計算機アーキテクチャ　83−21
（１９９０．７．１８）

統合型並列化コンパイラ・システム
−コンパイラ支援キャッシュ・コヒーレンス制御−

岩田英次　森　眞一郎　村上和彰　福田　晃　富田眞治
（九州大学総合理工学研究科）

現在筆者は，統合型並列化コンパイラ（IPC：Integrated Parallelizing Compiler）システム開発の一環として，コンパイラ支援キャッシュ・コヒーレンス制御アルゴリズムの開発を進めている．コンパイラ支援キャッシュ・コヒーレンス制御は，ハードウェアのみによる動的キャッシュ・コヒーレンス制御（スヌーピング・キャッシュ方式，グローバル・ディレクトリ方式，等）と比較して以下の特長を持つ．

①キャッシュは非透過型（non-transparent）である．各プロセッサは，コンパイラがコード中に挿入するコヒーレンス制御命令に従い，自キャッシュのみを無効化する．よって，実行時にコヒーレンス制御のためのプロセッサ間通信が発生しない．

②コンパイラが静的に行う，共有変数に関するデータフロー解析およびデータ依存解析の結果に基づき，コヒーレンス制御の最適化を図れる．

③コヒーレンス制御に必要なハードウェア量は，プロセッサ台数に関わらず一定である．したがって，大規模マルチプロセッサ・システムにも適用可能である．

以上の特長から，コンパイラ支援キャッシュ・コヒーレンス制御は，システム規模，相互結合網形態，プロセッサ間通信バンド幅，等に依存しないアプローチといえる．

本稿では，まず従来から提案されているコンパイラ支援キャッシュ・コヒーレンス制御アルゴリズムを概観したあと，IPCで採用する"高速選択的無効化アルゴリズム"および"バージョン制御アルゴリズム"について述べる．さらに，IPCのターゲット・マシンの一つである可変構造型並列計算機に対する，本制御アルゴリズムの実現方法について検討する．

Integrated Parallelizing Compiler System
−Compiler-Assisted Cache Coherence Scheme−

(in Japanese)

Eiji IWATA, Shin-ichiro MORI, Kazuaki MURAKAMI, Akira FUKUDA, and Shinji TOMITA
Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6-1, Kasuga-koen, Kasuga-shi, Fukuoka, 816 Japan
e-mail : iwata@kyu-is.is.kyushu-u.ac.jp

Compiler-assisted cache coherence schemes are proposed. They will be incorporated into the IPC (Integrated Parallelizing Compiler) system which has been developed at Kyushu University. These cache coherence scheme have the following features :

① Each processor has a private non-transparent cache. It can just invalidate the cache depending on a cache control instruction. A compiler generates such cache control instructions. Processors, therefore, do not need to intercommunicate with one another for enforcing cache coherence at run-time. As the result, network traffic can be reduced as compared with conventional hardware schemes.

② A compiler can utilize the data-flow and data-dependence analysis in order to optimize its cache coherence control.

③ Hardware cost for cache coherence control stays constant, regardless of the number of processors in a multiprocessor system. Large multiprocessor systems will welcome the advantage.

This paper, at first, gives background information on cache coherence, and reviews previous work on compiler-assisted cache coherence scheme. And it presents two algorithms − the fast selective invalidation algorithm and the version control algorithm, both of which will be implemented by IPC. Finally, it discusses several issues on adapting the algorithms on the Kyushu University Reconfigurable Parallel Processor.

1. はじめに

現在我々は，統合型並列化コンパイラ（IPC：Integrated Parallelizing Compiler）システム［1］,［2］開発の一環として，コンパイラ支援キャッシュ・コヒーレンス（CACC：Compiler-Assisted Cache Coherence）制御アルゴリズムの開発を進めている．CACC制御は，ハードウェアのみによる動的コヒーレンス制御（スヌーピング・キャッシュ方式［8］，グローバル・ディレクトリ方式［9］，等）と比較して以下の特長を持つ．

①キャッシュは非透過型である．各プロセッサは，コンパイラがコード中に挿入するコヒーレンス制御命令に従い，自キャッシュのみを無効化する．よって，実行時にコヒーレンス制御のためのプロセッサ間通信が発生しない．

②コンパイラが静的に行う，共有変数に関するデータフロー解析およびデータ依存解析の結果に基づき，コヒーレンス制御の最適化を図れる．

③コヒーレンス制御に必要なハードウェア量は，プロセッサ台数に関わらず一定である．したがって，大規模マルチプロセッサ・システムにも適用可能である．

以上の特長から，CACC制御は，システム規模，相互結合網形態，プロセッサ間通信バンド幅，等に依存しないアプローチといえる．

本稿では，まず従来から提案されているCACC制御アルゴリズムを概観したあと，IPCで採用する"高速選択的無効化アルゴリズム"および"バージョン制御アルゴリズム"について述べる．さらに，IPCのターゲットマシンの一つである可変構造型並列計算機（開発コード VIP：Variable Interconnection Parallel processor／Very Important Parallel processsor）［3］～［7］に対する，本制御アルゴリズムの実現方法について検討する．

2. キャッシュ・コヒーレンス制御

2.1 コヒーレンスと同期

いま，共有変数SVに対して，プロセッサPdが定義（書込み）を，プロセッサPrが参照（読出し）を行うものと仮定する．このとき，コヒーレンスおよび同期は，次のように定義される［14］．

①コヒーレンス：「PdがSVを定義したら，それ以降Prはその定義された値を参照しなければならない」という一貫性．

②同期：以下の順序関係（先行制約）を保証する手段．

i）PdとPr間にフロー依存関係（Pd→Pr）がある場合：Prは，PdがSVを定義するまでSVを参照してはいけない．

ii）PdとPr間に逆依存関係（Pr→Pd）がある場合：Pdは，PrがSVを参照するまでSVを定義してはいけない．

なお，同期は，上記の順序関係を保証する以外に，排他制御をも保証する手段である．

コヒーレンスを保証する方法は，次の2つのクラスに分類される［14］．

ⓐstrong ordering：常に，上記①のコヒーレンスの定義を満足させるような実現法．

ⓑweak ordering：上記②-i）を保証する同期手段を用いる．つまり，同期をとった場合にのみ，上記①のコヒーレンスの定義を満足させるような実現法．

ここで，コヒーレンスを保証することは，上記②のデータ依存や逆依存を保証することにはならない．

2.2 キャッシュ・コヒーレンス制御

各種のコヒーレンス制御方式は，2.1節で述べたコヒーレンス保証方法のクラス分類に応じて，同じく以下の2つのクラスに分類される．

ⓐstrong ordering：スヌーピング・キャッシュ方式，ブロードキャスト方式，等

ⓑweak ordering：グローバル・ディレクトリ方式，CACC制御方式，等

また，コヒーレンス制御の主体により，以下の2種類に分類できる．

①動的コヒーレンス制御：スヌーピング・キャッシュ方式，ブロードキャスト方式，グローバル・ディレクトリ方式，等

②CACC制御：2.4節参照

2.3 静的タスク・スケジューリングを前提としたコヒーレンス制御

従来のCACC制御アルゴリズムはすべて，動的タスク・スケジューリングを前提としている．そこで，ここではまず，静的タスク・スケジューリングを前提としたコヒーレンス制御について考察を行う．

図1のdoallループの並びを例にとって考える．以下，doallループの各イタレーションをタスクとする．キャッシュのラインサイズは1ワードとする．また，キャッシュのメモリ更新アルゴリズムはclean型（メモリに常に最新の更新結果が存在する）［7］とする．

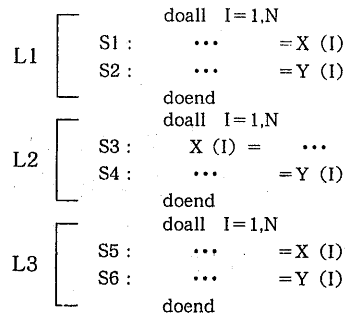

図1. doall ループ

図1の例では，以下の状況においてコヒーレンス制御が必要となる．

①ループL1のi番目のイタレーションがプロセッサP0に割り当てられ，P0が配列要素X（i）およびY（i）を参照する．

②L2のi番目のイタレーションがP1に割り当てられ，P1がX（i）を定義し，Y（i）を参照する．

③L3のi番目のイタレーションがP0に割り当てられ，P0がX（i）およびY（i）を再び参照する．

上記の状況で，配列要素X（i）について，③のP0は②のP1に対してフロー依存関係にある．したがって，③のP0は，P1が②においてX（i）を定義するまでX（i）を参照してはいけない（フロー依存の保証）．同時に，③のP0は，P1が②においてX（i）を定義したら，それ以降そのX（i）を参照しなければならない（コヒーレンスの保証）．weak orderingでは，同期によりフロー依存を保証すると同時に，コヒーレンスも保証する．

さて，静的スケジューリングであれば，どのプロセッサにどのタスクが割り当てられるかが，実行前に一意に定まる．よって，コンパイラはデータ依存解析の結果，プロセッサ間フロー依存を起こす共有変数を検出することが可能である．したがって，フロー依存関係にあるプロセッサは，当該共有変数を必ずメモリからロードするようにすればよい．これには，キャッシュのヒット／ミスヒットに関わらず，必ずメモリからロードを行う命令（メモリロード命令と呼ぶ）が必要である．図1の例では，ループL3のi番目のイタレーションの文S5をメモリロード命令とする．

このように，静的スケジューリングを前提とした場合，コヒーレンス制御は非常に簡単になる．しかし，実際問題として，静的スケジューリングが適用できる範囲には限界がある．よって，動的スケジューリングを前提としたコヒーレンス制御の必要性が生じる．

2.4 動的タスク・スケジューリングを前提としたコヒーレンス制御

従来から提案されている，4種類のCACC制御アルゴリズムを比較・検討する．表1に，これらのアルゴリズムの比較結果を示す．

2.4.1 Veidenbaumのアルゴリズム

このアルゴリズムは，1986年に米イリノイ大学のVeidenbaumが提案した［17］．

（1）前提

以下の3種のキャッシュ制御命令を備える．

①flush：キャッシュの全ラインを無効化する．

②cache_on：この命令以降のアクセスはキャッシュを経由する．

③cache_off：この命令以降のアクセスはキャッシュをバイパスする．

（2）コヒーレンス制御

doループのイタレーション間データ依存関係に応じて，以下のようにキャッシュ制御命令をオブジェクトコード中に挿入する．

①doall（データ依存なし）：ループの先頭でcache_on命令を実行して，キャッシュを経由させる．

②doacross（データ依存あり）：ループの先頭でcache_off命令を実行して，キャッシュをバイパスさせる．

③doserial（単一プロセッサ実行）：ループの先頭でcache_on命令を実行して，キャッシュを経由させる．

さらに，ループ境界では必ずflush命令を実行して，キャッシュの全ラインを無効化する．

（3）長所

キャッシュ全体を一括して無効化するため，処理時間が極めて小さ

表1. コンパイラ支援キャッシュ・コヒーレンス制御の各アルゴリズムの比較

パラメータ ＼ アルゴリズム		Veidenbaum	RP3およびNYU	Cheong&Veidenbaum	Cheong&Veidenbaum および Min&Baer
特長	コヒーレンス制御を行う単位	変数全体	個々の共有変数	個々の共有変数ロード命令	個々の共有変数
	コヒーレンス制御に必要な静的解析範囲	タスク（ループ）内のみ	タスク（ループ）内のみ	プログラム全体	プログラム全体
	タスク境界での処理	キャッシュ全体を無効化	共有変数を含むラインのみを選択的に無効化	キャッシュ全体を疑似的に無効化	指定された共有変数のバージョンを更新
	実行時情報の反映	不可	不可	不可	可
性能	タスク境界を越えてキャッシング可能な変数	なし	・非共有変数	・非共有変数 ・書込みのない共有変数 ・書込みのある共有変数（制限あり）	・非共有変数 ・書込みのない共有変数 ・書込みのある共有変数（制限なし）
	タスク境界での処理量	通常のキャッシュ一括無効化並み	ラインパージ処理×共有変数を含むライン数	通常のキャッシュ一括無効化並み	バージョン更新処理×指定された共有変数の数
実現	ロード命令の定義	通常のロード命令で可	特殊なロード命令が必要	特殊なロード命令が必要	通常のロード命令で可
	必要なハードウェア	不要	不要	・1ビット／ライン	・数ビット／ライン ・数ビット／共有変数

い.

(4) 問題点

doacrossループ中の変数は共有／非共有，書込み有り／無しに関わらず，すべてキャッシング不可である．また，ループ境界におけるキャッシュの一括無効化により，無効化の必要のない変数も無効化される．

2.4.2 RP3とNYUのアルゴリズム

これらのアルゴリズムは，米IBMワトソン研究所のRP3 [10] および米ニューヨーク大学のUltracomputer [15] で用いられている．Veidenbaumのアルゴリズムの問題点をある程度改善する．

(1) 前提

以下の2種類のロード命令を備える．

①ロードデータのキャッシング可

②ロードデータのキャッシング不可

また，キャッシュラインを選択的に無効化できる．

(2) コヒーレンス制御

ループ内において各共有変数への書込みがあるか／否かに応じて，以下のようにロード命令のタグを決定する．

①書込み有り：キャッシング不可

②書込み無し：キャッシング可

さらに，ループ境界では，共有変数を含むラインを選択的に無効化する．

(3) 長所

Veidenbaumのアルゴリズムの問題点を次のように改善している．

①doacrossループにおいても，非共有変数および書込みのない共有変数がキャッシング可となる．

②非共有変数がループ境界を越えてキャッシング可となる．

(4) 問題点

ループ境界でキャッシュラインを選択的に無効化するため，その処理時間が問題である．

2.4.3 Cheong&Veidenbaumのアルゴリズム

このアルゴリズムは，1988年に米イリノイ大学のCheongとVeidenbaumが提案した [11],[12]．文献 [11] に従い，本アルゴリズムを"高速選択的無効化アルゴリズム"と呼ぶ．

本アルゴリズムは，先の2つのアルゴリズムにおける問題点を同時に解決している．しかし，実行時情報をコヒーレンス制御に反映出来ない点で制限がある．

本アルゴリズムについては，3章で詳述する．

2.4.4 Cheong&VeidenbaumおよびMin&Baerのアルゴリズム

これらのアルゴリズムは，1989年にCheongら [13] とMinら [16] がそれぞれ独自に提案した．3章で述べる高速選択的無効化アルゴリズムの問題点を改良している．つまり，実行時情報をコヒーレンス制御に反映する．このためのハードウェア機構を導入している．

用いる実行時情報は共有変数およびキャッシュラインの世代であり，両アルゴリズムはこの世代管理を行うことでコヒーレンス制御を行う．Cheongらはこれをバージョンと，またMinらはタイムスタンプと呼んでいる．以後，両アルゴリズムを総称して，"バージョン制御アルゴリズム"と呼ぶ．

本アルゴリズムについては，4章で詳述する．

3. 高速選択的無効化アルゴリズム [11],[12]

3.1 前提

ラインサイズは1ワードとする．メモリ更新アルゴリズムはclean型である．

ライン対応にchangeビットと呼ぶ制御ビットを設ける．invalidate命令により，すべてのchangeビットを1サイクルでセットできる．

以下の2種類のロード命令を備える．

①memory__read：changeビットの値に応じて，次のようにキャッシュの経由／バイパスを決める．
- 0の場合：キャッシュを経由する．
- 1の場合：キャッシュをバイパスする．

②cache__read：通常のロード命令に相当する．必ずキャッシュを経由する．

3.2 コヒーレンス制御

まず，タスク境界では，invalidate命令により全changeビットをセットする．また，ラインフェッチの際には，当該ラインに対応するchangeビットをリセットする．

各変数のデータ依存解析結果に従い，当該変数をロードする命令を次のように決定する．

①read-only共有変数のロード命令：cache__read

②writable共有変数のロード命令：ロード命令とストア命令との順序関係により，次のようになる．

i) 最初のストア命令以前に実行するロード命令：cache__read

ii) 最初のストア命令以降に実行するロード命令：さらに，ロード命令とinvalidate命令との順序関係により次のようになる．

a) invalidate命令以前に実行するロード命令：cache__read（図2（a）参照）

b) invalidate命令以降に実行するロード命令：さらに，1つ前のinvalidate命令以前に当該変数のロードが行われたか否かで，次のようになる．

1) 行われていない場合：cache__read（図2（b-1）参照）

2) 行われた場合：memory__read（図2（b-2）参照）

上記②-ii）の場合分けは，2個のinvalidate命令で囲まれた区間のタスクは同一のプロセッサで実行されることを根拠にしている．

図1の例に本アルゴリズムを適用した場合，ループL3の文S5のX（i）ロード命令はmemory__read，文S6のY（i）ロード命令はcache__readとなる．

3.3 長所

RP3とNYUのアルゴリズム（2.4.2項）の問題点を次のように改善している．

①invalidate命令でキャッシュ全体を一括して疑似的に無効化する．そして，memory__read命令の機能により，真に無効化の必要なラインのみを無効化する．これは，実質的に選択的無効化に相当する．これにより，タスク境界での処理時間が小さくなる．

②タスク内で書込みのない共有変数がタスク境界を越えてキャッシング可となる．さらに，書き込まれた共有変数でも，3.2節における②-ii)-b)-1）の場合では，タスク境界を越えてキャッシング可となる．

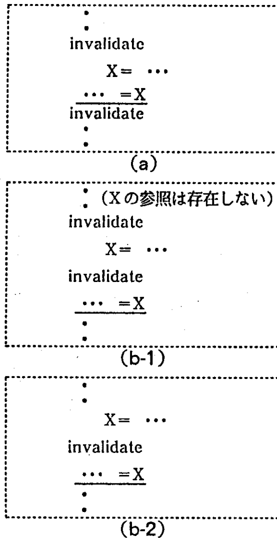

図2. 高速選択的無効化アルゴリズム

3.4 問題点

(1) 実行時情報

図1の例を用いて問題点を述べる. まず, 以下のような状況を仮定する.

①ループL1のi番目のイタレーションがプロセッサP0に割り当てられ, P0がX (i) およびY (i) を参照する.

②L2のi番目のイタレーションがP1に割り当てられ, P1がX (i) を定義し, Y (i) を参照する.

③L3のi番目のイタレーションがP1に割り当てられ, P1がX (i) およびY (i) を参照する.

このとき, ③のP1は, X (i) をキャッシュから得ても良いことがわかる.

ところが, 3.2節で述べたロード命令決定の規則に従うと, これは②-ii)-b)-2) の条件に相当し, memory__read命令となる. ループL2とL3の境界においては, invalidate命令が全changeビットをセットする. よって, 当該memory__read命令は, キャッシュ内に有効なX (i) が存在するにも関わらず, X (i) をメモリからロードすることになる.

この問題の原因は,「②のX (i) の定義を行ったP1上で, ③のX (i) の参照も行われる」といった実行時にしか判明しない情報がコヒーレンス制御に反映できない点にある.

(2) 1ワード/ライン

3.1節で述べたように, ラインサイズは1ワードを前提としている. しかし, この前提は, キャッシュ構成法として現実的でない. 1ラインが複数ワードから成る場合, 以下のいずれかの対策が必要となる [11].

①3.2節における②-ii)-b)-1) の条件に, さらに制限を加える.

②ライン中の各ワード対応に, 1ビットの制御ビットを追加する.

4.バージョン制御アルゴリズム [13],[16]

4.1 準備

(1) タスクグラフ

タスク間のデータ依存や制御依存といった先行制約を表現するのに, 図3 (a) に示すようなタスクグラフを用いる. タスクグラフは, DAG (Directed Acyclic Graph) とする.

タスクグラフはタスクレベルに分割されており, 同一レベルのタスク間にはデータ依存がない. つまり, 同じ共有変数にアクセスするタスクは存在しない.

(2) バージョン

共有変数およびライン対応に, 以下のバージョンを設ける.

①SVV : 共有変数のバージョンで, 各プロセッサの専有格納メモリ (ないしレジスタ) 内に置かれる.

②CLV : キャッシュラインのバージョンでタグ内に置かれる.

SVVに対しては, 命令によるアクセスが可能である. これらバージョンの世代管理を行うことで, コヒーレンス制御を行う.

(3) キャッシュ

ラインサイズは, 1ワードとする. メモリ更新アルゴリズムはclean型を前提とする. dirty型を用いる場合, 各タスクの実行が終了した時点で, そのタスクが更新した共有変数をメモリに書き戻す操作が必要となる.

4.2 コヒーレンス制御

(1) SVVの世代管理

コンパイル時, 同一タスクレベルに属する各タスクが書き込む可能性のある共有変数を求め, VARt (tはタスクレベル) と呼ぶリストに登録しておく (図3参照).

実行時, 各プロセッサに対する動的タスクスケジューリングの結果である以下の情報を用いる.

①i : 前回当該プロセッサに割り当てられたタスクのタスクレベル

②k : いまから当該プロセッサに割り当てられるタスクのタスクレベル

タスク割当て時, 当該タスクは上記のVARtおよびi, kを用いて, 以下の式を満足する共有変数の集合を求める. これは, タスクレベルi～ (k－1) 間で書き込まれた可能性のある共有変数の集合である.

$$\bigcup_{j=0}^{j=(k-1)-i} \mathrm{VAR}i+j \qquad \text{(式1)}$$

そして, この集合に含まれる各共有変数のSVVを1だけインクリメントする. すなわち, バージョンアップする.

(2) CLVの世代管理

CLVは, ハードウェアにより, 以下のように更新される.

①ラインフェッチ時 : フェッチした共有変数のSVVをCLVに代入.

②ストアアクセス時 : ストアした共有変数のSVVを1だけインクリ

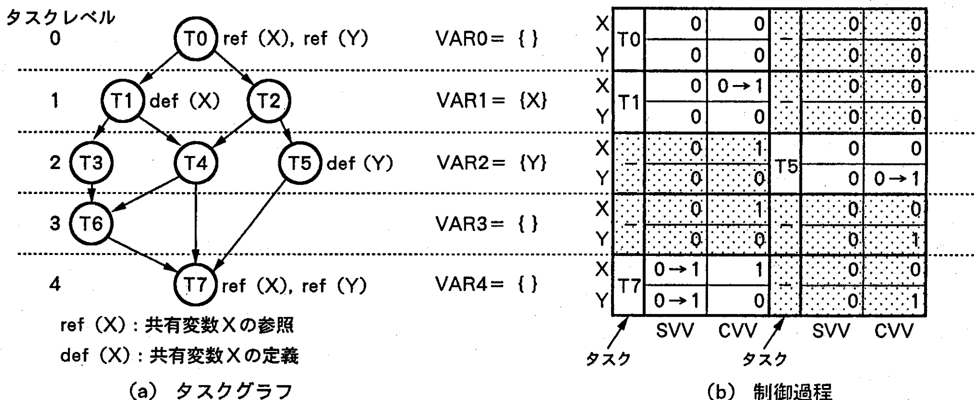

図3. バージョン制御アルゴリズム

メントしてCLVに代入.

(3) 世代のずれの修正

ロードアクセス時, SVVとCLVとを比較する. もし, SVV>CLVならばミスヒットとする.

4.3 例題

いま, 図3 (a) に示すように, 共有変数XおよびYに対して, 以下のタスクのみが定義/参照するものとする.

- X定義 : T1
- X参照 : T0およびT7
- Y定義 : T5
- Y参照 : T0およびT7

したがって, 共有変数XおよびYに注目した場合, 各VARt (t=0〜4) は, 図3 (a) のようになる.

また, 2台のプロセッサP0およびP1に対して, 上記のタスクが以下のように割り当てられたとする.

- P0 : T0→T1→T7
- P1 : T5

このとき, プロセッサP0上のタスクT7は, 共有変数Xをキャッシュから得ることができる. しかしながら, 共有変数Yはメモリから得なければならない. なぜなら, P0のキャッシュ中のYのコピーはT5が定義する以前のものであり, Yの最新値はメモリに存在するからである.

4.2節で述べたアルゴリズムがどのようにコヒーレンス制御を行ったかを順を追って説明する (図3 (b) 参照). 以下,

- SVVP (V) : プロセッサPにおける共有変数VのSVV
- CLVP (V) : プロセッサPにおいて, 共有変数Vのコピーを有するラインのCLV

とする (初期値は0).

①T0はP0に割り当てられた時,

- SVV0 (X) ←0
- SVV0 (Y) ←0

とする. そして, X, Yをロードした時,

- CLV0 (X) ←SVV0 (X) =0
- CLV0 (Y) ←SVV0 (Y) =0

となる.

②T1はP0に割り当てられた時, SVV0 (X) およびSVV0 (Y) を更新しない. Xをストアした時,

- CLV0 (X) ←SVV0 (X) +1=1

となる.

③T5はP1に割り当てられた時, SVV1 (X) およびSVV1 (Y) を更新しない. Yをストアした時,

- CLV1 (Y) ←SVV1 (Y) +1=1

となる. このとき, P0では

- CLV0 (Y) =0

のままである点に注意されたい.

④T7はP0に割り当てられた時, 式1により,

- SVV0 (X) ←SVV0 (X) +1=1
- SVV0 (Y) ←SVV0 (Y) +1=1

とする. そして, Xをロードした時,

- SVV0 (X) =CLV0 (X)

なのでキャッシュヒット. 一方, Yをロードした時は,

- SVV0 (Y) >CLV0 (Y)

となるので, キャッシュミスとなる.

4.4 問題点

(1) ハードウェア量

バージョン制御アルゴリズムの実現に当たっては, SVVおよびCLVを格納するメモリ等はじめ, バージョンのインクリメント/比較のためのハードウェアが必要となる. したがって, 他のアルゴリズムに比べてハードウェア量が多くなる. ただし, 動的コヒーレンス制御方式とは異なり, このハードウェア量はプロセッサ台数に関わらず一定である.

(2) 1ワード/ライン

4.1節で述べたように, ラインサイズは1ワードを前提としている. しかし, この前提は, キャッシュ構成法として現実的でない. よって, 1ラインが複数ワードから成る一般的なキャッシュ構成における, 本アルゴリズムの適用方法が課題となる. これについては, 5.3.1項で検討する.

(3) バージョンアップ

タスク割当て時における, SVVのバージョンアップ処理が問題である. この処理時間は, バージョンアップすべき共有変数の数に比例する. この問題は, タスク粒度, SVVと共有変数との対応, 等と密接に絡む. これについては, 5.3.2項で検討する.

(4) 循環タスクグラフ

4.1節で述べたように, タスクグラフはDAGを前提としている. よって, サイクルを含むタスクグラフに対して本アルゴリズムを適用する場合, 問題が生じる. すなわち, アルゴリズムで用いるiとk (4.2節) が, サイクルの同一イタレーションに属しているのか否かを認識することが難しい. よって, サイクルを構成する全タスクレベルのVARに登録されている共有変数について, そのSVVをバージョンアップする必要がある [13].

5. CACC制御アルゴリズムの実現

統合型並列化コンパイラ (IPC) システムのターゲットマシンの1つである可変構造型並列計算機VIPに対して, CACC制御アルゴリズムの適用を検討する. なお, VIP自身は, 分散グローバル・ディレクトリ方式に基づく動的コヒーレンス制御機構も備えている [6],[7].

5.1 アルゴリズムの選択

IPCでは, CACC制御アルゴリズムとして, 高速選択的無効化アルゴリズム (3章) およびバージョン制御アルゴリズム (4章) の双方を採用する. これは, 以下の理由による.

(1) 性能

表1に示すように, 両アルゴリズムとも, 非共有変数および共有変数がタスク境界を越えてキャッシング可能である. すなわち, 他の2つのアルゴリズムに比べて効率的である.

また, 両アルゴリズム間の優劣については, プロセッサ数が少ない場合バージョン制御アルゴリズムの方が性能 (ヒット率) が良いが, 多くなると両アルゴリズムの性能はほぼ同程度になることが報告されている [13].

(2) 実現

両アルゴリズムとも一長一短がある.

①高速選択的無効化アルゴリズム : 追加するハードウェアは, ライン当り1ビットで済む. しかし, 命令セットとして, 特殊なロード命令, すなわち, memory__read命令 (3.1節) が必要となる.

②バージョン制御アルゴリズム : 命令セットしては通常のロード命令だけでよく, 汎用マイクロプロセッサを用いたシステムに向いている. しかし, SVVおよびCLVをはじめ, バージョンのインクリメント/比較のためのハードウェアが必要となる.

5.2 高速選択的無効化アルゴリズム

本アルゴリズムは, 3.1節で述べたように, 以下の3種類の命令が必要である.

①memory__read命令 : 特殊なロード命令

②cache__read命令 : 通常のロード命令

③invalidate命令 : 全ラインのchangeビットの一括セット命令

VIPは, プロセッサとして汎用32ビット・マイクロプロセッサSPARCを搭載する. また, キャッシュおよび仮想アドレス変換を制御するためのMMU (メモリ管理ユニット) を備える [3].

まず, 上記命令の内, ②のcache__read命令は通常のロード命令なので問題ない. また, ③のinvalidate命令は, MMUに対するコマンドとしてサポート可能である. しかし, ①のmemory__read命令は, SPARCの命令セットにない命令である. よって, 本命令そのものを直接サポートするのではなく, その機能を間接的に実現する方法を考える.

そのために, 対MMUコマンドとして, 次の2つの命令を設ける.

ⓐmemory__read__on命令 : キャッシュは, この命令以降のロード命令をmemory__read命令として処理する.

ⓑmemory__read__off命令 : キャッシュは, この命令以降のロード命令をcache__read命令として処理する.

これにより, memory__read命令に相当するロード命令の前後をmemory__read__on命令およびmemory__read__off命令で囲めばよいことになる. ただし, このままだと, 本来1命令で済むところが単純に3命令へと増加してしまう. これには, 当該ロード命令の出現位置を出来るだけ特定領域に局所化して, 当該領域の入/出口にのみmemory__read__on命令およびmemory__read__off命令を置くようにすればよい. これは, 静的コード・スケジューリング [2] により可能である.

5.3 バージョン制御アルゴリズム

5.3.1 CLVと共有変数との対応

4.4節の (2) で述べたように, 1ワード/ラインという前提は, キャッシュ構成法として現実的でない. 1ラインが複数ワードから成る一般的なキャッシュ構成においては, CLVと共有変数との対応関係

を考慮する必要がある．これには，以下の2種類が考えられる．
(1) CLV：共有変数＝1：1
この対応関係を保持する方法としては，次の2つの選択肢が可能である．
①1ライン当りのワード数と等しいだけのCLVをタグに設ける．しかし，これはハードウェア量の増加をもたらす．
②1ライン当り1個のCLVしか設けない．ただし，1ラインには高々1個の共有変数しか存在させない．しかし，データ配置アルゴリズムが複雑になる．
(2) CLV：共有変数＝1：多
1ライン当り1個のCLVしか設けない．しかも，1ライン当り複数の共有変数の存在を許す．つまり，1個のCLVが複数の共有変数に共用される．この場合，これらCLVを共用する複数の共有変数を単一の共有変数とみなし，タスクグラフに反映させる必要がある．

5.3.2 SVVと共有変数との対応

4.4節の(3)で述べたように，SVVと共有変数との対応関係をどのように定めるかで，コヒーレンス制御の効率，バージョンアップ処理量，タスク粒度，等が影響を受ける．
(1) SVV：共有変数＝1：1
最もきめ細かなコヒーレンス制御が可能である．ただし，タスク割当て時におけるバージョンアップ処理で，バージョンアップすべきSVVをすべて1個ずつインクリメントする必要がある．
①細粒度タスク：個々のタスクがアクセスする共有変数は，たとえば配列の中の数要素であることから，きめ細かなコヒーレンス制御が適する．しかし，高々数ステップ程度のタスクの場合，バージョンアップすべきSVVがステップ数以上に多くなると，タスク実行時間に比べて，バージョンアップ処理時間の方が大きくなってしまう可能性がある．
②粗粒度タスク：たとえば，数千ステップ以上のタスクの場合，バージョンアップすべきSVVが数十程度となっても問題とはならない．ただし，個々のタスクがアクセスする共有変数が配列の全要素である場合，バージョンアップ処理時間は配列要素数に比例して大きくなる．ところが，この配列の全要素は単一の共有変数と見なせるので，要素毎にバージョンアップするのはまったくのオーバヘッドである．
(2) SVV：共有変数＝1：多
たとえば，仮想ページ単位にSVVを設け，当該ページ内の共有変数を単一のバージョンで制御する．上記(1)に比べて，きめ細かなコヒーレンス制御は不可能である．しかし，上記(1)②のように，配列の全要素が単一の共有変数と見なせる場合には適している．

5.3.3 SVV格納メモリの構成法

4.1節で述べたように，SVVは各プロセッサ内の専用メモリに格納される．この格納メモリに対するアクセスには，以下の3種類がある．
①CLVとの比較のため，キャッシュのタグアクセスと同時にアクセスする．使用アドレスは共有変数のアドレスであり，キャッシュ同様の連想アクセスとなる．
②SVVのインクリメントおよび初期化のために，命令が明示的にアクセスする．使用アドレスは，共有変数のアドレス（連想アクセス），あるいは，格納メモリ自身のアドレス（ランダムアクセス）となる．
③必要なら，SVVを格納メモリと主メモリとの間で退避／復元する．
SVV格納メモリの構成法として，次の2種類が考えられる．
(1) キャッシュとは独立した連想メモリ
キャッシュのタグアレイとは別に，独立のタグアレイを有する連想メモリとする．SVVと共有変数との対応関係（5.3.2項）は任意に定められる．問題点は，タグアレイを独立させた分だけハードウェア量が増えることである．
(2) キャッシュと連動する連想メモリ
キャッシュのタグアレイを共用する．よって，SVVと共有変数との対応関係は，CLVと共有変数との対応関係（5.3.1項）に等しくなる．つまり，SVVは共有変数もしくはライン対応に設けることになる．問題点は，ラインリプレースの際のSVVの退避／復元処理である．

5.3.4 VIPにおける実現

以下の仕様で，本アルゴリズムを現在VIP上に実装中である．
・ラインサイズ：8ワード（4バイト／ワード）
・CLV：ライン単位
・SVV：ページ（4KB）単位
・SVVとCLVのビット長：8ビット

6. おわりに

以上，IPCで採用した2種類のCACC制御アルゴリズムについて述べた．また，VIPへのこれらのアルゴリズムの適用方法について検討した．VIP自身は動的コヒーレンス制御機構を備えているが，本稿で述べたCACC制御アルゴリズムと互いにその能力を相補うことで，より効率的なコヒーレンス制御が期待できる．特に，128台のプロセッサをクロスバー網で相互結合したVIPの構成上，今後CACC制御に比重が移るものと考える．
なお，これらのアルゴリズムをVIP上に適用するのに必要となるハードウェアの詳細，ないし，VIPのハードウェア構成を意識したアルゴリズムの最適化，等については別の機会に報告する．

謝辞

IPCグループの音成　幹，赤星博輝の両氏，VIPグループの甲斐康司，上野智生，徳永尚哉の各氏，ならびに，日頃御討論頂く富田研究室の皆様に感謝致します．

参考文献

関連文献

[1] 村上ほか："統合型並列化コンパイラ・システム－概要－，" 情処40全大論文集，1G-1（1990年3月）．
[2] 入江ほか："統合型並列化コンパイラ・システム－局所コード・スケジューリング技法－，" 情処40全大論文集，1G-3（1990年3月）．
[3] K.Murakami et al.: "The Kyushu University Reconfigurable Parallel Processor－Design of Memory and Intercommunication Architectures－，" Proc. 1989 Int'l. Conf. Supercomputing, pp.351-360, June 1989.
[4] 森ほか："可変構造型並列計算機のPE間メッセージ通信機構，" 情報処理学会論文誌，vol.30, no.12, pp.1593-1602（1989年12月）．
[5] 甲斐ほか："可変構造型並列計算機のローカル／リモート・メモリ・アーキテクチャ，" 情処研報，90-ARC-80-11（1990年1月）．
[6] 岩田ほか："可変構造型並列計算機におけるキャッシュ・コヒーレンス処理，" 情処39全大論文集，5X-2（1989年10月）．
[7] 岩田ほか："可変構造型並列計算機のキャッシュ・システム，" 情処研報，89-ARC-79-3（1989年11月）．

一般文献

[8] J.Archibald and J.-L.Baer: "Cache Coherence Protocols : Evaluation Using a Multiprocessor Simulation Model, "ACM Trans. on Computer Systems, vol.4, no.4, pp.273-298, Nov. 1986.
[9] J.Archibald and J.-L.Baer: "An Economical Solution to the Cache Coherence Problem, "Proc. 11th Int'l Symp. Computer Architecture, pp.355-362, June 1984.
[10] W.C.Brantley, K.P.McAuliffe, and J.Weiss: "RP3 Processor-Memory Element, "Proc. 1985 Int'l Conf. Parallel Processing, pp.782-789, Aug. 1985.
[11] H.Cheong and A.V.Veidenbaum: "A Cache Coherence Scheme with Fast Selective Invalidation, "Proc. 15th Int'l Symp. Computer Architecture, pp.299-307, June 1988.
[12] H.Cheong and A.V.Veidenbaum: "Stale Data Detection and coherence Enforcement Using Flow Analysis, "Proc. 1988 Int'l Conf. Parallel Processing, Vol.I Architecture, pp. 138-145, Aug. 1988.
[13] H.Cheong and A.V.Veidenbaum: "A Version Control Approach to Cache Coherence, "Proc. 1989 Int'l Conf. Supercomputing, pp.322-330, June 1989.
[14] M.Dubois, C.Scheurich, and F.A.Briggs: "Synchronization, Coherence, and Event Ordering in Multiprocessors, " IEEE Computer, vol.21, no.2, pp.9-21, Feb. 1988.
[15] Jan Edler et al.: "Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach, "Proc. 12th Int'l Symp. Computer Architecture, pp.126-135, June 1985.
[16] S.L.Min and J.-L.Baer: "A Timestamp-based Cache Coherence Scheme, "Proc. 1988 Int'l Conf. Parallel Processing, Vol.I Architecture, pp.23-32, Aug. 1988.
[17] A.V.Veidenbaum: "Compiler-assisted Cache Coherence Solution for Multiprocessor, "Proc. 1986 Int'l Conf. Parallel Processing, pp.1029-1036, Aug. 1986.