ハイパフォーマンス
コンピューティング 84－9
（2000．12．8）

ソフトウェアDSMにおいて fetch-on-writeによる通信トラフィックを削減する手法

丹羽純平[†,††] 松本尚[†,†††] 平木敬[†]

コンパイラが支援するソフトウェアDSMでは、最適化コンパイラがソースのデータアクセスを解析することで、書き込まれるだけで読み出されないブロックを検知できる。上記のようなブロックに対しては一貫性維持操作を省略することが可能になり、無駄な通信が削減される。本手法を最適化コンパイラRCOPに実装し、その有効性をSS20クラスタ上でSPLASH-2を用いた実験により確認した。

Methods of Reducing Cache-miss Traffic at Fetch-on-write in Software DSM

JUMPEI NIWA ,[†,††] TAKASHI MATSUMOTO [†,†††] and KEI HIRAKI[†]

In the compiler-assisted software DSM scheme, an optimizing compiler can analyze data access patterns and eliminate coherence management operations for blocks whose data are written but not read. As a result, the run-time system need not fetch data to update the blocks. We have implemented this optimizing technique in the optimizing compiler called "Remote Communication Optimizer" (RCOP). The experimental results using the SPLASH-2 benchmark suite on the SS20 cluster show that this approach is effective.

1. はじめに

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、効率良く共有メモリ型並列プログラムを実行するためには、ソフトウェアDSM（分散共有メモリ）[8),9)]と呼ばれる、ソフトウェアで共有データをキャッシングする機構が必要になる。

分散共有メモリであるから、

1. 共有データを読み出したときには一般にコピーが作られ、
2. 共有データに書き込んだときにはコンシステンシ管理が行なわれる。

書き込み時のコンシステンシ管理は以下のものが考えられる

- 現存するコピーを全て無効化(invalidate)する
- 現存するコピーを全て更新(update)する(全方向更新)
- homeのみ更新して、home以外のコピーは全て無効化する(一方向更新)[3)]

コンシステンシ管理操作をいつするか？という方針には選択肢があるものの、緩和されたメモリモデルを採用する場合[8)]、書き込み→release→acquire→読み出しという順序の間に伝わっていればいいのである。

特殊な通信同期ハードウェアの存在を仮定しない場合、コヒーレンス管理操作(1.,2.)をトラップハンドラで行うのが既存の方式(トラップベースの方式)である[☆]。つまり、TLB/MMUを利用することで、コヒーレンス管理の必要な共有メモリアクセスを捕まえる。この方式の特徴として、共有アクセスは通常のメモリアクセス(load/store命令)と同様にコードが生成されるという点が挙げられる(図1)。つまり、上記のコヒーレンス管理操作がユーザからは見えないところで実行される。また、共有データへstoreする前にコンシステンシ管理操作を行う必要があるため、緩和されたメモリモデルを実現する場合には、書き込まれた内容の回収が必要になり[7)]、高オーバヘッドにつながる。

2. コンパイラが支援するソフトウェアDSM

コヒーレンス管理操作をユーザレベルで明示的に行なう方式はプログラムの情報を元に最適化をかけることが可能である。プログラマがコヒーレンス管理コードを直接挿入するシステム(CRL[6)])と比較して、UDSM(User-level Distributed Shared Memory)[10)]では最適化コンパイラが共有メモリアクセスを静的に検出し、最適なコヒーレンス管理コードを自動的に挿入してくれるので(図1)、プログラマの負担が少なくて済む。

また、トラップベースのシステムとUDSMのハイブリッドであるADSM (Asymmetric Distributed Shared Memory)[10),16)]では，読み出しミスはTLB/MMUにより動的に検知されるが、書き込みは最適化コンパイラにより静的に検知され、コンシステンシ管理コードが自動的に挿入される(図1)。

図1に示してあるように、トラップベースの方式では、ソースがそのままプラットフォームのバックエンドコンパイラに渡

† 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

☆ 緩和されたメモリモデルを実現する場合には更に同期プリミティブを使用する[7),9)]。

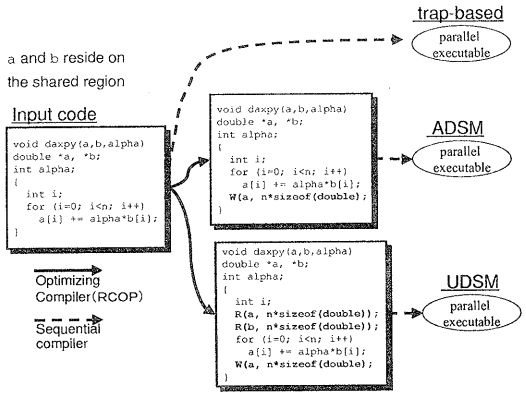

図1 従来のトラップベースの方式とコンパイラベースの方式の比較

される。コンパイラベースの方式では、最適化コンパイラがソースを解析／変換して、DSMのコヒーレンス管理コードを挿入する。UDSMでは読み出しに対してチェックコードRが挿入され、書き込みに対してコンシステンシ管理コードWが挿入される。ADSMでは、書き込みに対し、コンシステンシ管理コードが挿入される。緩和されたメモリモデルを採用する場合、チェック(コンシステンシ管理)は実際のload(store)命令よりも前(後)でまとめて行われる。然る後に、バックエンドコンパイラに渡され、実行コードが生成される。

3. コンパイラの支援の必要性

トラップベースの方式は、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は存在しないので、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なう。無効なブロック(ページ)に書き込もうとする場合、まず、ブロック全体を最新のものに更新した後に、書き込み時のコンシステンシ管理操作を行い、storeを実行する。

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見受けられる。double bufferingしてradix sortする場合(図2)には、一方の配列の一部を連続して読み出し、他方の配列全体に渡って短冊状の書き込みを行い、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3)も同様のことが発生する。

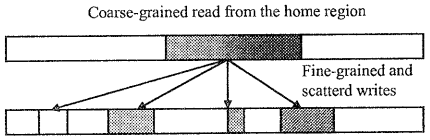

図2 Permutation phase.

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、全ての書き込みに対してキャッシュミスが発生し、無駄なデータ転送が発生する。トラップベースの方式では「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方式では、同期で囲まれたinterval[8]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「自分が書き潰した部分からしか読み出されないブロック」を検知できる。そのような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、無駄な通信を削減することが可能になる。

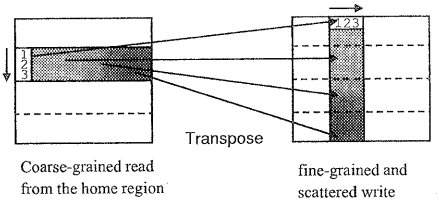

図3 Transpose phase.

ところが、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は困難である。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。我々はこの情報を元に、低オーバヘッドでfetch-on-writeを避ける方式を提案する。

4. Fetch-on-writeを避けるコード生成法

4.1 UDSM

読み書き共にコンパイラが挿入するコヒーレンス管理コードで実現される場合には、全ての書き込みに対して、書き込みの前のブロックを更新操作を省略する。無効なブロックに対してstoreを行い、そのコンシステンシ管理コードを発行する前に同一ブロックへのチェックコードが発行される場合、ブロック全体が更新されてしまい、先の最新のstoreの結果が失なわれる可能性がある。したがって、チェックコードが実行される前にコンシステンシ管理コードを実行する必要がある。つまり、共有書き込みの発行にとって、チェックコードの挿入場所*が、同期プリミティブと同様な意味を持つ。

1. 手続き間別名解析の手法で[13]共有読み出し、書き込みを検出する[4),12)]。
2. 読み出しに対しては、仮りにチェックコードRを挿入する。しかし、書き込みに対しては、チェックコードを挿入しない。
3. チェックコードRに対して、区間解析の枠組で、手続き間冗長性削除のデータフロー方程式を解いて、実際の最適な挿入場所を決める。詳細は文献11), 12)を参照されたい。
4. 書き込みに対して、管理コードWを仮りに挿入する。
5. 管理コードWに対して、3と同様に冗長性削除のデータフロー方程式を解いて実際の最適な挿入場所を決める[11),12)]。

上述のように、3.のパスで求めたチェックコードの挿入場所が同期プリミティブと同様な意味を持つものとみなす。

もちろん、管理コードが発行されるブロックと同一ブロックに対するチェックコードのみが同期と同じ意味を持つが、チェックコードと管理コード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、全てのチェックコードを同期プリミティブと同様にみなす。

また、書き込まれた値をそのまま読むような場合にはチェックコードそのものが省略される。

例えば、以下のようなコード

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

* 読み出しそのものの場所ではない

```
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。

R (S, N * sizeof(double)) はSから N * sizeof(double) バイト分の読み出しがあったことを表しており、Rは、このデータが乗っているブロックが無効な場合には、キャッシュミスハンドラを起動し、ブロック全体を最新の状態に更新するコードである。

W (S, N * sizeof(double)) はSから N * sizeof(double) バイト分の書き込みがあったことを表しており、これに必要なコンシステンシ管理操作を手続きWで実行する。Sに書き込む前に、ブロックは最新の状態になり、Wが発行された時には、最新のデータが既にSに書かれている。

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。

4.2 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を省略できる必要がある。UDSMでは書き込み前のチェックコードを省略することで対処できる。しかし、ADSMでは、ブロックミスは非明示的にTLB/MMU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。従って、書き込みのコードそのものを変換する必要が生じる。

例えば、4.1節の例のようなコードは、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。書き込み時のコンシステンシ管理コードの引数を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。

AURC[3]のような一方向の更新方式を採用する場合、このコンシステンシ管理コード（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時のみ、_W_Sの内容をSにコピーするといった操作を行う。このコピーは、自ノードが同期で囲まれたinterval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時に、正しいデータを読むために必要である。

自分が有効なブロックに対して書き込みを行う時はstoreの量が2倍になるため、全ての書き込みをW'として発行することは高オーバヘッドにつながる。そこで、各intervalに対して、どの読み出し部分とも重ならない書き込みを検出する。検出された書き込みに対しては、書き込みそのものを変換してfetch-on-write しない書き込みとして発行する方針を取る。それ以外の書き込みに関してはfetch-on-writeする書き込みとして発行する。

1. UDSMの場合と同様に共有読み出し、書き込みを検出する。
2. 書き込みに対して、コンシステンシ管理コードを仮りに挿入し、区間解析の枠組で、手続き間冗長性削除のデータフロー方程式を解くことで、実際の最適な挿入場所を決める。
   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[13]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はブロックbにおけるバイト単位の刻み幅（stride）、fは$0 \sim s-1$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できる。
3. 最後にfetch-on-writeする／しないに分けて、2.で求めた個所に以下の手順でコンシステンシ管理コードを挿入する。
   fetch-on-write しない場合には共有アクセスのサマリを表す共有アクセス集合(a, s, C)[☆4),12]に対応する元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時（サイズsが基本サイズに等しい時）、
     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。コンシステンシ管理コードの第三引数は&_W_aになる。
   - そうでない時、つまり、coalescing等で共有アクセス集合がマージされている時（サイズsが基本サイズより大きい時）、
     Stの各文の左辺の中で共有アクセス集合の第一要素aに該当する部分をローカルなクローン配列変数（_W_a）に置き替える。コンシステンシ管理コードの第三引数は_W_aになる。
     ```
     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、クローン変数を経由せず、書き込みそのもの(St)を省略する。その場合のコンシステンシ管理コードの第三引数は、以下のようにコピーされる配列の先頭アドレスになる。
     ```
     W' (S, N * sizeof(double), A);
     ```
   - クローン変数（_W_a）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

☆ この場合、aとsはWの第一、第二引数に相当し、CはWを含むループの制約を表す不等式に相当する。

わせて動的に確保する（malloc）。コンシステンシ管理コードが発行されると、対応するクローン領域は不要になるので、資源節約の点から、再利用可能なものは再利用する。

5. 性能評価

5.1 アプリケーション

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[14]から，9個のアプリケーションを使用して評価を行った(LU decomposition (LU-Contig), Radix, FFT, Barnes, Raytrace, Water-Nsquared (Water-NS), Water-Spatial (Water-SP), Ocean (Ocean-RW) と Volume rendering (Volrend))。その内の5個に関しては、他の多くのソフトウェアDSMシステムが行っているようにソースを変更した[2),4),5)]。具体的な変更点は表1に示されている。

表1 ベンチマークの修正部分

Program	Modification
LU-Contig	owner of block $(i,j) \leftrightarrow$ owner of block (j,i)
FFT	sender-initiated `Transpose`[4)]
Raytrace	elimination of unused lock-operation for ray ID[4)]
Ocean	rowwise partition (Ocean-RW)[5)]
Barnes	sequential tree-construction[2)]

5.2 実験環境

実験環境は以下のようになる

- コンパイラ
  今回提案した、fetch-on-writeを避ける方式を手続き間最適化コンパイラRCOP[10),12)]に実装した。RCOPはLRC[8)]モデルに基づいた共有メモリ並列プログラム☆を解析し、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、効率の良い、DSM用のキャッシュコヒーレンス管理コードを明示的に含むC言語プログラムに変換する。バックエンドコンパイラはgcc-2.7.2を使用した。
- ノード
  Axil 320 model8.1.1（Sun SS20互換機, 85MHz, SuperSPARCII×1）であり、Fast Ethernet SBus Adapter2.0を追加している。
- ネットワーク
  100BASE-TXのスイッチングハブ（3Com SuperStackII Switch 3900）でFast Ethernet接続されている。
- OS
  汎用超並列オペレーティングシステムSSS-CORE[15)]Ver.1.2を使用した。SSS-COREは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)[16)]を提供している。MBCF / 100BASE-TXの性能はピークバンド幅が11.93（MBytes/sec）でラウンドトリップレイテンシは49（μsec）である。
- ランタイムシステム
  明示的な通信コードによって、一方向の更新型プロトコルを実現するSAURC[16)]プロトコルを採用している。なお、細粒度の書き込みはコンバイニングしてhomeに転送している。各ブロックに関して、どこのintervalでどのプロセッサに書き込まれたかどうかの情報は管理せず、最新のバリアから現在にいたるまで単純に書き込みされたかどうかの情報のみを管理する。

☆ PARCMACSで拡張されたC言語で書かれている。

リモートリクエストは割り込みによって検知され、ユーザレベルハンドラが呼ばれて処理をする。

ADSMではブロックサイズはページサイズ固定の4KBであり、UDSMでは先行研究の結果12)から、1KBに設定した。

5.3 1台の性能

表2は問題サイズと逐次実行時間と、並列プログラムの一台の実行時間を示している。並列プログラムはRCOPが最適化を行い、自動生成したものである。"FOW"は従来通りfetch-on-writeをする方式、"w/o FOW"はfetch-on-writeを避ける本方式でコード生成したものを示す。

並列プログラム一台の実行はコヒーレンス管理の実行が含まれているため、逐次より遅くなるものの、最適化によりオーバヘッドは低く押さえられる。UDSMでは、fetch-on-writeを避けることにより、チェックコードの数そのものが削減されるので、実行時間は改善される。Water-NSやRaytraceでは、読み出しが同期プリミティブと同様な意味を持つことにより、コードをうまくマージできない個所があり、逆に実行時間が数%増大する。

ADSMでは、読み書き共にユーザコードで実現する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。従来方式だと、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、FFT、Ocean-RWはオーバヘッドが20%近く発生する。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が、これ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 16台の性能

図4がADSMの16台における最適化の効果である。各棒グラフ群の左が全く最適化を行わないもの、真中が、これまでのコンパイル時、実行時最適化を施したもの、右が、真中の最適化に加えて、fetch-on-writeしないコード生成を施したものである。"Sync"は同期の待ち時間、"CM"はキャッシュコンシステンシ管理コードの処理時間、"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。

RadixやFFTでは、3節の例に挙げたようなアクセスが存在し、fetch-on-writeしないコード生成により、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り、局所領域から共有領域へのコピー(4節の例でいうところの`_W_S`から`S`へコピー)のオーバヘッドが加算され、若干性能が落ちる。

UDSMの16台の性能は紙面の都合により割愛するが、ADSMのように、fetch-on-writeを避けることで性能が落ちるようなアプリケーションはなかった。

5.5 台数効果

逐次プログラムの実行時間を元に、UDSMとADSMの台数効果を測定した。今までは、Radix, FFTはUDSMの方が高速だったが、今回のコード生成により、両者の差異はほとんど見られなくなった。Radixはhomeとなる領域への粗粒度の読み出しが存在するため、ブロックサイズが大きく、チェックのオーバヘッドのないADSMの方が速い。FFTでは、「書き込まれるだけで読み出されない」ブロックが多数存在し、その中で自分が有効であるものの数が多い。その結果、ADSMのfetch-on-writeをしないためのコピーのオーバヘッドが大きくなり、UDSMより遅くなる。

6. 関連研究

Dwarkadas達[1)]は、コンパイラが明示的に並列に書かれた

表2　問題サイズと逐次実行時間と最適化された並列1台実行時間 (単位は秒、() の中は %).

プログラム	問題サイズ	逐次実行	ADSM 最適化並列一台実行		UDSM 最適化並列一台実行	
			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)

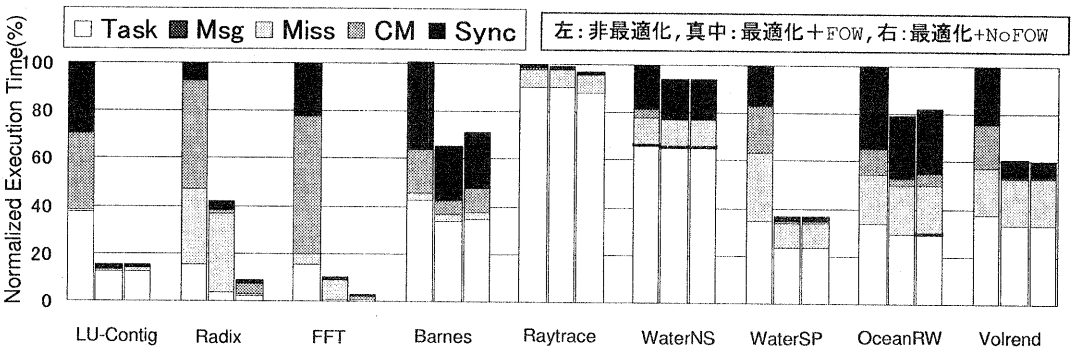

図4　ADSM における最適化の効果 (16 台の実行)

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[7] の性能を改善している。Regular section analysis を使用して、無駄なコンシステンシ管理の除去 (差分の生成、適用) が可能ならば、ランタイムにそれを知らせるルーチンを挿入する。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、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。本論文で述べる方式は、手続き間別名解析を使用しており、異なる共有変数が同一ブロックに乗っている場合でも問題ない。

7. 最　後　に

トラップベースのソフトウェア DSM ではアプリケーションの特性を利用した最適化をかけられないという欠点がある。コンパイラの支援がない場合には fetch-on-write しないことによる最適化を行うことが不可能である。

もちろん、ブロックサイズが (ハードウェア) キャッシュのラインサイズのように小さければ fetch-on-write によるオーバヘッドはある程度は軽減される☆。しかし、ブロックサイズを小さくすることは、トラップベースのシステムでは困難であり、コンパイラベースのシステムにおいても、(ソフトウェア) キャッシュミス時に通信のバンド幅を稼ぐことが困難になり、管理コストが増大する。文献 12) では、ブロックサイズが 64B の 16 台における実行時間は、1KB のそれと比較して最悪 2 倍遅くなることが報告されている。

動的に確保される共有変数がどのようにブロックに配置されるか？はコンパイル時には分からないので、「書き込まれるだけで、読み出されないブロック」や「自分が書き潰した部分からしか読み出されないブロック」の正確な検知は事実上困難である。しかし、メモリアクセスが重ならないかどうかは現在の手続き間ポインタ解析の技術で判定できるので、その結果を利用することで、fetch-on-write をしないようなコード生成を導入することが可能になった。ハイブリッド方式である ADSM においても、無駄な通信を抑制し、高速化を達成することができた。データの home の指定情報をコンパイル時に利用できるようになると、更にオーバヘッドを削減させることが可能になると思われる。

☆ 無駄なデータを入手する可能性が低下する

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なった。

参　考　文　献

1) DWARKADAS,S., COX,A.L.and ZWAENEPOEL,W. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, Proc. of ASPLOS-VII (Oct. 1996).

2) 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. of the 5th HPCA (Jan. 1999).

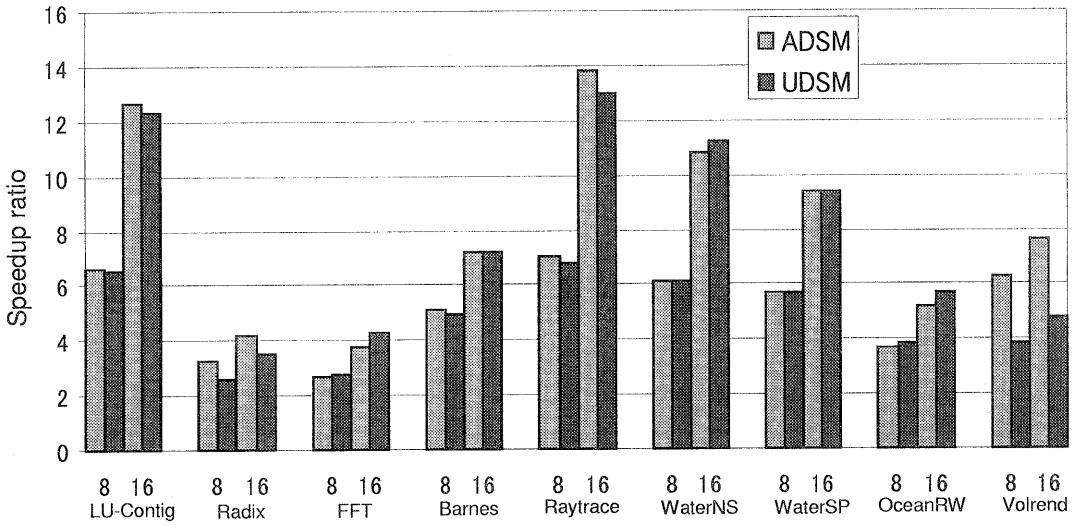

図5 ADSM と UDSM における台数効果

3) IFTODE, L., DUBNICKI, C., FELTEN, E. W. and LI, K. Improving Release-Consistent Shared Virtual Memory using Automatic Update, Proc. of the 2nd HPCA (Feb. 1996).

4) INAGAKI, T., NIWA, J., MATSUMOTO, T. and HIRAKI, K. Supporting Software Distributed Shared Memory with a Optimizing Compiler, Proc. of the 1998 ICPP (Aug. 1998).

5) JIANG, D., SHAN, H. and SINGH, J. P. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors, Proc. of the 6th ACM SIGPLAN Symp. on PPOPP (June 1997).

6) JOHNSON, K. L., KAASHOEK, M. F. and WALLACH, D. A. CRL: High-Performance All-Software Distributed Shared Memory, Proc. of 15th ACM Symposium on Operating System Principles (Dec. 1995).

7) KELEHER, P., COX, A. L., DWARKADAS, S. and ZWAENEPOEL, W. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proc. of the Winter 1994 USENIX Conf. (Jan. 1994).

8) KELEHER, P., COX, A. L. and ZWAENEPOEL, W. Lazy Release Consistency for Software Distributed Shared Memory, Proc. of the 19th ISCA (May 1992).

9) LI, K. IVY: A Shared Virtual Memory System for Parallel Computing, Proc. of the 1988 ICPP (Aug. 1988).

10) MATSUMOTO, T., NIWA, J. and HIRAKI, K. Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities, Proc. of the 1998 PDPTA, Vol. 2 (July 1998).

11) NIWA, J. *Study on Optimizing Compilers to Support Software Distributed Shared Memory Systems*, PhD thesis, Department of Information Science, Tokyo University (2000).

12) NIWA, J., MATSUMOTO, T. and HIRAKI, K. Comparative Study of Page-based and Segment-based Software DSM through Compiler Optimization, Proc. of 2000 International Conference on Supercomputing (May 2000).

13) WILSON, R. P. and LAM, M. S. Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of '95 Conf. on PLDI (June 1995).

14) WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P. and GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations, Proc. of the 22nd ISCA (June 1995).

15) 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, 第 96-OS-73 巻 (Aug. 1996).

16) 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).