配列間パディングを用いた粗粒度タスク並列処理のためのキャッシュ最適化

石坂一久[†] 小幡元樹[††] 笠原博徳[†]

マルチプロセッサシステムの普及に伴い自動並列化コンパイラの重要性が高まっている．従来自動並列化コンパイラの研究では，ループ並列処理を中心とした研究が行われてきたが，システムの実効性能を向上させるためには，ループ並列処理に加えループやサブルーチン間の並列性を利用する粗粒度タスク並列処理，ステートメント間の並列性を利用する近細粒度並列処理を階層的に利用するマルチグレイン並列処理が注目されている．また，プロセッサとメモリの速度差の増大によりメモリアクセスレイテンシが性能向上のボトルネックになっている．したがって，データローカリティ最適化によるキャッシュメモリの有効利用も性能向上の重要なファクタとなる．本論文では粗粒度タスク並列処理の性能の向上を目指した粗粒度タスク間キャッシュ最適化手法について述べる．本手法では，データローカライゼーション手法を用い，データおよびタスクをキャッシュサイズにフィットするように分割し，同一データにアクセスするタスクを連続的に実行できるようにプログラムの実行順序を変えた上で，配列間パディングを用いデータレイアウトを変えることによって，連続実行される粗粒度タスク間でのコンフリクトミスを削減する．本手法の性能をキャッシュ構成の異なる二台の 4 プロセッサマルチプロセッサワークステーション Sun Ultra 80（ダイレクトマップ）と IBM RS/6000 44p-270（セットアソシアティブ）上で性能評価を行った．Ultra 80 ではネイティブコンパイラ Sun Forte 6 update 2 の自動並列化の最高性能に対して SPEC CFP95 の tomcatv で 5.1 倍，swim で 3.3 倍，hydro2d で 2.1 倍，turb3d で 1.1 倍の性能向上が得られた．また RS/6000 では IBM XL Fortran 7.1 コンパイラに対して，tomcatv で 1.7 倍，swim で 4.2 倍，hydro2d で 2.5 倍，turb3d で 1.03 倍の性能向上が得られた．

Cache Optimization for Coarse Grain Task Parallel Processing using Inter-Array Padding

Kazuhisa Ishizaka,[†] Motoki Obata[††] and Hironori Kasahara[†]

Importance of automatic parallelizing compilers is getting larger with the widespread use of multiprocessor system. To improve the performance of multiprocessor system, currently multigrain parallelization is attracting much attention. In multigrain parallelization, coarse grain task parallelisms among loops and subroutines and near fine grain parallelisms among statements are used in addition to the traditional loop parallelism. The locality optimization to use cache effectively is also important for the performance improvement. This paper proposes inter-array padding for data localization to minimize cache conflict misses over loops. The proposed padding scheme was evaluated on the two commercial 4 processors workstations, namely Sun Ultra 80 and IBM RS/6000 44p-270, which have different cache configuration. Compared with the maximum performance of Sun Forte 6 update 2 compiler automatic loop parallelization on Ultra 80, the proposed padding with data localization gave us 5.1 times speedup for SPEC CFP95 tomcatv, 3.3 times for swim, 2.1 times for hydro2d, 1.1 times for turb3d. On IBM RS/6000 44p-270, it shows 1.7 times speedup for tomcatv, 4.2 times for swim, 2.5 times for hydro2d, 1.03 times for turb3d against automatic parallelization by IBM XL Fortran 7.1 compiler.

1. はじめに

現在マルチプロセッサアーキテクチャはハイパフォーマンスコンピュータをはじめエントリーレベルサーバー，デスクトップワークステーション，ゲーム機等で幅広く利用されるようになっている．そのようなマルチプロセッサシステムの実効性能やユーザーの使い易さを向上させるには，高性能の自動並列化コンパイラが重要である．自動並列化コンパイラの研究は従来から行なわれておりイリノイ大学の Polaris コンパイ

† 早稲田大学
Waseda University
†† 日立製作所システム開発研究所
Hitachi, Ltd., Systems Development Laboratory

ラ[1] はシンボリック解析，実行時データ依存解析，レンジテストのような依存解析手法を用いたループ並列化，またスタンフォード大学の SUIF コンパイラ[2] はユニモジュラ変換およびアフィンパーティショニング等を用いたデータローカリティ最適化を考慮したループ並列処理を研究している[3]．

しかし，マルチプロセッサシステムの最高性能と実効性能の差は拡大しており，今後さらに実効性能を向上させるためには，従来のループ並列化に加えてループ間やサブルーチン間といった粗粒度タスクレベルの並列性や，基本ブロック内での複数命令・ステートメント間の近細粗粒度並列性を利用することが重要である．早稲田大学の OSCAR コンパイラはループ並列性に加え，ループ，サブルーチンなどの粗粒度タスク間の並列性や，ステートメント間の並列性を利用するマルチグレイン並列処理を提案している[4]．

また，プロセッサとメモリの速度差の拡大によるメモリアクセスコストの増大や共有メモリへのアクセス集中による速度低下を避けるため，メモリ階層を有効利用するための最適化は性能向上に欠かすことができない．特にキャッシュ最適化に関する研究は従来より多く行われている．

コンパイラによるキャッシュ最適化では，ループインターチェンジ，ループ融合，タイリングなどのループリストラクチャリングによりデータアクセスパターンを変更する手法や，配列の次元入れ替えなどによって各プロセッサがアクセスするデータを連続的にする手法が提案されている．また，単一ループもしくはループ融合されたループを対象として，同一ループイタレーション内で起るキャッシュミスやループのイタレーション間でのキャッシュミスを削減するための配列内および配列間パディング手法も提案されている[5]．

最内側ループを対象にメモリ上でのならび順を他の配列と同一ループで利用されることが多い順とした上で，配列間にパディングを行ないコンフリクトミスを削減する手法[6] やコンフリクトミスの削減をパディングによって行うことでタイリングと独立させ，タイリング時にはコンフリクトミスを考慮することキャッシュの利用効率が最大となるようにタイルサイズの決定を可能する手法[7] も提案されている．

ループイタレーションのシフトを用いたループ融合，融合後のループに対するピーリングを用いたループ並列処理，及びループ内でアクセスされる配列をキャッシュ上に均等に分散させることによってコンフリクトミスの削減をする手法も提案されている[8]．

また物理アドレスキャッシュでは OS による論理アドレスから物理アドレスへのページ変換の方式がキャッシュ性能に影響を与えることが知られており，Page Coloring や Bin Hopping などの基本的な変換手法やそれらの拡張が研究されてきた[9]．コンパイラから得られるプログラムのアクセスパターン情報を用いてハードウェアの拡張なしに Page Coloring を行う OS とコンパイラが協調した手法も研究されている[10]．

本論文では，粗粒度タスク並列処理におけるデータローカライゼーションを伴うパディングを用いたキャッシュ最適化手法について述べる．本手法では，まず複数ループ間のデータ依存を解析し，各ループでアクセスするデータサイズがキャッシュサイズにフィットし，データ依存する分割後の小ループ間でのデータ授受がキャッシュを介して行われるように，それらのループを整合して分割する整合分割手法[11] を用いてデータ分割を行う．次に分割により生成された小ループを粗粒度タスクとして扱いそれらの最早実行可能条件を解析する．分割ループの中で大量の共有データを持つループは，粗粒度タスクスケジューリングによって同一プロセッサ上で連続実行される．さらに，パディングを用いたデータレイアウト変換を適用することにより，連続実行される粗粒度タスク間でのラインコンフリクトミスを削減する．

従来のコンパイラによるデータレイアウト変換は，単一ループもしくはループ融合後のループを対象としているのに対し，本手法では粗粒度タスク並列処理においてループ分割，連続実行を行った上でパディングを行うことで，オリジナルプログラム中で離れた位置にある複数ループ間でのグローバルなキャッシュ最適化を行うことが特徴となっている．また本手法は特別なハードウェアや OS の拡張を必要とせず，コンパイラのみで実現が可能である．

以下本論文では，2 章で粗粒度タスク並列処理手法について，3 章でデータローカライゼーション手法について，4 章でデータレイアウト変換手法について述べる．5 章では本手法の商用マルチプロセッサ上で行った性能評価について述べる．

2. 粗粒度タスク並列処理

粗粒度タスク並列処理ではソースプログラムを，粗粒度タスク（マクロタスク）に分割する．マクロタスクの種類には，単一の基本ブロックあるいはスケジューリングオーバーヘッドを考慮し複数の小基本ブロックを融合した疑似代入文ブロック（BPA），並列性向上のため単一の基本ブロックを分割して生成される疑似代入文ブロック，DO ループまたは IF 文による分岐

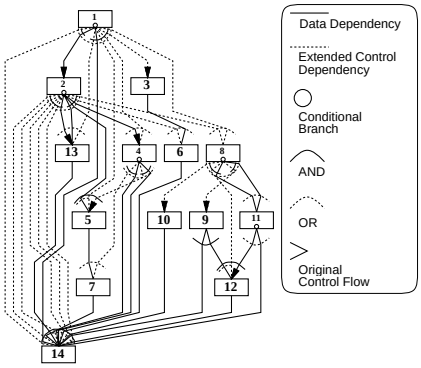


図 1 マクロタスクグラフ
Fig. 1 Macro Task Graph

よって生成される繰り返しブロック（RB），サブルーチンブロック（SB）がある．なお，マルチグレイン並列処理では，ステートメント間や命令間の並列性を利用する近細粒度並列処理もあるため，区別のため本論文では基本ブロックも粗粒度タスクと呼ぶ．

さらに，繰り返しブロック（RB）のうち，データ依存等により DOALL 処理ができないシーケンシャルループや IF 文（後方条件分岐文）によって作られるループ，またサブルーチンに対しては，必要に応じて階層的に内部をサブマクロタスクに分割する．

2.1 マクロタスクグラフの生成

コンパイラは各階層においてマクロタスク間のコントロールフローとデータ依存を解析する．それらの情報をもとにマクロタスク間の並列性を抽出するために，各マクロタスクに対して最早実行可能条件解析を行う．マクロタスクの最早実行可能条件とは，そのマクロタスクが最も早い時点で実行可能になる条件である．

各階層のマクロタスクの最早実行可能条件は，**図 1** に示すようなマクロタスクグラフ（MTG）で表わされる．マクロタスクグラフにおけるノードはマクロタスクを表し，ノード内の小円はマクロタスク内の条件分岐を表している．また，実線のエッジはデータ依存を表し，点線のエッジは拡張されたコントロール依存を表す．拡張されたコントロール依存とは，通常のコントロール依存だけでなく，データ依存とコントロール依存を複合的に満足させるため先行ノードが実行されないことを確定する条件分岐を含んでいる．また，実線アークはアークによって束ねられたエッジが AND 関係にあることを，点線アークは束ねられたエッジが OR 関係にあることを示している．

2.2 マクロタスクスケジューリング

粗粒度タスク並列処理では，マクロタスクのプロセッサへの割り当てには，スタティックスケジューリングまたはダイナミックスケジューリングが用いられる．

スタティックスケジューリングは，マクロタスクグラフがデータ依存エッジのみを持つ場合に適用され，コンパイル時にマクロタスクのプロセッサへの割り当てを決定し，スケジューリング結果に従って各プロセッサ用に異なった並列化コードを生成する．スタティックスケジューリングでは，実行時スケジューリングのオーバーヘッドを無くし，データ転送オーバーヘッド，同期オーバーヘッドを最小化することができる．

一方，マクロタスクグラフが条件分岐などの実行時不確定性を持つ場合は，実行時にマクロタスクを割り当てるダイナミックスケジューリングを用いる．ダイナミックスケジューリングを適用した場合，コンパイラは実行時スケジューリングのためのスケジューリングルーチンを生成し，マクロタスクコードと共に出力する並列化コードに埋め込む．OSCAR コンパイラでは，ダイナミックスケジューリングルーチンをユーザープログラムとして生成し，それを粒度の大きな粗粒度タスクのスケジューリングに用いることによって，実行時スケジューリングのオーバーヘッドを相対的に低くしている．

2.3 コード生成

OSCAR コンパイラは複数のバックエンドを持ち各ターゲット用にコード生成を行うことができる．本論文では OpenMP バックエンドを用いて，粗粒度タスク並列処理を実現した OpenMP FORTRAN を出力する．マシンコードを出力する方法に比べ，ポータビリティを高めることが可能で，出力コードを OpenMP をサポートする各種マルチプロセッサ上で実行，性能評価することが可能となる．

また出力した OpenMP FORTRAN を各マシン上のネイティブコンパイラに通すことにより，ネイティブコンパイラによる一般最適化やマシン固有の最適化を行うことが可能であるため，各マシン上でネイティブコンパイラによる自動並列化結果との性能比較も可能となる．

3. 粗粒度タスク間キャッシュ最適化

本章では，キャッシュの有効利用により，粗粒度タスク並列処理の性能を向上させる手法について述べる．

プログラムの持つデータローカリティを利用して粗粒度タスク並列処理の性能を向上させる手法として，従来よりデータローカライゼーション手法[11] が提案されている．同手法は，コンパイラにより制御可能な OSCAR アーキテクチャ[12] のローカルメモリや分散共有メモリ，SMP マシンのデータキャッシュを対象と

している．本論文ではキャッシュを対象としたローカライゼーションを用いるが，キャッシュはハードウェアでコントロールされるため，共有データの生死解析に基づくデータのリプレースを自由に制御する方式ではなく，ダイレクトマップキャッシュやLRUリプレース方式のセットアソシアティブキャッシュでも有効に働くデータ分散及びスケジューリング手法を用いる．

提案手法では，複数のループを分割することにより，分割ループによりアクセスされる合計データサイズがキャッシュに収まるようにし，さらに分割されたループのうち大量の共有データを持つループを同一プロセッサで連続的に実行することによって，データローカリティを高めキャッシュミスを削減する．

3.1 ループ整合分割

キャッシュサイズと比較して大きなデータを使うループでは，そのループの前半でアクセスしたデータを自分自身でキャッシュから追い出してしまうため，同一配列データにアクセスする後続ループの実行時にキャッシュミスが生じ，キャッシュの効率的な利用ができない．このようなキャッシュ利用効率の低下を避けるため，粗粒度タスク間キャッシュ最適化では，まずデータアクセス量の多いループに対して**ループ整合分割**[11]を用いて，複数のループ間のデータ依存を考慮しながら，アクセスするデータがキャッシュサイズより小さい小ループに分割する．

図2(a)に示すマクロタスクグラフで，マクロタスク2，3，7がキャッシュサイズより大きなデータを使用する並列処理可能ループであるとする．これらのループに対してループ整合分割を適用して，それぞれ4つの小ループに分割したときのマクロタスクグラフが図2(b)である．例えば図2(a)のマクロタスク2は，図2(b)のマクロタスク2_A，2_B，2_C，2_Dに分割されている．

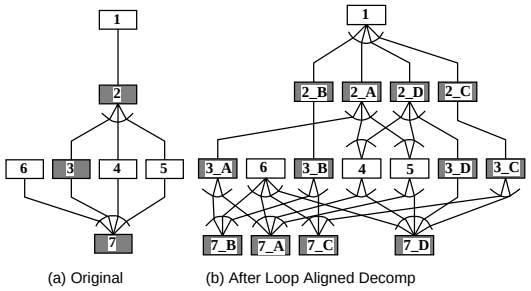


図2 ループ整合分割
Fig. 2 Loop Aligned Decomposition

ループ整合分割によって生成されたループもまたマクロタスクとして扱われ最早実行可能条件が解析される．さらにマクロタスクグラフ上でデータ依存エッジで結ばれたデータ共有量の多いマクロタスク群はデータローカライゼーショングループ（DLG）にグループ化される．

図2(b)上では，マクロタスク番号中に同一のアルファベットを持つマクロタスクが，同一DLGにグループ化されていることを示している．たとえば，マクロタスク2_A，3_A，7_Aは DLG_Aに属している．

3.2 マクロタスクの連続実行

図2(a)のマクロタスクグラフのオリジナルプログラム上でのマクロタスクの実行順番は，マクロタスク番号の増加順である．分割後のマクロタスクグラフで表すと，ループ整合分割で生成されたマクロタスクのオリジナルプログラム上での実行順はアルファベット順であるので，例えばマクロタスク2_Aから3_Dまでのオリジナルプログラム上での実行順は，2_A, 2_B, 2_C, 2_D, 3_A 3_B, 3_C, 3_Dであり，同一DLGに属するマクロタスクが連続実行されない．しかし，図2(b)の分割後のマクロタスクグラフで表されている最早実行可能条件を見ると，例えばマクロタスク3_Bは同一DLGのマクロタスク2_Bのみにデータ依存するため，2_Bの実行終了直後に3_Bの実行を行うことが可能であることが分かる．

粗粒度タスク間キャッシュ最適化では，粗粒度タスクスケジューリングにおいて，最早実行可能条件とDLGを利用し，同一DLGに属するマクロタスクを可能な限り同一プロセッサ上で連続実行し，キャッシュの利用効率を向上するようにマクロタスクスケジューリングを行う．このため，ダイナミックスケジューリングに対する拡張である**パーシャルスタティック割り当て**[13]や，スタティックスケジューリングに対する拡張[14]が用いられる．

図2(b)に示すマクロタスクグラフをシングルプロセッサに対して本連続実行方式でスケジューリングを行った例を図3に示す．オリジナルプログラムでの実行順では，同一DLGに含まれるマクロタスクは連続して実行されないため，キャッシュ利用効率が悪いが，本手法を適用した場合は，図3の DLG_Bに含まれるマクロタスク2_B，3_B，7_Bや DLG_Cの2_C，3_C，7_Cなどに示されるように，同一DLGに含まれるマクロタスクが連続して実行されるため，キャッシュを効果的に利用することができる．

4. コンフリクトミスの削減

本節では，同一DLGに属するマクロタスク間での

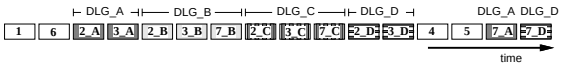


図 3 シングルプロセッサへのスケジューリング例
Fig. 3 Scheduling Result on Single Processor

コンフリクトミスを削減するためのパディングを用いたデータレイアウト変換手法について述べる．

4.1 DLG 内キャッシュコンフリクト

データローカライゼーションでは，一 DLG がアクセスするデータ（以下 DLG 内データ）のサイズがキャッシュサイズに収まるようにループを分割し，同一 DLG に属するマクロタスク（分割後のループ）は同一プロセッサ上で連続実行される．したがって，キャパシティミスが起こる前にデータの再利用を行うことが可能となるが，それらのマクロタスクでアクセスされるデータがキャッシュ上の同一セットに割り当てられる場合は，ダイレクトマップや連想度が小さいキャッシュではデータサイズがキャッシュサイズを越えていないにも関わらず，ラインコンフリクトミスによりデータがキャッシュから追い出される．

SPEC CFP95 ベンチマークの swim の 513 × 513 要素の各 1MB の二次元配列 13 個を 4MB のダイレクトマップのキャッシュに割り当てた場合のイメージを**図 4** に示す．図 4 中の太線のボックスが各配列を表し，ボックス中の文字列は配列名を表す．水平方向はキャッシュサイズを表し，先頭アドレスからキャッシュサイズ（4MB）を越える毎に一段下げて図示してある．すなわち，垂直方向に同一の位置にあるアドレスは同一ラインに割り当てられ，コンフリクトする可能性があることを示している．

図中の点線は，プログラム中のループを 4 分割した場合の各 DLG でアクセスされるデータの分割位置を示しており，灰色に塗ってある部分は先頭 DLG 内のループがアクセスする範囲を示している．図から分かるように，同一 DLG のループでアクセスする配列間でコンフリクトミスが起る配置となっている．このようなコンフリクトミスは，DLG 内マクロタスクの連続実行によるデータ再利用を無効にしてしまう．

なお，図 4 では各配列の先頭アドレスが同一キャッシュセットに割り当てられ，同一ループイタレーション内でコンフリクトミスが起るように見えるが，これは図の縮尺の関係であり実際には同一イタレーション内でのコンフリクトミスは起っていない．なぜなら，swim の各配列は単精度の 513 × 513 要素すなわち $4 \times 513 \times 513 Byte$ = 約 $1028 KByte$ であるため，例えば先頭配列 U と 5 番目の配列 VNEW との

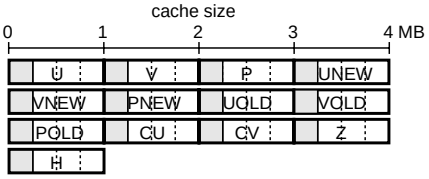


図 4 swim のキャッシュイメージ
Fig. 4 Cache Image of swim

キャッシュ上での距離は，間にある 4 配列の合計サイズとキャッシュサイズより $4 \times 1028KB - 4MB$ = 約 $16KB$ であるが，これは一般的なキャッシュラインサイズより十分大きいため，実際には一イタレーション内でアクセスされる配列要素は同一キャッシュセットには割り当てられていないからである．すなわち，本手法ではループイタレーション内のコンフリクトミスの削減ではなく，複数ループ間でのキャッシュミスの削減を行っている．

4.2 DLG 内コンフリクトミスの削減

多くの科学技術計算で見られるような複数のループにおいて配列を同一ストライドで同一方向に均等にアクセスしていくようなアクセスパターンやアクセスパターンがアフィン関係にある複数のループに対しては，ループ整合分割を適用可能である．ここで，ループ整合分割可能条件は，ループ間において同一配列の分割次元が一致し，その分割次元の配列添字がループ制御変数の 1 次式で表されており，かつ，ループ間にデータ依存を導く各配列に対して，配列添字中のループ制御変数係数のループ間での比が一定の場合である[11]．したがって，整合分割適用されたループに対しては，分割後のループのアクセスする配列範囲がコンパイルに推定可能であるため，分割により DLG 内ループがアクセスするデータサイズをキャッシュサイズ以下とした上で，提案するパディング手法によってデータレイアウトを変換することで，連続実行される DLG 内データのコンフリクトミスを削減する．なお，粗粒度タスク並列処理のタスクスケジューリング方式がスタティックスケジューリング，ダイナミックスケジューリングのどちらの場合でも，同一 DLG 内のループは同一プロセッサ上で連続実行されるようにスケジューリングされるため，どちらのスケジューリング手法に対しても同一のパディング手法が用いられる．

プログラムの実行フェーズによってアクセスパターンが異なる場合は，DLG 内データに重なりがあるフェーズを対象にパディングを行う．なぜならば，そのフェーズでの配列アクセスにおけるコンフリクトミスが性能向上のボトルネックとなっていると考えられ，その他

のアクセスパターンをもつフェーズにおいてはコンフリクトミスがもともと少ないと考えられるからである．本手法では DLG 内データにコンフリクトが生じるかを想定してからパディングを行っているため，そのような実行フェーズがないと判定されたプログラムについてはパディングは行われない．また，実行フェーズによってアクセスする配列が異なる場合は，全配列を対象に配列サイズがキャッシュにフィットするようにループ分割を行い，分割後の各 DLG 内データのコンフリクトを削減するようにパディングを行う．全配列間でコンフリクトを削減すれば，各実行フェーズでアクセスされる配列間でのコンフリクトも削減される．

また，本手法は配列アクセスの最適化であるため，プログラミング言語に依らず科学技計算に多く見られるような，本手法の対象となるアクセスパターンを持つアプリケーションには本手法を適用可能である．

以下では，DLG 内データのコンフリクトを削減するためのパディング手法について述べる．

4.3 配列間パディング

本節ではローカル変数の配列および FORTRAN 言語の共通ブロック☆ に属する配列で全てのプロシージャで同一サイズである配列に対するパディング手法を述べる．これらの配列に対しては，配列の宣言サイズを越えてアクセスがされない場合は，配列間にパディングを行ってもプログラムの意味を壊さない．そこで，配列間にパディングを行うことによって，同一ラインに割り当てられていた配列間の相対位置をずらしてコンフリクトミスを削減する．

また，プロシージャによって配列宣言サイズが異なる共通ブロックに対しては，配列間にパディングを行うとプログラムの意味を変えてしまう可能性があるため，配列間パディングを行うことは困難である．そのような場合のパディング手法は 4.4 節で述べる．

ステップ 1　対象配列グループの選択

本手法を実装した OSCAR コンパイラは逐次 FORTRAN プログラムを並列化して OpenMP FORTRAN プログラムを生成する．したがって，配列のメモリ割り当ては OpenMP FORTRAN からマシンコードを生成するネイティブコンパイラによって行われるため，OSCAR コンパイラは実際に配列をどのような順番でメモリに割り当てるかを決定することはできない．そのため，配列の割り当て順番がネイティブコンパイラによって変えられた場合も本手法によるデータレイアウト変換を有効とするため，同一サイズの配列を対象配列として選ぶ．

これは，ネイティブコンパイラによってメモリ割り当てが行われるという制約によるものであり，OSCAR コンパイラでマシンコードを生成する場合には，この制約をはずすことが可能である．また，科学技術計算では同一サイズの配列が利用されることが多いことから，この条件により本手法の一般性が大きく失われることはない．

ステップ 2　キャッシュ割り当てイメージの計算

選択された対象配列グループ内の配列のキャッシュ上での相対アドレスを求め，キャッシュ上での各配列の割り当てイメージを求める．ここでは前述のように大きさの等しい配列を対象としているため，図 4 に示すように，単純に宣言順に連続に割り当てていくことでアドレスを求める．

ステップ 3　最小分割数の決定

対象配列の合計サイズをキャッシュサイズで割ることにより，一 DLG で利用されるデータサイズをキャッシュサイズ以下にするための各ループの最小の分割数 div_num を求める．図 4 にあげた例では，総データサイズ 13 MB に対してキャッシュサイズは 4MB なので $div\_num = ceil(13/4) = 4$ である．

ステップ 4　最大 DLG アクセスサイズの決定

ループでは各配列を先頭から末尾へ均等にアクセスすると想定すると，配列サイズを div_num で割ったサイズ $part\_size$ が一つの DLG によってアクセスされる各配列の最大の大きさになる．

ステップ 5　コンフリクト判定

キャッシュイメージ上で各配列の $part\_size$ 部分（図 4 の薄い灰色部分）に重なりがある場合は，同一 DLG でアクセスされる部分配列間でコンフリクトが起ることを示している．コンフリクトが起らない場合は手順は終了してパディングは行わない．

なお，ここでは最小分割数 div_num で分割した場合でコンフリクト判定を行ったが，それ以上の分割数で分割した場合の各 DLG がアクセスする部分配列は，最大 DLG アクセスサイズ $part\_size$ より小さくなるため，最小分割数のときに重なりが無ければ，それ以上の分割数の場合もコンフリクトは生じない．

ステップ 6　パディングサイズの決定

重なり部分をなくすためには，各配列をキャッシュ上に割り当てた際に，キャッシュサイズを越えて下段に折り返した次の配列（例えば図 4 の配列 VNEW）のキャッシュ上での先頭アドレスが，一段上の配列（例えば図 4 の U）の先頭アドレスから $part\_size$（図 4

☆ FORTRAN の共通ブロックは，1 個以上の変数の順序付けられた記憶列であり，異なるプロシージャ間で共有される．

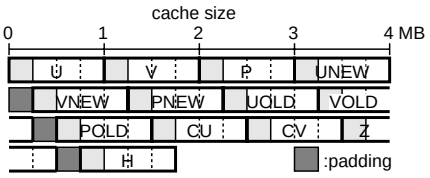


図 5 swim のパディング位置
Fig. 5 Padding for swim

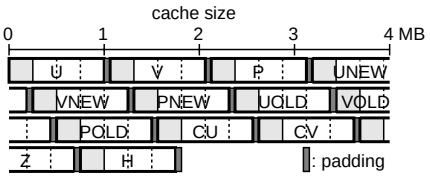


図 6 各配列サイズの変更によるパディング
Fig. 6 Padding by Changing Array Size

```
      SUBROUTINE TISTEP
        COMMON /VARH/VZ(MP,NP),VR(MP,NP),
     &    PR(MP,NP),TST(MN),DU3(4*MP+4*NP+4)
      SUBROUTINE TRANS1
        COMMON /VARH/VZ1(0:MP,NP),VR1(0:MP,NP),
     &    PR1(0:MP,NP),FL1(0:MP,NP),DU3(4*MP+4)
```

図 7 宣言形状の異なる共通ブロック（hydro2d）
Fig. 7 Different Declarations of a Common Block (hydro2d)

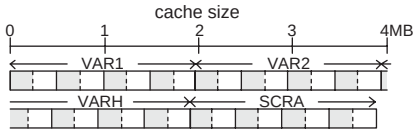


図 8 hydro2d のキャッシュイメージ
Fig. 8 Cache Image of hydro2d

の薄い灰色部分）だけ離れていれば良い．

したがって，先頭配列から順番に割り付けていった際に，キャッシュサイズ（$cache\_size$）を越えた次の配列の先頭アドレス（$base\_address$）が，($cache\_size + part\_size$) より小さい場合は，$padding\_size = (cache\_size + part\_size - base\_address)$ の大きさのパディングを行うことにより，キャッシュ上での配列の相対位置を $part\_size$ だけずらし，同一 DLG アクセス部分でのコンフリクトを削減する．以下同様に順番に配列を割り当てていき，次のキャッシュサイズでの折り返し時にも同様にパディングを行う．

図 4 に対して求められたパディングの結果を**図 5** に示す．図 5 では，キャッシュサイズで折り返した配列 UNEW, VOLD, Z の後ろにパディング（濃い灰色部分）を行っている．

ステップ 7　配列サイズの変更

ソースプログラム上でのパディングの実現方法には配列を共通ブロックに入れ，ダミー配列によってパディングを行う手法もあるが，ローカル変数をグローバル変数にすることによる性能への悪影響を避けるため，本手法では配列サイズを変更する手法をとる．

また，各配列のサイズを同一に保つために，ステップ 6 で求めたパディングサイズを各配列に分散させる．図 5 の例では，4 配列毎に $padding\_size$ のパディングを行っているので，各配列を $padding\_size/4$ だけ大きくするように，配列の最遅変化次元（FORTRAN の場合は最大次元）の大きさを増やす．**図 6** に各配列のサイズを変更させてパディングを実現させた場合のキャッシュイメージを示す．

4.4 共通ブロック間パディング

共通ブロックはプロシージャ毎に形状が異なることがある．このような共通ブロックに属する配列に対して，配列間にパディングを挿入するとプログラムの意味を変えてしまう可能性があるため，配列間パディングを行うのは困難である．

例えば SPEC CFP95 の hydro2d の共通ブロックの一つ VARH は**図 7** に示すように二つサブルーチンでは形状が異なる．ここで MP, NP はパラメータ文で定義されており全て同一の値である．この例では，サブルーチン TISTEP の先頭配列 VZ は 514560 バイトであるのに対して，サブルーチン TRANS1 での先頭配列 VZ1 は 515840 バイトである．

このように，共通ブロックの形がサブルーチン間で異なるため，前節で述べた配列間パディングは適用できないが，図 7 を見ると VARH はどちらのサブルーチンでも，ほぼ同じ大きさの四つの配列から構成されていることが分かる．同様に hydro2d の他の共通ブロック VAR1, VAR2, SCRA もそれぞれ VARH のそれとほぼ同じ大きさの四つの配列から構成されている．これらの共通ブロックを宣言の順番に VAR1, VAR2, VARH, SCRA の順にメモリ割り当てを行った際の，4MB のダイレクトマップのキャッシュ上でのイメージは**図 8** の様になる．図中の薄い灰色部分は，共通ブロックを構成する各配列を 2 分割したうちの前半の部分配列を示している．ここで分割数に 2 を選んだのは，四つの共通ブロックの合計サイズが約 7.9MB であるため，2 分割するとキャッシュサイズに収まるからである．

共通ブロックを構成する各配列がプログラム中の

ループで先頭から末尾へと均等にアクセスされると想定すると，各ループを 2 分割した場合の前半ループ（すなわち先頭 DLG）のアクセスする部分配列は，図 8 の薄い灰色部分になる．したがって，図で重なりがあることから分かるように，DLG 内データのコンフリクトが起ることになる．

実際のメモリ割り当ては，ネイティブコンパイラが行うので必ずしもソースコード上での宣言順に割り当てられるわけではないが，この順で割り当てられた場合はコンフリクトミスが起る．そこで，本手法ではネイティブコンパイラに共通ブロックの割り当て順を任せず，これらの共通ブロック（小共通ブロック）を結合して一つの共通ブロックを生成することで，小共通ブロックの割り当て順を決定する．また同時に，小共通ブロック間にダミー配列を挿入することによってパディングを行い，小共通ブロックを構成する配列間でのコンフリクトミスを防ぐ．以下に共通ブロック間パディングの手順を示す．

ステップ 1　配列サイズの推定

本手法ではほぼ同じ大きさの配列で構成されている共通ブロックを対象にするため，まず各共通ブロックが大きさのほぼ等しい配列のみから構成されているかどうかを判定する．構成されている場合，その配列の大きさを $common\_array\_size$ とする．

今回の実装では簡単のためプログラム中の実行文での共通ブロックのメンバ変数のアクセスパターンを調べずに，プログラムの全プロシージャでの変数宣言部のみ調べて求める．共通ブロックのメンバになっている配列のうち，ダミー変数（すなわちループ等の実行文で利用されない）と考えられる (1) 共通ブロック全体と同じ大きさの配列，(2) 共通ブロックの最後のメンバの配列を除いた全配列の大きさが，一定の差（現在の実装では 5%）以内に収まる場合に，その配列の大きさをその共通ブロックの $common\_array\_size$ とする．また，共通ブロック中のスカラー変数はサイズが小さいので無視する．

ステップ 2　対象共通ブロックグループ選択

大きさのほぼ等しい配列から構成されている共通ブロック中で，$common\_array\_size$ がほぼ等しい共通ブロックをグループ化して，共通ブロック間パディングの対象とする．hydro2d の共通ブロックの内 VAR1, VAR2, VARH, SCRA はほぼ同一の大きさの 4 つの配列から構成されており，対象共通ブロックグループに選ばれる．

ステップ 3　パディングサイズの決定

共通ブロックの合計サイズをキャッシュサイズで割って div_num を求め，また $common\_array\_size$ を div_num で割って $part\_size$ を求める．配列間パディングと同様に，対象共通ブロックグループに属する共通ブロックをキャッシュ上に割り当てたイメージを求め，キャッシュサイズをまたぐ共通ブロック（図 8 の VARH）の先頭アドレスが $cache\_size + part\_size$ となるように，共通ブロック間（図 8 の VAR2, VARH 間）にパディングを行う．hydro2d に共通ブロック間パディングを適用したキャッシュイメージを図 9 に示す．

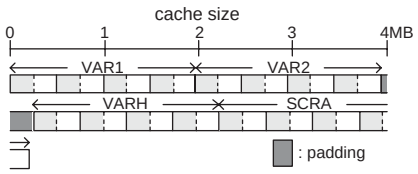


図 9　hydro2d での共通ブロック間パディング
Fig. 9　Padding for Common in hydro2d

4.5　セットアソシアティブキャッシュ

今回の性能評価では，セットアソシアティブキャッシュに対しても，同サイズのダイレクトマップキャッシュと同様として扱ってパディングを行った．n-way セットアソシアティブキャッシュのメモリイメージは，ダイレクトマップキャッシュのサイズが n 分の 1 になったものと考えられるが，ダイレクトマップ上で重なりを持たないようにパディングを行っていれば，n-way セットアソシアティブキャッシュでの重なりは n 以下になるため，キャッシュのリプレース方式が LRU の場合はコンフリクトが生じないと想定される．

4.6　OS のページ変換方式の影響

コンパイラによるデータレイアウト変換は論理アドレス上で行われるが，physically-indexed キャッシュでは，物理アドレスでキャッシングされるため，オペレーティングシステムの論理アドレスと物理アドレスのページ変換方式が重要な影響を及ぼす．

代表的なページ変換方式として Page Coloring と Bin Hopping が挙げられる．最も単純な Page Coloring では，論理アドレス上で連続するページに異なる色を連続的に割り当てるため，論理アドレス上で連続するページは異なる色をもつ物理ページへ変換されるためコンフリクトが起らないという特徴がある．一方，ちょうどキャッシュサイズ分だけ離れたページ間では同じ色をもつ物理ページに変換されるためコンフリクトが生じる．この方式では論理アドレス上でのアドレスの連続性は物理アドレス上でも保たれるため，コンパイラによるデータレイアウト変換が有効に働く．

表 1 性能評価環境
Table 1 Evaluation Environment

	Ultra 80	RS/6000 44p-270
プロセッサ	Ultra SPARC- II	Power3
周波数	450MHz	375MHz
プロセッサ数	4	4
L1D キャッシュ	32KB 1-way	64KB 128-way
L2 キャッシュ	4MB 1-way	4MB 4-way
主記憶	1GB	1GB
OS	Solaris 8	AIX 4.3
コンパイラ	Forte 6 update 2	XL Fortran 7.1

Bin Hopping では論理アドレスに関係なくページフォルトを起こした順番に連続する色を付けるため，連続してアクセスされるページ間でのコンフリクトを削減できるが，物理アドレス上での連続性が保たれないため，論理アドレス上でページサイズを越えたデータレイアウト変換を行なうことが難しい．

5. 性能評価

本章では提案するデータローカライゼーションを伴うパディング手法の性能について述べる．性能評価には**表 1** に示すように，4MB のダイレクトマップ L2 キャッシュを持つ 4 プロセッサ SMP ワークステーション Sun Ultra 80 と 4MB の 4-way セットアソシアティブキャッシュを持つ 4 プロセッサワークステーション IBM RS/6000 44p-270 を利用した．また両マシンの L2 キャッシュは physically-indexed キャッシュであるが，両 OS は Page Coloring と Bin Hopping の両方のページ変換方式をサポートしている．Ultra80 の Solaris 8 のデフォルトの方式は Page Coloring をベースとした Hashed VA であり，RS/6000 の AIX 4.3 のデフォルトは Bin Hoppoing である．Hashed VA では，連続する論理アドレスの連続する物理アドレスへの変換に加え，L2 キャッシュサイズ（4MB）毎に L1 キャッシュサイズとページサイズを考慮した少量のギャップを入れることにより，論理アドレス上でちょうど L2 キャッシュサイズだけ離れた二つのアドレス間でのラインコンフリクトミスを避けようとする方式である．ギャップサイズが少量であることから，この方式では論理アドレス上でのアドレスの連続性はほぼ保たれ，論理アドレス上でのデータレイアウト変換は有効に働く．

性能評価では，本手法を実装した OSCAR コンパイラを用いて，逐次 FORTRAN プログラムを自動並列化した結果を OpenMP FORTRAN プログラムとして出力し，その後マルチプロセッサマシン上のネイティブコンパイラを用いて実行ファイルを生成した．コンパイル方法としては，OSCAR コンパイラを用いてパディングを含むデータローカライゼーション手法を適用した場合とパディングを適用しないデータローカライゼーションに加え，ネイティブコンパイラのみによって自動並列化を行った際の最大性能と比較する．

また評価には SPEC CFP95 の tomcatv, swim, hydro2d, turb3d, mgrid の五つのプログラムを用いた．tomcatv, swim, hydro2d, mgrid については，SPEC のオリジナルソースコードを使用した．turb3d はサブルーチンコールを含むメインループを持ち，このループはインタープロシージャ解析を用いたループ並列化により並列化可能となる．しかし現在の OSCAR コンパイラではこの機能が実装されていないため，この機能を持つ APC コンパイラ[15] ☆を利用してループ並列化した結果を OpenMP で出力して，OSCAR コンパイラの入力としキャッシュ最適化を行った．Sun Forte コンパイラ，IBM XL Fortran コンパイラも同様に単独ではこのループ並列性を抽出できないため，APC コンパイラをプリプロセッサとして利用することにより，平等な条件で提案手法の評価を行った．

今回の性能評価では，対象としたプログラムのアクセスするデータサイズは L2 キャッシュサイズの倍以上であるため，ミス時のペナルティが大きく，処理性能に大きな影響を与える L2 キャッシュを対象に本手法を適用した．また，L2 キャッシュを対象とした場合，今回使用したプログラムのデータサイズは，いずれのマシン上でも全プロセッサの合計キャッシュサイズより小さいため，ループ分割数としてはプロセッサ台数を用い，スタティックスケジューリングを用いて各 DLG を各プロセッサに割り当てた．

tomcatv はメインプロシージャ単体からなるプログラムで，倍精度の 513×513 の約 2MB の二次元配列を 7 個使用し，合計データサイズは約 14 MB である．本手法ではこれらの配列間にパディングを適用した．4MB の L2 キャッシュを対象としたパディングでは，コンパイラにより各配列の宣言サイズは 513×576 に変更される．swim は単精度の 513×513 の約 1MB の配列を 13 個使用する．これらは共通ブロックに属する変数であるが，全プロシージャで同一の形をしているため配列間パディングを適用し，4MB の L2 キャッシュ

☆ APC コンパイラは，METI/NEDO ミレニアムプロジェクト IT21 Advanced Parallelizing Compiler によって開発されたコンパイラで，それぞれ単体でも動作可能な早稲田大学，日立，富士通の 3 コンパイラを OpenMP を中間言語として統合したコンパイラである．OSCAR コンパイラは早稲田大学により研究開発が行われており，APC コンパイラを構成する 3 コンパイラのうちのひとつである．

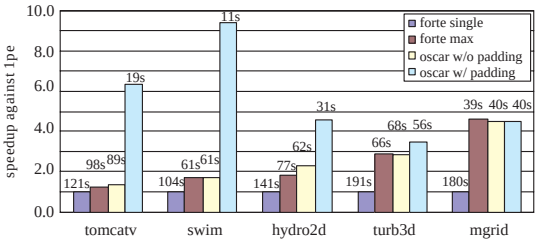


図 10 Speedups on Sun Ultra 80 (Hashed VA)
Fig. 10 Speedups on Sun Ultra 80 (Hashed VA)

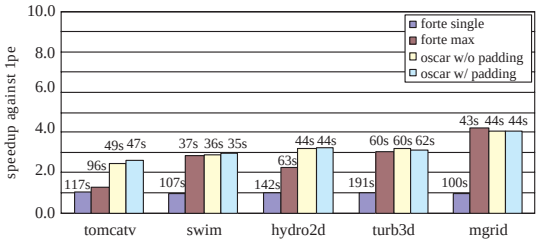


図 11 Speedups on Sun Ultra 80 (Bin Hopping)
Fig. 11 Speedups on Sun Ultra 80 (Bin Hopping)

に対して 513×544 の宣言サイズに変更した．hydro2d に対しては前述したように共通ブロック間パディングを適用した．4MB の L2 キャッシュに対しては，共通ブロック VAR1, VAR2, VARH, SCRA を結合して一つの共通ブロックとし，VAR2, VARH の間に 318696 バイトのダミー配列をパディングしてデータレイアウトを変更した．turb3d は倍精度の 66 × 64 × 64 要素の三次元配列 6 個すなわち合計約 12MB の配列にアクセスするが，ループ分割を行うと各プロセッサが配列をブロック分割した範囲をアクセスするアクセスパターンと，各プロセッサのアクセス範囲がインターリーブされているアクセスパターンが存在する．提案手法では前者のアクセスパターンにコンフリクトミスが生じていると推定し配列間パディングを適用し，配列サイズを 66×64×71 に拡張する．また，mgrid に対しては本手法ではコンパイル時にコンフリクトミスが生じると判定されないため，コンパイラはパディングを行わないコードを生成している．

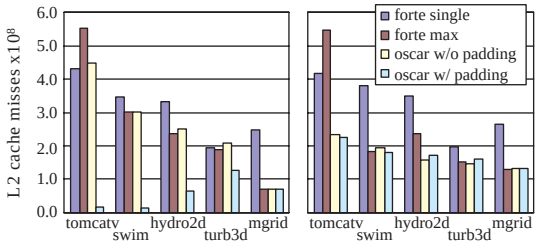
5.1 Sun Ultra 80 上での評価

Solaris 8 はページ変換方式として，デフォルトの Hashed VA に加え，V.addr=P.addr，Bin Hopping の三つを提供する[16]．V.addr=P.addr は論理アドレス上での連続アドレスは物理アドレス上でも完全に連続するように変換される方式である．デフォルトの Hashed VA も前述のように連続性は保たれるため，今回はデフォルトの Hashed VA と Bin Hoppoing の二つの方式での性能評価を行った．

5.1.1 Hashed VA 時の性能

Hashed VA 時の Sun Ultra 80 上で各コンパイル方法を適用した場合の，Forte コンパイラのみを用いた場合の逐次実行に対する速度向上率を**図 10** に示す．図中の棒グラフの上の数値は実行時間を表す．また Ultra SPARC-II の CPU Performance counter を用いて計測した L2 キャッシュミス回数を**図 12** に示す．

図 10 に示すように，tomcatv, swim, hydro2d の逐次実行時間は 121 秒，104 秒，141 秒であるのに対して，Forte コンパイラの自動並列化（図 10 forte max）を行った時の実行時間は 98 秒，61 秒，77 秒であり，それぞれ 1.2 倍，1.7 倍，1.8 倍の速度向上にとどまっている．またパディングを用いない場合の OSCAR コンパイラ（図 10 oscar w/o padding）での実行時間は 89 秒，61 秒，62 秒であり，速度向上率は 1.4，1.7，2.3 倍とプロセッサ台数による速度向上が得られていない．これはプロセッサ台数の増加によりキャッシュの合計サイズが増えるのにもかかわらず，前章で述べたようにコンフリクトミスのためにキャッシュミス回数が削減されず，キャッシュミスが実行時間の大半を占めているためと考えられる．例えば図 12(a) に示す swim のキャッシュミス回数を見ると，逐次実行（forte single）の場合のキャッシュミス回数は約 3.5 億回であるのに対して，Forte の自動並列化（forte max）では約 3.0 億回，パディングを用いない場合の OSCAR コンパイラ（oscar w/o padding）でも約 3.0 億回であり，逐次実行時の 86%までしか削減されていない．



(a) Hashed VA (b) Bin Hopping

図 12 L2 cache misses on Sun Ultra 80
Fig. 12 L2 cache misses on Sun Ultra 80

また，turb3d については，前述のように複数のアクセスパターンを持ち，コンフリクトミスが少ないアクセスパターンのプログラムの全実行時間に占める割合が他の 3 プログラムに比べて大きいため，並列実行時のキャッシュ利用効率が高くなっている．mgrid は OSCAR コンパイラによってコンフリクトミスが判定されなかったように，並列実行時においてコンフリク

トミスがボトルネックとならないため，性能向上が得られている．

Ultra 80 ではコンフリクトミスにより台数効果が妨げられているが，パディングを用いてコンフリクトミスを削減した OSCAR コンパイラ（図 10 oscar w/ padding）では，実行時間は tomcatv で 19 秒，swim で 11 秒，hydro2d で 31 秒，turb3d で 56 秒となった．逐次実行と比較するとそれぞれ 6.3 倍，9.4 倍，4.6 倍，3.4 倍の速度向上であり，台数効果が得られていることが分かる．また，パディングを用いない場合の OSCAR コンパイラ（図 10 oscar w/o padding）に対しては tomcatv で 4.7 倍，swim で 5.5 倍，hydro2d で 2.0 倍，turb3d で 1.2 倍と大幅な性能向上が得られており，パディングの効果が大きいことが確認された．パディングを用いたときのキャッシュミス回数は図 12(a) oscar w/ padding に示すように，tomcatv で用いない場合の 3.5%に，swim で 4.2%，hydro2d で 25%，turb3d で 61%に削減されており，パディングが速度向上に寄与していることがわかる．hydro2d の場合 tomcatv, swim と比べ速度向上率，キャッシュミス回数の削減率が悪いのは，プログラムの全実行時間に対する本手法の適用対象配列にアクセスするループの実行時間が短いことと共通ブロック間パディングであることが挙げられる．

また，turb3d の速度向上率が 1.2 倍にとどまっているのは，前述したようにアクセスパターンの異なるループが存在するためである．turb3d の実行時間のうち大部分を占める五つの連続する実行フェーズのキャッシュミス回数を**表 2** に示す．これらの 5 フェーズはメインループの内部に入っており繰り返し実行されるが，表に示したキャッシュミスは各フェーズの全実行のキャッシュミス回数の合計である．表の w/o padding 列がパディングを用いない場合のキャッシュミス回数，w/ padding 列がパディングを行った場合のキャッシュミス回数で，diff 列がその差で数値が大きいほど多くのキャッシュミスが削減されてことを示す．doall2-call uxw-doall3 のフェーズが提案手法の対象となるアクセスパターンを持つループであり，各フェーズでは各プロセッサが配列をブロック分割した領域をアクセスする．一方，doall1, doall4 は異なるアクセスパターンを持ち，各プロセッサのアクセス範囲がインターリーブされている．表から L2 キャッシュミスの削減がよく行われているのは，本手法の対象となったフェーズのうちの call uxw, doall3 であり，それぞれ 542 万回，536 万回のキャッシュミスが削減されており，キャッシュミス回数はパディングなしの場合の 4.1%, 10.0%になっている．これは doall2 の実行により各プロセッサのキャッシュにデータが乗り，call uxw, doall3 ではキャッシュ上のデータを再利用するためキャッシュミスの削減が削減されているからである．一方 doall1, doall4 は異なるアクセスパターンを持つため，doall2-call uxw-doall3 のフェーズとは異なるデータに各プロセッサがアクセスするので，パディングを用いた場合でもキャッシュミスが生じるため，削減回数はそれぞれ 50 万回，74 万回であり，キャッシュミスはパディングを行わない場合の 89%，88%までしか削減されていない．

表 2 turb3d の L2 キャッシュミス回数
Table 2 L2 cache misses of turb3d

	L2 cache misses		
	w/o padding	w/ padding	diff
doall1	4,680,561	4,178,363	502,198
doall2	9,852,230	9,115,141	737,089
call uxw	5,653,883	234,528	5,419,355
doall3	5,962,620	598,339	5,364,281
doall4	5,134,257	4,529,314	604,943

5.1.2 Bin Hopping 時の性能

次に**図 11** に示す Bin Hopping の場合を見ると，パディングを用いない場合の OSCAR コンパイラ（図 11 oscar w/o padding）の実行時間は，tomcatv で 49 秒，swim で 36 秒，hydro2d で 44 秒，turb3d で 60 秒であり，逐次実行に対する速度向上率はそれぞれ 2.4 倍，3.0 倍，3.2 倍，3.2 倍となっている．これを Hashed VA 時のパディングを用いない OSCAR コンパイラ（図 10 oscar w/o padding）と比較すると，それぞれ 1.8，1.7，1.4，1.1 倍 Bin Hopping の方が良い性能を示している．また Forte コンパイラの自動並列化同士で比較しても，Bin Hopping の方が Hashed VA に比べて良い性能となっている．これは Bin Hopping では論理アドレス上での連続アドレスが物理アドレス上では連続とならないため，論理アドレス上で想定したようなキャッシュコンフリクトが，物理アドレスでは起らないためである．

一方 Bin Hopping では，パディングを用いた場合の OSCAR コンパイラ（図 11 oscar w/ padding）の実行時間は，それぞれ 47 秒，35 秒，44 秒，62 秒であり，パディングを用いない場合の OSCAR コンパイラに比べて，数%の性能向上にとどまっている．これは，Bin Hopping では論理アドレス上でのパディングが物理アドレス上では Hashed VA のようには有効に働いてないためである．

パディングを用いた場合の Hashed VA の性能と Bin Hopping 時の最高性能を比較すると，例えば tomcatv

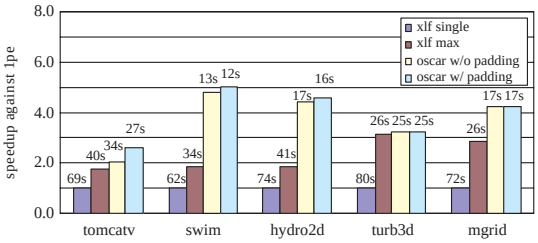


図 13 Speedups on RS/6000 44p-270 (Bin Hopping)
Fig. 13 Speedups on RS/6000 44p-270 (Bin Hopping)

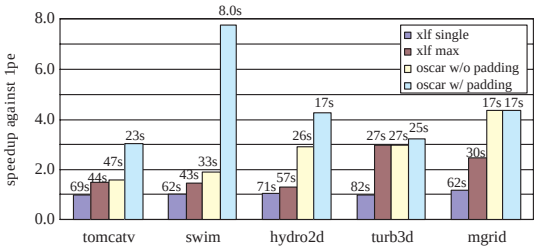


図 14 Speedups on RS/6000 44p-270 (Page Coloring)
Fig. 14 Speedups on RS/6000 44p-270 (Page Coloring)

の実行時間は，前者では 19 秒であるのに対して，後者では 47 秒であり，Hashed VA でパディングを適用した場合の方が 2.5 倍高い性能を示した．同様に swim では 3.2 倍，hydro2d では 1.4 倍，turb3d では 1.1 倍高い性能を示すことが分かり，Ultra 80 上では Hashed VA でパディングを利用した場合に最高性能が得られた．

5.2 IBM RS/6000 44p-270 上での評価

4-way セットアソシアティブの 4MB の L2 キャッシュを持つ IBM RS/6000 44p-270 での評価結果を**図 13**, **図 14** に示す．AIX 4.3 のデフォルトのページ変換方式は Bin Hopping であるが，Page Coloring も提供されているため，この両者で性能評価を行った．図の縦軸は Bin Hopping の場合の XL Fortran コンパイラ（以下 XLF コンパイラ）のみを用いて逐次処理をした場合に対する速度向上率を示している．本評価ではセットアソシアティブキャッシュに対してもダイレクトマップとして考えてパディングを行っており，各プログラムに対するパディングサイズは Ultra 80 の場合と同一である．

図 13 に示す Bin Hopping 時の OSCAR コンパイラでパディングを用いた場合の実行時間はそれぞれ 27 秒，12 秒，16 秒，25 秒であり，パディングを用いない場合の OSCAR コンパイラがそれぞれ 34 秒，13 秒，17 秒，25 秒であるのと比較すると，パディングによる性能向上は 27%，4.6%，3.2%，0.2%であった．turb3d は前述のようにアクセスパターンが異なるループが存在するため性能向上率が得られていないが，その他のプログラムでは数%から数十%の性能向上が得られた．

図 14 に示すページ変換方式を Page Coloring に変更した時の性能を見ると，パディングを行わない場合の OSCAR コンパイラの実行時間は 44 秒，33 秒，26 秒，27 秒と，Bin Hopping の場合に比べ遅くなっている．また，XLF の自動並列化でも Bin Hopping の方が Page Coloring より高速に処理を行うことができている．これは，Ultra 80 の場合と同様に，Bin Hopping では論理アドレス上で想定したコンフリクトミスが物理アドレス上では起らないためである．

一方 Page Coloring 時のパディングによる効果を見ると，パディングを用いた場合の OSCAR コンパイラの実行時間は 23 秒，8 秒，17 秒，25 秒であり，パディングを用いない OSCAR コンパイラに対して 2.0，4.1，1.5，1.1 倍の性能向上が得られた．

パディングを用いることにより Page Coloring 時の処理速度が向上し Bin Hopping の性能を上回り，RS/6000 上でも Page Coloring 時にパディングを用いることで最大性能が得られることが分かった．

6. ま と め

本論文ではデータローカライゼーションを用いた粗粒度タスク間キャッシュ最適化においてコンフリクトミスを削減するための配列間パディング手法について述べた．本手法では，複数のループをキャッシュサイズを考慮して分割し，分割後のループのうちデータを共有する複数ループを可能な限り同一プロセッサ上で連続実行させることにより，複数ループ間におけるローカリティを向上させる．さらに連続実行されるループ間でのコンフリクトミスを配列間パディングによって削減することによって，キャッシュの利用効率を向上させる．

本手法の性能評価をキャッシュ連想方式，OS，ページ変換方式の異なる複数のマルチプロセッサ上で行った．4MB のダイレクトマップ L2 キャッシュを持つ Sun Ultra 80 上での性能評価では，パディングを用いることで Sun Forte 6 update 2 コンパイラの自動並列化の最高性能に対して，SPEC CFP95 の tomcatv で 5.1 倍，swim で 3.3 倍，hydro2d で 2.1 倍，turb3d で 1.1 倍の性能向上が得られた．同様に 4MB の 4-way セットアソシアティブキャッシュである IBM RS/6000 44p-270 上でも Page Coloring 上で本手法を適用した場合が最高性能を示し，Bin Hopping の場合の最高

性能と比較すると，tomcatv で 1.7 倍，swim で 4.2 倍，hydro2d で 2.5 倍，turb3d で 1.03 倍の性能向上であった．

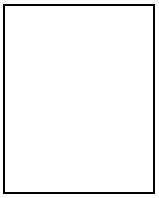
2 台のマルチプロセッサでの性能評価結果より，Page Coloring においてパディングを用いた粗粒度タスク間キャッシュ最適化によりコンフリクトミスが削減され速度向上が得られることが確かめられた．また，論理アドレス上での連続性が保たれない Bin Hopping では本手法の性能は数%にとどまったが，Page Coloring と Bin Hopping の両ページ変換方式を比較すると Page Coloring 上で本手法を適用した場合に各マシンの最高性能が得られることが分かった．

謝辞 本研究の一部は METI/NEDO ミレニアムプロジェクト IT21 “Advanced Parallelizing Compiler” 及び STARC（半導体理工学研究センター）の支援により行われた．

参 考 文 献

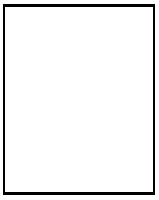
1) Eigenmann, R., Hoeflinger, J. and Padua, D.: On the Automatic Parallelization of the Perfect Benchmarks, *IEEE Trans. on parallel and distributed systems*, Vol. 9, No. 1 (1998).
2) Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S., Bugnion, E. and Lam, M. S.: Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer* (1996).
3) Lim, A.W., Liao, S.-W. and Lam, M.S.: Blocking and Array Contraction Across Arbitrarily Nested Loops Using Affine Partitioning, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2001).
4) 笠原, 小幡, 石坂: 共有メモリマルチプロセッサシステム上での粗粒度タスク並列処理, 情報処理学会論文誌, Vol. 42, No. 4 (2001).
5) Rivera, G. and Tseng, C.-W.: Eliminating Conflict Misses for High Performance Architectures, *Proc. of the 1998 ACM International Conference on Supercomputing* (1998).
6) Panda, P. R., Dutt, N. D. and Nicolau, N.: Memory Data Organization for Improved Cache Performance in Embedded Processor, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 2, No. 4, pp. 384–409 (1997).
7) Panda, P. R., Dutt, N. D., Nakamura, H. and Nicolau, A.: Augmenting Loop Tiling with Data Alignment for Improved Cache Performance, *IEEE Transactions on Computers*, Vol. 48, No. 2 (1999).
8) Manjikian, N. and Abdelrahman, T. S.: Array Data Layout for the Reduction of Cache Conflicts, *Proc. of 8th International Conference on Parallel and Distributed Computing Systems* (1995).
9) Kessler, R. E. and Hill, M. D.: Page Placement Algorithms for Large Real-Indexed Caches, *ACM transaction of Computer Systems* (1992).
10) Bugnion, E., Anderson, J. M., Mowry, T. C., Rosenblum, M. R. and Lam, M. S.: Compiler-Directed Page Coloring for Multiprocessors, *Proc. of the Seventh Internatinal Symposium of Architectural Support for Programing Languages and Operating Systems* (1996).
11) 吉田, 前田, 尾形, 笠原: Fortran マクロデータフロー処理におけるデータローカライゼーション手法, 情報処理学会論文誌, Vol. 35, No. 9, pp. 1848–1994 (1994).
12) Kimura, K. and Kasahara, H.: Near Fine Grain Parallel Processing Using Static Scheduling on Single Chip Multiprocessors, *Proc. of International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems* (1999).
13) 石坂, 中野, 八木, 小幡, 笠原: 共有メモリマルチプロセッサ上でのキャッシュ最適化を考慮した粗粒度タスク並列処理, 情報処理学会論文誌, Vol. 43, No. 4 (2002).
14) 中野, 石坂, 小幡, 木村, 笠原: キャッシュ最適化を考慮したマルチプロセッサシステム上での粗粒度タスクスタティックスケジューリング手法, 情報処理学会 ARC 研究報告 (2001).
15) APC: Japanese Millennium Project IT21 Advanced Parallelizng Compiler Technology Project, http://www.apc.waseda.ac.jp.
16) Mauro, J. and McDougall, R.: *SOLARIS Internal*, Prentice Hall (2001).

(平成 ? 年 ? 月 ? 日受付)
(平成 16 年 2 月 4 日採録)



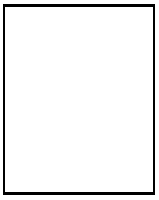
石坂 一久（学生会員）

昭和 51 年生．平成 11 年早稲田大学理工学部電気電子情報工学科卒業．平成 13 年同大学大学院修士課程修了．平成 13 年同大学大学院博士課程進学．平成 14 年同大学同学部助手，現在に至る．



小幡　元樹（正会員）

昭和 48 年生．平成 8 年早稲田大学理工学部電気工学科卒業．平成 10 年同大学大学院修士課程修了．平成 11 年同大学同学部助手．平成 13 年同大学理工学研究科博士課程修了．平成 14 年同大学理工学総合研究センター助手．工学博士．現在は株式会社日立製作所に勤務．在学中はマルチグレイン自動並列化コンパイラに関する研究に従事．



笠原　博徳（正会員）

昭和 32 年生．昭和 55 年早稲田大学理工学部電気工学科卒業．昭和 60 年同大学大学院博士課程修了．工学博士．昭和 58 年同大学同学部助手．昭和 60 年学術振興会特別研究員．昭和 61 年早稲田大学理工学部電気工学科専任講師．昭和 63 年同助教授．平成 9 年同大学電気電子情報工学科教授，現在に至る．平成元年〜2 年イリノイ大学 Center for Supercomputing Research & Development 客員研究員．昭和 62 年 IFAC World Congress 第一回 Young Author Prize．平成 9 年度情報処理学会坂井記念特別賞受賞．著書「並列処理技術」(コロナ社)．情報処理学会，電子情報通信学会，IEEE などの会員．