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

丹羽純平[†] 稲垣達氏[†]
松本尚[†] 平木敬[†]

我々は保護された高速なユーザー通信 / ユーザー同期を実現する “非対称分散共有メモリ:ADSM” を提案してきた。 ADSM は読み出しと書き込みの実現モデルが別々で、読み出しは通常の仮想共有メモリ方式と同様であるが、書き込みに関してはコンシステンシ維持コードが埋め込まれる。書き込みの自由度が高いから、様々な最適化が可能になる。我々はコンシステンシ維持コードの数を静的 / 動的に削減することで、書き込みのオーバーヘッドを削減する最適化手法を提案する。汎用並列オペレーティングシステム SSS-CORE と AP1000+ 上に作成したコンパイラ並びにランタイムシステムにおいて SPLASH-2 の LU-Contig を使って評価を行なった。実行時間は静的な最適化により 80% 向上し、更に動的な最適化を行なうことで 30% 向上した。

Performance Evaluation of Compiling Techniques on Asymmetric Distributed Shared Memory

JUNPEI NIWA,[†] TATSUSHI INAGAKI,[†] TAKASHI MATSUMOTO[†] and KEI HIRAKI[†]

We have proposed an “Asymmetric Distributed Shared Memory: ADSM”, that realizes user-level protected high-speed communications/synchronizations. In the ADSM, the shared-read is based on a cache-based shared virtual memory system. As for the shared-write, instructions for consistency management are inserted after the corresponding store instruction. Therefore, various optimizations can be performed. We propose an optimizing method of reducing overheads for consistency management. The algorithm coalesces a sequence of consistency management instructions statically/dynamically. We have implemented the prototype of the compiler and the runtime system for the ADSM on a multicomputer Fujitsu AP1000+ and the general-purpose massively-parallel operating system: SSS-CORE. The performance evaluation using LU-Contig of SPLASH-2 shows that the execution time is reduced by 80% using static optimization and it is further reduced by 30% using dynamic optimization.

1. はじめに

近年、分散メモリ型計算機が重要な計算資源になりつつある。しかし、分散環境でメッセージパッシングを使用してプログラミングを行なうのは多大な労力を必要とする。この労力を軽減するために、共有メモリモデルを提供する必要が生じる。

共有メモリモデルをハードウェアで提供する (ハードウェア DSM) 場合、大規模 NUMA 環境では invalidate プロトコルは最適化に逆効果になる。 update やキャッシュインジェクションタイプのプロトコルを選択するのが望ましいが、実装は容易ではない。

そこで、実装コストの低いソフトウェアで共有メモリモデル (ソフトウェア DSM) を提供する研究が多数 [6], [4], [1], [7] なされてきた。コンシステンシ管理単位がページ単位 [6] であるために false sharing が発生してデータの転送量が増大する問題が発生するが、ページの差分のみを転送したり [4]、同期変数に束縛されたデータの更新箇所のみを転送したり [1]、ハードウェア DSM をソフトでエミュレートして管理単位をラインサイズに下げることで [7] この問題を回避している。

我々は、特殊な通信同期ハードウェアを仮定しない分散メモリ環境で効率の良い共有メモリモデルを提供するために、保護され仮想化された高速なユーザ通信 / ユーザー同期を実現する非対称分散共有メモリ **(Asymmetric Distributed Shared Memory:ADSM)** を提案してきた [11]。 ADSM 上では共有メモリの読み出しに関しては従来の仮想共有メモリ方式に従い、共有メモリの書き込みと同期はユーザコードに一連の命令

† 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

シーケンスを静的に挿入する。従って共有メモリの書き込みや同期を一連の命令シーケンスに変換する最適化コンパイラとそれをサポートするランタイムが必須になる。しかしながら、それらによって既存の DSM では実行できなかった種々の最適化が可能になる。

本稿では、 ADSM 上で書き込みのオーバーヘッドを静的 / 動的に削減するためのコンパイル手法 / ランタイムサポートを提案する。提案した手法の有効性を AP1000+ とワークステーションクラスタ上の汎用並列オペレーティングシステム SSS-CORE [11] における実験によって評価する。

2. 非対称分散共有メモリ

ADSM では読み出しと書き込みの実現モデルが異なっている☆

- 共有領域への読みだし
  キャッシュベースの通常の shared virtual memory model に基づいている。従来のソフトウェア DSM と同様にページ単位のキャッシュを用意して、プロセッサのロード命令で行なう。ページトラップでキャッシュミスを検知して対処する。
- 共有領域への書き込み
  書き込まれるデータの従うべきプロトコルに従って、コンシステンシ維持のためのコード列 (遠隔メモリアクセスや付加的なメモリ操作) を実行コードとして埋め込む。

共有領域への書き込みをストア一命令から一連のコード列に変換してしまうために従来のハードウェア、及びソフトウェア DSM では不可能であった種々の最適化が可能になる。

- コンシステンシ維持コード列のコアレシング
  ある同期区間において連続した共有アドレスへの書き込み列がある場合、コンシステンシ維持コードの列を連続領域に対する一つの維持コードに変換することによって、実行時のオーバーヘッドが軽減される。
- 通信パケットのコンバイニング
  送り先が同じパケットをコンバイニングしてメッセージ数を減らし、更新型メモリアクセスのオーバーヘッドを削減する。

3. プロトコル (SAURC) の実現

我々は ADSM 上で 3 種類のプロトコルの実装を行なってきた [14]。今回はその中で SAURC プロトコルについて詳しい説明を行なう。 SAURC はソフトウェアで Automatic update release consistency(AURC) [3] をエミュレートするプロトコルである [11]。 AURC では各ページに home が存在する。 home 以外にコピーページを有するプロセッサが一台の時は、互いに更新しあう (CopySet-2 プロトコル) が、それ以外の時はコピーから home への更新のみが伝えられる (CopySet-N プロトコル)。これにより home は常に最新の値に保たれる。 home 以外のコピーページは WriteNotice [5] を使用した Lazy Release Consistency(LRC) [4] と同様な無効型プロトコルで管理される。

3.1 書き込みのコンシステンシ維持コード

バッファをノード台数個用意する。

- アドレスからページ番号 (P) を求める。 P への初めての書き込みの場合には WriteNotice [5] を作成する。
- 自分が P の home(Home) でない場合、アドレスとサイズと中身を Home に対応するバッファにコピーする (ただし、オーバーヘッド削減のため、過去に同じアドレスへの書き込みがあったかどうかは調べない)。
- 自分が P の Home であり、かつ CopySet-2 である場合、アドレスとサイズと中身を、コピーを持つノードに対応するバッファにコピーする (ただし、オーバーヘッド削減のため、過去に同じアドレスへの書き込みがあったかどうかは調べない)。

書き込みの都度バッファの中身を転送しないで、バッファが一杯になった時点 / 同期点に到達した時点でバッファの中身をそのまま対応するノードに転送する。つまりコンパイラレベルでメッセージのコンバイニングを行なう。同期点に到達した場合はコンシステンシ維持完了の確認のためにメモリバリアを張る。対応するノードは転送されたデータを元に自分のページを更新し、TimeStamp を更新する。更新後に転送されたデータは棄却する。

3.2 アクセスミス

ページフォールトが発生した場合、該当ページの home にページ要求のリクエストを発行する。該当ページの home はその要求を受け取ると、ページの中身と TimeStamp をページフォールトを起こしたノードに転送する。

3.3 同　　期

各ロックには round-robin で割り振られた管理人が存在する。自分がロックを持っていない限り、全てのロック acquire のメッセージは管理人に転送し、管理人の元で逐次化されて管理される。管理人はロック acquire のメッセージを受け取ると、そのメッセージを “最後にロック acquire のメッセージを発行したノード” にフォワードする。ロックを aquire した時には、ロックを release したノードが生成した WriteNotice の情報が伝えられる。それを元に対応するページを無効化する。ただし、 TimeStamp の比較を行なって TimeStamp が古いページのみ無効化する [3]。

バリアは集中管理アルゴリズムで実装されている。各ノードはバリアに入ったら、管理者に欠けている WriteNotice の情報を管理者に転送する。管理者は全

☆ これが名前の由来である。

てのノードから情報を集めて対応するページの無効化を行なう。然る後に、各ノードに欠けている WriteNotice の情報を各ノードに転送する。各ノードは転送された WriteNotice の情報を元に該当するページの無効化を行なう。ただし、 TimeStamp の比較を行ない TimeStamp が古いページのみ無効化する [3]。

4. 共有領域への書き込みの検出

コンパイラが共有領域への書き込みをコンシステンシ維持コードを含んだコード列に変換する。コンパイラが共有領域への書き込みを検出するために、手続き間ポインタ解析 [8], [2] の手法を用いる。手続き間ポインタ解析は共有変数の reference や動的な共有領域の確保を源とした前進型手続き間データーフロー解析である。手続き間ポインタ解析によって共有アドレスへのポインタを保持し得る変数を検出し、該当する書き込みの後にコンシステンシ維持コードを挿入する。

5. 共有領域への書き込みのオーベーヘッドの削減

連続した共有領域に対する書き込みをできるだけまとめて処理して、冗長なコンシステンシ維持コードを除去することができれば、共有領域への書き込みのオーバーヘッドが削減できる。

5.1 コンシステンシ維持コードの静的なコアレシング

一連の共有領域への書き込みが連続したアドレスに対して行なわれていて同期コードを間に含まない場合、コンシステンシ維持コードの情報としては連続した大きな領域全体に対する一つのコンシステンシ維持コードと等価である。

以下に誘導変数の情報を利用してアフィンなメモリアクセスの検知を行ない、ループレベルのコンシステンシ維持コードのコアレシングを行なうアルゴリズムを示す。ループには *depth* があり、 n 重ループネストの最外ループの *depth* は 1 で最内ループの *depth* は n とする。コンシステンシ維持コードはアドレスとサイズの組 $I(A,S)$ で表す。まず、必ず実行される共有書き込みを含む n 重ループネストを列挙して、各ループネストに対して以下を実行する

(0) 各共有書き込み毎にコンシステンシ維持コードを生成
(1) $Depth := n$
(2) $Depth = 0$ ならば終了
(3) $depth = Depth$ を満たす各ループ (L) に対して以下を実行する
(3.1) ループ L の中にコンシステンシ維持コードが複数存在して、そのアドレス部分が連続であり、コード間に同期コード / リターンコードがなければ、コンシステンシ維持コードをコアレシングする
(3.2) 同期コード / リターンコードがあれば (3) へ
(3.3) $I(A,S)$ で表される各コンシステンシ維持コードに対して
- A(アドレス部分) がループ不変である場合:
  ループ L からコンシステンシ維持コードを削除して、ループ L の外側にコンシステンシ維持コードを挿入する
- A がループ L の誘導変数である場合:
  もし A のストライド (A_{stride}) が S 以下であれば、コアレシング可能であり、ループ L からコンシステンシ維持コードを削除して、ループ L の外側にコンシステンシ維持コードを挿入する。挿入されるコンシステンシ維持コードは $I\,(A_{low}, (c-1) * A_{stride} + S)$ と表現される (ただし c はループ L の回数で A_{low} は A の最小値)

(4) $Depth := Depth - 1$ として (2) へ

以下にコアレシングのアルゴリズムを適用した例を挙げる。

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
    I (&a[i], sizeof (double));
}
```

⇓ Static Coalescing Optimization

```
for (i = 0; i < n; i = i + 1) {
    a[i] = a[i] + alpha * b[i];
}
I (&a[0], n * sizeof (double));
```

5.2 コンシステンシ維持コードの動的なコアレシング

一連の共有領域への書き込みが以前に書きこんだ領域を上書きしてしまうが、静的には検知できない場合が存在する。また、静的には連続領域に書き込みを行なっていると解析できないが、連続領域に書き込みを行なっている場合も存在する。上記のような場合でもコンシステンシ維持コードのオーバーヘッドを削減するランタイムサポートを提案する。

3.1 で述べた実装におけるコンシステンシ維持コード内の操作では、以前に同じアドレスに書いたかどうかを調べずに、アドレスとサイズと中身をバッファに書き込んでいた。この方式は静的に全ての連続領域への書き込みを解析できるような場合にオーバーヘッドが一番少ない。しかし、静的な解析では検出できないような連続領域への書き込みが多い場合はこの方式はオーバーヘッドが大きくなる。静的な解析では検出できない連続領域への書き込みが多い場合でも SoftwareDirtyBit 方式 [10] を利用することでオーバーヘッドを削減できる。

SoftwareDirtyBit 方式では、各共有アドレスがそのアドレスへの書き込みが行なわれたかどうかを反映する bit(SoftwareDirtyBit) を持っている。コンシステンシ維持コード内で該当アドレスとサイズと中身をバッファに書き込むかわりに、該当アドレスに対応する SoftwareDirtyBit の領域を算出してサイズ分の

SoftwareDirtyBit をセットする (図 1)。

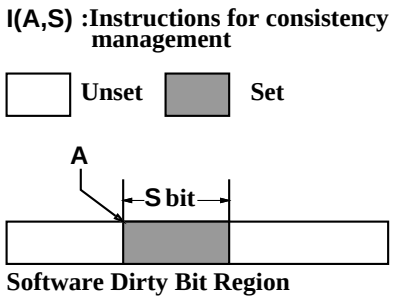

図1 Software Dirty Bit

3.1 の実装では対応するバッファに該当アドレスとサイズと中身を書き込んでしまうため、バッファが一杯になると対応するノードへの通信が発生する。つまり update が生じる。従って共有領域への書き込みが以前に書きこんだ領域を上書きしてしまうような場合が何度も繰り返されると、無駄なメッセージが飛び、無駄な update が発生する危険がある。 SoftwareDirtyBit 方式では、 SoftwareDirtyBit をセットするローカルな計算のオーバーヘッドが課せられるだけで済む。

同期点に到達した時に、 SoftwareDirtyBit の領域を探索することで自分が修正した部分を検出する。ただし、この部分は 3.1 で述べた実装には存在しないオーバーヘッドである。つまり 3.1 の実装ではバッファの中身がそのままメッセージのパケットになっていたが、SoftwareDirtyBit 方式ではメッセージのパケットを作成しなければいけない。しかしながら、この時点では自分が書き込んだ共有領域が最大限にコアレシングされて検出される。パケットを生成したら対応するノードに転送を行なう。つまり SoftwareDirtyBit 方式では update は可能な限り lazy に行なわれる。

6. 関 連 研 究

6.1 OS ベースのソフトウェア DSM

従来のソフトウェア分散共有メモリ (DSM) [6] では、共有領域への書き込みに対してコンパイラが単なるストア命令を用意して、ページの書き込みトラップルーチンでコンシステンシ維持コードを実行した。ここが ADSM とは大きく異なる点である。共有領域への書き込みの都度トラップを起こすオーバーヘッドを削減するために、同期区間内のページの差分を計算する diff 方式が提案された [4]。ページの書き込みトラップは同期区間内で一回に押えられる。しかし、 diff を作成するためには、ページのコピーを生成したり、ページの修正された部分とは無関係にページ全部を調べなければいけない。このオーバーヘッドは無視できない。 AURC [3] ではデータ転送の軽いハードウェアを使用して、共有領域への書き込みの結果を全て home に転送し、 home を常に最新の状態に保つ [3] ことで、 diff 方式が抱える問題を解決する。

6.2 コンパイラベースの DSM

オブジェクトベースのソフトウェア DSM である Midway [1] では各共有データは同期変数に束縛されている。 lock の acquire の時点で、その lock に束縛されたデータの変更のみが伝えられる (Entry Consistency)。 Midway ではコンパイラが共有領域への書き込みをストア命令と定まったコード列☆に変換するだけである [10]。プロトコルの切替えやコンシステンシコード列のコアレシングといった最適化は実行していない。

Eager RC をソフトウェアで実装した Shasta [7] では、アセンブラレベルの instrumentation によって局所領域以外のアクセスにコンシステンシ維持コードが挿入される。 batching Miss Check や Invalid Flag Technique といった種々の最適化を行なっているが、ループレベルのコンシステンシ維持コードのコアレシングは行なっていない。

7. 実験と評価

我々は ADSM のコンパイラとランタイムシステムのプロトタイプを AP1000+ と NOW 版の SSS-CORE に実装してきた。本実験ではプロトコルは SAURC を使用し、アプリケーションとして SPLASH-2 [9] の中の Kernel の LU-Contig を採用し、データの home の配置の最適化を行なった。

7.1 AP1000+

各ノードは 50MHz SuperSPARC(二次キャッシュなし, 16MBytes メモリサイズ) でネットワークは 2 次元トーラスでバンド幅は各リンク毎に 25MBytes/sec である。 AP1000+ の OS はユーザレベルの割り込みハンドラをサポートしていないため、仮想記憶機構を利用できない。今回の実装では共有ページへのアクセスの前にページの有効性をチェックするコードを挿入した。もし有効でないページにアクセスしたら、ユーザーレベルのページフォールトハンドラを呼ぶ。外部からのメッセージに割り込みで反応することができないから、リモートノードからのメッセージは polling で処理する。コンパイラがループのバックエッジと関数呼びだしの所に polling のコードを挿入した。

図 2は静的なコアレシングの効果を調べたものである。 **S** はコアレシングしたもの、 **N** はしないものを示す。問題のサイズは 256×256 行列でブロックサイズが 16 である。グラフの縦軸は実行時間 (秒)、横軸はプロセッサ台数である。以降の実験の実行時間の内訳は表 1 を参照されたい。静的なコアレシングにより、ボトルネック部分のオーバーヘッド軽減に成功している。コアレシングによってデータ参照の局所性が変化すること、

☆ 該当アドレスの Software dirty bit をセットする

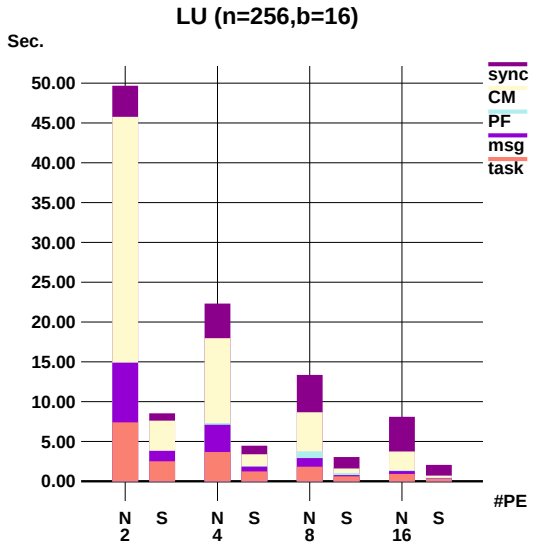

図2　静的なコアレシングの効果 (AP1000+)

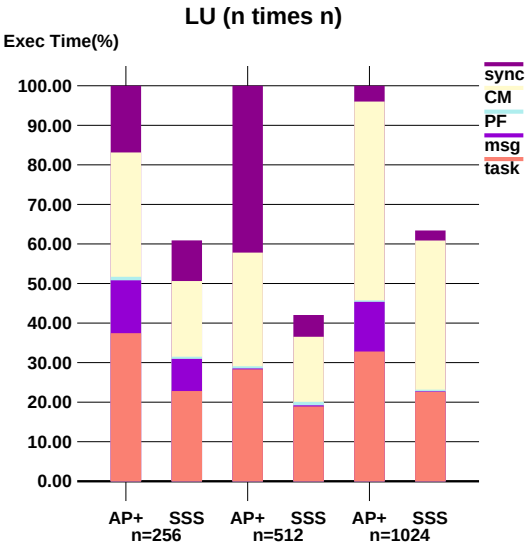

図3　SSS-CORE と AP1000+ の比較 (#PE=4)

表 1 実行時間の内訳

sync	ロックやバリアなど同期処理にかかる時間
CM	コンシステンシ維持コードの時間
PF	ページフォールト処理にかかる時間
msg	task 実行中にメッセージを処理した時間
task	アプリケーションプログラム本来の計算時間

またコンシステンシン維持コードへの手続き呼び出しは task にカウントされていることから task も減少している。

7.2　SSS-CORE

ワークステーションクラスタ上の汎用超並列オペレーティングシステム SSS-CORE で実験を行なった。各クラスタノードは Axil 320 model8.1.1(Sun SS20 互換機, 85MHz SuperSPARC × 1) からなり、 Fast Ethernet SBus Adapter2.0 を追加して 100BASE-TX のスイッチで Fast Ethernet 接続されている。ユーザーレベルの保護された高速なメモリベース通信 (**MBCF**:Memory Based Communication Facilities) [11], [12], [13] を実行し、ピークバンド幅は 11.2MBytes/sec である。メモリベースシグナルを使用してリモートからのメッセージに対処する。仮想記憶機構を利用して割り込みでページフォールトを検出する部分は現在実装中である。現時点ではページフォールトは AP1000+ の実装と同様にソフトウェアで検出している。図 3は AP1000+ との比較を行なった結果である。プロセッサ台数は 4 台で行列の一辺のサイズを 256、 512、 1024 と変化させた。縦軸は実行時間で AP1000+ の時間で正規化している。 SAURC は通信量の多いプロトコルであるからネットワークが高速な AP1000+ に有利であるが、SSS-CORE のメモリベース通信機能が十分性能を発揮しているために、実行時間の比はプロセッサのクロック比に近い値になっている。

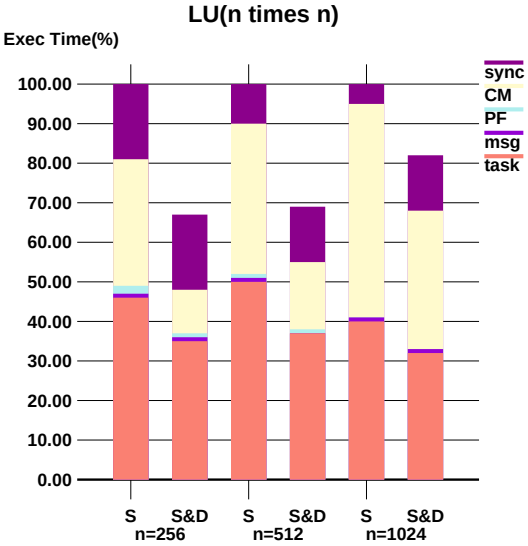

図4　動的なコアレシングの効果 (SSS-CORE)

図 4は動的なコアレシングの効果を調べたものである。 **S** は静的にコアレシングしたもの、 **S&D** は静的かつ動的にコアレシングしたものを示す。プロセッサ台数は 4 台で、行列の一辺のサイズを 256、 512、 1024 と変化させた。縦軸は実行時間で静的にコアレシングしたもので正規化している。動的にコアレシングを実行することで実行時間の削減に成功している。動的にコアレシングを実行した方がコンシステンシ維持コードの時

間 (CM) の削減に成功している。ただし、バリアに到着した時に自分が修正した部分を検出するオーバーヘッドが加算されて同期処理の時間 (sync) は増大している。SSS-CORE ではメモリベース通信で kernel が update をしてくれるが、その時間は task にも含まれている。従って update の量が減少したことで task の時間も減少する。

表 2 は各プロセッサが転送するメッセージの数量である。動的なコアレシングを行なうことによりメッセージ

表 2 通信トラフィックの平均

行列のサイズ	最適化	メッセージ数	転送量 (Mbytes)
n = 256	S	11258	12.0
	S&D	460	0.9
n = 512	S	88694	99.0
	S&D	2376	6.3
n = 1024	S	703826	791.0
	S&D	14996	49.0

の数量が大幅に削減できることを示している。

現実装での静的なコアレシングは手続き内のループに限られている。 LU は手続きの呼び出しがネストしていて、手続き間で更にコアレシング可能な場合が非常に多い。従って動的なコアレシングが効果的に作用する。一番効果的な最適化は静的なコアレシングを手続き間で行なうことであり、現在実装中である。

8. ま と め

我々は ADSM 上で共有書き込みの際のオーバーヘッドを削減するコンパイル技法を提案した: 1) 誘導変数の情報を利用してアフィンなメモリアクセスの検知を行なってコンシステンシ維持コードを静的にコアレシングする、 2) 静的にコアレシングできない場合でも SoftwareDirtyBit 方式を使用してランタイムで動的にコアレシングする。

我々は AP1000+ と SSS-CORE 上にコンパイラとランタイムシステムのプロトタイプを作成した。プロトコルとして SAURC を選択し、プロトタイプ上で SPLASH-2 の LU 分解を走らせて、この手法が有効であることを確認した。今後は SSS-CORE 上のランタイムを完成させ、手続き間で静的にコアレシングを行なう部分を実装し、より多くのアプリケーションを走らせて更なる性能評価を行なう予定である。

参 考 文 献

1) BERSHAD, B. N., ZEKAUSKAS, M. J. and SAWDON, W. A. The Midway Distributed Shared Memory System, Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93) (Feb. 1993).
2) EMAMI, M., GHIYA, R. and HENDERN, L. J. Context-Sensitive Interprocedural Points-To Analysis in the Presence of Function Pointers, Proc. of '94 Conf. on PLDI (June 1994).
3) IFTODE, L., DUBNICKI, C., FELTEN, E. W. and LI, K. Improving Release-Consistent Shared Virtual Memory using Automatic Update, Proc. of the 2nd Inter. Symp. on HPCA (Feb. 1996).
4) KELEHER, P., COX, A. L. and ZWAENEPOEL, W. Lazy Release Consistency for Software Distributed Shared Memory, Proc. of the 19th ISCA (May 1992).
5) KELEHER, P., DWARKADAS, S., COX, A. L. and ZWAENEPOEL, W. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proc. of the Winter 1994 USENIX Conference (Jan. 1994).
6) LI, K. IVY: A Shared Virtual Memory System for Parallel Computing, Proc. of the 1988 ICPP (Aug. 1988).
7) SCALES, D. J., GHARACHORLOO, K. and THEKKATH, C. A. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, Proc. of 7th Int. Conf. on ASPLOS (Oct. 1996).
8) WILSON, R. P. and LAM, M. S. Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of '95 Conf. on PLDI (June 1995).
9) 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).
10) ZEKAUSKAS, M. J., SAWDON, W. A. and BERSHAD, B. N. Software Write Detection for a Distributed Shared Memory, Proc. of the 1st Symp. on OSDI (Nov. 1994).
11) 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).
12) 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).
13) 松本 尚, 平木 敬 汎用並列オペレーティングシステムにおける資源保護と仮想化, 情報処理学会研究報告, 第 97-OS-75 巻 (June 1997).
14) 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コンパイル技法, 情報処理学会研究報告, 第 97-HPC-67 巻 (Aug. 1997).