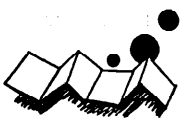
解　説



オブジェクト指向プログラミングのハードウェア・ファームウェアサポート†

小　方　一　郎††

1. はじめに

メッセージ・パッシングは通常の手続き型言語にはない，オブジェクト指向言語に特徴的な機能である．メッセージ・パッシングという用語はさまざまな意味に使われるが，Smalltalk をはじめとするオブジェクト指向言語においてのメッセージ・パッシングとは一言でいえば手続き（メソッド）を実行時に動的に決定して呼び出す機能だということができる．一般にオブジェクト指向プログラミングではメソッドを小さく作り，モジュール性を高くする傾向があるので，メッセージ・パッシング性を高くする傾向があるので，メッセージ・パッシングは非常に頻度の高い操作となる．そこで，これを高速に実行することが性能を向上させる鍵となる．

また，複雑なデータを大量に扱うオブジェクト指向プログラミングでは自動記憶管理は必須の機能である．しかし，通常のアーキテクチャでは自動記憶管理はややオーバヘッドが大きい．そこで，高速な記憶管理の実行をサポートできるアーキテクチャが望ましい．つまり，オブジェクト指向言語に特徴的で，しかも性能上のボトルネックになるような部分として，メッセージ・パッシングのための負荷，動的記憶管理のための負荷があげられる．

オブジェクト指向プログラミングはまだ歴史が浅く，また実行メカニズムも複雑なためハードウェア，ファームウェアでの実現例はそれほどはない．しかし，Smalltalk-80[5),6)] はこの分野での草分けであり，広く知られているため数々のアプローチを受けている．Smalltalk-80 のオリジナルのインプリメンテーションはゼロックス社のマイクロ・プログラマブルな個人用ワークステーション Dorado[2)] の上のものである．Dorado は非常に高性能であったが，一方非常に高価でもあった．またインプリメンテーションは開発実験の意味もあったため非常にシンプルで新しいインプリメンテーションの技法が取り入れられていない．

そこで登場したのが Soar[18),14)], AI 32[12),8)], COM[1)], Swamp[9)] などの専用ハードウェアである．

2. オブジェクト指向プログラミングの特徴

この章では，まずオブジェクト指向言語の実行において特徴的な機能を解説し，その後で具体例として Smalltalk-80 の VLSI インプリメンテーションである Soar と AI 32 の実現の研究をとおして，オブジェクト指向プログラミングのハードウェア，ファームウェアのサポートについて解説する．

メッセージ・パッシングの負荷，動的記憶管理の負荷は他のオブジェクト指向言語同様，Smalltalk-80 においても端的にみられる．したがって，Smalltalk-80 専用ハードウェアの戦略はオブジェクト指向言語一般について当てはまると考えられる．

2.1 メッセージ・パッシング

＊ メッセージ・パッシングとは

メッセージ・パッシングにはいろいろなバリエーションが存在するが，ここでは，Smalltalk-80 に沿って解説する．Smalltalk-80 のメッセージ・パッシングは一般のオブジェクト指向言語のなかでも代表的な形であると考えられる．

Smalltalk-80 では，どんな式も一つのメッセージ・パッシングとなる．3＋4という式を評価することを考えてみよう．3はレシーバと呼ばれ＋4がメッセージと呼ばれる．まず，レシーバである3のクラスを調べる．3は小整数 (Small Integer) のクラスに属するオブジェクトである．3の型 (type) は小整数であると考えてもよい．さてクラス小整数のメソッド辞書の中からセレクタ＋という見出しを検索する．＋という見出しに対応する内容が実行すべきメソッドである．

† Hardware and Firmware Support for Object-Oriented Programming Languages by Ichiro OGATA (Computer Science Section, Electrotechnical Laboratory).

†† 電子技術総合研究所

探索されたメソッドを実行する新しいコンテクスト（スタック・フレームに相当するもの）を作りその上の局所変数を初期化して，新しいメソッドの実行が始まる．この一連の手続きがメッセージ・パッシングである．

ここでメソッドは通常の手続き型言語の関数に相当する．メッセージ・パッシングと関数呼出の違いは特別な引数であるセレクタの型によって探索する辞書が変わり，結果として違う関数（メソッド）を実行することであるといえる．

＊ メソッド探索の高速化

通常の手続き型言語では変数の型の情報や関数の名前により，コンパイル時に一意に呼び出すべき関数が分かっている．ところがメッセージ・パッシングでは実行時になってレシーバの値が定まらないことには実行すべきメソッドが分からない．そこでメッセージ・パッシングはオブジェクト指向言語の実現上のオーバヘッドとなる．

また，通常オブジェクト指向言語にはメソッド継承機能がある．メソッドの継承とは自分のクラスに求めるメソッドが見つからない場合，探索「親」のクラス（スーパクラス）に広げていく機能である．この機能により，オブジェクト指向言語はメソッドを共有したいわゆる差分プログラミングを可能としているわけである．この機能が生かされているほど，複数のクラスのメソッド辞書を探索しなければならない．そこでメッセージ・パッシングのメソッド辞書探索（メソッド探索）はますます重くなる傾向があり，この部分の高速化が重要となる．

大規模プログラミングを目指すオブジェクト指向プログラミングでは，一つ一つのメソッドを小さく作る傾向がある．これはメソッドのモジュール性を高め，再利用しやすいコードとするためである．また，データ隠ぺい機能のために，オブジェクト内部のデータにアクセスするためにもメッセージ・パッシングが必要となる．そこで，メッセージ・パッシングは非常に頻度の高い操作となる．

したがってオブジェクト指向言語の効率的な実現では，メッセージ・パッシングの高速な実行のサポートが重要である．単純なインプリメンテーションではメソッド探索が重く，実行上のボトルネックとなることは明白である．そこで，メソッド探索をハードウェア，ファームウェアでサポートすることが研究の焦点となる．

＊ メソッド・キャッシュ

メソッド探索の高速化においてソフトウェア，ハードウェア両面でよく使われる技法にメソッド・キャッシュがある．メソッドはレシーバのクラスとセレクタで一意に決定する．そこで，クラスとセレクタをある種の演算，たとえば加算などで鍵を作り内容を対応するメソッドとするような表（キャッシュ）を作れば二度目以降の探索の手間を大きく削減できる．2種類のキャッシュ・メカニズムが知られている．

1) グローバル・キャッシュ

全世界で1種類のキャッシュ・テーブルをもち，すべてのメソッド探索をこの表の探索を介して行う．

2) インライン・キャッシュ

一つのメッセージ・パッシングのコードごとに一つのエントリの表をもち，これをキャッシュとする．これは多義性があるとはいえ，メッセージ・パッシングごとにその起動するメソッドは局所性があるという事実に基づいた工夫である．グローバル・キャッシュとも両立し，併用することが多い．

2.2 自動記憶管理

オブジェクト指向プログラミングでは自動記憶管理が行われるのが普通である．

＊ 自動記憶管理とは

自動記憶管理を行うプログラミング・システムとしては，Lisp が有名であり，ハードウェア・ファームウェアの研究は Lisp の実現を目指して行われてきた．オブジェクト指向プログラミングのサポートの研究はこの延長線上にある．

オブジェクト指向プログラミングでは，特に Smalltalk-80 を代表として，Lisp と同様に変数に型をつけない場合が多い．これは，多義性（adhocpolymorphism）を生かしたプログラミングのためである．型がないため変数にあらかじめ記憶領域を割り当てることはできない．そこで，Lisp のようにポインタを用いた実現となる．Pascal の言葉でいえば，すべての変数はすべての型を指す可能性のあるポインタ型というわけである．（もちろん Pascal では特定の型へのポインタしか定義できないから，これは Pascal の文法には納まらない．）

この場合，データ・インスタンスは，システムに対する要求により動的に作り出され，そのインスタンスを参照するポインタが変数に代入される．ちょうど Pascal の標準関数 new に対応する操作である．このようにしてインスタンスは動的にどんどん作り出さ

れ，プログラムの実行とともに記憶領域は消費されていく．

システム内のどの変数からもあるインスタンスへのポインタが消えたとき，そのインスタンスはごみとなる．ごみは以後参照されることは決してないので，ごみの占めている記憶領域を解放して，再利用に提供することができる．これがごみ集め（garbage collection）と呼ばれる操作である．

通常の Pascal の処理系ではごみ集めの機能はなくプログラマはデータが使われていないことを確認した上で free などの標準関数を明示的に呼ばなければならない．

このようにごみ集めはこのような煩雑で，また誤りの起こりやすい記憶管理の操作からプログラマを解放する機能である．

＊ オブジェクト指向プログラミングと自動記憶管理

より複雑で大規模なプログラミングを可能とするために，プログラミング言語はより複雑なデータ構造を扱うように発展してきた．たとえば Fortran では，配列が唯一の構造であったが，Pascal では，レコード型，ポインタ型をもっている．オブジェクト指向計算においてプログラマがクラスを作り，インスタンス変数を定義することは，インスタンスの構造を定義する（型定義する）ことにほかならない．

つまり，オブジェクト指向言語は型を豊富にする方向に進んだ一つの姿である．そこで，ごみ集めをサポートせず記憶管理をユーザに任せるのは，動的にさまざまなデータ＝オブジェクトを数多く扱うオブジェクト指向プログラミングをひどく煩雑にする．またユーザが記憶領域を正しく管理することは（特にオブジェクト指向プログラミングの得意とする大規模ソフトウェアでは）事実上不可能でもある．そこで，自動記憶管理はオブジェクト指向言語に必須の機能となる．

＊ Smalltalk-80 に特有な記憶管理のオーバヘッド

Smalltalk-80 はコンテクスト（スタック・フレーム）をオブジェクトとして扱い通常のヒープに割り当てる．これはたとえば高機能なデバッガの作成などに大きな力を発揮する．しかし，これの素直なインプリメンテーションは1回のメッセージ・パッシングごとに一つのコンテクストのメモリ割当をともなうため，メモリ管理に大きな負担を掛け効率が悪い．また，メモリの割り当て，初期化にも時間が掛かるからメッセージ・パッシングの速度も落とす．文献[6]によれば，マイクロ・プロセッサ上での素直なインプリメンテーションではメッセージ・パッシングとメモリ管理を合わせるとシステムの実に 60% の時間を消費する．

したがって，通常の FILO のスタックでコンテクストを構成すれば，このオーバヘッドを大きく削減しシステムの速度の大きな向上が得られるだろう．

そこで，新しい Smalltalk-80 のインプリメンテーションでは，この問題を解決するリニア・スタック・アルゴリズム[3),16)]を採用している．このアルゴリズムはコンテクストを，

1) 通常の言語のスタックのように FILO の割付けを行う実行に適した形

2) データ参照に適した普通のオブジェクトの形

の二つの形態をもたせ，要求に応じてコンテクストの形態を変換するというものである．このアルゴリズムではデータとして参照されることのないコンテクストはヒープに割り当てられることなく消滅する．そこでメモリ管理部の負担が劇的に減少する[21)]．

もちろん，コンテクストをヒープに割り当てるという Smalltalk-80 の設計自体にもやや問題がある．オブジェクト指向も含めて，通常の言語ではこのような設計とはなっていないだろう．Smalltalk-80 に特有なオーバヘッドというのはこのようなわけである．しかしこの問題は深刻なものなので，すべてのインプリメンテーションではこの問題に取り組んでいる．

3. Smalltalk-80 実行専用ハードウェア

3.1 SOAR によるアプローチ

SOAR (Smalltalk On A Risc) は，その名のとおり Smalltalk-80 を RISC[13)] のうえで実現しようという試みである．これは RISC (Reduced Instruction Set Computer) 論争に火をつけたパターソン教授（カリフォルニア大バークリー校）のプロジェクトである．これは RISC でも，Smalltalk-80 のような非常に高機能な言語の高速な実行が可能であるということの実証を目指している．また逆に RISC だからこそ複雑な命令の構成にチップ面積を犠牲にせずに必要なハードウェア・サポートを盛り込めるという考え方である．

SOARチップは約3万5千個のトランジスタを集積した，nmos テクノロジによる VLSI である．RISC の思想に基づき次のような特徴をもつ．

1) すべての命令の命令長は同じ．また1サイクル

ですべての命令の実行を終了.

2) すべて布線ロジック. マイクロコードなし.

3) 豊富なレジスタウィンドウ.

4) ハードウェアによるタグのサポート.

5) ハードウェアによるインライン・キャッシュのサポート.

6) インクリメンタル・コンパイルによる Risc 命令の直接実行.

7) レジスタ・ウィンドウによるスタックの実現.

＊ 型の判定

SOAR は1ビットのタグがデータに付けられている. 整数は MSB が0であり, それ以外のオブジェクトすなわちポインタには1が立っている (図-1). またポインタには自動記憶領域管理のアルゴリズムに用いる3ビットの世代タグが付加されている. これについては後述する.

前述のように, メッセージ・パッシングでは演算子が決定していても, 演算は決定できない. レシーバのクラスを動的に調べなければならない. しかし

$a<-b+c$

という式で+が整数の加算である頻度は非常に高い. 一般にどんなシステムでも整数の演算は多用されているため, この部分の最適化は非常に重要である.

普通のインプリメンテーションではソフトウェアもしくはファームウェアで加算を実行する前に型のチェックを行っていた. もちろん型チェックはいくつかのサイクルを消費するのでこの方法は遅いわけである. SOAR はこれを1サイクルで実行する. つまり SOAR の加算命令ではデータの加算が実行されるのと並行してタグがチェックされる. もし二つの引数の型がともに整数ならばなにも起こらず, 結果が1サイクルで得られる. それ以外の場合には, これは稀なケースなのだが, タグ割り込みが発生し正しい処理が行われた後, 復帰する.

これと同様なことが条件分岐についてもいえる. 整数演算の条件判定は1サイクルで行われるが, そうでない場合はタグ割り込みが発生し適切な処理が行われるようにする.

＊ インタプリテーション

Smalltalk-80 システムは Dorado 上でインプリメンテーションをベースとしたスタックマシンモデル

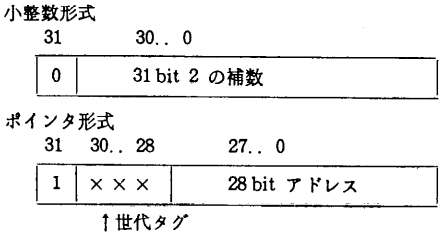


図-1 SOAR のポインタ形式

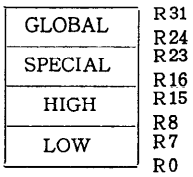


図-2 SOAR のコンテクスト・レジスタ

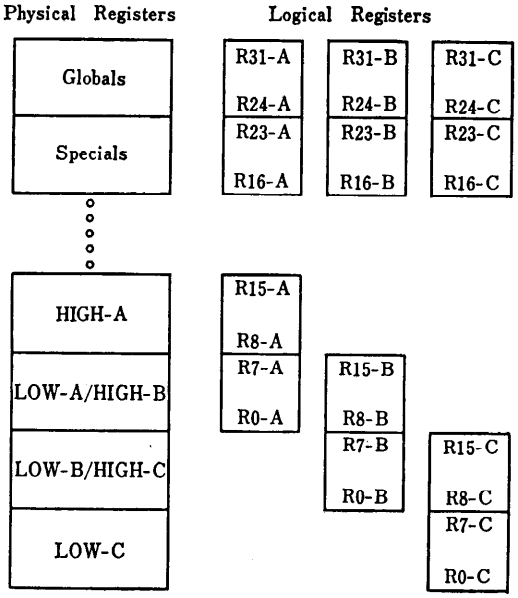


図-3 SOAR のウィンドウ・レジスタ

図-2 のコンテクスト・レジスタの半分 (High, Low) はその上下のコンテクストと共有され, この部分を通じて, 引数及び結果の受け渡しを行う.

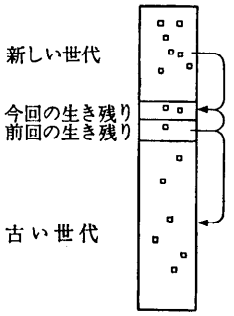


図-4 SOAR の世代管理

「新しい世代」のみがごみ集めの対象となる

を用いた仮想機械ベースで記述されている．この記述はSmalltalkの本[5]の4.にある．さてここで使われるのがbyte codeと呼ばれる仮想マシンコードである．ゼロックスから配布されるシステムのイメージにはこのbyte codeが含まれている．したがってbyte codeに基づくインプリメンテーションはもっとも普通の考えであり，事実この後のAI 32やいくつかのソフトウェアによるインプリメンテーションではbyte codeが命令セットとしてそのまま用いられている．

SOARが特徴的なのはまさにこの点でRISC派のかねてからの主張どおり，Smalltalk-80をオリジナルなRISC命令セットにコンパイルして実行しようというものである．byte codeは密度が高くメモリを節約はするが，命令セットの解釈には時間がかかる．今日のように高速なメモリが安価に大量に手にはいる状況ではメモリと性能を取り引きできるというものである．

＊ メッセージ・パッシングの高速化

メッセージ・パッシングの高速化のためにDeutch-Schiffman[3]の方法と同じインライン・キャッシュが用いられている．

SOARはこれをハードウェアでサポートする．SOARはメッセージ・パッシングと同時にキャッシュの内容で指定されたメソッドの起動を始める．それと同時にレシーバのクラスとセレクタをキャッシュの鍵との比較を行う．鍵と一致していれば，メソッドの実行が開始されるが，そうでない場合には割り込みが起動され正しいメソッドの探索が始まる．

またRisc II[7]チップから引き継いだレジスタ・ウィンドウを用いて，Deutch-Schiffman[3]やSuzuki-Terada[16]と同じ手法であるリニア・スタックを実現している．これは実行中のメソッドのコンテクスト(スタック・フレーム)をCPUチップの内部にキャッシュする．またコンテクストがレジスタであるため局所変数の初期化(Nil Filling)とパラメータのレジスタ・ウィンドウによる受渡しがハードウェアによりサポートされる．

このような工夫でメッセージ・パッシングに関してはDoradoインプリメンテーションの4.1倍の性能を発揮する．

＊ ごみ集め

SOARは世代管理法に文献[10),17)]に基づくごみ集めをサポートしている．これは1回のごみ集めで生き残る(ごみとならない)ごとにオブジェクトの世代数を増やしていき，世代数がオーバフローしたときにこの何世代も生き残ったオブジェクトをごみ集めの対象とならない領域へ移動するという方法である．ごみ集めは新しいオブジェクトの領域のみに対して行う．こうしてごみ集めの作業領域を小さく保ちオーバヘッドの増加を防いでいる．

古い世代のオブジェクトの中から指される新しいオブジェクトは特別なテーブルで管理される．このテーブルはリバース・ポインタをもち，新しいオブジェクトが古いオブジェクトになる場合の移動に備えている．また古い世代のオブジェクトに新しい世代のオブジェクトへのポインタをストアする場合にはテーブルを更新する必要があるのでこれはすべてのストアのオーバヘッドとなる．

SOARではポインタのストアをハードウェアでサポートしテーブルの更新が必要な場合にはトラップを発生する．ストアのほとんどは新しい世代のオブジェクトに対して起こるため，テーブル更新のオーバヘッドは無視できる．またポインタには指しているオブジェクトの世代数を示す3ビットのタグが付加してある．

3.2 AI 32によるアプローチ

AI 32は日立製作所で開発中のSmalltalk-80やLisp, Prologを高速に実行する専用VLSIである．ホストMPUのコ・プロセッサとして動作する．AI 32は1.3ミクロンCMOSテクノロジを用い382,000トランジスタを集積した．これはSoarを1桁上回る集積度である．

AI 32は次のような特徴をもつ．

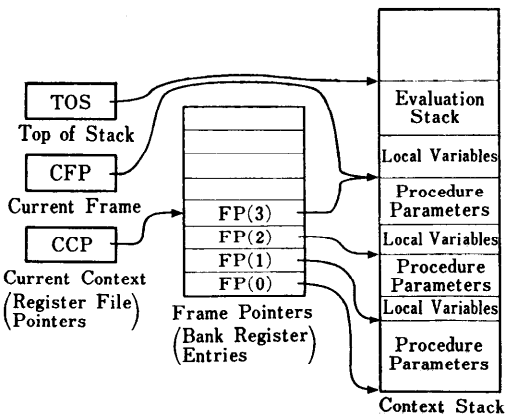


図-5 AI 32のチップ内コンテクスト
AI 32コンテクストをチップ内部のレジスタに格納し，高速化を図る．

1) ネクスト・アドレスをもつマイクロコード.(マイクロコード・シーケンサなし.)

2) 256 ワードの大容量レジスタ・ファイルとこれをポイントするレジスタ・ポインタ.

＊ 型の判定

AI 32 で特徴的なのは次に実行すべきマイクロコード・アドレスをネクスト・フィールドで指定することである.ネクスト・フィールドとタグの演算の結果を次に実行すべきマイクロコードアドレスとできるため,タグ・ディスパッチが2マシン・サイクルと高速に実行できる.この仕組みにより,タグのついたデータである,小整数の演算,比較のコードが高速に実行できる.

＊ インタプリテーション

AI 32 は Smalltalk-80 バイトコードを直接実行する.AI 32 はバイトコードのディスパッチにもネクスト・フィールドを用いる.前にも述べたようにこの操作は高速なので汎用マイクロ・プロセッサでしばしば問題となるバイトコードをインタプリットするオーバヘッドは無視できるものとなる.

＊ メッセージ・パッシングの高速化

AI 32 はリニアスタック・アルゴリズムを採用した上,その LIFO の形のコンテクストをチップ内部のレジスタに割り当て,通常のスタック操作をチップ内部で行う.こうすることでメモリとのトラフィックを減らし,高速化を計っている.またコンテクストの管理のために専用のレジスタ・ポインタを備え,コンテクスト関連のレジスタを高速にアクセスできる.またレジスタ・ポインタはオーバフローをハードウェアで検出するため,チップ内レジスタに納まっているかどうかのチェックは不要である.

＊ ごみ集め

AI 32 はごみ集めに Deutsch-Schiffman[3] と同じ遅延参照カウンタ法を用いる.チップ内にはオブジェクトの参照カウンタ部のキャッシュが含まれ,主記憶へのアクセスなく高速に参照カウンタの増減ができる.

AI 32 はその豊富なレジスタ資源を生かし,グローバルキャッシュ,スタックフレーム,機械命令などのキャッシュを積極的に行っている.このため主記憶へのアクセスは少なくなっている.将来的には,メモリの動作速度の向上はそれほど見込めないが,チップを作るテクノロジが進歩し動作クロックを上げられるようになることは考えられる.その場合メモリ・アクセスの要求が厳しくない.AI 32 ではクロックの向上が直接の性能向上に結び付く.SOAR は基本的には主記憶のクロックに CPU のクロックが同期する必要がある.そのためチップのクロックが上がった場合,キャッシュが必要となるであろう.

3.3 専用マシンの性能について

ソフトウェアによる Smalltalk のインプリメンテーションで有名なものに,PS[3] がある.これは,Smalltalk の主要な開発メンバであった Deutsch が当初,モトローラのマイクロ・プロセッサ MC 68020 をターゲットとして開発したものである.現在はアルゴリズムをより効率のよいものに置き換えた Parc Place Systems Smalltalk-80 となっている.

新しい PS[20] は次のような特徴をもつ.

1) MC 68020 や Spark などの機械命令に実行時にインクリメンタルにコンパイルし実行する,インクリメンタル・コンパイル技術の採用.

2) コンテクスト(スタック)をその CPU での自然な形のままで実行し,必要に応じてオブジェクトに変換する,リニア・スタック・アルゴリズム.

3) 世代別ごみ集め.

4) インライン・メソッド・キャッシュの採用.

25 MHz の MC 68020 をエンジンとする Sun 3/200 では Parc Place Systems Smalltalk-80 は Dorado の2倍にも達する性能を発揮する.同様に 16.6 MHz の Spark チップ (Risc) をエンジンとする Sun 4 では性能は Dorado の 3.5 倍にも達するという[20].

一方 Soar や,AI 32 の性能はまだ一般に使われてはいないので評価は難しいが,おおむね Dorado 程度である.

この現象は次のように説明される.

1) 汎用 LSI と専用 LSI が使うことのできるテクノロジとの間には大きな差がある.また,LSI の性能はたとえ同じテクノロジを用いていても,その最適化の程度により性能に大きな違いが生じる.したがって莫大な開発力を注がれている汎用 LSI と,研究レベルの専用 LSI とでは性能の差がますます大きくなる.

2) 性能評価を繰り返し,注意深く作られたソフトウェアは汎用マイクロ・プロセッサで生じるオーバヘッドの大部分を吸収する.また汎用機には性能評価ツールや非常に最適化されたコンパイラなどがすでに存在していて,生産性も高い.したがって,最適化も短時間で行える.PS は実際に高級言語である C言語で記述されている.

4. おわりに

この章では代表的オブジェクト指向プログラミング言語である Smalltalk-80 の実現手法の研究をとおして,オブジェクト指向プログラミングのハードウェア,ファームウェア・サポートについて解説した. オブジェクト指向プログラミングは従来にない仕組みをもちそれが従来のプログラミング言語にはない負荷となっている. これを回避するためソフトウェア, ハードウェアの両面からの高速化の研究が進められてきた.

オブジェクト指向プログラミングは大規模プログラミングを可能とするための一つの試みであり, まだ発展途上にある. これからも新しい言語仕様が研究されこれとともにハードウェア, ファームウェア・サポートも研究されていくことであろう. 汎用マイクロプロセッサの性能が非常に高まりまた安価にはなった. しかし, VLSI の高性能化はとどまるところを知らず, まだまだ高速化, 高集積化が期待できる.

そうなった場合, あり余る資源をどのように使うかというのは大きなテーマであり, 高機能のサポートでメモリへのアクセスを減らし, また特殊な並列チェック (タグチェック, 境界チェック) が期待できる専用 VLSI は興味深いテーマである.

参 考 文 献

1) Dally, W. J. and Kajiya, J. T.: An Object Oriented Architecture, In OOPSLA 1985 ACM (1985).
2) Deutsch, L. P.: The Dorado Smalltalk-80 Implementations: Hardware Architecture's Impact on Software Architecture, In G. Kansner, editor, Smalltalk-80: Bits of History, Words of Advice, pp. 113-126, Addison-Wesley, Reading, Massachusetts (1983).
3) Deutsch, L. P. and Schiffman A. M.: Efficient Implementation of the Smalltalk-80 System, In 11 th Principles of Programming Languages, ACM (1984).
4) Falcone, J. R.: The Analysis of the Smalltalk-80 System at Hewlett-packard, In G. Kansner, editor, Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading, Massachusetts (1983).
5) Goldberg, A. and Robson, D.: Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts (1983).
6) Kansner, G. editor: Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading, Massachusetts (1983).
7) Katevenis, M.: Reduced Instruction Set Computer Architectures for VLSI. ACM Doctoral Dissertation Award 1984, The MIT Press, Cambridge, Massachusetts (1985).
8) Kawasaki, S., Nojiri, T. and Sakoda, K.: A User-adaptable Vlsi Engine for Artifical Intelligence, In Information Processing 86, IFIP (1986).
9) Lewis, D. M., Galloway, D. R., Francis, R. J. and Thomson, B. W.: Swamp: A Fast Processor for Smalltalk-80, In OOPSLA 1986 ACM (1986).
10) Lieberman, H. and Hewitt, C.: A Real-time Garbage Collector Based on the Lifetimes of Objects, CACM, 26 (6) (June 1983).
11) Moon, D.: Architecture of the Symbolics 3600 In 12th Annual Symposium on Computer architecture, IEEE, pp. 76-83 (1985).
12) Nojiri, T., Kawasaki, S. and Sakoda, K.: Microprogrammable Processor for Object-Oriented Architecture, In 13th Annual Symposium on Computer architecture, IEEE (1986).
13) Patterson, D.: Reduced Instruction Set Computer, CACM, 28 (1) (Jan. 1985).
14) Samples, A. D., Unger, D. and Hifginger, P.: Soar: Smalltalk without Bytecodes, In OOPSLA 1986 ACM (1986).
15) Suzuki, N., Kubota, K. and Aoki, T.: Sword 32: A Bytecode Emulating Microprocessor for Object-Oriented Languages, In International Conference on Fifth Generation Computer Systems 1984, ICOT (1984).
16) Suzuki, N. and Terada, M.: Creating Efficient Systems for Object-Oriented Languages, In 11 th Principles of Programming Languages, ACM (1984).
17) Unger, D.: Generation Scavenging: A Non-disruptible High Performance Storage Reclamation Algorithm, Software Engineering Notes, ACM, 9(2) (May 1984).
18) Unger, D., Blau, R., Foley, P., Samples, D. and Patterson, D.: Architecture of SOAR: Smalltalk on a Risc, In 11 th Annual Symposium on Computer architecture, IEEE (1984).
19) Unger, D. and Patterson, D.: Berkley Smalltalk: Who Knows Where the Time Goes? In G. Kansner, editor, Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading, Massachusetts (1983).
20) Parcplace Systems' demonstration at OOPSLA 1987, ACM.
21) 鈴木則久, 小方一郎: 多態実行環境, 情報処理, Vol. 26, No. 11 (Nov. 1985).

(昭和 62 年 11 月 18 日受付)