Vol.2014-ARC-213 No.19
Vol.2014-HPC-147 No.19
2014/12/9

# 自動並列化コンパイラによるソフトウェアキャッシュコヒーレンシ制御手法の評価

岸本 耀平[1,a)]　間瀬 正啓[1,†1]　木村 啓二[1]　笠原 博徳[1]

概要：主記憶共有型マルチコアプロセッサにおいて，一般にキャッシュコヒーレンシ制御はハードウェアにより実現されている．今後のプロセッサコア数の増加に伴いキャッシュコヒーレンシハードウェアの回路規模は大きくなり，チップへの実装が困難になること，電力消費が大きくなること，設計期間及び開発費用が増大することが懸念されている．本稿ではこのハードウェアコヒーレンシ制御の問題を解決するために，ハードウェアコヒーレンシ制御機構を持たない主記憶共有型ノンコヒーレントキャッシュマルチコアに対して，並列化コンパイラがソフトウェアに対し自動的にコヒーレンシ制御を行う手法を提案する．本手法を実装した OSCAR 自動並列化コンパイラと，4 コアのクラスタを 2 つ持ちクラスタ間ではハードウェアコヒーレンシを持たない情報家電用マルチコア RP2 を用い性能評価を行った．9 つの科学技術計算アプリケーションを対象として評価を行ったところ，4 コアのハードウェアコヒーレンシ制御使用時の性能は平均で 1 コア性能の 2.80 倍であったのに対し，ハードウェアコヒーレンシを使用せず本手法を適用した 4 コア実行時の性能は平均で 1 コア性能の 2.61 倍となりほぼ同等の速度向上が得られ，さらに 8 コアハードウェアコヒーレンシ制御無効時には平均で 1 コア性能の 3.66 倍とスケールアップすることが確認できた．

## 1. はじめに

より高い計算能力への要求及び半導体集積度の向上に伴い，チップに実装されるコア数は増加している．現在主流の主記憶共有型マルチコアプロセッサでは，一般にプロセッサ近傍に高速かつ小容量のメモリであるキャッシュメモリを搭載している．その際，現在の多くのマルチコアプロセッサではハードウェアによりキャッシュコヒーレンシ制御が実現されている．しかしながら，ハードウェアコヒーレンシ機構のチップへの実装は，コア数の増加に伴い消費電力の増大，ハードウェア設計コスト及び期間の増大が顕著になり，特に 64 コア以上のメニーコアでは困難になることが懸念されている．

このような課題に対処するため，ハードウェアコヒーレンシ機構を持たずソフトウェアによりコヒーレンシ制御を行うノンコヒーレントキャッシュアーキテクチャが提案及び実装されてきた [1][2][3]．一方で，ハードウェアコヒーレンシ機構が存在しない場合，陳腐化したデータ (Stale Data)[4] への参照，及び False Sharing が発生し，これらの問題に対処しない限りソフトウェアの実行が正常に行われない．これらの問題に対処するためにはソフトウェアにおいて複雑なコヒーレンシ制御のプログラミングを行う必要があり，ソフトウェア生産性に問題があった．

本稿では，自動並列化コンパイラによるソフトウェアキャッシュコヒーレンシ制御手法を提案する．さらにノンコヒーレントキャッシュアーキテクチャ上で提案手法の性能評価を行った結果について述べる．本手法では，コンパイラが自動並列化及びキャッシュ最適化を行うとともに，Stale Data への参照を回避するキャッシュ操作命令の挿入，及び False Sharing を回避するためのコード変形を自動で行う．すなわち，提案手法により手動でのソフトウェアの改変なしにノンコヒーレントキャッシュアーキテクチャ上でプログラムが動作可能となり，ソフトウェア開発者の負担が軽減される．

本手法を OSCAR 自動並列化コンパイラ [5] に実装し，8 コア集積の情報家電用マルチコアプロセッサ RP2[6] において性能評価を行った．RP2 は 4 コアクラスタ内はハードウェアコヒーレンシ機構を持つが，5 コア以上ではソフトウェアによるコヒーレンシ維持が必要な動作となる．

以下，2 章ではノンコヒーレントキャッシュアーキテクチャ，3 章ではコンパイラによるキャッシュコヒーレンシ制御手法について，4 章では評価について，5 章ではまとめについて述べる．

[1] 早稲田大学
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，日立製作所 中央研究所
Presently with Hitachi Central Research Laboratory
[a)] kisimoto@kasahara.cs.waseda.ac.jp

ⓒ 2014 Information Processing Society of Japan

## 2. ノンコヒーレント キャッシュアーキテクチャ

本章では本制御手法の対象とするハードウェアコヒーレンシ制御機構を持たないマルチコアアーキテクチャの仕様と，対象アーキテクチャでのコヒーレンシ制御の問題点について述べる．

### 2.1 対象アーキテクチャ

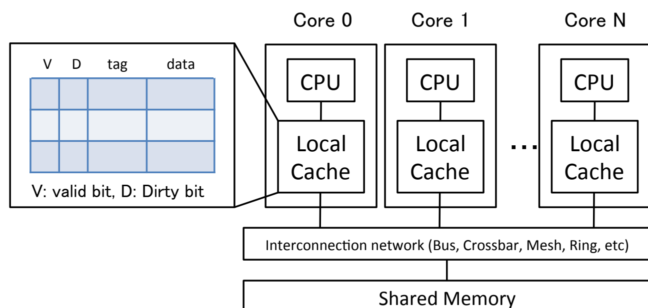


図 1 ノンコヒーレントキャッシュアーキテクチャ
**Fig. 1** Non-coherent cache architecture

本稿の対象とするアーキテクチャは，複数のプロセッサコアがコアごとのプライベートキャッシュを持つ主記憶共有型のシステムであるが，キャッシュ及びメインメモリのコヒーレンシを保つためのハードウェアコヒーレンシ機構を持たないことを仮定している．ソフトウェアでキャッシュコヒーレンシ制御を行う場合，キャッシュライン毎に Invalid, Valid, Dirty の 3 状態を管理する機能を持つものとする．Invalid は，当該キャッシュラインのデータは無効であることを示し，キャッシュラインの初期状態及び無効化が行なわれた後の状態である．Valid は，当該キャッシュラインのデータは利用可能であり，メインメモリ上の値と同一 (Clean) な状態である．Dirty は，当該キャッシュラインのデータはプロセッサコア内では利用可能であるが，メインメモリ上の値から更新されメインメモリとキャッシュ上の値が異なる状態である．そのため，キャッシュラインはタグ，データの他に，この 3 状態を管理するための Valid ビット，及び Dirty ビットを持つものとする．

また，メインメモリとキャッシュのコヒーレンシ維持をソフトウェアで行うために，キャッシュの内容をメインメモリに書き戻すライトバック (Writeback)，及びキャッシュを無効化するセルフインバリデート (Self-Invalidate) のキャッシュ操作命令をサポートしていることを想定している．

### 2.2 対象アーキテクチャに対して考慮すべき点

ノンコヒーレントキャッシュアーキテクチャでは，ハードウェアコヒーレンシ維持機構を持たないため，キャッシュコヒーレンシを考慮していない並列プログラムを動作させると以下の 2 つの問題が生じる．

まず，キャッシュラインが更新され最新の内容になったという情報が同一キャッシュラインのコピーを所有している他のコアに伝わらないため，Stale なキャッシュラインの参照 (陳腐化したデータの参照) が発生するという問題がある．このとき，Stale なキャッシュラインはメインメモリ上の値と異なるにもかかわらず Valid となっている．この状態で，コヒーレンシ維持のためにはあるコアでキャッシュラインの更新を行った後，別のコアでそのキャッシュラインの参照がある際，キャッシュラインの一貫性制御をソフトウェアで明示的に行う必要がある．

次に，True Sharing あるいは False Sharing に起因する問題がある．

複数のコアが同一キャッシュラインを更新する際，ハードウェアコヒーレンシ機構の存在下ではキャッシュラインのインバリデーションとライトバックが逐次化されるために，当該キャッシュラインのミス時には全コアによる処理がメインメモリに正しく反映される．一方で，ハードウェアコヒーレンシ機構が存在しない場合，最終的にメインメモリに反映されるキャッシュラインは時系列順で最後にライトバックを行ったコアが持つキャッシュラインとなり，そのコアが更新したデータのみしかメインメモリに反映されないという問題がある．このような状況は，複数コアによりキャッシュライン内の同一部分が更新される場合 (True Sharing) またはキャッシュライン内の異なる部分がで更新される場合 (False Sharing) に発生する．True Sharing または False Sharing が発生する場合，コヒーレンシ維持のためには変数配置を変更し同一キャッシュラインへの更新が同時に起きないようにするか，同一キャッシュラインへの更新を含む処理をソフトウェア側で逐次化する必要がある．

## 3. コンパイラによるキャッシュコヒーレンシ制御手法

本章ではノンコヒーレントキャッシュアーキテクチャに対してコンパイラが行う制御手法を提案する．本章で述べるコヒーレンシ制御手法を OSCAR マルチグレイン自動並列化コンパイラに実装した．

### 3.1 粗粒度並列処理

図 2 に OSCAR コンパイラにおける階層的並列処理の実行イメージを示す．階層的粗粒度タスク並列処理では，まずプログラムは基本ブロック，ループ，関数呼び出しなどのマクロタスク (MT) として分割する．この際ループ及び関数呼び出しの内部に対しては，階層的に MT 生成を行う．次に MT 間のコントロールフローとデータ依存の解析結果から最早実行可能条件解析 [7] を行い，MT 間の粗粒度並列

ⓒ 2014 Information Processing Society of Japan

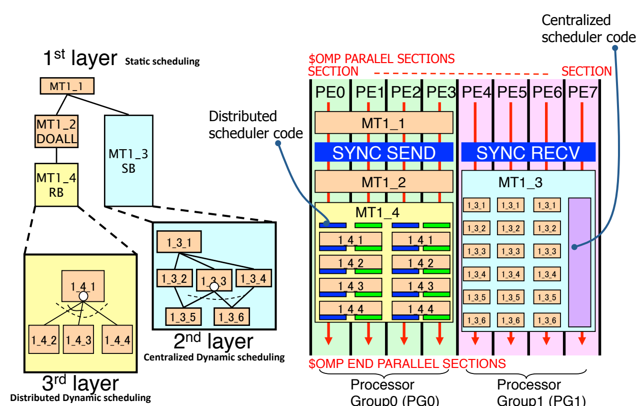


図 **2** 階層的タスク並列処理の実行モデル
**Fig. 2** Data layout transformation

性を表現した結果としてマクロタスクグラフ (MTG) を生成する．その後，各 MTG の持つ並列性に応じ，プロセッサコア (PE) のグルーピング [8] を行い，タスクに対し階層的なプロセッサグループ (PG) を割り当て，スケジューリングを行う．階層的な MTG が PG，PE に割り当てられ，並列実行される際のイメージを図 2 に示す．

### 3.2 並列化に伴うコヒーレンシ制御

本手法では，まずプロセッサグループ割り当てにより複数のプロセッサコアが割り当てられ並列処理されるタスクについて，False Sharing の検出と回避を行う．その後，スケジューリング結果に基づき各プロセッサコアにタスクを割り当てるとともにタスク間の依存関係を保持する同期を挿入して並列コード生成を行う．並列コード生成と同時に，タスク間のデータ依存に基づいてキャッシュ操作の挿入を行う．

### 3.3 陳腐化したデータへの参照の回避

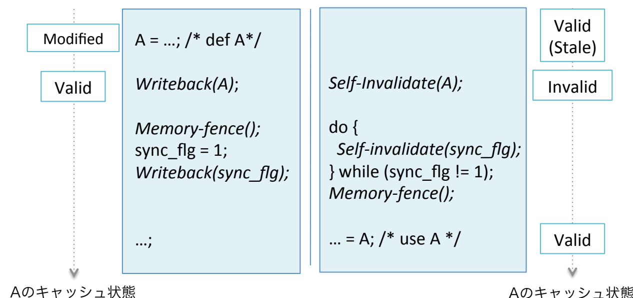


図 **3** キャッシュ操作
**Fig. 3** Cache operation

本手法ではデータ依存解析に基づき，陳腐化したデータへの参照の回避を行う．図 3 のように，あるコアでデータの定義を行った後，他のコアでデータの参照を行うというフロー依存が存在する場合，キャッシュコヒーレンシ操作コードの挿入を行う．具体的には，データの更新側のコアでは更新の後に更新を行ったキャッシュラインをライトバックし，その後メモリアクセスの逐次化を行うメモリフェンス命令の実行を行い，最後に同期を行う．読み出し側のコアでは，キャッシュラインの状態は Stale 状態となっているので，キャッシュラインをセルフインバリデートし，同期及びメモリフェンスの後に読み出しを行う必要がある．出力依存についてもフロー依存と同様にキャッシュ操作命令の挿入を行う．逆依存については陳腐化したデータの参照は発生しないため，キャッシュコヒーレンシ操作コードの挿入は必要なく，同期により変数へのアクセスが逐次化されていることを保証すれば良い．コンパイラによるデータ依存解析が不十分でメモリアクセス範囲が特定できない場合，キャッシュ操作はキャッシュ全域に対して行う必要が生じる．プログラム中に True Sharing が発生する場合，データ依存解析により依存が生じることが特定され，本節の方法を適用してりキャッシュラインへのアクセスを逐次化し，正しい動作を保証することができる．

### 3.4 False Sharing の検出

```
int A[1000][3]; /* 4Byte per element */
for (i = 0; i < n; i++) {
    A[i+1][2] = ...;
}
```

図 **4** False Sharing が発生する可能性のあるループの例
**Fig. 4** An example of loop of possibly false sharing

変数間での False Sharing は 3.5.1 節の方法で解消することを前提とすると，複数のコアから同一キャッシュライン上の異なる位置でアクセスされる配列要素間では False Sharing が存在するため，False Sharing の検出及び回避を行う必要がある．そのため並列実行されるタスクまたはループイタレーションで，配列の定義される範囲の解析をそれぞれに対し行い，False Sharing が発生する配列と，False Sharing が起こる配列次元を求める．

キャッシュラインサイズを 32Byte とした際の具体的な解析例を図 4 に示す．図 4 では配列 `A` は `int` 型 (1 要素 4Byte) の，1 次元目の大きさが 3，2 次元目の大きさが 1000 の配列として宣言されており，その後の for ループで配列要素の定義がされている．

まず，並列化対象ループの内側で定義される配列の範囲を解析した結果は，`A[i+1][2]` である．ループが最大分割数，つまり 1 イタレーションごとに分割され並列実行される場合を考慮すると，ループインデックス `i` が変化して `A[i+1][2]` と `A[(i+1)+1][2]` $(0 \leq i < n)$ が異なるコアで更新される．ここで，キャッシュラインサイズは 32Byte であるのに対し，A の 2 次元目のサイズは 4Byte $\times\ 3 = 12$Byte とキャッシュラインサイズより小さいので，`A[i+1][2]` 及び `A[(i+1)+1][2]` が同一キャッシュライン

ⓒ 2014 Information Processing Society of Japan

に配置されて複数コアによる更新が発生する可能性があることがわかる．その結果，配列 A については 2 次元目で False Sharing するという解析結果が得られる．

### 3.5 False Sharing の回避

False Sharing 解析結果を元に，False Sharing を回避し，同一キャッシュラインへの更新が異なるコアで同時に起きないことを保証する．本節では False Sharing の各回避方法についてそれぞれ述べる．False Sharing が解析できない，あるいは False Sharing の回避が不可能な場合，そのタスクは 1 つのコアで逐次処理される．

#### 3.5.1 変数のアライメント

並列化により複数のプロセッサコアが同一のラインを処理してしまう場合の False Sharing の問題に対しては，変数をキャッシュラインの先頭に配置することで，変数間でキャッシュラインを共有しないようにする．これにより変数間で False Sharing が発生しないことを保証できる．また，本操作により配列や構造体の先頭がキャッシュラインの先頭に配置されることを前提に配列内部の False Sharing の解析が可能となる．

#### 3.5.2 キャッシュラインを考慮したループ分割

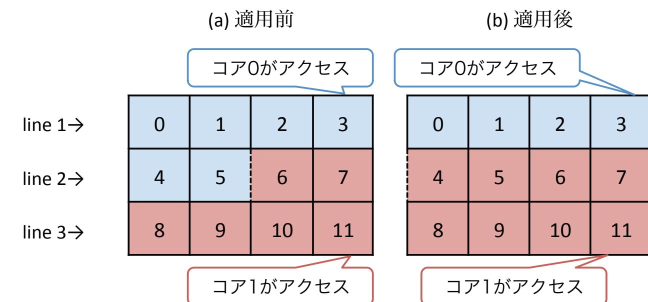


図 5 キャッシュラインに沿ったループ分割
**Fig. 5** Loop division aligned to cacheline

ループイタレーション間に依存がなく独立に実行可能な場合，通常はループは負荷均衡を考慮して均等分割して並列処理を行うが，分割の境界がキャッシュラインの途中に存在する場合 False Sharing が発生する．例えば，図 5(a) のように 12 要素の配列を定義する処理を 6 要素ずつコアに均等分割して並列処理すると，A[6] の先頭で False Sharing が発生する．そこで図 5(b) のように分割をキャッシュライン先頭で行うことにより，False Sharing を回避する．キャッシュラインに沿ったループ分割では，各コアに割り当てられる回転数が均等にならない可能性があり，そのような場合負荷不均衡が生じて速度低下の原因となる．

#### 3.5.3 パディングによるデータレイアウト変換

データレイアウト変換では，False Sharing の検出結果から False Sharing を起こす配列変数について，False Sharing の発生する次元の要素の先頭がキャッシュラインの境界と

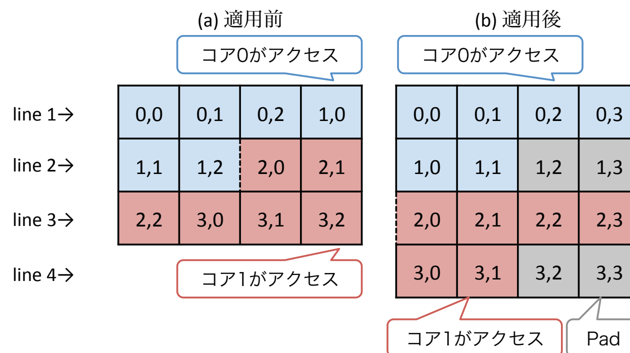


図 6 データレイアウト変換
**Fig. 6** Data layout transformation

一致するように配列要素にダミー要素 (Pad) を挿入して，False Sharing を回避する．配列の左側の次元を上位，右側の次元を下位の次元として，False Sharing が発生する次元の 1 つ下位の次元に Pad を挿入する (Padding)．最下位の次元で False Sharing が発生する場合，配列次元を 1 次元拡張 (Array Expansion) する．図 6 のように，配列 A[4][3] に対する連続アクセスを均等分割する場合，配列 A の 2 次元目で False Sharing が発生するので，1 次元目のサイズがキャッシュラインサイズで割り切れるよう宣言サイズを A[4][4] に変更する．

またデータレイアウト変換を行った時，ポインタ解析結果を元に，変換を行った配列変数を参照するポインタ変数についても，型と参照を変換する．

## 4. 評価

本章では 8 コアマルチコアプロセッサ RP2 上で，科学技術計算アプリケーションに対しコンパイラによるキャッシュコヒーレンシ制御手法を適用し，その有効性を評価する．

### 4.1 評価環境

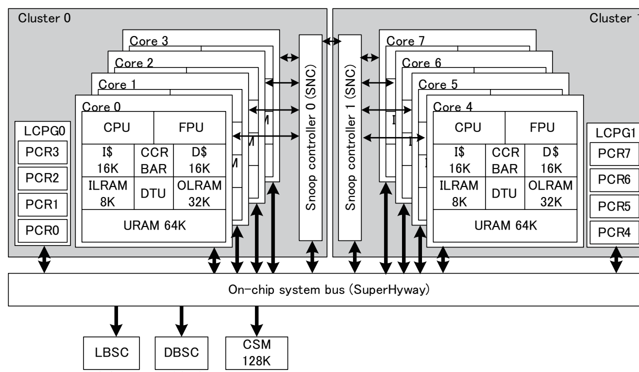


図 7 4 コアまでのコヒーレンシ制御機構を持つ 8 コアマルチコア RP2
**Fig. 7** RP2: 8-core multicore processor with cache coherence up to 4 core

ⓒ 2014 Information Processing Society of Japan

図7に評価環境であるRP2のブロック図を示す．RP2は1チップ上にSH-4Aコアを8コア搭載したホモジニアスマルチコアであり，各プロセッサコアは16KByteのデータキャッシュを持ち，キャッシュラインサイズは32Byteである．ハードウェアによるキャッシュコヒーレンシは4コアで構成されたクラスタ内で保証され，MESIプロトコルが採用されている．クラスタ間では明示的にキャッシュコヒーレンシ制御を行う必要がある．また，クラスタ内のハードウェアキャッシュコヒーレンシ機構は無効化することができる．本アーキテクチャは2.1節で述べたノンコヒーレントキャッシュアーキテクチャの要件を満たしており，キャッシュ操作命令としてはキャッシュラインのライトバック，セルフインバリデートを備える．

### 4.2 評価に使用するアプリケーション

本ソフトウェアコヒーレンシ制御手法を適用するアプリケーションとして，SPEC95ベンチマークより179.art, 183.equake, SPEC2006ベンチマークより456.hmmer, 470.lbm, NAS Parallel Benchmarks v3.0よりSP, CG, LU, MG, BTを用いた．いずれもParallelizable C[9]に準拠した逐次のC言語で記述されたアプリケーションであり，OSCARコンパイラによる自動並列化を適用し，SH Cコンパイラにより実行バイナリを生成した．一部のアプリケーションでは標準のデータセットではメインメモリに載り切らないため，表1に示すとおり縮小したデータセットを使用している．

表 1 アプリケーションとデータサイズ
**Table 1** Applications and data size

| アプリケーション | データサイズ |
|---|---|
| SPEC95 art, equake, SPEC2006 hmmer | ref |
| SPEC2006 lbm | $100 \times 100 \times 15$ |
| NPB SP, CG, LU, MG, BT | CLASS S |

### 4.3 並列処理性能

ハードウェアキャッシュコヒーレンシ制御が有効の場合(SMP)及びハードウェアキャッシュコヒーレンシが無効でコンパイラによるソフトウェアキャッシュコヒーレンシ制御を適用した場合(NCC)において評価を行った際の並列処理性能を図8に示す．図8の横軸はアプリケーション及びコア数を表し，縦軸は各アプリケーションのSMP1コア実行時の実行時間を1とした際の速度向上率を表している．グラフの系列は左がSMP，右がNCCの速度向上率を示している．

図8において，まずSMP4コアとNCC8コアの速度向上率を比較すると，評価に用いた9アプリケーションの平均でSMP4コアが1コア性能の2.80倍に対しNCC8コアが3.66倍とNCC8コアの速度向上率がSMP4コアの速度向上率を上回っている．ここから，ソフトウェアによるコヒーレンシ維持が必要な5コア以上でもコア数に対しスケーラブルに性能向上しているといえる．特にCGにおいてはSMP4コアが1コア性能の3.68倍に対しNCC8コアで最大の5.31倍の速度向上が得られた．また，SMP4コア平均で1コア性能の2.80倍であったのに対し，NCC4コアの性能は平均で1コア性能の2.61倍と7%の性能劣化にとどまりハードウェアキャッシュコヒーレンシが無効の時でも概ね同等の速度向上率が得られているといえる．art, equake, lbmではいずれのコア数でもNCCがSMPの性能を上回っており，特にartでは2コアでSMP1.57倍に対しNCC1.67倍，4コアでSMP2.66倍に対しNCC3.01倍であった．一方SP及びLUでは，4コアのSMPに対するNCCの性能を比較すると，SPでは2.14倍に対し1.60倍，LUでは2.76倍に対し1.18倍の速度向上率であり，NCCのSMPに対する性能低下が顕著に見られる．

図9にソフトウェアキャッシュコヒーレンシ制御を構成する要素ごとの性能に対する影響を示す．図9の横軸はアプリケーション及びコア数を表し，縦軸は各コアのSMPを1とした際の相対性能を表す．グラフのそれぞれの系列は以下のとおりである．

**SMP** 基準となる，MESIハードウェアコヒーレンシ制御が有効でソフトウェアキャッシュコヒーレンシ制御を行わない際の性能．

**Stale data handling** ソフトウェアキャッシュコヒーレンシ制御のうち，Stale Dataへの対処のみ行うが，ハードウェアコヒーレンシは有効化した際の性能．キャッシュ操作による影響を表す．

**False sharing avoidance** ソフトウェアキャッシュコヒーレンシ制御のうち，False Sharingへの対処のみ行うが，ハードウェアコヒーレンシは有効化した際の性能．アライメント，キャッシュラインに沿ったループ分割，及びデータレイアウト変換による影響を表す．

**NCC (hardware coherence)** ソフトウェアキャッシュコヒーレンシ制御を行い，かつハードウェアコヒーレンシ制御を有効化したままの状態の性能．コヒーレンシ制御による影響を表す．

**NCC (software coherence)** ソフトウェアキャッシュコヒーレンシ制御を行い，ハードウェアコヒーレンシ制御を無効化した際の性能．

図9において，art，equake，及びlbmでは，4コアでのNCC (software coherence) とNCC (hardware coherence) を比較するとNCC(hardware coherence) が4%から14%の性能向上がある．ハードウェアコヒーレンシ制御を無効化することによる性能への影響を与える要素として2つの要因が挙げられる．1つめの要因として，コア間でキャッシュラインの授受がある際，ハードウェアコヒーレンシ制

Vol.2014-ARC-213 No.19
Vol.2014-HPC-147 No.19
2014/12/9

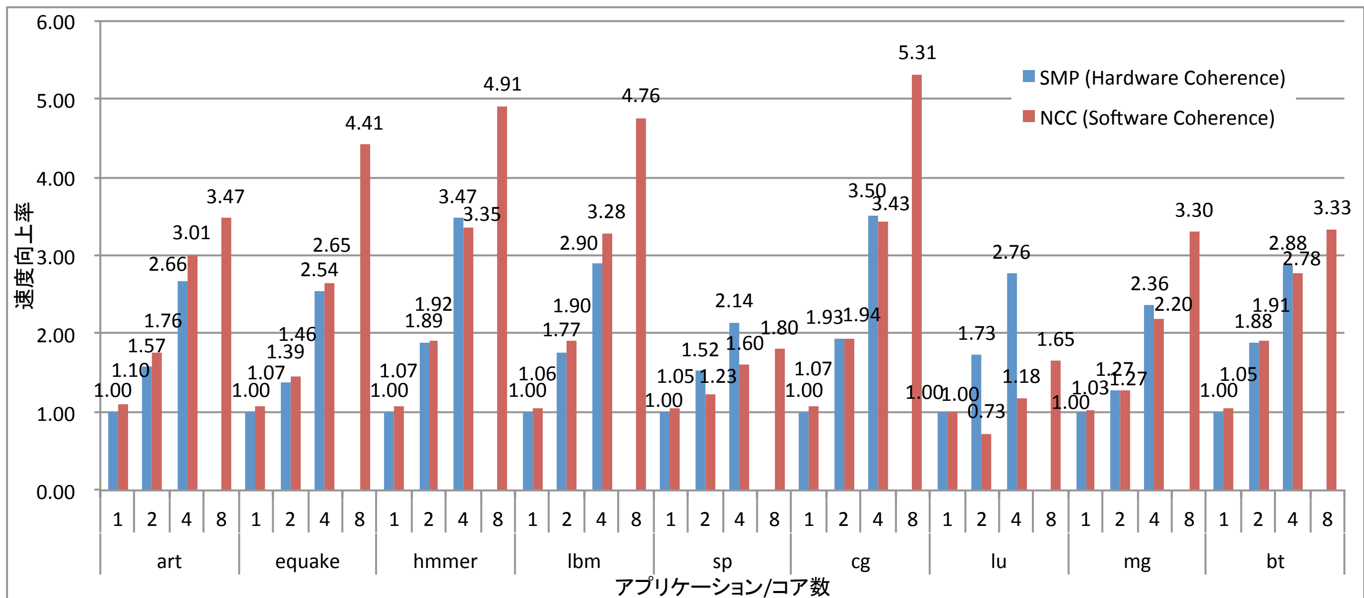


図 8　提案ソフトウェアコヒーレンシ制御の性能

**Fig. 8**　Parallel performance on RP2

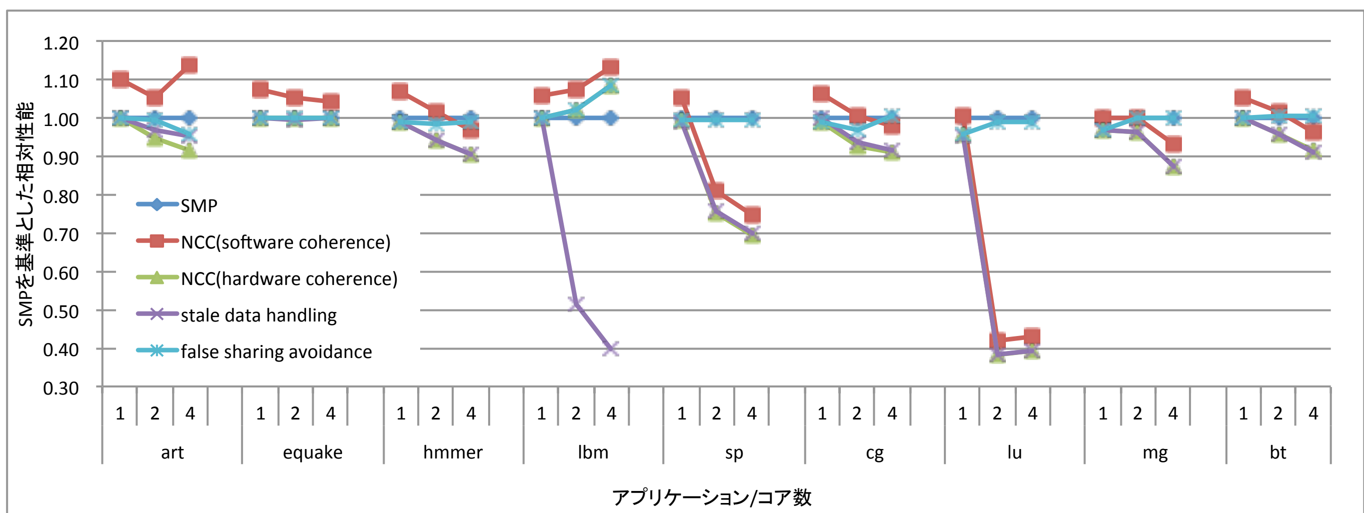


図 9　ソフトウェアキャッシュ制御による性能に対する影響

**Fig. 9**　Performance effects of software cache

御下ではキャッシュ間転送を行うが，ソフトウェアキャッシュコヒーレンシ制御ではメインメモリ経由でのキャッシュライン授受を行い，更にキャッシュ操作命令のオーバーヘッドがある (1)．2 つめの要因として，他のコアのキャッシュに存在しないキャッシュラインのアクセスを行う場合，ハードウェアコヒーレンシ制御下ではそのキャッシュラインが他のコアに存在するか確認を行うバストランザクションが発生するのに対し，ソフトウェアキャッシュコヒーレンシ制御では他のコアのキャッシュラインの状態を確認する必要が無いため，その分性能向上に寄与する (2)．

(1)(2) の要素が性能に影響を与える度合いはアプリケーション内のデータアクセスの特性に応じて変化すると考えられる．art, equake, 及び lbm でハードウェアコヒーレンシ制御無効化により性能向上が得られたのは，(2) のようなコア間でキャッシュラインを共有しないデータアクセスが支配的であるためである．

図 9 における LU と SP の 4 コアでの Stale data handling と SMP を比較すると，Stale data handling は SMP と比較して SP で 30%，LU で 41%の性能低下を起こしており，Stale Data への対処が性能に悪影響を与えていることがわかる．これらのアプリケーションでは並列化階層が外側のループにより繰り返し実行されるという制御構造を持っており，並列化対象の処理の粒度が小さいという特徴がある．NCC では並列化階層において陳腐化したデータへの参照の回避のためキャッシュ全体に対する操作を行っているの

ⓒ 2014 Information Processing Society of Japan

で，外側の繰り返しに存在するデータローカリティを有効活用できなくなり性能低下が起きたと考えられる．外側のループイタレーション間でのメモリアクセス範囲の解析結果をキャッシュ操作に適用し，キャッシュ操作の範囲を限定することで性能改善が見込まれる．

## 5. おわりに

本稿では，自動並列化コンパイラによるソフトウェアキャッシュコヒーレンシ制御手法を OSCAR コンパイラに実装し，ノンコヒーレントキャッシュアーキテクチャ RP2 上で評価を行った．本手法はソフトウェアに対し自動で Stale Data 参照及び False Sharing の回避を行うために，手動での困難なキャッシュコヒーレンシ制御のプログラミングを必要とせずノンコヒーレントキャッシュアーキテクチャ上でハードウェアキャッシュコヒーレンシ制御下と同等の動作を実現可能である．マルチコアプロセッサ RP2 上で評価を行った結果，9 アプリケーションで 4 コアのハードウェアコヒーレンシ制御使用時の性能は平均で 1 コア性能の 2.80 倍であったのに対し，ハードウェアコヒーレンシを使用せず本手法を適用した 4 コア実行時の性能は平均で 1 コア性能の 2.61 倍となり，ほぼ同等の速度向上が得られた．本手法の適用により，ノンコヒーレントキャッシュアーキテクチャに対するソフトウェア開発コスト及びメニーコア開発時のハードウェアコストを削減可能であり，今後のコンピューティング技術の発展に寄与するものと考えられる．

## 参考文献

[1] 坪井芳朗, 太田裕, 山下高廣：東芝の次世代 SoC「Venezia」, ホモジニアス・マルチコアを採用，日経エレクトロニクス，No. 981, pp. 105–118 (2008).
[2] Howard, J. et al.: A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS, *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108 –109 (2010).
[3] Johnson, D., Johnson, M., Kelm, J., Tuohy, W., Lumetta, S. and Patel, S.: Rigel: A 1,024-Core Single-Chip Accelerator Architecture, *Micro, IEEE*, Vol. 31, No. 4, pp. 30 –41 (2011).
[4] Hennessy, J. L. and Patterson, D. A.: *Computer Architecture: A Quantitative Approach 5th edition*, p. 112, Elsevier (2012).
[5] 笠原博徳：最先端の自動並列化コンパイラ技術，情報処理，Vol. 44, No. 4, pp. 384–392 (2003).
[6] Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Ito, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K. and Kasahara, H.: An 8640 MIPS SoC with independent power-off control of 8 CPUs and 8 RAMs by an automatic parallelizing compiler, *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, Vol. 51, pp. 90–92 (2008).
[7] 本多弘樹，岩田雅彦，笠原博徳：Fortran プログラム粗粒度タスク間の並列性検出手法，電子情報通信学会論文誌 (1990).
[8] 小幡元樹，白子準，神長浩気，石坂一久，笠原博徳：マルチグレイン並列処理のための階層的並列性制御手法，情報処理学会論文誌，Vol. 44, No. 4, pp. 1044–1055 (2003).
[9] 間瀬 正啓，木村 啓二，笠原博徳：マルチコアにおける Parallelizable C プログラムの自動並列化，情報処理学会研究報告. 計算機アーキテクチャ研究会報告， Vol. 2009, No. 15, pp. 1–10 (2009).

ⓒ 2014 Information Processing Society of Japan