コンパイラが支援するソフトウェアDSMにおけるレイテンシ削減技法

丹羽 純平†

ソフトウェア DSM は，汎用の分散並列環境において，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．ソフトウェア DSM では，遠隔メモリアクセスのレイテンシの削減のために，遠隔ノードのデータを自ノードの局所メモリにキャッシュする．本稿は，ソフトウェア DSM において，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により，遠隔メモリアクセスのレイテンシをさらに削減する手法を提案する．すなわち，遠隔メモリアクセスのレイテンシを削減するための，プリフェッチを行うコンパイル技法を提案し，それを可能にするインタフェースを導入する．上記のコンパイル技法を最適化コンパイラ（RCOP）に実装し，ギガビットイーサ接続された汎用の PC クラスタ上にランタイムを構築した．PC クラスタにおいて，SPLASH-2 ベンチマークを用いた実験により，本最適化の効果を検証した．

Latency Tolerance Techniques for Compiler-assisted Software DSM System

Jumpei Niwa†

Software Distributed Shared Memory (S-DSM) provides shared address space at run-time and accepts a wide range of applications on parallel computer systems with commodity hardware. S-DSM caches remote data in the local memory in order to reduce remote-memory-access latency. This paper proposes the methods for further reducing the latency in S-DSM by utilizing an optimizing compiler that directly analyzes explicitly parallel shared-memory source programs. That is to say, this paper suggests the compiling techniques of issuing prefetch for remote-memory access and introduces the interface that enables prefetch mechanism. I have implemented the techniques in optimizing compiler, Remote Communication Optimizer :RCOP. I also have implemented the lightweight runtime systems on PC cluster connected with the Gigabit Ethernet (1000 BASE-T). The experimental results using the SPLASH-2 benchmark suite show that the latency tolerance techniques are effective.

1. はじめに

本研究の目的は，計算機クラスタのような，汎用のネットワークで接続され，固有の共有メモリ機構を持たない分散並列環境において，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（明示的に並列な共有メモリプログラム）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（Software Distributed Shared Memory: S-DSM）[4),5)] と呼ばれる，遠隔ノードのデータを自ノードのメモリにキャッシュする（ソフトウェアキャッシュ）機構が求められる．

従来の S-DSM [4),5)] では，実行コードの共有メモリアクセスはプロセッサの単純な load/store に変換されていることに固執している．しかしながら，現在，並列アプリケーションはソースコードで流通するのが主流であり，マシンが異なれば再コンパイルすることが当たり前である．バイナリコンパチビリティにこだわる必要性がないのであれば，共有メモリアクセスをコンパイラが自らキャッシュコヒーレンス管理コード（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行うコードで，場合によっては，他ノードと通信を行う）と通常のメモリアクセス（load/store）に分割してコードを生成するコンパイラ支援方式[8),16)] でもかまわないことになる．

はじめて遠隔ノードにあるデータにアクセスする場合には，キャッシュミスが発生し，データを持っているノードにキャッシュブロックリクエストが転送される．従来の S-DSM では，実際にアクセスが行われる

† 科学技術振興機構さきがけ研究 21「機能と構成」領域
PRESTO, Japan Science and Technology Agency

(load/store) ときになってはじめて，トラップハンドラによって上記の操作が行われる．しかも，データが到着するまでノードは待機するだけである．そこで，コンパイラ支援方式では，コンパイラが，上記の操作を行うキャッシュコヒーレンス管理コードを実際のメモリアクセスのできるだけ前で発行することによって，通信と計算をオーバラップさせることができるのではないかと考察した．

本稿では，以下の綱目を新規に提案し，それを実装し，実験によってその有効性を評価する．

- コンパイラが支援する S-DSM においてプリフェッチを可能にするインタフェース（HDSM）
- プリフェッチ挿入に対して手続き間部分冗長性削除の枠組みを適用
- 収益性解析によるプリフェッチのオーバヘッド削減
- 履歴を活用した低コストなプリフェッチ

2. プリフェッチを可能にするインタフェース

まず，コンパイラが支援する S-DSM の 2 個のインタフェースについて，プリフェッチに関連する事項を述べる．インタフェースの詳細は文献 8)，16) を参照されたい．次に，本稿で提案するプリフェッチを可能にするインタフェースについて述べる．

Asymmetric DSM（ADSM）[16] 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/ヒット判定はページ管理機構で実現される．したがって，実アクセスまでキャッシュのミス/ヒット判定が行われない．プリフェッチは不可能である．

一方，User-level DSM（UDSM）[8] では，キャッシュミス/ヒット判定はユーザレベルのコードで明示的に実現される．すなわち，実アクセスの前にキャッシュのミス/ヒット判定を行うことが可能であり，原理的にプリフェッチは可能である．

従来の実装では，コンパイラが “キャッシュミス/ヒット判定を行って，無効な場合にはキャッシュブロックリクエストを転送し，ブロックが到着するまで待機する” 同期式コードを挿入していた．もちろん，“キャッシュブロックリクエストを転送し，すぐに元のタスクに復帰する” 非同期式コードを挿入する方針は可能である．しかし，非同期式コードを挿入する方針を選択した場合には，実アクセスの前に，再度，キャッシュのミス/ヒット判定を行って，有効かどうか確認するコードが必要になる．したがって，非同期式コードの場合には，キャッシュのミス/ヒット判定回数が同期式コードの場合の 2 倍になり，高オーバヘッドを引き起こすと予想される．そのため，UDSM では非同期式コードは実装されてこなかった．

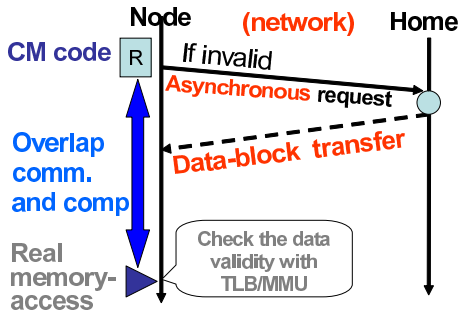


図 1 HDSM の共有読み出し時のランタイムの動作
Fig. 1 Behavior of shared-read operations in HDSM.

2.1 Hybrid DSM

ADSM と UDSM のハイブリッド方式：Hybrid DSM（HDSM）を用いると，低オーバヘッドなプリフェッチが可能になると考察した．

- 最適化の観点から，コンパイラが，キャッシュのミス/ヒット判定を明示的に行うユーザレベル非同期式コードを明示的に挿入する．
- 無効なキャッシュをアンマップすることで，読み出し時のコヒーレンスの維持は最終的にトラップハンドラが保証する（ただし，キャッシュのブロックサイズはページサイズとなる）．つまり，最新のデータが到着したかどうかの判定にページ管理機構を流用する．

図 1 が共有読み出し時の動作を簡潔に示している．Node がキャッシュのミス/ヒットを調査し（R），ミスした場合には Home にリクエストを転送する．そして，即座に元のタスクに復帰する．Home は Node にキャッシュブロックを転送する．Node のリクエストハンドラがそれを受け取ってブロックを更新し，実アクセス時には何事もおこらずに計算が続いていく．もちろん，データの転送が間に合わなかった場合には，実アクセス時にトラップが発生し，トラップハンドラで待つことになる．

2.1.1 コヒーレンス管理コード

HDSM において，最適化コンパイラは 2 種類のユーザレベルのコードを挿入し，それらはランタイムによって実行される．

- 非同期式読み出しの発行（asynchronous read commitment），すなわち，プリフェッチ
  - `Ra(address, size)`

  読み出しの前に挿入される．ただし，コンパイラがコストを考慮して，挿入されない場合がある（4 章参照）．実アクセスの前にキャッシュのミス/ヒット判定を実行する．ミス時にはハンドラを起動して，

非同期式のリクエストをホームに転送し，メッセージの到着を待たずにすぐに元のタスクに復帰する．

- 書き込みの発行（write commitment）
  - `W(address, size)`

ADSM や UDSM と同様に書き込み時のコヒーレンス管理を行う．

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 $\{x \mid$ `address` $\leq x <$ `address` $+$ `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．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操作を行う．

ページフォールトトラップハンドラは，リクエストが発行済みの場合には，キャッシュブロックが到着するまで待機する．リクエストが未発行の場合には，ホームにリクエストを発行して，キャッシュブロックが到着するまで待機する．

3. プリフェッチに対する手続き間部分冗長性削除の適用

HDSM においてプリフェッチをできる限り早く一括して発行するために，ソフトウェア分散共有メモリを支援する最適化コンパイラ Remote Comunication OPtimizer（RCOP）[10),17)] のもとで，プリフェッチに対して手続き間部分冗長性削除の枠組みを適用する．

3.1 RCOP の概要

まず，最適化コンパイラ RCOP のコード生成プロセス（**図 2** 参照）について以下に述べる．入力は，Lazy Release Consistency: LRC モデル[4)] 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[1)] 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，区間解析ならびに冗長性削除の手法を活用し，共有アクセスのサマリを算出して，コヒーレンス管理コードをできるだけ大きな粒度でソースに挿入する．そのソースをバックエンドのコンパイラ（gcc 等）にかけて，コヒーレンス管理のライブラリとリンクさせて実行コードを生成する．RCOP に関しての詳細は，文献 10), 17) を参照されたい．

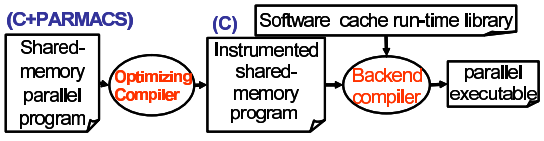


図 2 コード生成プロセス
Fig. 2 Overall compilation process.

3.2 プリフェッチ発行の障害と解決方法

HDSM において，プリフェッチをできるだけ早く一括して発行しようとする際に，障害となるものが 2 つある．1 つ目は同期プリミティブである，プリフェッチを同期プリミティブを越えて遡って発行することは，並列プログラムの意味の変化させてしまう．さらに，適用可能なケースは通信集合が完全に解析できる場合のみと大きく制限される．よって，本手法は同期プリミティブを遡ってプリフェッチを発行する方針はとらない．

2 つ目は条件分岐（制御の交わるところ）である．ただ条件分岐を遡ってプリフェッチを挿入するだけでは，実際には実行されない共有メモリアクセスに対応したプリフェッチが実行される危険がある．その場合，無駄なキャッシュミス/ヒット判定や無駄な通信が発生し，システム全体の性能が低下してしまう．

そこで，本手法は，プリフェッチを条件分岐を遡って挿入するときには，その条件式を複製してプリフェッチに付加する，すなわち，条件付きプリフェッチを挿入する．本手法はプリフェッチに対して，部分冗長性削除の枠組みを適用することにほかならない．これによって上記の危険は回避される．

3.3 例

図 3 に示してある例を用いて具体的に説明する．RCOP [10),17)] は UDSM 用のコードを生成する際に，最終的に一括化された（同期式）読み出しの発行 `R(a, 8 * n)` を条件分岐で分かれた直後に挿入する（図 3 左下）．それより上に移動しない理由は，条件分岐の前にもってくると，if 節の中を通らない場合には，無駄なミス/ヒット判定や無駄な通信を誘発する危険が

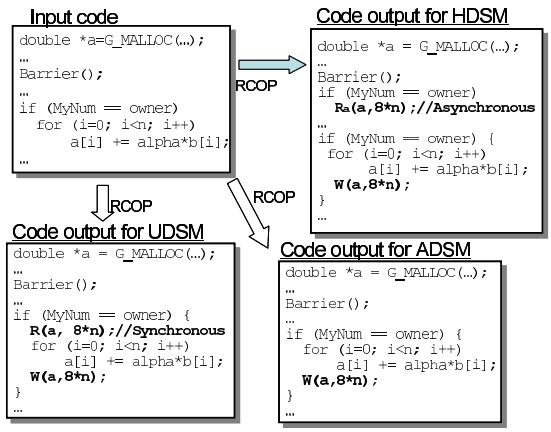


図 3 コード生成例
Fig. 3 Example of generated codes.

$$\mathrm{PANTOUT}(i) = \bigsqcup_{s \in \mathrm{succ}(i)} \mathrm{PANTIN}(s)$$

$$\mathrm{PANTIN}(i) = \mathrm{COMP}(i) \bigcup \mathrm{TRANS_i}[\mathrm{PANTOUT}(i)]$$

$$\mathrm{PAVIN}(i) = \bigsqcup_{p \in \mathrm{pred}(i)} \mathrm{PAVOUT}(p)$$

$$\mathrm{PAVOUT}(i) = \mathrm{TRANS_i}[\mathrm{COMP}(i) \bigcup \mathrm{PAVIN}(i)]$$

$$\mathrm{INSERT}(i) = \mathrm{PANTIN}(i) - \Big(\bigsqcup_{p \in \mathrm{pred}(i)} \mathrm{TRANS_p}[\mathrm{PANTIN}(p)] \Big) - \mathrm{PAVIN}(i)$$

($\mathrm{pred}(i)$ は文 i に先行（precede）する文の集合であり，$\mathrm{succ}(i)$ は文 i に後継（succeed）する文の集合である)

図 4 共有アクセス集合を使用した，冗長なプリフェッチを削除するデータフロー方程式
Fig. 4 Redundancy elimination dataflow equations using shared-access sets for prefetches.

あるからである．

RCOP は HDSM 用のコードを生成する場合，if 節の条件式（図 3 の例の場合には，(`MyNum == owner`)）が副作用を持たないときにのみ，条件式を複製してプリフェッチに加えたコード，すなわち，条件付きプリフェッチ “`if (MyNum == owner) R_a(a, 8 * n)`” を挿入して，さらに上へと移動する．その結果，図 3 右上のようなコードが生成される．

3.4 データフロー方程式

3.3 節の例で述べた最適化を行うために，プリフェッチの部分冗長性削除を行うデータフロー方程式を求める（**図 4**）．各変数の意味を以下に簡潔に述べる．各変数はプリフェッチの集合を表す☆．プリフェッチは共有アクセス集合 $S = (a, s, \mathcal{C})$ という三つ組で表現される．a と s はプリフェッチの先頭アドレスとサイズ，$\mathcal{C}$ はプリフェッチを生成する不等式制約の集合であり，プリフェッチを含むループの誘導変数や，`if` 文や `switch` 文の条件式を表している．

まず，先行する別名解析の結果から各文 i に対して，以下の 2 変数を求める．

$\mathrm{COMP}(i)$ 文 i が行う共有読み出しに対応したプリフェッチの集合

$\mathrm{TRANS_i}[X]$ 引数（プリフェッチの集合）X の中で，文 i を越えて移動してもプログラムの意味を変更しないプリフェッチの集合を返す関数．
具体的には，文 i が同期プリミティブである場合には空集合を返す．また，文 i が X 内のあるプリフェッチ S のパラメータを更新する場合には，X から S を除いた集合を返す．

次に，プログラム内の任意の点 p（CFG の任意のエッジ）に対して，以下の 2 種類のデータフロー集合を計算する．

Partial availability p にいたる，“ある” パスにおいて，発行されるプリフェッチの集合．
ただし，あるパスを選択する条件式が副作用を持たない場合に限る．もし副作用を有する場合には，“ある” を “すべて” に置きかえて考える．つまり，UDSM 上で冗長な読み出しを削除するデータフロー方程式の Availability と同様の意味を持つ．

Partial anticipatability p から始まる，“ある” パスにおいて，発行されるプリフェッチの集合．
ただし，あるパスを選択する条件式が副作用を持たない場合に限る．もし副作用を有する場合には，“ある” を “すべて” に置きかえて考える．つまり，UDSM 上で冗長な読み出しを削除するデータフロー方程式の Anticipatability と同様の意味を持つ．

実際には以下の計 4 種類の変数を計算する．

$\mathrm{PANTIN}(i)$ 文 i の実行前の partial anticipatability

$\mathrm{PANTOUT}(i)$ 文 i の実行後の partial anticipatability

$\mathrm{PAVIN}(i)$ 文 i の実行前の partial availability

$\mathrm{PAVOUT}(i)$ 文 i の実行後の partial availability

図 4 の $\bigsqcup$ はパスの合流点における “条件付き集合和” 演算を表す．すなわち，合流点に至るすべてのパスにおいて，パスを選択する条件式が副作用を持たない場合，個々のパスにおけるプリフェッチの集合の和をとる演算子である．ただし，あるパスにおいてのみ発行されているプリフェッチに対しては，そのパスを選択する条件式の情報を，プリフェッチを表現する共有アクセス集合の第 3 要素に加える．逆に，あるパスにおいて，そのパスを選択する条件式が副作用を持つ場合には，$\bigsqcup$ は “集合積” を表す．すなわち，すべてのパスにおいて発行されているプリフェッチの集合を返す演算子である．

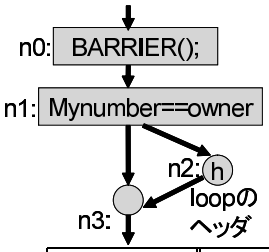
CFG の下流から上流に向かって，PANTOUT，PANTIN を求め，CFG の上流から下流に向かって，PAVIN，PAVOUT を求める．

以上の変数を元に，$\mathrm{INSERT}(i)$，すなわち，文 i を実行する前に実際に挿入されるプリフェッチの集合を求める．

$\mathrm{INSERT}(i)$ は i の前で，

- partial anticipatable であり，かつ
- 先行する文のどれかにおいて，

☆ TRANS のみ，プリフェッチの集合を引数として，プリフェッチの集合を返す関数となる．



Node No. → Property ↓	n0	n1	n2	n3
PANTIN	$\emptyset$	$\{S_2\}$	$\{S_1\}$	$\emptyset$
PANTOUT	$\{S_2\}$	$\{S_2\}$	$\emptyset$	$\emptyset$
PAVIN	$\emptyset$	$\emptyset$	$\emptyset$	$\{S_2\}$
PAVOUT	$\emptyset$	$\emptyset$	$\{S_1\}$	$\{S_2\}$
INSERT	$\emptyset$	$\{S_2\}$	$\emptyset$	$\emptyset$

図 5 図 3 のデータフロー方程式の解
Fig. 5 Solution of dataflow equations described in Fig. 3.

– partial anticipatable ではない
– 同期プリミティブが実行される☆

のいずれかが成立し，かつ

- partial available でない

ような共有読み出しの集合である．

最後に，CFG の上流から下流に向かって，INSERT を計算する．

手続き間でデータフロー方程式（図 4）を計算する場合には，従来の RCOP と同様に，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 3 の例を実際にこの方程式で解く過程を示したのが**図 5** である．図の上が対応する CFG であり，下がプリフェッチの集合である．プリフェッチは共有アクセス集合 $S_1 = (a, 8*n, \emptyset)$，$S_2 = (a, 8*n, MyNum == owner)$ を用いて表現されている．`if` 文内の区間（ループ）に関しては解析済みであるとする．CFG の下流から上流に向かって，PANTOUT，PANTIN を求め，CFG の上流から下流に向かって，PAVIN，PAVOUT を求め，最後に INSERT を求める．

4. Profitability Analysis（収益性解析）

本章は，3 章で求めた INSERT が空集合ではない個所に，対応する共有アクセス集合が表すプリフェッチを実際に挿入するかどうか判定する最適化である．

HDSM におけるプリフェッチの役割は，コヒーレンスの維持よりもレイテンシの削減にある．したがって，INSERT が空集合ではないすべての個所に，プリフェッチを挿入する必要はない．プリフェッチの実行は，当然，オーバヘッドをともなうので，キャッシュミスをしそうな個所に選択的に配置する方が望ましい．しかしながら，静的にキャッシュミスを予測することは困難である．

RCOP では，手続き間別名解析の結果から，プリフェッチを表す共有アクセス集合は，まず，$S = (a, sizeof(type\_a), \emptyset)$（$type\_a$ は a の基本型である）の形で求められる．この形を singleton と定義する．singleton な共有アクセス集合が，fusion，coalescing，redundant index elimination 等の最適化[10),17)] で他の共有アクセス集合と融合されていく．最適化をすべて適用した後でも singleton である共有アクセス集合は，他のアクセスとの関連性が低いと判断できる．逆に，singleton でない共有アクセス集合には，複数の共有アクセスが関連していると見なすことができる．

そこで，本手法は，INSERT が空集合ではない個所で，singleton でない共有アクセス集合に対してのみ，それが表現するプリフェッチを挿入する方針をとる．

5. 履歴を用いたプリフェッチ

本章も前章と同様に，3 章で求めた INSERT が空集合ではない個所に，対応する共有アクセス集合が表すプリフェッチを実際に挿入する際の最適化である．ただし，前章とは違ってランタイムの支援を必要とする．

HDSM では，実行時にキャッシュミスを発行したページ番号の履歴を残すことが可能である．たとえば，バリア同期を使用し，毎回同じ領域にアクセスするような反復計算に関しては，履歴を活用することで，さらにレイテンシを削減することが可能である．すなわち，最初の 1 回目のイテレーションでキャッシュミスを発行したページ番号を記録しておき，2 回目以降のイテレーションでは，その情報を使用して，バリア同期の際にキャッシュブロックリクエストをまとめて発行することでレイテンシを削減できる．2 回目以降のイテレーションでは，キャッシュのミス/ヒット判定も不要となるので，そのオーバヘッドも削減できる．

5.1 コンパイラの解析

まず，バリア同期プリミティブの API を `Barrier(void)` から `Barrier(int)` に変更する，つまり，引数を 1 個増やす．ただし，ソースはそのままにしておく．RCOP が前処理としてバリア同期（`Barrier()`）に番号 (b) を割り振る．テキスト上で異なるバリア同期には，異なる番号を割り振る．それをバリア同期の引数に加える（`Barrier(b)`）．

☆ 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．

以下に，どのプリフェッチを履歴を残すプリフェッチ（履歴プリフェッチ：5.2.2 項参照）に変更するかを判断するアルゴリズムを述べる．

INSERT(i)（i は文）の各プリフェッチ S に対して以下の操作を行う．ただし，終了時には，通常のプリフェッチを挿入するものとする．

（1） 文 i がどの区間（ループ）にも属していないとき，または，並列実行領域にない場合は終了．

（2） 文 i が並列実行領域にある区間 I に属しているとき，

- I がバリア同期を含んでいる場合
  1) S を表現する共有アクセス集合の 3 要素が，I に対してループ不変である場合
  2) または，S を表現する共有アクセス集合の第 2，第 3 要素がループ不変で，第 1 要素（先頭アドレス）が再帰型アクセス（recursive data structure access： RDS access）[6] をしている場合
  には，第 1 イテレーションに対して，履歴プリフェッチを挿入し，第 2 イテレーション以降に対しては何も挿入しない．
  1)，2) 以外の場合には終了．
- I がバリア同期を含んでいない場合 I = “I の親区間” として再帰的に調査．

5.2 ランタイムの支援

履歴を用いたプリフェッチを可能にするために，キャッシュミス時の操作とバリア同期時の操作を改良し，それらを支援するデータ構造を用意する．

5.2.1 データ構造

コンパイル時に割り振ったバリア番号の最大値を `BMAX` とする．ページ番号のリストの配列，`parray[BMAX]` を用意する．さらに，直前に実行されたバリア同期の番号を表す整数 `previous` を用意する．

5.2.2 キャッシュミス時の操作

履歴プリフェッチを定義する．これは従来のプリフェッチと同様の操作を行うことに加えて，キャッシュミスを検知した場合に，そのページ番号を記録する．すなわち，`parray[previous]` のリストに当該ページ番号を加える操作を行う．

履歴プリフェッチはバリア同期を含む反復計算の最初のイテレーションにしか挿入されない．共有読み出しが再帰型アクセスの場合にはループ不変である保証はない．よって，2 回目以降のイテレーションでは，違う領域にアクセスし，プリフェッチが失敗する危険がある．その場合には，実アクセス時にページフォールトハンドラでキャッシュブロックリクエストを発行する．

5.2.3 バリア同期の操作

バリア同期に新たに加えられた引数（5.1 節第 1 段落参照）を `b` とする．従来のバリア同期の操作を完了した後に，`parray[b]` のリスト内のページ番号に対応するページに関して，キャッシュブロックリクエストを発行する．これにより，プログラムの意味を変更しないままで最速にリクエストを発行することが可能になる．バリアがループにない場合や，履歴プリフェッチを含まないループにある場合には，`parray[b]` が空リストのままなので，バリア同期時にキャッシュブロックリクエストは発行されない．

6. 実　　験

まず，HDSM のランタイム “RS3H” の概要を述べ，実験環境について述べる．次に，実験結果を示し，最後に考察を述べる．

6.1 ランタイムシステム：RS3H

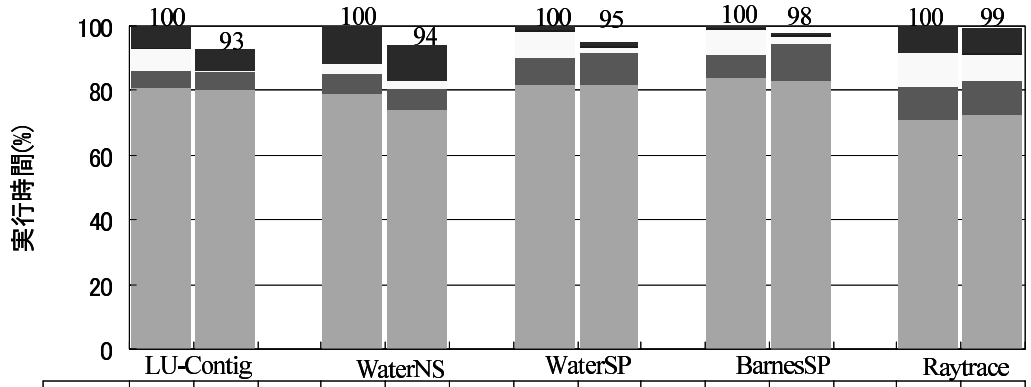
本ランタイムシステム RS3H は，SSS-CORE 上の ADSM/UDSM 用のランタイムシステム：RS3 [10] を FreeBSD/Linux に移植して，プリフェッチを行えるように改良したものである．ポータビリティ性を考慮して，通信は TCP/IP プロトコルを使用している．

RS3H は LRC モデルを実現するシステムであり，コヒーレンスプロトコルとして AURC [3] 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 [16] プロトコルを採用している．以下に特徴をあげる．

- 共有書き込みは RCOP により静的に検知されて，write commitment が挿入される．write commitment はブロックのホームに対して通信を行う．Twin/diff 機構[4] に見られるオーバヘッドは存在しない．
- Vector timestamp[4] を用いて書き込み間の半順序を管理する方式とは異なり，dirty bit table [10] を使用して低オーバヘッドで同時にすべての書き込みを管理する．
- プリフェッチは非同期式リクエストを発行し，すぐにユーザのタスクに復帰する．ページフォールトトラップハンドラは，リクエストが発行済みの場合にはキャッシュブロックが到着するまで待機する．もし，リクエストが未発行の場合には，ページフォールトトラップハンドラはホームにリクエストを発行して，キャッシュブロックが到着するまで待機する．また，5 章で記述したように，バリア同期を改良する．

6.2 実 験 環 境

- コンパイラ：RCOP + gcc3.3(“-O3”)



	LU-Contig		WaterNS		WaterSP		BarnesSP		Raytrace	
Sync	6.773	6.696	11.36	11.18	1.173	0.726	0.481	0.52	8.234	8.109
WC	0.18	0.169	0.014	0.012	0.717	0.668	0.417	0.402	0.195	0.19
Miss	7.017	0.385	3.202	2.047	8.015	1.573	7.896	2.461	10.44	8.342
Msg	5.285	5.614	6.502	6.771	8.194	10.04	7.008	11.16	10.48	10.54
Task	80.75	80.06	78.92	74.12	81.9	81.91	84.2	83.07	70.66	72.5

図 6 プリフェッチの効果（8 台実行）
Fig. 6 Effects of prefetch optimizations.

表 1 最適化の適用結果
Table 1 Effectiveness of optimizations for each program.

アプリケーション	I	P	H
LU-Contig	○	○	×
Water-NSquared	○	○	○
Water-Spatial	○	○	○
Barnes-Spatial	○	○	○
Raytrace	○	○	×

- ノード：Dell PowerEdge 1650 × 8
（1.26 GHz PentiumIII，512 KB cache，2 GB memory）
- OS：FreeBSD 5.1-RELEASE
- ネットワーク：
NIC は Intel 1000XT で，1000BASE-T のスイッチングハブ（BUFFALO の LSW-GT-8W）で接続．
- ランタイム：RS3H
- ベンチマーク
SPLASH-2[13] から以下の 5 個のアプリケーション（LU-Contig（n=4096），Water-NSquared（n=32768），Water-Spatial（n=32768），Raytrace（balls4），Barnes-Hut（n=65536）を選択．Barnes-Hut に関しては文献 12) をもとに，文献 7) を並列化して，それを使用（これを Barnes-Spatial と呼ぶ）．他のアプリケーションを選択しなかった理由を述べる．FFT と Radx は，fetch-on-write の除去[10),17)]によりキャッシュミス数が 0 になるので，本稿で提案した最適化の効果を測定するのにふさわしくないと判断した．また，Volrend に関しては，逐次の計算時間があまりに短かすぎて並列化する意味がないと判断した．

本稿で提案した最適化の効果を測定する．実験は各アプリケーションに対して 5 回行い，以下の結果はその平均をとったものである．また，時間の計測は“rdtsc” 命令を使って行った．

表 1 は各アプリケーションに対する，本稿で提案した最適化の適用状況である．“I” は手続き間部分冗長性削除を表し，“P” は収益性解析を表し，“H” は履歴を活用したプリフェッチを表している．“○” は最適化が適用できたことを表し，“×” は最適化が適用できなかったことを表している．

表 1 の上段 2 個は配列ベースのアプリケーションであり，下段 3 個はポインタベースのアプリケーションである．LU-Contig では，バリア同期を含む大外のループに対して共有アクセスがループ不変にならないので，履歴を用いたプリフェッチは使用されない．Raytrace はバリア同期を含まないロックベースのアプリケーションなので，履歴を用いたプリフェッチは使用されない．

6.3 実験結果

図 6 が各アプリケーションの，ADSM と HDSM における実行時間（8 台）のブレイクダウンである．ADSM の実行時間で正規化してある．“Msg” はメッセージハンドリングの時間を表す．“Miss” はキャッシュミス時の待ち時間を表す．“WC” は write com-

表 2 8 ノードにおける LU-Contig の実行時間（秒）
Table 2 8-processor execution time (sec) for LU-Contig.

	Total	Task	Msg	Miss	WC	Sync
A	35.040	28.294	1.852	**2.459**	0.063	2.373
H	32.560	28.052	1.967	**0.135**	0.059	2.346

mitment の時間を表す．“Sync” は同期時間を表す．“Task” は推定計算時間を表す．すなわち，実行時間から，上記の Msg，Miss，WC，Sync を引いたものである．ただし，ADSM では，シグナル（SIGSEGV）を扱う時間が含まれる．HDSM では，プリフェッチの実行時間とシグナルを扱う時間が含まれる．

図 6 は，HDSM ではプリフェッチの発行により Msg 時間が多少増大するものの，Miss 時間が大幅に減少して ADSM よりも高性能をあげていることを示している．HDSM の Msg 時間が増大する主な理由は，ページ更新のメッセージが到着したときに，メッセージハンドラがページをマップするシステムコール（mmap）を呼ぶからである（ADSM ではキャッシュミスハンドラがそれを行う）．また，HDSM では Raytrace を除いて Task 時間が減少していることが分かる．さらに，ADSM と HDSM の両方において，RCOP の最適化により書き込みのオーバヘッドが低く抑えられていることが分かる．

以下において，各アプリケーションにおける実験結果を詳しく見ていく．

6.3.1 LU-Contig

LU-Contig は，密行列の LU 分解を行う，配列ベースのアプリケーションである．各ノードが担当する部分行列内の要素が，連続した共有アドレス上に配置される．したがって，共有アクセスは手続き間にわたって一括して発行される．HDSM では，部分冗長性削除の最適化によりプリフェッチがバリア同期の直後に実行されるので．レイテンシ削減の効果が期待できる．**表 2** の “A” は ADSM の実行結果であり，“H” は HDSM の実行結果である（以下ではこの記法を使用する）．収益性解析の適用有無にかかわらず，時間計測部分のコードは変化しなかった．表 2 が示しているように，HDSM における Miss 時間は，ADSM の Miss 時間より大幅に減少している．本アプリケーションでの，各ノードのプリフェッチリクエスト数の平均（平均プリフェッチリクエスト数と呼ぶ）は 10,374 であり，そのうち，データ転送が実アクセス前に間に合わなかったもの（ページフォールトトラップで待たされたもの，平均ミス数と呼ぶ）は 225 回である．プリフェッチの実行時間分だけ，HDSM の Task 時間が ADSM の Task 時間より低速になると予想されたが，実際は，シグナル（SIGSEGV）を扱う時間の削減の効果が大きく，HDSM の Task 時間の方が約 1%高速になっている．

表 3 8 ノードにおける Water-NSquared の実行時間（秒）
Table 3 8-processor execution time (sec) for Water-NSquared.

	Total	Task	Msg	Miss	WC	Sync
A	238.358	**188.123**	15.498	**7.632**	0.033	27.072
H	224.385	**176.680**	16.138	**4.880**	0.028	26.659
H-H	230.657	181.517	16.506	5.326	0.036	27.274

表 4 8 ノードにおける Water-Spatial の実行時間（秒）
Table 4 8-processor execution time (sec) for Water-Spatial.

	Total	Task	Msg	Miss	WC	Sync
A	3.795	3.108	0.311	**0.304**	0.027	0.045
H	3.602	3.109	0.381	**0.060**	0.025	0.028
H-H	3.778	3.190	0.304	0.221	0.027	0.036

6.3.2 Water-NSquared

Water-NSquared は，水分子の分子間力を計算する配列ベースのアプリケーションである．水分子の配列がブロック分割されているので，各ノードの共有データの参照は規則的となり，一括して発行される．近傍の分子を参照する際にキャッシュミスが発生する．

表 3 の “H-H” は HDSM において履歴によるプリフェッチを適用しなかった場合の実行結果である（以下ではこの記法を使用する）．収益性解析の適用有無にかかわらず，時間計測部分のコードは変化しなかった．

HDSM における平均プリフェッチリクエスト数は 41,334 であり，平均ミス数は 37,807 である．HDSM の Task 時間の方が ADSM の Task 時間より 6%高速になっている．この理由は，シグナル（SIGSEGV）を扱う時間の削減の効果だけでなく，参照の局所性が向上したためであると考察される．

H と H-H の比較から，手続き間部分冗長性削除のみでも Miss 時間を削減できるが，履歴を活用したプリフェッチにより，さらに Miss 時間を削減できることが分かる．

6.3.3 Water-Spatial

Water-Spatial は，Water-NSquared と同じ問題を，水分子が属する空間を分割するアルゴリズムを用いて解く，ポインタベースのアプリケーションである．

表 4 から，HDSM の Task 時間と ADSM の Task 時間はほぼ同様であり，HDSM の Sync 時間は ADSM の Sync 時間よりも削減されていることが分かる．この理由は，HDSM において，Miss 時間の削減により全体のロードバランスが向上したためであると考察さ

表 5 8 ノードにおける Barnes-Spatial の実行時間（秒）
Table 5 8-processor execution time (sec) for Bornes-Spatial.

	Total	Task	Msg	Miss	WC	Sync
A	173.315	145.927	12.146	**13.685**	0.723	0.834
H	169.185	143.976	19.348	**4.265**	0.696	0.901
H-H	184.385	160.971	13.178	8.580	0.694	0.962

れる．HDSM における平均プリフェッチリクエスト数は 2,282 であり，平均ミス数は 579 である．

表 4 の H と H-H の比較により，手続き間部分冗長性削除だけでは，Miss 時間は約 27%しか削減できなかったが，履歴を活用したプリフェッチにより，さらに約 43%削減できたことが分かる．プログラムの解析の結果，主要なデータ構造である水分子を表すリスト構造が変化しないということが分かった．そのため，履歴によるプリフェッチが非常に効果的に作用したと推測される．

収益性解析の適用により，プリフェッチの数は少量削減された．しかし，実行時間とそのブレイクダウンは変化しなかった．

6.3.4 Barnes-Spatial

Barnes-Spatial は，N 体問題を Barnes-Hut のアルゴリズムを用いて解く，ポインタベースのアプリケーションである．Barnes-Spatial は，Water-Spatial と同様に粒子が属する空間を分割するアルゴリズムを使用している．

主要なデータ構造である八分木は，バリア同期で囲まれたループにおいて毎イテレーション少しずつ変化するので，履歴によるプリフェッチが効率的に作用すると推測される．HDSM の平均プリフェッチリクエスト数が 113,268 で，平均ミス数が 23,955 であるのに対して，ADSM の平均ミス数は 91,288 である．つまり，履歴によるプリフェッチにより通信量が増大していることが分かる．しかし，ページ更新のメッセージが実アクセス前に届く効果の方が大きく，実行時間が改善していることが**表 5** から分かる．

表 5 の H と H-H の比較から以下のことが分かる．手続き間冗長性削除のみ適用した場合には，プリフェッチが挿入されたコードが，力の計算の毎イテレーション実行されて，命令オーバヘッドが増大する．その結果，Task 時間が増大して，全体の実行時間が ADSM より増大してしまう．

収益性解析の適用有無にかかわらず，時間計測部分のコードは変化しなかった．

6.3.5 Raytrace

Raytrace は，タスクキューを使用して並列にレイトレーシングを行うポインタベースのアプリケーションである．共有データの参照は不規則だが，物体のデータは無効化されないので，キャッシュミス数そのものは少ない．

表 6 8 ノードにおける Raytrace の実行時間（秒）
Table 6 8-processor execution time (sec) for Raytrace.

	Total	Task	Msg	Miss	WC	Sync
A	2.803	1.980	0.294	**0.292**	0.005	0.231
H	2.794	2.032	0.296	**0.234**	0.005	0.227
H-P	3.285	2.633	0.302	0.168	0.006	0.176

表 6 の “H-P” は HDSM において，収益性解析を適用しなかった場合の実行結果である．

表 6 から，HDSM では，不規則参照が多いものの，プリフェッチが引き起こすオーバヘッドが低くおさえられていることが分かる．この理由は，収益性解析により，singleton な共有アクセス集合が表すプリフェッチが挿入されないためであると考察される．HDSM の平均プリフェッチリクエスト数は 1,430 であり，平均ミス数は 1,320 である．

また，H と H-P の比較から，収益性解析を適用しない方が，Miss 時間が削減されていることが分かる．しかしながら，収益性解析を適用しない方は，プリフェッチのオーバヘッドにより Task 時間が増大していることが分かる．実行時間全体としては，収益性解析を適用した方が高速になる．

6.4 考　　察

上記の結果から，以下のことが分かる．

- 手続き間部分冗長性削除はすべてのアプリケーションに対して適用可能であり，通信の遅延を隠蔽する効果があることが観測された．特に配列ベースのアプリケーションに対して有効である．

ただし，Barnes-Spatial にみられるように，プリフェッチのオーバヘッドを低くおさえることができないような場合が存在する．そのような場合には，プリフェッチしない場合よりも実行時間全体として遅くなる．

- 履歴を活用したプリフェッチは，適用範囲が限定されるものの，効果的であることが確認された．特に，同じ領域で繰り返し作業するようなポインタベースのアプリケーションに対して有効である．

Barnes-Spatial にみられるように，厳密に同じ領域にアクセスしない場合でも適用可能であり，効果があることが確認できた．また，履歴の管理のオーバヘッドは低くおさえられているということが分かった．

- 収益性解析はプリフェッチのコストとレイテンシ削減のトレードオフの問題であることが確認できた．すべての共有メモリアクセスに関して，手続き間部分

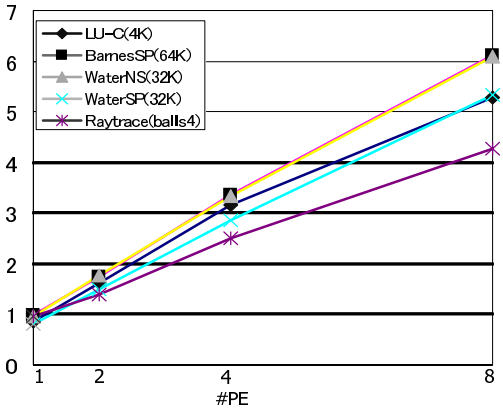


図 7 HDSM における高速化率
Fig. 7 Speed-up ratios in HDSM.

冗長性削除の枠組みに従ってプリフェッチを挿入すると，確かに，キャッシュミスの待ち時間は最小になる．しかし，Raytrace にみられるように，不規則参照を多く含むアプリケーションの場合には，他のプリフェッチと一括化することが不可能になり，プリフェッチの命令オーバヘッドが無視できなくなってくる．

図 7 は，最適化をすべて適用したときの HDSM における各アプリケーションの高速化率を示したものである．高速化率は，逐次の実行時間をもとに算出している．ロック/アンロックを扱う Raytrace では 8 台で 4 倍強しか高速にならないものの，その他のアプリケーションでは 8 台で，5 倍以上高速になっていることが分かる．

7. 関 連 研 究

S-DSM におけるプリフェッチの研究としては，まず，Dwarkadas ら[2] の研究があげられる．彼らは，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[4] 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おり，プリフェッチも行っている．しかし，手続き呼び出しと条件節が解析の障害になる．

次に Mowry ら[9] の研究があげられる．彼らの提案する history-pointer prefetching や data-linerization prefetching[6] は，コンパイラが自動的に適用することがほぼ不可能であり，コード生成は人手で行われている．

本研究は，我々がこれまで提案/実装してきたコンパイラが支援する S-DSM 機構[8),16)] の延長と見なすことができる．コンパイラが支援する S-DSM 機構には，他に Shasta[11] や Omni/CSDSM[14),15)] が存在する．いずれもプリフェッチは扱っていない．

8. ま と め

本稿は，S-DSM において，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により，遠隔メモリアクセスのレイテンシをさらに削減する手法を提案ならびに実装し，その効果を測定した．プリフェッチにより最大 7%実行時間を削減できることを確認した．また，配列ベースのアプリケーションに対してだけでなく，ポインタベースのアプリケーションに対しても効果的であることを確かめた．今後は，プリフェッチを他のノードが起動する “キャッシュインジェクション” を実装し，さらなるレイテンシの削減を目指す予定である．

参 考 文 献

1) 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).
2) Dwarkadas, S., Cox, A.L. and Zwaenepoel, W.: An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System, *Proc. ASPLOS-VII* (Oct. 1996).
3) Iftode, L., Dubnicki, C., Felten, E.W. and Li, K.: Improving Release-Consistent Shared Virtual Memory using Automatic Update, *Proc. 2nd HPCA* (Feb. 1996).
4) Keleher, P., Cox, A.L. and Zwaenepoel, W.: Lazy Release Consistency for Software Distributed Shared Memory, *Proc. 19th ISCA* (May 1992).
5) Li, K.I.: A Shared Virtual Memory System for Parallel Computing, *Proc. 1988 ICPP* (Aug. 1988).
6) Luk, C.-K. and Mowry, T.C.: Compiler-Based Prefetching for Recursive Data Structures, *Architectural Support for Programming Languages and Operating Systems* (1996).
7) Makino, J.: NBODY: An implementation of Barnes-Hut treecode. http://grape.astron.s.u-tokyo.ac.jp/~makino/softwares/C++tree/index.html
8) 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).
9) Mowry, T.C., Chan, C.Q.C. and Lo, A.K.W.: Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory, *Proc. 4th IEEE Symp. on High-Performance Computer Architecture* (*HPCA-4*) (1998).
10) Niwa, J.: Study on Optimizing Compilers to Support Software Distributed Shared Memory Systems, Ph.D. thesis, Department of Information Science, The University of Tokyo (2000).
11) 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).
12) Shan, H. and Singh, J. P.: Parallel Tree Building on a Range of Shared Address Space Multiprocessors: Algorithms and Application Performance, *Proc. IPPS/SPDP 1998* (1998).
13) 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).
14) 佐藤茂久，佐藤三久：OpenMP 向けコンパイラ支援ソフトウェア DSM における最適化コンパイル手法，情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム，Vol.42, No.SIG12, pp.77–94 (2001).
15) 佐藤茂久，草野和寛，佐藤三久：OpenMP 向けコンパイラ支援ソフトウェア分散共有メモリ，情報処理学会論文誌，Vol.42, No.4, pp.788–801 (2001).
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).
17) 丹羽純平，松本　尚，平木　敬：ソフトウェア分散共有メモリ機構を支援する最適化コンパイラ，情報処理学会論文誌，Vol.42, No.4, pp.879–897 (2001).

(平成 16 年 10 月 4 日受付)
(平成 17 年 2 月 4 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）．現在，科学技術振興機構，さきがけ研究 21 研究員．並列化/最適化コンパイラに関する研究に従事．他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．