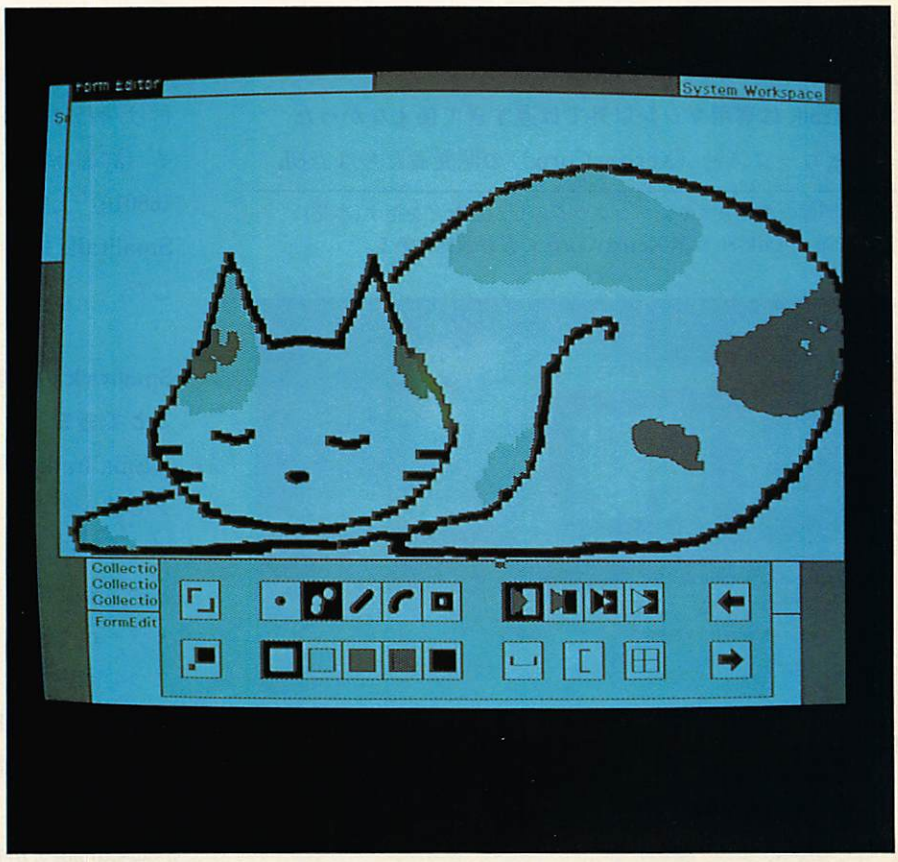
論文

Smalltalkシステムを マイクロプロセサ利用システム上で 最適化し，実行速度を向上

Allen Wirfs-Brock
米 Tektronix Inc.

米 Tektronix Inc. の人工知能開発用ワークステーション「4404」は，Smalltalk-80 の実行速度を既存の専用マシンに比べ 2 倍程度に上げた。マイクロプロセサ（68010）使用のシステムで Smalltalk の実行速度を上げるため，ソフトウェアを工夫した。実行環境（コンテクスト）をオブジェクトとせずスタックに置く，バイトコードのディスパッチを改良するなどである。Smalltalk のプログラミング環境（仮想イメージ）を実行する仮想マシンは 68010 のレジスタを効率良く使うためアセンブラで記述した。（本誌）



Smalltalkシステムをマイクロプロセサ利用システム上で最適化し，実行速度を向上●目次

1970 年代，人工知能分野で研究をしていたコンピュータ科学者，技術者は，さまざまなプログラミング手法を用いてプログラムを開発していた。関数型，論理型，オブジェクト指向などのプログラミング・パラダイム[注1]は，複雑な知的システムを開発するのに非常に役立つことがわかった。これらのプログラミング・パラダイムの一つ，あるいは複数を統合した新しいプログラミング言語が開発された。新しい言語を効率良く実行するコンピュータ・アーキテクチャの実験的開発もなされた。

1980 年代になって個人用の人工知能マシンを開発し，商品にしよう，という目標がコンピュータ業界に起こってきた。この目標には，前述したプログラミング・パラダイムに対応する Lisp, Smalltalk, Prolog などの言語を使える高性能で低価格なシステムが必要となる。われわれの開発した 4404 は，これらの言語をサポートするように設計した机上型コンピュータの一つである（図 1）。Smalltalk-80[注2]を利用できた従来の専用マシンに比べ，半分以下の価格で，2 倍以上の性能を達成した。

Smalltalk は専用マシン以外では遅すぎて使えなかった

米ゼロックス社（Xerox Corp.）の開発者たちは当初，Smalltalk 専用の命令セットを専用プロセサ上にマイクロコードで記述した。1981 年にゼロックス社は Smalltalk-80 のライセンスを他の研究機関に開放した。ライセンスを取得した研究機関は，汎用のコンピュータ上に Smalltalk を移植しようとした。移植はできたものの，ゼロックス社の最も処理速度の遅い Smalltalk システムよりも性能は悪かった。実際の応用プログラムを開発するには遅すぎて使えなかった。こうした結果[5]から，ゼロックス社以外のコンピュータで実用に耐える Smalltalk を実行するには，新しい実現技術を取り入れなければならないことが明らかになった。あるグループは，Smalltalk プログラムの実行に最適化したマイクロプロセサを開発する道を選んだ[6),7)]。

米テクトロニクス社（Tektronix Inc.）を含む他のグループは，汎用のプロセサを使って Smalltalk を効率良く実現しようとした[8)~10)]。

われわれは，Smalltalk の実行速度を遅くしている処理を分析し，それを取り除く道を選んだ。速度を低下させているのは，ガーベジ・コレクションに使うリファレンス・カウント（reference count）の計算，オブジェクトの割り付け/解放，中間言語であるバイトコードのテストなどである（詳細は後述）。こうした処理を汎用マイクロプロセサ（68010）上で最適化し，実行速度を上げた。最適化しない Smalltalk システムに比べ，実行速度は 400 倍程度に向上した。

注 1）問題の表現方式や，プログラムのしかたなどを指す（本誌）。
注 2）Smalltalk-80 は米 Xerox Corp. の登録商標である。

図 1　4404 の外観　14 型のモノクローム・ディスプレイ，40 M バイトの固定ディスク装置，フロッピー・ディスク装置，キーボード，マウスなどで構成する。主記憶容量は標準 1 M バイト。プロセサは 68010 を使う。

Smalltalk の実現とは仮想マシンのエミュレータを作ることである

Smalltalk の開発者は，Smalltalk を多種のコンピュータで使えるようにしようと考えた。こうした移植性に対し，二つの問題が生じた。オブジェクトの汎用的な表現方法をどのように決めたらいいか，Smalltalk のプログラミング・ツールを移植可能にするにはどうすればよいか，という二つの問題である。第 1 の問題は，データと手続きを一体化した Smalltalk のオブジェクト指向という性質から生じた。2 番目の問題は他のプログラミング言語にも共通している。

この問題を解決するため，開発者は Smalltalk システム

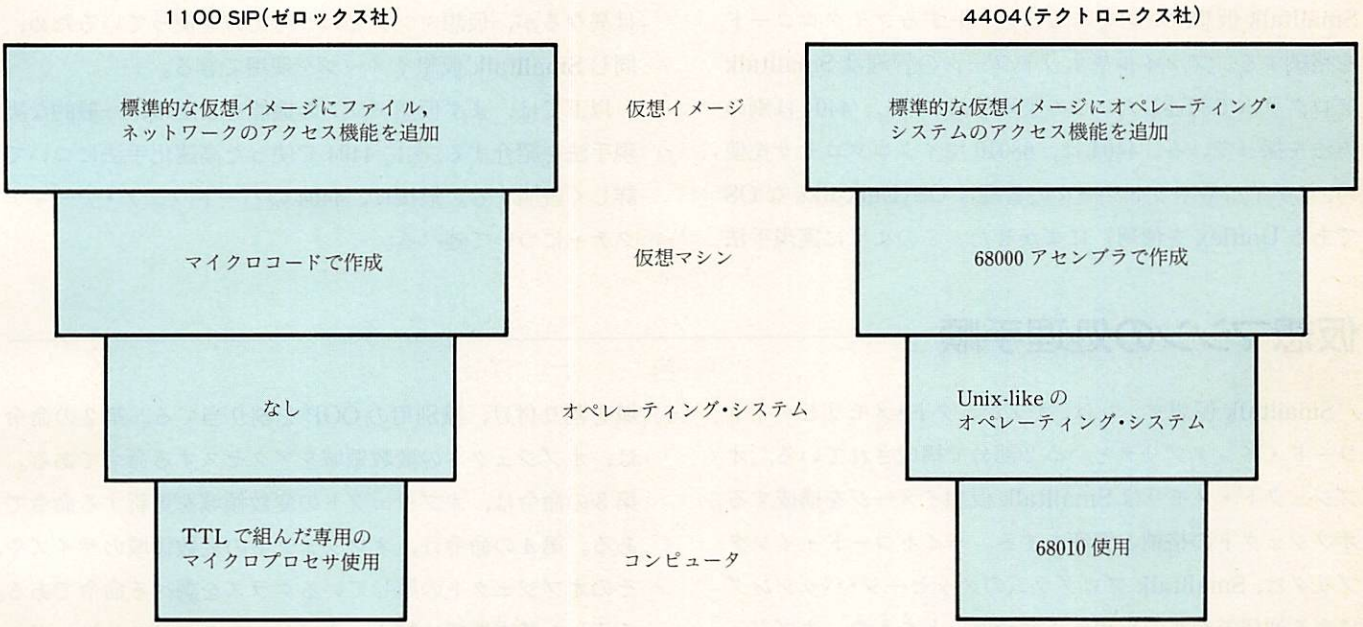


図２　Smalltalk システムの階層構成　Smalltalk システムは，仮想イメージ（virtual image）と仮想マシン（virtual machine）に分かれる。仮想イメージは，Smalltalk のプログラミング環境になっており，コンパイラ，デバガなどを含む。仮想マシンは，仮想イメージを実行するプログラムである。図中には，２通りの実現形態を示した。ゼロックス社の 1100 SIP では，専用プロセサ上にマイクロコードで仮想マシンを記述している。オペレーティング・システム（OS）は使わない。ファイル管理やネットワーク管理は Smalltalk 仮想イメージ自体で行なっている。これに対し，4404 では，汎用マイクロプロセサ 68010，Unix-like の OS の上に仮想マシンを作成した。ファイル管理などは OS にまかせる。仮想イメージには，OS のアクセス機能を追加した。実現手法は異なるが，1100SIP と 4404 の仮想マシンの機能は基本的に等しい。仮想イメージはどのコンピュータ用に作った仮想マシン上でも実行できる。

をプログラミング環境に当たる仮想イメージ(virtual image)と，Smalltalk 用のアーキテクチャである仮想マシン(virtual machine) という二つのソフトウェアに分離する方向を採った。仮想イメージは，オブジェクトの集まりで，ハードウェアや仮想マシンから独立している。したがって，どの仮想マシン上でも実行できる。仮想マシンは，仮想イメージを実行するソフトウェアである。オブジェクトの管理や表現，Smalltalk の実行メカニズムであるメッセージ・パッシングを直接実行する。すべての実行状態はオブジェクトとして表現する。したがって，デバガのような Smalltalk プログラムは，実行状態を直接参照したり，操作したりできる。

Smalltalk の実現とは，ホスト・コンピュータの機能を生かし，仮想マシンのシミュレータあるいはエミュレータを作ることである。実現にはどんなソフトウェア技術を使ってもよい。必要なことは，そのようなすべての実現方式が Smalltalk プログラムから見えないようにすることである。**図２**に，異なる二つのコンピュータの実現手法を比較した。

ゼロックスの 1100 SIP は，書き換え可能な制御記憶を備えた TTL の専用プロセサを使用している。制御記憶に，

略語一覧(ABC 順)

CRT : cathode ray tube
OOP : Object Oriented Pointer
OS : operating system
RAM : random access memory
ROM : read only memory
TTL : transistor transistor logic
ZCT : Zero Count Table

Smalltalk 仮想マシンをエミュレートするマイクロコードを格納する。ファイルやネットワークの管理は Smalltalk プログラムで記述した。この実現手法に対し，4404 は別の方法を採っている。4404 は，68010 マイクロプロセサを使い，ファイルやネットワークの管理は OS（Unix-like な OS である Uniflex を使用）にまかせた。このように実現手法は異なるが，仮想マシンという考え方を使っているため，同じ Smalltalk 仮想イメージを使用できる。

以下では，まず仮想マシンの機能をまとめ，一般的な実現手法を紹介する。次に 4404 で使った高速化手法について詳しく説明する。最後に，4404 のハードウェア・アーキテクチャについて述べる。

仮想マシンの処理手順

Smalltalk 仮想マシンは，オブジェクト・メモリとバイトコード・インタプリタという 2 部分で構成されている。オブジェクト・メモリは Smalltalk 仮想イメージを構成するオブジェクトの格納と管理をする。バイトコード・インタプリタは，Smalltalk プログラムのメッセージ・パッシングによる通信モデルを実現する命令セットをもつ。オブジェクト・メモリはオブジェクトの静的なデータ格納場所を提供するが，バイトコード・インタプリタは，オブジェクトの動的な振る舞いを表す。また Smalltalk 仮想マシンは，多くの原始メソッド（Smalltalk のメソッドは，従来型言語でいう手続きとよく似ている）も含む。原始メソッドは Smalltalk プログラムで実現しにくく効率を落としてしまう機能を提供する。

オブジェクト・メモリは，仮想イメージで実行中のすべてのオブジェクトの格納場所を提供する。各々のオブジェクトには名前が付いており，識別子であるポインタ（Object Oriented Pointer：OOP）によって参照することができる。オブジェクトは，いくつかの変数領域の集合である。各々の変数領域は，普通 OOP（他のオブジェクトへのポインタ）を含んでいるが，2 進データを含むこともある。属しているクラスを指す OOP もオブジェクトのなかに含まれる。

オブジェクト・メモリは，オブジェクトの操作とガーベジ・コレクションの機能を備える

オブジェクト・メモリは，オブジェクトを操作する 4 種類の命令を提供する。最初の命令は，あるクラスに属する特定の大きさのインスタンスを生成する命令である。この命令は，オブジェクト・メモリ中にオブジェクトの格納領域を割り付け，識別用の OOP を割り当てる。第 2 の命令は，オブジェクトの変数領域をアクセスする命令である。第 3 の命令は，オブジェクトの変数領域を更新する命令である。第 4 の命令は，オブジェクトの変数領域のサイズや，そのオブジェクトの属しているクラスを調べる命令である。こうした基本機能に加え，オブジェクト・メモリは，ガーベジ・コレクションも行なう。ガーベジ・コレクションとは，すでに使うことのないオブジェクトの占めている領域を自動的に回収し，再び使えるようにすることである。

バイトコード・インタプリタは，仮想マシンの命令セット（バイトコード）を実行する。この命令セットは，オブジェクトへのメッセージ・パッシング機能を備えている。バイトコードの命令長は，1 バイトである。バイトコード・インタプリタは，実行環境の保持用にオブジェクト・メモリを利用する。この情報は Smalltalk のメソッドをコンパイルしたバイトコード列と，バイトコードやメソッドの実行に使う制御情報，データを含む。オブジェクト・メモリは，オブジェクトを格納するだけなので，バイトコード・インタプリタには，実行状態を保持するオブジェクトのクラスをいくつか定義してある。**図 3** にバイトコード・インタプリタの実行状態を表す典型的なオブジェクトの集まりを示した。

バイトコード・インタプリタは，CompiledMethod というクラスのインスタンスに当たるオブジェクトを扱う。このオブジェクトは，Smalltalk のメソッドのソース・コードをコンパイルしたバイトコードや定数データを含む。実際の命令セットは，条件，無条件分岐を実行する従来型の命令に加えてスタック指向の評価モデルを使っている。サブルーチン呼び出しやデータ操作用の命令を含んでいず，他

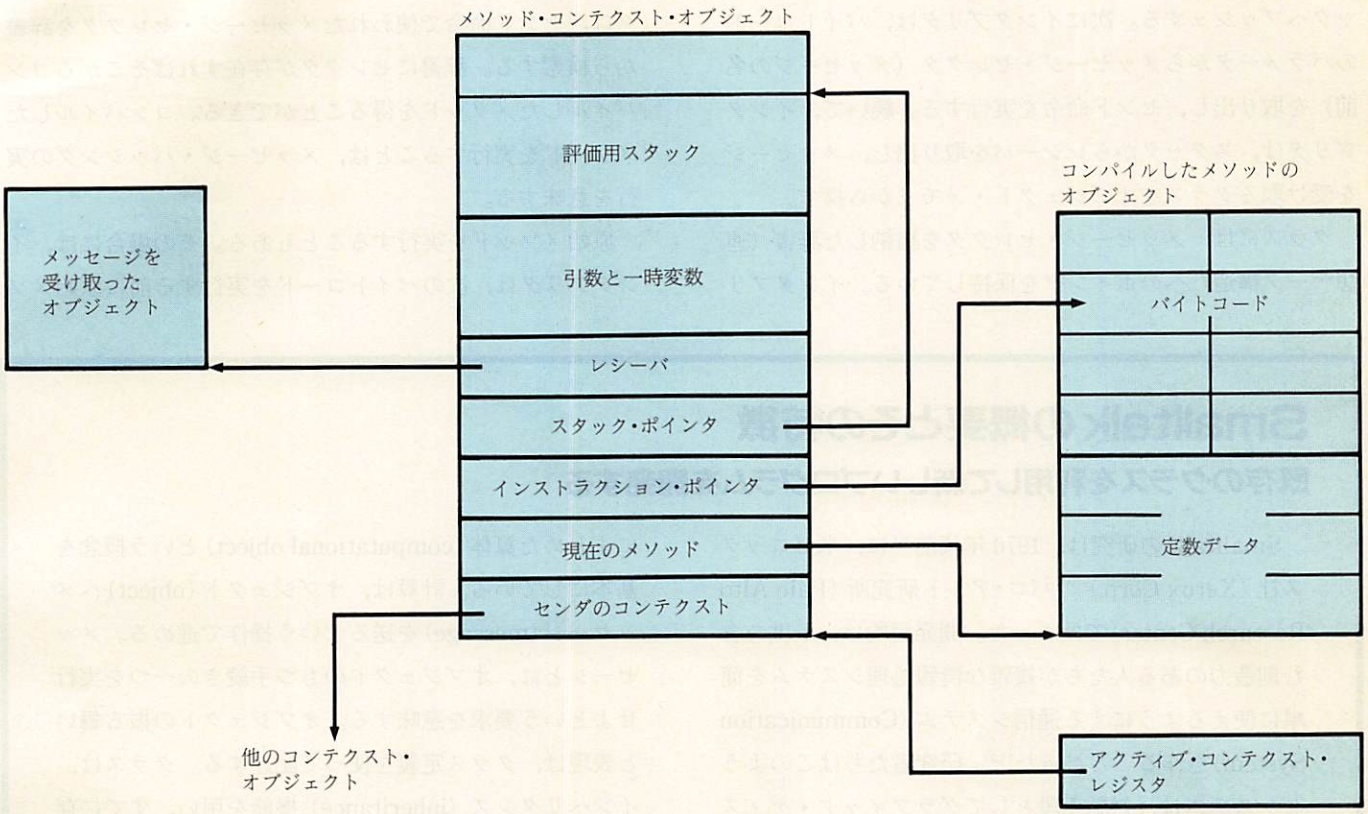


図３　メソッド実行中の状態　メソッド呼び出しが起こると，MethodContext というクラスのインスタンスが生成される。このオブジェクトは，評価用スタック，引数や一時変数，レシーバ・オブジェクト，スタック・ポインタ，実行中のバイトコードを指すインストラクション・ポインタ，センダのコンテクストを含んでいる。コンパイルしたメソッドは，バイトコード列と定数データから成る。

の言語の命令セットと異なる。その代わり，オブジェクトへメッセージを送るための命令を含む。メッセージ・パッシング命令は，手続き呼び出しとよく似ている。ただし，手続き呼び出しでは通常呼び出す手続きが静的に決まっているが，メッセージ・パッシングでは，実行すべきメソッドは動的に決まる。命令セットはメッセージの呼び出しから復帰するためのリターン命令も含む。

実行環境をコンテクスト・オブジェクトとして保存

実行中のメソッドやメソッド内の命令の識別子を保持する評価用スタックは MethodContext というクラスのインスタンスである(図３参照)。このスタックには，メソッドに渡す引数やメッセージを受け取るオブジェクト（レシーバ），メッセージの送り元のコンテクストなどを含んでいる。コンテクストは，そのメソッドで使う一時変数用の領域を含む。要するにメソッド・コンテクストとは，メッセージが送られたときに動的に作られるアクティベーション・レコード（手続き呼び出しの履歴）である。他の言語で使うスタック・フレームとの違いは，Smalltalk ではコンテクストをオブジェクトとして表している点にある。この結果，Smalltalk プログラムで直接操作できる。この方法により，一般的には，コンテクストをスタックに割り付けたり解放したりする処理は要らない。コンテクスト解放のためにガーベジ・コレクションをする必要もなくなる。

メッセージ・パッシング用の実行コードは，一連のスタック・プッシュ命令（バイトコード）に，メッセージ・センド用のバイトコードを続けた形式をしている。プッシュ命令は，レシーバと引数を現在のコンテクストを表すスタ

ックへプッシュする。次にインタプリタは，バイトコードのパラメータからメッセージ・セレクタ（メッセージの名前）を取り出し，センド命令を実行する。続いて，インタプリタは，スタックからレシーバを取り出し，メッセージを受け取るクラスをオブジェクト・メモリから探す。

クラスには，メッセージ・セレクタを格納した辞書（連想データ構造）へのポインタを保持している。インタプリタは，センド命令で使われたメッセージ・セレクタを辞書から検索する。辞書にセレクタが存在すればそこからコンパイルしたメソッドを得ることができる。コンパイルしたメソッドを実行することは，メッセージ・パッシングの実行を意味する。

原始メソッドを実行することもある。その場合には，インタプリタは，次のバイトコードを実行する前に原始メソ

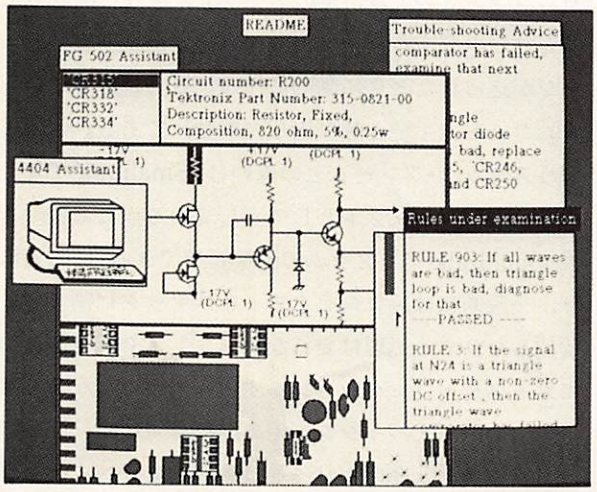
Smalltalkの概要とその特徴

既存のクラスを利用して新しいプログラムを開発する

Smalltalkの研究は，1970年代前半に，米ゼロックス社（Xerox Corp.）のパロ・アルト研究所（Palo Alto Research Center）で始まった。開発目的は，子供を含む創造力のある人たちが複雑な情報処理システムを簡単に使えるようにする通信システム（Communication System）を作ることだった[1),2)]。研究者たちはこのようなシステムは，対話手段としてグラフィック・ディスプレイを備えた高性能のパーソナル・コンピュータ上で実現すべきだと考えた。

Smalltalkは，データとデータに対する操作を一つにまとめた算体（computational object）という概念を基本にしている。計算は，オブジェクト（object）へメッセージ（message）を送るという操作で進める。メッセージとは，オブジェクトのもつ手続きの一つを実行せよという要求を意味する。オブジェクトの振る舞いと表現は，クラス定義を使って記述する。クラスは，インヘリタンス（inheritance）機能を用い，すでに存在しているクラスを改良して定義できる。

Smalltalk-80システムは，この研究の現時点での成果である。システムは，オブジェクト指向プログラミング言語とディスプレイ指向のプログラミング環境を含む。さらに，データ構造から，制御，グラフィックス，アプリケーションまでにわたる200以上のクラスを含む。Smalltalk-80システムはすべてSmalltalk-80自身で記述した。新しいクラスを定義するとき，インヘリタンス機能より，すでに存在するクラスと結び付けることができる。このようなビルディング・ブロック（building-block）方式は，応用プログラムの開発に有効である。複雑で，応答性の良い，データ構造主体の応用，たとえばFreilingとAlexanderのElectronics Troubleshooter[3)]やFinzerとGouldのProgramming by Rehearsal Systemなどの開発に，このプログラミング・スタイルは効果的であった。図AにElectronics Troubleshooterの表示画面を示す。



図A　Smalltalkの表示画面の例

ッドを実行する。もし，原始メソッドが呼び出されない場合には，新しいメソッドの実行状態を示すメソッド・コンテクストを作らなければならない。インタプリタは，オブジェクト・メモリ内にメソッド・コンテクストを確保する。そして，そのコンテクスト，レシーバへのポインタ，新しいメッセージに対応するコンパイルしたメソッド，現在のコンテクストへのポインタなどを書き込む。メッセージの引数は，現在のコンテクスト・スタックから取り出し，新しいコンテクストへ複写する。最後に，新しいコンテクストを現在のコンテクストから作る。インタプリタは新しいメソッドの最初のバイトコードを取り出し実行する。

Smalltalk 仮想マシンをどのように実現しようと，オブジェクト・メモリとバイトコード・インタプリタの全機能を含むようにする必要がある。どんな方法で実現してもよい。最適な実現方法を選ぶことは，高性能な Smalltalk インタプリタを設計するうえで最も重要なことである。

仮想マシンのどの処理が実行速度を低下させているか

Smalltalk 仮想マシンを構成する最も簡単な方法は，参考文献 11）に従ってオブジェクト・メモリ，バイトコード・インタプリタ，原始メソッドの仕様を満たすプログラムを作ることである。この実現方法では，オブジェクト・メモリは単純なメモリの割り付けやリファレンス・カウンティング方式のガーベジ・コレクションを使用する。インタプリタは一般に，オブジェクト・メモリ内のコンテクストやコンパイルしたメソッドを操作する。これらの手順は，仮想マシンの仕様のなかに厳密に定義されている。この実現方法では，オブジェクト・メモリとインタプリタは機能的に分離したモジュールとして定義してある。この方法を採ると，インタプリタを非常に短期間で作成できる。

2〜3 の最適化手法を用いて，この簡単な仮想マシンの実行効率を大幅に改善できる。インタプリタはメッセージ検索用のキャッシュ（message lookup cashe）を使うことで，メッセージ・セント時のメソッド検索時間を短縮できる。またプロセサ内のレジスタにコンテクストの値（スタック・ポインタやスタック・トップの値など）を格納することにより，現在のアクティブ・コンテクストへのアクセスを高速にできる。オブジェクト・メモリ操作用のサブルーチンの代わりに，オブジェクト・メモリ中のデータを直接にアクセスすると，内部モジュール間の通信に伴うオーバヘッドを減らせる。こうした最適化は，マイクロコードで記述した Smalltalk の実現では効果的だった。しかし，われわれのマイクロプロセサ上への実現ではそれほどの効果を上げられなかった。この実現は，最も遅いマイクロコードで記述したインタプリタより 10 倍以上も遅かった。

メモリ管理のオーバヘッドを減らせば実行速度を上げられる

こうした結果から，実用に耐える実行速度を得るには，仮想マシンを別の方法で実現する必要があることがわかった。一方，仮想マシンが遅い理由を知るために，このインタプリタを使って実行特性を調べた。

測定によれば，仮想マシンの実行時間のうち 70% 以上は，オブジェクト・メモリの管理に費やされている。処理の中心は，ほとんどがオブジェクトの割り付け/解放と，リファレンス・カウンティングだった。また別の測定では，仮想マシンの実行中に生成したオブジェクトの 90% 以上がコンテクスト・オブジェクトであるという結果を得た。インタプリタはほとんどの時間をアクティベーション・レコード（コンテクスト）のガーベジ・コレクションに費やしている。

実際のバイトコード・インタプリタを解析すると，わずかの時間（全体の数%）のみが，バイトコードの機能を遂行するのに使われていることがわかった。事実，ほとんどの時間は，めったに起こらない状態をテストしたり，ハードウェアによって操作可能なデータ表現と仮想マシンが操作するデータ表現との間の変換に費やされている。

こうした結果から，メモリ管理のオーバヘッドを取り除けば，効率を向上させられることがわかる。さらに，データ表現の変換や特殊な場合のテストを減らせば実行速度を改善できる。この考察を基に，新しいインタプリタを設計した。新しいインタプリタは 4404 の Smalltalk 仮想マシンの実現に生かされている。

仮想マシンから不要な処理を取り除き，高速化を実現

4404では，Smalltalk仮想マシンを実現する際に次の三つの基本原理に従った。これらの基本原理は，仮想マシンの設計や実現方法の選択に当たって繰り返し適用された。

第1の原理は，実行速度を向上させる最も良い方法は，不必要な計算を完全に取り除く，ということである。より高速なアルゴリズムを見つけることや，特定の計算用に効果的なコード列を見つけ出す前に，やることはある。なぜその計算が必要なのか，問題を再構成することでその計算を除去できないか，を別の視点から調べるべきである。最も高速なコード列は空のコード列であるから。

第2の基本原理は，どうしても計算の必要な場合に使う。すべてのアルゴリズムおよびコード列は，最もよく実行する部分を最適化すべきで，めったに実行しない部分は処理速度が遅くても無視してもよいという原理である。めったに起こらない処理を高速化しても，よく使う処理のオーバヘッドは減らせない。

たとえば，整数同士の算術演算は非常にひんぱんに起こる。このため，算術演算はオペランドが二つとも整数のとき最も実行時間を減らせるように最適化する。オペランドが整数以外の場合は少なく，その実行効率を低下させてもかまわない。

第3の基本原理は，データ構造設計の指針である。最適なデータの表現方法は，直接あるいは簡単にホスト・コンピュータでそれを操作できるようにすることである。仮想マシンのデータ表現とホスト・コンピュータのデータ表現との間の変換命令は最小にすべきである。

これらの基本原理に従い，4404仮想マシン設計の第1段階では，除去できる不必要な計算は何かを抽出した。以前の実現では，実行時間の70%をメモリ管理に費やしていた。このメモリ管理のなかから不必要な計算を見つけるのが重要である。

(a) ソース・プログラム

```
a←b+c
```

(b) コンパイル後のバイトコード

```
push b
push c
send+
pop a
```

(c) リファレンス・カウンティング使用時の処理手順

```
push b
  [bの値のリファレンス・カウントを1増やす]
push c
  [cの値のリファレンス・カウントを1増やす]
send+
  [cの値のリファレンス・カウントを1減らす]
  [bの値のリファレンス・カウントを1減らす]
  [b+cの値のリファレンス・カウントを1増やす]
pop a
  [aの以前の値のリファレンス・カウントを1減らす]
  [b+cの値のリファレンス・カウントを1増やす]
  [b+cの値のリファレンス・カウントを1減らす]
```

(d) 改良したリファレンス・カウンティング使用時の処理手順

```
push b
push c
send+
pop a
  [aの以前の値のリファレンス・カウントを1減らす]
  [b+cの値のリファレンス・カウントを1増やす]
```

図4　コンテクストから参照されているオブジェクトのリファレンス・カウントをしないと，リファレンス・カウントの数を大幅に減らせる　(a)のプログラムをコンパイルすると(b)のようなバイトコード列となる。通常のリファレンス・カウント方式では(c)のように，この処理中8回リファレンス・カウントをしなければならなかった。これを改良すると，リファレンス・カウント2回ですむ(d)。

コンテクストから参照されているオブジェクトはリファレンス・カウントしない

メモリ管理の大部分は，リファレンス・カウンティングに費やされている。DeutschとBobrowの提案したディファード・リファレンス・カウンティング(deferred reference counting)法は，リファレンス・カウンティングに要する大部分の時間を減らす技術である。この方法は，次のような事実を基にしている。つまり，新しくオブジェクトを参照するのはアクティベーション・レコード（コンテクスト）からがほとんどだが，アクティベーション・レコードはすぐ

に消滅してしまう，という事実である。すなわち，アクティベーション・レコードからのリファレンス・カウンティングをする必要はない。これを除去できれば，ほとんどのリファレンス・カウンティングは除去できる。この技術はSmalltalk仮想マシンに適用できる。

4404のSmalltalkの実現では，コンテクストはリファレンス・カウンティングをしない特別なオブジェクトとして扱う。コンテクスト以外のオブジェクトからのリファレンスは数える。この方法を採ることで，コンテクストへオブジェクトを格納したり，コンテクストからオブジェクトを取り出すバイトコードの処理から，リファレンス・カウンティングのコードを取り除くことができる。

図4に簡単なプログラムでその例を示した。プログラムは，bとcを加え，それをaに代入する(a←b＋c)という意味である。このプログラムをコンパイルすると同図(b)のような四つの命令になる。バイトコードは実際は0～255の値になるが，ここでは意味がわかりやすいようにニーモニックで示した。

通常のリファレンス・カウンティングでは，同図(c)のように処理を進める。“push b”でメソッド・コンテクスト中の評価スタックにbの値をプッシュする。このとき，bの値は，メソッド・コンテクストというオブジェクトから参照されていることになるので，bの値に対するリファレンス・カウントを1増やさねばならない。以下の処理は，cの値をスタックへプッシュし(push c)，スタック中の二つの値を加え（send＋），結果をスタックからポップする（pop a)。加算の後では，cとbの値はもうコンテクストから参照されないので，リファレンス・カウントをそれぞれ1減らさなければならない。また，加算の結果得たb＋cの値のリファレンス・カウントを1増やす。このように，単純な加算の例でも，全体で8回リファレンス・カウントを計算する必要があった。

4404では，コンテクストから参照されているオブジェクトのリファレンス・カウントはしない。したがって，図4(d)に示すように，評価スタック中のオブジェクトのリファレンス・カウンティングはしないですむ。aの値は，以前の値からb＋cの値に変わった。その部分だけリファレンス・カウンティングすればよい。

こうした最適化は，リファレンス・カウンティングに関する大部分のオーバヘッドを除去できる。さらに，非常によく実行するバイトコードや原始メソッドに対するコード列を単純に（高速に）できる。たとえば，プッシュ命令は68010の“move”命令を使って実現できる。リファレンス・カウントを読み出したり，増加/減少させたり，調べたりするコードは不必要になる。

ディファード・リファレンス・カウンティング方式はリファレンス・カウンティングのオーバヘッドを減らす。しかし，ごみ（garbage）となったオブジェクトの解放は難しくなる。通常のリファレンス・カウンティングでは，リファレンス・カウントが0になったオブジェクトをごみとして解放する。ディファード・リファレンス・カウンティングではそうはできない。というのは，リファレンス・カウント0のオブジェクトはコンテクスト以外のオブジェクトからは参照されていないが，コンテクストから参照されているかもしれないからである。単純にごみとは決められない。

ディファード・リファレンス・カウンティングでは，すべてのコンテクスト・オブジェクトからの参照を周期的に調べる必要がある。本当にリファレンス・カウントが0のオブジェクトはこの過程で解放することもできる。この過程の効率を上げるため，4404の実現方式では，そうしたオブジェクトのOOPをゼロ・カウント・テーブル（Zero Count Table：ZCT）という配列に蓄えている。ZCTが一杯になったら，ガーベジ・コレクションのために，コンテクストからの参照を調べればよい。

コンテクストはオブジェクトとせずにスタックに置く

リファレンス・カウンティングのオーバヘッドを除去しても，まだ実行速度を落とす要因はある。実行時間の大部分はオブジェクトの割り付け/解放に費やされているためである。大部分のオブジェクトはコンテクスト（アクティベーション・レコード）である。コンテクスト・オブジェクトを，特別な割り付けルーチンを用意したり，フリー・リストを用意したりして管理することはできる。しかし，

それでは計算の高速化にしかならない。必要なことは，メソッド呼び出しにコンテクスト・オブジェクトを要らなくすることである。こうするとオブジェクトの割り付け/解放に必要な大部分の処理を省ける。

Smalltalk プログラムの実行中には，50% 以上のメソッド・コンテクストは，アクティベーション・ツリーの葉(leaf) であるという事実がある。つまり，ほとんどのメソッドはセンダに復帰するとき，別のコンテクストを生成しない。さらに，ほとんどのコンテクスト・オブジェクトは Smalltalk プログラムからオブジェクトとして操作されない。

これらの事実を基にすると，新しいコンテクストは，さらに新しいメソッド呼び出しをしないと仮定してメッセージ・センド・バイトコードを作ってもよくなる。コンテクストの保存には，オブジェクト・メモリの管理しない静的に作られたスクラッチ・コンテクスト (scratch context) を利用するようにした。スクラッチ・コンテクスト内のコンテクストは，そのメソッドが他のメソッドから呼び出されたり，オブジェクトとして参照されたときに初めてコンテクスト・オブジェクトに変換する。こうするとごみとなるべきオブジェクトの 50% 以上をなくすことができた。

次に，スクラッチ・コンテクストをスタックとして維持するように拡張した。これによって，アクティベーション・ツリーの葉以外のコンテクストもオブジェクトとする必要がなくなった。この改良により，コンテクスト・オブジェクトの管理にかかるほとんどの計算時間を除去できた。

オブジェクト・メモリの管理以外にコンテクストを表すスタックを作ると，ホスト・コンピュータの能力を最も生かせるスタックの表現を選択できる。たとえば，一般的に使われるプッシュ/ポップ・バイトコードを 68000 の単一命令で記述できるようなスタックを作れる。

コンテクストを保存するスタックは，68010 が内蔵する二つのアドレス・レジスタに格納したアドレス間の連続したメモリ空間として表現した。アドレス・レジスタの一つは，コンテクスト・スタックのトップのアドレスを保存する。このアドレスは，実行中のコンテクスト中にある評価スタックのトップを表すことにもなる。プッシュやポップなどの操作は，自動的にインクリメント/デクリメントのできるアドレシング・モードを備えた 68000 の “move” 命令で実現できる。別のレジスタには実行中のコンテクストのベース・アドレスを格納する。一時変数は，ベース・ディスプレースメント・アドレシング・モードを使ってこのレジスタでアドレシングできる。

バイトコードのディスパッチを高速化

メモリ管理やバイトコードの意味に関連した実行のオーバヘッドを減らすと，他の構成要素のオーバヘッドが目立ってくる。このような構成要素の一つに，バイトコード・ディスパッチャがある。このルーチンは，現在実行中のコンパイル後のメソッドから次のバイトコードをフェッチし，そのバイトコードの機能を実現しているルーチンへ制御を渡す。この処理は概念的には単純だが，Smalltalk 仮想マシンの仕様では，各々のバイトコードを実行する前にプロセス・スイッチが必要かどうかを調べなければならず，処理は複雑である。この単純なディスパッチャの設計では，プロセス・スイッチの間に何千ものバイトコードを実行するとしても，各バイトコードの実行前にプロセス・スイッチの可能性を調べなければならない。どんなに注意深くプログラムしても，プロセス・スイッチの検査をするとバイトコードのディスパッチには 2 倍時間がかかってしまう。

このオーバヘッドは，バイトコードのディスパッチ・コードを動的に変更することで除去することができる。バイトコード・ディスパッチャは，単にバイトコードをフェッチし，ディスパッチするだけで，プロセス・スイッチが必要かどうかを検査をしないようプログラムした。プロセス・スイッチが起きたときには，バイトコード・ディスパッチャの最初の命令を，プロセス・スイッチを行なうルーチンへの分岐命令に置き換えるようにした。プロセス・スイッチを行なうルーチンは，ディスパッチャのプログラムを元に戻してから，プロセス・スイッチを実行する。

これらの最適化によって，バイトコード・ディスパッチは 68000 の 6 命令によって，3〜4 μs で実行できるようになった(**図 5**)。最も一般的なバイトコードの処理ルーチンは 68000 の 1 命令で，約 1.5 μs で実行できる。バイトコー

ド・ディスパッチのオーバヘッドは，相変わらずバイトコード実行時間の大部分を占めている。システム・レベルから見れば4404では，実行時間の約20%をバイトコード・ディスパッチャに費やしている（図6）。

バイトコード・ディスパッチのオーバヘッドは，Smalltalkの実現にバイトコード・インタプリタを用いているために起こる。バイトコードを用いる主な利点は，移植性は言うに及ばず，コンパクトなコードを作れる点である。4404の仮想イメージは600Kバイト以上のバイトコードで構成されている。もし，これをマシン・コードに変換するとしたら，5～6Mバイトを占めることになろう。こうした大きなプログラムを実行するときに起こるページングのオーバヘッドは，バイトコード・ディスパッチのオーバヘッドよりも大きくなる。この理由からバイトコードを使うのは，1～2Mバイトの主記憶をもつ4404にとっては良い選択であると思える。物理メモリが増えれば，この選択は考え直す必要があるかもしれない。

われわれの初期の目標は，一連のアプリケーション開発に適当な実行速度をマイクロプロセサ上で実現することだった。目標の実行速度はマイクロコードで実現した最も遅いSmalltalk-80システムだった。実現が完了し，Smalltalk-80システムの標準のベンチマークを実行してみた結果，マイクロコードの実現の2.5倍の実行速度を示した。図7に4404でのベンチマーク・テストの結果を示す。

(a) バイトコード・ディスパッチャの処理手順

(1) プロセス・スイッチが必要かどうかを調べる
(2) インストラクション・ポインタの指している実行中のメソッドから次のバイトコードをフェッチする
(3) インストラクション・ポインタを1増やす
(4) バイトコードをデコードし，呼び出すべきインタプリタのルーチンを決める
(5) そのルーチンへ分岐する

(b) 4404のディスパッチャ

```
moveq #0,d0          ; プロセス・スイッチが起きるときはこの命令を変更する
move.b (A4)+,d0      ; A4には次のバイトコードを格納してある。バイトコードをフェッチし，次のバイトコードのアドレスをセット
add.w d0,d0          ; バイトコードを2倍し，0番地から始まるテーブル（2バイトが1エントリ）のエントリを指すようにする
move.w d0,a0         ; 2倍したバイトコードをベース・レジスタへ移す
move.w (a0),a0       ; 2倍したバイトコードを使ってアドレス・テーブルから必要なルーチンのアドレスをフェッチする
jmp (a0)             ; バイトコードを実現するルーチンへ分岐する
```

図5　バイトコード・ディスパッチャ　処理手順は(a)に示した。この処理を68000の六つの命令で実現した(b)。プロセス・スイッチが起きると，最初の命令（moveq #0, d0）を動的に変更し，プロセス・スイッチ処理用のルーチンへ分岐する。

インタプリタの機能	実行時間中で占める割合
バイトコードの実行	20.7%
バイトコード・ディスパッチ	19.6%
コンテクストの管理	16.9%
メッセージ検索	14.5%
原始メソッド	13.3%
メモリ管理	10.8%
その他	4.2%

図6　インタプリタのどの機能のオーバヘッドが大きいか　バイトコードの実行には処理時間の20%しか使っていない。プロセス・スイッチの検査など，特殊な場合の検査に他の時間を使っている。

add 3+4(10回)	255 μs
メソッドの呼び出し/復帰(3万2000回)	3.77秒
add 3.1+4.1(20回)	3400 μs
bit BLTの呼び出し(10回)	2.14秒
クラス定義の表示(20回)	3.38秒
ダミー・メソッドのコンパイル(5回)	10.8秒
テキストの表示(10回)	4.55秒

図7　4404の仮想マシンのベンチマーク　Smalltalk仮想マシン中に記述してあるベンチマーク・テストを行ない，時間を計測した。

68010 マイクロプロセサを中心にハードウェアを構成

ここまで述べてきたように，マイクロプロセサでも高性能の Smalltalk-80 システムを実現できることがわかった。次の目標は，低価格な机上型コンピュータを設計することである。Lisp, Prolog を含む人工知能向き言語をサポートする必要もあった。

4404 は，クロック 10 MHz の MC68010 を内蔵する（図8）。68000 ファミリを選択したのは，次の理由による。大きな連続アドレス空間，豊富なレジスタ，人工知能開発用言語に向いた自由度のあるアドレシング・モードなどである。算術演算の実行速度を上げるため，浮動小数点演算プロセサ（32000 ファミリの 32081）も使用している。

Smalltalk や他の人工知能向き言語では，大容量のメモリを必要とする。4404 は，標準で 1 M バイトの主記憶を実装している。メモリには，256 K ビット・ダイナミック RAM を使った。オプション・ボードを追加すると，主記憶は 2 M バイトまで拡張できる。仮想記憶機能ももたせた。

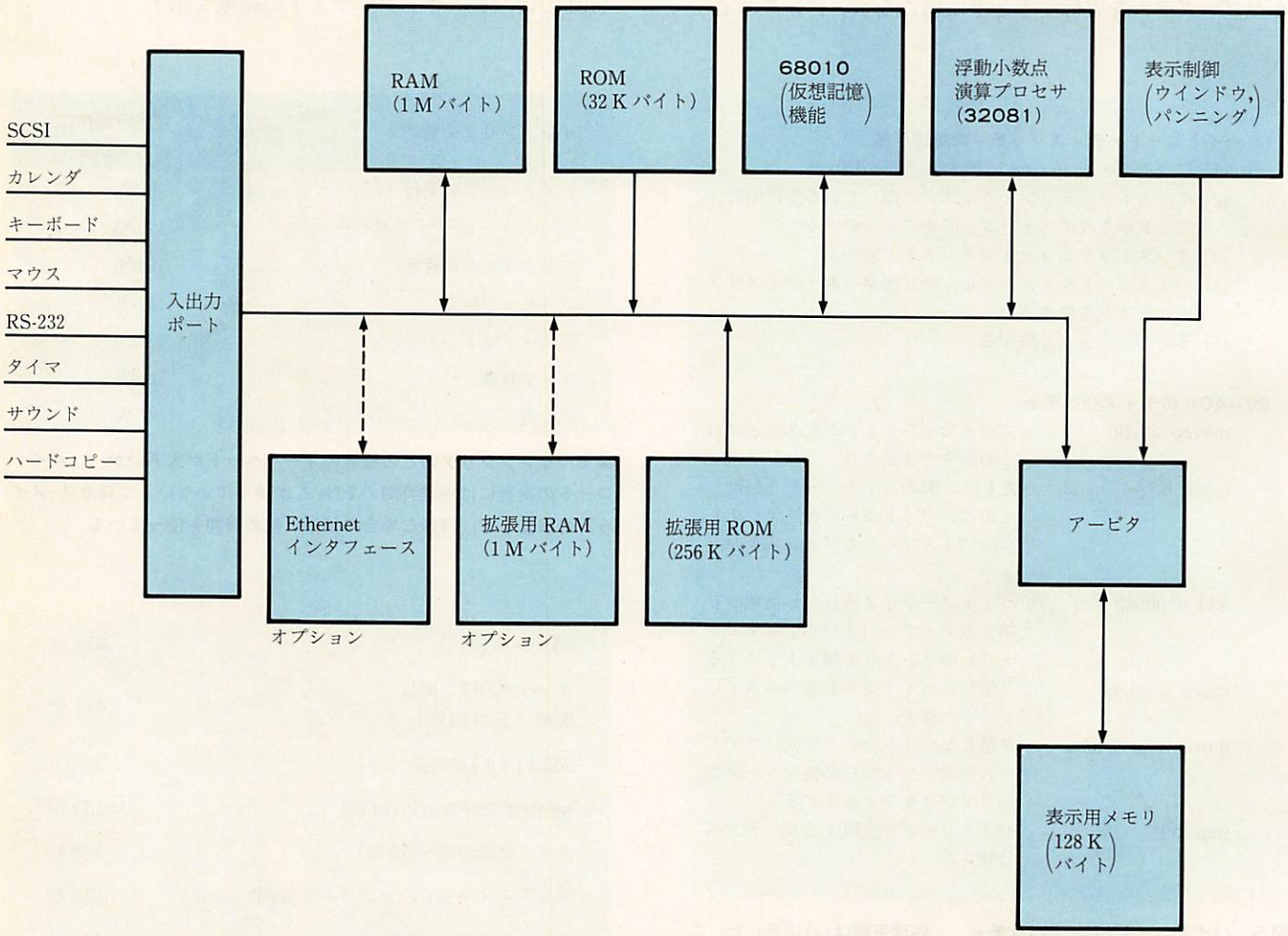


図 8　4404 のブロック図　68010（仮想記憶機能追加），1 M バイトの RAM（標準），32 K バイトの ROM，浮動小数点演算プロセサ（32081），表示制御用ハードウェアなどを中心に構成する。バス構成は採らず，2 枚のボードにオプション以外の部分を実装し，それをつなげた。表示用メモリは 128 K バイトで，1024×1024 画素の画面を蓄えることができる。実際の表示領域は，640×400 画素である。

仮想アドレス空間は8Mバイトである。実行速度を最大限に引き出すため，実記憶および仮想記憶は，待ち状態なしで動作するように設計した。

ビットマップ・ディスプレイは，Smalltalkの環境には欠かせない。4404では，68010のアドレス空間内に直接アクセス可能な1024×1024画素分のフレーム・バッファを置いた。ディスプレイは，そのフレーム・バッファから640×400画素をビューポートとして表示する。フレーム・バッファの全内容を表示するために，マウスを使ってビューポートを滑らかに移動できるようにした。

4404は汎用の内部バス構成を採らなかった。バス中心の構成は，価格を増すばかりでなく，システム全体の実行効率を低下させることにもなる。4404では，標準的な周辺インタフェースを利用できるように設計した。インタフェースとしては，SCSIインタフェース（固定/フロッピー・ディスク用），通信用のRS-232C，プリンタ用のパラレル・インタフェースなどを用意した。Ethernetインタフェースはオプションで追加できる。

4404のプロセサ，メモリなどは，標準的なテクトロニクス社の端末用筐体に収めた2枚のボード上に実装している（**図9**）。増設用メモリとEthernetインタフェースは，各々別の小さなボードに実装する。汎用の筐体を利用したことで，製品の価格を下げることができた。

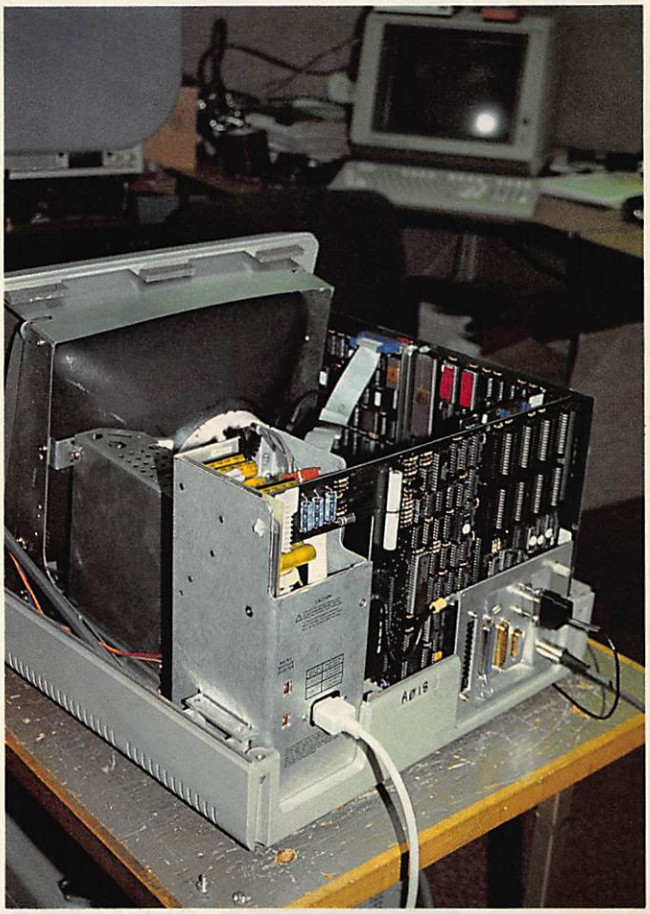


図9　4404の内部　本体筐体内にCRT，プロセサ，メモリなどを実装している。

参考文献

1) Kay, A. and Goldberg, A., "Personal Dynamic Media," *IEEE Computer,* vol. 10, no. 3, pp. 31-41, Mar. 1977.
2) Ingails, D. H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages,* pp. 9-16, Jan. 1978.
3) Alexander, J. and Freiling, M., "Building an Expert System in Smalltalk-80," *Systems and Software,* vol. 4, no. 4, Apr. 1985.
4) Finzer, W. and Gould, L., "Programming by Rehearsal," *Byte,* vol. 9, no. 6, pp. 187-210, June 1984.
5) McCall, K., "The Smalltalk-80 Benchmarks," *Smalltalk-80 : Bits of History, Words of Advice,* Krasner, G. ed., pp. 153-173, Addison-Wesley, 1983.
6) Ungar, D., Blau, R., Foley, P., Samples, D. and Patterson, D., "Architecture of SOAR: Smalltalk on a RISC," *Eleventh Annual International Symposium on Computer Architecture,* June 1984.
7) Suzuki, N., Kubota, K. and Aoki, T., "Sword 32: a Bytcode Emulating Microprocessor of Object-Oriented Languages," *Proceedings of FGCS 84,* Nov. 1984.
8) Deutsch, L. P. and Schiffman, A. M., "Efficient Implementation of the Smalltalk-80 System," *Proceedings of the 11th Annual ACM SIGART-SIGPLAN Symposium on the Principles of Programming Languages,* pp. 297-302, Jan. 1984.
9) Suzuki, N. and Terada, M., "Creating Efficient Systems for Object-oriented Languages," 同上, pp. 290-296, Jan. 1984.
10) Wirfs-Brock, A., "Design Decisions for Smalltalk-80 Implementors," *Smalltalk-80 : Bits of History, Words of Advice,* Krasner, G. ed., pp. 41-56, Addison-Wesley, 1983.
11) Goldberg, A. J. and Robson, D., *Smalltalk-80 : The Language and its Implementation,* Addison-Wesley, 1983.
12) Deutsch, L. P. and Bobrow, D. G., "An Efficient Incremental Automatic Garbage Collector," *Communications of the ACM,* vol. 19, no. 9, pp. 522-526, Sept. 1976.