ソフトウェアDSM機構を支援する最適化コンパイラ

丹羽純平[†,††] 松本尚[†,†††] 平木敬[†]

ソフトウェアDSMは低い実装コストで，実行時に共有アドレス空間を提供できるため，幅広いアプリケーションを扱うことが可能である．ただし，高性能を得るためには，アプリケーションプログラムのソースを直接解析する最適化コンパイラの支援が必要である．最適化コンパイラの目的は，コヒーレンス管理にともなう通信/命令のオーバヘッドを削減することにある．その実現のために，手続き間ポインタ解析を行い，緩和されたメモリモデルの上で，手続き間区間解析の枠組で冗長性削除のデータフロー方程式を解くことで，共有メモリアクセスのサマリを算出する．さらに，fetch-on-writeを抑制するコード生成を行う．我々は上記の最適化コンパイラ（RCOP）を作成して，ワークステーションクラスタ上に低オーバヘッドなランタイム実装した．SPLASH-2ベンチマークを用いた実験により，コンパイラの最適化の効果を確かめた．台数効果を得るのが困難とされているアプリケーション，FFTとRadixに対して，高い台数効果を得ることができた（16台で逐次の約4倍）．

An Optimizing Compiler to Support Software DSM Systems

Junpei Niwa,[†,††] Takashi Matsumoto[†,†††] and Kei Hiraki[†]

Software Distributed Shared Memory (DSM) provides shared address space at run-time and accepts a wide range of applications, and it is easy to implement on the existing systems with commodity hardware. An optimizing compiler that directly analyses shared-memory source programs is indispensable for improving the performance of software DSM systems. The purpose of compiler optimization is to generate codes reducing the communication and instruction overheads for software cache-coherence management. The optimizing compiler performs interprocedural points-to analysis and interprocedural shared-access set calculations by using interval analysis to solve redundancy elimination equations along with lazy release consistency model. It generates the codes that avoid fetch-on-write. We have implemented this optimizing compiler, Remote Communication Optimizer: RCOP. We also have implemented the lightweight runtime systems on an SS20 workstation cluster connected with the Fast Ethernet (100BASE-TX). The experimental results using the SPLASH-2 benchmark suite show that the combination of the optimizing compiler and software DSM is very effective. Our proposed system had high speed-up ratios for the FFT and Radix programs, that is, challenging applications for software DSM (about 4 times faster than sequential programs on 16 processors).

1. はじめに

本研究の目的は，共有メモリを支援するハードウェアのないNUMA（Non-Uniform Memory Access）環境上であっても，共有メモリモデルに基づいて書かれた明示的に並列なプログラムを効率良く実行することにある．そのためには，ソフトウェア分散共有メモリ（Distributed Shared Memory: DSM）[15),16)]と呼ばれる，遠隔ノードのデータをアクセスしたプロセッサの近くにキャッシュする機構が求められる．

従来のソフトウェアDSMでは，逐次コンパイラによるアプリケーションのコード生成が大前提になっている．すなわち，コヒーレンス管理操作（複数のノード間で共有データの内容に矛盾が生じないような制御）を必要とする共有メモリアクセスは局所メモリアクセスと同様にプロセッサのload/storeで実現される．コヒーレンス管理操作を必要とする共有メモリアクセスはトラップによって検知され，コヒーレンス管理操作はトラップハンドラ内でOSにより実行される．したがって，コヒーレンス管理操作はアプリケーションからは見えなくなり，アプリケーションからコヒーレンス管理操作に最適化をかけることができない．また，

† 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
††† 科学技術振興事業団さきがけ研究21「情報と知」領域
PRESTO, Japan Science and Technology Corporation

コヒーレンス管理操作自体もページ全体のコピーや比較という高オーバヘッドをともなうものである．

本稿では，高性能なソフトウェア DSM 機構を実現する下記ユーザレベル最適化技術を提案し，実装と実験によりその有効性を評価する．

- コンパイル時最適化
  最適化コンパイラが明示的に並列に書かれたプログラムのソースを解析して，共有データアクセスを検出する．次に，検出されたデータアクセスに対し，緩和されたメモリモデルのもとで，コヒーレンス管理操作を行うコード（コヒーレンス管理コードと呼ぶ）をできるだけ大きな粒度で明示的に埋め込むコード生成を行う．ただし，手続き間区間解析や，冗長性削除といったフレームワークを活用して，fusion，coalescing，redundant index elimination といった最適化を統一的に行い，共有アクセスのサマリを計算するところが既存の研究 30) とは異なる．
- 実行時最適化
  緩和されたメモリモデルを低オーバヘッドで実現するプロトコルの下で，ユーザレベルのランタイムがコヒーレンス管理コードを効率良く実行する．通信が必要な場合には，ブロック転送や細粒度通信のコンバイニングを活用する．低オーバヘッドで書き込みを反映する方式を採用しており，論理タイムスタンプのベクトル vector timestamp [15] を用いて，書き込み間の半順序を管理する既存の方式 30) とは異なる．

2. 最適化対象のソフトウェア DSM 機構

Asymmetric DSM [28]

ユーザレベルの最適化を可能にするインタフェースの 1 つが Asymmetric DSM（ADSM）である．ADSM では，既存の OS ベースのシステムと同様に（ソフトウェア）キャッシュミス，ヒット判定にページ管理機構を使用する．書き込み時のコヒーレンス管理操作は，store 命令とは分離して，ユーザレベルのコードとして最適化コンパイラが自動的に挿入し，緩和されたメモリモデルを活用して最適化を行う．コヒーレンス単位（ソフトウェアキャッシュのブロック）はページであるために，false sharing と不必要なデータ通信が問題である．

User-level DSM [17]

もう 1 つのインタフェースである User-level DSM（UDSM）では，ノード間の通信オーバヘッドを最小にするために，コヒーレンス単位（ソフトウェアキャッシュのブロック）をユーザ定義のセグメントにしている．したがって，キャッシュミス，ヒット判定ならびに書き込み時のコヒーレンス管理操作をすべて，ユーザレベルのコードで実現する．それらは対応する load/store 命令とは分離され，最適化コンパイラが自動的に挿入し，ADSM の場合と同様に最適化される．コヒーレンス管理コードのオーバヘッドをいかに押さえるかが問題である．

コヒーレンス管理コード

コヒーレンスを維持するため最適化コンパイラは 2 種類のユーザレベルのコヒーレンス管理コードを挿入し，それらはランタイムによって実行される．

- 読み出しの発行（read commitment）：
  `R(address, size)`
  読み出しのコヒーレンス管理操作を実行するコードであり，（ソフトウェア）キャッシュのミス/ヒット判定を実行し，ミス時にはハンドラを起動するルーチンであり，UDSM において，読み出しの前に挿入される．ADSM ではこの操作はページ管理機構を利用して非明示的に実現される．
- 書き込みの発行（write commitment）：
  `W(address, size)`
  書き込みのコヒーレンス管理操作を実行するコードであり，ADSM と UDSM の両機構において，書き込みの後に挿入される．他ノードへの無効化情報にこのブロックを加える処理等を行う．

引数は，共有アクセスが行われる可能性のある開始アドレス（`address`）とデータサイズ（`size`）からなり，コンパイラが「連続した領域 $\{x \mid \texttt{address} \leq x < \texttt{address} + \texttt{size}\}$ に対してアクセスがある」ことをランタイムに通知することになる．ランタイムは実行時にこの領域が共有領域かどうかを確かめて，共有領域であった場合には上述のコヒーレンス管理操作を行う☆．

3. コンパイル時最適化

3.1 最適化技法の概要

本節で述べるコンパイル時の最適化の目的は，キャッシュ管理にともなう通信と命令のオーバヘッドを削減するコードを生成することにある．本アプローチはソースコードを直接解析することで，ループ，手続き呼び出しといったプログラムの階層構造を活用する．

☆ コヒーレンス管理コードは，共有メモリアクセスになる可能性のあるすべてのメモリアクセスに対して静的に挿入されるので，実際には，着目したメモリアクセスが非共有領域に対してなされる場合もある．

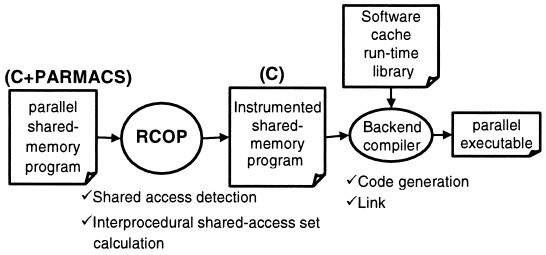


図 1 全体のコンパイルプロセス
Fig. 1 Overall compilation process.

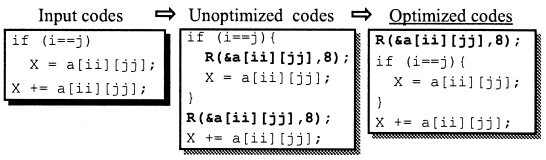


図 2 冗長性削除の例
Fig. 2 Example of redundancy elimination.

まず，共有アクセスを漏れなく，かつ，できるだけ正確に検知する．本アプローチでは，コンパイラは並列プロセスの問題を解くのではなく，並列プログラムを構成する各逐次スレッドの振る舞いを解析する．そして，緩和されたメモリモデルを活用して，共有アクセスをできるだけ大きな粒度で発行する．

本アプローチの有効性を検証するために，Remote Communication Optimizer（RCOP）[12),18),22),30)] という最適化コンパイラを作成した．入力は，Lazy Release Consistency（LRC）モデル[15)] に基づいて書かれた，明示的に並列な共有メモリプログラムで，API は PARMACS[2)] という並列化マクロ☆で C を拡張したものである．RCOP は手続き間別名解析を行い，区間解析，冗長性削除のフレームワークを活用し，fusion，coalescing☆☆，redundant index elimination によるループレベルの最適化を行い，アクセスパターンによっては，fetch-on-write の除去を行い，共有アクセスのサマリを手続き間で計算する．

図 1 を用いて全体のコンパイルプロセスを説明する．RCOP は C+PARMACS のプログラムを直接解析し，スカラーデータフロー解析，手続き間別名解析，手続き間共有アクセス集合計算を行い，コヒーレンス管理コードを明示的に含む C プログラムを出力する．出力された C プログラムをプラットフォーム上の逐次コンパイラにかけ，4 章で述べる実行時最適化を含むキャッシュのコヒーレンス管理のライブラリとリンクして実行コードを生成する．

3.2 共有アクセスの検知

各ノードの論理アドレスは共有領域と非共有領域に分割され，共有領域では全ノードで同一アドレス上に同一データが配置されると仮定する．入力プログラムは C プログラムなので，共有アドレスがポインタ変数に代入されたり，手続き間で使用される可能性がある．RCOP はすべての共有アクセスを検知するために，手続き間別名解析[8),24)] を用いる．手続き間別名解析には正確なポインタ情報が得られるというメリットがある．この情報は後続の最適化におけるコード移動の際に必要であり，共有でないアクセスにキャッシュ管理コードが挿入されることにより，後続の最適化のコストや実行時のオーベヘッドを増大させるのを防ぐのに必要である．詳細は文献 8)，24) を参照されたい．

手続き間別名解析の結果から，`G_MALLOC` オペレーション☆☆☆の返り値を推移的に指す可能性のあるポインタを検出する．ターゲットが ADSM の場合，上記ポインタを使用した書き込みの後に書き込みのコヒーレンス管理コード（write commitment）が挿入される．ターゲットが UDSM の場合には，上記ポインタを使用した読み出しの前に読み出しのコヒーレンス管理コード（read commitment）が挿入され，書き込みの後に書き込みのコヒーレンス管理コード（write commitment）が挿入される．ここで挿入されたユーザレベルのコヒーレンス管理コードは後続のパスで最適化される．

3.3 Read Commitment を用いた最適化

LRC モデルを活用し，共有アクセス集合を使用する最適化技法を述べる．以下の議論は UDSM における read commitment（`R`）に関して進める．

3.3.1 冗長な Read Commitment の削除

LRC モデルでは，他ノードの書き込みの結果は自分が *acquire* するとき☆4にのみ伝播される．したがって，acquire a_1 の実行後，かつ，acquire a_2 の実行前にアクセスされるすべての共有データは acquire a_1 の直後にフェッチしてきてもかまわない．なぜならば，フェッチしてきてから，実際に使用するまで，無効化されることはないからである．つまり，対応する共有読み出しを支配していて直前の同期より後ろならば，read commitment をどこに挿入してもプログラムの正しさは変わらない．つまり，同期プリミティブから

☆ PARMACS は共有メモリの確保，task の生成，同期プリミティブを提供する並列化マクロである．
☆☆ fusion と coalescing はループ変換が名前の由来である．
☆☆☆ PARMACS の提供する共有メモリ確保のプリミティブ．
☆4 lock 時，バリアから出発するとき．

$$\mathrm{ANTOUT}(i) = \bigwedge_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \quad (1)$$

$$\mathrm{ANTIN}(i) = \mathrm{COMP}(i) \vee (\mathrm{TRANS}(i) \wedge \mathrm{ANTOUT}(i)) \quad (2)$$

$$\mathrm{AVIN}(i) = \bigwedge_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \quad (3)$$

$$\mathrm{AVOUT}(i) = (\mathrm{COMP}(i) \vee \mathrm{AVIN}(i)) \wedge \mathrm{TRANS}(i) \quad (4)$$

$$\mathrm{INSERT}(i) = \mathrm{ANTIN}(i) \wedge \neg\Big(\bigwedge_{p \in \mathrm{pred}(i)} (\mathrm{TRANS}(p) \wedge \mathrm{ANTIN}(p)) \Big) \wedge \neg \mathrm{AVIN}(i) \quad (5)$$

〔pred(i) は文 i に先行（precede）する文の集合であり，succ(i) は文 i に後継（succeed）する文の集合である〕

図 3 冗長な read commitment を削除するデータフロー方程式
Fig. 3 Dataflow equations used to remove redundant read commitments.

対応する読み出しにたどり着く前に，同じアドレスと大きさの組に対して何度 read commitment を発行しても，最初の 1 回だけ発行したときと意味は変わらない．この read commitment の挿入位置の自由度により，redundancy elimination のフレームワーク[5),19)]が適用可能になる．

簡単な例（**図 2**）を用いて説明する．a は共有領域上に確保された `double` の二次元配列とする．手続き間別名解析の結果から，まず，図 2 の真中のように，同一の場所（`&a[ii][jj]`）への両方の読み出しの前に read commitment が挿入される．そこで，条件節の中の read commitment の発行を条件節の前にまで早めれば，条件節の後の方が冗長になる．その結果，図 2 の右のようなコードが生成される．

3.3.1.1 冗長性削除のデータフロー方程式

上記の最適化を行うために，read commitment の冗長性削除（RER: Redundancy Elimination for Read commitments）を行うデータフロー方程式を求める．上の例のように，同じ場所に対して発行される read commitment を 1 つ固定して考える（`R(a,s)`）．別名解析の結果から，各文 i ☆に対して以下の 2 種類の真偽値をとる定数を求める☆☆．

COMP(i)　文 i が着目した連続共有領域 $\{x \mid \mathtt{a} \le x < \mathtt{a}+\mathtt{s}\}$ に対して読み出しを行う（以下，単に「当該共有読み出しを行う」と表現する）．

TRANS(i)　文 i を越えて read commitment `R(a,s)` を移動することはプログラムの意味を変更してしまう．

文 i が同期プリミティブである場合，または，read commitment のパラメータ（a もしくは s）を更新する場合に，TRANS(i) は偽となる．

上述の定数の値から，プログラム内の任意の点☆☆☆ p に対して真偽値をとる以下の 2 種類のデータフロー変数を算出する．

Availability　p に至る任意の制御経路において，当該共有読み出しが行われる．仮りに p で当該共有読み出しを行った場合，どの制御経路を経て p に到達しても，その最後の当該共有読み出しと同一の結果が得られる．

Anticipatability　p から始まる任意の制御経路において，当該共有読み出しが行われる．p に read commitment を置くことで，p から始まるすべての制御経路上の最初の当該共有読み出しが有効になる．

Read commitment の数を最小にするには，

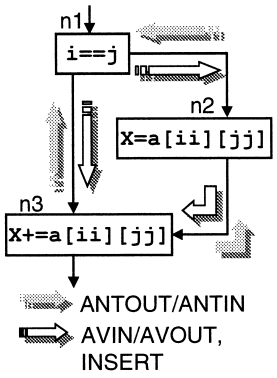
- 当該共有読み出しが anticipatable であり，かつ
- 先行する文のどれかにおいて，
  - 当該共有読み出しが anticipatable ではない
  - 同期プリミティブが実行される☆4

  のいずれかが成立し，かつ
- 当該共有読み出しが available でない

ような点にだけ，read commitment を置けばよい☆5．

文 i を実行する前と実行した後の Anticipatability を ANTIN(i)，ANTOUT(i) と表記する．Availability に関しても同様に AVIN(i)，AVOUT(i) と表記する．文 i を実行する前に read commitment を実際に挿入するかどうかを示す真偽値をとるデータフロー変数を INSERT(i) と表すと，各データフロー変数は**図 3**

☆ 以下，文は，C 言語のソースプログラム内の文ではなく，中間語における文を指すものとする．
☆☆ COMP は computation，TRANS は transparency を示す．
☆☆☆ 基本ブロックの中で，最初の文の前，2 つの隣り合う文の間，および最後の文の後のこと．
☆4 共有変数を引数にとる同期プリミティブが存在するから．
☆5 点（中間語における文の境界）に挿入されるので，C 言語のレベルでは式の途中に挿入される場合もある．



Node No. → / Property ↓	1	2	3
COMP		T	T
TRANS	T	T	T
ANTIN	T	T	T
ANTOUT	T	T	
AVIN			
AVOUT	T	T	
INSERT	T		

図 4 データフロー方程式の解
Fig. 4 Solution of data flow properties.

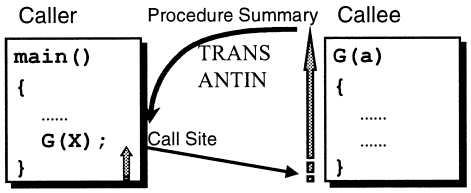


図 5 手続き間解析のモデル
Fig. 5 Interprocedural analysis model.

のデータフロー方程式に従って算出される．INSERT が真になるのは，同期やパラメータの書き換え後か，制御が分かれた直後（条件分岐で分かれた直後やループボディの最初）である．

図 2 の例を実際にこの方程式で解く過程を示したのが，**図 4** である．図の左が対応するコントロールフローグラフ（CFG）であり，右が真理値表である．まず，各ノード（文）に対して COMP，TRANS を求め，その値をもとに，CFG の下流から上流に向かって，ANTOUT，ANTIN を求め，CFG の上流から下流に向かって，AVIN，AVOUT を求め，最後に INSERT を求める．

3.3.2 手続き間解析への拡張

手続き間で RER を行うデータフロー方程式（図 3）を計算する．呼び出しグラフを主手続きから深さ優先で探索する．callee を解析するときには，call_site（手続き呼び出し）における caller の情報がマッピングされる．**図 5** に示されているように，`main` から解析して，手続き呼び出し `G(X)`（call_site）を発見したときには，手続き `G`（callee）がすでに解析済みならば，サマリを適用する．未解析の場合は `G` を解析して，サマリを計算し，caller（この場合は `main`）に適用する．

手続きのサマリは

- 各文における transparency の積（$\bigwedge_{i \in \mathtt{callee}} \mathrm{TRANS}(i)$）
- 入り口（enter）における anticipatability の値（ANTIN(`enter`)）

からなる．前者は必ず caller の transparency（TRANS(`call_site`)）に反映される．後者は手続きが閉じた場合[4] にのみ，caller の ANTIN(`call_site`) に変換されて反映される．再帰や，関数ポインタで呼ばれるような開いた手続きの場合には read commitment が手続きから出ることはなく，手続きの入り口に挿入される．

3.3.3 共有アクセス集合の導入

Read commitment は，アドレスだけでなくサイズをパラメータに持つので，複数の読み出しを同時に扱うことが可能になる．一方が他方を包含していたり，上位の制御フローで融合できるような場合を検出する．

たとえば，LU 分解の中の以下のようなコードを考える：

```
void daxpy(double *a, double *b,
           int n, double alpha)
{
  int i;
  for (i=0; i<n; i = i + 1)
    a[i] = a[i] + alpha*b[i];
}
```

`a` と `b` が共有領域上の double の配列であるとする．最内ループに read commitment を挿入する代わりに，

```
void daxpy(double *a, double *b,
           int n, double alpha)
{
  int i;
  R (a, n * sizeof(double));
  R (b, n * sizeof(double));
  for (i=0; i<n; i = i + 1)
    a[i] = a[i] + alpha*b[i];
}
```

とすることができ，コヒーレンス管理の回数を削減できる．

上記のように，複数の read commitment をまとめるには，一連の read commitment を**共有アクセス集合**で表記すると便利である．共有アクセス集合 $S = (a, s, \mathcal{C})$ は 3 つ組で，a と s は read commitment の先頭アドレスとサイズ，C は read commitment を生成する不等式制約の集合であり，read commitment を含むループの誘導変数を表している．実際には，read commitment を含むループの誘導変数の組 $(\mathtt{j\_1}, \mathtt{j\_2}, \ldots, \mathtt{j\_k})$ を $\mathtt{j}$ とすると，共有アクセス集合は $S = (a(\mathtt{j}), s(\mathtt{j}), \mathcal{C}(\mathtt{j}))$ と表記される．

Node No. → Property ↓	1	2	3
COMP	$\emptyset$	$\{S_1\}$	$\{S_2\}$
TRANS	Id	Id	Id
ANTIN	$\{S_1\}$	$\{S_1\}$	$\{S_2\}$
ANTOUT	$\{S_1\}$	$\{S_2\}$	$\emptyset$
AVIN	$\emptyset$	$\emptyset$	$\emptyset$
AVOUT	$\{S_1\}$	$\{S_2\}$	$\emptyset$
INSERT	$\{S_1\}$	$\emptyset$	$\emptyset$

図 6 共有アクセス集合の集合を使用したときの図 4 のデータフロー方程式の解
Fig. 6 Solution of data flow properties using sets of shared-access sets for the Fig. 4.

さらに，各データフロー変数を真偽値ではなく，共有アクセス集合の集合 $\{S_1, S_2, \ldots, S_n\}$ として表す．文 i における transparency，TRANS(i) は共有アクセス集合の集合を引数にして，共有アクセス集合の集合を返す関数になる．RER を行うデータフロー方程式（図 3）の論理演算 $\vee, \wedge, \neg$ は集合演算 $\cup, \cap, -$ になる．概念的には，個々の要素（共有アクセス集合）S_i について図 3 の演算を行って得た真理値の集合を求めることになる．図 4 の問題を集合演算として解いたものが**図 6** である．Id は任意の共有アクセス集合の集合をそれ自身に変換する恒等関数，$\emptyset$ は空集合である．$S_1 = \mathrm{COMP}(2) = (\&a[ii][jj], 8, \emptyset)$，$S_2 = \mathrm{COMP}(3) = (\&a[ii][jj], 8, \emptyset)$ である．$S_1 \equiv S_2$ であるから，たとえば，$\mathrm{ANTIN}(1) = \bigcap_{s\in\{2,3\}} \mathrm{ANTOUT}(s) = \mathrm{ANTOUT}(2) \cap \mathrm{ANTOUT}(3) = \{S_1\} \cap \{S_2\} = \{S_1\}$ となる．

3.3.4 区間解析[3]の適用

手続き間別名解析の直後に共有読み出しが検出された直後は，すべての共有アクセス集合はただ 1 つの read commitment を含む．つまり $s =$ `sizeof(X)`，$C = \emptyset$ である（`X` は共有アクセスされた型）．ループ構造を利用して，先ほどのデータフロー方程式を効率良く解くために，区間解析のフレームワークを適用する．区間は自然なループであり，区間のヘッダが区間内のすべてのノードを支配する．もちろん，CFG が可約☆であると仮定している．区間解析は循環道の影響を計算する消去相と区間順の走査を行う伝播相に分けられる．

☆ 強連結成分の入口が 1 個しかない．

- 消去相
  以下の操作を最内区間から最外区間に向かって実行する．
  (1) 区間をヘッダからラッチまでの無閉路グラフだと見なして，データフロー変数を計算し，1 イテレーション分のサマリを求める．図 3 の方程式の (2)，(4) において，論理和を計算する際に，つまり，共有アクセス集合から成る集合の和集合を計算する際に，プログラム中の異なる読み出しから派生する複数の read commitment をまとめて発行する fusion（3.3.4.1 項）という最適化を行う．
  (2) (1) の結果に区間の制約を表すインデクスの不等式（$\mathcal{C}$）を付加してループ全体のサマリを算出する．その際，イテレーション間の連続アクセスを利用した，coalescing（3.3.4.2 項）や，イテレーション間の冗長性を削除する redundant index elimination（3.3.4.3 項）を実行する．
  (3) 区間全体をヘッダに簡約する．つまり，外のループを扱うときには，中のループは 1 個のノードとして扱われる．
- 伝播相
  もとの CFG（簡約されていない）を 1 回探索することで，消去相の結果を各ノード（文）に伝播する

共有アクセス集合が，coalescing や redundant index elimination なしに区間の外に伝播するときは，ループ変換における fission（または distribution）を行っていることに相当する．つまり，対象のループを read commitment を含むループともとの共有アドレスへの読み出しを行うループに分割することに対応している．

3.3.4.1 Fusion

たとえば，以下のようなコードを考える（`p` は共有領域上の int の配列である）．

```
n1:x = p[0];
n2:y = p[1];
```

$S_1 = (\&p[0], 4, \emptyset) = (p, 4, \emptyset)$,
$S_2 = (\&p[1], 4, \emptyset) = (p + 4, 4, \emptyset)$

とすると $\mathrm{COMP}(\texttt{n1}) = \{S_1\}, \mathrm{COMP}(\texttt{n2}) = \{S_2\}$ であり，`n1` における ANTIN を求める際に

$$\begin{aligned}\mathrm{ANTIN}(\texttt{n1}) &= \mathrm{COMP}(\texttt{n1}) \cup \mathrm{TRANS}(\texttt{n1})(\mathrm{ANTIN}(\texttt{n1})) \\ &= \{S_1\} \cup \{S_2\} = \{S_1, S_2\}\end{aligned}$$

という共有アクセス集合の集合が出現する．この中の 2 個の要素 S_1, S_2 を以下のように 1 個の要素 S' に融

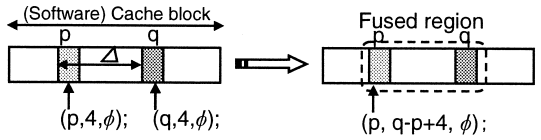


図 7 Fusion の例
Fig. 7 Example of fusion operation.

合する（fusion）.

$$S_1 \circ S_2 \rightarrow S' = (p, 8, \emptyset)$$

“$\circ$” は，プログラム中の異なる読み出しから派生する read commitment をまとめる操作である.

ただし，fusion を実施するためには，上の例のように必ずしも共有アクセス集合が連続である必要はない. コヒーレンスはワード単位ではなく，キャッシュブロック単位で維持されるので，チェックされる領域の連続性より，同一キャッシュブロック上にあるかどうかが問題になる.

したがって，**図 7** に示されるように，$S_1 = (p, 4, \emptyset)$, $S_2 = (q, 4, \emptyset)$ で，$\Delta = (q - p)$ とすると，$\Delta - 4 \leq$ block_size が成立する場合，つまり，領域間の距離がブロックサイズより小さい場合には，同一キャッシュ上にある可能性が高くなるから，以下のように，

$$S_1 \circ S_2 \rightarrow (p, q - p + 4, \emptyset)$$

fusion する．もちろん，fusion された領域が複数ブロックに跨るような場合も出てくるが，そのような場合は read commitment の実行時に検出され，対処される.

定式化を以下に示す.

Property 3.1（Fusion） 共有アクセス集合の集合の中に，2 個の異なる要素 $S_1 = (a_1(\mathtt{i}), s_1(\mathtt{i}), \mathcal{C}_1(\mathtt{i}))$, $S_2 = (a_2(\mathtt{j}), s_2(\mathtt{j}), \mathcal{C}_2(\mathtt{j}))$ が存在し，$\mathcal{C}_1 \equiv \mathcal{C}_2$，かつ，$\forall \mathtt{i} \in \mathcal{C}_1$ $(a_1(\mathtt{i}) < a_2(\mathtt{i})$ かつ，$a_2(\mathtt{i}) - a_1(\mathtt{i}) \leq$ block_size $+ s_1(\mathtt{i}))$ が成立するときに，S_1，S_2 は以下のように 1 個の要素に融合される.

$$S_1 \circ S_2 \rightarrow (a_1(\mathtt{i}), \max(a_2(\mathtt{i}) - a_1(\mathtt{i}) + s_2(\mathtt{i}), s_1(\mathtt{i})), \mathcal{C}_1(\mathtt{i})).$$

3.3.4.2 Coalescing

Coalescing というのはイテレーション間で連続な共有アクセスが存在した場合，それらを一括して発行する最適化である．3.3.3 項の `daxpy` の例では，共有読み出しを行う共有アクセス集合は $S_1 = (\&a[i], 8, \{0 \leq i < n\})$, $S_2 = (\&b[i], 8, \{0 \leq i < n\})$ である. $\&a[i]$, $\&b[i]$ の連続性により，$S_1 \rightarrow (\&a[0], 8*n, \emptyset) = (a, 8*n, \emptyset)$, $S_2 \rightarrow (\&b[0], 8*n, \emptyset) = (b, 8*n, \emptyset)$ とまとめられる．Fusion と同様，連続性より同一ブロック上にあるかどうかが問題なので，チェックされる領域がイテレーション間で連続でなくても，その間隔がブロックサイズ以下ならば，一括化する．定式化を以下に示す.

Property 3.2（Coalescing） 共有アクセス集合 $S = (a(\mathtt{j}), s(\mathtt{j}), \mathcal{C}(\mathtt{j}))$, $\mathtt{j} = (\mathtt{j\_0}, \mathtt{j\_1}, \ldots, \mathtt{j\_k})$ に対して，c をインデクス j_0 のストライドとし，$\mathtt{c} = (c, 0, \ldots, 0)$ と表現するときに $\forall \mathtt{j} \in \mathcal{C}$ $(\Delta(\mathtt{j}) = a(\mathtt{j}+\mathtt{c}) - a(\mathtt{j})$ としたときに，$\Delta(\mathtt{j}) = \Delta(\mathtt{j}+\mathtt{c})$，かつ，$s(\mathtt{j}) = s(\mathtt{j}+\mathtt{c})$，かつ，$\Delta(\mathtt{j}) \leq$ block_size$+s(\mathtt{j}))$ が成立するならば，S は以下のように一括化される.

$$S' = (a(\tilde{\mathtt{j}}), \Delta(\tilde{\mathtt{j}}) \cdot (n-1) + s(\tilde{\mathtt{j}}), \mathcal{C}(\mathtt{j}) - \mathcal{I}(\mathtt{j\_0}))$$

（ただし，n はイテレーションの数であり，$\tilde{\mathtt{j}} = (\mathtt{min\_0}, \mathtt{j\_1}, \ldots, \mathtt{j\_k})$（min_0 はインデクス j_0 の最小値）であり，$\mathcal{I}(\mathtt{j\_0})$ は j_0 に関しての制約不等式である.）

一括化された領域が複数ブロックに跨るような場合は実行時に検出され，対処される.

3.3.4.3 Redundant Index Elimination

Redundant index elimination は，あるイテレーション i_0 において行われる共有読み出しが，他のすべてのイテレーション {i_1, ..., i_n} において行われる共有読み出しを包含する場合に，イテレーション i_0 における共有読み出しでループ全体における共有読み出しを表す最適化である．すなわち，redundant index elimination はループを表すインデクスを消去する操作である．インデクスの消去は Fourier-Motzkin 消去で実現できる.

たとえば，LU 分解の中の以下のようなコード

```
for (k=0; k<dimi; k = k + 1) {
  for (j=0; j<dimj; j = j + 1) {
    c[k+j*stride_c] /= a[k+k*stride_a];
    alpha = -c[k+j*stride_c];
    length = dimi - k - 1;
    daxpy(&c[k+1+j*stride_c],
          &a[k+1+k*stride_a],
          dimi-k-1, alpha);
  }
}
```

で，c が共有領域上の double の配列とする．c を使った読み出しから共有アクセス集合 S_1 が得られ，手続き daxpy（3.3.3 項）のサマリから共有アクセス集合 S_2 が得られ，S_1 と S_2 は fusion によって S_3 に融合することができる．以下では sizeof(double)=8 として議論を進める.

$S_1 = (\&c[k + j * stride\_c], 8, \emptyset)$,

$S_2 = (\&c[k + 1 + j * stride\_c], 8 * (dimi - k - 1), \emptyset)$,

$S_1 \circ S_2 \rightarrow S_3 = (\&c[k+j*stride\_c], 8*(dimi-k), \emptyset)$
S_3 は j について coalescing できないので，j のループのサマリ S_4 は，j の制約を S_3 に加えたものになる．

$$S_3 \rightarrow S_4 = (\&c[k + j * stride\_c], 8 * (dimi - k), \{0 \leq j < dimj\})$$
$$\rightarrow S_5 = (\&c[k + j * stride\_c], 8 * (dimi - k), \{0 \leq j < dimj,\ 0 \leq k < dimi\})$$

S_5 は k について coalescing できないが，S_5 のすべての共有読み出しは $k = 0$ のときの共有読み出しに含まれている．これを検出するために，共有アクセス集合で表される場所を示す補助変数 x を導入する．簡単のために c のインデクスによって場所を表すとする．S_5 の場所の集合は，インデクス変数 x の集合で表される．x の制約はループの不等式制約 $\mathcal{C}$ を加えた以下の不等式集合 $\mathcal{C}'$ で表すことができる．

$\mathcal{C}' = \{8 * (k + j * stride\_c) \leq x < 8 * (j * stride\_c + dimi),\ 0 \leq j < dimj,\ 0 \leq k < dimi\}$．$k$ に注目すると，$\mathcal{C}'$ の中の不等式を 3 種類に分類できる．

- $\mathcal{L}(k, \mathcal{C}')$：$\mathcal{C}'$ の不等式のうち k の上限を与えるもの
- $\mathcal{G}(k, \mathcal{C}')$：$\mathcal{C}'$ の不等式のうち k の下限を与えるもの
- $\mathcal{I}(k, \mathcal{C}')$：$\mathcal{C}'$ の不等式のうち k を含まないもの

$\mathcal{L}(k, \mathcal{C}') = \{8 * k \leq x - 8 * j * stride\_c,\ k < dimi\}$,
$\mathcal{G}(k, \mathcal{C}') = \{0 \leq k\}$,
$\mathcal{I}(k, \mathcal{C}') = \{x < 8 * j * stride\_c + 8 * dimi,\ 0 \leq j < dimj,\ 0 < dimi\}$.

である．k の上限と下限についての制約を組み合わせることで，$\mathcal{C}$ から k を消去することができる．この操作を演算子 “×” で表すとする．k を消去して得られる不等式は，

$\mathcal{L} \times \mathcal{G} = \{0 \leq x - 8 * j * stride\_c,\ 0 < dimi\}$,
$\mathcal{L} \times \mathcal{G} \cup \mathcal{I} = \{0 \leq x - 8 * j * stride\_c,\ 0 < dimi,\ x < 8 * j * stride\_c + 8 * dimi,\ 0 \leq j < dimj\}$,

であり，$\{8 * j * stride\_c \leq x < 8 * j * stride\_c + 8 * dimi,\ 0 \leq j < dimj\}$ とまとめられる．

この制約から，k が消去された共有アクセス集合を復元すると，$S_5 \rightarrow S_6 = (\&c[j * stride\_c], 8 * dimi, \{0 \leq j < dimj\})$ となる．

Fourier-Motzkin 消去は，線形でなくても，単調であれば適用可能である．たとえば，FFT の中の以下のようなコード（N は非負の整数）

```
for (q=1; q<=M; q = q + 1) {
  L = 1<<q; r = N/L; Lstar = L/2;
  u1 = &u[2*(Lstar-1)];
  for (k=0; k<r; k = k + 1) {
    x1 = &x[2*(k*L)];
    x2 = &x[2*(k*L+Lstar)];
    for (j=0; j<Lstar; j = j + 1) {
      omega_r = u1[2*j];
      omega_c = direction*u1[2*j+1];
      x_r = x2[2*j];
      x_c = x2[2*j+1];
      tau_r = omega_r*x_r - omega_c*x_c;
      tau_c = omega_r*x_c + omega_c*x_r;
      x_r = x1[2*j];
      x_c = x1[2*j+1];
      x2[2*j] = x_r - tau_r;
      x2[2*j+1] = x_c - tau_c;
      x1[2*j] = x_r + tau_r;
      x1[2*j+1] = x_c + tau_c;
    }
  }
}
```

で，x が共有領域上の double の配列とする．最内ループ j 内には 4 個の共有読み出し（S_0, S_1, S_2, S_3）が存在し，fusion で S_4, S_5 に融合される．

$$S_0 = (\&x2[2 * j], 8, \emptyset)$$
$$S_1 = (\&x2[2 * j + 1], 8, \emptyset)$$
$$S_2 = (\&x1[2 * j + 1], 8, \emptyset)$$
$$S_3 = (\&x1[2 * j + 1], 8, \emptyset)$$
$$S_0 \circ S_1 \rightarrow S_4 = (\&x2[2 * j], 16, \emptyset)$$
$$S_2 \circ S_3 \rightarrow S_5 = (\&x1[2 * j], 16, \emptyset)$$

S_4, S_5 はループ j に関して以下のように coalescing される．

$$S_4 \rightarrow S_6 = (\&x2[2 * j], 16, \{0 \leq j < Lstar\})$$
$$\rightarrow S_7 = (x2, 16 * Lstar, \emptyset)$$
$$= (\&x[2 * (k * L + Lstar)], 16 * Lstar, \emptyset)$$
$$S_5 \rightarrow S_8 = (\&x1[2 * j], 16, \{0 \leq j < Lstar\})$$
$$\rightarrow S_9 = (x1, 16 * Lstar, \emptyset)$$
$$= (\&x[2 * (k * L)], 16 * Lstar, \emptyset)$$

S_7, S_9 は以下のように fusion される．

$$S_7 \circ S_9 \rightarrow S_{10} = (\&x[2 * (k * L)], 32 * Lstar, \emptyset)$$

ループ k に関する制約を加え，Lstar を展開する．

$$S_{10} \rightarrow S_{11} = (\&x[2 * (k * L)], 32 * (L/2), \{0 \leq k < N/L\})$$

Property 3.2 における Δ（アドレスの差分）の値は，

$$\Delta(k) = a(k+1) - a(k)$$
$$= \&x[2 * ((k+1) * L)] - \&x[2 * (k * L)]$$
$$= 8 * (2 * L)\ (x \text{ は double の配列})$$
$$= 16 * L$$

と k によらずに求められる．$L = 2^q$ で $1 \le q \le M$ だから L は偶数となり，$s(k) = 32*(L/2) = 32*2^{q-1} = 16 * 2^q = 16 * L$ が成立し，k に依存しない値になる．ゆえに，任意の $k \in \{0 \le k < N/L\}$ に対して $\Delta(k) = s(k)$ が成立し，coalescing される．

$$S_{11} \to S_{12} = (x, 16 * L * (N/L), \emptyset) = (x, 16 * 2^q * (N/2^q), \emptyset)$$

ループ q に関する制約を加えて以下のようになる，

$$S_{12} \to S_{13} = (x, 16 * 2^q * (N/2^q), \{1 \le q \le M\})$$

S_{13} のすべての共有読み出しは $q = 1$ のときのそれに含まれている．これを検出するために，先程と同様に共有アクセス集合で表される場所を示す補助変数 x を導入する．x の制約は，ループの不等式制約 $\mathcal{C}$ を加えた以下の不等式集合 $\mathcal{C}'$ である．

$$\mathcal{C}' = \{0 \le x < 16 * 2^q * (N/2^q), 1 \le q \le M\}$$

$\mathcal{C}'$ を先程のように 3 種類に分類する．非負の整数 q に対して $f(q) = 2*2^q*(N/2^q)$（N は非負の整数，演算子'/' は整数の除算）と定義する．2^q は q に関して単調増大であり，非負の整数 Q に関して $2*Q*(N/Q)$ は単調減少であることから，$f(q)$ は非負の整数 q に関して単調に減少する．さらに，「単調減数関数 $DEC(z)$ と定数 VAL に対して，$VAL < DEC(z)$ は z の上限を与える」という事実から，$x < f(q)$ が q の上限を与えることが分かる．

よって

$$\mathcal{L}(q, \mathcal{C}') = \{0 < 16 * 2^q * (N/2^q), x < 16 * 2^q * (N/2^q), q \le M\}$$
$$\mathcal{G}(q, \mathcal{C}') = \{1 \le q\}$$
$$\mathcal{I}(q, \mathcal{C}') = \{0 \le x,\ 1 \le M\}$$
$$\mathcal{L} \times \mathcal{G} \cup \mathcal{I} = \{0 \le x < 32 * (N/2),\ 0 < 32 * (N/2),\ 1 \le M\}$$

が得られる．上式から共有アクセス集合を復元すると，

$$S_{13} \to S_{14} = (x, 32 * (N/2), \emptyset)$$

となる☆．

定式化を以下に示す．

Property 3.3 Redundant index elimination 共有アクセス集合 $S = (a(\mathtt{j}), s(\mathtt{j}), \mathcal{C}(\mathtt{j}))$, $\mathtt{j} = (\mathtt{j\_0}, \mathtt{j\_1}, \ldots, \mathtt{j\_k})$ に対して，j_0 に関する制約を $\mathcal{I}(\mathtt{j\_0})$ とする．$\exists \alpha \in \mathcal{I}\ \forall \mathtt{j} = (\mathtt{j\_0}, \mathtt{j\_1}, \ldots, \mathtt{j\_k}) \in \mathcal{C}$ に対して $\tilde{\mathtt{j}} = (\alpha, \mathtt{j\_1}, \ldots, \mathtt{j\_k})$ と置くとき，$a(\tilde{\mathtt{j}}) \le a(\mathtt{j}) + s(\mathtt{j}) \le a(\tilde{\mathtt{j}}) + s(\tilde{\mathtt{j}})$ が成立するなら，インデクス変数 j_0 は冗長であり，以下のように削除される．

$$S \to S' = (a(\tilde{\mathtt{j}}), s(\tilde{\mathtt{j}}), \mathcal{C}(\mathtt{j}) - \mathcal{I}(\mathtt{j\_0}))$$

☆ SPLASH-2 [25] のアプリケーション群の中で，上記のように，非線形式に対して Fourier-Motzkin 消去が成功した例は FFT のみである．

3.4 Write commitment を用いた最適化

LRC モデルでは，共有データに書き込みが行われたことは，書き込みを行ったノードがその後に同期プリミティブを発行するまで外には知らされない．したがって，write commitment は，対応する共有書き込みの後から，その後最初に現れる同期プリミティブの前までの，制御フロー中のどこに置いても意味は同じである．また，同期プリミティブにたどり着く前に，同じアドレスと大きさの組に対して何度 write commitment を発行しても，最後の 1 回だけ発行したときと意味は変わらない．我々はこの自由度を利用して，read commitment の冗長性削除（RER）と同様なデータフロー方程式を解くことで，冗長な write commitment の削除（REW: Redundancy Elimination for Write commitments）を実行する [12),18)]．REW を行うデータフロー方程式は，RER を行うデータフロー方程式と同様に，区間解析と手続き間解析の枠組みで効率良く解くことが可能である．REW と RER との違いを以下に示す．

- RER の計算は anticipatability を主に使用していたが，REW の計算は主に availability を使用する．
- RER の計算では fusion と coalescing の条件は近傍性であるが，REW の計算ではその条件は厳密な連続性である．

  つまり，REW の fusion における条件は $a_2(\mathtt{i}) - a_1(\mathtt{i}) \le \text{block\_size} + s_1(\mathtt{i})$ ではなく $a_2(\mathtt{i}) - a_1(\mathtt{i}) \le s_1(\mathtt{i})$ であり，coalescing における条件は $\Delta(\mathtt{j}) \le \text{block\_size} + s_1(\mathtt{j})$ ではなく $\Delta(\mathtt{j}) \le s_1(\mathtt{j})$ である．

REW では coalescing はインデクスが誘導変数でなくても，連続であれば可能である．たとえば Radix の中のコード：

```
for (i=key_start; i<key_stop; i++) {
  this_key = key_from[i] & bb;
  this_key = this_key >> shiftnum;
  tmp = rank_ff_mynum[this_key];
  key_to[tmp] = key_from[i];
  rank_ff_mynum[this_key]++;
} /* i */
```

で，key_to が共有領域を指すとする．key_to を用いた共有書き込みのインデクス変数 rank_ff_mynum[this_key] は，誘導変数ではない．したがって，このままではループ内で整数サイズの write commitment が発行される．

しかし，rank_ff_mynum[this_key] は，共有書き

込みがあるときだけ 1 ずつ増える．ループが実行されると，各 rank_ff_mynum[this_key] のループ前の初期値からそれぞれの最終値までのインデクス値を使って共有書き込みが行われる．したがって，key_to への書き込みの write commitment は，rank_ff_mynum[this_key] の初期値と最終値を使って以下のように coalescing される．

```
for (i = 0; i <= bb >> shiftnum; i++) {
    init_rank_ff_mynum[i] = rank_ff_mynum[i];
}

for (i = key_start; i < key_stop; i++) {
    this_key = key_from[i] & bb;
    this_key = this_key >> shiftnum;
    tmp = rank_ff_mynum[this_key];
    key_to[tmp] = key_from[i];
    rank_ff_mynum[this_key]++;
}   /* i */

for (i = 0; i <= bb >> shiftnum; i++) {
    W (&key_to[init_rank_ff_mynum[i]],
      (rank_ff_mynum[i] - init_rank_ff_mynum[i])
       * sizeof (int));
}
```

このように，あるループ等のコード領域で，初期値から最終値までのすべての値をとるような変数を連続変数（continuous variable）と呼ぶ[12]．正規化された誘導変数は連続変数である．REW では，coalescing 最適化は連続変数に対して適用される．連続変数は誘導変数を検出する条件を緩めることで検出できる．詳細は文献 12)，20) を参照されたい．

実際のプログラムで誘導変数ではないが連続変数になるのは，

- ループ不変の場所の値が条件的にループ不変の増分値で増加する場合
- ループ依存の場所がループ不変の増分値で増加する場合

である．上の Radix の例は後者の場合である．

実際に連続変数を使って coalescing を行う場合には，以下の条件を確かめる必要がある．

- 共有書き込みと連続変数の更新の間の支配関係
  誘導変数は各イテレーションごとに更新されるので，イテレーション内で必ず実行される文に対しては，支配/非支配の関係にある．それに対して，連続変数は条件的に更新/使用されることがある．したがって「連続変数が更新されたときに，必ずその値を使った共有書き込みが行われたか」どうかを確かめる必要がある．そのためには，書き込みが連続変数の更新を dominate または post-dominate している必要がある．Radix の例では，共有書き込みが連続変数の更新を dominate している．
- ループ依存の場所が列挙可能
  ループ依存な場所に格納された連続変数を扱うときには，その場所は列挙できる必要がある．Radix の例では，変数 this_key の作り方から，配列 rank_ff_mynum のインデクスの下界（0）と上界（bb >> shiftnum）が求められる．さらに配列 rank_ff_mynum の要素数が bb >> shiftnum より大きいことを確認する必要がある．本文中には記されていない部分のコード[25]から，配列 rank_ff_mynum の要素数が radix であり，bb = radix - 1 << shiftnum であることが分かる．したがって，上界（bb >> shiftnum） < 要素数（radix）が得られる．

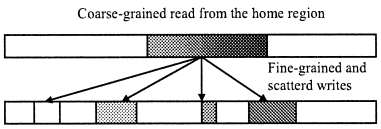


図 8 配列の並べ替え
Fig. 8 Permutation phase.

3.5 Fetch-on-write の除去

従来の OS ベースのシステムでは，ブロックサイズ（コヒーレンス単位）がページであり，ページ全体を書き潰す命令が存在しないため，fetch-on-write 方式を採用している．つまり，書き込む前に読み出しを行う．無効なブロックに書き込もうとする場合，キャッシュミスが発生し，キャッシュミスハンドラがブロック全体を最新のものに更新してから書き込み時のコヒーレンス操作を実行する．その後に，store が実行される．

しかしながら，fetch-on-write を行うことが高オーバヘッドを誘発する場合が多数ある．double buffering して radix sort する場合（**図 8**）には，一方の配列の一部を連続して読み出し，他方の配列全体にわたって短冊状の書き込みを行い，同期を実行して，配列を交換して同様の操作を繰り返す．つまり，双方の配列の大部分は書き込みされるだけで，読み出しされない．行列の転置操作を行う場合（**図 9**）も同様のことが発生する．

上記のような書き込みを fetch-on-write 方式で実現すると，ただ書き込むだけで後に使用しないにもかかわらず，ほとんどすべての書き込みに対してキャッシュ

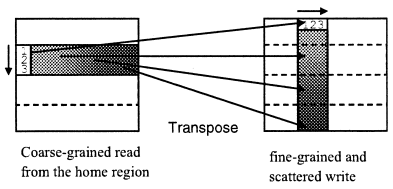


図 9 行列の転置
Fig. 9 Transpose phase.

ミスが発生し，無駄なデータ転送が発生する．OS ベースのシステムでは「あるブロックが書き込まれるだけで，読み出しはされない」ということを検出する手段はない．

一方，コンパイラベースのシステムでは，同期で囲まれた interval[15] において，「書き込まれるだけで，読み出されないブロック」や「自ノードが書き潰した部分からしか読み出されないブロック」が検知可能である．上記のブロックに対しては，書き込みの前にブロックを更新する操作を省略することで，無駄な通信を削減可能である．

共有領域が動的に確保されるような場合に，異なるデータが同一ブロックに存在する場合が生じる．その結果，上記のブロックを完全に検出することが困難になる．しかし，手続き間別名解析を使用すれば，同一個所を読み書きする可能性がないどうかは判断できるため，低オーバヘッドで fetch-on-write を避けることが可能である．

3.5.1 UDSM

共有データのコヒーレンスが読み書き共にコンパイラが挿入するコードで実現される場合，すべての書き込みに対して，書き込みの前のブロック更新操作を省略する．つまり，書き込みの前に read commitment を発行しない．

無効なブロックに対して store を行い，その write commitment を発行する前に同一ブロックへの read commitment が発行される場合，ブロック全体が更新されてしまい，先の最新の store の結果が失われる可能性がある．したがって，read commitment が実行される前に write commitment を実行する必要がある．つまり，write commitment の発行にとって，read commitment の挿入場所☆が，同期プリミティブと同様な意味を持つ．

(1) 3.2 節の手続き間別名解析の手法で共有読み出し，書き込みを検出する．ただし，書き込みの前に読み出しは行わない．

(2) 読み出しに対しては，3.3 節の最適化を施し，read commitment の挿入場所を求める．

(3) (2) で求めた read commitment の挿入場所が同期と同じ意味を持つと仮定して，書き込みに対し，3.4 節の最適化を施す．もちろん，write commitment が発行されるブロックと同一ブロックに対する read commitment のみが同期と同じ意味を持つが，read commitment と write commitment が同一ブロックに対してなされるかどうかの判定は容易ではないので，すべての read commitment が同期プリミティブと同様の意味を持つと仮定する．また，書き込まれた値をそのまま読むような場合には read commitment そのものが省略される．

たとえば，以下のようなコード

```
Barrier ();
for (i = 0; i < N; i = i + 1)
   S[i] = .0;
Barrier ();
```

に対して，通常では

```
Barrier ();
R (S, N * sizeof(double));
for (i = 0; i < N; i = i + 1)
   S[i] = .0;
W (S, N * sizeof(double));
Barrier ();
```

という出力になる．

Fetch-on-write をしない場合には，次のようなコードが生成される．

```
Barrier ();
for (i = ; i < N; i = i + 1)
   S[i] = .0;
W (S, N * sizeof(double));
Barrier ();
```

たとえ S のブロックが無効でも，S からの読み出しはないので，書き込む前の S のブロックの更新操作を省略できる．

3.5.2 ADSM

Fetch-on-write を避けるためには，書き込みの前のブロック更新操作が省略できることが必要である．しかし，ADSM では，ブロックミスは非明示的にページ管理機構によって検出され処理される．つまり，無効なブロックに書き込もうとすると，ページフォルトが発生し，ミスハンドラが呼ばれ，自動的にブロック更新操作が行われる．

書き込みの前に，「ミスしてもコヒーレンス管理操作を省略するようにランタイムに伝える」ルーチンを挿

☆ 読み出しそのものの場所ではない．

入する方法では，同一ブロックの書き込まれた個所以外への読み出しの対処が困難になる．したがって，書き込みのコードそのものを変換する必要が生じる．

たとえば，前述のコードは，以下のように変換される．

```
Barrier ();
for (i = 0; i < N; i++)
   _W_S[i] = .0;
W′ (S, N * sizeof(double), _W_S);
Barrier();
```

ローカルなクローン配列 _W_S を用意して，S への書き込みを _W_S への書き込みに変換する．write commitment の引数を 3 個に増やす．3 番目の引数はローカルな書き込みを行った領域へのポインタである．

一方向の更新方式（AURC[11] 等）を採用する場合，この write commitment（W′）内では，まず，他ノードへの無効化情報にこのブロックを加える．次に，_W_S の内容をブロックの home ノードに転送する．書き込みを行った自ノードにおいて S のブロックが有効なときのみ，_W_S の内容を S にコピーする．このコピーは，自ノードが同期で囲まれた interval[15] を越えて（後続の同期を実行後に）書き込んだ個所を読んだときに，正しいデータを読むために必要である．

自ノードが有効なブロックに対して書き込みを行うときは store の量が 2 倍になるため，すべての書き込みを W′ として発行することは高オーバヘッドにつながる．読み出す領域と書き込む領域が重なる場合には，その書き込みを W′ として発行する必要はなくなる．この事実をもとに，以下の手順に従ってコード生成を行う．

(1) 3.2 節の手続き間別名解析の手法で共有読み出し，書き込みを検出する．ただし，書き込みの前に読み出しは行わない．

(2) 書き込みに対して，共有アクセス集合を用いた 3.4 節の最適化を適用し，write commitment の挿入場所を決める．
その際，同期で囲まれた interval において，書き込まれる領域と重なる可能性のある領域からの読み出しが含まれる場合は，fetch-on-write する．そうでない場合は fetch-on-write しない．
プログラム内の場所は location set[24] と呼ばれるデータ構造 (b, f, s) で表されている．b はメモリブロック名で，変数名かヒープを表す名前である．s はメモリブロック b におけるバイト単位の刻み幅（stride），f は 0〜$s-1$ のバイト単位のオフセットであり，メモリブロック b の $\{f+is|i \in \mathcal{Z}\}$ というアドレスをアクセスすることを表す．この location set を比較することで，アクセスする領域が重なる可能性があるかどうかを判定する．

(3) 最後に fetch-on-write する/しないに分けて，write commitment（W/W′）を挿入する．
Fetch-on-write しない場合には共有アクセス集合 $S = (a, s, \mathcal{C})$ に対応するもとの書き込み文の集合（St）を変換する．

- 共有アクセス集合がマージされていないとき（サイズ s が基本サイズに等しいとき），
St の各文の左辺を，ローカルなクローン変数（_W_a）に変換する．コヒーレンス管理コードの第 3 引数は &_W_a である．
- 共有アクセス集合が coalescing 等でマージされているとき（サイズ s が基本サイズより大きいとき），
3.5.2 項の最初の例で示したように，St の各文の左辺の中で共有アクセス集合の第 1 要素 a に該当する部分をローカルなクローン配列（_W_a）に置き替える．コヒーレンス管理コードの第 3 引数は _W_a である．

```
for (i = 0; i < N; i = i + 1)
   S[i] = A[i]
```

のような，配列から配列へのコピーであるような場合には，クローン配列を経由せず，書き込みそのもの（St）を省略する．その場合の write commitment（W′）の第 3 引数は，以下のようにコピーされる配列の先頭アドレスである．

```
W′ (S, N * sizeof(double), A);
```

- クローン変数は静的に確保する．クローン配列の確保に関してだが，サイズ s が静的に確定できるものは静的に確保する．静的に確定できないものに関しては，対応する共有変数の G_MALLOC に合わせて動的に確保する（malloc）．コヒーレンス管理コードが発行されると，対応するクローン配列は用済みになるので，資源節約の点から，再利用可能なものは再利用する．

4. 実行時最適化

共有メモリはブロック単位で管理される．各ブロックにはホームノードが定められる．ホームノードはユーザが指定することが可能である．各ノードは，共有ブロック数の大きさの以下のビットテーブルを持つ．

Valid bit table 各ブロックの内容が valid であることを示す．

Dirty bit table 現在の interval において，自分が

ブロックに書き込んだことを示す．

また，ノード数の大きさの，次のビットテーブルを持つ．

Acknowledge table そのノードがホームであるブロックに自分が書き込んだことを示す．

各ロックは，dirty bit table を持ち，指定可能なホームノードによって管理される．以下，各操作でのランタイムシステムの振舞いを述べる．

コヒーレンスプロトコルとして先行研究の結果[21]から，AURC[11]を明示的な通信コードによってソフトウェアエミュレーションする SAURC[28]プロトコルを採用する．

write commitment（W）では，引数で指定された領域の各ブロックに対して以下のことを行う．該当ブロックを dirty bit table に記録する．ホームノードに，書き込まれたデータを送る．ただし，データサイズが小さいときはいったんアドレスとサイズとデータをバッファ上にセーブして，同一宛先のものはマージする（つまり，パケットコンバイニングを実行する）．データを送った相手を acknowledge table に記録する．

read commitment（R）では，引数で指定された領域の各ブロックに対して以下のことを行う．該当ブロックが有効かどうか，valid bit table を引いて調べる．無効なときはミスハンドラを呼ぶ．ミスハンドラはホームノードにリクエストを出して，ホームノードにブロック全体を転送してもらう．ブロックが転送されるまでノードは待つ．

Acquire 操作では，ロックのホームからロックの dirty bit table を転送してもらう．受け取った dirty bit table を自分の valid bit table に適用して，自ノードがホームでないブロックを無効化する．受け取った dirty bit table と自分の dirty bit table の和をとる．この方式は同期情報量，管理メモリ量が少なくて済む利点がある．だが，同一のノードがバリアをはさまずに同一のロックを再びとったとき，ロックの dirty bit table によってその間に書き込まれていないブロックであっても，再び無効化される可能性があるという欠点を持つ．

Release 操作では，acknowledge table に登録されているノードにメッセージを送って，自分が送ったメッセージがすべて到着したことを確認する．ロックのホームに自分の dirty bit table を送る．

バリア操作では，以下の手順をとる．

(1) 各ノードは，バッファリングしていて未転送のデータがある場合，それを転送する．acknowledge table を見てメッセージの到着を確認する．
(2) 各ノードはバリアのマスターに自分の dirty bit table を送る．
(3) マスターは送られた dirty bit table の和をとって，全員に送る．
(4) 各ノードはマスターから dirty bit table を使って，ホーム以外のページを無効化する．
(5) 各ノードは自分の dirty bit table，ロックの dirty bit table をクリアする．

ロックの acquire/release は，ホームに明示的にメッセージを転送することで逐次化される．たかだか，数ブロックしか書き込まれていないときに，dirty bit table としてビットテーブル全体を転送することは通信に負担をかける．したがって，dirty bit table として，ビットテーブルのサイズを越えない限りは，ブロック番号のリストを同時に管理する．ノードが dirty bit table を転送するときには，リストかビットテーブルかいずれか短い方を転送することで無駄なデータ転送を防ぐ[20),22)]．

5. 性能評価

5.1 アプリケーション

評価ではハードウェア DSM 用の明示的に並列に書かれたプログラム群 SPLASH-2[25]から，**図 10** に示されている 9 個のアプリケーションを使用した☆．Block-home，lock-home の配置の最適化をコメントをもとに実現した．LU-Contig，FFT，Raytrace，Ocean，Barnes に関しては，他の多くの Software DSM システムの評価と同様にソースを変更した[7),12),13)]．具体的な変更点を**表 1** に示す．

5.2 実験環境

実験環境は以下のようになる．

Program	内容
LU-Contig	密行列のブロック化LU分解
Radix	基底法による整数のソート
FFT	一次元、基底$\sqrt{n}$6段FFT、sender-iniateな転置
Barnes	Barnes-HutアルゴリズムによるN体問題
Raytrace	タスクキューを使用したレイトレーシング
WaterNS	水分子のN体問題$O(n^2)$
WaterSP	水分子のN体問題$O(n)$ 空間分割法
OceanRW	海洋の水流シミュレーション、横長ブロックに分割
Volrend	ボリュームレンダリング

図 10 SPLASH2 のアプリケーション群
Fig. 10 SPLASH2 applications

☆ 残りの 3 個に関しては，ソースに対する理解が十分ではないので，性能評価から外した．

表 1 ベンチマークの修正部分
Table 1 Benchmark configuration.

Program	Modification
LU-Contig	owner of block $(i, j) \leftrightarrow$ owner of block (j, i)
FFT	sender-initiated `Transpose` [12]
Raytrace	elimination of unused lock-operation for ray ID [12]
Ocean	rowwise partition (Ocean-RW) [13]
Barnes	sequential tree-construction [7]

表 2 問題サイズと逐次実行時間（秒）と最適化された並列 1 台実行時間（秒）とオーバヘッド（%）
Table 2 Problem size, sequential execution time (secs) and optimized parallel 1-node execution time (secs) and the overheads (%).

プログラム	問題サイズ	逐次実行	ADSM 最適化並列 1 台実行		UDSM 最適化並列 1 台実行	
			FOW	w/o FOW	FOW	W/O FOW
LU-Contig	2048^2 doubles	435.62	436.34 (0.16)	436.34 (0.16)	457.73 (5.1)	439.55 (0.99)
Radix	4M integer keys	6.42	6.44 (0.30)	7.80 (21)	8.55 (33)	6.75 (5.1)
FFT	1M complex doubles	18.27	18.29 (0.11)	22.03 (20)	18.49 (1.2)	18.48 (1.2)
Barnes	2^{15} bodies	66.63	67.14 (0.77)	67.86 (1.84)	75.10 (13)	74.83 (12)
Raytrace	balls4, 128^2 pixels	171.41	171.44 (0.017)	174.44 (1.76)	194.82 (14)	200.0 (17)
Water-NS	4096 molecules	464.11	471.84 (1.7)	472.59 (1.82)	477.74 (2.9)	479.36 (3.2)
Water-SP	4096 molecules	53.23	54.01 (1.4)	54.01 (1.4)	55.21 (3.7)	54.54 (2.4)
Ocean-RW	258^2 ocean	21.00	21.67 (3.2)	23.93 (14)	23.38 (11)	23.34 (11)
Volrend	head	4.11	4.12 (0.43)	4.14 (0.73)	5.36 (30)	5.00 (21)

- コンパイラ
  RCOP+gcc-2.7.2（最適化オプションは “-O2”☆）
- ノード
  Axil 320 model8.1.1（Sun SS20 互換機，85 MHz, SuperSPARCII×1）
  Fast Ethernet SBus Adapter2.0
- ネットワーク
  100BASE-TX のスイッチングハブ（3Com SuperStackII Switch 3900）で Fast Ethernet 接続
- OS – 汎用超並列オペレーティングシステム SSS-CORE [27] Ver.1.2.
  保護と仮想化の機能を保存したまま他ノードのメモリをユーザレベルで直接操作するメモリベース通信（**MBCF**: Memory-Based Communication Facilities）[28),29)] が提供されている
  – ピークバンド幅は 11.93（MBytes/sec）
  – ラウンドトリップレイテンシは 49（μsec）
- ランタイムシステム – 4 章で提案したもの
  通信に MBCF を使用
  – 遠隔データの読み書き
  – リモートからのリクエストメッセージの処理
  – コンバインされたパケットの転送

☆ Ocean-RW は `R` と `W` の数が非常に多く，“-O1” でしかコンパイルできなかった．

ADSM ではブロックサイズはページサイズ固定の 4 KB であり，UDSM では先行研究の結果 22) から，1 KB に設定した．また，すべての実験において，手続き間別名解析による共有アクセスの検知を行った．

5.3 並列プログラムの 1 台の実行における最適化の効果

表 2 に問題サイズと逐次実行時間と，RCOP が自動生成した並列プログラムの 1 台の実行時間を示す．“FOW” は fetch-on-write の枠組で共有アクセス集合を使用した最適化を行ったもの，“w/o FOW” は fetch-on-write を避ける枠組で共有アクセス集合を使用した最適化を行ったものを示している．1 台における並列実行ではキャッシュミスは発生しないので，ユーザレベルのコヒーレンス管理コードの命令オーバヘッドのみが加算される．ADSM，UDSM 両機構において，共有アクセス集合を使用した最適化により命令オーバヘッドが低く抑えられることが分かる．

UDSM では，fetch-on-write を避けることにより read commitment の数そのものが削減されるので，LU-Contig や Radix や Volrend では 5∼28%実行時間が改善される．Water-NS や Raytrace では，read commitment が同期プリミティブと同様な意味を持つことにより，write commitment を一括して発行できない箇所があり，逆に実行時間が数%増大する．

ADSM では，読み書きともにユーザコードで実現す

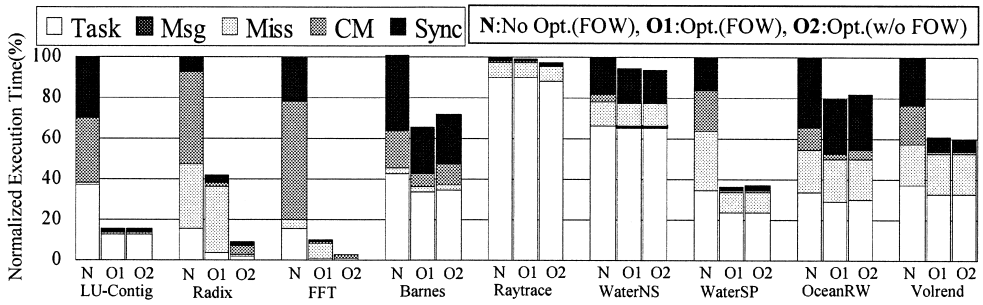


図 11 ADSM における最適化の効果（16 台の実行）
Fig. 11 Effects of compiler optimization for ADSM (executed on 16 nodes).

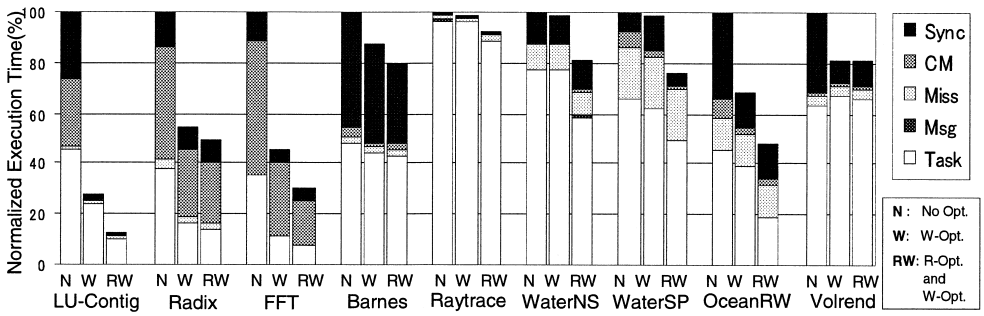


図 12 UDSM における最適化の効果（16 台の実行）
Fig. 12 Effects of compiler optimization for UDSM (executed on 16 nodes).

る UDSM に比べて，書き込みのみユーザコードで実現するため低オーバヘッドになる．Fetch-on-write 方式だと，オーバヘッドは 3%以下と低く押さえられる．Fetch-on-write を避けることにより，Radix や FFT や Ocean-RW ではオーバヘッドが 14〜20%発生する．これは fetch-on-write をしないためのコピーのオーバヘッドが主な原因である．このオーバヘッドは台数が増えるにつれて各ノードの仕事（書き込む量）が減るために，減少する．

5.4 並列プログラムの 16 台実行における最適化の効果

5.4.1 ADSM

図 11 が ADSM の 16 台実行における最適化の効果である．各棒グラフ群の左が fetch-on-write の枠組で共有アクセス集合による最適化を行わなかったもの，真中が fetch-on-write の枠組で共有アクセス集合による write-commitment の最適化を行ったもの，右が fetch-on-write を避ける枠組みで共有アクセス集合による write-commitment の最適化を行ったものを示している．“Sync” は同期の待ち時間，“CM” は write commitment の実行時間，“Miss” はキャッシュミスの待ち時間，“Msg” はリクエストメッセージを処理する時間，“Task” はプログラム本来の計算時間を示す．

SAURC プロトコルでは，write commitment をまとめる最適化が直接，通信の最適化につながる．したがって，“CM” だけでなく，“Sync” も削減される．Water-SP と Volrend においては “Miss” も短縮される．また，write commitment をまとめる最適化により，計算の局所性が向上し，“Task” そのものも削減される．Raytrace，Water-NS は，もともと “Task” に対する共有書き込みの割合が低いので最適化の効果は数%にとどまる．

Radix や FFT では，3.5 節で例にあげた「書かれるだけで，読み出されないブロック」が存在するので，fetch-on-write を避けるコード生成によりキャッシュミスの通信トラフィックが回避できて（“CM” が削減されて）効果的である．Barnes や Ocean-RW では逆に，fetch-on-write しないコード生成による局所領域から共有領域へのコピーのオーバヘッドのために，若干性能が落ちる．

5.4.2 UDSM

図 12 が UDSM の 16 台実行における最適化の効果である．各棒グラフ群の左が fetch-on-write を避ける枠組でコード生成したもの，真中が左に加えて write

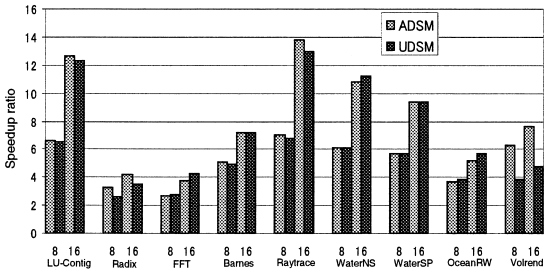


図 13 8 台と 16 台における台数効果
Fig. 13 Speedup ratio on 8 and 16 nodes.

commitment の共有アクセス集合による最適化を行ったもの，右が真中に加えて read commitment の共有アクセス集合による最適化を行ったものを示している．

“Sync” 等の意味は 5.4.1 節と同様である．write commitment の最適化の効果は ADSM の場合と同様である．UDSM の read commitment の実行時間は “Task” に含まれる☆．したがって，read commitment の最適化により，“Task” 時間が大きく削減される．特に，緩和された条件でのループレベルの最適化により，Water-NS，Water-SP，Ocean-RW で実行時間が削減される．Barnes では，ロードバランスの向上により，“Sync” も削減される．

5.5 台数効果

図 13 は本稿で提案してきたユーザレベル最適化をすべて行ったときの ADSM と UDSM における高速化率を表している．高速化率は，並列 1 台実行ではなく，逐次の実行時間を基準に計算している．**表 3** は 16 台実行の実行時間のブレイクダウンである．

LU-Contig，Raytrace，Water-NS，Water-SP は高い高速化率を示している．FFT と Radix は，通信量/計算量の比が大きく，高い通信バンド幅を必要とするプログラムであり，計算機クラスタにおいては高性能を得ることは困難である[9),13)]．にもかかわらず，fetch-on-write の除去により無駄なキャッシュミスが削減し，両機構において 16 台で 4 倍近くの台数効果を得ている．Barnes，Raytrace，Volrend では，共有データ参照の不規則性により，read commitment のオーバヘッドが大きくなり，ADSM の方が高速である．逆に，Water-SP，Water-NS，Ocean-RW では，ブロックサイズが 1KB である UDSM の方が，キャッシュミス時の無駄なデータ転送がない分高速である．

☆ read commitment の数が非常に多いので，時間測定のオーバヘッドが無視できなくなるから，read commitment の実行時間の測定は行っていない．

表 3 16 ノードの実行における平均実行時間のブレイクダウン
Table 3 Average time breakdowns(secs) for 16-node execution.

program	scheme	Sync	CM	Miss	Task
LU-Contig	ADSM	3.940 s	0.030 s	1.829 s	28.524 s
	UDSM	4.604 s	0.138 s	2.319 s	28.308 s
Radix	ADSM	0.298 s	0.779 s	0.088 s	0.377 s
	UDSM	0.325 s	0.897 s	0.112 s	0.511 s
FFT	ADSM	0.655 s	3.206 s	0.000 s	0.971 s
	UDSM	0.620 s	2.600 s	0.000 s	1.026 s
Barnes	ADSM	4.308 s	0.186 s	0.344 s	4.374 s
	UDSM	3.600 s	0.313 s	0.327 s	5.016 s
Raytrace	ADSM	0.172 s	0.028 s	0.960 s	11.251 s
	UDSM	0.135 s	0.034 s	0.273 s	12.734 s
Water-NS	ADSM	7.509 s	0.059 s	5.085 s	29.993 s
	UDSM	6.102 s	0.081 s	4.950 s	30.207 s
Water-SP	ADSM	0.383 s	0.110 s	1.589 s	3.568 s
	UDSM	0.357 s	0.162 s	1.521 s	3.641 s
Ocean-RW	ADSM	1.337 s	0.231 s	0.994 s	1.450 s
	UDSM	1.067 s	0.150 s	0.975 s	1.507 s
Volrend	ADSM	0.050 s	0.004	0.177 s	0.295 s
	UDSM	0.050 s	0.003	0.179 s	0.360 s

6. 関連研究

6.1 手続き間データフロー解析

近年の計算速度の向上によって，実際の問題に対して手続き間解析が使われるようになった．たとえば，手続き間 points-to 解析[8),24)] や，粗粒度並列性の抽出や array privatization に使用される手続き間アレイ・データフロー解析[10)] や，inspector/executor 方式のスケジューラーや通信コードの配置の最適化に使用される手続き間の部分冗長性の除去（IPRE[1)]）等である．これまで，こうした手続き間解析が明示的に並列な共有メモリプログラムに対して使用されてこなかった．RCOP は手続き間 points-to 解析を共有アクセスを検知するのに実際に使用している．RCOP の冗長性削除のデータフロー方程式は，部分冗長性を計算しないために単一方向問題の組み合わせに帰着できるので，IPRE のそれに比べて簡潔である．RCOP がデータフロー方程式の束として不等式系を使用するのは，手続き間アレイ・データフロー解析からきている．我々がこれまで作成し，使用してきたコンパイラ[21),30)] は RCOP のプロトタイプであり，手続き間別名解析は行っていたものの，手続き内で coalescing する変換しか行ってこなかった．

6.2 ソフトウェア DSM を支援する最適化コンパイラ

最適化コンパイラが明示的に並列な共有メモリプログラムを直接解析してソフトウェア DSM を支援するものとして，Shasta コンパイラ，TreadMarks を支援するコンパイラ，Omni コンパイラ等がある．

Shasta[23] は完全なユーザレベル方式で細粒度共有メモリアクセスを提供する機構である．そのコンパイラは明示的に並列な共有メモリプログラムのバイナリを入力とし，flag value techniques，batching といった最適化を実行しているが，ループレベル，手続き間の解析は行っていない．その結果，高バンド幅低レイテンシのネットワークが必要になる．

Dwarkadas ら[6] は，コンパイラが明示的に並列に書かれた Fortran のソースを解析することで，トラップベースのソフトウェア DSM，TreadMarks[14] の性能を改善している．Regular section analysis を用いて通信の一括化，無駄なコヒーレンス管理の除去を行っている．しかし，手続き呼び出し，条件節が解析の障害になっており，誘導変数のみが解析の対象であり，連続変数を用いた最適化は行っていない．また，共有変数はページアラインを前提とし，別名関係は考慮されていない．

Omni コンパイラ[26] は OpenMP 用の Compiler で，SMP クラスタ向けの最適化を行っていて，スレッド間のデータフロー解析を行っている．

7. ま と め

汎用分散環境において共有メモリプログラムを効率よく実行するため，ユーザレベル最適化方式を提案し，実装し，SPLASH-2 を使用した実験により性能評価を行った．

高性能を得るためには，通信のバンド幅を生かし，キャッシュ管理のオーバヘッドを削減する最適化コンパイラの支援が必須であることが示された．手続き間区間解析の枠組で冗長性削除のデータフロー方程式を解くことで，共有メモリアクセスのサマリを算出する本方式は，データ参照が規則的なアプリケーションに対してだけでなく，データ参照が不規則なアプリケーションに対しても効果的であることが確認された．すべてユーザレベルで実現する UDSM の場合であっても，キャッシュコヒーレンス管理のオーバヘッドは本来の計算時間の 0.99〜21%に押さえられる．

次に，false sharing を避けるプログラミングが必要になる．ただし，fetch-on-write のキャッシュミスはコンパイラ/ランタイムの支援により削減可能である．

今回の ADSM と UDSM の性能評価を通じて ADSM のボトルネックは，ページサイズ固定による無駄なデータ転送といったネットワーク，つまり，プロセッサの外部にあり，UDSM のボトルネックは read commitment の命令オーバヘッドであり，プロセッサ内部にあるということを確認した．

ユーザレベル最適化，メモリベース通信機構，ページ管理機構の活用により，汎用のネットワークで接続された計算機クラスタにおける分散共有メモリ機構は実用レベルに到達したといって過言ではない．また，本研究により，汎用のハードウェアのみによるスケーラブルな共有メモリ型並列計算機の可能性が開かれた．

謝辞 本研究は情報処理振興事業会（IPA）が実施している独創的情報技術育成事業の一環として行った．

参 考 文 献

1) Agrawal, G., Saltz, J. and Das, R.: Interprocedural Partial Redundancy Elimination and its Application to Distributed Memory Compilation, *Proc. '95 Conf. on PLDI* (June 1995).
2) Boyle, J., Butler, R., Disz, T., Glickfeld, B., Lusk, E., Overbeek, R., Patterson, J. and Stevens, R.: *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc. (1987).
3) Burke, M.: An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis, *ACM Transactions on Programming Languages and Systems*, Vol.12, No.3, pp.341–395 (1990).
4) Chow, F.C.: Minimizing Register Usage Penalty at Procedure Calls, *Proc. of '88 Conf. on PLDI* (June 1988).
5) Cocke, J.: Global Common Subexpression Elimination, *Proc. Symposium on Compiler Optimization, SIGPLAN Notices*, Vol.5, No.7, pp.20–24 (1970). .
6) Dwarkadas, S., Cox, A.L. and Zwaenepoel, W.: An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, *Proc. ASPLOS-VII* (Oct. 1996).
7) Dwarkadas, S., Gharachorloo, K., Kontothanassis, L., Scales, D.J., Scott, M.L. and Stets, R.: Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory, *Proc.5th HPCA* (Jan. 1999).
8) Emami, M., Ghiya, R. and Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers, *Proc. '94 Conf. on PLDI* (June 1994).

9) Erlichson, A., Nuckolls, N., Chesson, G. and Hennessy, J.: SoftFLASH: Analyzing the Performance of Clusterd Distributed Virtual Shared Memory, *Proc. ASPLOS-VII* (Oct. 1996).
10) Hall, M.W., Murphy, B.R., Amarasinghe, S.P., Liao, S. and Lam, M.S.: Interprocedural Analysis for Parallelization, *Proc.8th Int. Workshop on LCPC*, Springer-Verlag (Aug. 1995).
11) Iftode, L., Dubnicki, C., Felten, E.W. and Li, K.: Improving Release-Consistent Shared Virtual Memory using Automatic Update, *Proc. 2nd HPCA* (Feb. 1996).
12) Inagaki, T., Niwa, J., Matsumoto, T. and Hiraki, K.: Supporting Software Distributed Shared Memory with a Optimizing Compiler, *Proc. 1998 ICPP* (Aug. 1998).
13) Jiang, D., Shan, H. and Singh, J.P.: Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors, *Proc. 6th ACM SIGPLAN Symp. on PPOPP* (June 1997).
14) Keleher, P., Cox, A.L., Dwarkadas, S. and Zwaenepoel, W.: Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems, *Proc. Winter 1994 USENIX Conf.* (Jan. 1994).
15) Keleher, P., Cox, A.L. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th ISCA* (May 1992).
16) Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. 1988 ICPP* (Aug. 1988).
17) Matsumoto, T. and Hiraki, K.: Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory, *Proc. 1997 International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, Los Alamitos, CA, IEEE Computer Society (1998).
18) Matsumoto, T., Niwa, J. and Hiraki, K.: Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities, *Proc. 1998 PDPTA*, Vol.2 (July 1998).
19) Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
20) Niwa, J.: *Study on Optimizing Compilers to Support Software Distributed Shared Memory Systems*, PhD Thesis, Department of Information Science, Tokyo University (2000).
21) Niwa, J., Inagaki, T., Matsumoto, T. and Hiraki, K.: Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory, *Proc. 1997 ISPAN* (Dec. 1997).
22) Niwa, J., Matsumoto, T. and Hiraki, K.: Comparative Study of Page-based and Segment-based Software DSM through Compiler Optimization, *Proc. 2000 International Conference on Supercomputing* (May 2000).
23) Scales, D.J., Gharachorloo, K. and Thekkath, C.A.: Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, *Proc. ASPLOS-VII* (Oct. 1996).
24) Wilson, R.P. and Lam, M.S.: Efficient Context-Sensitive Pointer Analysis for C Programs, *Proc. '95 Conf. on PLDI* (June 1995).
25) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd ISCA* (June 1995).
26) 佐藤茂久, 草野和寛, 田中良夫, 佐藤三久：SMP クラスタ向けの OpenMP コンパイラ, 情報処理学会研究報告, 99-HPC-77 (Aug. 1999).
27) 松本 尚, 駒嵐丈人, 渦原 茂, 竹岡尚三, 平木 敬：汎用超並列オペレーティングシステム SSS-CORE—ワークステーションクラスタにおける実現, 情報処理学会研究報告, 96-OS-73 (Aug. 1996).
28) 松本 尚, 駒嵐丈人, 渦原 茂, 平木 敬：メモリベース通信による非対称分散共有メモリ, コンピュータシステムシンポジウム論文集 (Nov.1996).
29) 松本 尚, 平木 敬：100BaseTX によるメモリベース通信の性能評価, コンピュータシステムシンポジウム論文集 (Nov. 1997).
30) 丹羽純平, 稲垣達氏, 松本 尚, 平木 敬：非対称分散共有メモリ上における最適化コンパイル技法の評価, 情報処理学会論文誌, Vol.39, No.6, pp.1729–1737 (1998).

(平成 12 年 9 月 18 日受付)
(平成 13 年 2 月 1 日採録)

丹羽 純平

1972 年生．2000 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．2000 年学術振興会特別研究員．並列化/最適化コンパイラに関する研究に従事．ほかに並列計算機アーキテクチャ，並列分散オペレーティングシステムに興味を持つ．

松本 尚（正会員）

1962 年生．1985 年東京大学工学部計数工学科卒業．1987 年大阪市立大学大学院理学研究科物理学専攻修士課程修了．日本アイ・ビー・エム（株）東京基礎研究所研究員を経て，1991 年 11 月より東京大学大学院理学系研究科情報科学専攻助手．並列計算機アーキテクチャ，並列分散オペレーティングシステム，最適化コンパイラに関する研究に従事．ほかに数値計算による制約解消系，グラフィックス，ニューラルネットワーク等に興味を持つ．電子情報通信学会，日本ソフトウェア科学会，ACM 各会員．

平木 敬（正会員）

1976 年東大理学部物理学科卒業．1982 年同大学大学院理学系研究科物理学専攻博士課程修了．理学博士．1982 年通商産業省工業技術院電子技術総合研究所入所．1988 年より 2 年間 IBM 社 T.J.Watson 研究センタ客員研究員．1990 年より東京大学理学部情報科学科（現在大学院理学系研究科情報科学専攻）に勤務．現在，超並列アーキテクチャ，超並列超分散計算，並列オペレーティングシステム，ネットワークアーキテクチャ等の高速計算システムの研究に従事．日本ソフトウェア科学会会員．