並列化コンパイラによるソフトウェアコヒーレンシ制御

間瀬正啓[†1] 木村啓二[†1] 笠原博徳[†1]

近年，あらゆる情報機器において処理性能の向上および低消費電力化のため，マルチコアの採用が進んでおり，今後チップ上に集積されるコア数はさらに増え，メニーコア時代が訪れると考えられている．しかしながら，64，128 コア以上のメニーコアプロセッサにおけるコヒーレントキャッシュハードウェアは回路規模的にも消費電力的にも実装コストが大きくなりすぎるため，実用化が困難と考えられている．本稿では，キャッシュコヒーレンシ制御機構を持たない共有メモリ型のマルチプロセッサシステムにおいても，並列化コンパイラによりコヒーレントキャッシュと同等な処理を可能とする，ソフトウェアコヒーレンシ制御手法を提案する．本手法を OSCAR 自動並列化コンパイラに実装し，4 コアまではハードウェアコヒーレント機構を持つが，5 コア以上はノンコヒーレント共有メモリ動作となる，8 コア構成の情報家電用マルチコア RP2 において評価を行ったところ，4 アプリケーションプログラムにおいて 4 コアまででノンコヒーレントキャッシュモードでもコヒーレントキャッシュモードと同等以上の性能が得られ，さらに 8 コア使用時にも 1 コア使用時と比較して平均 4.88 倍の速度向上が自動で得られた．

Parallelizing Compiler Directed Software Coherence

Masayoshi Mase,[†1] Keiji Kimura[†1]
and Hironori Kasahara[†1]

As multicore processor becomes widely used in various computer systems, the number of cores integrated in a chip is increasing for improved performance and reduced power consumption toward manycore era. However, cache coherency hardware in manycore processors which integrates over 64 or 128 cores is hard to implement both for circuit area and energy cost. This paper proposes a parallelizing compiler directed software coherence for shared memory multiprocessor systems without hardware cache coherence mechanism that enables as same parallelization as on hardware coherent cache. We implemented the proposed method in OSCAR automatic parallelizing compiler and evaluated on RP2, a multicore for consumer electronics integrating 8 cores, that can support hardware coherent cache mode under 4 cores and non-coherent shared memory multiprocessor mode over 4 cores. The evaluation results shows that automatic parallelization with the proposed software coherence mechanism achieves as much or even better performance than hardware coherence under 4 cores. Also, the software coherence gives us 4.88x speedup for 8 cores in average on 4 application programs against sequential execution.

1. はじめに

組込み機器から PC，スーパコンピュータに至るあらゆる情報機器において 1 チップ上に複数のプロセッサコアを集積したマルチコアプロセッサの普及が進んでおり，チップ内に集積するコア数の増加による高性能・低消費電力化が期待されている．現在主流の 4 から 8 コア程度のマルチコアでは主記憶共有型のマルチプロセッサシステム (SMP) が一般的であるが、そのキャッシュコヒーレンシ制御機構のハードウェアは，プロセッサコア数の増加に伴い，その実装が回路規模的にも消費電力的にも困難となることが知られている[1)]．そのため，今後コア数が 32 コアから 64 コア以上のメニーコアとなっていくと，共有メモリ空間とコア毎のコヒーレンシ制御機能を持たないキャッシュ上でも，ソフトウェアによりコヒーレンシ制御を行うことができれば，コスト及び電力消費を抑えつつ効率的な並列処理を実現できると期待されている．特にハードウェアコストや消費電力およびリアルタイム制約に厳しい組込み系マルチコアでは，4 コアの富士通 FR1000[2)]，8 コアの東芝 Venezia[3)] 等において，すでにノンコヒーレントキャッシュアーキテクチャが採用されている．また，ルネサステクノロジ／日立製作所／早稲田大学の RP2[4)] は 8 コア集積しているが，ハードウェアではチップ価格を抑えるためにコヒーレンシ機構は 4 コアまで対応とし，5 コア以上のコア数ではノンコヒーレントキャッシュとして利用する設計となっている．高性能計算分野においても，イリノイ大の Rigel[5)]，Intel の Single-Chip Cloud Computer (SCC)[6)] は，それぞれのコアがキャッシュを持つが，チップ全体でのコヒーレンシ制御はソフトウェアにより行うことが想定されている．

しかしながら，このようなアーキテクチャにおける単一のアプリケーションプログラムの並列処理による高速化では，ソフトウェアによるコヒーレンシ制御のプログラミングが必須であるが，ソフトウェア開発は長期間に伴う非常に複雑な作業となり，ソフトウェア開発生産性が問題となる．そのため，ユーザからコヒーレンシ制御の複雑さを隠蔽し，簡単にチッ

†1 早稲田大学 基幹理工学部 情報理工学科
Dept. of Computer Science and Engineering, Waseda University

プを使うことを可能とするソフトウェア開発環境を提供する必要がある．その際，単一のアプリケーションプログラムの並列処理による高速化を実現するためには，コヒーレンシ操作のオーバヘッドを最小限に抑える効率的な実装が必須となる．

本稿では，ノンコヒーレントキャッシュマルチコア及びメニーコアプロセッサにおけるアプリケーションプログラムの自動並列化をターゲットとして，コンパイラがソフトウェアでコヒーレンシ制御を行う手法を提案する．本手法の特徴は，コンパイラによる高度な自動並列化とキャッシュ最適化[7)–9)] を適用した上で，コア間で共有するデータに対する並列処理タスク単位の粗粒度なキャッシュ操作と，ノンキャッシャブル変数指定を用いた効率的なコア間通信をコンパイラが自動生成することにある．また，各コアでは自コアのキャッシュに対してのみ操作を行うため，コヒーレンシ制御のために余分なコア間トラフィックが発生しない．これにより，プログラマの手を煩わせることなくソフトウェアによるコヒーレンシ制御を実現し，多くのコア数まで性能がスケールアップするメニーコアハードウェアを低コストで容易に開発できるようになる．

提案するソフトウェアコヒーレンシ制御手法を，OSCAR 自動並列化コンパイラ[9)] に実装し，4 コアまではハードウェアコヒーレント機構を持つが，5 コア以上はノンコヒーレント共有メモリ動作となる，8 コア集積の情報家電用マルチコア RP2 において評価を行った．その結果，コンパイラによるソフトウェアコヒーレンシ制御を利用することで，ハードウェアによるコヒーレンシ制御をサポートする 4 コアまででコヒーレントキャッシュと同等に性能向上が得られ，さらにハードウェアではコヒーレンシ制御を行えない 5 コアから 8 コアまでスケーラブルな性能向上が得られることが確認された．

本稿の構成は以下の通りである．まず，2 章で提案するソフトウェアコヒーレンシ制御の概要を述べる．次に，3 章で提案する並列化コンパイラによるソフトウェアコヒーレンシ制御手法について述べ，4 章ではコンパイラの実装と実行時環境におけるインターフェースとなる API について述べる．そして，5 章でノンコヒーレントキャッシュアーキテクチャ上での並列処理性能について述べる．最後に 6 章でまとめを述べる．

2. ソフトウェアコヒーレンシ制御

コヒーレンシとは，あるメモリアドレスに格納されるデータをある時刻において全てのプロセッサコアから同一の値としてアクセスできることである．並列処理のプログラムを正常に実行するためには，ソフトウェアやハードウェアを用いてコヒーレンシを維持するための枠組みが必要となる．

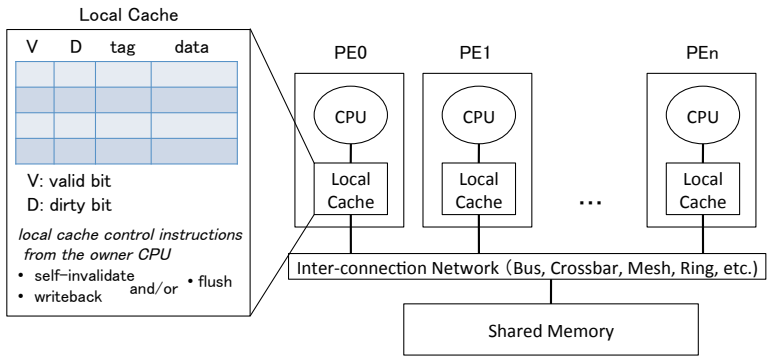

図 1 ノンコヒーレントキャッシュアーキテクチャ
Fig. 1 non-coherent cache architecture.

2.1 ノンコヒーレントキャッシュアーキテクチャ

対象とするノンコヒーレントキャッシュアーキテクチャでは，図 1 に示すように，複数のプロセッサコアが主記憶メモリを共有しており，それぞれのプロセッサコアは独立にキャッシュを持つが，コヒーレンシ制御用のハードウェア機構は持たない．各コアのローカルキャッシュのハードウェアは，一般的なマイクロプロセッサで利用されているような各キャッシュライン毎に valid bit と dirty bit を持つものを想定しており，各プロセッサコアにおいて，そのキャッシュラインが有効かどうか (valid bit)，キャッシュライン上のデータがメモリから読み込まれた後にキャッシュ上で更新されているかどうか (dirty bit) をハードウェアで管理する．すなわち，それぞれのプロセッサコアが同一のメモリアドレスに格納される値を個別にキャッシュすることがあるため，ソフトウェアで整合性を取る必要がある．

あるプロセッサコアがある変数の値をキャッシュしている場合に，他のプロセッサコアがその変数を更新するとき，更新前からキャッシュしているプロセッサコアにおいて次にその変数を参照する前に自プロセッサコアのキャッシュラインの内容をインバリデートしないと，他プロセッサコアが更新する前の古いデータを誤って参照してしまう．また，他のプロセッサコアで更新された変数を参照する前には，更新したプロセッサコアはキャッシュ上で更新したデータを主記憶メモリにライトバックして，明示的にメモリを更新することによって，他のプロセッサで参照可能となることを保証する．

その際に，キャッシュのハードウェアはプログラム中の個々の変数ごとではなく，キャッシュライン単位でメモリとのデータ一貫性の維持を行うため，プログラム上では異なるメモ

リ領域として宣言されていても同一のキャッシュラインに対応するメモリアドレスに割り付けられる変数がある．そのためにキャッシュラインが意図せず複数のプロセッサコアで共有されてしまうフォルスシェアリングについても，コヒーレンシ制御で対処する必要がある．特に，ソフトウェアによるコヒーレンシ制御では，フォルスシェアリングへの対処の実行時オーバヘッドが大きくなるため，重要な課題となる．

2.2 ソフトウェアによるキャッシュコヒーレントプロトコル

本稿では，フォルスシェアリングの回避をコンパイラで実現し，特殊なハードウェアサポートを必要とせず既存のプロセッサコアを多数集積するのみで実現可能なコンパイラによるソフトウェアコヒーレンシ制御手法を提案する．提案するソフトウェアによるキャッシュコヒーレントプロトコルを図 2 に示す．提案するコヒーレンシ制御のプロトコルは Modified，Valid，Invalid，Stale の 4 状態の状態遷移で実現される．各状態は以下のような意味である．

- Modified: そのコアでキャッシュ上のデータが更新されて書き戻されていない状態
- Valid: そのコアでキャッシュ上のデータが有効でありメモリ上の値と一致している状態
- Stale: キャッシュ上のデータに対応するメモリ領域が他のプロセッサコアで更新されて古いデータとなってしまっている状態
- Invalid:キャッシュが無効化された状態

ここで，状態遷移図中の括弧はキャッシュラインの dirty bit の値を表しており，Valid と Stale はハードウェアの状態としては同じ状態であり，コンパイラにおける解析時の状態管理における扱いで区別している．図中の状態遷移において，一重線はロード (load)，ストア (store) 等のプロセッサコアによるメモリアクセスを表す．二重線は自コアのキャッシュラインのライトバック (writeback) やセルフインバリデート (self-invalidate) 等のキャッシュ操作を表す．提案手法では，これらのキャッシュ操作指示をコンパイラが自動生成し，全てのコアは他のコアの動作に影響されることなくキャッシュ操作を行いながら並列処理を行うことが特徴である．コンパイラは対象のメモリアドレスのデータについて，複数のコアのキャッシュで同時に Modified の状態とならないように，また Stale 状態のときに読み込みを行わないように，プログラムの最適化およびキャッシュ操作の挿入を行う．

以下，本稿では上記のソフトウェアコヒーレンシ制御を行うための，コンパイラの実装について述べる．

3. コンパイラによるソフトウェアコヒーレンシ制御手法

本章では提案するコンパイラによるコヒーレンシ制御手法について述べる．

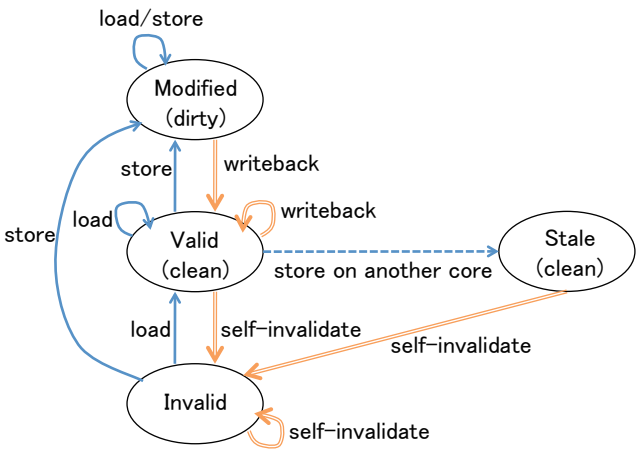

図 2 ソフトウェアによるキャッシュコヒーレントプロトコル
Fig. 2 Software based cache coherent protocol.

3.1 階層的粗粒度タスク並列処理

本稿では，階層的な並列性を汎用的に表現可能な階層的粗粒度タスク並列処理を対象にソフトウェアコヒーレンシ制御を適用する．

階層的粗粒度タスク並列処理では，プログラムは基本ブロック (BB)，ループ等の繰り返しブロック (RB)，関数やサブルーチン呼び出し等のサブルーチンブロック (SB) 等のマクロタスクとして分割され，マクロタスク間のコントロールフローとデータ依存より並列性を抽出したマクロタスクグラフとして表現される[10]．繰り返しブロック (RB) とサブルーチンブロック (SB) はその内部についても階層的にマクロタスクを生成していく．また，ループ繰り返し間のループレベル並列性は，ループ分割により粗粒度タスク並列性に変換され，粗粒度タスク並列処理の枠組みで扱われる．

階層的粗粒度タスク並列処理のイメージを図 3 に示す．プログラムは階層的なマクロタスクグラフとして表現され，各マクロタスクの持つ並列性を元にコンパイラがプロセッサコアのグルーピングを行う．階層的にグルーピングしたプロセッサグループへスケジューリングを行うことで，階層的な並列処理を実現する．

3.2 並列処理階層の決定

コンパイラではプログラムの各部分の持つ並列性に応じて，階層的なマクロタスクグラフ

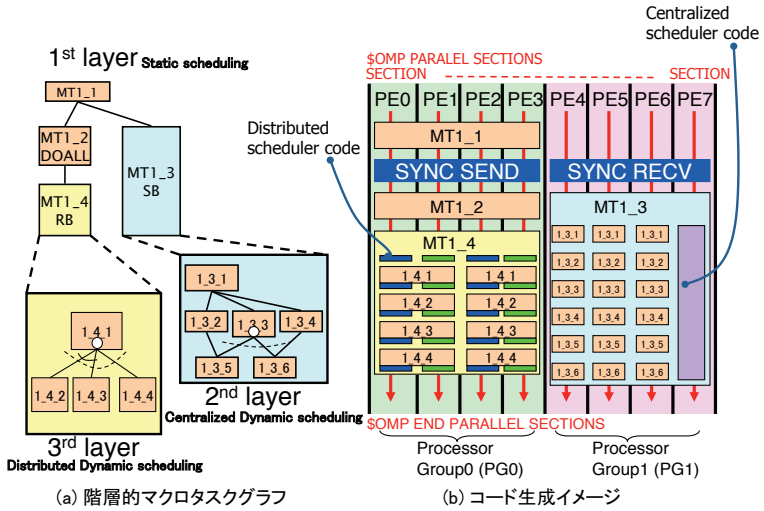

図 3 階層的粗粒度タスク並列処理のイメージ
Fig. 3 Hierarchical coarse grain task parallel processing image.

に対して，プロセッサグループを割り当てていく[11]．この過程で複数のプロセッサグループが割り当てられた階層について，粗粒度タスク並列処理が行われる．そこで複数のプロセッサグループが割り当てられた階層において，並列処理される可能性があるタスクのメモリアクセス範囲から，フォルスシェアリングが発生する可能性があるかどうかを解析する．

3.3 変数配置のアラインメント

まず，フォルスシェアリングの回避のために，プロセッサコア間で共有する変数の先頭アドレスをそれぞれキャッシュライン境界へアラインメントして配置する．malloc 等で動的確保されるヒープ領域についても，先頭アドレスがキャッシュラインの先頭にアラインメントされるものとする．これにより異なる変数への参照によるフォルスシェアリングを回避するとともに，配列や構造体等の内部要素に対するアクセスによるフォルスシェアリングの検出と回避を行いやすくする．

3.4 フォルスシェアリングの解析

本節では，各変数の先頭がキャッシュラインの境界にアラインメントされていることを前提とした場合に，その変数の内部のメモリ領域について，フォルスシェアリングが発生する可能性があるかどうかを検出する手法を述べる．

この解析は並列処理のためのタスク分割後のタスクグラフにおける各タスクのメモリアクセス範囲を解析する．並列処理のためのタスク分割が行われるタスク，すなわち並列ループやデータローカライゼーションの対象と判定されたループについては，タスク分割後のアクセス範囲を考慮してフォルスシェアリングの検出を行う．ここでは，最大分割数でタスクを分割した際，すなわちループ 1 イタレーションが部分タスクとなるような場合を考えて，フォルスシェアリングが発生する可能性を考える．各タスクのメモリアクセス範囲に重なりがなく，それらのアクセス範囲の境界をまたぐキャッシュラインが存在しなければ，フォルスシェアリングは発生しないことを保証できる．

図 4(a) のタスクグラフは，並列性抽出とデータローカリティ最適化[12] が適用されるため，最大分割数で分割されると想定すると図 4(b) のようにタスク分割が行われる．ここで，プログラム上のデータ依存だけを考慮して並列処理される可能性があるタスク，例えば doall1_1 と doall1_2 や doall1_2 と doall1_3 のメモリアクセス範囲を解析すると，それぞれのタスクで同一キャッシュラインを複数プロセッサコアで当時に更新する可能性があり，フォルスシェアリングによる出力依存 (Write after Write) があることが分かる．また，loop3_1 と doall2_2 においては，doall2_2 で参照するキャッシュラインを loop3_1 で更新するため，フォルスシェアリングによる逆依存 (Write after Read) があることが分かる．さらに，loop3_2 と doall4_1 においては，doall4_1 で参照するキャッシュラインを loop3_2 で更新するため，フォルスシェアリングによるフロー依存 (Read after Write) があることが分かる．

このように，フォルスシェアリングが発生する可能性があるタスク群と，対象の変数を解析し，フォルスシェアリングによるデータ依存を解析する．コンパイラのこの後のフェーズでは，フォルスシェアリングの回避のための最適化を行い，もし回避できなかった場合は解析されたフォルスシェアリングによるデータ依存情報に基づいて逐次的に処理される．

3.5 フォルスシェアリングの回避

本手法では節で検出したフォルスシェアリングによるデータ依存に対して，配列データの形状拡張，ループの分割位置の調整，キャッシュを無効化したメモリ領域上のバッファの利用，CPU コアプライベートデータの利用と共有領域への書き戻し逐次化を行うことで，フォルスシェアリングを回避する．

3.5.1 データレイアウト変換

データレイアウト変換は，ある変数の変数宣言を変更することで，データレイアウトを変更する操作である．配列のパディングおよびエキスパンション，構造体のフィールドへ

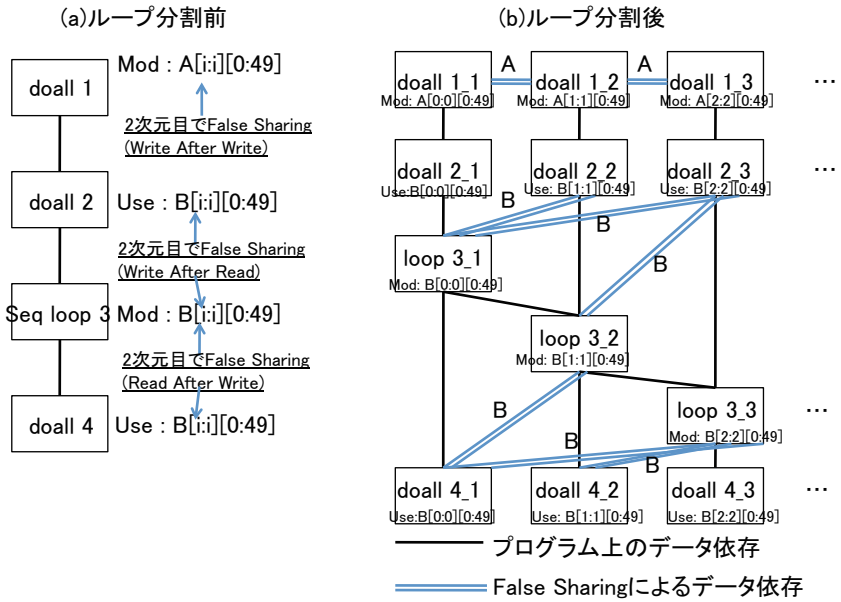

図 4 分割後の部分ループに関するフォルスシェアリングの検出
Fig. 4 False sharing detection on decomposed partial loops.

のパディングにより行われる．パディングの例を図 5 に示す．プログラムの変数宣言の int a[6][6] では外側の i ループを均等分割して並列処理を行うとフォルスシェアリングが発生してしまう．そこで，配列の宣言サイズを変更し，int a[6][8] とすることで，ループ分割後のタスクのメモリアクセス範囲の境界とキャッシュライン境界が整合し，フォルスシェアリングを回避することができる．

ある変数に関する，データレイアウト変換を行う際には，その対象の変数とエイリアスする可能性のある全ての変数に対してデータレイアウト変換を適用する必要がある．そのような変数をポインタ解析結果より解析し，まとめてデータレイアウト変換を適用する．また，それらの変数を指すポインタの型についても，あわせて変更を行う．

3.5.2 キャッシュライン境界に合わせたループ分割

対象の変数の先頭がキャッシュラインの先頭にアラインメントされている場合は，ループ分割時のイタレーション数を適切に設定することで，分割後の部分タスクによるデータアクセス範囲の境界がキャッシュライン境界と一致し，フォルスシェアリングが発生しないことを保証できる．

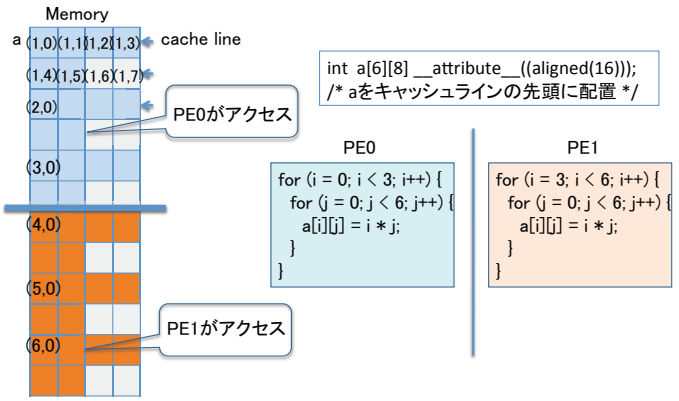

図 5 配列のパディングによるフォルスシェアリングの回避
Fig. 5 False sharing avoidance by array padding.

図 6 における配列 a については，PE0 に 4 イタレーション，PE1 に 2 イタレーション割り当てられるようにループ分割を適用することでフォルスシェアリングを回避している．

3.5.3 ノンキャッシャブルバッファを用いた通信の挿入

ループ中に複数の変数への書き込みがあり，それぞれのアクセス時のオフセットが異なるため，ループ分割のみではフォルスシェアリングを回避できない．そのような場合には，境界領域については一時的にノンキャッシャブルなバッファへ値を格納し，そのバッファを用いて通信を行いながら並列処理を行う．

図 6 の配列 b の例では，配列 a へのアクセスパターンに合わせてループ分割を適用した場合に，配列 b については分割後のメモリアクセス範囲とキャッシュラインの境界が整合しない．そのため，境界領域については一度ノンキャッシャブルバッファに書き込み，そのバッファを用いて値を通信することで，フォルスシェアリングを回避している．

3.5.4 変数のローカル化と書き戻しの逐次化

各プロセッサコアで書き込みを行う変数におけるアクセスパターンがストライドアクセス等で各プロセッサコアがアクセスする範囲に重なりがある場合は，変数のローカル化と書き戻しの逐次化を行う．変数が関数の引数の指し先の領域等で先頭がキャッシュラインにア

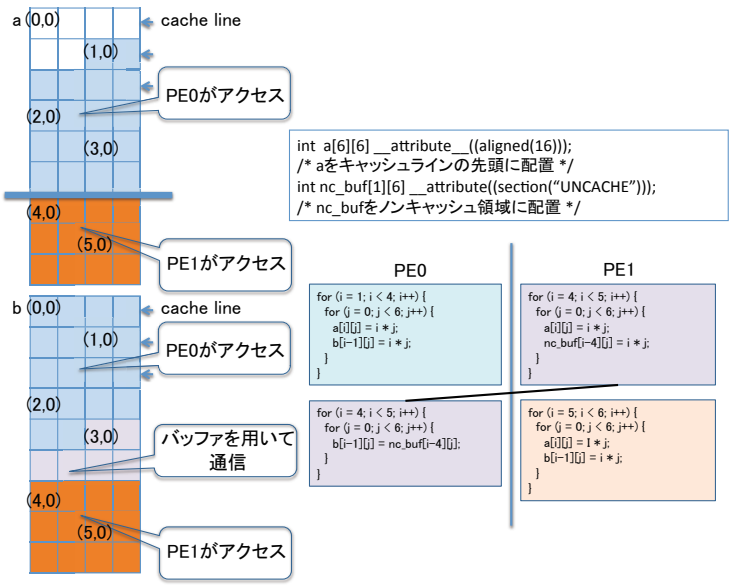

図 6 キャッシュライン境界に整合したループ分割とノンキャッシャブルバッファを用いた通信の生成
Fig. 6 Loop division aligned to cache line boundary and communication generation using noncacheable buffer.

ラインメントされているかどうか不明な場合にも，このローカル化と書き戻しの逐次化を行う．

3.6 キャッシュのセルフインバリデートおよびライトバック操作の挿入

本手法では，コンパイラのデータ利用状況の解析結果より，マクロタスクの実行開始前に，他のプロセッサコアで更新されて Stale 状態となっている変数について，自コアのキャッシュをセルフインバリデートすることで，それ以降で不正に参照されないようにする．また，マクロタスクの実行終了後に他コアで実行する後続マクロタスクで参照する変数がキャッシュ上に存在する場合は，そのデータを共有メモリ上にライトバックすることにより，他コアがその変数を参照できるようにする．粗粒度タスクスケジューリング結果を元にプロセッサグループをまたぐデータ依存についてコンパイラが同期コードの生成を行う際に，キャッシュ操作指示をあわせて挿入する．

マクロタスクグラフ上に存在する依存エッジは，フロー依存，出力依存，逆依存の 3 種類となる．このうち，フロー依存 (Read after Write) および出力依存 (Write after Write) の箇所について図 7 のようにキャッシュ操作指示を挿入する．ただし，逆依存 (Write after Read) については同期は必要であるが，データに対するキャッシュ操作指示は挿入しなくてよい．

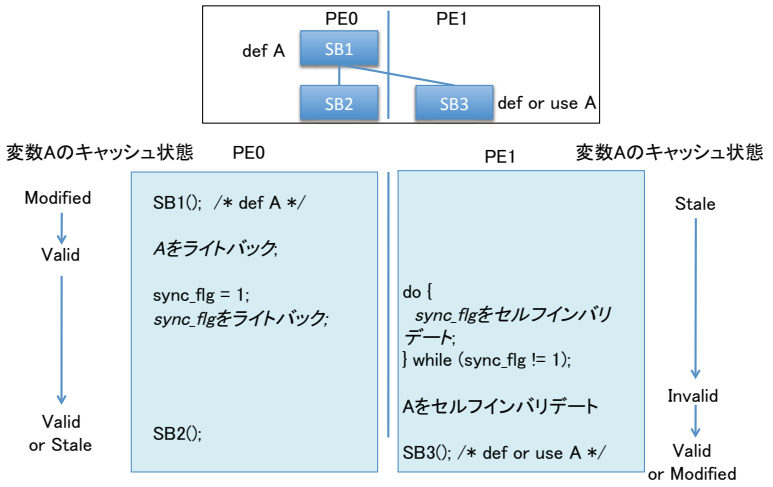

図 7 キャッシュのセルフインバリデート及びライトバック操作指示の挿入
Fig. 7 Inserting cache self-invalidate and writeback operations.

4. コンパイラの実装と実行時環境および API

本稿で提案するソフトウェアコヒーレンシ制御手法は，対象プログラムのソースコードレベルの変換で実現することができる．本手法を実装した OSCAR コンパイラでは逐次プログラムを入力とし，プログラムのデータ依存とコントロールフローの解析結果より自動並列化を行い，OSCAR API[13] で記述された並列プログラムを自動生成する．ソフトウェアコヒーレンシ制御のためのキャッシュ操作や変数配置については，OSCAR API の仕様に新たな拡張を行うことで，変換されたプログラムのコード生成を行う．

4.1 ノンコヒーレントキャッシュ向け OSCAR API 指示文

OSCAR API は情報家電用の並列処理 API として開発されており，OpenMP のサブセットである，parallel sections，flush，critical 指示文による並列処理向けの指示文に加えて，

新たに定義したデータのメモリ配置，データ転送，電力制御，グループバリア同期およびタイマ指示文から構成される．OSCAR API の処理系においては，各スレッドの処理は各コアにバインドされていることを前提に処理を行うことを期待している．

本稿で提案するコンパイラによるソフトウェアコヒーレンシ制御を実現するにあたり，以下の 5 つの指示文を新たに追加した．

- noncacheable: 変数をノンキャッシャブルにする
- aligncache: 変数の先頭をキャッシュライン境界にアラインメントする
- cache_writeback: 自コアのキャッシュ上のダーティラインの書き戻し
- cache_selfinvalidate: 自コアのキャッシュラインの無効化
- complete_memop: メモリ操作の終了

4.2 コンパイラの実装

本稿では，コヒーレンシ制御が必要なキャッシュ操作が必要な箇所において，まずはキャッシュ全体を対象にキャッシュ操作を行うように実装し，評価を行った．また，セルフインバリデートあるいはライトバックが必要な箇所について，全てキャッシュフラッシュ，すなわち対象のキャッシュラインをライトバックした後にセルフインバリデートする操作を行うものとして実装している．

各タスクにおけるメモリアクセス範囲の解析情報から，対象のメモリ領域を指定して，ライトバックやセルフインバリデートのキャッシュ操作を個別に行うことで，より効率的な実装が可能と考えられる．つまり，本稿での評価はソフトウェアコヒーレンシ制御のベースラインの評価となり，選択的なキャッシュ操作の実装と評価は今後の課題となる．

5. 性能評価

本稿で提案するソフトウェアコヒーレンシ制御手法を OSCAR 自動並列化コンパイラに実装し，情報家電用マルチコア RP2 を用いてその有効性を評価した．

5.1 情報家電用マルチコア RP2

情報家電用マルチコア RP2 のブロック図を図 8 に示す．RP2 は SH-4A コアを 8 コア集積したマルチコアであるが，内部はクラスタ構造となっており，コヒーレンシ制御を行うハードウェアを持つ 4 コアの SMP がクラスタを構成しており，クラスタ間ではハードウェアによるコヒーレンシ制御を行わない．すなわち，5 コア以上で使用する場合はソフトウェアによりコヒーレンシを維持する必要がある．

RP2 はキャッシュを明示的に操作する命令を持っており，自コアの対象のキャッシュラインに対するライトバック (OCBWB 命令)，インバリデート (OCBI 命令)，ライトバックの後にインバリデートを行うキャッシュフラッシュ(OCBP 命令) が利用可能である．コンパイラによるキャッシュ操作指示により，これらのキャッシュ操作命令が実行されることになる．なお，RP2 のデータキャッシュは 16KB であり，ラインサイズは 32Byte である．

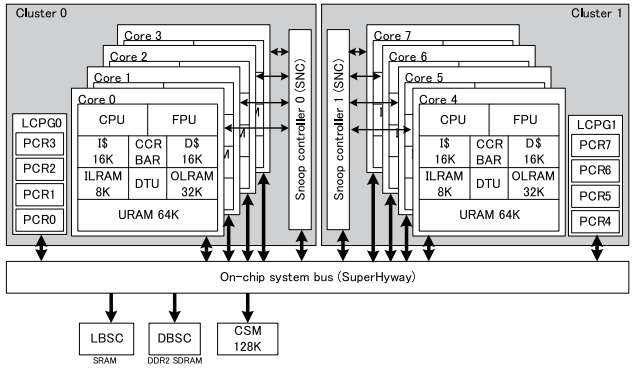

図 8 情報家電用マルチコア RP2 のブロック図
Fig. 8 Block diagram of a multicore for consumer electronics RP2.

また，RP2 は全コアで共有するオンチップ CSM を持っており，これを通信用のノンキャッシャブルバッファとして用いた．

5.2 評価条件

逐次 C プログラムを OSCAR コンパイラで自動並列化し，OSCAR API[13] を用いた並列コードを自動生成する．その並列プログラムを，API 解釈機能を持った SH C コンパイラでコンパイルすることで，実行形式コードを生成する．

比較評価では，4 コアまでのハードウェアによるコヒーレンシ制御と，8 コアまでのコンパイラによるソフトウェアコヒーレンシ制御の処理速度の比較を行う．ソフトウェアコヒーレンシ制御の評価時にはハードウェアによるコヒーレンシ制御機能は無効となるように設定を行った．また，同期やリダクションループの総和計算におけるコア間の通信については，いずれの評価もオンチップ CSM 上のノンキャッシャブル領域を利用するものとする．

対象のアプリケーションとして，マルチメディアアプリケーションから，AAC エンコーダ，MPEG2 エンコーダと，科学技術計算から SPEC2000 より art と equake を用いて評価を行った．今回対象とするプログラムは Parallelizable C で記述され，コンパイラによる自動並列化が適用可能となっている[14]．

5.3 並列処理性能

ハードウェアによるコヒーレンシ制御を有効にした場合の SMP 構成における並列処理性能および，ハードウェアによるコヒーレンシ制御を無効にしソフトウェアによるコヒーレンシ制御を適用した場合のノンコヒーレントキャッシュ構成における並列処理性能を図 9 に示す．図中の横軸が評価アプリケーションと使用 PE 数，縦軸がコヒーレントキャッシュ (SMP) モードの逐次実行時の性能に対する速度向上率となっている．左側のバーが SMP モードの性能，右側のバーがコヒーレンシをソフトウェアで保証するノンコヒーレントキャッシュ(NCC) モードの性能を示している．

まず，SMP モードと NCC モードの 1 コア使用時の性能を比べると，NCC モードでは art において最大 7%ほど性能が向上するが，これはコヒーレンシ制御ハードウェアのオーバーヘッドに起因するものである．次に，4 コア使用時の性能を比較すると，5 本のアプリケーションプログラムにおいて，MPEG2 エンコーダにおいて SMP と比較して最大 5%の性能劣化が見られるが，概ね同等の性能が得られ，art では 13%の性能向上が得られた．

NCC モードでは 8 コア使用時にもそのまま性能向上が見られ，それぞれ SMP モードの逐次実行時と比較して，AAC エンコーダで 5.78 倍，MPEG2 エンコーダで 5.74 倍，art で 3.47 倍，equake で 4.41 倍の性能向上が得られ，平均 4.88 倍の速度向上が得られた．

本稿の評価では，ソフトウェアコヒーレンシ制御におけるキャッシュ操作を非常に保守的に実装したにも関わらず，ハードウェアによるコヒーレンシ制御と遜色の無い，高い性能を得ることができた．

5.4 ソフトウェアコヒーレンシ制御による影響

ソフトウェアコヒーレンシ制御とハードウェアによるコヒーレンシ制御の性能を比較した際の，性能に影響を与える要因について評価を行った．ソフトウェアコヒーレンシ制御を行うことによる性能変化は，(1) ハードウェアコヒーレンシ制御機構の無効化，(2) コンパイラによるフォルスシェアリング回避，および (3) 明示的なキャッシュ操作 (キャッシュフラッシュ) による影響が考えられる．上記の要因を解析するために，ハードウェアの設定とコンパイラオプションを変更した評価を行った．図 10 に評価結果を示す．図中の横軸がアプリケーションおよび使用コア数，縦軸がソフトウェアコヒーレンシ制御時を基準とした，各オプションの性能である．折れ線のそれぞれの系統は以下の通りである．

- NCC (software coherence)：基準となるソフトウェアコヒーレンシ制御適用時の性能．
- NCC (hardware coherence enabled): ソフトウェアコヒーレンシ制御を行うプログラムをハードウェアコヒーレンシを有効にした状態で実行した際の性能，すなわち (1) の要因を示す．
- false sharing avoidance (hardware coherence): ソフトウェアコヒーレンシ制御の構成要素のうちコンパイラによるフォルスシェアリングの回避のみを適用し，ハードウェアコヒーレンシ制御により実行した場合の性能，すなわち (2) の要因を示す．
- stale data handling (hardware coherence): ソフトウェアコヒーレンシ制御のもう一つの構成要素であり，コンパイラによる明示的なキャッシュ操作指示の挿入のみを行った場合の性能，すなわち (3) の要因を示す．
- SMP (hardware coherence): ソフトウェアコヒーレンシ制御のためのコンパイラの機能を用いずに，コヒーレントキャッシュ向けにコンパイルした際の性能．

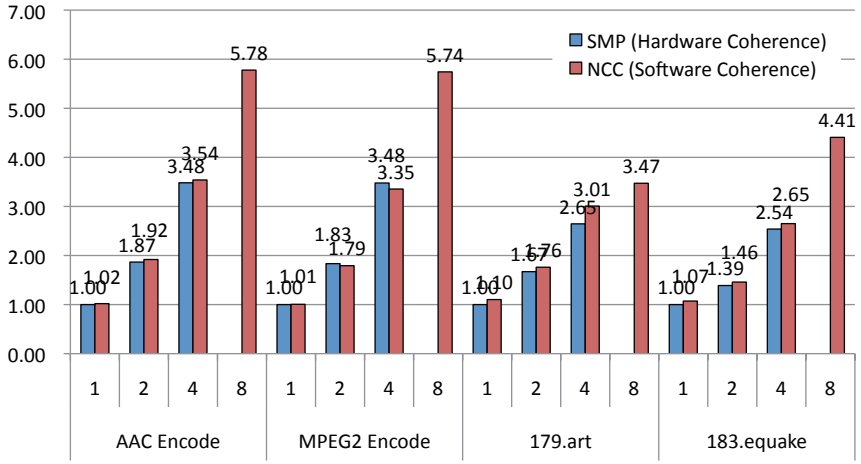

図 9 RP2 における並列処理性能
Fig. 9 Parallel processing performance on RP2.

評価結果を見ると，いずれのアプリケーションについても，NCC (software coherence) が NCC (hardware coherence enabled) よりも高い性能となっており，ハードウェアコヒーレンシ制御を無効化したことによる性能向上が得られていることが分かる．特に art，equake ではハードウェアによるコヒーレンシ制御を無効化した場合の性能向上が大きく，結果的にソフトウェアコヒーレンシ制御の性能がハードウェアコヒーレンシ制御の性能を art で 4 コア使用時に 13.0%，equake で 4 コア使用時 4.2%上回っている．

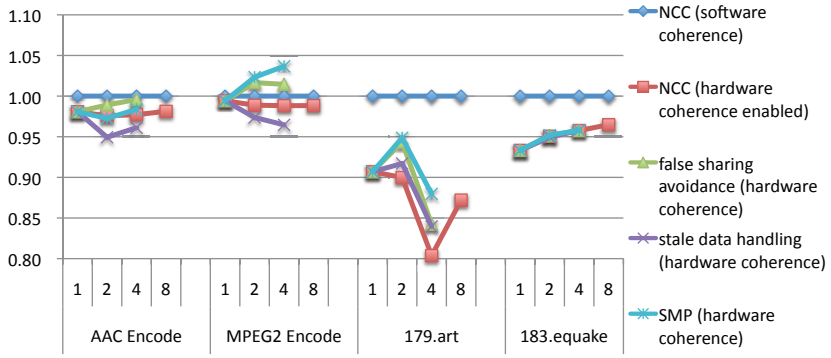

図 10 ソフトウェアコヒーレンシ制御における性能への影響要因
Fig. 10 Performance influence factors on software coherence.

一方で，AAC エンコーダと MPEG2 エンコーダにおいては，stale data handling (hardware coherence) が SMP(hardware coherence) に対して性能低下を起こしており，明示的なキャッシュ操作指示による影響が見えている．その結果，AAC エンコーダでは 4 コア使用時に 1.7%の性能向上にとどまり，MPEG2 エンコーダでは 4 コア使用時に 3.6%の性能低下を起こしている．ただし，本稿の評価では，キャッシュ操作についてはキャッシュ全体のフラッシュを行うというナイーブな実装を行っているため，今後ライトバックとセルフインバリデートの分離やキャッシュ操作対象のメモリ領域の限定等の最適化を行っていくことで，ソフトウェアコヒーレンシ制御の性能向上が見込まれる．しかしながら，RP2 のようにデータキャッシュの容量が 16KB 程度と小容量なアーキテクチャでは，キャッシュ全体をフラッシュする実装においても 3.6%程度の性能劣化にとどまっている．

6. ま と め

本稿では，並列化コンパイラによるソフトウェアコヒーレンシ制御手法を提案した．本手法は一般的なキャッシュハードウェアで利用可能であり，提案手法を OSCAR コンパイラに実装し，8 コア集積の情報家電用マルチコアである RP2 において，ソフトウェアによるコヒーレンシ制御により，ハードウェアによるコヒーレントキャッシュと遜色の無い，高い性能が得られた．さらに，ソフトウェアによるコヒーレンシ制御ではハードウェアがサポートしない 8 コア使用時までスケールアップし，マルチメディア処理および科学技術計算の 4 アプリケーションについて，コヒーレントキャッシュの逐次実行時の 4.88 倍の性能が得られた．

ソフトウェアコヒーレンシ制御をコンパイラで自動的に行うことにより，ハードウェアによるコヒーレンシ制御機能が不要となり，特に組み込み系のマルチコア等において，コスト・開発期間とも小さく抑えることが可能になる．さらに，多くのコア数まで性能がスケールアップする，32 コアから 64 コア，さらに 128 コアを越えるメニーコアプロセッサシステムを低コストで容易に開発できるようになり，将来の大規模マルチプロセッサ構築技術の大きな柱になると考えられる．

謝辞 本研究の一部は NEDO“情報家電用ヘテロジニアス・マルチコア技術の研究開発”，“低消費電力メニーコア・プロセッサシステム技術（グリーン IT プロジェクト）の先導研究”，および早稲田大学グローバル COE“アンビエント SoC” の支援により行われた．

参 考 文 献

1) Culler, D. E., Singh, J. P. and Gupta, A.: Parallel Computer Architecture - A Hardware / Software Approach, *Morgan Kaufmann Publishers, Inc* (1999).
2) Shiota, T. et al.: A 51.2 GOPS 1.0 GB/s-DMA single-chip multi-processor integrating quadruple 8-way VLIW processors, *ISSCC'05* (2005).
3) Tanabe, J. et al.: A.9.7mw aac-decoding, 620mw h.264 720p 60fps decoding, 8-core media processor with embedded forward-body-biasing and power gating circuit in 65nm cmos technology, *ISSCC'08* (2008).
4) 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 CPU and 8 RAMS by an Automatic Parallelizing Compiler, *ISSCC'08* (2008).
5) Kelm, J.H. et al.: Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator, *ISCA'09* (2009).
6) Howard, J. et al.: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS, *ISSCC'10* (2010).
7) M.Wolfe: High Performance Compilers for Parallel Computing, *Addison-Wesley Publishing Company* (1996).
8) Aho, A.V., Lam, M.S., Sethi, R. and Ullman, J.D.: Compilers : Principles, Techniques, and Tools Second Edition, *Pearson Education, Inc* (2007).
9) 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(通巻 458 号),

pp.384-392 (2003).
10) 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， Vol.J73-D-1, No.12, pp.951–960 (1990).
11) 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処理制御手法，情報処理学会論文誌， Vol.44, No.4 (2003).
12) 吉田明正，前田誠司，尾形　航，笠原博徳：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).
13) *Optimaly Scheduled Advanced Multiprocessor Application Program Interface (OSCAR API) version 1.0.* http://www.kasahara.cs.waseda.ac.jp/.
14) 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，情報処理学会研究会報告 2009-ARC-174-15(SWoPP2009) (2009).