Vol. 40 No. 5 情報処理学会論文誌 May 1999

適応的オブジェクトによる排他制御の実行時緩和

江口重行[†] 八杉昌宏[††]
鎌田十三郎[†††] 瀧和男[†††]

並列処理においてはデータの一貫性保証に関する記述が容易な言語が望まれるが，オブジェクトの性質が生成/利用フェーズなど実行時に変化する場合に，これを反映した最適化は行なえていなかった．本論文では，状態に応じた排他制御規則が利用可能なオブジェクト (適応的オブジェクト) により，効率の良い並列処理を実現する手法を提案する．提案する実行時メソッド置換によりメソッド名と起動するメソッドの関係を変更することで，状態に応じた緩い排他制御規則を適用可能である．また，適応的オブジェクトを用いてインスタンス変数が変化しなくなった区間を解析することで，さらなる最適化を行う手法 (free メソッド) を提案している．提案方式を共有メモリ型および分散メモリ型並列計算機に実装，評価を行い，その効果を確認した．

Dynamic Relaxation of Mutual Exclusion using Adaptive Objects

Shigeyuki EGUCHI,[†] Masahiro Yasugi,[††] Tomio KAMADA[†††]
and Kazuo TAKI [†††]

For parallel processing, it is desirable for language systems to guarantee data consistency of objects. However, few systems support object consistency models that allow the programmer to utilize dynamic changes of the state of each object. This paper proposes an object framework called *adaptive objects*. *Dynamic method replacement* changes the correspondence of a method name to the method being invoked. This technique enables to adopt the relaxed mutual exclusion rules depending on each object state. In addition, we propose an analysis framework of analyzing the intervals in which instance variables are immutable, to perform further optimization techniques (*free method*). This paper also discusses implementation techniques of adaptive objects. Performance measurements on both shared memory and distributed memory parallel architectures indicate the effectiveness of our approach.

1. はじめに

並列処理では，一連の処理やデータの読み書きを排他的に行なうなど，常にデータへの並行アクセスを意識した処理が必要である．ここで，不用意にクリティカルセクションを設定すると，アクセスが集中するオブジェクト上でボトルネックが発生し，システム全体の実行効率が低下する．ロックなどの低レベルな同期機構を提供する場合，問題の性質に応じた効率の良いプログラムを記述可能であるが，一方プログラムは繁雑でバグの可能性も高いものになる．このプログラマの負担を軽減するため，言語が自動的にデータの一貫性を保証することが望まれる．この場合，問題の性質を活かし切ることができるかどうかが問題となる．

言語による一貫性の保証は，従来から研究が行なわれている．例として，メソッドを読み込み専用/読み書き両用に分類する，オブジェクトの不変部分について，排他制御なしでアクセス，ローカル環境へのコピーを利用する，などが挙げられる．

しかし，これらの機構でも，オブジェクトの実行時に変化する性質は利用できていない．例えば，オブジェクトの状態をデータの構築フェーズと利用フェーズに分けることができる場合，本来利用フェーズではデータの読み出しに排他制御は不要だが，これに応じた排他制御規則を適用するといったことは行なえない．

このような問題を解決するため，本論文では，状態に応じた排他制御規則が利用可能なオブジェクト (適

† 神戸大学大学院 自然科学研究科
The Graduate School of Science and Technology, Kobe University
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
††† 神戸大学工学部 情報知能工学科
Dept. of Computer and Systems Engineering, Faculty of Engineering, Kobe University

応的オブジェクト）と，その記述手段として実行時メソッド置換を提案する．

2章では，我々が開発中のオブジェクト指向並列言語 OPA の排他制御モデルについて述べる．OPA は，instant メソッドという緩い排他制御の枠組を提供している．排他制御規則はメソッドごとに静的に決定され，オブジェクトの状態の更新の有無によって読み出し専用（RO）と読み書き両用（RW）メソッドに分類される．

さらに，処理系によってオブジェクトの不変部分のみ利用すると解析された RO メソッドは，排他制御不要メソッド（free メソッド）として扱われ，排他制御，分散処理に関するさらなる最適化が施される．

プログラマは，実行時メソッド置換を用いることで，状態に応じた緩い排他制御規則を用いることが出来るようになる（3章）．実行時メソッド置換は，メソッド名と実際に起動するメソッドの対応関係を変更する．その時点での処理に応じたメソッドに切替えることで排他制御を緩和できるほか，free メソッド解析のための情報として利用する．free メソッドの検出方法については，4章で述べる．処理系が RO メソッドを free メソッドに切替えるのに十分な条件を，実行時メソッド置換によるメソッド状態遷移の情報などから静的に解析するアルゴリズムを示す．5章では，instant メソッド，実行時メソッド置換の効率的実装法，ならびに分散環境での free メソッドを用いた効率的な実装法について述べる．並列，分散環境で性能評価を行い，適応的オブジェクトの有効性について6章で議論する．7章で関連研究について述べ，まとめを8章で行う．

2. OPA の提供する排他制御モデル

本章では，オブジェクト指向並列言語 OPA の排他制御モデルについて述べる．OPA は Java 言語 [1] を並列に拡張したもので，Java のスレッドの代わりに，構造化された並列構文，同期機構，提案する排他制御モデルを組み込んだものである．

排他制御の緩和にむけ，OPA では各オブジェクト内で複数のスレッドが一貫性を保ちつつ並列実行することを許すための枠組として，instant メソッドを提供している．また，変化しなくなったデータを読み出すのみのメソッド (free メソッド) をできるだけ排他制御なしで実行することで処理効率の向上をはかる．

2.1 instant メソッド

OPA では，キーワード `instant` をつけて宣言されたメソッドを，メソッドのインスタンス変数への文面上のアクセスに応じて，RO/RW にコンパイラが分類する．RO メソッドはオブジェクト内で複数スレッドにより同時実行可能であり，また起動時にスレッドがブロックされることはない．

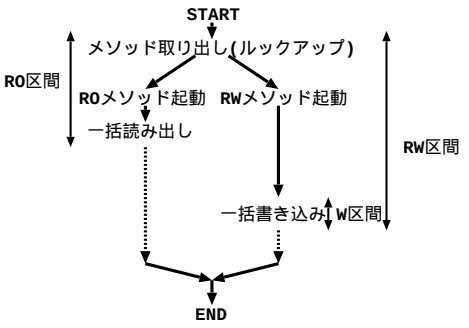

図 1 instant メソッド
Fig. 1 The execution model of instant methods

OPA では，RO メソッド同士の並列実行だけではなく，RW メソッドと RO メソッドの並列実行も行うべく，instant メソッドの実行ではオブジェクトデータをコピーし，これを利用してメソッドを実行する．instant メソッドの動作を図 1 に示す．instant メソッドはまず最初に必要なオブジェクトの変数を暗黙の局所変数に一括読み出し (コピー) する．RW メソッドで変更された局所変数の値は，Schematic[2] と同様に，オブジェクト本体に一括して書き戻される．OPA では，それ以上の局所変数の更新がない点を一括更新点とし，フロー解析でこれを求める．

RO メソッド： (RO1) メソッドのルックアップ (RO2) メソッド起動 (RO3) オブジェクトの状態をローカル変数へ一括読み込み (RO4) メソッド本体の実行．

RW メソッド： (RW1) メソッドのルックアップ (RW2) メソッド起動 (RW3) 一括読み込み (RW4) 一括更新までの処理 (RW5) オブジェクトを一括更新 (RW6) メソッドの残りの実行．

RO1–RO3 を RO 区間，RW5 を W 区間，RW1–RW5 を RW 区間とする．排他制御は，データを読み出し，加工して書き戻すという処理の一貫性を保つために RW 区間実行間で，一貫性のあるデータを読み出すため RO 区間実行と W 区間実行の間でのみ必要になる．RO1，RW1 が排他制御が必要な区間に入っているのは，実行時メソッド置換を利用する場合，メソッドのルックアップから起動までと，オブジェクトの更新との間で排他制御が必要になるからである．

2.2 free メソッド

オブジェクトの変化しないデータを読み出すのみのメソッドは，排他制御なしで実行できる．OPA では，これを free メソッドと呼ぶ．オブジェクトの不変部分

はローカルな環境にコピーでき，これによってfreeメソッドがローカルに実行できる．よって，freeメソッドは分散メモリ環境での性能向上に特に有効である．

OPAでは，freeメソッドの解析もコンパイラが行う．実行時メソッド置換による状態の変化に即した解析を行うことで，実行時のfreeメソッドの増加も可能にしている．但し，この解析に要する計算量が爆発する場合は解析をあきらめることとする．つまりfreeメソッドは利用しなくてもかまわない．コンパイラの解析能力によって言語の意味論が変わるのは好ましくないため，OPAではfreeメソッドを言語の上では採用せず，コンパイラで最適化できた場合にのみ自動的に利用されるものとする．

3. 実行時メソッド置換

OPAでは，実行時メソッド置換を行なうことで，状態に応じたメソッドが呼び出されるようにする事ができる．これにより，状態に応じて緩和された排他制御規則を用いることや，freeメソッドを増やすことができるようになる．

3.1 文　　法

OPAでの実行時メソッド置換の利用法を示す．置換可能メソッドm1をメソッドm2に置換する場合，

```
setmethod(m1, m2);
```

のように記述する．置換後，そのオブジェクトに対するm1の呼び出しはm2の呼び出しに置き換えられる．

メソッドを初期状態に戻すには置換先にキーワードinitialを指定する．また，そのメソッドが今後呼べないこと(呼ぶと例外発生)を明示するためには，置換先にキーワードnullを指定する．置換結果の反映はインスタンス変数の一括更新時にまとめて行なう．また，置換可能メソッドには宣言部にキーワードreplaceableをつけるものとする．メソッド置換先が置換メソッドであった場合，参照が循環して無限ループとなる可能性があるが，これはプログラマの責任とする．

オブジェクト指向の立場からみると，メソッド置換はオブジェクトに届いたメッセージの名前を変更し，自オブジェクトに送り直すと定義できる(図**2**)．メッセージを送る側から実際に起動されるメソッドが特定できない点は通常の遅延束縛と同様である．

実行時メソッド置換では，置換先をメッセージではなく静的に決定されるメソッドそのものとする仕様も考えられ，この場合キーワードinitialが不要になる，循環参照が起こらない等の利点がある．しかし，親クラスで置換先として使用されているメソッドを置換可能メソッドで再定義した場合，再定義したメソッドが置換先として呼ばれる際に再ルックアップが行なわれないため，メソッドが正しく起動されない．この仕様では，他者が提供するクラスを拡張する場合にその実装を知る必要があり，クラスのインターフェースと実装の独立を保てなくなる．

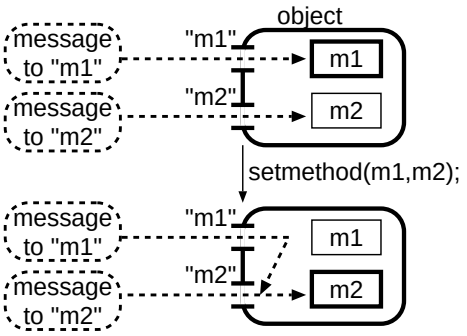

図2 実行時メソッド置換
Fig. 2 The message renaming by the dynamic method replacement

3.2 排他制御の実行時緩和

オブジェクトが生成され，ある程度処理が進むと(オブジェクトが成熟すると)，RWメソッドの中に，二度と呼ばれない，またはオブジェクトの更新を行なわないものが現れる場合がある．OPAでは，実行時メソッド置換により，更新を行なわなくなったRWメソッドを対応するROメソッドに置換することができる．

図**3**にプログラム例として二分木辞書クラスを示す．このプログラムにおいて，RWメソッドinsertは左右の子ノードが生成された後，インスタンス変数の更新を行なわなくなる．よって，等価な動作をするROメソッドinsertFに置換することで，処理が高速化できる．ROメソッドは並列実行可能であるので，木構造の根などでもボトルネックとならない．

RWメソッドをROメソッドに置換することで，(1)排他制御が必要な区間が縮小されることによるスレッド間の並列性向上，(2)置換先のROメソッドを記述する際に，その時点での処理に応じたメソッド内容の最適化が可能，の二点により処理の高速化が行なえる．

3.3 メソッド置換による free メソッドの実行時増加

実行時メソッド置換を利用してオブジェクトの状態遷移を解析することで，実行時にfreeとなったメソッドを利用した最適化が可能である．

freeメソッドであるためには，ROメソッドでかつ，メソッドが読み出す部分が今後変化しないことが必要である．注意しておきたいのは，オブジェクトから読

```
class BinTree{
 internal int key, value;
 internal BinTree left = null,right = null;
 public BinTree(int k, int val) {
   key = k; value = val;
 }
 public instant int search(int k, int v) { ... }
 public replaceable instant void insert(int k, int val) {
  if (k < key) {
    if (left != null) left.insert(k, val);
    else {
      left = new BinTree(k, val);
      if (right != null) setmethod(insert, insertF);
    }
  } else {
    if (right != null) right.insert(k, val);
    else {
      right = new BinTree(k, val);
      if (left != null) setmethod(insert, insertF);
    }
  }
 }
 internal instant void insertF(int k, int val){
  if (k < key) left.insert(k, val);
  else right.insert(k, val);
 }
}
```

図 3 実行時メソッド置換のプログラム例 (二分木辞書)
Fig. 3 Example of dynamic method replacement (binary tree dictionary)

み出す情報にはそのメソッドの置換情報も含まれるため，対象となるメソッドの置換が今後起こらないことも条件となることである．

free メソッドの中には，オブジェクトの生成と同時に free なものもあるが，これら定数の読み込みのみ行なうメソッドの数は限られている．一方，途中から変化しなくなった変数を含めて free メソッドを利用するには，オブジェクトの状態遷移を考慮した解析が必要である．但し，ユーザからの情報もなしに状態遷移を考慮した解析を行なうのは困難あるいは不可能である．我々は，実行時メソッド置換がオブジェクトの状態の変化に即して利用されることに着目することで，free メソッド解析を実現する．

オブジェクトの各インスタンス変数が更新される可能性があるか確かめるには，オブジェクトのその時点で呼ばれ得るメソッドの集合(メソッド集合)を知る必要がある．このためメソッド集合を決定するオブジェクトの状態 (メソッド状態) を単位にした解析を行なう．オブジェクトの状態遷移は，実行時メソッド置換によるメソッド状態の変化を解析することで捉えることができる．

解析の結果ある状態遷移で free になるメソッドが検出されたとする．この場合，対応するメソッド置換が起こった時点で当該メソッドを free メソッドとして扱い，各種の最適化を施すことが可能となる．

実行時の状況変化に応じて free メソッドの増加を図る別の手段として，変数の最終書き込みを明示するという手段も考えられる．しかし，誤った最終書き込み指定の静的な検出は困難であり，一方，実行時のチェックを行なう場合，変数への書き込み毎のオーバーヘッドが大きいという問題がある．

最後に，OPA では，アクセス制限を示す修飾子 `internal` を導入し，一切の外部からのアクセスを禁止することで，変数の更新を制限できるようにする．これは，現在の Java の文法で最も強いアクセス制限を示す修飾子 `private` でも，同じクラスの他オブジェクトからの直接参照を禁止できないためである．

4. free メソッドの検出アルゴリズム

4.1 アルゴリズム

free メソッド検出では，オブジェクトのその時点で呼ばれ得るメソッドの集合 (メソッド集合) を決定するメソッド状態の，メソッド置換による遷移を表した有向グラフを考える．ループを構成するノードをまとめて不可逆な状態遷移に関する情報を得るため，グラフの強連結成分を抽出し，この成分ごとに free メソッドの検出を行なう．以下この節では，必要な定義を行った後，free メソッド検出アルゴリズムの概略を述べる．

メソッドのルックアップは，メッセージ (a) をメソッド (m) に写す関数とみなせる．また，メソッドテーブル (T) は基本的にクラス毎に一つ用意される．ただし，free メソッドを利用する際は，一部のメソッドを free メソッドに置き換えたメソッドテーブルを別に (必要なら複数個) 用意しメソッドテーブルそのものを切り替えるものとする．

T はメッセージをテーブルエントリ (e) に写す：

$$T = \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\}$$

つまり，$T(a_i) = e_i \ (1 \leq i \leq n)$ で、$\mathrm{Dom}(T) = \{a_1, \ldots, a_n\}$ (定義域) である．各置換可能メソッドの置換情報を記憶するための場所は，その置換可能メソッドを持つオブジェクト毎に用意される．その状態が記憶される位置 x はサブクラスのおいても共通である．テーブルエントリ e はメソッドのタイプ (RO, RW, or Free) が指定されたメソッド m か，位置 x である：

$$e ::= \mathrm{RO}\, m \mid \mathrm{RW}\, m \mid \mathrm{Free}\, m \mid \mathrm{Replaceable}\, x$$

メソッド状態は $M = (T, S, H, R)$ で表される．ここで R は変名規則であり，各場所で置換情報 (対応するメッセージ) を保持する：

$$R = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$$

また、$S(m)$ は，メソッド m の実行により発生し得るメソッド置換（これも R で表す）の集合を与える：

$$S = \{m_1 \mapsto \{R_{11}, \ldots, R_{1l_1}\}, \ldots, m_n \mapsto \{R_{n1}, \ldots, R_{nl_n}\}\}$$

例えば，m の実行により「x_1, x_2 に置換情報を記憶するメソッドの呼び出しについて，それぞれの置換情報を a_1, a_2 に変更する」ような二つの setmethod 文がそれぞれ独立の条件下で実行され得る場合，$S(m) = \{\{x_1 \mapsto a_1\}, \{x_2 \mapsto a_2\}, \{x_1 \mapsto a_1, x_2 \mapsto a_2\}\}$ となる．また，集合 $H = \{a_1, \ldots, a_n\}$ は，置換可能メソッドそれぞれの背後に導入されたメッセージの集合で，ユーザはこれらのメッセージを直接（置換可能メソッドを使うことなく）送ることはできない．これら隠れたメッセージは，各置換可能メソッドの置換情報の初期値となるとともに，T により各置換可能メソッドの初期メソッド用エントリに写される．

メソッド置換 R によって，変名規則 R_1 は以下のように関係する場所を上書きすることで，R_2（R_1R と表記）に更新される：

$$\mathrm{Dom}(R_2) = \mathrm{Dom}(R_1) \cup \mathrm{Dom}(R)$$
and
$$R_2(x) = \begin{cases} R_1(x) & \text{if } x \in \mathrm{Dom}(R_1) - \mathrm{Dom}(R) \\ R(x) & \text{if } x \in \mathrm{Dom}(R) \end{cases}$$

メソッドルックアップ関数 *lookup* は以下のように定義される：

$$\mathit{lookup}(a, T, R) = \mathit{rename}(T(a), T, R)$$

where，
$\mathit{rename}(\mathrm{RO}\, m, T, R) = m$
$\mathit{rename}(\mathrm{RW}\, m, T, R) = m$
$\mathit{rename}(\mathrm{Free}\, m, T, R) = m$
$\mathit{rename}(\mathrm{Replaceable}\, x, T, R) = \mathit{lookup}(R(x), T, R)$

この定義の直観的な意味は，図 5 も参照してほしい．

メソッド状態の可能な遷移 $M_1 \to M_2$ は以下のように定義できる：

$M_1 \to M_2$
iff $((\exists m \in \mathit{MS}(T_1, H_1, R_1) \,.\, \exists R \in S(m) \,.\, R_2 = R_1 R) \wedge T_1 = T_2 \wedge S_1 = S_2 \wedge H_1 = H_2)$
where
$M_1 = (T_1, S_1, H_1, R_1)$ and $M_2 = (T_2, S_2, H_2, R_2)$

ただし，$\mathit{MS}(T, H, R)$ は，現在のメソッド状態において，呼び出され得るメソッド集合を与えるもので，以下のように定義される：

$$\mathit{MS}(T, H, R) = \{m \mid \exists a \in (\mathrm{Dom}(T) - H) \,.\, m = \mathit{lookup}(a, T, R)\}$$

ここで，直接送ることのできないメッセージは除かれていることに注意してほしい．

以上の定義を元に free メソッド検出アルゴリズムの概要を以下に示す．また，これを図 4 に示す．

free メソッド検出アルゴリズム

入力:初期メソッド状態 $M_0 = (T_0, S_0, H_0, R_0)$
出力:遷移条件付き DAG(a Directed Acyclic Graph) $N_{out} = (G_{out}, c_{out})$

G_{out}：頂点をメソッドテーブル（T）とする DAG

c_{out}：G_{out} の辺に対して，遷移条件 $\{R_1, \ldots, R_n\}$ を対応させる関数

遷移条件 $\{R_1, \ldots, R_n\}$：メソッド状態の変名規則が $R_1, \ldots, R_n$ のいずれかであること

出力の意味：遷移条件が成立した時，メソッドテーブルを G_{out} に沿って切替えられる．

1．初期状態をセットする．
$G_0 = (V_0, E_0) = (\{M_0\}, \{\})$

2．$G = G_{n+1} = G_n$ が成立するまで以下の式にしたがって G_n $(= (V_n, E_n))$ から G_{n+1} $(= (V_{n+1}, E_{n+1}))$ を計算し、有向グラフ G を生成する．(図 4-(1))

$$(V_{n+1}, E_{n+1}) = (V_n \cup \{M' \mid \exists M \in V_n \,.\, M \to M'\}, E_n \cup \{(M, M') \mid \exists M \in V_n \,.\, M \to M'\})$$

3．G 上すべての強連結成分を検出する．強連結成分を頂点とするグラフ G_{sc}(図 4-(2)) は DAG の形を成す．

4．各強連結成分毎に free メソッドを検出する．この際，G_{sc} 上末端側の頂点から行なうことで重複した計算を避けられる．(a, m) が free メソッドである条件は，(1) メソッドの型が RO である，(2) G_{sc} 上現在以降のどの頂点においても，a における *lookup* の結果が常に m となる，(3)G_{sc} 上現在以降のどの頂点においても，MS で計算されるメソッド集合に含まれるメソッドが，m が読み出す変数の更新を行なわない．

5．各強連結成分毎に検出される free メソッドにつき T を更新した T' を準備する．

$$T' = T\{a_1 \mapsto \mathrm{Free}\ m_1, \ldots\}$$

6．G_{out} として T' を頂点とする G_{sc} と同形（ただし，T' が同一の頂点は一つにする）の DAG を構成すると共に，その辺に c_{out} の与える遷移条件を $\{R \mid (T, S, H, R)$ が G_{sc} の辺に対応する G 上のいずれかの辺の終点である $\}$ とする．(図 4-(3))

4.2 解 析 例

2 分木辞書クラスにおける G は以下のようになる．ただし，`search` メソッドは考慮に入れてない．

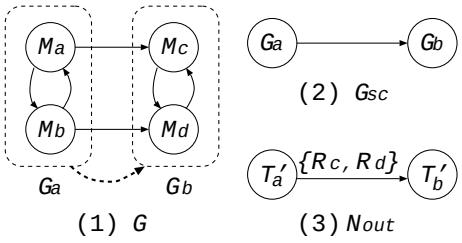

図 4 free メソッド検出アルゴリズム
Fig. 4 Graphs for free method detection algorithm

$$G = (\{M_0, M_1\}, \{(M_0, M_1)\})$$

where,

$M_0 = (T_0, S_0, H_0, R_0) \quad M_1 = (T_1, S_1, H_1, R_1)$

$T_0 = T_1 = \{\texttt{insert} \mapsto \text{Replaceable } x_{\text{insert}},$
$\texttt{insertF} \mapsto \text{RO } m_{\text{insertF}},$
$a_{\text{insert}} \mapsto \text{RW } m_{\text{insert}}\}$

$S_0 = S_1 = \{m_{\text{insert}} \mapsto \{\{x_{\text{insert}} \mapsto \texttt{insertF}\}\},$
$m_{\text{insertF}} \mapsto \{\}\}$

$H_0 = H_1 = \{a_{\text{insert}}\}$

$R_0 = \{x_{\text{insert}} \mapsto a_{\text{insert}}\} \quad R_1 = \{x_{\text{insert}} \mapsto \texttt{insertF}\}$

この例において M_0 ,M_1 は，それぞれ強連結成分である．結局得られる結果は以下の通りである．

$$N_{out} = (G_{out}, c_{out}),$$
$$G_{out} = (\{T_0, T_1'\}, \{(T_0, T_1')\}),$$
$$c_{out} = \{(T_0, T_1') \mapsto \{R_1\}\}$$

where,

$T_1' = \{\texttt{insert} \mapsto \text{Free } m_{\text{insertF}},$
$\texttt{insertF} \mapsto \text{Free } m_{\text{insertF}},$
$a_{\text{insert}} \mapsto \text{RW } m_{\text{insert}}\}$

ここで， $a_{\text{insert}} \in H_1$ なので，RW m_{insert} が呼び出されることはない．

5. 実　　装

OPA のコンパイラは，OPA のソースプログラムを C コードに変換する．ランタイムは各プロセッサに一つ OS のプロセスを生成し，OPA のスレッドは全てこのランタイムで管理される．これらのプロセスは，共有メモリ環境では仮想メモリ空間を共有する．

5.1 オブジェクトの実装

オブジェクトは，C 言語上では構造体で表現され，メソッドテーブルへの参照，インスタンス変数テーブル，排他制御のための管理情報などで構成される．

メソッドテーブルは，そのオブジェクトに対するメッセージと起動されるコードの対応を記憶し，同じクラスのオブジェクト間で共有される．各メソッドについて，メソッドが置換可能でなければ，メソッドタイプとメソッドを実現している C の関数への参照が格納される．置換メソッドの場合，メソッドの置換情報はオブジェクトのインスタンス変数テーブルに格納され，メソッドテーブルにはメソッドタイプ replaceable とオブジェクト内のメソッドの置換情報へのオフセットが格納される．置換情報は，メソッドテーブル内の置換先メソッドのメソッド情報へのオフセットからなる．置換メソッドのルックアップは図 5 のように表される．

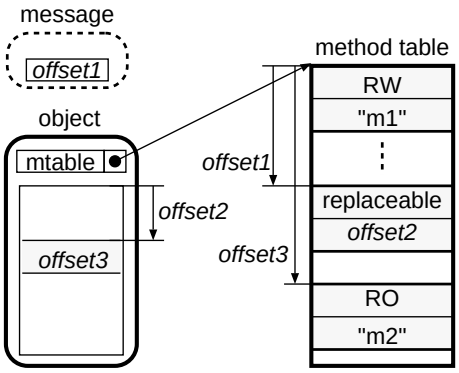

図 5 置換メソッドのルックアップ
Fig. 5 The method lookup for replaceable method

5.2 置換メソッドの起動シーケンス

2.1章で述べた通り，instant メソッドの排他制御は RW 区間実行間と，RO 区間実行と W 区間実行の間でのみ行なわれる．ただしメソッド置換がある場合，メソッドを取り出すまでメソッドタイプが RO，RW，free のいずれか分からず，また置換がいつ発生するかも分からないため，メソッドのルックアップ自体にも排他制御が必要になる．OPA では a) RW 区間実行中を表すオブジェクト使用中フラグと実行待ちスレッドキュー，b) RO 区間と W 区間の相互排他を実現するバージョン番号 [3]，c) これらを原子的に更新するためのロック (busy-wait lock) により必要な排他制御を実現する．

ルックアップの結果 RO メソッドが起動される場合，ルックアップから一括読み込み完了まで，RW メソッドが起動される場合，ルックアップからオブジェクトの使用中フラグのセットまでの区間を，オブジェクトの更新 (W 区間) と排他制御する．RW メソッドを起動する場合に，使用中フラグをセットできなければスレッドをオブジェクトのキューに enqueue して実行を中断する．これは実行中の RW 区間終了時などに dequeue される．実装の上では，ルックアップで得られたメソッドが RO のときに，高コストなロック操作などを使わないようにする工夫が重要となる．

ルックアップを行なう際はオブジェクトをロックせ

ず，ルックアップ後に更新が発生していればリトライする方式をとる．OPA では instant メソッドによりオブジェクトへの書き込み頻度が低く，またルックアップ区間自体が短いため，このような楽観的な戦略が有効に働く．ただし，頻繁に更新が発生する場合，ルックアップが繰り返し成功せず処理が進まない可能性があるため，再ルックアップはオブジェクトをロックして行なう．図 **6** に，疑似コードによる `obj.message(args)` に対応するメソッド起動手順を示す．

RO 区間と W 区間の排他制御は 4) を改良したバージョン番号方式で実現する．各オブジェクトに番号 (初期値 1，更新中 0，更新後 更新前の値+2) をつけてオブジェクトを管理する．オブジェクトの読み出し前後でのバージョン番号を比較することにより，読み出し中の更新の有無を判断できる．この方式は load と store のみで実現されるため，複数のスレッドが同時に値を読み出せ，低コストで高い並列性が実現できる．

オブジェクトの更新中にルックアップを開始しないため，ルックアップ前のバージョン番号を保存する際，バージョン番号が 0 でなくなるまで待つ．よって番号は必ず奇数となり，偶数を特別な状態の通知に利用できる．0 でオブジェクトがロックされている場合 (一括読み込み後ロック解放) を，2 でその RO メソッドが free メソッドである場合 (ルックアップ，一括読み込みが無条件で成功) を表す．

以上により，メソッドを取り出すまでメソッドタイプが不明な置換メソッドを RO メソッドの軽さを残しつつ安全に起動できる．

5.3 分散環境での free メソッドの利用

分散メモリ環境では他ノードのメモリを直接参照できないため，排他制御は共有メモリの場合より単純に実装できる．リモートオブジェクトへのアクセスは，自ノード上にオブジェクトごとに生成したコピー (プロキシ) を介して，リモートメソッド呼び出しを行なうことで実現する．このとき無視できない遅延が発生するが，これはオブジェクトの不変部分をプロキシにコピーし，free メソッドをローカルに実行可能とすることで解消できる．

プロキシのメソッドが呼ばれた場合，メソッドが free でなければプロキシはオブジェクトが存在するノードのサーバに依頼し，リモートメソッド呼び出しを行なう．メソッドの実行後，呼ばれたメソッドが free になっていれば，サーバは戻り値に加え，オブジェクトのコピーをメッセージとして返す．これを受けとったプロキシは，以後同じメソッドの呼びだしを全てローカルに処理することが出来る．

```
START:  /* wait while  objbeing updated by other threads */
do { vn=obj.version_num; } while(vn==0);
RETRY:
switch(method = lookup(obj, message)) {
case RO:
  status = invoke(method, vn, obj, args);
  if (status == ROfail) { /* objwas updated before read(s) completed */
    spin_lock(obj); vn=0; goto RETRY;
  }
  break;
case RW:
  if (vn) {
    spin_lock(obj);
    if (vn != obj.version_num){vn=0; goto RETRY;}
  }
  /* check if used by a thread in RWsection */
  if (obj.use) {suspends(obj); goto START;};
  obj.use = TURE;
  unlock(obj);
  status = invoke(method, obj, args);
  break;
case Free:
  if (vn == 0) unlock(obj);
  status = invoke(method, 2, obj, args);
  break;                  /* "vn = 2" means method is free */
}

ROmethod(vn, obj, args) {
  (read instance variables)
  if (vn != o.version_num)
    switch(vn) {
      case 0 : unlock(obj); break;
      case 2 : break; /* method is free */
      default : return ROfail;
    }
  (method body)
}
```

図 **6** 疑似コードによる置換メソッドの起動シーケンス

Fig. 6 Pseudo code of method lookup and method invocation for **obj.message(args)** on shared memory architectures

6. 評　　価

処理系を実装し，性能評価を行なった．

共有メモリ環境の実装対象として，SGI 社製 POWER Onyx (MIPS R10000(195MHz) × 12 個，メモリ 2GB) を用いた．分散メモリ環境の実装対象としては，Sun Microsystems 製 WS SPARCstation5 (microSPARCII 110MHz)4 台を完全結合した WS クラスタを用いた．WS 間の結合には高速シリアル通信インターフェース STAFF-link[5] を用いた．

評価には図 3 の二分探索木クラスを用いた．このクラスは実行時メソッド置換により，RW メソッドを RO メソッドに置換して処理効率を改善できる (3.2節)．また free メソッド解析がある場合，メソッド置換後に探索，挿入両メソッドが free メソッドとなる (4.2節)．

このクラスを用いて，ランダムに生成したキーによって単一の木構造に 30,000 個の要素を登録するのに要した時間を評価基準とした．各プログラムは 4 回実行し，2 番目に速い結果を採用した．

共有メモリ用処理系の評価に，以下の 3 種類のプログラムを用いた．

- **Adaptive** メソッド置換，free メソッドを利用

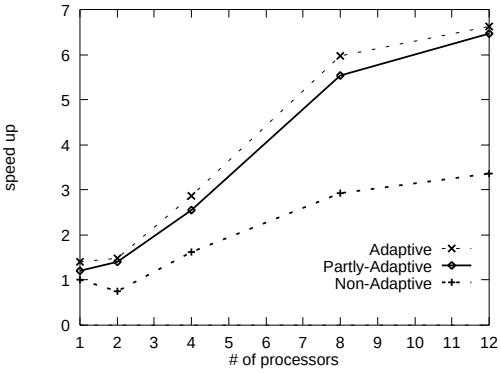

図 7 共有メモリ環境における性能比較
Fig. 7 The speedup with dynamic method replacement (Adaptive / Partly-Adaptive) on a shared memory architecture

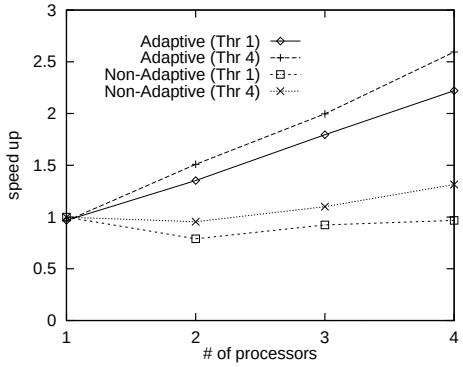

図 8 分散メモリ環境における性能比較
Fig. 8 The speepup with dynamic method replacement (Adaptive) on a distributed memory architecture

- **Partly-Adaptive** メソッド置換のみ利用 (free メソッドは利用しない)
- **Non-Adaptive** メソッド置換を用いない

Non-Adaptive，PE 数 1 の場合 (0.949s) を 1 とした，各プログラムの性能向上を図 **7** に示す．

全体的に，ほぼ Adaptive，Partly-Adaptive，Non-Adaptive の順に良い性能が出ている．並列性のない PE 数 1 の場合より，実行時メソッド置換によってメソッドの処理が軽くなり，free メソッドによってそれがさらに改善されたことがわかる☆．複数 PE 下では実行時メソッド置換による処理の並列性向上により，Non-Adaptive との性能差がより顕著になっている．また，共有メモリ環境では RO メソッドと free メソッドの間に大きな性能差はないことが確認された．これは，分散メモリ環境では free メソッドのときのみソフトウェアで行なわれる不変部分の複製が，ハードウェアキャッシュで自動的に行なわれているためである．

分散メモリ用処理系の評価には，Adaptive，Non-Adaptive について，プロセッサあたりのスレッド数を 1 または遅延隠蔽のため 4 とした，計 4 種類のプログラムを用いて測定を行なった．Non-Adaptive，PE 数 1，Thr 数 1 の場合 (2.80s) を 1 とした，各プログラムの性能向上を図 **8** に示す．

図より，PE 数 2 ~ 4 では Adaptive が Non-Adaptive と比較してほぼ 2 倍速くなっている．また，同じプログラムの PE あたりのスレッド数が 1 と 4 の場合を比較して，分散環境ではスレッドの多重化による遅延隠蔽が有効であることが示された．

☆ 挿入メソッドは約 52 万回呼ばれ，80 %以上を RO または free として処理できている．

7. 関連研究

Actors[6] には become コンストラクトがあり，これは実行時にオブジェクトのクラス全体を入れ換えるものに相当する．OPA では，ユーザにはメソッド置換のみを提供することで，メソッドテーブルの切替えといった低次の処理を隠蔽している．

Concurrent Aggregates[7] は，プログラマが任意の逐次オブジェクトを組み合わせ，これらを一つのオブジェクトとして扱うことでオブジェクト内並列処理を実現する aggregate という機構を持つ．OPA では同種の機能をよりシステム化された方法で提供している．

文献 8) では，部分計算を利用し，変化しなくなった変数に適応したメソッドのコードを実行時に生成する．free メソッドは変化しなくなったインスタンス変数のみを利用するため，実行時生成されたコードは無効化されることが無い．実行時メソッド置換は free メソッドを実行時に増加させてゆく技術であり，これと実行時コード生成を組み合わせることにより，処理系の性能をより向上させることが出来ると考えられる．

8. 結論と今後の課題

本論文では，並列処理において言語がデータの一貫性を保証する場合に，問題の性質を活かした処理が行なえるよう，状態に応じた排他制御規則が利用可能なオブジェクト (適応的オブジェクト) と，その記述手段として実行時メソッド置換を提案した．また，提案方式の共有メモリ型，分散メモリ型並列計算機への実装と，処理系の性能評価結果について述べた．この結果，提案方式の有効性が確認された．

今後はより総合的な性能評価，実行時コード生成機能の処理系への組み込みの検討を行なう．

謝辞　STAFF-Link を使用した WS クラスタの使用に便宜を図っていただいた神戸大学の中條拓伯博士に感謝いたします．本研究の一部は文部省科学研究費（奨励 (A)09780278）による．

参 考 文 献

1) Arnold, K. and Gosling, J.(eds.): The Java Programming Language, Addison-Wesley Publishing Company (1996).
2) Taura, K. and Yonezawa, A.: Schematic: A Concurrent Object-Oriented Extension to Scheme, Technical report, Dept. of Information Science, University of Tokyo (1995).
3) 八杉昌宏, 瀧和男: 並列処理のためのオブジェクト指向言語 OPA の設計と実装, 情報処理学会研究報告, Vol. 96, No. 82, pp. 157–162 (1996).
4) Lamport, L.: Concurrent reading and writing, *CACM*, Vol. 20, No. 11, pp. 806–811 (1977).
5) Nakajo, H., Ichikawa, A. and Kaneda, Y.: An Implementation and Evaluation of a Distributed Shared-Memory System on Workstation Clusters Using Fast Serial Links, *Proc. of Int.Symp. on High Performance Computing(ISHPC)*, pp. 143–158 (1997).
6) Agha, G.: *Actors : A Model of Concurrent Computation in Distributed Systems*, The MIT Press (1987).
7) Chien, A. A.: *Concurrent Aggregates*, The MIT Press (1993).
8) Fujinami, N.: Automatic Run-Time Code Generation in C++, *Proc. of ISCOPE'97*, LNCS, No. 1343, Sprnger-Verlag (1997).

(平成 8 年 2 月 4 日受付)
(平成 8 年 5 月 11 日採録)

江口　重行

1975 年生．1997 年神戸大学工学部情報知能工学科卒．1999 年同大学大学院自然科学研究科情報知能工学専攻博士前期課程修了．修士（工学）．並列・分散処理，言語処理系などに興味を持つ．

八杉　昌宏（正会員）

1967 年生．1989 年東京大学工学部電子工学科卒．1991 年同大学大学院電気工学専攻修士課程修了．1994 年同大学大学院理学系研究科情報科学専攻博士課程修了．1993 ~ 1995 年日本学術振興会特別研究員（東京大学，マンチェスター大学）．1995 年神戸大学工学部助手．1998 年より京都大学大学院情報学研究科通信情報システム専攻講師．博士（理学）．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

鎌田十三郎（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒．1995 年同大学大学院理学系研究科情報科学専攻修士課程修了．1998 年同博士課程単位修得退学．1996 ~ 1998 年日本学術振興会特別研究員（東京大学）．1998 年より神戸大学工学部助手．修士（理学）．並列・分散処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM 会員．

瀧　　和男（正会員）

昭和 27 年生．昭和 51 年神戸大学工学部電子工学科卒業．昭和 54 年同大学院修士課程システム工学修了．工学博士．同年（株）日立製作所入社．昭和 57 年（財）新世代コンピュータ技術開発機構に出向．遂次型および並列型推論マシンと並列応用プログラムの研究開発に従事．平成 2 年同機構第 1 研究室室長．平成 4 年 9 月神戸大学工学部情報知能工学科助教授．平成 7 年 4 月同学科教授．LSI 設計技術と CAD，並列処理とマシンアーキテクチャ，脳型コンピュータ等に興味を持つ．電子情報通信学会，IEEE，ソフトウェア科学会，ACM，日本神経回路学会各会員．