スロット仮想空間によるオブジェクト間通信の高速化

光澤 敦[†] 横手靖彦[††] 所 真理雄[†,††]

本論文では，オペレーティングシステムにおいて同一計算機内のオブジェクト間通信を高速化するための仮想記憶管理技法について述べる．本技法では，仮想空間を同じ大きさの断片（スロットと呼ぶ）に分割し，オブジェクトに一つあるいは複数のスロットを与える．これにより，(1)複数のオブジェクトが仮想空間を共有するため，同一計算機内のオブジェクト間通信時に，コンテキストスイッチとそれに伴う TLB (Translation Lookaside Buffer) エントリおよびキャッシュエントリの削除が必要なくなり，(2)仮想空間間でオブジェクトを移動することにより，異なる仮想空間間での通信頻度を削減でき，(3)オブジェクト間通信を手続き呼び出しに書き換えることにより，通信速度の高速化と通信経路の最適化ができる．本技法をオペレーティングシステム Apertos に実装し，*Null* メソッドの実行時間を測定した．その結果，従来の各オブジェクトに独立の仮想空間を与える場合と比較して，(1)および(2)によりオブジェクトが仮想空間を共有した場合で 1.7 倍から 4 倍の高速化が，さらに(3)と組み合わせることにより約 200 倍から約 400 倍の高速化が達成できた．

Fast Inter-Object Communication Using Slotted Virtual Space

Atsushi Mitsuzawa,[†] Yasuhiko Yokote[††] and Mario Tokoro[†,††]

In this paper, we propose the slotted virtual memory management scheme which enhances performance of inter-object communication within the same machine in operating systems. It divides a virtual space into equal sized fragments, called slots, and gives each object one or more slots. It enables following techniques: (1) Context-switches and flushes of both translation lookaside buffer (TLB) entries and cache entries, become unnecessary, by placing several objects into the same space. (2) Frequencies of communication between virtual spaces can be reduced by moving objects dynamically. (3) The execution of extra paths can be avoided by replacing them with local procedure calls. Measures on the Apertos operating system shows that an invocation of *Null* method becomes 1.7 to 4 times faster due to (1) or (2) and about 200 to 400 times faster due to (3) compared to conventional management schemes which give each object a distinct virtual space.

1. はじめに

ユーザからの要求や管理すべきハードウェアの多様化により，オペレーティングシステムはその規模が拡大し，複雑さを増してきている．オペレーティングシステムをオブジェクト指向技術を用いて構築する機運が高まっているが[11]，その理由はオブジェクト指向技術がシステムのモジュール化を進め，モジュールの再利用やシステムの管理を容易にできるからである．我々は，大規模分散環境を実現するための開放的かつ自己発展可能なオペレーティングシステム Apertos☆をオブジェクト指向技術を用いて開発している[13]．

Apertos はオブジェクトとメタオブジェクトの分離，およびリフレクションを基本としたオブジェクトアーキテクチャをもとに構築されている[14]．システムの基本要素はオブジェクトであるが，オブジェクトからメモリ管理，メッセージ通信，同期等のメタ機能をメタオブジェクトとして分離している．メタオブジェクトのグループはメタオブジェクト空間として管理され，オブジェクトの実行環境を提供している．ここでメタオブジェクトはオブジェクトとして定義されるため，メタオブジェクトを支援するためのメタオブジェクト空間が必要になる．このようにメタオブジェクト空間は階層構造になる．また Apertos は，Eden[3]，Amoeba[12]，Chorus[10] 等のオブジェクト指向オペレーティングシステムのように，カーネルとそのカーネルによってサポートされるオブジェクトからシステムが構成されるのではなく，システムの構成要素はすべて

† 慶應義塾大学理工学部計算機科学科
Department of Computer Science, Faculty of Science and Technology, Keio University
†† (株)ソニーコンピュータサイエンス研究所
Sony Computer Science Laboratory Inc.

☆ Apertos（アペルトスと読む）の名称は，イタリア語で開放性を表す "Aperto" と OS (Operating System)" の造語である．

オブジェクトである，という特徴を持っている．すなわち，カーネル自体もオブジェクト指向技術を用いて構築されている．

システムをすべてオブジェクトで構成する場合，次の2つの理由により，オブジェクトのメソッド実行速度がオペレーティングシステムの性能に大きく影響する．

1. オブジェクトの粒度が細かくなると，オブジェクト間通信の頻度が増加する．
2. メタオブジェクト空間の階層が深くなると，通信経路が長くなる．

しかし，RPC (Remote Procedure Call) を始めオペレーティングシステムが提供するオブジェクトやプロセスの呼び出しは，言語処理系における関数／手続きの呼び出しと比較してオーバヘッドが大きい．そのため，UNIX のような巨大なカーネルがオペレーティングシステムのほとんどすべてのサービスを実現する単層システム (monolithic system) と比較すると，オブジェクト指向技術によるオペレーティングシステムは性能的に劣るという問題が生じる．同様の問題はマイクロカーネル技術を用いたオペレーティングシステムである Mach[15] をはじめ，モジュール化および階層化されたオペレーティングシステムでも生じる．

そこで本論文では，オブジェクトのメソッドを高速に呼び出すための仮想記憶管理技法として，スロットによる仮想記憶管理技法を提案する．この技法では，仮想空間をスロットと呼ばれる同じ大きさの断片に分割し，オブジェクトに仮想空間全体ではなく1つあるいは複数のスロットを与える．そして，以下の手法と組み合わせることにより，同一計算機内に存在するオブジェクトのメソッド実行を高速化する．

（1） 複数のオブジェクトに仮想空間を共有させることにより，通信時に大きなオーバヘッドとなっていたコンテキストスイッチとそれに伴う TLB エントリおよびキャッシュエントリの無効化が必要なくなる．

（2） ある仮想空間から別の仮想空間へオブジェクトを移動させることにより，異なる仮想空間間での通信頻度を動的に削減できる．

（3） オブジェクトが仮想空間を共有する状態で，オブジェクト間通信を手続き呼び出しに書き換えることにより，通信速度がさらに高速になるとともに，通信経路の最適化ができる．

なお通信には，同一計算機内の通信とネットワーク通信があるが，本研究では同一計算機内の通信に焦点を当てた．これは，以下の2つの理由による．

- 最近のオブジェクト指向技術およびマイクロカーネル技術によるオペレーティングシステムでは，ネットワーク通信が同一計算機内の通信を伴うため，ネットワーク通信より重要性が高いと考えられる．例えば Mach では，ネットワーク通信機能がマイクロカーネル内でなくサーバとして実装されているため，ネットワーク通信がマイクロカーネルとサーバ間およびサーバとサーバ間の通信といった同一計算機内の通信を伴う．
- 単層システムである UNIX やシステムの機能が複数のサーバで構成されるVシステムにおける測定では，両者ともに RPC のうち 94% 以上が同一計算機内の通信であった[1]．

さらに，本論文ではスロットによる仮想記憶管理技法とそれに基づく手法を，ソニー PWS-1550 ワークステーション☆ 上の Apertos に実装し，オブジェクトのメソッド実行時間を測定する．実装したシステムは，仮想記憶管理，クラスシステム，および通信管理の3つからなる．仮想記憶管理は，計算機アーキテクチャに依存する部分と独立した部分がそれぞれ1つのオブジェクトとして実装され，仮想記憶領域の割当／開放を行う．クラスシステムは，仮想記憶管理の機能を用いてオブジェクトの生成／消去を行い，またスロットの管理を行う．通信管理は，手法(2)および手法(3)を効果的に行うために，通信の統計情報を収集する．

最後に，本論文の構成を以下に示す．まずオペレーティングシステムにおけるモジュール間通信の高速化技法に関してこれまで行われてきた研究について触れ，本研究との関連を明らかにする．次に，スロットによる仮想記憶管理，およびオブジェクトのメソッド実行を高速化するための3つの手法を提案する．第3には，Apertos における実装について簡単に述べた後，オブジェクトのメソッド実行時間および今回の実装に関する評価を行う．最後に，今後の課題について述べ，まとめる．

2. 関連研究

単層オペレーティングシステムにおける同一計算機内のモジュール間通信の一般的な管理を示したのが図

☆ Motorola MC68030(25 MHz) で主記憶 8 Mbytes (最大 16 MBytes).

1 である．一般に，*ObjectA* と *ObjectB* は異なる仮空想間が与えられ，カーネル (*Kernel*) は両者の仮想空間にマッピングされている．*ObjectA* から *ObjectB* のメソッドを実行する場合，まず *ObjectA* はトラップ命令によりカーネルプログラムの実行を行い，カーネルが必要な処理を終えた後，*ObjectA* の実行から *ObjectB* の実行に遷移する．

上記の処理に要する時間の内訳は次のようになる．

(1) 引数の転送時間
(2) コンテキストスイッチ時間
(3) TLB およびキャッシュエントリの削除時間
(4) トラップの処理時間
(5) 送り先のオブジェクトの探索時間

このうち(3)は，それ自体は大きくないが，TLB およびキャッシュエントリの削除がそれらの利用効率を落とすため，システムの性能が落ちる．これに関連して，文献 9) はコンテキストスイッチ時のキャッシュエントリの削除がシステムの性能に非常に大きく影響することを指摘しており，また文献 2) ではオペレーティングシステムのモジュール間通信の性能に TLB およびキャッシュエントリの削除が大きく影響することを指摘している．

これらの時間を軽減することによりモジュール間通信を高速化するための研究が，いくつか行われてきた．

(1)を小さくするために，BSD 系 UNIX では mbuf と呼ばれるバッファを用いて，モジュール間のデータ転送をバッファへの参照を渡すことで実現している[7]．また Mach では，データ転送が仮想記憶の機能によりタスク間でデータ共有することにより実現される[15]．

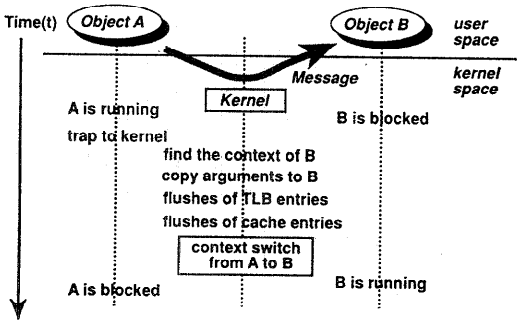


図 1 単層オペレーティングシステムにおける同一計算機内のモジュール間通信の管理
Fig. 1 Typical management of local communication between modules supported by monolithic operating systems.

これは仮想アドレスのエイリアス機能☆ で実現されているため，仮想アドレスキャッシュのアーキテクチャにおいては，キャッシュエントリに無駄が生じることが指摘されている[4]．

(2)および(3)に関しては，*ObjectA* と *ObjectB* が仮想空間を共有することが考えられる．そうした研究に，スレッドと Opal がある．スレッドは，オブジェクト（あるいはプロセス）のコンテキストを少なくするために考案された概念で，実行あるいは制御の流れと定義される．スレッドは仮想空間を共有するため，スレッド間通信において(2)が非常に小さくできるとともに，(3)が必要なくなる．スレッドには，SunOS LWP (Light-Weight Process)[8] のようにユーザレベルで実現されているものと，Mach のように共有メモリ型マルチプロセッサ計算機をサポートするためにカーネルレベルで実現されているものがある．しかし，スレッドはある特定分野のプログラミングを抽象化する概念であったり，共有メモリ型マルチプロセッサ計算機の利点を生かすためのプログラミング技法であり，一般的なオブジェクトの実行の高速化手法ではない．

一方 Opal は，同一機種の計算機が複数ネットワークに接続された分散環境のためのオブジェクト指向オペレーティングシステムである．Opal は分散環境で単一仮想空間を実現しており，オブジェクトは単一仮想空間に配置される．そのため，オブジェクトの名前付けを仮想アドレスで行うことができ，オブジェクトのメソッド実行を言語処理系の手続き呼び出しに類似した方法で行っている．すなわち(2)および(3)が両方とも必要ないため，通信が非常に高速にできる．Opal の難点は，64 ビットの仮想空間と仮想空間内での保護機能を持つ MMU を前提としていることである．

また，同一計算機内の RPC を高速化するものに LRPC[1] がある．第 1 章で述べた RPC のほとんどが同一計算機内のものであるという測定も，LRPC に関する論文中で行われたものである．LRPC では，RPC における引数が多くの場合小さくそして単純な構造（例えば整数）であることに着目し，ネットワーク通信の機能を維持しながら，同一計算機内の通信の高速化を行っている．LRPC では，サーバとクライアントは別々の仮想空間に配置されることを前提としており，サーバとクライアント両方の仮想空間にマッピン

☆ virtual address aliasing.

グされる A-stack と呼ばれる引数や返り値のための領域と，サーバの実行に使われる E-stack と呼ばれる領域を用いて RPC を高速化する．また，LRPC は共有メモリ型マルチプロセッサ計算機に対応しており，そこでは TLB エントリの削除の問題を非常にうまく解決している．

さらに，上記(1)から(5)を直接軽減する技術ではないが，カーネル内の実行を高速化するための機構およびプログラミング技法に Continuation[5] がある．Continuation により，カーネル内の実行の際にプロセスモデルと割り込みモデルの両方の利点を利用できる．プロセスモデルでは，カーネル内の実行状態はスタック上に記録される．割り込みモデルでは，後で再開できるように補助的なデータ構造体に保存される．Continuation とは，本来このデータ構造体の名称である．Continuation は，プロセスモデルの使いやすさと割り込みモデルの性能上の利点があり，他のオペレーティングシステムで開発された多くの最適化技術を取り入れることができる．そうして最適化技術に，stack discarding, stack hand-off, および continuation recognition がある．Continuation により再設計された Mach 3.0 は，従来の Mach 3.0 と比較して，スレッドが消費する記憶領域が 85% 減少し，異なるアドレス空間に渡る RPC および例外処理がそれぞれ 14% と 60% 高速になった．このことからわかるように，Continuation は仮想空間を共有するモジュール間通信時の記憶領域の消費を減少させることにより，TLB およびキャッシュの有効性を増加させ，結果的に高速な通信を実現する．

3. スロットによる仮想記憶管理

こうした研究を踏まえ，本論文ではスロットによる仮想記憶管理技法を提案する．本技法では，複数のオブジェクトが仮想空間を共有する．このため，オブジェクト間通信において TLB エントリとキャッシュエントリの削除が必要なくなり，またデータ転送を参照を渡すことで実現できる．これらの利点は，スレッドや Opal と同様である．スレッドとの違いは，すべてのプログラム分野に適応できることであり，Opal との違いは多重仮想空間を用いていることである．単一仮想空間を用いなかった理由は，それが大域的な記憶領域管理を必要とするため，管理のためのオーバヘッドが大きく大規模分散環境に適さないと考えたからである．多重仮想空間を用いることにより，仮想空間を保護の単位として使用できる．また，頻繁に通信を行うオブジェクトが仮想空間を共有することにより，通信の高速化が実現できる．さらに，本技法ではオブジェクト間の関係が動的に変化することに対応できるように，仮想空間間でのオブジェクトの移動が容易にできる．

スロットによる仮想記憶管理では，Continuation を用いてプログラムを記述することもできる．これは，高速な通信が必要なオブジェクト（主にシステムオブジェクト）が明示的に要求することにより，仮想空間を共有することが保証されるからである．Apertos においては，Continuation が単層システムにおけるカーネルの機能を実現するシステムオブジェクトの実装に使われている．

本章では，スロットによる仮想記憶管理およびオブジェクトのメソッド実行を高速化するための3つの手法について詳説する．

3.1 スロット

スロットとは，仮想空間を同じ大きさに分割した時の一つの断片である．オブジェクトは，その大きさや概念に応じて仮想空間あるいはスロットが与えられる．後者の場合，与えられるスロットの個数はオブジェクトの大きさに依存する．

スロットの大きさと仮想空間あたりのスロットの個数は反比例の関係にある．スロットの大きさは，システムオブジェクトおよびアプリケーションが必要とする記憶容量より十分大きいスロットが必要数提供できるように決めなければならない．Apertos における実装では，**図2**に示すオブジェクトの大きさ☆ の分布を考慮し，スロットサイズを 256 Kbytes にした．なお，スロットには仮想空間に一意なスロット番号 n が付けられている．

3.2 オブジェクトのスロットへの初期配置

オブジェクトのスロットへの配置は，オブジェクトの生成時に行われる．オブジェクトは，クラスから生成される．クラスは，オブジェクトの静的かつ不変なテンプレートであり，オブジェクトのバイナリイメージ，およびオブジェクトの粒度や性質といったオブジェクトの属性を管理している．**表1**にオブジェクトの属性を管理する構造体（以下，クラス構造体と呼ぶ）

☆ テキストおよびデータ領域の大きさの和．Apertos オブジェクトのデータ領域や実行中に扱うオブジェクトは細粒度であるが，Apertos オブジェクトそのものはテキスト領域やスタック領域があるので全体としては細粒度と呼べない大きさになっている．

を示す.

オブジェクトのスロットへの配置は，以下の3つの項目によって決定される.

- そのオブジェクトのクラス構造体.
- オブジェクト生成時に割り当てられるオブジェクト識別子[6].
- 新しいオブジェクトを生成しようとしているオブジェクトの仮想空間あるいはスロットに関する情報.

オブジェクトは，新しいオブジェクトを生成しようとしているオブジェクトと同じ MachineID と SpaceID を持ち，適当なハッシュ関数 H より，

$n=H(OID,$ ExecutionMode)

で与えられる n を持つスロットに配置される. ExecutionMode は，空間の指定に使われる. ユーザモードのオブジェクトおよびカーネルモードのオブジェクトはそれぞれユーザ空間およびカーネル空間に割り当てられる. カーネル空間は仮想空間の高位アドレス側で全体の半分をしめ，すべての空間にマップされている.

上記で述べた初期配置を，詳しく述べると以下のようになる. オブジェクトの大きさ Size とスロットの大きさを比較し，スロットより小さい場合には，オブジェクトに H で計算されたスロット n を1つだけ割り当てる. スロットより大きい場合には，n から連続して必要個数のスロットを割り当てる. n が既に使用されている場合には，起点を $n+1$, $n+2$ と順番にずらして割り当てる. ただし，このアルゴリズムだけでは非常に大きいオブジェクトに対するスロットの割り当てに時間がかかるので，そうしたオブジェクトには単独の仮想空間が与えられる. また，ExecutionMode から，システムオブジェクトには無条件にスロットを割り当てるが，ユーザオブジェクトをスロットに配置するかどうかは，LanguageInofo の情報によって決定する. 例えば，C言語で作られたオブジェクトは他のオブジェクトの領域を侵す恐れがあるので，スロットに配置しないが，Modula-3 で作られたオブジェクトはスロットに配置する.

スロットの割り当てが難しいのは，次のような場合である.

(1) 別々のオブジェクトに対して同じスロット番号 n が割り当てられた時.

(2) オブジェクトがその実行中に関数 *malloc* 等を実行して，オブジェクトの大きさが変化し，ついにスロットに入らなくなった時.

(1)は，オブジェクト生成時にわかる場合と，オブジェクト移動時にわかる場合がある. オブジェクト生成時の場合は，ハッシュ関数で得られたスロットが n である時，$n+1$, $n+2$ の順番で空いているスロットを探して割り当てる. 移動時に関しては，後述する. (2)で次のスロットが既に使われている場合には，新たな仮想空間が作成されオブジェクトのスロットはすべて新しくできた仮想空間にマップされる.

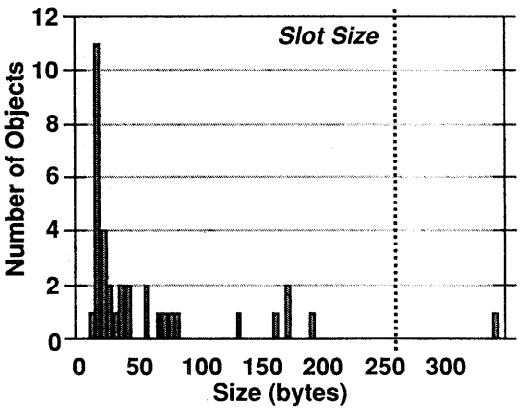


図 2 Apertos オブジェクトの大きさ
Fig. 2 Sizes of Apertos objects.

表 1 クラス構造体
Table 1 Class structure.

変数	解説
SymbolName	クラスのシンボル名.
SuperClassName	スーパクラスのシンボル名.
MethodTable	実行できるメソッドのリスト.
ObjectFormat	実行可能ファイルのフォーマット. a.out 形式および COFF 形式の指定に使用する. オブジェクトの大きさ Size は対応する形式のヘッダファイルから計算される.
Attribute	スケジューリングの際の優先度 (SchedPriority), 実行時の優先度 (Priority), 実行時のプロセッサモード (ExecutionMode), および実時間制約等オブジェクトを実行する際の制約 (Constraints).

このように，新しく生成されるオブジェクトは生成したオブジェクトと同一の仮想空間に配置する．これは，両者間の通信頻度が大きいと予想できるからである．

3.3 オブジェクトの移動による仮想空間切替えの省略

また当初仮想空間を共有していなくても，オブジェクトがその実行中に仮想空間間を移動することにより，異なる仮想空間間での通信頻度を削減することができる．

図3において初期状態(a)では，仮想空間Aに配置された *Object 3* が仮想空間 B に配置された *Object 1* と頻繁に通信している．*Object 3* が仮想空間 B に移動することにより（状態(b)），*Object 1* と *Object 3* が仮想空間を共有でき，両者の間の通信が高速化される．この時，*Object 3* のスロット番号 n は仮想空間 B でも変化しない．こうすることにより，オブジェクト移動時に再配置が不必要となるので，オブジェクトを移動するコストが小さくなる．

なお図3の例では，*Object 1* を仮想空間 A に移動することによっても同様の効果が得られる．どのオブジェクトをいつ移動させるかという戦略については現在検討中である．また，分散環境におけるオブジェクトの移動機構についても現在検討中である．

3.4 手続き呼び出しによる書き換え

オブジェクトの初期配置あるいは仮想空間間の移動の結果，オブジェクトが仮想空間を共有する状態で，オブジェクトのメソッド実行を手続き呼び出しに書き換えることにより，通信の高速化と通信経路の最適化を行うことができる．

オブジェクトのメソッド実行は，オブジェクトにリンクされたライブラリによって行われる．すなわち，ライブラリには通常のトラップ命令によるメソッドの実行と手続き呼び出しによる実行の両方が含まれており，前回のメソッド実行の際に通信管理システムが返す情報によって次回のメソッド実行の方法を決定する．

この手法により，オペレーティングシステムが提供するメソッド実行速度を言語処理系の提供するそれに近づけることができる．その際，通信経路がどのように最適化されるかについては，6.1.2 節で具体例を用いて説明する．

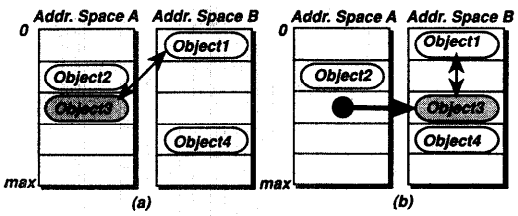


図3 オブジェクトの移動による仮想空間切替えの省略．オブジェクトが仮想空間間を移動しても，スロット番号は変化しない．

Fig. 3 Omitting context-switches by moving an object from one virtual address space to another. When an object moves between virtual address spaces, its slot number n does not change.

4. Apertos オペレーティングシステム実装の概要

本章では，Apertos の実装のうち本論文に関係する部分について概観する．第1章で述べたように，Apertos はオブジェクトとメタオブジェクトの分離およびリフレクションを基本として構成される．また，メタオブジェクトのグループはメタオブジェクト空間として管理される．そこで，ここでは以下の2つの点について説明する．

- リフレクションを，オブジェクトの実行に共通なプリミティブとして提供する *MetaCore*.
- メタ階層中に表現されるメタオブジェクト空間を定義するための機構であるリフレクタ．

4.1 MetaCore

MetaCore は，メタオブジェクト空間を持たない終端オブジェクトであり，マイクロカーネル技術によって構成されたオペレーティングシステムのマイクロカーネルに相当する．**表2**に *MetaCore* が提供するプリミティブを示す．オブジェクトがこれらのプリミティブを実行する場合は，通常のオペレーティングシステムにおいてシステムコールを実行する場合と同

表2 *MetaCore* のインタフェース
Table 2 The MetaCore interface.

操作	説明
M	メタ計算の呼び出し
R	任意のコンテキストの実行
CNew	コンテキストの作成
CDelete	コンテキストの削除
CBind	割り込みハンドラの登録
CUnbind	割り込みハンドラの削除
CSetAttribute	コンテキストの値の設定
CGetAttribute	コンテキストの値の読み込み

様，トラップ命令が発行される．

4.2 リフレクタ

リフレクタは，オブジェクトにメタ計算を提供するとともに，メタオブジェクトのグループを定義する．提供されるメタ計算は，リフレクタが定義したグループに含まれるメタオブジェクトによって提供される機能を用いて実現される．また，リフレクタはプログラミング言語におけるクラス階層中に定義されており，我々はそれをリフレクタクラス階層と呼んでいる．これはリフレクタを記述する際の便を図るもので，既存のリフレクタの再利用が可能になる．階層の一番上は，m^{Common}* である．m^{Common} は，すべてのリフレクタに共通の機能を提供する抽象クラスである．

図4に，m^{Common} のサブクラスとして定義されるリフレクタのいくつかを示す．例えば，m^{Drive} は，デバイスドライバにおける割り込み処理のためのメタ計算を提供するリフレクタである．そこでは，割り込み処理のための時間が最小限になるように実装されている．我々は，新しいメタ計算を持つリフレクタを，既存のリフレクタのサブクラスとして定義することができる．

現在の実装では，オブジェクトのためのリフレクタとして，m^{Base}, m^{Class}, および m^{Storage} 等がある．これらのリフレクタは m^{Zero} と呼ばれるリフレクタのためのリフレクタにより，メタオブジェクト空間を構成する．以下に代表的なリフレクタである m^{Base} と m^{Zero} について述べる．

4.2.1 m^{Base}

m^{Base} は，並行オブジェクト指向プログラミングをサポートするためのリフレクタである．m^{Base} の機能を**表3**および**表4**に示す．表3は m^{Base} が管理するメタオブジェクト空間に属するオブジェクトのためのインタフェースであり，表4は m^{Base} が属するメタオブジェクト空間を管理するリフレクタである m^{Zero} に対するインタフェースである．m^{Base} は，RPC のセマンティクスによる同期通信および非同期通信の機能を提供している．

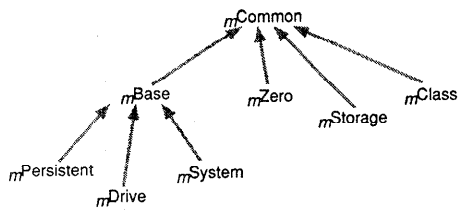


図4 リフレクタクラス階層の一部
Fig. 4 A portion of the reflector class hierarchy.

* 本論文では，リフレクタを表すのに $m^{\text{Reflector}}$ の記法を用いる．

4.2.2 m^{Zero}

m^{Zero} は，リフレクタのプログラミングのための機構を提供しており，**表5**に示す機能を提供する．これらの機能は，リフレクタからのみ使われる．m^{Zero} は，Conitinuation を用いた同期通信および非同期通信の機能を持つ．そのため，リフレクタは同期通信等のブロック操作を実行した後に，別の要求を受け付けることが可能になっている．これにより，リモート遅延を削減することができる．

m^{Zero} も含めリフレクタは仮想空間を共有することを前提にプログラミングされており，リフレクタ間のメソッド実行は Conitinuation による高速化が図られている．

5. スロットによる仮想記憶管理技法の実装

第3章で提案した機構を Apertos に実装したのが**図5**である．グラデーションのかかった大きな楕円はメタオブジェクト空間を表し，影つきの楕円および影つきの長円はそれぞれオブジェクトおよびリフレクタを表している．なお，簡略化のために *MetaCore* は省略した．実装したシステムは，仮想記憶管理，クラス

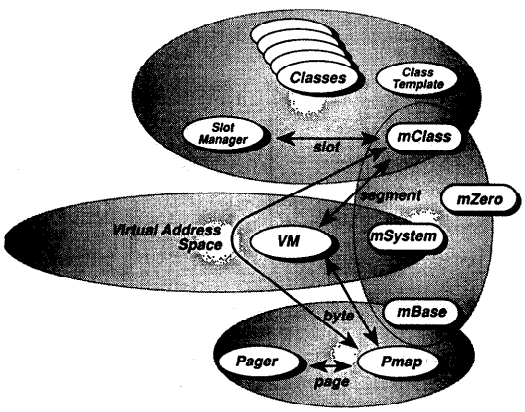


図5 Apertos におけるオブジェクト管理機構の実装
Fig. 5 An implementation of object management mechanism in Apertos.

システム，および通信管理の3つからなり，本章ではそれらについて述べる．

5.1 仮想記憶管理

仮想記憶管理は，図5において，左中間部と下部の2つのメタオブジェクト空間から構成される．左中間部は計算機アーキテクチャに依存しない管理を行い，*VM* オブジェクトを含む．下部は計算機アーキテクチャに依存した管理を行い，*Pager* オブジェクトと *Pmap* オブジェクトを含む．*VM* は，データ構造体としてセグメントを扱い，セグメントの割当／開放および属性の変更を行う．また，*Pager* と *Pmap* は，データ構造体としてページを扱う．*Pager* は定期的に起動されるオブジェクトで，不要なページの掃き出し（ページアウト）を行う．*Pmap* は以下の機能を持

表 3 m^{Base} のオブジェクトに対するインタフェース
Table 3 The m^{Base} intarface to objects.

メソッド	説明
Call	対象オブジェクトの指定メソッドを起動する．実際の起動は内部スケジューラが行う．また，呼びだし側はメソッドの終了までブロックされる．
Send	Call とほぼ同じであるが，呼びだし側はブロックされない．
Reply	メソッドの実行結果を起動したオブジェクトに返す．
New	指定されたクラスの新たなオブジェクトを生成する．
Delete	指定されたオブジェクトを消去する．
Grow	メモリを確保してオブジェクトの大きさを大きくする．
Shrink	Grow とは逆にオブジェクトの大きさを小さくする．
Yield	自身の実行権を放棄して，内部スケジューラを呼びだし，次のオブジェクトを起動させる．
Find	オブジェクトとリフレクタの対応を得る．
Install	オブジェクトとリフレクタの対応を与える．
Migrate	オブジェクトを別のリフレクタへ移動する．

表 4 m^{Base} の m^{Zero} に対するインタフェース
Table 4 The m^{Base} interface to m^{Zero}.

メソッド	説明
Deliver	特定のオブジェクトの指定するメソッドを起動する．これは異なるリフレクタ上のオブジェクト間の通信に利用される．
Migrate	オブジェクトが移動してくる処理を扱う．
Preempt	リフレクタ間の実行権利の委譲に用いられる．

表 5 m^{Zero} のインタフェース
Table 5 The m^{Zero} interface.

メソッド	説明
Call	ターゲットオブジェクト（リフレクタ）の指定メソッドを起動する．この起動は実際には m^{Zero} のスケジューラが行う．呼びだし側は処理の終了を待つ．
Force	Call とほぼ同じであるが，内部スケジューラを呼ぶことなしに直接対象となるリフレクタが起動される．
Send	Call とほぼ同じであるが，処理の終了を待たない．
Reply	起動された処理（メソッド）の終了であり，呼びだし側を実行可能にする．必要であればメソッドの結果を返す．
Return	Reply とほぼ同じであるが，Reply が内部スケジューラを呼ぶのに対し呼びだし側のリフレクタを直接起動して実行結果を返す．
Preempt	リフレクタ間の実行権の横取りを行う．現在は割り込みハンドラ用リフレクタ m^{Drive} からのみ呼ばれている．
Exit	リフレクタが，その実行を終了する．
Find	オブジェクトとリフレクタの対応を返す．
Install	オブジェクトとリフレクタの間の関係を与える．
Migrate	リフレクタを他の m^{Zero} に移動させる．

表 6 オブジェクトフォーマットの比較
Table 6 Comparison of object formats.

項目	絶対コード		位置独立コード	
	再配置情報無	再配置情報有	レジスタ方式	マッピング方式
実行速度	通常と同じ		遅い場合有り	
生成時間	通常と同じ	遅い	通常と同じ	
生成時の再配置	不可能	可能		
実行中の再配置	不可能		可能	
多重仮想空間でのテキスト共有	可能			
仮想空間内でのテキスト共有	不可能			可能

つ.

- 仮想空間の生成／消去／切替え
- 物理ページの割当／解放および属性の変更
- ページフォルト処理

なお，*Pmap* の各機能の呼び出しにおいては，サイズの指定がバイト数で行われる．そのため，*VM* は完全に計算機アーキテクチャに依存しない実装になっている．

5.2 クラスシステム

クラスシステムは，図5において上部に位置するメタオブジェクト空間であり，*Class* とそのテンプレートである *ClassTemplate*，およびクラスシステムのためのリフレクタである m^{Class} からなる．*Class* は，オブジェクトの静的かつ不変なテンプレートであり，オブジェクトはいずれかのクラスに属する．*Class* は，クラス構造体およびオブジェクトの実行コードを持っている．*ClassTemplate* は，a.out 形式や COFF (Common Object File Format) 形式等オブジェクトのフォーマットに関する情報が格納されている．m^{Class} は，(1) オブジェクトの生成／消去，(2) *Class* や *ClassTemplate* の性質の定義，および(3) 言語処理系に関する情報の管理，を行う．例えば，プログラミング言語がクラス階層を提供する場合，m^{Class} はコンパイラにクラス階層に関する情報を提供する．クラスシステムの役割をまとめると以下のようになる．

- クラスシステムは，オペレーティングシステムが提供するプログラミングのための環境である．
- クラスシステムは，言語処理系やハードウェアの異種性を隠蔽し，オブジェクトの生成や仮想記憶管理等のオブジェクト管理に有用な情報を提供する．

現在のクラスシステムは，C++ 言語のためのものであるが，将来的にはクラスシステムの概念を複数言語を支援するように拡張する予定である．この拡張は，m^{Class} や *ClassTemplate* を言語ごとに用意することにより，すなわち言語ごとのメタオブジェクト空間を用意することにより実現する．

オブジェクト生成時にオブジェクトのスロットへの配置を行うのもクラスシステムである．スロットへの配置は，生成時に割り当てられるオブジェクト識別子を基に行われる．そのため，*Class* の持つオブジェクトの実行コードは再配置可能でなければならない．一方，同じ *Class* のインスタンスであるオブジェクトが仮想空間を共有する場合に，テキスト領域の共有ができた方が望ましい．以下，オブジェクトのフォーマットを再配置とテキスト領域の共有の点から議論する．

各種のオブジェクトフォーマットに関して，再配置およびテキスト領域の共有についてまとめたのが**表 6** である．コンパイル時に絶対番地が書き込まれているフォーマットでは，通常オブジェクト生成時に再配置ができないので，本論文の技法を活用することができない．しかし，そうしたフォーマットでも再配置情報が付加されている場合☆，オブジェクト生成時に再配置を行うことができる．さらに，位置独立コード (Position Independent Code) では，実行中の再配置も可能である．計算機ハードウェアが提供する位置独立コードによる再配置機構には，レジスタ方式とマッピング方式がある．レジスタ方式は，ベースリロケーションレジスタ (BRR) を持ち，このレジスタにプログラムの先頭番地をセットし，実行中主記憶をアクセ

☆ UNIX では，ld コマンドを r オプションを付加して実行することで，再配置情報付きの実行コードを出力することができる．

スする時に BRR の内容をプログラムが指定する番地に加える方式である．マッピング方式は，仮想記憶のページングと組み合わせて用いられるのが一般的で，主記憶およびプログラムをページに分割し，ページの主記憶上での先頭番地を表として保持することにより，主記憶をアクセスするごとにこの表を調べてプログラムの指定する番地に加える方法である．

今回の実装では，CPU に MC 68030 を使用したため，再配置情報有りの絶対コードかレジスタ方式の位置独立コードからの選択となった．前者は，オブジェクト生成時に，再配置情報を用いてプログラムコードの書き換えをしなければならないため，オブジェクトの生成に時間がかかる．後者は，プログラムの実行速度が遅くなるという欠点を持っている．どちらが望ましいかは，そのオブジェクトの生存期間に依存する．生成されてすぐに消去されるオブジェクトでは，位置独立コードが有利であるし，システムオブジェクトのようにシステムの動作中ずっと存在するオブジェクトでは再配置情報有りの絶対コードが望ましい．両方を実装し性能評価をした上で，そのトレードオフを議論すべきであるが，本研究ではシステムオブジェクトの高速実行を優先させるため再配置情報有りの絶対コードを用いることにした．なお，オブジェクト生成時の再配置のコストについては，6.3 節で評価を行う．

5.3 通信管理

通信管理は図5の右中部すなわちリフレクタによって行われる．オブジェクト間通信頻度の監視を行うためのプログラムの実装は，m^{Common} に定義することによって実現される．

6. 性能評価

本章では，(1)オブジェクトのメソッド実行時間と，(2)仮想空間間のオブジェクト移動のコスト，(3)オブジェクト生成時間に占める再配置に要する時間の割合，を評価する．測定は，ソニー製 PWS-1550 ワークステーションにタイマボードを装着した計算機で行った．その諸元をまとめたのが**表7**である．

6.1 オブジェクトのメソッド実行時間

オブジェクトの *Null* メソッドの実行時間を測定することにより，メソッド実行に伴うオブジェクト間通信のオーバヘッドを評価する．測定したオブジェクト（*Null* メソッドを持つオブジェクト）のプログラムを**図6**に示す．図6は，3つのプログラム（ファイル）からなる．TargetObject.h では TargetObject の定義および本体が記述されており，_Apertos_stub.cc および _TargetObject_stub.cc はそれぞれ C++ 言語およびアセンブリ言語のスタブである．測定はリフレクタの *Call* と呼ばれるメタ計算の実行時間を測定することにより行うため，測定時間にオブジェクトの *Null* メソッドを実行する時間が含まれる．*Null* メソッドの実行は，図6においてアセンブリ言語の 21 行目の ENTRY (_Null) から行われる．その後，アセンブリ言語のスタブは C++ 言語のスタブ（14 行目）を実行する．実行時間は，全体で数マイクロ秒であり通信時間と比較して無視できる程度である．

また，メソッドを呼び出すオブジェクトと呼び出されるオブジェクトとの関係において，(1)両者が同じメタオブジェクト空間に存する場合と，(2)異なるメ

表 7 測定に使用した計算機の諸元
Table 7 Characteristics of the platform used for measurement.

CPU	MC 68030 (25 MHz)
主記憶	8 Mbytes
I/O プロセッサ	なし
タイマボードの精度	1 μsec
タイマボードの値の読み出し時間	2〜4 μsec (平均 2.6 μsec)

```
 1: /* Objects/TargetObject/h/TargetObject.h */
 2: Class   TargetObject {
 3: public:
 4:                  TargetObject () {}
 5:         void     Null () {}
 6: };
 7:
 8: /* Objects/TargetObject/_Apertos_stub.cc */
 9: #include         <TargetObject.h>
10:
11: TargetObject     Self;
12:
13: extern "C" void
14: Null (TargetObjectMsg* pMsg)
15: {
16:         pMsg -> status = Self.Null ();
17:         (void) Epilogue ();
18: }
19:
20: /* _TargetObject_stub.s */
21: ENTRY(_Null)
22:         pea     (a0)
23:         jbsr    _Null
24:         tst.l   (sp)+
25:         rts
```

図 6 メソッドの実行時間の測定に用いたプログラムの一部
Fig. 6 Code fragments of a measured object.

タオブジェクト空間に存在する場合，について測定する．なお，測定するオブジェクト（*ObjectA* および *ObjectB* の名称で参照する）はユーザモードで実行され，リフレクタはすべてカーネルモードで実行される．前述したように，カーネル空間はすべてのオブジェクトにマップされているものとする．

6.1.1 同じメタオブジェクト空間における測定

表8に，メタオブジェクト空間が同じオブジェクト間での *Null* メソッドの実行，すなわち，**図7**で示される通信経路における通信時間を示す．図7で，*Reflector 1* は m^{Base} を，*Reflector 2* は m^{Zero} を用いて測定した．また，オブジェクトから他のオブジェクトを呼び出す場合，*MetaCore* の *M* 計算が行われ，呼び出されたオブジェクトが呼び出したオブジェクトに結果を返す場合は，*MetaCore* の *R* 計算が行われる．詳しいプログラムの実行系列は，付録に**図10**として示す．

これより，*ObjectA* および *ObjectB* が異なる仮想空間を持つ従来の記憶管理と比較して，両者が仮想空間を共有した場合メソッドの実行が4倍高速になり，さらに手続き呼び出しによる書き換えを行った場合には約200倍の高速化が達成できる．比較のために言語処理系におけるメソッド実行も表8に示した．手続き呼び出しによる書き換え後の通信速度は，言語処理系の10倍以下であり高速であることがわかる．また，TLB およびキャッシュエントリの効果を考察するために，仮想空間を共有した状態でそれぞれのエントリの削除を行った結果も表8に示した．今回の測定では，以下の2つの理由により TLB およびキャッシュエントリを削除する影響が少ないと考えられる．

- 2つのオブジェクト間における測定であること．
- 実行されるメソッドの本体が非常に小さいこと．

それにもかかわらず，表8は TLB およびキャッシュエントリを削除する影響が無視できないことを示しており，通常の状態では両者の影響は非常に大きいと予想できる．

キャッシュエントリの削除についてさらに考察するため，上記のメソッド実行を通じてのキャッシュの効果を測定したのが**表9**である．データキャッシュより命令キャッシュの影響が大きいことがわかる．また表8および表9の結果から，通信時にキャッシュエントリの削除を行っても，キャッシュを全く使わなかった場合よりは実行が速くなっていることがわかる．

なお，Call 等リフレクタのメタ計算および *MetaCore* のプリミティブのみの性能については文献13)を参照されたい．

6.1.2 メタオブジェクト空間間における測定

次に**表10**に，異なるメタオブジェクト空間に属するオブジェクトの *Null* メソッド実行，すなわち，図

表8 メタオブジェクト空間が同じオブジェクト間の通信時間の高速化
Table 8 Communication time between two objects which share the same meta-object space.

手法	備考	時間 (μs)	比率
従来の管理	仮想空間間	2,847.3	1
スロット仮想空間	TLB およびキャッシュ使用	712.2	4
スロット仮想空間	キャッシュエントリ無効化	745.6	
スロット仮想空間	TLB エントリ無効化	760.6	
スロット仮想空間	TLB およびキャッシュエントリ無効化	770.5	
手続き呼び出し		15.5	184
手続き呼び出し	言語処理系（参考値）	1.9	

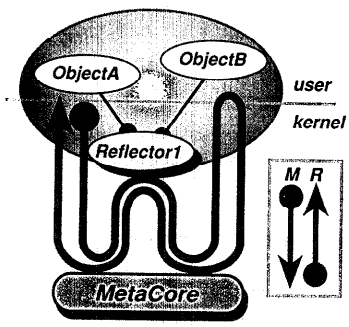


図7 メタオブジェクト空間が同じオブジェクトのメソッドを実行する場合の通信経路
Fig. 7 Communication path when invoking a mothod of an object within the same meta-object space.

表9 メタオブジェクト空間が同じオブジェクト間の通信時間に対するキャッシュの影響
Table 9 Cache effects on communication time between two objects which share the same meta-object space.

条件	時間 (μs)
仮想空間内	712.2
仮想空間内＋データキャッシュ未使用	739.4
仮想空間内＋命令キャッシュ未使用	827.6
仮想空間内＋命令およびデータキャッシュ未使用	845.4

8で示される通信経路における通信時間を示す．*Reflector 1* および *Reflector 3* は，m^{Base} と同じ機能を持ち，*Reflector 2* には m^{Zero} を用いている．また付録に，詳しいプログラムの実行系列を**図 11** として示す．

これより，*ObjectA* および *ObjectB* が異なる仮想空間を持つ従来の記憶管理と比較して，両者が仮想空間を共有した場合メソッドの実行が 1.7 倍高速になり，さらに手続き呼び出しによる書き換えを行った場合には約 400 倍の高速化が達成できた．メタオブジェクト空間が同じ場合と比較すると，仮想空間を共有しただけでは効果が小さいように見えるが，これはもともと**図 8** で *Reflector 1*, *Reflector 2*, および *Reflector 3* がすべて仮想空間（カーネル空間）を共有しているという設定であり，それらの間で Continuation による高速化がすでに行われているからである．一方，手続き呼び出しによる書き換えの効果が大きいのは，図 8 の通信経路が**図 9** のようになるからである．図 9 で，*ObjectA* は *Reflector 2* に手続き呼び出しをすることにより *ObjectB* の実行を宣言した後，*ObjectB* の *Null* メソッドを実行する．*Null* メソッドの実行後，*ObjectB* から *ObjectA* への実行の遷移は，上記の逆の経路になる．*Reflector 2* を一度ずつ呼び出すのは，*Reflector 2* が管理しているスケジューリング情報を書き換えるためである．これからわかるように，手続き呼び出しへの書き換え後のメソッドの実行時間は，オブジェクトが属するメタオブジェクト空間に関わらず一定であることがわかる．

表 10 異なるメタオブジェクト空間に属するオブジェクト間の通信時間の高速化
Table 10 Communication time between two objects which belong to different meta-object spaces.

手　法	時間 (μs)	比　率
仮想空間間	5,832.2	1
スロット仮想空間	3,755.4	1.6
手続き呼び出し	15.5	376

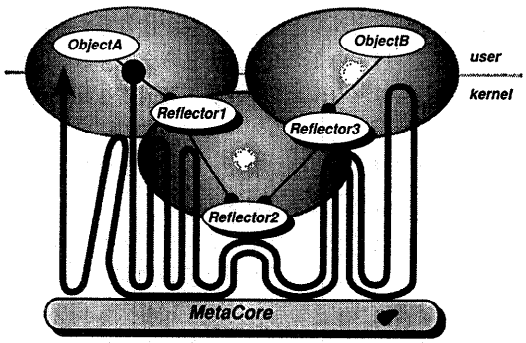


図 8 異なるメタオブジェクト空間に属するオブジェクトのメソッドを実行する場合の通信経路
Fig. 8 Communication path when invoking a method of an object which belongs to a different meta-object space.

6.2 仮想空間間のオブジェクト移動のコスト

ここでは，同一計算機内で仮想空間間を *Null* オブジェクト* が移動するコストを測定した．移動は，m^{Class} がオブジェクトの各セグメントに対して，*VM* の *Remap* メソッドを用いて仮想記憶領域をマッピングし直すことにより実現する．*VM* はさらに *Pmap* の *Remap* メソッドを実行する．測定は，図 5 で *VM* と *Pmap* が仮想空間を共有しているが手続き呼び出しによる書き換えをしていない状態で行った．

移動に要する時間は，*Null* オブジェクトで 13,276.8 マイクロ秒であった．この時間は，オブジェクトの大きさに比例しており，オブジェクトの大きさが 10000 バイト増えるごとに約 800 マイクロ秒増加する．

仮想空間間でのオブジェクト移動のコストと，通信コストの削減のトレードオフについて考察する．図 7 で，*ObjectB* が *Null* の場合，*Null* が仮想空間間を移動することにより良い結果が得られるのは，表 8 の結果から以下のように計算できる．

移動コスト/移動前後の通信コストの差
$=13276.8/(2847.3-712.2)$

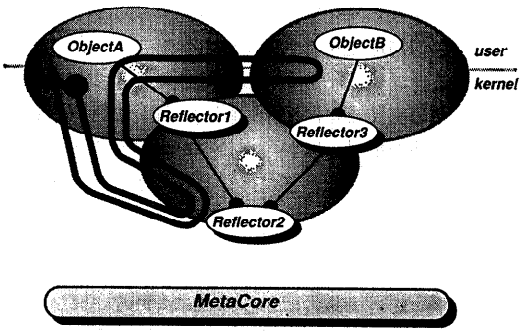


図 9 異なるメタオブジェクト空間に層するオブジェクトのメソッドを実行する場合の通信経路を手続き呼び出しを用いて最適化した時の経路
Fig. 9 The optimized communication path of the previous example.

* a.out 形式にリンクされており，テキスト領域，データ領域，および BSS 領域はそれぞれ 9820 bytes, 1008 bytes, および 4188 bytes である．

$=6.2$

すなわち，移動後に7回以上のメソッド実行が行われなければ，移動のコストの方が小さくなることがわかる．同様の計算を，異なるオブジェクト空間に属するオブジェクトの場合について表 10 から計算すると，やはり 6.4 になる．なおトレードオフに関しては，動的な状況をシミュレーションで検証すべきである．今回の結果をパラメータとして用いてのシミュレーションによる解析に関しては，今後の課題としたい．

6.3 オブジェクト生成における再配置のオーバヘッド

Apertos における今回の実装では，オブジェクトのフォーマットとして再配置情報有りの絶対コードを使用した．このフォーマットは，コードの実行が速い反面オブジェクト生成時に再配置が必要になる．そこで，オブジェクト生成時の再配置のコストを測定した．**表 11** に，*Null* オブジェクトの生成時間の内訳を示す．これより，再配置はオブジェクトの生成時間の約 30% であり，それほど大きくないと考えられる．なお，この割合はオブジェクトの大きさに依存せずほぼ一定であることがわかっているが，ここでは紙面の都合で割愛する．

7. 今後の課題

本研究で未解決の問題は多い．特に重要な問題として，(1)オブジェクトの保護，および(2)オブジェクトの移動に関する戦略がある．

7.1 オブジェクトの保護

今回の実装に用いた MC 68030 等のチップは，仮想空間内の保護機能を提供していないので，仮想空間を共有するオブジェクトがお互いの領域を侵す恐れがある．現在の実装では，Modula-3 のように言語処理系が保護機構を提供している場合は，スロットによる仮想記憶管理を用いる．また，C言語のように言語処理系が保護機構を持っていない場合には，仮想空間を保護の単位として使う．ただし，単層システムにおいてカーネルが提供していた機能を実現するオブジェクト（カーネルモードで動作するオブジェクト）は，システム全体の高速化のため，言語処理系にかかわらずスロット仮想記憶管理を用いている．

将来的には，仮想記憶領域内で保護機構を持つ MMU (Memory Management Unit) を利用することにより，言語処理系ではなくハードウェアでスロットの保護を行いたいと考えている．チップとしては，Advanced RISC Machines (ARM) 社が開発した ARM 610 が考えられる．ARM 610 の MMU では，仮想記憶制御とアクセス権付きメモリアクセスを提供している．仮想記憶制御では，間接ポインタのリストをたどるための表検索を高速化している．また，メモリアクセスでは，メモリをいくつかの領域に分割し，その領域にオブジェクトを作ることができる．

表 11 スロットに配置されるオブジェクトの生成時間
Table 11 Creation time of a new object which is assigned to one or more slots.

段　　階	時間 (μsec)	比率(%)
仮想記憶領域の確保	6096	24.0
テキスト領域のコピーとデータ領域のコピーと初期化	1259	5.0
再　配　置	7593	29.8
そ　の　他	10493	41.2
合　　計	25443	100

7.2 移動および手続き呼び出しへの書き換えに関する戦略

本論文では，オブジェクト間通信を高速化する機構について述べてきた．しかし，オブジェクトの移動に関する戦略（いつ，どのオブジェクトをどこに移動するか）については議論しなかった．これは現在の実装では同一計算機内の移動しか実現していないがため，その重要性があまりないからである．つまり，オブジェクト生成時に行われるオブジェクトのスロットへの初期配置で十分だからである．オブジェクトの移動に関する戦略については，この技法を分散環境に適応した時に議論するつもりである．

8. おわりに

本論文では，スロットによる仮想記憶管理を提案した．これは，以下の3つの手法と組み合わせることにより，同一計算機内のオブジェクト間通信を高速化する．

(1) オブジェクト生成時に，複数のオブジェクトが仮想空間を共有させる．
(2) 仮想空間間でオブジェクトを移動することにより，異なる仮想空間間での通信頻度を削減する．
(3) オブジェクト間通信を手続き呼び出しに書き換えることにより，通信速度の高速化と通信経路の最適化を行う．

オペレーティングシステム Apertos に，この技法を

実装した．また，*Null* メソッドの実行時間を測定した．その結果，従来の各オブジェクトに独立の仮想空間を与える場合と比較して，(1)および(2)によりオブジェクトが仮想空間を共有した場合で1.7倍から4倍の高速化が，さらに(3)と組み合わせることにより約200倍から約400倍の高速化が達成できた．

オブジェクトのフォーマットとして再配置情報有りの絶対コードを用いた．このフォーマットは，オブジェクト生成時に再配置が必要であるが，再配置のオーバヘッドはオブジェクトの生成時間の約30%であった．

謝辞 実装に際して貴重な助言を数多く頂いた(株)ソニーコンピュータサイエンス研究所の天満隆夫博士に感謝いたします．また，タイマボードを作成してくださった手塚宏史氏をはじめとするソニー(株)スーパーマイクロ事業部本部の方々に感謝いたします．

参考文献

1) Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M.: Lightweight Remote Procedure Call, *Proceedings of the 12th ACM Symposium on Operating System Principles*, pp. 102-113 (Dec. 1989).
2) Bershard, B. N.: The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating System, *USENIX Workshop on Micro-Kernel and Other Kernel Architectures*, USENIX Association, pp. 205-211 (Apr. 1992).
3) Black, A. P., Lazowska, E. D., Noe, J. D. and Sanislo, J.: The Eden System: A Final Review, Technical Report No. 86-11-01, Department of Computer Science, University of Washington (Nov. 1986).
4) Chao, C., Machey, M. and Sears, B.: Mach on a Virtually Addressed Cache Architecture, *USENIX Mach Workshop*, USENIX Association, pp. 31-51 (Oct. 1990).
5) Draves, R. P., Bershad, B. N., Rashid, R. F. and Dean, R. W.: Using Continuations to Implement Thread Management and Communication in Operating Systems, *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 122-136 (Oct. 1991).
6) Fujinami, N. and Yokote, Y.: Naming and Addressing of Objects without Unique Identifiers, *Proceedings of the 12th International Conference on Distributed Computing Systems*, pp. 581-588 (June 1992).
7) Leffler, S. J., McKusick, M. K., Karels, M. J. and Quarterman, J. S.: The *Design and Implementation of the 4.3 BSD UNIX*, Addison-Wesley Publishing Company Inc. (1989).
8) Sun Microsystems: *System Services Overview*, Part Number: 800-1753-10 (1988).
9) Mogul, J. C. and Borg, A.: The Effect of Context Switches on Cache Performance, *ASPLOS-IV*, pp. 75-85 (Apr. 1991).
10) Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P. and Neuhauser, W.: Chorus Distributed Operating Systems, *Computing Systems*, Vol. 1, No. 4, pp. 305-370 (1988).
11) Shapiro, M. and Russo, V.(eds.): *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, IEEE Computer Science Press (1991).
12) Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, J. and van Rossum, G.: Experiences with the Amoeba Distributed Operating System, *Communications of the ACM*, Vol. 33, No. 12, pp. 46-63 (1990).
13) Yokote, Y.: The Apertos Reflective Operating System: The Concept and Its Implementation, *Proceedings of Object-Oriented Programming Systems, Languages and Applications in 1992*, ACM Press, pp. 414-434 (Oct. 1992).
14) Yokote, Y., Teraoka, F., Mitsuzawa, A., Fujinami, N. and Tokoro, M.: The Muse Object Architecture: A New Operating System Structuring Concept, *ACM Operating Systems Review*, Vol. 25, No. 2, pp. 22-46 (1991).
15) Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D. and Baron, R.: The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proceedings of the 11th ACM Symposium on Operating System Principles*, pp. 63-76 (Nov. 1987).

付録 メソッド実行におけるプログラムの実行系列

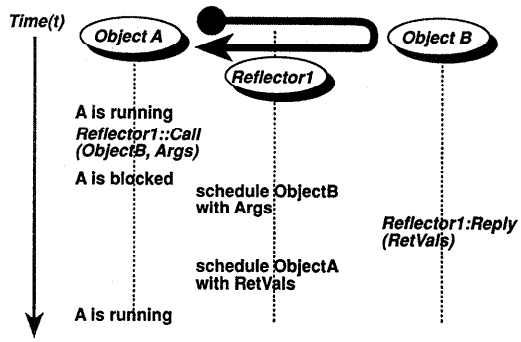


図 10 メタオブジェクト空間が同じオブジェクトのメソッドを実行する場合の実行遷移
Fig. 10 Program execution path when invoking a method of an object within the same meta-object space.

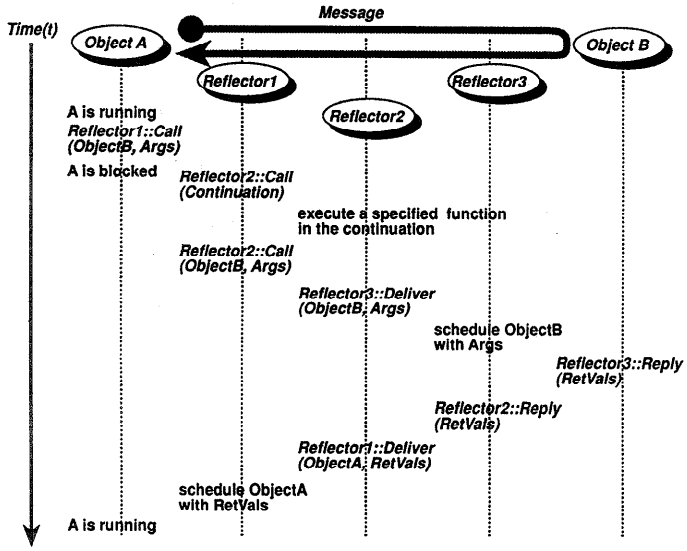
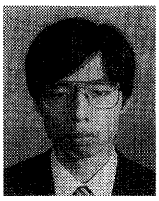


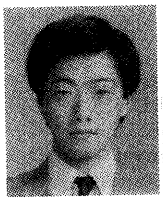
図 11 異なるメタオブジェクト空間に属するオブジェクトのメソッドを実行する場合の実行遷移
Fig. 11 Program execution path when invoking a mothod of an object which belongs to a different meta-object space.

(平成4年9月29日受付)
(平成5年3月11日採録)



光澤　敦（正会員）

1965 年生．1987 年慶應義塾大学理工学部物理学科卒業．1989年同大学院理工学研究科電気工学専攻修士課程修了．現在同大学院理工学研究科計算機科学専攻後期博士課程在学中．超大規模分散オペレーティングシステム Apertos の研究・開発に従事．オペレーティングシステム，コンピュータネットワーク，物理問題のシミュレーション等に興味を持つ．日本ソフトウェア科学会，USENIX 各会員．



横手　靖彦

1960 年生．1979 年慶應義塾大学工学部電気工学科卒業．1988年同大学院理工学研究科博士課程修了．工学博士．同年(株)ソニーコンピュータサイエンス研究所勤務，現在に至る．超大規模分散オペレーティングシステムの研究・開発に従事．オブジェクト指向計算，プログラミング言語，プログラミング環境，オペレーティングシステム，分散処理等に関心を持つ．日本ソフトウェア科学会，ACM，IEEE，USENIX 各会員．

所　真理雄（正会員）

1947 年生．1970 年慶應義塾大学工学部電気工学科卒業．1972年同大学院管理工学専攻修士課程修了．1975 年同大学院電気工学専攻博士課程修了．工学博士．同大学電気工学科助手専任，講師，助教授を経て 1991 年 4 月教授．その間カナダウォータールー大学 (1979年 1 月～1980年 6 月)，米国カーネギーメロン大学 (1980年 7 月～12月) 訪問助教授．1988年 4 月よりソニーコンピュータサイエンス研究所副所長を兼務．計算モデル，プログラミング言語，分散・開放型システム，人工知能等に興味を持つ．著者に「計算システム入門」(岩波講座ソフトウェア科学 1)，「システム構成技術」(岩波講座マイクロエレクトロニクス 10，共著)，編著に "Object Oriented Concurrent Programming" (MIT Press)，"Concepts and Characteristics of Knowledgebased Systems" (North Holland) がある．電子情報通信学会，日本ソフトウェア科学会，ACM，IEEE 各会員．