並列計算機 JUMP-1 における分散共有メモリ管理

秤谷雅史[†,††] 斎藤康二[†] 小西将人[†]
五島正裕[†] 森眞一郎[†] 富田眞治[†]

並列計算機 JUMP-1 は，物理的構成に合わせた，多階層/クラスタリングされたアーキテクチャを持つ．クラスタに分散配置された共有主記憶の一部をリモートに対する 3 キャッシュとして利用する．大容量の 3 次キャッシュによって平均メモリ・アクセス・レイテンシを短縮する一方で，クラスタ間の処理はプログラム・ベースで柔軟に行う．そのプログラムを実装して評価したところ，リモートへのアクセス・レイテンシは 409 サイクルとなった．

Distibuted Shared Memory System of the JUMP-1 Multiprocessor

Masashi Hakariya,[†,††] Yasuji Saito,[†] Masahito Konishi,[†]
Masahiro Goshima,[†] Shin-ichiro Mori[†] and Shinji Tomita[†]

The JUMP-1 multiprocessor has hierarchical, clustered architecture reflecting its physical configuration. JUMP-1 uses a part of the shared main memory distributed among clusters as a third-level cache for remote clusters. While the large third-level cache improves average memory access latency, inter-cluster consistency control is flexibly performed by a special processor and its program. The evaluation result shows JUMP-1 performs remote memory access in 409 cycles.

1. はじめに

計算機ハードウェアは普通，物理的な階層構造を持つ．例えば現在の計算機は，筐体，ボード，チップといった要素によって階層的に実装されている．このような階層的な実装は，やはり多階層化，クラスタリングされたアーキテクチャの採用を促す．

この傾向は，システムの大部分がチップに集積されるようになっても緩和されるものではいない．例えばメモリ・ウォールは，以前には，プロセッサとメモリのチップ間の速度差の問題として捉えられてきた．LSI の集積度が向上するとそれは，短い配線と，ワード線やビット線といった長い配線の遅延の差に置き換わる．この遅延の差は，LSI の微細化が進み配線遅延が支配的なるにつれて，むしろ大きくなる．例えば最近のプロセッサでは 1MB 近くの SRAM が集積できるようになってきたが，それは 1 次キャッシュの増量にではなく，2 次キャッシュの集積に充てられる．それは，1MB もの SRAM をプロセッサ・コアと同等のスピードで動かすことは困難だからである．

このような傾向に逆らって，単階層で強力なメモリ・システムやフラットなアーキテクチャを採用することは，不可能ではないにしても，高コストとなる．コストの増大は，将来受け入れ難いものになるであろう．

したがって，アーキテクチャは階層性を受け入れ，そのために生じる不具合をソフトウェアで解消するというアプローチが，将来的に重要になると考えられる．

しかし確かに現状では，ソフトウェア技術が未成熟なため，商用の計算機にとって，階層的なアーキテクチャを採用することは現実的な解ではない．

そこで，本稿で述べる並列計算機 **JUMP-1** は，階層的なアーキテクチャ上のソフトウェアに対するテストベッドの提供を目標の 1 つとして開発された．JUMP-1 は，階層的な実装に合わせたクラスタ構造を持つ分散共有メモリ (**DSM**) 型並列計算機である．

本稿では，JUMP-1 の DSM について述べる．JUMP-1 では，DSM もやはり多階層的である．各クラスタに分散された主記憶をリモートに対する大容量のキャッシュとして利用し，それによってメモリ・アクセスの平均レイテンシを短縮する一方，クラスタ間の処理はプロセッサによって，プログラム・ベースで柔軟に行うというアプローチを採る．

以下まず，2 章で，JUMP-1 の DSM について概説する．次いで，3 章で DSM を管理する上での問題点について論じた後，4 章で DSM 管理プログラムについて説明する．そして 5 章で，開発したプログラムの評価結果について述べる．

† 京都大学情報学研究科 Grad. School of Informatics, Kyoto U.
†† トヨタ自動車株式会社 Toyota Motor Co. Ltd.

2. DSM 管理

本章では，まず 2.1 節でアーキテクチャ全体について概説した後，2.2 節以降で，DSM について説明する．

2.1 アーキテクチャの概要

JUMP-1 はクラスタ構造を採用している．クラスタは主に，4 つのプロセッサと主記憶からなる．システム全体は，複数のクラスタがクラスタ間ネットワークで接続された構造を持つ．

図 1 にクラスタのブロック図を示す．クラスタは，いわゆる 4-way の SMP に類似した構造を持っている．1 クラスタのプロセッサ数 —— 4 は，1 枚のボードに搭載できるという実装上の制約によって決められている．クラスタ内の 4 つのプロセッサは，1 枚のボード上にあるという利点を活かし，クラスタ・バスというスヌープ・バスによって接続され，高バンド幅/低レイテンシの通信を行うことができる．

プロセッサ・ユニットは主に，要素プロセッサ SuperSPARC+と，そのプライベート・キャッシュからなる．プライベート・キャッシュは，SuperSPARC+の内蔵 1 次と，外部 2 次[1),2)]の 2 階層からなる．表 1 に，キャッシュのパラメータを示す．

メモリ・ユニットの中央には，**MBP-light**[3)] と呼ぶ専用の LSI がある．図 1 に示したように，MBP-light は主に 4 つのポートを持ち，それぞれ，クラスタ・バス，主記憶，クラスタ間ネットワーク，そして，ローカル・バスに接続されている．ローカル・バスには，STAFF-Link I/O ネットワーク，メンテナンス用のネットワークなどが接続される．

4.1 節，図 2 に，MBP-light 内部のデータ・パスを示す．MBP-light は，主に，コア・プロセッサ **MBP Core** と Main Memory Controller (**MMC**) からなる．MBP-light については，4.1 節で詳しく述べる．

クラスタ間ネットワークは，**Recursive Diagonal Torus** (**RDT**)[5)] と呼ばれる．RDT は，基本のトーラスの上に目の粗いトーラスを 45° ずつ傾けながら最大 4 段まで再帰的に積み上げた，やはり多階層的な構造を持つ．RDT のノード —— すなわちクラスタは，基本トーラスと，1 つの粗いトーラスのノードを兼務する．したがって，クラスタあたりのリンクの数は 8 である．

RDT は，トーラスを基本構造としながらも，最も粗いトーラスのノードを根とする Fat Tree 状の構造が埋め込まれており，マルチキャスト/コンバイニングを効率よく行うことができる．この能力は，DSM 管理にも重要な役割を果たす．

2.2 アドレシング・システム

各クラスタの主記憶の一部 —— 例えば半分程度は，リモートのクラスタの主記憶に対するキャッシュとしても使われる．前述したように，要素プロセッサ

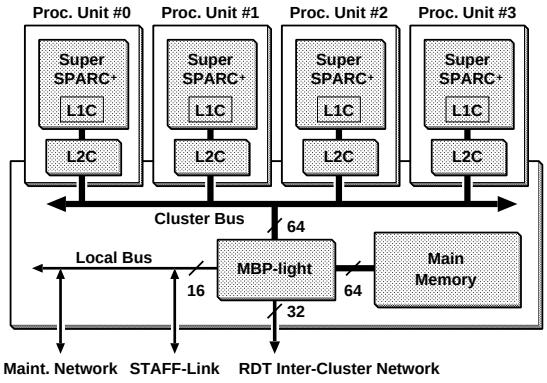

図 1　JUMP-1 のクラスタ

SuperSPARC+は内蔵 1 次/外部 2 次のプライベート・キャッシュを持つから，このキャッシュはクラスタ内の 4 つのプロセッサによって共有される第 3 次のキャッシュにあたる．表 1 に，そのパラメータを示しておく．

アドレス・マッピング

3 次キャッシュのアドレス体系は，**Shared Virtual Memory** (**SVM**)[4)] に準ずる．SVM では，キャッシングはページを単位として行われる．リモートにあるオリジナルとなるページをローカルにキャッシングするには，ローカルの主記憶上にページ枠を確保し，そこにオリジナル・ページをコピーする．

SVM では，クラスタ内では，通常の仮想記憶のシステムをそのまま用いることができる．アドレス変換は，アクセスされる物理ページがコピーかオリジナルかに関わらず，要素プロセッサの TLB によって自動的かつ高速に行うことができる．また，クラスタ内のキャッシングは，物理アドレスを基に行えばよい．なお，ページのサイズは TLB が規定することになる．SuperSPARC+ MMU では 4KB である．

一方クラスタ間では，テーブル検索によるアドレス変換が必要になる．SVM では，1 つの仮想ページに対して，クラスタごとに自由に物理アドレ割り当てることができる．そのためクラスタ間では，オブジェクトを一意に特定するための大域的な仮想アドレスが必要となる．そのようなアドレスとしては，普通，プロセスの仮想アドレスを大域的なプロセス ID で拡張したものを用いる．その場合，クラスタ間の一貫性制御を行うには，送信時には逆引きページ・テーブルの，受信時はページ・テーブルの，検索が必要となる．

レベル		連想度	サイズ	ライン サイズ	レイテンシ (サイクル)
1 次	命令	5	20KB	32B	1
	データ	4	16KB		
2 次		1	1MB		5
3 次		∞	—		29

表 1　キャッシュのパラメータ

一貫性制御

基本的なSVMでは，ミスや共有データへの書き込みは，ページ・フォルトを起こし，一貫性制御はOS，あるいは，それに類するソフトウェアによって行われる．

JUMP-1では，false sharingを軽減するために，ページではなく，32Bのラインを単位として一貫性制御を行う．そのために主記憶にはライン単位でタグが付加されている．

仮想ページへの最初のアクセスは，通常の仮想記憶システムと同様，ページ・フォルトとなる．OSは，必要であればリプレースの処理を行って，ページ枠を確保し，ページ・テーブルなどの設定を行う．この時点で，1ラインをフェッチするか，あるいは，ページ全体をプリフェッチするかは，戦略による．それ以降のアクセスはページ・フォルトにはならず，一貫性制御はライン単位で行われる．

2.3 一貫性制御プロトコルの概要

クラスタ内のスヌープ方式に対して，クラスタ間では標準的な分散ディレクトリ方式に基づいて行う．

ディレクトリの方式

ディレクトリのマップの形式は，Coreのプログラム次第である．縮約階層ビットマップ形式を用いると，クラスタ間ネットワークRDTのマルチキャスト/コンバイニング機能を利用することができる[5]．

SVMのオリジナルとなるページを持つクラスタに，そのページのディレクトリも持たせることが自然である．このクラスタをそのページの **home** と呼ぶ．

homeのマッピングもやはり自由なので，クラスタ間の一貫性制御を開始する際には，homeを求めるためのテーブル検索が必要となる．

一貫性維持動作

一貫性維持動作としては，無効化型に加えて，更新型とそれらの組み合わせをサポートする．これらの動作は，書き込み時だけでなく，読み出し時にも起動することができる[1]．

ラインの状態

標準的なMOESIプロトコルに準ずる．ただし，アーキテクチャの階層性に合わせて，コピーの存在範囲が，プロセッサ，クラスタ，システム全体 であることに対応する，exclusive，local shared，glboal sharedの3段階の共有状態を持ち，トランザクションができる限り狭い範囲で閉じるように工夫されている．

2次キャッシュは，スヌープに対してキャッシュの状態を応答するなどして，主記憶アクセスの頻度を低減する．2次キャッシュが処理できない場合に限り，MMCは主記憶アクセスを行う[1]．MMCは，local/globalのタグを読み出し，localである場合には自動的に処理する．globalであった場合には，Coreに通知し，パケットを引き渡す．したがって，一貫性制御を行う上でのMBP Coreの役割は，クラスタ間の処理に限定される．

3. トランザクション

プロセッサからの1つメモリ・アクセスに起因する一連の動作を **トランザクション** という．システムの各部におけるトランザクションの処理は，基本的には，到着したパケットに対応して，データ・アレイと各種のディレクトリを更新し，必要であればいくつかのパケットを送出する —— これを繰り返せばよい．ただし，必ずしもパケットの届いた順序で，すなわちFIFOで，処理すればよいというわけではない．

本章では，トランザクション，あるいは，パケットの処理順序の問題について議論する．以下まず，3.1節で，トランザクションにおけるパケットの流れをまとめる．そして，3.2節でトランザクションの競合と呼ばれる状況について説明した後，トランザクションの処理順序について，3.3節で論じる．

3.1 トランザクションの流れ

Coreによってプログラム・ベースで行われるので，変更可能であるが，三角通信などを考えず，すべてのトランザクションは一旦homeを経由するものとする．

システム内で唯一のラインを保持しているクラスタを **owner**，home以外でラインを共有しているクラスタを **renter** と呼ぶ．また，トランザクションを開始したクラスタは **initiator** と呼ぶ．

1つのトランザクションにおけるパケットの流れは，基本的には，以下の4つのフェーズからなる;すなわち，要求パケットが ① initiatorからhome，② homeからowner/renterへと送られ，それに対する応答パケットが ③ owner/renterからhome，そして，④ homeからinitiatorに返される．owner/renterとは，読み出しに対してはownerを，書き込み（無効化/更新）に対してはrenterを意味する．もちろん，homeのコピーの状態によっては，② と ③ は省略できる．

3.2 トランザクションの競合

同一のラインに対するトランザクションが，別々のクラスタからほぼ同時に開始されることがある．その場合homeでは，先行するトランザクションの ③ 応答パケットがrenterからまだ返って来ていないうちに，② 後続の要求パケットが到着してしまう．このような状況を **トランザクションの競合** という．以下本稿では，単に競合ということにする．

先行するトランザクションの応答を待たずに後続のパケットの処理を開始することは，一般に面倒である．したがって普通，homeでは，未応答のトランザクションの情報を記録することによって競合を検出し，競合を検出した場合，先行するトランザクションの応答が返るまで，何らかの方法によって，後続のトランザクションの処理の開始を遅延させる．

無効化トランザクションの競合

競合のうち，無効化トランザクションの「相撃ち」

について，特に考えておく必要がある．

無効化型の場合，書き込みに対してラインを不可逆的に更新するのは,「相撃ち」に「負け」なかったことが判明した後でなければならない．それ以前にラインを不可逆的に更新してしまうと,「負け」ていた場合に書き込みの内容が失われてしまうからである．

余分なパケットを流さないとすると,「負け」なかったことが判明するのは ④ home から応答が返される，すなわち，トランザクションが完了する時である．それまでは，書き込みの情報を保存しておく必要がある．

したがって，同時に未完了 (pending) にできる無効化要求の数には，書き込みの情報を保持するバッファの数による制約が生じる．各プロセッサごとに，JUMP-1 では 3 つ，Cenju-4 では 4 つ[6] までの無効化要求しか，同時に未完了にはできない．それを越える数の無効化を行ったときには，1 つ目の無効化が完了してバッファが空くのを待つ必要がある．

このバッファの数は，書き込みのレイテンシを隠蔽するのに十分であるとは言い難く，緩和されたメモリ・モデルを採用した意味が薄れてしまう．

更新トランザクションの競合

一方 更新要求の場合には，競合に対して特別の配慮をする必要がなく，実は，プロトコル自体は無効化型より簡単にすることができる．

それは，更新要求パケットには，書き込みの情報がすべて保存されているからである．

したがって更新型では，更新要求パケットを送出後，直ちにバッファを解放してよい．資源の制約によってレイテンシが表面化することはなく，緩和されたメモリ・モデルの利点を十分に引き出すことができる．実際 JUMP-1 では，この方法を採用している．

3.3 トランザクションの処理の順序

パケットの処理の順序に関しては，以下の 2 つの点に注意する必要がある．1 つ目は，デッドロックであり，2 つ目は，平均レイテンシ．前者は当然 解決しなければならないが，後者は二義的である．以下3.3.1 項と 3.3.2 項で，それぞれについて述べる．

3.3.1 デッドロック

あるパケットの処理の結果，別のパケットを送信しなければならない場合，ネットワークへの出口が塞がっていると，そのパケットの処理を進めることはできない．ここで，ただ単に出口が空くのを待つとすると，デッドロックに陥る．

デッドロックへの対処法には，① チャネルの多重化，② 再試行 (retry)，③ 十分長バッファ の 3 つが考えられる．以下，それぞれについて説明する．

① チャネルの多重化

チャネルの多重化は，ネットワークのトポロジとルーティングに起因するデッドロックへの対処法として，広く用いられているている方法である．これは，デッドロックが発生する 4 つの必用条件のうち，循環待ち条件が成立しないようにする方法にあたる．

前節で述べたように，1 つのトランザクションは普通，4 つのフェーズからなる．したがって，ネットワークは 4 本のチャネルを用意する必要がある．

それに加えて，トポロジとルーティングに起因するデッドロックも同時に防止しなければならない．そのために n 本のチャネルが必要だとすると，全体では $4 \times n$ 本ものチャネルが必要になる☆．

DASH は，要求用と応答用の 2 つの個別のネットワークを用意している[7]が，デッドロックを防止するにはそれでは不十分である．

ハードウェア・コストが過大になるため，JUMP-1 もこの方法を採っていない．

② 再試行

再試行は，デッドロックの発生条件のうち，横取り不能条件が成り立たないようにする方法にあたる．

再試行は，実装が容易であるため，採用例が多い．

ただし再試行は，スターベーションを引き起こすので，エイジングと組み合わせなければならない．

しかしスターベーションは，実際には，ほとんど作為的なプログラムでしか発生しないので，発生の可能性を残したまま放置されることも少なくない[7]．

また，プロトコルが非常に複雑になるため，エイジングを実装した大規模な DSM の例は知られていない．

③ 十分長バッファ

システム内に存在可能な要求パケットのすべてを収容できる，十分な長さの受信バッファを用意できれば，資源競合がそもそも発生しないのでデッドロックは起こらない．スターベーションも発生しない．

そのような受信バッファを大容量の 1 個のメモリで実装することは非現実的であるから，普通，次のような方法が採られる；すなわち，バッファ本体と別に少容量の高速なバッファを用意し，それをキャッシュとして用いるのである．デッドロックの危険性が高まった時のみ，パケットをバッファ本体に待避すればよい．

このような方法を採れば，十分長バッファは，ここに挙げた 3 つの方法の中では，最も現実的な解であると結論づけられる．

Cenju-4 もこの方法を採用している．Cenju-4 では，プロセッサあたりたかだか 4 個のトランザクションしか同時に未完了にできないので，バッファ本体は数十 KB あれば十分である．バッファ本体はノードの主記憶上に予め確保され，待避/復帰はハードウェアによって行われる[6]．

JUMP-1 では，同時に未完了にできるトランザクションの数には制限がないので，バッファ本体を予め確保しておくことはできず，待避/復帰をハードウェアのみによって行うことは難しい．

なお，Alewife[8]では，パケット数の上限を定めるこ

☆ それ以下で済む方法は分かっていない．

とができないという理由からではないが，OS がパケットを主記憶に待避する方法を示している．

3.3.2 平均レイテンシ

パケットの処理の中には，以下のように，通常のラインの処理に比べて，最悪何桁も長い時間がかかるものが存在する：

ページ単位の処理 2.2 節で述べたように，3 次キャッシュのリプレースはページ単位で行われる．

マルチキャスト home から renter に無効化/更新要求を送信する際に，性能上の理由から，ネットワークのマルチキャスト機能を使わない方がよいことがある．使わない場合には，renter ごとにパケットをユニキャストする必要がある．

競合 3.2 節で述べたように，競合した場合には，先行する処理が終了するまで後続の処理は待たされる．

高機能アクセス Core のプログラムで対応する，複雑な高機能アクセスが定義されている[1)]．

これらの処理のうち，最後の 1 つは JUMP-1 に固有のものであるが，最初の 2 つは DSM に一般に存在するものである．

これらの処理は，ただ単に時間がかかるというだけで，これまで述べたのとは別の資源要求関係を新たに導入するものではない．したがって，前項で述べた方法によってデッドロックが解決されているなら，後続の処理を待たせてもデッドロックが発生することはない．ただしもちろん，待たせると，平均レイテンシが著しく悪化する可能性がある．

以下では，競合についてもう少しく詳しく述べる．

競　　合

3.2 節で述べたように，競合した場合には，先行する処理 A が終了するまで，後続の処理 B は待たされる．B に時間がかかるのはやむを得ないが，問題は，B の後に届く，競合しないパケット C の処理である．それを待たせてもよい合理的な理由はない．

この問題に対しても，前節で述べた ① チャネルの多重化，② 再試行，③ 十分長バッファ が考えられる．

理想的には，B は待たせ，C を先に処理したいが，そのためにはラインごとに ① チャネルを多重化する必要があり，ハードウェアでそれを行うことは難しい．

そのため DASH などでは，B には ② 再試行 を要求しておいて C の処理を行う方法が採られている[7)]が，やはり，スターベーションの問題が生じる．

Cenju-4 では，スターベーションを嫌って，③ 十分長のバッファ を採用している．しかしこのバッファは FIFO であり，B 以降の（一部例外を除く）すべての要求を待避されるため，C も待たせることになる[6)]．ただし，競合が発生する頻度は高くはないため，現実的な解と言えよう．

4. MBP Core における DSM 管理

本章では，前章での議論を踏まえた上で，MBP Core 上の DSM 管理プログラムについて述べる．まず 4.1 節で，Core とその周辺についてまとめた後，4.2 節以降で，プログラムについて詳しく述べる．

4.1 MBP Core

図 2 に，MBP-light 内のデータ・パスを示す．MBP-light は，MBP Core，MMC の他，**PBR** (Packet Buffer Register) と呼ぶ，特殊なメモリを内蔵する．

MBP Core

MBP Core は，基本的には，ごくシンプルな 16b のスカラー RISC プロセッサである．16 本の 16b 汎用レジスタ(**GPR**) を持ち，ロード/ストア・アーキテクチャを採用する．

記憶階層も単純である．キャッシュ，命令バッファ，ストア・バッファなどは持たず，1 サイクルでアクセス可能な外付けの SRAM を主記憶として動作する．要素プロセッサの主記憶との混同を避けるため，この SRAM は **ローカル・メモリ** と呼ばれる．このような構成のため，ロード/ストアを実行した場合には，命令フェッチが行えず 1 サイクルのストールを伴う．

PBR

PBR には，ネットワーク送受信用と汎用がある．

送受信用 PBR は，その名のとおり，ネットワークに対するパケットの送受信用バッファとして機能する．送信用/受信用 PBR はそれぞれ，8b×8B×8 フリット ×3 パケットの語構成となっている．

一方，汎用 PBR は，スクラッチ・パッド として利用される．送受信用とは異なり，パケットの境界はない．語構成は，8b×8B×64 フリットとなっている．

PBR アクセス

PBR は，GPR ほどではないが，ローカル・メモリよりはずっと小容量である．したがって，そこそこ高速であり，2 ポート (1-read/1-write) 化も可能である．その特性を活かすため，Core からのアクセスには，普通考えられるようなメモリ・マップ I/O ではなく，専用の命令を用いる．

PBR は，容量的にはレジスタに近いが，命令形式上はメモリとして扱われる．PBR 番号は，GPR 番号

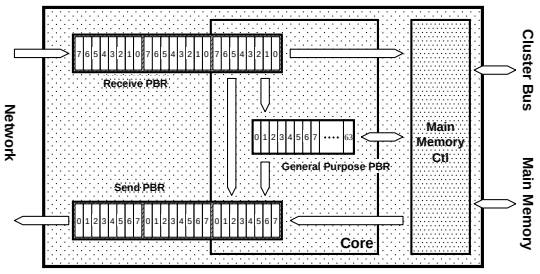

図 2　MBP-light 内部のデータ・パス

のように命令に埋め込まれるのではなく，GPR+4b オフセットによるレジスタ間接で指定される．これは主に，命令の幅が不足しているためである．

命令形式としては，PBR–GPR 間の転送命令の他，PBR–PBR 間の直接転送命令や，CISC のメモリ–レジスタ演算に似た PBR–GPR，PBR–即値演算命令がある．PBR–PBR 間の直接転送命令は，8b，16b に加えて，1 フリット単位での転送をサポートする．

パケットの送受

ネットワークから到着するパケットは受信用 PBR に直接格納され，送信用 PBR に置かれたパケットはネットワークへ直接送出される．

送受信用 PBR は，パケット単位での循環バッファとなっている．受信用の先頭，および，送信用の末尾の 1 パケット分だけが Core の送受信用 PBR のウィンドウに写像されており，Core によるパケットの操作は基本的にはこのウィンドウに対して行われる．ウィンドウのポインタの更新は，Core の命令によって実行される．なお，送受信用 PBR は，循環バッファ —— FIFO であり，チャネルの多重化はなされていないことに注意する必要がある．

一方，クラスタ内部とのパケットのやりとりの方法は，ネットワーク側とは異なる．クラスタ内部に対しては，MMC 内部のバッファと PBR の間でパケットをコピーする必要がある．

この非対称性は，各ユニット間での結合の強さによる．クラスタ間ネットワークとのパケットのやりとりには必ず Core が介在するのに対して，クラスタ・バスからのパケットの多くは MMC によって処理され，Core は関与しない．したがって，MMC 内部は，クラスタ・バスと主記憶間の接続に最適化されており，Core との接続はやや弱くなっている．

特 殊 命 令

MBP Core は，パケットの処理向けの特殊な命令を持つ．特に，PBR のパケット・ヘッダのアドレス部分のハッシュ値を得る命令が効果的である．この命令は，通常の RISC の命令セットなら十命令以上もかかる処理を，1 サイクルで実行することができる．

4.2 MBP Core プログラムの設計方針

MBP Core プログラムの仕事は，基本的には，受信したパケットに対応して，3 次キャッシュと主記憶を更新し，必要であればいくつかのパケットをクラスタの内外に送出することである．ここで，パケットの届いた順に FIFO で処理できるなら問題はないが，前章の議論から，パケットの処理順序には以下の制約があることが分かっている：

無限長バッファ　送信バッファに空きがないために処理が進められない場合には，後続のパケットをバッファ本体に取り込む作業を行う必要がある．
特に，home から renter への無効化/更新要求を複数回のユニキャストによって送信する場合，2 つ目以降のパケットを送信しようとしたらバッファが空いていない，ということがある．その時には，当該パケットの処理を中断する必要がある．

競合　競合に「負け」た場合には,「勝っ」たパケットの処理の終了を待つ必要がある．

平均レイテンシの短縮

(1)　長い時間がかかるパケットの処理は，後続のパケットが届いたときに中断し，後回しにすることが望ましい．また，長いパケットが複数存在するときには，タイマによってそれらを切替えることが望ましい．

(2)　競合が起こった時にも，競合していないパケットを処理できることが望ましい．

これらの要求は，Core プログラムをマルチスレッド化することによって解決できる．すなわち，個々の受信パケットに対してそれを処理するスレッドを生成し，それらのスケジューリングの問題として対処すればよい．その場合 Core プログラムは，通常の OS などと同様に，スレッドの横取り，事象待ちなどをサポートすることになる．

しかし，中断を必要とするものが確かに存在する一方で，ほとんどのパケットは中断の必要はない．しかもそれらは，5 章で述べるように，百サイクル程度の短い時間で終了する．そのような処理をスレッド化すると，そのオーバーヘッドのため著しく性能が悪化する可能性が高い．

そこで，パケットを 2 種類に分けることを考える．すなわち，処理時間の長い，あるいは，中断の必要のあるパケットに対してはスレッドを生成する一方，短いパケットに対しては，スレッドを生成せず，で手続き的に実行するのである．

また，短いパケット間では，高度なスケジューリングを行わず，到着した順序で実行する．3.1 節で述べたように，トランザクションは 4 つのパケットの流れからなる．これらのうち，より後にあるものの優先順位を高めた方が，理論上はレイテンシが短縮される．しかし，そこで横取りなどを行うと，やはりオーバーヘッドのためかえって性能が悪化すると推測される．

4.3 MBP Core プログラムの実装

Core プログラムは，スレッドの横取り，事象待ちなどをサポートするため，通常の OS などと同様の，スレッド・テーブル，レディ・キュー，イベント記述子などのデータ構造を持つ．これらのデータ構造は，当然のことながら，Core の主記憶であるローカル・メモリ上に置かれる．Core プログラムはその他に，**Pending Transaction Table** (**PTT**) と呼ぶ特殊なデータ構造を必要とする．

以下，4.3.1 節で PTT について説明した後，4.3.2 節で短いパケットの処理の方式について述べる．4.3.3 節では，アドレス変換を高速化するソフトウェア TLB について述べる．

4.3.1 PTT

PTT は，クラスタの物理アドレスをキーとするハッシュ・テーブルで，アドレスごとに未完了のトランザクションに関する情報を管理する．PTT の主な目的は，以下のようにして，競合に対応することである．

要求パケットを送信する時には，まずPTT を検索する．同一ラインに対するパケットが既に登録されていれば，競合が発生したことが分かる．その場合，ローカル・メモリ上に領域を確保して当該パケットを待避し，PTT の対応するエントリにキューイングする．

応答が返ってきた時には，対応するエントリを削除する．競合を起こしたパケットがキューイングされていれば，先頭の 1 つを実行可能状態とする．

このように，競合パケットのキューイングはラインごとに行われるので，競合発生後も別のラインに対するパケットを処理することができる．

4.3.2 短いパケットの処理

処理時間の短いパケットは，ちょうど古典的な UNIX のカーネルのように，手続き的に実行する．スレッド記述子やスタックの割り付け，レディ・キューへの登録など，スレッド化のための処理をすべて省略し，その時実行されていたスレッドのスタック上で，割り込みを不許可として中断することなく一気に実行される．

パケットの到着

パケットの到着は割り込みによって知らされる．短いパケットの処理中は割り込みを不許可にするので，割り込みが発生するのは，スレッド実行中である．なお無負荷時には，アイドル・スレッドが実行されている．

MBP Core は，ポーリングによってもパケットの到着を知ることができるが，割り込みの方が性能がよいと考えられる．適切な場所でスレッドが自らポーリングを行うことによって，確かにコンテクストの待避/復帰のコストを削減することができる．しかし，そのような場所を見つけるのは一般には困難であり，レスポンス・タイムが悪化する可能性が高い．割り込みと同等のレスポンス・タイムを実現するためには，割り込み処理と同等の時間の間隔でポーリングを行う必要があり，現実的でない．

また，割り込みの処理には，実際には関数呼び出し程度のコストしかかからない．特に，実行中のスレッドがアイドル・スレッドであった場合には，コンテクストの待避は完全に省略できる．

実行準備

短いパケットの処理では，実行途中での中断を省略するため，実行開始以前に環境を整えておく．具体的には，前述のように競合が起こっていないこと，そして，必要な送信バッファが空いていることを確認する．

送信バッファが空いていない場合には，ローカル・メモリ上に領域を確保してパケットを待避し，送信バッファの空きを表すイベント記述子にキューイングする．このキューが，十分長のバッファとして機能する．

処理の終了

短いパケットの処理は手続き的に実行されるので，終了した後は，直前に走っていたスレッドが再開される．

4.3.3 ソフトウェア TLB

2.2 節で述べたように，パケットをクラスタ外に送信する時には物理から大域仮想へ，受信するときにはその逆の，アドレス変換を行う．そのため，逆引き/正引きのページ・テーブルを検索する必要がある．また，initiator では，home を求めるテーブルの，home では，ディレクトリの検索が必要である．

これらの変換を高速化するため，ソフトウェア TLB を実装する．ローカル・メモリ上のハッシュ表に，主記憶に置かれた各テーブルの内容をキャッシングする．

また，home を求めるテーブルの TLB と逆引きページ・テーブルの TLB，そして，ディレクトリの TLB と PTT は，それぞれ統合することができる．

5. 性能評価

本章では，JUMP-1 DSM の基本的な性能を評価し，その結果から MBP Core の改良に関して考察する．

5.1 評価結果

評価は，JUMP-1 の VHDL 記述をシミュレータ☆にかけて行った．実機の設計ではこの VHDL 記述を論理合成したので，評価結果は実機のものと同一である．

Core プログラムは C で記述した．C コンパイラとしては，gcc-2.8.1 を Core 向けにポーティングした．

クラスタ間のトランザクションのうち最も基本的な，読み出し要求が 3 次キャッシュにミスし home でヒットした場合のサイクル数を求める．トランザクションは，以下の 3 つの部分からなる：

① initiator では，3 次キャッシュ・ミスによって Core に割り込みがかかる．Core は，読み出し要求を home に転送する．

② home の Core は，MMC に依頼して主記憶を読み出す．そのコピーが有効であることを確認し，initiator に応答を返す．

③ 応答パケットを受け取った initiator の Core は，それをクラスタ内に転送する．

なお，要求が到着したとき Core はアイドル・スレッド実行中とした．したがって，コンテクストの切替えの処理は不要である．また，各種ソフトウェア TLB はすべてヒットするものとした．

表 2 に，この処理の所要サイクル数を示す．表中 () 内は，Core プログラム/それ以外のハードウェアの処理時間を表す．全体のレイテンシは 409 サイクル，そのうち Core の処理時間は 235 サイクルとなった．Core の処理の，ハードウェア部分に対する割合，すなわちソフトウェア・オーバーヘッドは 135.0% になる．

☆ メンター・グラフィクス・ジャパン株式会社，qhsim v8.5．

5.2 MBP Core の改良

Core プログラムにおける処理時間の増加の要因としては，以下があげられる：

① Core は 16bit プロセッサであるため，アドレスなどの 16b を越えるデータの扱いに時間がかかる．
② PBR 番号は GPR によるレジスタ間接指定であるため，GPR に PBR 番号を設定する必要がある．
③ パケット・ヘッダのフィールドがバイト境界に整列していない☆ため，余分なロード/ストア，マスク，シフト操作が必要である．
④ ロード/ストア命令は 1 サイクルのストールを伴うため，ローカル・メモリ上の各種データ構造へのアクセスに時間がかかる．

参考までに，これらの点が改善された場合の処理時間を計測してみた．Core の処理は，それぞれによって，① 9，② 16，③ 20，④ 40 サイクル削減され，150 サイクルとなる．全体のレイテンシは 324 サイクル，ソフトウェア・オーバーヘッドは，86.2% となる．

6. おわりに

JUMP-1 DSM では，大容量の 3 次キャッシュによってメモリ・アクセスの平均レイテンシを短縮を図る一方で，ノード間の処理は MBP Core のプログラムによって柔軟に行うというアプローチを採る．

Core プログラムを実装し評価した結果，3 次キャッシュ・ミスにおけるソフトウェア・オーバーヘッドは 135.0% になった．またこれは，自明な改良によって 86.2% まで削減できることが分かった．

このデータは，たとえノード間の処理のすべてをハードウェアで行ったとしても，レイテンシは 1/2 にはならないことを示している．

JUMP-1 の実装は .5〜.4μm のテクノロジの時代のものであり，このデータは，多少古いものとなっている．現在，あるいは，将来のテクノロジを用いれば，ネットワークの遅延時間に対してプロセッサの性能は著しく向上するため，ソフトウェア・オーバーヘッドは更に小さいものとなるだろう．

したがって，DSM におけるノード間の処理をプロセッサを用いてソフトウェアで行うというアプローチは，将来的には，ますます有効になっていくと結論づけることができる．

さて JUMP-1 は，現在，64 プロセッサからなるシステムが稼働している．今後は実機上に実際的なアプリケーションを実装し，評価を行っていく予定である．

処理内容		サイクル数
①	送信バッファの確認	6 (6/ 0)
	MMC からのパケット転送	20 (6/ 14)
	PTT 検索	18 (18/ 0)
	アドレス変換，home の検索	22 (22/ 0)
	パケット種 判別	12 (12/ 0)
	ヘッダ作成	24 (24/ 0)
	小計	102 (88/ 14)
②	送信バッファの確認	6 (6/ 0)
	PTT，ディレクトリ検索	18 (18/ 0)
	アドレス変換	20 (20/ 0)
	パケット種 判別	16 (16/ 0)
	主記憶読み出し	25 (11/ 14)
	ヘッダ作成	20 (20/ 0)
	小計	105 (91/ 14)
③	送信バッファの確認	6 (6/ 0)
	PTT 検索，アドレス変換	23 (23/ 0)
	パケット種 判別	16 (16/ 0)
	ヘッダ作成	10 (10/ 0)
	MMC へのパケット転送	15 (1/ 14)
	小計	70 (56/ 14)
クラスタ内パケット転送		42 (0/ 42)
クラスタ間パケット転送		90 (0/ 90)
合計		409 (235/174)

表 2 読み出し要求処理時間

☆ パケット設計時にはこのような扱いは想定されていなかったため．

謝　辞

JUMP-1 の基本的アイディアは，松本尚氏による．本プロジェクトに携わった方々には，幾多の有益なご助言，ご教示を賜った．また JUMP-1 の開発にあたっては，多くの企業のご賛助を賜った．ここに深甚なる謝意を表したい．

本研究の一部は，文部省科学研究費補助金 重点領域研究 (1)「超並列ハードウェア・アーキテクチャの研究」，および，並列・分散処理研究推進機構による．

参考文献

1) Goshima, M., et al.: The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1, *IWIA'97*, pp. 116–124 (1997).
2) 五島正裕他: Virtual Queue : 超並列計算機向きメッセージ通信機構, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1399–1408 (1996).
3) 佐藤充他: 超並列マシン JUMP-1 のための分散共有メモリ管理プロセッサ, *JSPP'97*, pp. 265–272 (1997).
4) Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *ICPP'88*, pp. 94–101 (1988).
5) 西村克信他: 相互結合網 RDT 上での階層マルチキャストによるメモリコヒーレンシ維持手法, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1367–1377 (1996).
6) 細見岳生他: 並列計算機 Cenju-4 の分散共有メモリ機構, *JSPP'99*, pp. 15–22 (1999).
7) Gupta, A., et al.: Reducing Memory and Traffic Requirement for Scalable Directory-Based Cache Coherence Scheme, *ICPP'90*, pp. 312–321 (1990).
8) Agarwal, A., et al.: The MIT Alewife Machine: Architecture and Performance, *ISCA'95* (1995).