複数の浮動小数点表現法を処理するシステム環境の設計と実現†

中原雅彦††* 早川栄一†† 岡野裕之††**
並木美太郎†† 高橋延匡††

本論文では，複数の浮動小数点表現法を処理するシステムプログラム体系の実現を通して，ペリフェラルプロセッサを効率よくシステムに接続するための一手法について提案する．従来ソフトウェアで実行していたものをハードウェア化した際に問題となるのは，ハードウェア増設に伴うコンテキストの拡大とその管理方法，ユーザへの提供形態である．この問題に対し，われわれの研究室で開発しているOS/omicronの浮動小数点演算機能をハードウェア化した際に，「拡張コンテキスト」の機能を実現した．これは，管理対象ハードウェアのコンテキスト退避が必要な場合だけスイッチングを行う機構で，ハードウェアの追加に対するコンテキストスイッチのオーバヘッドとメモリ消費量を軽減している．また，複数の浮動小数点演算プロセッサを一つの実行形式で扱える機能を実現するため，プログラムの実行直前に評価される情報として，言語処理系に「型情報」の機能を実現した．これにより，ハードウェア特有の情報を言語処理系内に抽象化した形で実現でき，ソフトウェアからハードウェアへの移行が言語処理系の変更なしに行うことができる．

1. はじめに

近年のLSI技術の発達で，従来ソフトウェアで行っていた処理をハードウェア化することが比較的容易になってきた．しかし，ハードウェア化を行った場合，そのハードウェア資源をどのように管理し，システムに組み込むかが重要な問題である．また，それに伴ってシステムプログラムの変更が必要になる場合もある．特に，ハードウェア資源の管理には，OSのサポートが不可欠であり，その対応は容易でない．そのハードウェアの処理性能やアクセス頻度，コンテキストの大きさなどの要因でシステム性能に与える影響が異なるためである．

われわれの研究室では，新しい浮動小数点表現法の評価および実現方式の研究を目的として，OS/omicron[3]上にIEEE規格の方式[2]，URR[1]など複数の浮動小数点表現法の開発を行ってきた[3),5]．そのプロトタイプとして1986年にOS/omicronの開発システムとして使用していたCP/M-68K上でOS/omicron用言語Cコンパイラ CAT (C compiler developed at Tokyo University of Agriculture and Technology)に複数の浮動小数点表現法を実現した[5]．この実現では，浮動小数点演算にソフトウェアルーチンを使用し，浮動小数点表現法の選択に次の二つの方式を用いた．

（1） コンパイル時に表現法の選択を行う方法
（2） 実行時に表現法の選択を行う方法

特に(2)の方法は，一つの実行形式で複数通りの浮動小数点表現法が適用可能という特徴をもっており，定数をプログラム実行中に毎回変換し，演算はエミュレータ・トラップによる方式（ソフトウェア割込み）で実現していた．

今回われわれはこの環境をOS/omicron上に実現すると共に，浮動小数点演算のハードウェア化を行った．しかし，上記の機能をそのままOS/omicronの環境に移行するにはいくつかの問題が生じた．

第1点は，冒頭に述べたハードウェア管理の問題である．OS/omicronでは，複数の浮動小数点表現をサポートする立場から，複数の浮動小数点プロセッサをシステム内に持つことになり，さらに，将来にわたる拡張についても考慮しておかなければならない．コンテキストの拡大にどう対処するか，プロセッサの差異をどう扱うか，ユーザへの提供形態をどうするか，といった問題もある．

第2点は，OS/omicronがOS自身もタスクとして扱われるため[6]，エミュレータ・トラップ方式で演算を行うと，その度にコンテキストスイッチを伴ってしまうことである．これは，演算実行時間に対して大き

† Design and Implementation of System Environment for Multiple Schemes of Floating Point Representation by MASAHIKO NAKAHARA, EIICHI HAYAKAWA, HIROYUKI OKANO, MITAROU NAMIKI and NOBUMASA TAKAHASHI (Department of Computer Science, Faculty of Technology, Tokyo University of Agriculture and Technology).
†† 東京農工大学工学部電子情報工学科
* 現在 (株)日立製作所 Hitachi Ltd.
** 現在 日本IBM IBM Japan

なオーバヘッドとなる.

このため，CAT の浮動小数点演算機能に見直しをはかり，浮動小数点演算を強化したCAT/N (CAT for Numerical computation) を開発し，OS/omicron の機能拡張を行った．本論文では，OS/omicron における複数の浮動小数点表現の実行環境の実現を通して，システムの機能拡張方式について述べる.

2. OS/omicron が扱う浮動小数点表現

OS/omicron では，新しい浮動小数点表現法の比較・評価を行いたいと考え，次に示す浮動小数点表現法を実現している[5].

(1) URR[1]

オーバフロー・アンダフローの問題の解消を目的として考案された表現法である．また，表現形式の定義が語長に依存しない，すべてのビットパターンが異なる数値に対応するなどの特徴を持っている.

(2) IEEE 浮動小数点標準[2]

汎用大型機における互換性欠如の問題の反省から，IEEE が浮動小数点演算の規格化を目的として提案したものである．現在ワークステーション等中小型計算機を中心に広く採用されている．オーバフロー・アンダフロー後の処理の規定や，非数の概念の導入など，従来の浮動小数点表現法が持っていた問題を解決しようとしている意図がうかがえる.

3. URR のハードウェア化

OS/omicron が採用している浮動小数点表現法のうち，IEEE 規格の方式については i 80387, MC 68881 といったコプロセッサ，あるいは CPU 組込みという形でハードウェア化が進んでいる．しかし，URR についてはその動きはなかった．そこで，われわれは他の研究機関と共同で URR 演算のハードウェア化を試み，URR プロセッサを開発した[7].

URR プロセッサは，URR の実用化研究を目的とした，URR の演算を行うコプロセッサである．われわれは，これを利用したコードを生成する言語処理系を設計する立場から URR プロセッサの設計に加わった．次に URR プロセッサのアーキテクチャを示す.

(1) CPU とのインタフェース

CPU とのインタフェースには，実現上の問題からプログラム入出力方式を採用した．この方式は，URR プロセッサのすべてのレジスタを CPU のアドレス空間に割り当て，CPU からの命令発行によって URR プロセッサの起動，制御を行う.

(2) レジスタ構成

URR プロセッサのレジスタは，プログラム入出力方式との親和性からマルチレジスタ構成とした．データレジスタは 64 bit 長のものを 16 本備えている．URR プロセッサはこれらのレジスタのうちの任意の2本のレジスタ間でのみ演算を行うことができる.

(3) 命令セット

URR プロセッサの有する命令セットを表 1 に示す．この命令の中には，演算速度を考慮して，四則演算以外にも 2 の乗除算命令 (MUL 2, DIV 2)，比較命令 (CMP) 等を加えている．また，初等関数のうち，他の初等関数が容易に記述できる要素的な基本関数を選び，これを組込み命令とした.

(4) 丸めの機能

演算結果に対する丸めの機能は，ハードウェア仕様上の制約から RM (−∞ 方向への丸め) の機能しか有していない.

4. 言語 C コンパイラ CAT/N

浮動小数点演算の機能をハードウェア化しても，こ

表 1 URR プロセッサの組込み命令
Table 1 The instruction set of URR-processor.

命令名	命令の内容
SET0	定数 0 のロード
SET1	定数 1 のロード
MOV	データ転送
NEG	符号反転
ADD	加算
SUB	減算
MLT	乗算
DIV	除算
MLT2	2 を乗ずる
DIV2	2 で除する
REN	剰余
CMP	比較
MANT	指数と仮数の分離
SCAL	指数 (64 bit) と仮数の結合
SCALS	指数 (32 bit) と仮数の結合
CLRA	データレジスタの初期化
SQRT	平方根 ($\sqrt{x}$)
SIN	正弦関数 ($\sin(x+n\times\pi/2)$)
TAN	正接関数 ($\tan(x+n\times\pi/2)$)
ATAN	逆正接関数 ($\tan^{-1}x$)
COSH	双曲余弦関数 ($\cosh x$)
SINH	双曲正弦関数 ($\sinh x$)
TANH	双曲正接関数 ($\tanh x$)
ATANH	逆双曲正接関数 ($\tanh^{-1}x$)

れを高級言語でサポートしなければ実用的でない．特に現在の OS/omicron では，浮動小数点演算に URR プロセッサ，IEEE 規格方式の MC 68881 を利用できる環境が整っており，これらを効率よく使用できる機能が必要である．このため，OS/omicron 用言語Cコンパイラ CAT/N を開発した[8]．CAT/N は基本的には浮動小数点表現法の評価・研究を目的とするものであるが，それのみでなく，OS/omicron 上で動作するアプリケーションプログラムの記述にも使用される．このため，CAT/N では浮動小数点演算に対して，ユーザの使用目的に応じて自由に選択できる次の4種類のコード出力形式を実現した．

（1） 関数コール型

この方式は，浮動小数点演算プロセッサを有しないシステム上で実行されるアプリケーションプログラム用である．CAT/N は浮動小数点演算に対して関数呼出しと同等のコードを生成する．浮動小数点演算はソフトウェアで実行され，演算を行う関数はライブラリとして用意される．浮動小数点表現として，先に示した URR と IEEE 規格の方式の2種類が使用できる．浮動小数点表現の選択はコンパイル時に行う．

（2） URR プロセッサ対応型

この方式は，URR の高速演算を必要とするアプリケーションプログラム用である．CAT/N は浮動小数点演算に対して，基本的に URR プロセッサで演算を行うようにコードを生成する．命令コードは URR プロセッサへのアクセスコードをインライン展開する．この際，URR プロセッサの性質に合わせたオプティマイズを行っている．

なお，この URR プロセッサについては，ユーザへの提供形態が問題となり，新しい方式を考案した．これについては第5章で述べる．

（3） MC 68881 対応型

この方式は，IEEE 規格方式の高速演算を必要とするアプリケーションプログラム用である．CAT/N は浮動小数点演算に対して基本的に MC 68881 で演算を行うようにコードを生成する．命令コードは MC 68881 の命令をインライン展開する．

（4） 実行時選択型

この方式は，主として浮動小数点表現法の評価を目的とするもので，一つの実行形式で複数の浮動小数点表現法を扱えるという利点がある．従来のCATでは，この方式に対してエミュレータ・トラップ型と呼ばれるコード生成方式を用いて行っていたが，CAT/N では新しい方式を実現した．これについては第6章で述べる．

これら4種類（関数コール型の場合は URR と IEEE 規格の方式の2種類があるので，実質的には5種類）のコードは，プログラマがコンパイル時にコンパイルオプションによって指定する．さらに，実行時選択型では実行時に URR と IEEE 規格の方式を選ぶことができる．したがって，プログラマは自分の目的に合わせて必要な形式を選んでコンパイルを行えばよい．

5. OS/omicron と CAT/N の URR プロセッサへの対応

URR プロセッサを OS/omicron の実行環境で使用する場合，言語処理系の対応は CAT/N で実現した．本章では，OS/omicron と CAT/N の URR プロセッサへの対応について述べる．

5.1 URR プロセッサの提供形態とその問題点

OS/omicron はマルチタスク OS である．このため，URR プロセッサについてもマルチタスク環境に関する対策を考える必要がある．一つの方法として，特定のタスクしか使用できないようにしてしまう方法がある．しかしそれでは，OS/omicron がマルチタスクをサポートしていながら，タスクが並行に走らなくなってしまう．やはり，特定のタスクだけでなく，すべてのタスクから使用できるようにしたい．そこで，URR プロセッサのようなペリフェラルプロセッサをマルチタスク環境下で使用する場合，ユーザへの提供形態として，大別して次の二つの方式が考えられる．

［方式1］ 疑似プロセッサ方式

各タスクに対して，URR プロセッサが1個ずつ割り当たっているように見せかける方法である．プロセッサの内部状態はタスクスイッチの際，CPU の内部状態と一緒にコンテキストスイッチを行う．

［方式2］ 外部デバイス方式

URR プロセッサを外部デバイスとし，各タスクの共有資源として扱う．各タスクからの使用は，URR プロセッサの排他制御によって行う．

UNIX[9),10)] の場合，ペリフェラルプロセッサは入出力装置の一つとして扱う方法，すなわち方式2の外部デバイス方式を採っている．しかし，アクセスが頻繁に発生するペリフェラルプロセッサで外部デバイス方式を用いた場合，資源獲得の手続きが頻繁に起きることになる．一般的には OS に資源の割当てを要求する (PV, lock 命令など) ことになるため，そのオーバヘ

ッドは大きく，各タスクがどの処理単位でプロセッサを占有するかが重要な問題となる．外部デバイス方式を採用するとして，われわれが検討したのは次の二つの方法である．

(a) タスク全体の実行開始から終了まで

(b) 浮動小数点演算のある演算の開始から終了まで

方式(a)では，一つのタスクが実行開始から終了までプロセッサを占有してしまうため，プロセッサを使用するタスクが並行に走らなくなってしまう．方式(b)では，式や関数等の演算単位でプロセッサを占有する方法が考えられるが，いずれにしてもOSへの資源獲得要求が頻繁に発生することになり，システム全体の性能を圧迫する．また，外部デバイス方式の利点の一つにI/O装置のDMA転送等に見られるCPUとの並行実行があるが，プロセッサの実行速度に比べてタスクスイッチのオーバヘッドが大きい場合には使えない．

これに対して，疑似プロセッサ方式は一つのタスクに一つの疑似プロセッサを割り当てるモデルである．ユニプロセッサシステムでは，running状態のタスクは常に一つであるため，プロセッサのコンテキストスイッチを行うだけで各タスクに疑似プロセッサを割り当てることができる．running状態のタスクは常にプロセッサ使用可能状態にあり，プロセッサ獲得の手続きやそれに伴うオーバヘッド，実行待ちが発生しない．

以上の理由により，われわれのシステムでは，方式1の疑似プロセッサ方式を採用した．ただし，疑似プロセッサ方式を採る場合，URRプロセッサではコンテキストの大きさが問題となる．

URRプロセッサのフルコンテキストスイッチを行う場合，64ビットデータレジスタ16本と環境レジスタのうち5本の退避，および16本のデータレジスタと3本の環境レジスタの復帰を行う必要がある[7]．これをすべてCPUのデータ転送命令で行うことになる．現在URRプロセッサのホストマシンを日立製作所製の2050/32 (CPUはMC68020)としているため，レジスタの退避に37個，復帰に35個，計72個のデータ転送命令が必要となる．これは，明らかに大きなオーバヘッドになると予想できる．実際，ホストマシン上で実測した結果，データ転送だけでコンテキストセーブに99.7 μs，コンテキストリストアに75.3 μsを要した．これは，コンテキストスイッチのオーバヘッドとしてはかなり大きいものと考えなければならない．さらに問題なのは，URRプロセッサを利用しないタスクでもタスクスイッチのたびに，このコンテキストスイッチが伴うことである．これはタスクスイッチの実行時間の低下だけでなくTCB (Task Control Block)の増大を招き，OSのメモリ消費量を増大させる．このため，URRプロセッサのコンテキストセーブ／リストアをCPU同様に行うと，システム全体の効率を大きく損なうことになる．

URRプロセッサとOS/omicronに限れば，OS/omicronのタスク管理のTCBにURRプロセッサ用のコンテキストを追加し，URRプロセッサ使用中のフラグを設けてコンテキストのセーブ／リストアを行うことで上記問題は解決される．しかし，われわれの目標は複数の浮動小数点方式を提供することにある．この場合，浮動小数点ごとにコンテキストを割り当ててしまうと，TCBの大きさは際限なく増加してしまう．

また，一般的なペリフェラルプロセッサをシステムに追加した場合，同様の問題が起こることが考えられる．専用プロセッサであっても，その内部コンテキストは増加する傾向にあり，コンテキストのオーバヘッドの拡大の問題について，一般性を持った対策が必要であると考える．

そこでわれわれは，拡張コンテキストという新しい機構を考案し，OS/omicronに実現した[8]．

5.2 拡張コンテキスト方式の提案

URRプロセッサで問題なのは，コンテキストが大きいことである．そこで，URRプロセッサのようなデバイスを各タスクに割り当てるときに考えなければならないことは

(A) デバイスの無駄なコンテキストスイッチ

(B) TCBサイズの増大

を避けることである．拡張コンテキストはこの2点を踏まえて次の方式を採った．

(1) デバイスのコンテキストをTCBから分離

デバイスのコンテキスト退避領域をTCB内には置かず，全タスク共有の拡張コンテキスト用メモリ領域に置く．デバイスのコンテキストセーブが必要になった場合だけ，この共有領域から必要な量の退避領域を取り，TCBにリンクしてデバイスのコンテキストを追加する．そして，この領域にデバイスのコンテキストを退避する．

(2) コンテキストスイッチが必要な区間を指定

ユーザプログラムは，デバイスの使用にあたってデ

バイスに対応する使用フラグを ON にする．そして，使用終了後このフラグを OFF にする．このプロトコルによってユーザが OS に対してデバイスの使用区間を指定する．OS はタスクのコンテキストスイッチの際，デバイスの使用フラグが ON になっているタスクのみ(1)の機構を用いてデバイスのコンテキストを退避する．フラグをどこに置くかは，実現に依存する．

(1)(2)の機能により，そのデバイスを使用しないタスクでは，デバイス依存のコンテキストセーブ／リストアは発生しない．また，デバイスを使用しているタスクでもデバイスを使用していない区間ではコンテキストセーブ／リストアが起こらない．これにより，次の利点が生じる．

(1) コンテキストスイッチに時間のかかるデバイスでもコンテキストスイッチのオーバヘッドを最小限に減らすことができる．

(2) TCB の仕様を変更せずにコンテキストスイッチが必要なデバイスの追加が可能である．

(3) 任意のデバイスに対して拡張コンテキストが適用可能となり，OS の拡張性も向上する．

上記の方式がプロセッサ，デバイス，OS に依存しない一般的な方式であることに注意されたい．

われわれは，URR プロセッサに対して，OS/omicron 上でこの拡張コンテキストを実現した．

5.3 OS/omicron における拡張コンテキストの実現

上記の方式を具体的に OS/omicron 上で実現した．OS/omicron における拡張コンテキストは**図 1** に示す構成を採っている．TCB には，使用フラグの退避領域と最初の拡張コンテキストへのポインタ領域のみがあり，URR プロセッサのコンテキストセーブ／リストアが必要になったときにそのタスクに割り当てられる．拡張コンテキストは 1 個のデバイスに 1 個ずつ割り当たり，その内容はコンテキスト ID，チェーンポインタ領域，コンテキスト退避領域からなる．コンテキスト ID はその拡張コンテキストに退避するデバイスの種類を示す．チェーンポインタ領域は一つのタスクに対して複数のデバイスのコンテキストをリンクするためのものである．ただし，現在は拡張コンテキストの対象が URR プロセッサのみなので未使用である．コンテキスト退避領域が実際にデバイスのコンテキストを退避する空間である．対象デバイスによってその大きさは異なる．

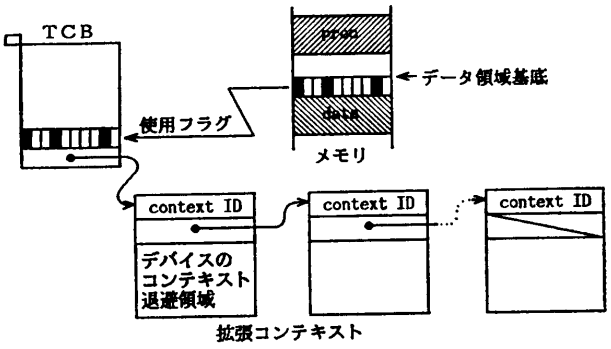

図 1 OS/omicron の拡張コンテキストの構成
Fig. 1 A structure of TCB and extended context.

拡張コンテキストを必要とするデバイス（ここでは URR プロセッサ）を使用する場合，使用に先立って，各デバイスに対応した使用フラグを ON にする．OS/omicron では，このフラグ領域をユーザタスクの静的データ領域の先頭 32 ビットに割り当てている．したがって，現在 32 個までのデバイスに対する拡張コンテキストが定義可能である．URR プロセッサにおける拡張コンテキストの動作を**図 2** に示す．図 2 において，フラグが OFF の時はコンテキストスイッチが起こっても URR プロセッサのコンテキストセーブ／リストアを行わない．これにより，コンテキストスイッチ時のオーバヘッドが減少する．

5.4 言語処理系の対応

拡張コンテキストにおいて，ユーザプログラム側で対応しなければならないことに，使用フラグ ON・OFF の操作の記述がある．基本的には，システム全体のオーバヘッドが最小限になる方法がよいが，言語 C のように分割コンパイル可能な言語では，コンパイ

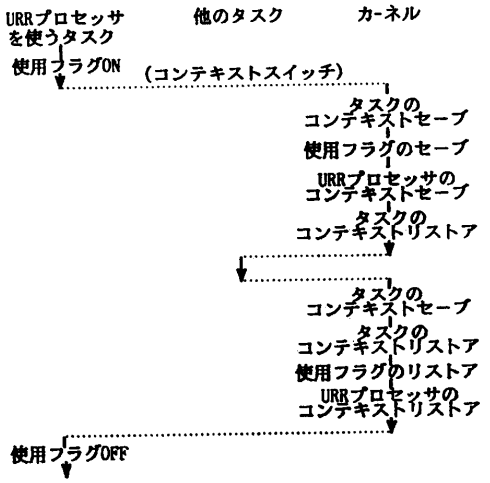

図 2 URR プロセッサ使用時のコンテキストスイッチ
Fig. 2 A process of context switching under running URR-processor.

ラからはプログラム全体の見通しはきかず，あらかじめ資源割当てをコンパイラで自動的に行うのは困難である．

そこで，一つの方法として，ユーザにフラグを解放して，ユーザにフラグ操作を記述させる方法がある．しかし，ユーザにとっては，常にURRプロセッサの使用フラグの管理をしなければならず，負担が大きくなる．また，ユーザが使用フラグONを忘れた場合，システムとしてプロセッサのコンテキストを保証できない．コンパイラの役目として，ユーザの手間を軽減すること，システムのプロトコルを保証してコードを生成することが必要である．

このため，CAT/NにおいてURRプロセッサ対応型を選択してコンパイルを行った場合，浮動小数点演算に対してURRプロセッサを使用するコードを生成するが，このとき，使用フラグのON･OFFの手続きをCAT/Nが自動的に生成，挿入することとした．これにより，プログラマは拡張コンテキストに対する特別な処理をプログラム中に記述する必要はない．現在CAT/Nでは使用フラグのON・OFFを，浮動小数点演算を使用している言語Cの式単位に行っている．

このように，OS，言語処理系の双方からURRプロセッサの拡張コンテキストをサポートすることにより，ユーザに対して効率のよい疑似プロセッサ方式でURRプロセッサを提供することを可能にした．

6. ハードウェア対応の実行時選択型の実現

6.1 実行時選択型の設計方針と問題点

実行時選択型は，主として浮動小数点表現法の評価目的としている．この浮動小数点表現法を評価する環境として，OS/omicornおよびCATでは，以前から次の設計方針のもとに開発を行ってきた[5]．

（1） 1回のコンパイル・リンクの作業で複数の浮動小数点表現法を適用できること．

（2） 比較対象となる浮動小数点表現法の数を限定しないこと．

上記のような設計方針を実現する場合に問題となるのは，プログラム中に現れる定数の扱いについてである．上記の設計方針を満足するためには，定数をコンパイル時に内部表現に変換することはできない．そこで，以前に開発したエミュレータ・トラップ方式では，定数をロードモジュール中に文字列のまま残し，実行時に毎回対応する表現形式に変換していた[5]．このため，ユーザプログラムの実行効率は低かった．また，OS/omicron第2版では，OS自身もタスクであるため，エミュレータ・トラップを行う場合，コンテキストスイッチが伴うことになる．したがって，浮動小数点演算実行ごとにコンテキストスイッチが発生することになる．これは，浮動小数点演算に対するオーバヘッドとなる．そこでCAT/Nでは，定数変換をプログラムロード時に行い，演算に対しては関数コールの形式で，実行直前に演算ルーチンをリンクする方式を採用した．この方式にすることで，関数コール型のコード生成方式と全く同じ形の実行形式が実現でき，実行速度も初期ロードの時間を除いて同じになる．また，演算ルーチンをハードウェア対応にすることで，実行速度の向上もはかれる．ただし，これを実行するためには次の点が問題となる．

（1） ロード時に定数変換を行うための情報をどのように持たせるか

ロード時に定数変換を行うためには，ロードモジュール中に次の情報が必要である．

（A） 変換前の定数の文字列

（B） 変換した定数を埋め込む場所の情報

これらの情報をどのような形式でロードモジュール中に格納するかが問題となる．

（2） ロード時に，選択された浮動小数点表現用のルーチンをリンクするにはどうしたらよいか

選択する浮動小数点表現によって演算方法は異なる．したがって，それぞれ選択された浮動小数点表現法に対応した演算ルーチンを用意し，リンクしなければならない．しかし，演算ルーチンを静的にリンクしてしまうと上記(2)の目的，すなわち浮動小数点表現の数を限定しないことと矛盾が生じる．このため，プログラムロード時に選択された浮動小数点表現用の演算ルーチンを決定し，ロードおよびリンクする方式にする必要がある．ただし，この操作はOS/omicronのような静的リンクの環境を持つものに必要な処理であり，動的リンクの環境では(1)のみ達成すればよい．

上記(1)(2)の問題に対して，われわれは言語処理系内に新たに型情報を導入することで解決を図った[8]．

6.2 型情報

ロード時に浮動小数点定数の変換や演算実行ルーチンとのリンクを行うためには，変換前の定数や修飾アドレスなどの情報が必要となる．ロード時に評価の対象となるものの情報を型情報と呼ぶ．型情報は次に示す三つの要素から構成されている．

(A) 型の名前

そのロードモジュール内で行われる初期化の種類(型)を定義する．いかなる初期化が必要かをローダに知らせる．

(B) 初期化データ情報

具体的な初期化データの定義を行うもので，型の名前と実体の組で表す．この型名と実体情報からロードモジュール中に実際に埋め込むデータの生成を行う．

(C) 初期化対象アドレス情報

初期化される対象のアドレスと，そこに行うべき初期化情報((B)の情報へのポインタ)の組で表す．この情報から，ロードされたモジュールの修飾を行う．

この三つの情報で，上記の目的を達成する．これらの型に関する情報をロードモジュール内に定義するために，アセンブラレベルで**表2**に示す四つのアセンブラ制御指令を定義した．そして，表2に示したアセンブラ制御指令によって，ロードモジュール中に型情報のレコードを生成する．

型情報を用いたCAT/Nの実行時選択型のコード生成例を**図3**に示す．

また，型情報を扱う環境ではローダが非常に重要となる．これは，ローダが型情報に対する実質的な処理を行うからである．型情報に対する処理は次のとおりである(**図4**)．

(1) 型名から，追加するモジュール(これを以下追加モジュールと呼ぶ)が必要な場合はこれをロードする．追加モジュールは，CATで言えばローケタブル・オブジェクトモジュールである．

(2) 初期化情報，初期化対象アドレス情報から，初期化データを生成し，ロードモジュールにデータの埋め込み，およびアドレス解決を行う．

型情報で使われる型名およびそれに対する処理はすべてローダ内に定義されていなければならない．逆にローダにそれらを定義しさえすれば言語処理系には何ら変更を加えずに新しい型情報を使用することができる．つまり，この機能はさまざまの他の用途への使用が可能である．例えば，現在のURRプロセッサが専用コプロセッサ化され，命令体系をMC 68881と同じにできれば，型情報を利用することで，URRとIEEE規格の方式に限られるが，現在のハードウェア対応型のコード出力形式で実行時選択型を実現することも可能となる．

表2 型情報定義用アセンブラ制御指令
Table 2 Pseudo instructions of assembler for generating type information.

制御指令名	機　　能
def_type	型名を定義する
type	実体を定義する
def_op_type	type で定義された型に対して行う演算名を定義する
op_type	type で定義された実体に演算を加えて新しい実体を定義する

7. 結果の検討

7.1 CAT/N の浮動小数点演算実行時間

CAT/Nでは，浮動小数点演算に対して第4章で示した4種類のコード生成機能を実現した．そこで，OS/omicronにおける浮動小数点の基本演算の平均演算時間を測定した[8]．測定は現在OS/omicron第2版

```
言語Cソース
    float  a = 2.0 + 3.0 ;

    f( ) {
        float  a , b ;
        a = b * 3.0 ;
    }

アセンブラソース
        def_type     単精度定数
        def_type     単精度演算
        def_op_type  単精度加算
単精度乗算  type     単精度演算
        data
定数1   type     単精度定数,"2.0"
定数2   type     単精度定数,"3.0"
定数3   op_type  単精度定数,単精度加算,定数1,定数2
a       dc.l     定数3
        proc
f
        move.l   b,-(SP)
定数4   type     単精度定数,"3.0"
        move.l   #定数4,-(SP)
        move.l   リターンアドレス,(An)
        jmp      単精度乗算
リターンアドレス
        move.l   (SP)+,a
```

図3 言語Cとアセンブラの対応例
Fig. 3 An example of code-generation for run-time selection method.

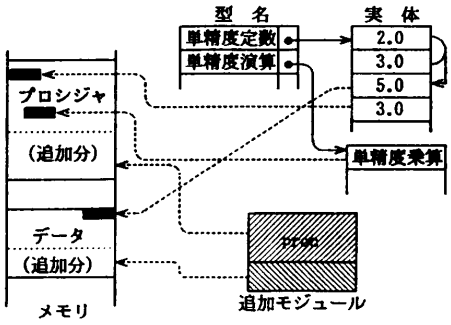

図4 ローダの処理
Fig. 4 A process of loader.

が稼働している 2050/32 上（表3）で行った．

測定結果を表4に示す．実行時選択型は浮動小数点演算そのものは関数コール方式と同じであるため，表中の関数コールの中に含めている．なお，関数コール方式におけるソフトウェアルーチンは，以前にわれわれの研究室で作成したものを OS/omicron 上に移植して使用した．表4内の式に使用されている変数は，次の宣言がされているものとする．

float a, b, c;

double d, e, f;

表4から，URR プロセッサを導入したことによる URR の演算速度は平均 5.5 倍の向上であった．また，MC 68881 を導入したことによる IEEE 規格の方式の浮動小数点演算速度は平均 8.0 倍の向上であった．

関数コール方式において，演算にハードウェアを利用する場合の演算速度は，ハードウェア対応型に比べて単精度で平均 13 μs，倍精度で 21 μs の差がある．URR プロセッサ，MC 68881 専用のコード生成に対して5割から 10 割程度の速度低下となる．しかし，それでも実行時選択型の場合，ソフトウェアルーチンを使用するのに比べて平均4倍程度の速度向上が期待できる．実行時選択型としては，かなり有効な数字であると思われる．

表3 測定システム (2050/32) の仕様
Table 3 A specification of test system (2050/32).

プロセッサ	MC 68020 (20 MHz)
コプロセッサ	MC 68881 (20 MHz) URR プロセッサ
主記憶装置	8 Mbyte
補助記憶装置	88 Mbyte ハードディスクドライブ 1 Mbyte フロッピーディスクドライブ 300 Mbyte 追記型光ディスクドライブ

表4 言語C基本演算の CAT/N のコード生成に対する平均演算時間
Table 4 An execution time of codes compiled by CAT/N.

演算	URR プロセッサ対応	MC 68881 対応	URR 関数コール		IEEE 関数コール	
			ハード	ソフト	ハード	ソフト
$a=b+c$	18.8	13.1	30.9	120.5	26.9	93.1
$a=b-c$	18.8	13.1	30.8	120.8	26.9	93.5
$a=b*c$	32.7	14.2	46.9	122.9	27.1	96.8
$a=b/c$	32.7	16.3	47.0	143.5	28.2	111.8
$d=e+f$	23.2	17.4	44.2	145.4	38.8	116.2
$d=e-f$	23.2	17.4	44.1	145.9	38.8	116.5
$d=e*f$	37.0	18.0	59.1	190.3	39.7	162.0
$d=e/f$	37.0	20.2	59.1	255.7	41.5	234.5

(注) 単位はすべて μs

7.2 拡張コンテキストのオーバヘッド

OS/omicron 第2版における拡張コンテキストは URR プロセッサに対して使用されている．この URR プロセッサのコンテキストスイッチに対するオーバヘッドを 2050/32 上で測定した結果，コンテキストセーブに 99.7 μs，コンテキストリストアに 75.3 μs を要した．この URR プロセッサ自体のコンテキストスイッチ時間はタスクスイッチのオーバヘッドとしてはかなり大きいものである．しかし，このオーバヘッドがかかるのはユーザプログラムが URR プロセッサ使用中にタスクスイッチが発生した場合のみであり，しかも使用区間を言語Cの式単位にしているため，実際に起こる確率はそれほど多いものとはならない．したがってシステム全体から見た場合，オーバヘッドはかなり小さいと思われる．

次に，拡張コンテキストに対する使用フラグの ON・OFF についてのオーバヘッドであるが，同様に 2050/32 上で測定した結果 3.3 μs であった．したがって，単純計算における使用フラグの ON・OFF の実行に対するオーバヘッドは平均 12% 3.3 μs/28 μs である．しかし，このオーバヘッドは一つの式に対しては同じであるため，式の中に含まれる項の数が増えると，その占める割合は急速に小さくなる．また，3.3 μs を要する原因にホスト計算機 (2050/32) のメモリアクセスに大きな時間を取られていることがある．最近の CPU では CPU 内部にキャッシュを持つものも登場しており，このキャッシュ内にフラグ領域を置くことなどでフラグ設定時間はかなり小さくなるはずである．したがって，演算実行時間に対するオーバヘッドとして，大きな問題にならないと言える．

8. おわりに

OS/omicron 上のアプリケーションプログラムのために，また，新しい浮動小数点表現法の比較・評価を行う環境を与えるために，複数の浮動小数点表現法，複数のコード生成機能を持った言語 C コンパイラ CAT/N を実現し，それに伴う OS/omicron の機能拡張を行った．この結果，

(1) 拡張コンテキスト方式によるマルチタスクのオーバヘッドの軽減

（2） 型情報の利用による，URR と IEEE 規格の方式，2種類の浮動小数点表現法のハードウェアを用いた演算処理環境の実現

を達成した．これらの機能は URR プロセッサを OS/omicron 環境に組み込むのに有効な方法であった．今後の課題として，URR プロセッサの専用 LSI 化，OS/omicron の仮想記憶化による動的リンクの環境の実現などがある．

最近“ソフトウェア IC”と称して，従来ソフトウェアで行っていた処理をペリフェラルプロセッサ化することが盛んになりつつある．これに対し，拡張コンテキストと型情報は，ソフトウェア IC を効率よくシステムに接続するための一手法になる．また，このほかにも，同一 CPU 異種 FPU マシン間のオブジェクトモジュールの共有，さらには，浮動小数点表現，文字コードの異なるシステム間接続にも有効に使えるのではないかと考えている．

謝辞 本研究の一部は，文部省科学研究費試験研究Ⅰ・計算機内部数値表現の研究（62880009）の助成を受けた．われわれに共同研究の機会を与えてくださった，早稲田大学・中島勝也教授，筧捷彦教授，立教大学・故島内剛一教授，電気通信大学・浜田穂積教授，日立製作所中央研究所・武市宣之主任研究員（現日立情報システムズ開発研究所主管研究員），和田健一主任研究員（現日立製作所マイクロエレクトロニクス機器開発研究所第1部長），大山光男主任技師に深謝する．

参 考 文 献

1) 浜田穂積：二重指数分割に基づくデータ長独立実数値表現法Ⅱ，情報処理学会論文誌，Vol. 24, No. 2, pp. 149-156 (1983).
2) Stevenson, D., Chairman, Floating-Point Working Group Microprocessor Standards Committee : A Proposed Standard for Binary Floating-Point Arithmetic, Draft 8.0 of IEEE, *Computer*, Vol. 14, No. 3, pp. 51-62 (1981).
3) 高橋延匡：研究プロジェクト総説：OS/omicron の開発，情報処理学会オペレーティングシステム研究会資料，39-5 (1988).
4) 並木美太郎，屋代 寛，田中泰夫，篠田佳博，藤森英明，中川正樹，高橋延匡：OS/omicron 用システム記述言語C処理系Catのソフトウェア工学的見地からの方式設計，電子情報通信学会論文誌 D, Vol. J71-D, No. 4, pp. 652-660 (1988).
5) 森 岳志，中川正樹，高橋延匡，中森眞理雄：各種浮動小数点表現法の評価方式の実現，情報処理学会論文誌，Vol. 29, No. 8, pp. 807-814 (1988).
6) 並木美太郎，鈴木茂夫，岡野裕之，堀 素史，横関 隆，中川正樹，高橋延匡：マルチプロセッサシステム向けの OS/omicron タスク管理の設計と実現，情報処理学会論文誌，Vol. 31, No. 6, pp. 894-905 (1990).
7) 中原雅彦，中川正樹，高橋延匡：URR 浮動小数点演算コプロセッサのアーキテクチャの検討と言語C処理系 cat による利用方式，第36回情報処理学会全国大会論文集，pp. 219-220 (1988).
8) 中原雅彦，岡野裕之，横関 隆，早川栄一，並木美太郎，高橋延匡：複数の浮動小数点方式を処理する OS/omicron と言語C処理系 CAT/N の評価，情報処理学会オペレーティングシステム研究会資料，46-1 (1990).
9) Bach, M. J. : *The Design of the UNIX Operating System*, Prentice-Hall (1986).
10) Leffer, S. J., Mckusick, M. K., Karels, M. J. and Quarterman, J. S. : *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley (1989).

（平成3年6月12日受付）
（平成4年2月14日採録）

中原 雅彦（正会員）

昭和40年生．昭和63年東京農工大学工学部数理情報工学科卒業．平成2年同大学院工学研究科修士課程修了．同年(株)日立製作所システム開発研究所入社．在学中，コンパイラなどシステムソフトウェアの研究に従事．

早川 栄一（正会員）

平成元年東京農工大学工学部数理情報工学科卒業．平成3年同大学大学院博士課程前期修了．同年4月同大学大学院博士後期課程入学，現在に至る．オペレーティングシステムなどのシステムソフトウェアの研究に従事．

岡野 裕之（正会員）

昭和63年東京農工大学工学部数理情報卒業．平成2年同大学院修士課程修了．同4月(株)日本 IBM に入社．在学中，仮想マシン，マルチプロセッサ用オペレーティングシステムの研究に従事．

並木美太郎（正会員）

昭和 59 年東京農工大学工学部数理情報卒業．昭和 61 年同大学院修士課程修了．同 4 月(株)日立製作所基礎研究所入社．昭和 63 年より東京農工大学工学部数理情報助手．平成元年 4 月より電子情報助手．並列処理，日本語情報処理のソフトウェア／ハードウェアアーキテクチャに興味を持ち，コンパイラ，オペレーティングシステムなどシステムプログラムの研究・開発に従事する．

高橋　延匡（正会員）

昭和 8 年生．昭和 32 年早稲田大学第一理工学部数学卒業．同年(株)日立製作所中央研究所入社，HITAC 5020 モニタ，TSS の開発に従事．昭和 52 年より東京農工大学工学部数理情報教授．平成元年電子情報教授．理学博士．オペレーティングシステム，日本語情報処理，パターン認識の研究に従事．電子情報通信学会，ソフトウェア科学会，計量国語学会，ACM 各会員．