排他的なメソッドの並行な呼び出しを融合する機構を持つ言語

大山恵弘[†,††] 田浦健次朗[†] 米澤明憲[†]

並行拡張されたオブジェクト指向言語に，排他的なメソッドの複数の並行な呼び出しを融合する機構を導入する方法を示す．通常，排他的なメソッドの複数の並行な呼び出しは逐次化される．我々は逐次化された結果として実行待ち状態にある複数の排他的なメソッドの呼び出しを融合する．具体的には，それらの呼び出しを，たとえば単一の呼び出しなどのより高速な処理に動的に切り替える．たとえば，カウンタオブジェクトに対する，1を加算するメソッドの呼び出しと，2を加算するメソッドの呼び出しを融合して，3を加算するメソッドのみを呼び出す．どんな呼び出しの組をどんな処理に融合するかの仕様（融合規則）はプログラマが記述する．本機構は，排他的なメソッドの呼び出しが長時間実行待ち状態になるオブジェクト（同期ボトルネック）での効率化に特に役立つ．我々は本機構を持つ言語の処理系を実装し，共有メモリ並列計算機 Sun Enterprise 10000 を用いて実験を行った．その実験では，本機構が同期ボトルネックを持つプログラムの性能を大きく向上させた．

Fusion of Concurrent Invocations of Exclusive Methods

Yoshihiro Oyama,[†,††] Kenjiro Taura[†] and Akinori Yonezawa[†]

This paper describes a mechanism for “fusing” concurrent invocations of exclusive methods. The target of our work is object-oriented languages with concurrent extensions. In such languages, invocations of exclusive methods are serialized: only one invocation is executed at one time and the others must wait for their turn. The proposed mechanism fuses multiple waiting invocations to a cheaper operation, such as a single invocation. For example, when an add-1 method and an add-2 method are waiting on a counter object, we replace these methods with an add-3 method. Programmers describe fusion rules, which specify method invocations that can be fused and an operation that is used to substitute for the invocations. The mechanism works effectively in the execution of synchronization bottlenecks, which are objects on which exclusive methods wait a long time for their turn. We implemented a language that has the mechanism and tested the usefulness of the mechanism through experiments on a symmetric multiprocessor, the Sun Enterprise 10000. We confirmed that the mechanism made programs with synchronization bottlenecks significantly faster.

1. はじめに

並行オブジェクト指向言語と並行拡張されたオブジェクト指向言語の多くには，1つのオブジェクトに複数のスレッドによって並行に呼び出された複数のメソッドを排他的に，すなわち逐次化して実行するための機構が用意されている．たとえば，Java においては，1つのオブジェクトに対して同時にたかだか1つの synchronized メソッドしか実行されない．Java の synchronized メソッドのように，1つのオブジェクトに対して同時に複数呼び出されたとき，逐次化されて実行されるメソッドを，以降では**排他メソッド**と呼ぶ．

本研究の目標は，複数のスレッドが1つのオブジェクトに対し並行に排他メソッドを呼び出す状況において，逐次化された複数の排他メソッドの実行を，より高速な処理の実行（たとえば1つの排他メソッドの実行）に動的に切り替える（「融合」する）ことである．たとえば，カウンタオブジェクトに対する，1を加算するメソッドの呼び出しと，2を加算するメソッドの呼び出しを融合して，3を加算するメソッドのみを呼び出す．別の例をあげると，GUI のウィンドウオブジェクトに対し，repaint メソッドがほぼ同時に複数回呼び出されたとき，repaint メソッドを1回だけ実行する．以降では，この方法を**メソッド融合**と呼ぶ．メソッド融合は，**同期ボトルネック**（あるスレッドによって排他メソッドが実行されている間に，別のスレッドが排他メソッドを呼び出すことが頻繁に起こ

† 東京大学大学院理学系研究科情報科学専攻
Department of Information Science, Faculty of Science, University of Tokyo
†† 日本学術振興会特別研究員
JSPS Research Fellow

るオブジェクト）での実行を高速化する．

本論文が対象とする言語は，並行オブジェクト指向言語と並行拡張されたオブジェクト指向言語（例：Java）である．本論文が対象とする計算機は共有メモリ並列計算機である．

排他メソッドの実行の効率化に関して，これまでに様々な技術が提案された．まず，同じメンバ変数を更新しない複数の排他メソッドを並行に実行する技術が存在する[4),20),23)]．そして，同期ボトルネックとなるオブジェクトの複製を作る技術が存在する[3),18)]．前者は，同じメンバ変数を更新する排他メソッドの組合せ，たとえば上述の加算メソッドの複数の呼び出しを効率化できない問題を持つ．後者は，実行するメソッドの回数や種類の変化を，たとえば上述の repaint メソッドの呼び出しの融合を簡潔には記述できない問題を持つ．メソッド融合はそれらの問題を解決している．

本論文の貢献を以下に示す．

- 1 つのオブジェクトに対して並行に呼び出された複数の排他メソッドの実行を融合する枠組みを最初に提案し，そのための言語設計と実装方式を示した．
- 上記の枠組みを言語処理系に実装し，並列計算機上での実験を通してその有効性を確認した．

本論文は以下のように構成される．2 章でメソッド融合の概略を述べる．3 章で対象言語について述べる．4 章でプログラム例を示す．5 章で対象言語の設計について議論する．6 章で対象言語の実装を解説する．7 章で実験結果を報告する．8 章で関連研究と本研究を比較する．9 章で本論文を総括する．

2. メソッド融合の概略

メソッド融合の概略を，以下の Java プログラムを用いて説明する．

```
class Counter {
  private int value;
  ...
  public synchronized void inc(int n) {
    value += n;
  }
  public synchronized int get( ) {
    return value;
  }
}
```

このクラスは整数の値を保持するカウンタオブジェクトを作る．inc メソッドはカウンタの値をその引数の値だけ増やす．get メソッドはカウンタの値を返す．

Counter クラスのオブジェクトに対し inc(x) と inc(y) を実行することと inc(x + y) を実行することは，最終的に得られるカウンタの値に関して同じ効果を与える．そこで我々は Counter クラスのオブジェクトに対して呼び出されたが排他処理により実行待ち状態にある inc(x) と inc(y) を inc(x + y) に融合する．

我々の対象言語では，融合されるメソッド呼び出しの組と，それらが融合した結果実行される文を，プログラマが**融合規則**の形で記述する☆．たとえば，Counter クラスの定義に以下のように融合規則を加える．

```
class Counter {
  ...
  fusion void inc(int x) & void inc(int y) {
    inc(x + y);
  }
}
```

上の融合規則は，

> Counter クラスのオブジェクトに対して，
> inc(x) と inc(y) を実行する代わりに，
> inc(x + y) のみを実行してもよい

ことをコンパイラに教えている．対象言語のランタイムは，この融合規則に従い，カウンタオブジェクトに 2 つの inc メソッドが呼び出されたとき，それらを 1 つの inc メソッドの呼び出しに融合するかもしれない（しないかもしれない）．融合の結果実行される呼び出し inc(x + y) は，さらに別の inc メソッドの呼び出しと融合されるかもしれない．

3. 言語仕様

対象言語は C++から継承と例外の機構を取り去り，スレッドと排他メソッドによって拡張した言語である．

3.1 スレッドと排他メソッド

対象言語にはスレッド生成のためのプリミティブが用意されており，それを実行することによってスレッドを動的に生成できる．

宣言に sync キーワードが付加されたメソッドが排他メソッドである．Java における synchronized メソッドと同じく，排他メソッドは 1 つのオブジェクトで同時に複数実行されない．一方，排他メソッドでないメソッドは他のあらゆるメソッドと並行に実行される．対象言語は，Java と異なり，排他メソッドの本体

☆ コンパイラがプログラムを解析することによって融合規則を自動生成できるかもしれないが，本論文ではその問題は扱わない．その問題と，与えられた融合規則に従って融合処理を実行する問題は独立している．

中で receiver object に対して再び排他メソッドを呼び出すことを許さない☆.

対象言語の仕様は，排他メソッドがオブジェクトに対して呼び出された順番と，それらの排他メソッドが実行される順番の関係については何も規定しない．言い換えれば，対象言語の実装は，実行待ち状態にある複数の排他メソッドの呼び出しの実行順を自由に入れ換えてもよい．この仕様は，単純なスピンロックなどの単純なロックで排他メソッドを実装することを可能にする．

3.2 融合規則

メソッド融合の仕様は，プログラマがクラス定義に融合規則を加えることによって与えられる．以下に融合規則の文法を示す．

```
fusion t_p  p(t_{x_1} x_1, ..., t_{x_m} x_m)
     & t_q  q(t_{y_1} y_1, ..., t_{y_n} y_n) {
    S
}
```

p, q はメソッド名である．p と q は同じでもよい．$x_1, \ldots, x_m, y_1, \ldots, y_n$ はすべて異なる変数である．$t_{x_1}, \ldots, t_{x_m}, t_{y_1}, \ldots, t_{y_n}$ はそれぞれ $x_1, \ldots, x_m, y_1, \ldots, y_n$ の型である．t_p, t_q はそれぞれメソッド p, q の返り値の型である．S は文である．以降では S を**融合規則の本体**と呼ぶ．

以下に融合規則の意味を示す．上記の融合規則がクラス C の定義の中に記述されたとする．さらに，クラス C のオブジェクト O に対して，2 つのメソッド呼び出し $p(x_1, \ldots, x_m)$ と $q(y_1, \ldots, y_n)$ が呼び出されたが，どちらもまだ O での他の排他メソッドの実行の終了を待っている状態にあるとする．そのとき，その融合規則は，**それらの 2 つのメソッド呼び出しを実行せず，文 S を実行してもよい**ことをコンパイラに伝える．詳細を以下に記す．

- S はオブジェクト O における他の排他メソッドの実行と並行に実行される．
- S の中で receiver object を指定せずにクラス C のメソッドが呼び出された場合，そのメソッド呼び出しの receiver object は O である．
- p と q の呼び出しは，それらが呼び出された順番に関係なく融合される．すなわち，融合規則の `&` の前と後を入れ換えても，その規則の意味は変わらない．たとえば，

  ```
  fusion void p(void) & void q(void) { ... }
  ```

  という融合規則をすでに持つクラスに，

  ```
  fusion void q(void) & void p(void) { ... }
  ```

  という融合規則を加えることには意味がない．
- p と q の呼び出し元に返される返り値については以下の仕様で定める．

  t_p と t_q の片方だけが void 型であるとき： S は return 文によって実行を終えなければならない．その return 文の返り値が，返り値が void 型でない方のメソッドの呼び出し元に返される．

  t_p も t_q も void 型でないとき： S は mreturn 文によって実行を終えなければならない．mreturn 文は 2 つの値を返すために我々が導入するプリミティブである．S の中で

  ```
  mreturn a and b
  ```

  という文を実行すると，a が p の呼び出し元に，b が q の呼び出し元に返される．

3.3 融合が行われる場面

融合されるのは動的に逐次化された複数の排他メソッドの呼び出しである．プログラムの文面に連続して現れる複数の排他メソッドの呼び出しの静的な融合は行わない．たとえば，2 章に示した `Counter` クラスのオブジェクト `c` に対して排他メソッドを順番に呼び出す以下のプログラムに静的に融合規則を適用することは考えない．

```
c->inc(3);
c->inc(5);
```

より複雑な例である以下のプログラムに静的に融合規則を適用するプログラム変換は reduction 変換[6),11),15)]にきわめて似ており興味深いが，やはり本論文では扱わない．

```
for (i = 0; i < n; i++) {
  c->inc(a[i]);
}
```

4. プログラム例

本章ではプログラム例を 3 つ示す．他のプログラム例を付録の A.1 に付した．

4.1 GUI のイベント処理

以下のプログラムは，`Window` クラスのオブジェクトに対する 2 つの `repaint` メソッドの呼び出しを，1 つの `repaint` メソッドの呼び出しに「まびき」する．

```
class Window {
  ...
  fusion void repaint(void)
       & void repaint(void) {
```

☆ それを許すように言語仕様および言語実装を拡張することは可能である．

```
    repaint();
  }
}
```

4.2 バッファへの操作

put メソッドと get メソッドを持つバッファオブジェクトのクラス Buffer を考える．融合規則によって，put メソッドの呼び出しと get メソッドの呼び出しを，バッファを操作せず「バイパス処理」することができる☆．

```
class Buffer {
  int length;
  obj* elements[MAXBUFFERLEN];
  ...
  sync void put(obj* o) {
    elements[length++] = o;
  }
  sync obj* get(void) {
    return elements[--length];
  }
  fusion void put(obj* o)
       & obj* get(void) {
    return o;
  }
}
```

4.3 データベース検索

Web サーバを表現するオブジェクトのクラス WebServer を考える．そのクラスのメソッド get は，引数が表現する URL に存在するファイルの文字列データを返す．融合規則によって，同じ URL を引数に持つ 2 つの get メソッドの呼び出しを融合できる．すなわち，get メソッドを 1 回だけ実行して（web のデータをディスクから 1 回だけ読んで），その結果のデータを get メソッドの複数の呼び出し元に返せる．

```
class WebServer {
  ...
  sync char* get(char *url) {
    return fetchFromDisk(url);
  }
  fusion char* get(char *url1)
       & char* get(char *url2) {
    if (strcmp(url1, url2) == 0) {
      char *d = get(url1);
      mreturn d and d;
    } else original;
  }
}
```

original は，融合されたメソッド呼び出し（この場合は get(url1) と get(url2)）を融合規則の本体から再び呼び出すためのプリミティブである．

現実の web サーバに対し要求される各々の URL の頻度には大きな偏りがある．たとえば index.html が要求される頻度が抜きんでて高い．よって，2 つの get メソッドの呼び出しが同じ URL の引数を持つことを期待するのは，理にかなっている．

5. 議　　論

5.1 メソッド融合の用途

メソッド融合の我々が考える用途は性能向上である．メソッド融合によってプログラムの挙動を変えることは（可能ではあるが）我々は意図していない．融合規則はあくまで性能ヒントであるべきである．そうでないと，プログラムが読みにくく誤りを含みやすいものになる．

プログラムの挙動を保存する融合規則を我々は**透明な**融合規則と呼ぶ．透明な融合規則とは，別の言葉でいえば，その規則による融合処理が行われたかどうかを，プログラマが性能以外から判定できない融合規則である．たとえば，2 章で示した Counter クラスに書かれた融合規則は透明である．一方，加える前のプログラムのいかなるタイミングの実行でも起こりえない挙動を，加えた後のプログラムが示しうる融合規則は透明でない☆☆．

我々の長期的な目標の 1 つは，できるだけ広い範囲の透明な融合規則を受け入れ，できるだけ広い範囲の透明でない融合規則にエラーを出す言語処理系を構築することである．

5.2 プログラマが融合処理を実装する方法との比較

融合規則は，プログラマがプログラムの効率化の仕様を簡潔に記述するためのインタフェースと見なせる．従来の並行プログラミングの枠組みでは，融合のよう

☆ バッファオーバフロー/アンダフローをチェックするコードは省略した．このバッファ内の要素は FIFO の順で管理される必要はないと仮定する．さらに，メソッド融合を使わないときに生じるバッファオーバフローがメソッド融合を使ったときにも生じることを保証しなくてよいと仮定する．

☆☆ 融合規則が透明であるかどうかの定義は，プログラムの挙動およびその等しさの定義に依存する．たとえば，4.1 節の例において，複数の repaint の呼び出しの融合によって，ウィンドウのちらつきが減るかもしれない．ちらつきの有無をプログラムの挙動に含めるならば，repaint のための融合規則は，プログラムの挙動を変えているので，透明ではない．含めないならば，透明である．現在我々は「プログラムの挙動」という言葉に厳密な定義を与えていない．今後，厳密な定義を与えていきたい．

な効率化は，各アプリケーションごとに，再利用の難しい adhoc な形で実装されていた．たとえば，4.1 節に示したような GUI イベントのまびきを実現するために，GUI オブジェクトにイベントキューを関係づけ，再描画イベントをまびく処理が Java の処理系[21]に記述されている．そのまびきの実装を他のアプリケーションで再利用することは単純ではない．我々の枠組みでは，コンパイラとランタイムが効率化のための**実装**を隠蔽しており，プログラマは効率化のための**仕様**を記述するだけでよい．上の例でいえば，我々の言語のプログラマは，イベントキューを管理する繁雑なコードを書かずに，数行からなる融合規則を書き，同様の効果を得る．

6. 実　　装

本章ではまず，メソッド融合を含まない実装アルゴリズムを述べ，次に，それを拡張する形で，メソッド融合を含む実装アルゴリズムを述べる．

6.1 データ構造

我々の実装におけるオブジェクトの表現を**図 1** に示す．排他メソッドを持つクラスのオブジェクトにはロック（以降では**オブジェクトロック**と呼ぶ）が関係づけられる．オブジェクトロックは，そのオブジェクトで排他メソッドが実行されているかどうかを示すフラグと，実行待ちタスクのキューの組によって表現される：

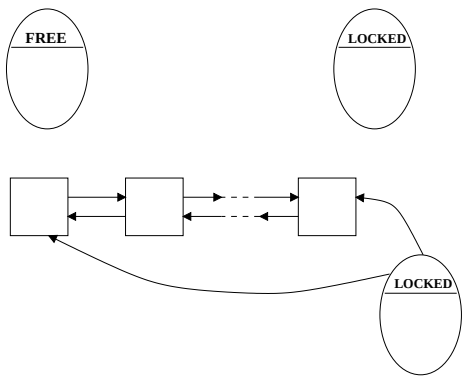

図 1 オブジェクトの表現．左上図はメソッドが実行されていないオブジェクトを示す．右上図はメソッドが実行されている（しかし実行待ちタスクを持たない）オブジェクトを示す．下図はメソッドが実行されており，かつ，実行待ちタスクを持つオブジェクトを示す．

Fig. 1 Representations of objects. The top left figure represents an object on which no thread executes an exclusive method. The top right one and the bottom one represent an object on which a thread executes an exclusive method (the top right object has no task while the bottom one has tasks).

フラグ フラグは FREE と LOCKED の 2 種類の値を持つ．そのオブジェクトに対して排他メソッドが実行されているときにはフラグの値は LOCKED である．そうでないときにはフラグの値は FREE である．

実行待ちタスク 実行待ちタスクは，オブジェクトに対して呼び出されたが，オブジェクトロックの獲得操作の衝突によって実行を遅らされているメソッド呼び出しを表現するデータ構造である．それは，メソッド名とメソッド引数を格納している．以降では適宜，単に**タスク**と呼ぶ．

オブジェクトに関係づけられたタスクのキューを以降では**待ちキュー**と呼ぶ．フラグと待ちキューはスピンロックなどのより低レベルなロックを通じて排他的に操作される．

6.2 メソッド融合を含まない実装アルゴリズム

複数の排他メソッドはオブジェクトロックを用いて排他的に実行される．各スレッドは，オブジェクトに対して排他メソッドを実行する前には，そのオブジェクトのオブジェクトロックを獲得し，排他メソッドを実行した後には，そのオブジェクトのオブジェクトロックを解放する．あるスレッドによって呼び出されたメソッドは，それが融合されない限り，必ずそのスレッド自身がいつかは実行を行い，そのメソッドの実行権が他のスレッドに移ることはない．

オブジェクトロックの獲得を試みるスレッドは以下の処理を行う．まず，フラグを読み，フラグが FREE であったら，フラグを LOCKED に変え，すぐにメソッドを実行する．フラグが LOCKED であったら，そのスレッドは，自分が実行しようとしたメソッドのタスクを作成し，それを待ちキューの末尾に挿入する．挿入されたタスクには同期データ構造が関係づけられている．スレッドは同期データ構造に対して通知を行うことができ，通知を待つことができる．タスクを待ちキューに挿入したスレッドは，そのタスクに関係づけられた同期データ構造からの通知を得た後に，そのタスクを実行する．その通知は，そのタスクが表現するメソッドに実行権が移ったことを意味する．

オブジェクトロックを解放するスレッドは，まず，待ちキューにタスクがあるかどうか調べる．タスクがなければフラグを FREE に変える．タスクがあれば，待ちキューの先頭からタスクを 1 つ取り出し，そのタスクに関係づけられた同期データ構造に通知を行う．

6.3 メソッド融合の実装アルゴリズム

融合処理は，待ちキューにタスクを挿入しようとしているスレッドが挿入の直前に試みる．もし挿入しよ

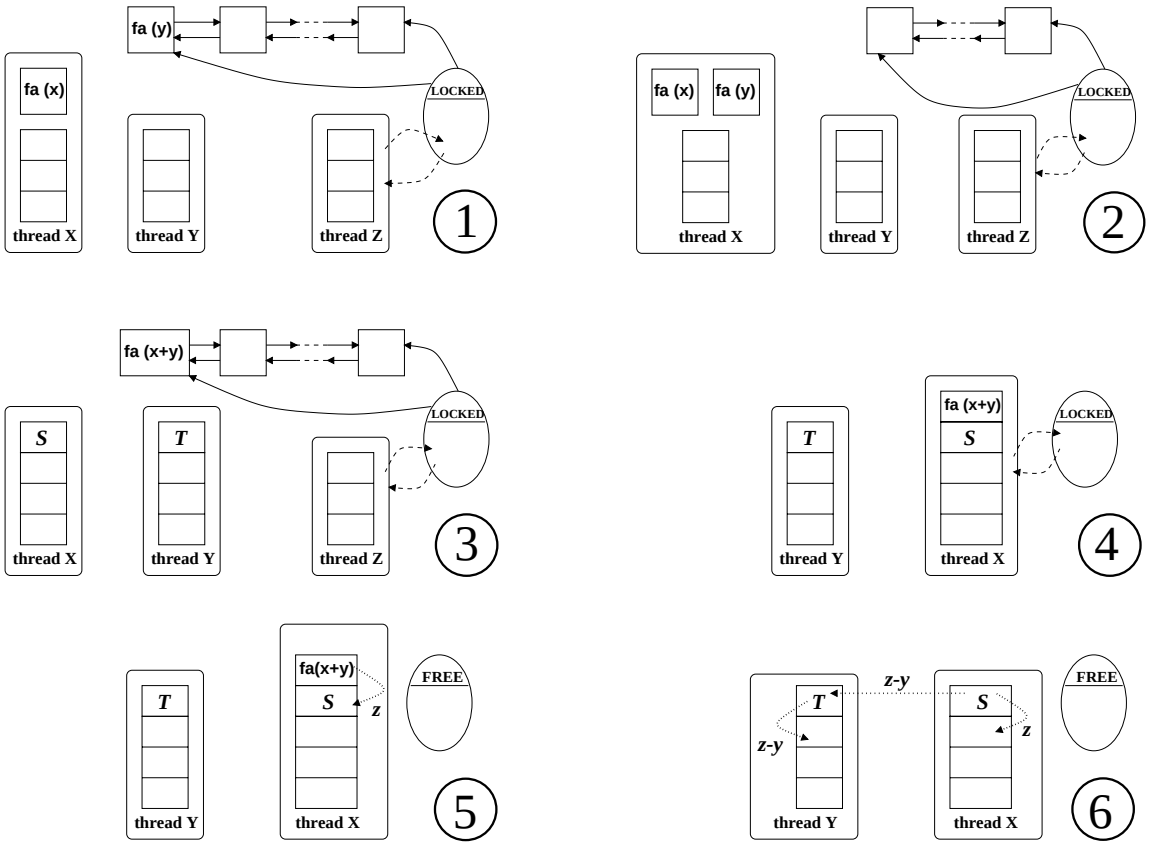

図 2 融合が行われる様子
Fig. 2 How two method invocations are fused.

うとしているタスクと，待ちキューの末尾にあるタスクが融合可能であるならば，末尾にあるタスクを待ちキューから取り出し，それら 2 つのタスクからメソッド呼び出しの情報を読み出し，融合規則の本体を実行する．待ちキュー末尾のタスクは融合可能でない場合には待ちキューから取り出されない．融合される可能性のあるのは，挿入されようとしているタスクと，待ちキュー末尾のタスクの組のみであり，待ちキューの中の末尾以外のタスクは融合の対象とならない☆．

メソッド融合の実装を以下のプログラムを例にとってより具体的に説明する．このプログラムは fetch-and-add 操作を持つカウンタを実装している．

```
class Counter {
  int value;
 public:
  sync int fa(int n) { /* fetch and add */
    int tmp = value;
    value += n;
    return tmp;
  }
  ...
  fusion int fa(int x) & int fa(int y) {
    z = fa(x + y);
    mreturn z and z - y;
  }
}
```

上のプログラムにおいて，タスクの融合処理が行われる様子を**図 2** に示す．図 2 に含まれる各図の解説を以下に記す☆☆．

(1) スレッド Z がオブジェクトに対しメソッドを実

☆ 待ちキュー内のどのタスクを融合の対象とするかについては性能面でのトレードオフが存在する．融合できるかどうかのチェックを待ちキューの末尾のタスクだけではなく待ちキュー内のすべてのタスクに対して行う選択肢もある．その選択肢をとると，融合の回数は増加するが，タスクをチェックするコストが増大する．

☆☆ 図 2 では，1 つのスレッドが 1 つのスタックを持っているが，それは本質的ではない．Cilk [8] や，StackThreads/MP [19] のような，1 つのスタック中に複数のスレッドのフレームを混在させる言語処理系に対しても我々の実装方式は適用できる．

行している．スレッド Y は fa(y) というメソッド呼び出しを表現するタスクを待ちキューに入れ，その呼び出しに実行権が移るのを待っている．スレッド X は fa(x) をオブジェクトに対して呼び出そうとしている．

(2) スレッド X は，検査の結果，呼び出し fa(x) と，待ちキューの末尾にあるタスクによって表現される呼び出し fa(y) が融合可能であることを知り，fa(y) を表現するタスクを待ちキューから取り出す．

(3) スレッド X のスタックには融合規則の本体を実行するためのフレーム S が積まれる．スレッド Y のスタックには，スレッド X から値が送られるのを（融合時に作られた同期データ構造を通じて）待ち，送られた値を親フレームに返すためのフレーム T が積まれる．スレッド X は fa(x + y) を呼び出すが，ロック獲得に失敗し，呼び出し fa(x + y) を表現するタスクを待ちキューに入れる．

(4) スレッド X に実行権が移る．

(5) スレッド X は fa(x + y) の返り値をフレーム S に返す．

(6) スレッド X は mreturn 文を実行する．mreturn 文の片方の返り値を親フレームに返し，もう片方の返り値を，同期データ構造を通じてスレッド Y に送る．スレッド Y はフレーム T においてその返り値を受け取り，それを（あたかも自分が fa(y) を実行したかのように）親フレームに返す．void 型のメソッド呼び出しの融合の場合でも，スレッド間で同期する必要があるため，同期データ構造を通じてダミーな値の送受信が同様な形で行われる．

6.4 議　　論

我々の現在の実装では，タスクを挿入しようとするスレッドは，そのタスクと待ちキュー末尾のタスクが融合可能であるならば，**必ず**融合する．たとえば，6.3 節のプログラムで fa メソッドのみが多数呼び出された場合，待ちキューの長さはたかだか 1 にしかならない．待ちキューに挿入されようとする fa メソッドのタスクは，待ちキュー内にただ 1 つ存在する fa メソッドのタスクと必ず融合されるからである．

我々の実装では，各スレッドは，スレッド生成時に割り当てられた，タスク用の記憶領域と同期データ構造用の記憶領域からタスクや同期データ構造を確保する．タスクや同期データ構造は 1 スレッドあたり同時にたかだか 1 つしか必要でないため，タスクや同期データ構造のための領域を malloc や new を使って動的に確保する必要はない．

融合による性能向上の源には，以下の 2 つがある．

- 計算の量が減る．典型的にはメソッド実行の回数が減る．
- 排他メソッドの実行の前後におけるオブジェクトロックの操作の回数が減る．すなわち排他区間が粗粒度になる．

我々の現在の実装では，後者による性能向上は期待できない．なぜなら，融合処理では，融合対象のタスクをキューの末尾から取り出すのにオブジェクトロックの獲得と同程度のコストがかかり，加えて，返り値をスレッド間で通信するコストがかかるためである．メソッド融合が性能を向上させるには，融合前に行われるはずだった計算と，融合後に行われる計算の量の差が，上に述べたコストの増加を打ち消すほど大きいことが必要である．よって，メソッド融合が効果的である場面は，排他メソッドの本体で入出力や GUI 処理などの大きい処理を行い，かつ，融合によってその排他メソッドの実行回数が減る場面である．

7. 実 験 結 果

対象言語のコンパイラを実装し，性能測定を行った．コンパイラは The EDG C++ Front End [5] を改造して実装された．使用した計算機は Sun Enterprise 10000（UltraSPARC 250 MHz × 64，Solaris 2.7）である．

実験には以下のアプリケーションを用いた．プログラム中で作られる各スレッドは並列計算機上の各プロセッサの上で並列に走る．本実験に関する記述では，スレッドとプロセッサを同一視してよい．

Counter 各スレッドが，カウンタオブジェクトに対し，カウンタの値に 1 を足す排他メソッドのみを並行に繰り返し呼び出すプログラム．このプログラムは 2 章のコードとほぼ同じコードを含む．

FileWriter 各スレッドが，ファイル記述子を含むオブジェクトに対し，ファイル出力のための排他メソッドのみを並行に繰り返し呼び出すプログラム．その排他メソッドは，引数の文字列をその記述子が示すファイルに（バッファリングなしに）出力する．その排他メソッドの複数の呼び出しを融合する融合規則を記述した．その融合規則は，排他メソッドの引数の 2 つの文字列を結合した文字列を作り，その新しい文字列を引数として排他メソッドを呼び出す．

FileReader 各スレッドが，ディスクを抽象化した

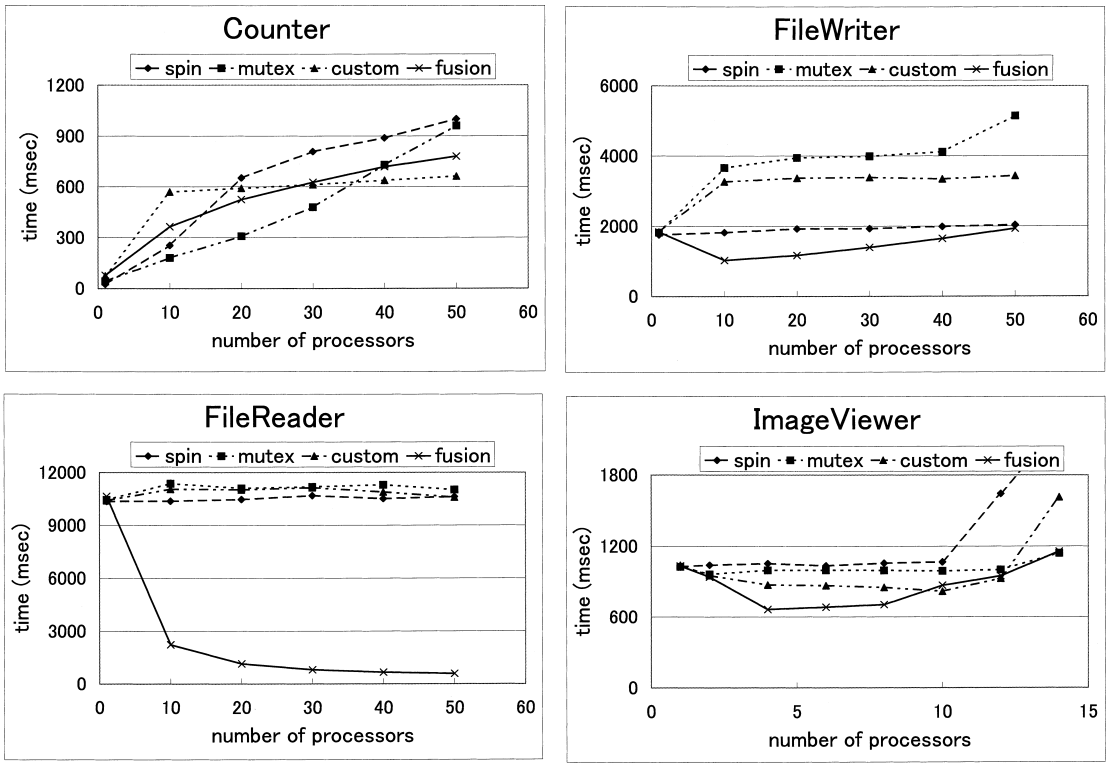

図 3 実行時間
Fig. 3 Execution times.

オブジェクトに対し，ファイルのデータを読み出す排他メソッドのみを並行に繰り返し呼び出すプログラム．その排他メソッドは，引数に与えられたパスにあるファイルをオープンし，その内容を別の引数に与えられた文字列配列に格納する．その排他メソッドの引数に与えられるパスはつねに同じである．その排他メソッドの，パスが同一である複数の呼び出しを融合する融合規則を記述した．

ImageViewer 画像ファイルを読み，その画像を 1 ピクセルずつ徐々に表示するプログラム．GUI ツールキット GTK+ [10] を用いて書かれた．ピクセル集合を各スレッドで等分する．各スレッドは画像に 1 つピクセルを表示するたびに，画像のそのピクセルが含まれる「行」を再描画する．再描画は画像オブジェクトに対する排他メソッドの呼び出しによって行われる☆．その排他メソッドの複数の呼び出しを融合する融合規則を記述した．

すべてのアプリケーションにおいて，プログラムの実行全体を通じて排他メソッドが呼び出される回数は，スレッドの数にかかわらず一定である．上の 4 つのアプリケーションで，排他メソッドの 1 回の実行時間は，排他メソッドの呼び出しだけを実行するプログラムの実行時間を，排他メソッドの実行回数で割ったものにほぼ等しい．その方法で求めた，排他メソッドの 1 回の実行時間は，Conter，FileWriter，FileReader，ImageViewer でそれぞれ 0.25，17，1035，137 マイクロ秒である．

図 3 は実行時間を示す．排他メソッドを実装するのに，spin はスピンロックを使用し，mutex は OS が提供する mutex ロックを使用し，custom は 6 章で述べたロックで融合機構を含まないものを使用し，fusion は 6 章で述べたロックで融合機構を含むものを使用した．ImageViewer では GUI ライブラリの使用の都合上，14 プロセッサ構成の Sun Enterprise 10000 を使用した．

図 4 は排他メソッドが実行された回数を示す．normal はメソッド融合を使わないプログラムでの回数を示し，fusion はメソッド融合を使うプログラムでの回

☆ GTK+は thread-safe ではないため，GTK+のライブラリ関数は必ず排他メソッド内で呼び出さなければならない．GTK+は，GTK+を thread-safe にするための機構を用意しているが，それは GTK+のライブラリ関数の呼び出しを逐次化するだけの単純なものである．

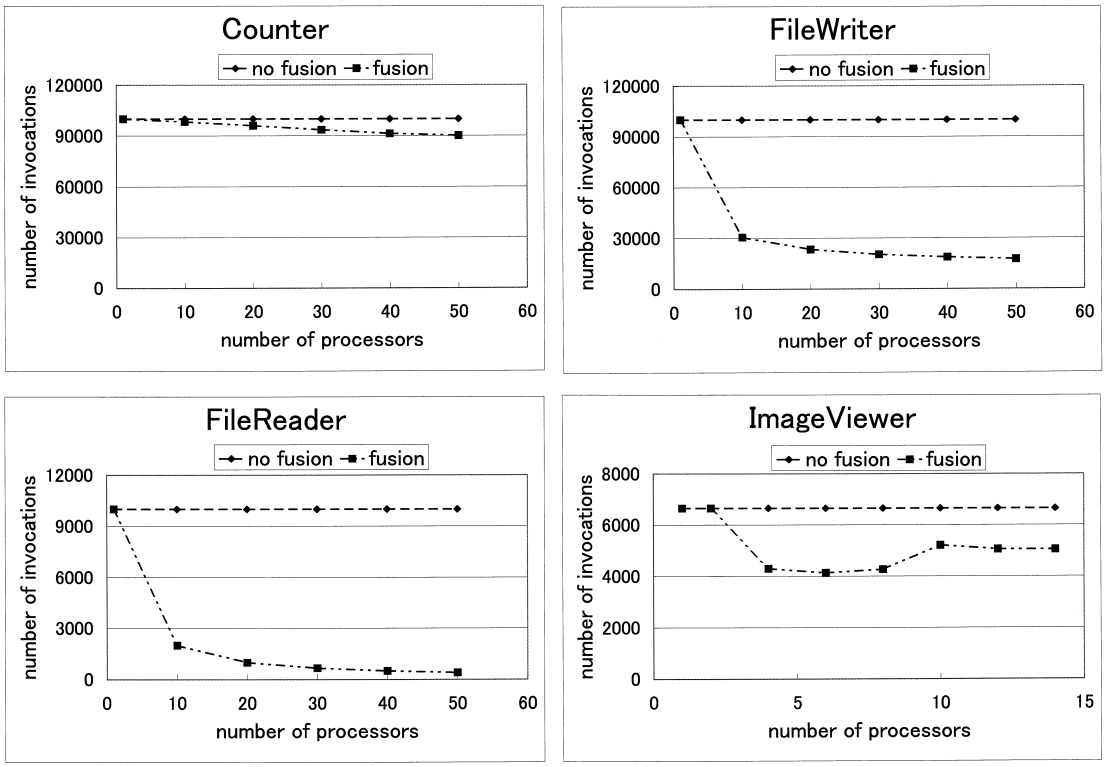

図 4 排他メソッドの呼び出し回数
Fig. 4 Number of invocations of exclusive methods.

数を示している．

一般に，実験で使用したような，並列なアクセスが集中するオブジェクトを持つアプリケーションでは，スレッド数の増加につれて実行時間が増加することがある[16),24)]．その現象は上のすべてのアプリケーションで見られた．

FileWriter，FileReader，ImageViewer のすべてにおいて，メソッド融合を行うプログラムが，ほとんどのプロセッサ数上で，最も高い性能を示した．Counter では特に優位な方式はなかった．FileWriter では，メソッド融合を行うプログラムの実行時間が最も小さかったものの，その実行時間は，プロセッサ数の増加につれて，一度下がった後，着実に上がっていった．FileReader は排他メソッドの実行がファイルをオープンする重い処理を含むため，メソッド融合による性能向上比が最も大きかった．ImageViewer では，プロセッサが 8 台以下の場合にはfusion が最も高い性能を示した．ImageViewer では，プロセッサ数の増加とともに，fusion ばかりでなくcustom の実行時間も減少したが，その原因は分かっていない．

Counter では排他メソッドの実行回数がメソッド融合によって少ししか減っていない．FileWriter ではスレッド数が増えるにつれて，排他メソッドの実行回数は実行時間と異なり着実に減っている．つまり FileWriter におけるfusion の実行時間の漸増は，融合回数の減少が原因ではなく，何らかのオーバヘッドの増加が原因であると考えられる．FileReader と ImageViewer では，実行時間の曲線と排他メソッドの実行回数の曲線は似た形を示した．

8. 関 連 研 究

8.1 結合的な排他的操作の並行実行

操作の結合性を利用して排他的操作を並行実行する方法が多数提案されている．それらの方法は，結合性を持つ排他的操作を並行に実行し，部分的な実行結果を，オブジェクトの複製などの別々の記憶領域に蓄積する．部分的な実行結果は後で 1 つにまとめられる．この方法を用いているものに，Chien による Concurrent Aggregates（CA）[3)]，Rinard らによる adaptive object replication [18)]，ループ並列性を利用する言語における reduction [6),11),15)] がある．Hu らの研究[12)]は再帰関数の呼び出しによる繰返しを reduction の方法で実行するためのプログラム解析とプログラム変換を形式的かつ一般的な形で提案している．

上の方法は，部分的な実行結果をまとめるべきタイミングが明確に定まっている regular な実行モデルにのみ適用でき，そのタイミングをどこまで遅らせてよいか自明ではない irregular な実行モデルには適用できない．そのタイミングは，adaptive object replication では parallel phase の末尾，reduction ではループの末尾と定まっている．CA ではそのタイミングをプログラマが決定する．メソッド融合は，irregular な実行モデルにも適用できる．

加えて，上の方法では，実行するメソッドの回数や種類を変化させる処理を簡潔には記述できない．そのような処理を行うプログラムの例は，7 章の実験の FileReader と ImageViewer に見られる．

上の方法とメソッド融合は補完的であり，片方によって他方が無意味になることはない．上の方法が適したプログラムと，メソッド融合が適したプログラムの両方が存在する．特に，上の方法が適用できる場合のほとんどでは，メソッド融合よりも上の方法の方が，高速な実行を与える．上の方法とメソッド融合の両方を用いれば，互いの短所を補いあえる．

8.2 干渉しあわない複数の排他メソッドの並行実行

いくつかの言語では，並行に実行しても，それらを逐次化して実行したときの意味が保たれる排他的なメソッドの組をプログラム解析やプログラマの助けによって得ることにより，複数の排他的なメソッドを並行に実行する．典型的には，同じメンバ変数を更新しない複数の排他メソッドを並行に実行する．その例として，ICC++[4]，Schematic[20]，OPA[23] がある．また，reader-writer ロックも，同様の着想に基づく機構である．これらの機構は，並行に実行できないメソッドの組が並行に呼び出される場合には無力である．これらは，並行に実行できないものを効率化するメソッド融合とは直交しており，メソッド融合と組み合わせて相補的に用いるべきものである．

8.3 Network Combining

NYU Ultracomputer の研究[9] において，Network Combining と呼ばれる方法が提案されている．この方法は，並列計算機内部のネットワークに流された複数の命令を，ネットワークのスイッチが融合する．たとえば，ネットワークを流れてきた 2 つの fetch-and-add 命令を融合し，1 つの fetch-and-add 命令のみをその先のネットワークに流す．彼らは**命令の単位で**プログラムの逐次実行の意味を維持しながら複数の処理を融合するが，我々はオブジェクト指向言語の**メソッドの単位で**プログラム全体の挙動を維持しながら複数の処理を融合する．

8.4 プログラム変換による複数の処理の融合

複数の操作を静的に融合するプログラム変換が提案されている．関数型言語の研究では，Bird-Meertens Formalism[1]，Wadler による Deforestation Algorithm[22]，HYLO システムで使われているプログラム融合変換[14] がある．Imperative な言語の研究では，伝統的な最適化技法の loop fusion，配列の scan 操作の並列化における合成関数の利用[2),6)] がある．これらの方法は，**静的に検出された**複数の操作を 1 つの操作に**静的に**変換するが，我々の方法は，**動的な制御フロー内に出現した**複数の操作の実行を 1 つの操作の実行に**動的に**切り替える．静的な融合は，メソッド融合と異なり，融合のための実行時オーバヘッドをともなわない．よって，静的な融合が適用できる場面では，静的な融合を用いるのが有利である．一方，メソッド融合は，実行時オーバヘッドをともなう反面，静的な融合が適用できない場面でも使える利点を持つ．2 つの方法を補完的に用いればきわめて大きい高速化が得られる．

8.5 Join-Calculus

Join-calculus[7] は，複数の関数呼び出しを 1 つのコード片の実行に置換することの繰返しによって，プログラムの実行を表現する．置換の意味は本論文の融合処理の意味に似ており，置換の仕様を記述するための API（join pattern）も，融合規則のそれにきわめて似ている．彼らの研究は，理論的側面に重点を置いており，join pattern を性能向上のために使う視点を持たない．本論文は，逐次化された複数のメソッド実行を効率化するためのヒントとして join pattern に似た API を用いる斬新なアイデアを，最初に示した．

8.6 我々の以前の研究

我々の以前の研究[16),24)] も，複数の排他メソッドの呼び出しが頻繁に逐次化される並列プログラムを効率的に実行する方法を示している．しかし，その方法は，排他処理そのものの効率化と，排他メソッドを実行するプロセッサをできる限り変えないことによるメモリ効率の向上に主眼を置いており，実行するメソッドそのものを変えることはしない．その方法とメソッド融合は互いに補完しあう関係にある．

9. 結論と今後の課題

9.1 結　　論

メソッド融合の機構を持つ言語の設計とその実装方式を示した．メソッド融合は，異なるスレッドにまたがる動的な制御フロー内に出現した連続する排他区間の処理を融合する．それを行う枠組みは，我々の知る

限りこれまでに存在せず，既存研究の盲点であった．メソッド融合が特に効果的であるのは，GUI の再描画のような，2 回の実行を 1 回の実行で代用できる処理と，バッファへの put と get のような，打ち消し合う 2 つの処理においてである．実験を通じてメソッド融合の有効性を確認した．排他メソッド内で入出力などの重い処理をするプログラムでは大きな性能向上が見られた．

より広い視点で捉えると，メソッド融合は，並行に実行できる部分を排他区間の外に出すことによって，2 つの連続する排他区間の実行を，より短い排他区間と非排他区間の組の実行に置き換えることを可能にする．4.1 節の repaint の例は，非排他区間が残らない特殊な場合であり，4.2 節のバッファの例は，排他区間が残らない特殊な場合である．

本研究を含む我々の大局的な目標は，様々な効率化を高級言語の処理系に行わせることによって，快適な並列プログラミング環境を提供することである．我々は，その目標のためには高級言語はどんなインタフェースをプログラマに提供するのがよいかという問題に取り組み，その 1 つの解答として，融合規則を提案した．融合規則は，プログラマが処理系の効率化を助けるための先進的なインタフェースの 1 つである．

9.2 今後の課題

第 1 に，プログラマが透明な融合規則を記述することを支援する枠組みを開発していきたい．現状では，

```
fusion void inc(int x) & void inc(int y) {
  inc(x - y);
}
```

のような透明でない融合規則もコンパイラはエラーを出さずに受け入れる．融合規則の透明性の検査には，symbolic execution[13] に基づく commutativity analysis[17] の技法が利用できる．commutativity analysis は，呼び出す順番を逆にしてもプログラムの最終結果が変わらない 2 つの連続したメソッド呼び出しを検出する．彼らの方法を拡張すれば，算術オペレータの結合性を利用した単純な融合規則の透明性は保証できる．たとえば，`inc(x)` と `inc(y)` を連続して実行しても，`inc(x + y)` を実行しても，プログラムの最終結果が変わらないことをコンパイラが判断できる．

第 2 に，継承を含む言語仕様を作りたい．継承の導入は複雑な問題をともなう．

```
class C {
  ...
  void p(void) { ... }
  void q(void) { ... }
  void r(void) { ... }
  fusion void p(void) & void q(void) {
    r();
  }
}
```

というクラスを考える．`C` を継承して子クラス `D` を作ったとする．上の融合規則はクラス `D` のオブジェクトに対しても適用されるべきか？ `p` や `r` が override されたらどうか？

最後に，現在の実装では，同時にたかだか 1 つのスレッドしか待ちキューを操作できないが，待ちキューを並列に操作できるように実装を改良していきたい．

謝辞 京都大学の八杉昌宏講師，東京大学の小林直樹講師，東京工業大学の脇田建講師，東京大学の胡振江助教授，査読者の方々から有益なコメントをいただきました．つくばメタプログラミング研究会のメンバおよび東京大学の米澤研究室のメンバとの議論が本論文の改善に非常に役立ちました．

参 考 文 献

1) Bird, R.S.: Algebraic Identities for Program Calculation, *The Computer Journal*, Vol.32, No.2, pp.122–126 (1989).
2) Callahan, D.: Recognizing and Parallelizing Bounded Recurrences, *Proc. 4th International Workshop on Languages and Compilers for Parallel Computing* (*LCPC '91*), Lecture Notes in Computer Science, Vol.589, pp.169–185, Springer-Verlag (1991).
3) Chien, A.A.: *Concurrent Aggregates* (*CA*), The MIT Press (1991).
4) Chien, A.A., Reddy, U., Plevyak, J. and Dolby, J.: ICC++ – A C++ Dialect for High Performance Parallel Computing, *Proc. 2nd JSSST International Symposium on Object Technologies for Advanced Software* (*ISOTAS '96*), Lecture Notes in Computer Science, Vol.1049, pp.76–95, Springer-Verlag (1996).
5) Edison Design Group: The EDG C++ Front End, http://www.edg.com/.
6) Fisher, A.L. and Ghuloum, A.M.: Parallelizing Complex Scans and Reductions, *Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation* (*PLDI '94*), pp.135–146 (1994).
7) Fournet, C. and Gonthier, G.: The reflexive CHAM and the join-calculus, *Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '96*), pp.372–385 (1996).

8) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (*PLDI '98*), pp.212–223 (1998).
9) Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L. and Snir, M.: The NYU Ultracomputer–Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. Comput.*, Vol.32, No.2, pp.175–189 (1983).
10) GTK+ Home Page: http://www.gtk.org/.
11) Hall, M.W., Amarasinghe, S.P., Murphy, B.R., Liao, S.-W. and Lam, M.S.: Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler, *Proc. Supercomputing '95* (1995).
12) Hu, Z., Takeichi, M. and Chin, W.-N.: Parallelization in Calculational Forms, *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL '98*), pp.316–328 (1998).
13) King, J.C.: Symbolic Execution and Program Testing, *Comm. ACM*, Vol.19, No.7, pp.385–394 (1976).
14) Onoue, Y., Hu, Z., Iwasaki, H. and Takeichi, M.: A Calculational Fusion System HYLO, *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, pp.76–106, Chapman&Hall (1997).
15) OpenMP Architecture Review Board: *OpenMP C and C++ Application Program Interface* (1998).
16) Oyama, Y., Taura, K. and Yonezawa, A.: Executing Parallel Programs with Synchronization Bottlenecks Efficiently, *Proc. International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications* (*PDSIA '99*), pp.182–204, World Scientific (1999).
17) Rinard, M.C. and Diniz, P.C.: Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers, *ACM Trans. Programming Languages and Systems*, Vol.19, No.6, pp.942–991 (1997).
18) Rinard, M.C. and Diniz, P.C.: Eliminating Synchronization Bottlenecks in Object-Based Programs Using Adaptive Replication, *Proc. 1999 ACM International Conference on Supercomputing* (*ICS '99*), pp.83–92 (1999).
19) Taura, K., Tabata, K. and Yonezawa, A.: StackThreads/MP: Integrating Futures into Calling Standards, *Proc. 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (*PPoPP '99*), pp.60–71 (1999).
20) Taura, K. and Yonezawa, A.: Schematic: A Concurrent Object-Oriented Extension to Scheme, *Proc. Workshop on Object-Based Parallel and Distributed Computation* (*OBPDC '95*), Lecture Notes in Computer Science, Vol.1107, pp.59–82, Springer-Verlag (1996).
21) Transvirtual Technologies Inc.: Kaffe Home Page, http://www.transvirtual.com/.
22) Wadler, P.: Deforestation: Transforming programs to eliminate trees, *Theoretical Computer Science*, Vol.73, No.2, pp.231–248 (1990).
23) Yasugi, M., Eguchi, S. and Taki, K.: Eliminating Bottlenecks on Parallel Systems using Adaptive Objects, *Proc. International Conference on Parallel Architectures and Compilation Techniques* (*PACT '98*), pp.80–87 (1998).
24) 大山恵弘，田浦健次朗，米澤明憲：同期ボトルネックが存在する並列プログラムの効率的実行，情報処理学会論文誌，Vol.41, No.5, pp.1448–1458 (2000).

付　　録

A.1 プログラム例

A.1.1 Aggregate の操作

ベクタを表現するオブジェクトのクラス `Vector` を考える．そのクラスは，ベクタ内のすべての値を，引数に与えられた数で乗ずる排他メソッド `mul` を持つとする．融合規則によって，ベクタを a 倍する `mul` メソッドの呼び出しと，ベクタを b 倍する `mul` メソッドの呼び出しを，ベクタを $a*b$ 倍する `mul` メソッドの呼び出しに融合できる☆．この融合は，伝統的な最適化技法の loop fusion にきわめて似た処理である．

A.1.2 最大値の計算

最大値を保持するオブジェクトのクラス `MaxNum` を考える．そのクラスの排他メソッド `setmax` は，保持されている最大値よりも引数の値が大きければ最大値をその引数で更新する．以下の融合規則によって，2 つの `setmax` メソッドの呼び出しを 1 つの呼び出しに「まびき」できる．

```
fusion void setmax(int a)
     & void setmax(int b) {
  if (a > b)  setmax(a);
  else        setmax(b);
}
```

☆ 浮動小数演算の順序変更により生じる計算誤差は考えない．

A.1.3 メモリ領域確保操作

ヒープを表現するオブジェクトのクラスを考える．そのクラスのメソッド `alloc` は，引数に与えられたサイズのメモリ領域をヒープから確保し，その領域へのポインタを返す．以下の融合規則によって，`alloc` メソッドの 2 つの呼び出しを 1 つの呼び出しに変えられる．

```
fusion int *alloc(size_t s1)
     & int *alloc(size_t s2) {
  int *p = alloc(s1 + s2);
  mreturn p and (p + s1);
}
```

上の種の融合規則の記述には注意が必要である．いくつかのヒープでは，確保されたメモリ領域は明示的な関数呼び出しによって解放される．たとえば C 言語では，malloc 関数で確保された領域は free 関数で解放される．そのようなヒープを使ったプログラムでこのような融合規則を書くと，プログラムが正しく動かなくなる．C 言語の例でいえば，malloc 関数によって確保された領域の内部を指すポインタに対して free 関数が呼び出されるからである．プログラムを正しく動かすためには，融合規則の中で，malloc 関数によって確保された領域に参照カウンタを付加し，メモリの確保と解放を行うメソッドの中で，その参照カウンタを管理するなどの工夫が必要である．

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

大山 恵弘（学生会員）

1973 年生．1996 年東京大学理学部情報科学科卒業．1998 年東京大学大学院理学系研究科情報科学専攻修士課程修了．現在同大学院博士課程在学中．主に並列プログラミング言語の研究に従事．

田浦健次朗（正会員）

1969 年生．1996 年より東京大学大学院理学系研究科助手．1997 年東京大学大学院より理学博士取得．並列プログラミング言語の設計，実装に関する研究に従事．

米澤 明憲（正会員）

1947 年生．1977 年 Ph.D. in Computer Science（MIT）．1989 年より東京大学大学院理学系研究科情報科学専攻教授．並列・分散モバイルソフトウエア等に興味を持つ．ドイツ国立情報処理研究所（GMD）科学顧問，ACM TOPLAS 副編集長，IEEE Parallel & Distributed Technology および Computer 編集委員等を歴任，元日本ソフトウェア科学会理事長．ACM Fellow．