるいは単なる代入演算では，最下位桁の右側に有効な桁を補うことができない。
- 乗除算の結果の桁数は，元のオペランドの桁数

より長くなることができる。
上述の機構が実現されるためには，計算機システムは任意長のデータを扱い，任意精度の演算をサポートする基本アーキテクチャを持たなければならない。

② 静 的 解 析

再計算を開始する演算の位置すなわち再開始位置は，後述するように静的解析によって決定される。再計算を正しく行なうためにプログラムの実行中に保存すべき変数もまた，静的解析によって見い出される。プログラム内の各加減算に対し，再計算の開始位置の候補が決定される。しかし，精度落ちを生じる加減算の2つのオペランドのうち，少なくとも一方が乗除算で定義されていなければ，再計算を行なうことはできない。

オペランドを定義する乗除算がいくつかあるときは，再計算の開始位置として2つのオペランドの各々に対して最後の乗除算が選ばれる。また，プログラム中の各命令が図1に示すグラフの中で節で表わされるならば，上述の基本原理は，(a), (b)および(e)の場合はそのまま適用できる。もし，求めた演算が(c), (d)の場合のようなループ内にある場合は，再開始位置は，ループの入口の位置に変更される。この場合は，たとえ，計算機が1回だけ乗除算へバックトラックし，そこから再計算を開始しても丸め誤差がループ内で累積し，信頼ある結果が得られない。

再開始位置が決定された後，基本原理に従って実行中に保存すべき変数が静的に定義される。しかし，プログラム中にループがあるときは，ループ内で定義され，かつ参照されるすべての変数を保存することとし，そのループ内の演算に対しては基本原理による解析を行なわない。

③ 変数の保存と再計算

静的解析は，再計算の再開始位置と保存すべき変数を決定する。これらのデータに基づいて，計算機は命令の実行時に再計算のために必要な情報を保存しなければならない。すなわち，ある変数が乗除算によって，陰にあるいは陽に定義されるならば，定義された位置とその時の環境が実行中に保存される。また，保存すべき変数は，関係する命令あるいは，ループの入口の命令が実行される時に保存される。そして，精度落ちが検出されると，再開始位置が保存された情報から選ばれ，

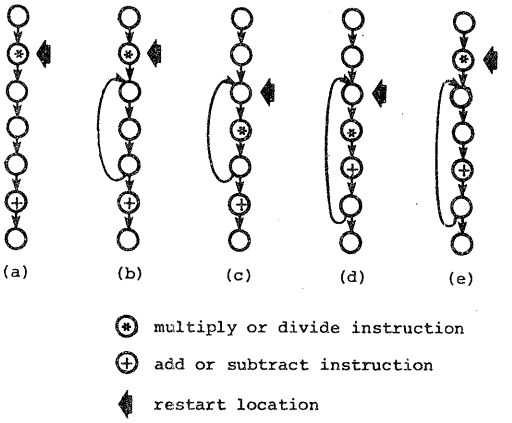



図1．プログラムのグラフモデル

変数が再格納され，再開始位置へバックトラックが行なわれる。演算桁数は必要な桁数だけ増され，再計算が開始される。

### 3－3 アルゴリズムの詳細な記述

静的な解析手順を以下に示す。動的な手順を図2，3に示す。

DEFINITIONS

□ Control flow graph of the program $G$

$G = (s,N,E)$

where, $s$ is the entry node,

$N$ is the set of nodes representing a sequence of program instructions,

$E$ is the set of edges of orderd nodes representing the flow of control.

□ Path $P(n_1, n_K)$

A path from $n_1$ to $n_K$ is an ordered sequence of nodes $(n_1,n_2,\ldots,n_K)$ and their connecting edges in which each $n_i$ is an immediate predecessor of $n_{i+1}$.

□ $ND(P)$

$ND(P)$ is the set of nodes contained in the path $P$.

□ $mod(\alpha,n_1,n_K,P) = 1$ iff there is a node $n_k$ for some k with $1 \leq k \leq K-1$ which can modify the value in $\alpha$ when control flows along the path $P$.

□ $use(\alpha, n_1, n_K, P)$ = 1 iff there is a node $n_k$ for some k with $1 \leq k \leq K$ which can use the value in $\alpha$ when control flows along the path $P$.

□ $DEF(\alpha, P)$
A set of nodes representing multiply or divide instructions which can define the value in $\alpha$ explicitly or implicitly when control flows along the path $P$.

□ $pre(P, A)$
A node which is the predecessor of any other node in $A$, where $A$ is a subset of $ND(P)$.

□ $suc(P, A)$
A node which is the successor of any other node in $A$, where $A$ is a subset of $ND(P)$.

An algorithm to find the restart location for $n_k$ along the path $P$.

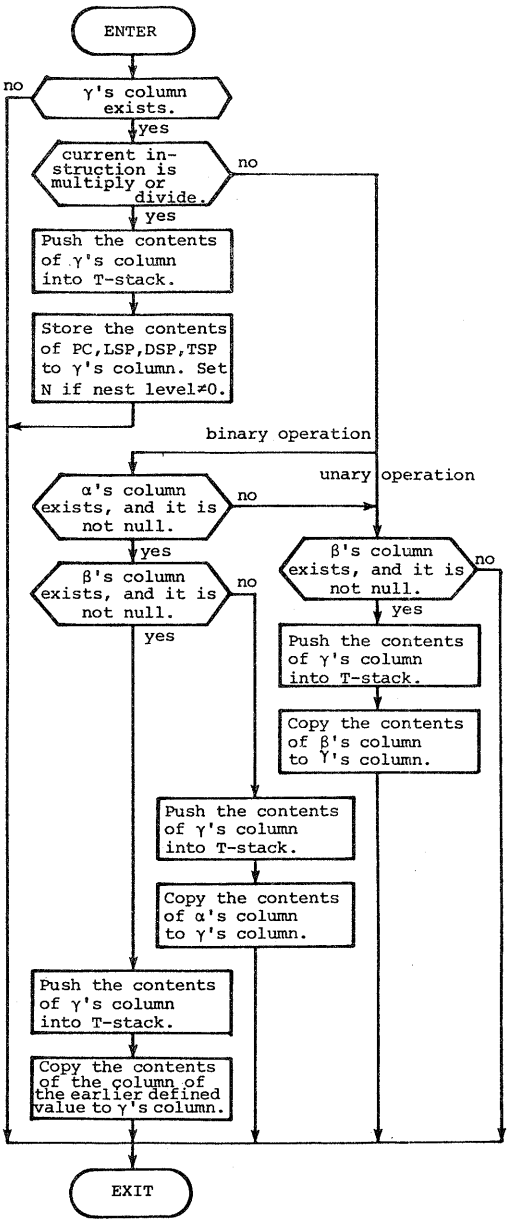
$n_K$ : a node representing an add or subtract instruction whose source operands are $\alpha$ and $\beta$, and destination operand is $\gamma$.
$P$ : a path from $s$ to $n_k$.

```
begin
 let n_R be
       pre(P,(suc(P,DEF(α,P)) ∪ suc(P,DEF(β,P))))
 if n_R does not exist
  then no restart locations exist along P.
  else if nest level of n_R is 0
        then n_R is the restart location.
        else if nest level of n_k is 0
              then the entrance of the loop which
                   contains n_R and whose nest
                   level is 1 is the restart loca-
                   tion.
              else if nest level of n_k is less
                      than the nest level of n_R
                   then the entrance of the loop
                        which contains n_R and
                        whose nest level is the
                        same as that of n_k is the
                        restart location.
                   else the entrance of the loop
                        which contains n_R is the
                        restart location.
end;
```

An algorithm to find variables to be saved.

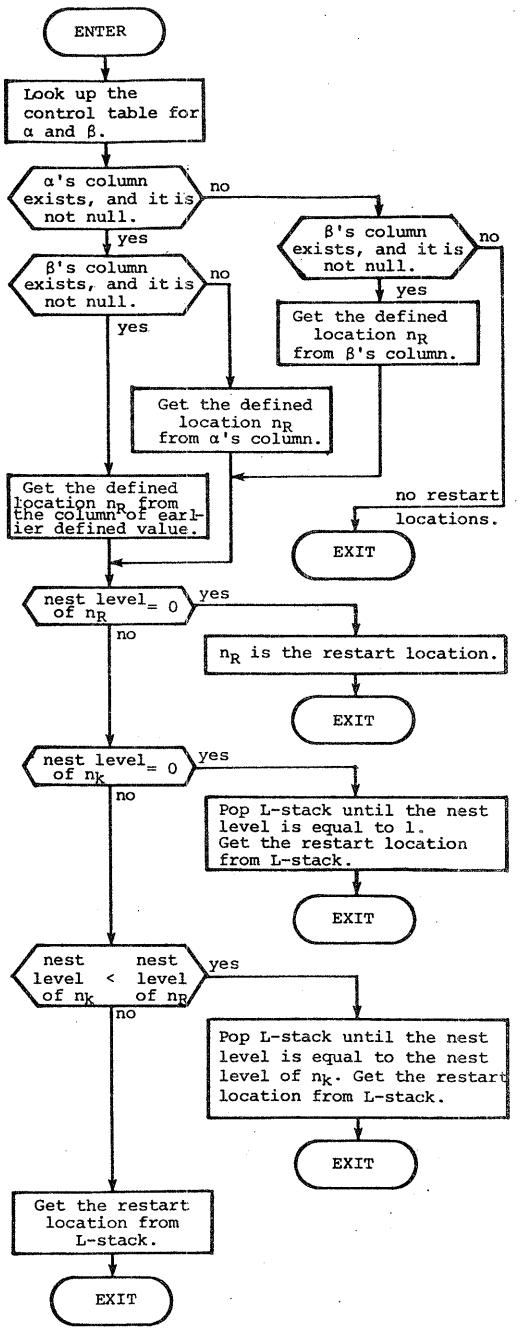
$n_k$ : a node representing an add or subtract instruction
$n_L$ : a restart location for $n_k$.
$\gamma_i$ : a variable which can be modified by i-th instruction

```
begin
 for all the paths from n_L to n_k do
  begin
   n_i ← n_L
   repeat
    if n_i is the entrance of the loop L then
     begin
      The variables which can be modified and be
      used in L are to be saved.
      if n_i is in L then the analysis is finished.
      n_i ← the immediate successor of the exit
           of L.
     end else
     begin
      if γ_i can be modified at n_i then
       begin
        if use(γ_i,n_L,n_i,P)=1 and mod(γ_i,n_L,n_i,P)=0
         then γ_i is to be saved.
       end;
      n_i ← the immediate successor of n_i
     end;
    until n_i = n_k
   end;
end;
```




α,β: source operands of the current instruction
γ : destination operand of the current instruction

図2．バックトラックの準備(実行時)




α,β: source operands of the current instruction
γ : destination operand of the current instruction

図3．再開始位置の発見(実行時)

## 4．精度落ちの自動復活機構を備えた仮想計算機のアーキテクチャ

自動復活機構を使って，精度落ちの発生を防ぐために命令の実行中に演算桁数を制御できる機能を持たなければならない。この機能は，マイクロプログラミング技術を利用して可能であるが，特にダイナミックマイクロプログラミングが重要な役割を果す。

マイクロプログラム制御の計算機では，演算桁数が，レジデュアル制御の一つとして指定されることがある。たとえば，B1700のコントロールパラレルレングスレジスタは，ファンクションボックスのデータ長を指定する。演算桁数は，レジデュアル制御の情報を新しい値に変えることによって制御することができる。

この場合，プログラムの実行中にできなければならない。ダイナミックマイクロプログラミング技術が，プログラムの実行中に，マイクロプログラムのシーケンスを全く変えることなく，レジデュアル制御情報の内容を変更することを可能にする。これは，ダイナミックマイクロプログラミングの本来の可能性によって計算機の動作をダイナミックに変えることを意味する。

### 4－1　データ構造

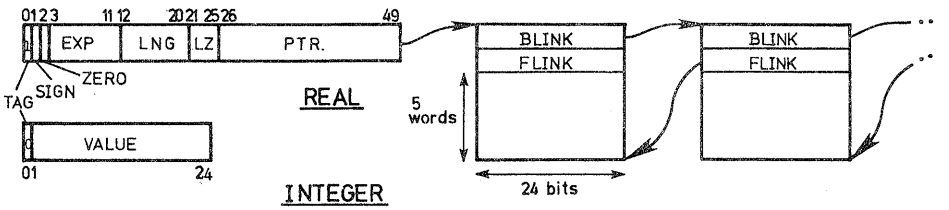
自動復活機構を用いて精度落ちを防ぐために，演算が可変精度で行なわれることが要求される。従って，実数データは，データ長を示す部分を持つ。データ長は演算中に変化するので，仮数部の位置は固定できない。そこで，仮数部は指数部と分離される。指数部と仮数部は図4に示すようなリンクドブロックによって管理される[4]。長い桁数のデータの仮数部を正規化するには時間がかかるので，仮数部のリーディングゼロの数を持たせ，正規化の必要をなくした。さらに，値のゼロを示すタグを設け，演算のスピードアップを図っている。1ビットのタグによって示された整数型のデータも用意した。

### 4－2　仮想計算機の動作と命令セット

命令の実行に先だって，そのデスティネーションオペランドが保存されるべきかどうかチェックされる。このチェックは静的解析によって，オペレーションコード中にすでにセットされている情報によって成される。

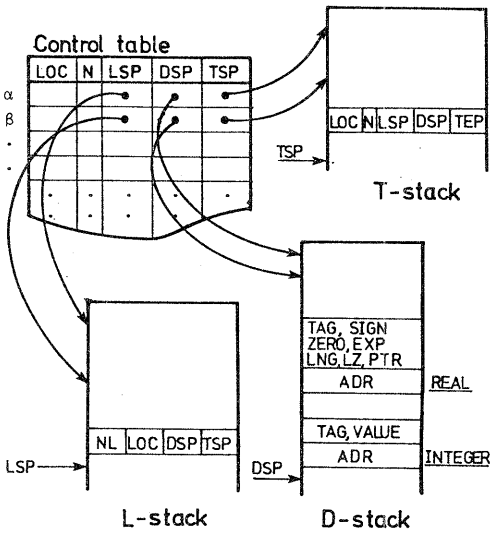
仮想計算機は，図5に示すように再計算のためにデータスタック(Dスタック)，テーブルスタック(Tスタック)，ループスタック(Lスタック)と呼ばれる3つのスタックと1つの制御テーブルを持つ。保存すべき変数はDスタックにプッシュされる。制御テーブルに関する前のデータはTスタックにプッシュされる。ループの入口の位置および，ループに関する情報はLスタックにプッシュされる。制御テーブルは，変数が定義された位置とその位置に関する情報の軌跡を保存する。

実際には，乗除算が実行されたとき，プログラムカウンタおよび，D，L，Tスタックの各スタックポインタが制御テーブルのデスティネーションオペランドの欄に記憶される。現在のネストレベルがゼロならば制御テーブルのNフィールドがゼロにリセットされる。命令が加減算あるいは代




TAG : type indicator, VALUE: 2's complement, SIGN: sign of mantissa
ZERO : zero indicator, LNG: word length of mantissa, EXP: exponent based $2^{24}$
LZ : number of leading zero words, PTR.: pointer to the first block of mantissa
BLINK : pointer to backward block, FLINK: pointer to forward block

図4．データ構造




LOC : location of multiplication or division instruction or location of loop entrance

N : loop indicator (=0 if the nest level of LOC =0)

LSP: a stack pointer for the L-stack

DSP: a stack pointer for the D-stack

TSP: a stack pointer for the T-stack

NL : nestlevel of the loop

ADR: address of the saved variable

TEP: a pointer for the table entry

図5．再計算のための表およびスタック

入演算ならば，制御テーブルのソースオペランドの欄が，デスティネーションオペランドの欄に記憶される。精度落ちが検出されると，制御テーブルのソースオペランドの欄が，再計算の再開始位置を得るために調べられる。もし，ソースオペランドの欄の内容が空ならば，再計算は不可能である。なぜならば，ソースオペランドの値は乗除算によって定義されなかったからである。もし，内容が空でないなら，制御テーブルに示された位置まで，Dスタックおよび Tスタックの内容が戻される。もし，Nフィールドがゼロでないならば，再計算の再開始位置および，Dスタック，Tスタックの内容を戻す位置は，制御テーブルを使うかわりに，Lスタックに保存された情報から得る。

バックトラックが成された後，再開始位置がプログラムカウンタにセットされる。演算桁数が，精度落ちした桁数だけ増やされ，再計算が開始される。

表1に，仮想計算機の典型的な命令を示す。「ENTER」，「EXIT」のような，それぞれ，ループの入出口で用いられる特別な命令がある。

## 5．評　　価

### 5－1　バロースB1700での実験

前章で述べたように計算機アーキテクチャレベルで，精度落ちの発生を防止できるように設計された適応型計算機の仮想計算機を，バロースB1700計算機システムでエミュレートした。

表1．仮想計算機の主な命令

| NAME | OPERANDS | MEANING |
|---|---|---|
| ADD | A,B,C | C ← A+B, and check the subtract error. |
| SUB | A,B,C | C ← A-B, and check the subtract error. |
| MULT | A,B,C | C ← A×B |
| DIV | A,B,C | C ← A/B |
| MOD | A,B,C | C ← mod(a,B) |
| COMP | A,B | Compare A with B. |
| MOVE | A,B | B ← A |
| NEG | A,B | B ← -A |
| ABS | A,B | B ← \|A\| |
| CALL | S | Call subroutine S. |
| RTN | | Return from subroutine. |
| BRANCH | CC,L | Condition branch, CC is the condition. |
| INCBR | I,J,K,L | Increment & branch I←I+J, branch if I≤K. |
| CASE | N,M,$L_1$,$L_2$,...,$L_M$ | Branch to $L_N$, if N≤M. |
| ENTER | L | Enter the loop, and save variables(L is the save list) |
| EXIT | | Exit the loop. |

表2は仮想計算機をB1700で実現した場合のマイクロプログラムの大きさを示している。大きさは約4K語であり，B1700上のFORTRANマシンなどとほぼ同じ大きさである。

提案した適応型計算機の性能を評価するために，比較対象として，バロース社がFORTRAN Sーランゲージインタプリタと呼ぶFORTRANマシンを使う。

表2．マイクロプログラムの大きさ

| | Variable precision arithmetic | Recomputation control | others | total |
|---|---|---|---|---|
| MICRO INSTRUCTION COUNT | 1,763 | 319 | 1,521 | 3,603 |

## 5－2　ベンチマーク問題

以下のベンチマーク問題が考えられた。(i)二次方程式の解法。(ii)正弦関数のテイラー級数による計算。(iii)係数行列にヒルベルト行列を持つ連立方程式の解法。

これらの問題を解くときには，精度落ちが生じる。これらの問題は，適応型計算機のためにFORTRANの高級言語からコンパイルされた。B1700 FORTRANマシンでは，問題はバロース社が提供するFORTRANで記述し，FORTRANコンパイラでオブジェクトコードにされた。

表3は，ソースコード，オブジェクトコードおよび適応型計算機で必要なデータサイズを示している。適応型計算機では，精度落ちの発生を防ぐような工夫が必要ないので，ソースコードの大きさはB1700のFORTRANに比べて半分以下である。このためにオブジェクトコードもより小さくなっている。ソースプログラムレベルでのステップ数の減少はソフトウェアの生産性の改善を意味する。

18桁以下のオペランドを持つ問題では，B1700 FORTRANマシンのほうが適応型計算機よりデータサイズがより小さいが，もし，オペランドの長さが18桁以上になると，データサイズに大きな違いはなくなる。

## 5－3　実行時間と誤差

表4に実験の結果を示す。適応型計算機では，初期のオペランドの精度が十分ならば，まえもって決めた精度が得られている。しかし，従来のFORTRANマシンでは，誤差が生じている。もし，精度落ち防止のための工夫がとられていなければ，さらに大きな誤差が生じる。

表4．実行時間と誤差

Execution time (sec.)

| | A | B | | C | |
|---|---|---|---|---|---|
| | | a | b | a | b |
| Adaptive computer | 0.0299 | 0.0128 | 0.0289 | 0.0413 | 0.3206 |
| B1700 FORTRAN machine | 0.0316 | 0.0430 | 75.7 | 0.1433 | 0.9633 |
| ratio* | 1:1.06 | 1:3.36 | 1:2619 | 1:3.47 | 1:3.00 |

*Adaptive computer : B1700 FORTRAN machine

Relative errors

| | A | B | | C | |
|---|---|---|---|---|---|
| | | a | b | a | b |
| Adaptive machine | 0 | 0 | 0 | 0 | $0.19\times10^{-11}$ |
| B1700 FORTRAN machine | 0 | $0.36\times10^{-15}$ | $0.63\times10^{-25}$ | $0.22\times10^{-14}$ | $0.26\times10^{-7}$ |

表3．ベンチマークプログラムの大きさとデータの大きさ

| | SIZE OF SOURCE PROGRAM (steps) | | | | SIZE OF OBJECT PROGRAM (bytes) | | | | SIZE OF DATA (bytes) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | B | | C | A | B | | C | A | B | | C | |
| | | a | b | | | a | b | | | a | b | a | b |
| Adaptive computer | 3 | 17 | 17 | 10 | 73 | 232 | 232 | 140 | 186 | 765 | 1,498 | 1,659 | 11,854 |
| B1700 FORTRAN machine | 6 | 17 | 380 | 35 | 99 | 248 | 4,453 | 729 | 54 | 72 | 1,206 | 243 | 729 |

A : quadratic equation(18 digits)
B-a: Taylor series (18 digits)
B-b: Taylor series (32 digits)
C-a: simultaneous eq. (4-th,18 digits)
C-b: simultaneous eq. (8-th,18 digits)

実行時間としては，適応型計算機での全処理時間が，バックトラック，再計算およびその準備のオーバーヘッド時間を含んでいるにもかかわらず，FORTRANマシンより少ない。ユーザーがソースプログラムレベルで，可変精度にしないと，B1700 FORTRANマシンでは，18桁以上のオペランドを扱うことができないことに注目すべきである。たとえば，正弦関数のテーラー級数を32桁で計算しようとすると，B1700での実行時間は，適応型計算機に比べて2500倍以上の時間が要求される。これは，いかに非本質的なステップで多くの時間が，従来の計算機では費やされているかを示している。

表5は，再計算のオーバーヘッドを示している。オーバーヘッドは，全時間の10%以下である。実験のほとんどは，1回のバックトラックしか生じなかった。

しかし，二次方程式とテーラー級数の計算では予期したほど多くのスペースを必要としなかったが，連立方程式の解法には，行列の要素が増すにつれて非常に多くのスペースが必要になった。

したがって，適応型計算機は，代数方程式を解くような単一変数の計算により適している。適応型計算機は，実時間の環境で正確な結果を計算するために用いるときその価値を発揮するだろう。

表5．全実行時間に対するバックトラックおよび準備のオーバーヘッド時間

| | A | B | | C | |
|---|---|---|---|---|---|
| | | a | b | a | b |
| $\frac{\text{overhead time}}{\text{total time}}$(%) | 1.75 | 9.48 | 5.56 | 5.71 | 5.30 |

表6．再計算のためのスタック使用量 (bytes)

| | A | B | | C | |
|---|---|---|---|---|---|
| | | a | b | a | b |
| D-stack | 0 | 35 | 35 | 2,214 | 23,522 |
| T-stack | 87 | 464 | 638 | 1,015 | 7,830 |
| L-stack | 10 | 10 | 10 | 210 | 730 |

## 6．結　　論

この論文は，システムレベルで精度落ちの発生を防止する適応型計算機のアーキテクチャを述べた。提案した適応型計算機は，予測が非常に難しい精度落ちを防ぐための自動復活機構を持つ。

精度落ちの特長を解析し，これを防ぐための適応アルゴリズムを提案した。

科学計算向き計算機のＩＳＰレベルの記述を行なった。実験的な適応型計算機を，バロースB1700のダイナミックマイクロプログラムを用いて実現した。実験によって，実行時間，オブジェクトコードの大きさ，使い易さに関する限り，適応型計算機が従来の計算機より優れていることが示された。

ユーザーは，適応型計算機では，精度落ちを心配する必要がなく，不要なアルゴリズムの改良や，複雑なプログラムを書くのに時間をかける必要もない。

この論文では取り上げなかった丸め誤差の解析や，学習機構の実現は別の論文で論じる。学習マシンの概念[5]がバックトラックの繰返し数を減少させ，結果としてオーバーヘッドの減少をもたらす可能性がある。

この論文で述べたアルゴリズムが，半導体や，計算機技術の目をみはる進歩によって可能になることはいうまでないことである。しかし，適応アルゴリズムの導入と，計算機アーキテクチャの適応性は，将来の高機能計算機システムに非常に重要なものになると信じる。


## 謝　　辞

本研究を遂行するに当り，バロース株式会社には全面的御支援をいただいた。バロース株式会社，特に公共企業支店の水沢課長をはじめとするスタッフの方々の御援助に厚く感謝する次第である。


## 文　　献


(1) K. SAKAMURA, et al.: "A DEBUGGING MACHINE—AN APPROACH TO AN ADAPTIVE COMPUTER", Proc. IFIP 1977, pp.23-28.

(2) W. T. WILNER: "Design of the Burroughs B1700", AFIP Conf. Proc. vol.41,1972 FJCC, pp. 489-497.

(3) G. E. FORSYTHE: "Pitfalls in Computation, or why a Math Book Isn't Enough", The American Mathematics Monthly, vol.77, 1970, pp.931-956.

(4) D. E. KUNUTH: "THE ART OF COMPUTER PROGRAMMING, 1, Fundamental Algorithms", ADDISON-WESLEY PUBLISHING COMPANY, 1975, pp.251.

(5) 坂村, 相磯: "A learning machine", 情報処理学会計算機アーキテクチャ研究会29－3, 1977－11.