Vol. 41 No. 8 情報処理学会論文誌 Aug. 2000

並列計算機 JUMP-1 の分散共有メモリ・システム

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

JUMP-1 は，物理的構成に合わせたクラスタ構造を持つ分散共有メモリ型並列計算機である．JUMP-1 では，大容量のリモート・アクセス・キャッシュによってメモリ・アクセス・レイテンシの短縮を図る一方で，クラスタ間の処理はプログラム・ベースで行われる．そのプログラムにおいては，クラスタ間で交換されるパケットの処理のスケジューリングの方式が重要である．中断が必要な処理に対してはスレッドを生成し，実行時間の短い処理に対しては，スレッドを生成せず手続き的に実行することで，オーバーヘッドの削減を図った．そのプログラムを実装して評価したところ，リモート・アクセス・キャッシュのミス・ペナルティは 469 サイクルとなった．そのうち，ソフトウェア・オーバーヘッドはハードウェアのレイテンシの 159.1% にあたり，許容範囲であると言える．

Distibuted Shared Memory System of the JUMP-1 Multiprocessor

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

The JUMP-1 multiprocessor has hierarchical, clustered architecture reflecting its physical configuration. While the large remote access cache improves memory access latency, inter-cluster consistency control is performed by a program on a special-purpose processor. In this program, scheduling scheme of packets exchanged among clusters is important. The program creates a thread for a packet which needs to be suspended, while it doesn't create a thread for a short-lived packet and handles it by a procedure call to minimize the scheduling overhead. The evaluation result of the program shows the miss penalty of the remote access cache is 469 cycles, and the ratio of the software overhead to the hardware lantecy is 159.1%. We think the overhead is acceptable.

1. はじめに

計算機ハードウェアは，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．階層的な実装は，アーキテクチャ，特にメモリ・システムの階層化，クラスタリングを促す．

この傾向は，LSI の集積度が向上しても緩和されるものではない．メモリ・ウォールは，以前には，プロセッサ・チップとメモリ・チップの間の速度差として表われていた．LSI の集積度が向上するとそれは，ゲートや短い配線の遅延に対する，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．LSI の微細化が進み配線遅延が支配的なるにつれて，この遅延の差はむしろ拡大していく．大容量のメモリをプロセッサ・コアと同程度の速度で動かすことはますます困難になる．例えば最近のプロセッサでは，1MB 程度の SRAM を集積す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ことが多い．このように，LSI の集積度が向上するにつれメモリ・システムは多階層化される傾向にある．

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，将来的には受け入れ難いものになるであろう．

したがって，アーキテクチャは実装の物理的な階層構造を受け入れ，そのために生じる不具合をソフトウェアによって解消するというアプローチが重要になると考えられる．

そこで，本稿で述べる並列計算機 **JUMP-1** は，階

† 京都大学情報学研究科
Graduate School of Informatics, Kyoto University
☆ 現在，トヨタ自動車株式会社
Presently with Toyota Motor Co. Ltd.
☆☆ 現在，三洋電機株式会社
Presently with SANYO Electric Co. Ltd.

層的なアーキテクチャの実用化研究と，その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成を持つ分散共有メモリ (**DSM**) 型並列計算機である．

JUMP-1 のアーキテクチャの階層性は，プログラムからは主に DSM の階層性として観測される．JUMP-1 では，各クラスタに分散配置された主記憶の一部—例えば半分を，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利用する．クラスタ内の各要素プロセッサは 1 次と 2 次のプライベート・キャッシュを持つため，このキャッシュとして利用される主記憶の一部はクラスタ内のプロセッサによって共有される大容量の 3 次キャッシュとして機能することになる．

この 3 次キャッシュでは，主記憶と同等の大容量によって高いヒット率を見込むことができるため，ミス時のレイテンシに対する要求を低いレベルに抑えることができる．

そこで，クラスタ間の処理をソフトウェアによって行うことを考える．大規模並列計算機のノード間の処理をソフトウェアによって行うことのメリットには，開発コストの削減などの一般的なものの他，処理の高機能化による効率化が挙げられる．例えば，ディレクトリのマップの形式の動的な変更，高機能のアクセスによる通信，同期の効率化[1] などが考えられている．

JUMP-1 では，各クラスタのクラスタ間ネットワークに対するインターフェイス部には，コア・プロセッサ **MBP Core** を内蔵する **MBP-light** (Memory Based Processor light) と呼ぶ専用の LSI が配置される．クラスタ間の処理は，基本的に，MBP Core 上のプログラムによって行われる．

本稿の目的は，この MBP Core 上の DSM 管理プログラムの設計上の問題点を明らかにし，実際に開発されたプログラムの基本性能を評価することにある．

MBP Core プログラムにおける DSM 管理とは，端的に言えば，クラスタの内外から到着した処理要求パケットに対して，3 次キャッシュと主記憶を更新し，必要ならいくつかのパケットを送信することである．したがって，パケットの届いた順序で，すなわち First Come First Service (FCFS) で処理すればよいのなら，考えるべきことは多くはない．

しかし実際には，いくつかの制約のため，必ずしも FCFS で処理できる訳ではなく，より複雑なスケジューリングが必要となる．そのオーバーヘッドが問題となる可能性があるため，処理のスケジューリングの方式について考慮する必要がある．

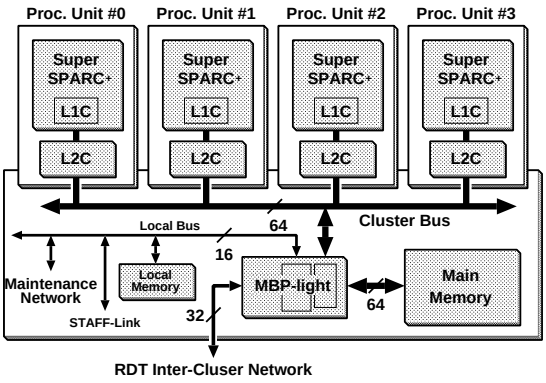


図 1 JUMP-1 のクラスタ
Fig. 1 JUMP-1 Cluster

本稿では，まず 2 章で JUMP-1 について概説した後，3 章で，パケットの処理順序の制約について論じる．そして 4 章で，パケットの処理のスケジューリングの方式に重点をおいて，MBP Core プログラムについて説明する．最後に 5 章で，開発したプログラムの評価結果について述べる．

2. JUMP-1 の概要

本章では，まず 2.1 節で JUMP-1 のアーキテクチャ全体について概説し，2.2 節で MBP Core についてより詳しく述べる．2.3 節では，JUMP-1 の 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．

前章で述べた MBP-light[3)] は，メモリ・ユニットの中央に位置する．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**と呼ばれる I/O 用のバスに接続されている．ローカル・バスには，STAFF-Link[4] 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．MBP-light については，次節で詳しく述べる．

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，粗いトーラスのうちの 1 つの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，ノード数によらず 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いる．その Tree を利用して，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．この機能は，DSM 管理においては，無効化/更新メッセージのマルチキャストと，それに対する応答メッセージ (Ack) の収集に役立つ．

2.2 MBP Core

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 Core の設計方針

MBP Core は，そのために割けるゲート数が限られていたため，面積効率を第一の目標として設計された．ゲートの消費量を抑えるため，基本となるアーキテクチャをシンプルなものとする一方，少ないゲート数で実現できるパケット操作向けの特殊な命令を用意することによって実行サイクル数の削減を図るアプローチを採っている．実際 MBP Core は，ロジックとしては 13,355 ゲートという少ないゲート数で実装されている．

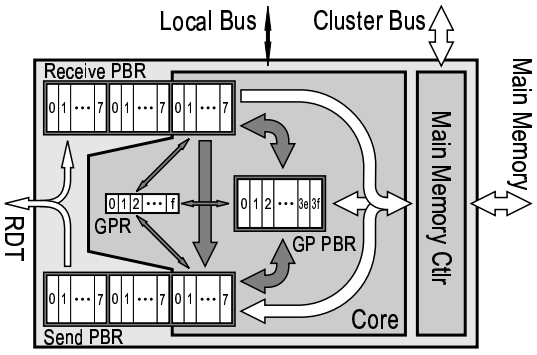


図 2 MBP-light 内部のデータ・パス
Fig. 2 Internal data paths of MBP-light

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．

記憶階層もシンプルである．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 MBP Core の **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/ストア実行時には命令フェッチが行えず，1 サイクルのバブルが発生する．

MBP Core はパケット操作向けの特殊命令を持つ．特殊命令には以下のものがある：

- パケットのバッファに対するアクセス命令
- パケット送受信命令
- その他の特殊命令

これらの命令について，以下で詳しく述べる．

PBR

MBP-light は **PBR** (Packet Buffer Register) と呼ぶ，特殊なメモリを持つ．PBR には，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．

汎用 PBR は，スクラッチ・パッドとして利用する．そのため，送受信用とは違ってパケットの境界はなく，8b×8B×64 フリットの語構成となっている．

PBR アクセス命令

PBR は，GPR ほどではないが，ローカル・メモリよりは小容量である．そのため，2 ポート (1-read/1-write) 化されており，Core からのアクセスには，通常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．すなわち，PBR 番号は，GPR 番号のように命令に埋め込まれるのではなく，GPR+4b 即値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–即値間の演算命令が用意されている．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．

PBR–GPR，PBR–即値演算命令は，パケットのヘッダを一部修正するのに都合がよい．

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．なおパケットのコピーは，専用の命令 1 つによって実行される．

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．

その他の特殊命令

MBP Core は，その他にも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 の命令セットなら数命令かかる処理を，1 サイクルで実行することができる．

2.3 JUMP-1 DSM

本節では，JUMP-1 の DSM について説明する．まず 2.3.1 項でメモリ・モデルについて簡単に触れる．2.3.2 項でアドレス体系について説明する．一貫性制御プロトコルについては，2.3.3 項で述べる．

2.3.1 メモリ・モデル

JUMP-1 のメモリ・モデルは，SuperSPARC+が採用する Partial Store Ordering[6] に準ずる．Partial Store Ordering は，緩和されたメモリ・モデルの一種で，その名のとおり，ストアの実行が半順序関係になることを許す．

JUMP-1 では，このモデルの特長を活かすため，複数のストアを同時に未完了することができるように実装されている．

2.3.2 アドレス体系

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 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，キャッシュのパラメータを示す．

Shared Virtual Memory

3 次キャッシュのアドレス体系は，**Shared Virtual Memory (SVM)**[7] 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 を用いると，クラスタ内では，通常の仮想記憶のシステムをそのまま流用することができる．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スを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，それぞれ必要となる．

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

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
Table 1 Cache parameters

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．ライン単位の一貫性制御の方式については，次節以降で詳しく述べる．

2.3.3 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，クラスタ間の一貫性制御は標準的な分散ディレクトリ方式に基づいて行われる．

ディレクトリ方式

ディレクトリのマップの形式は，MBP 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[8]．

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]．

共有状態の管理

ラインの状態は，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，global shared の 3 段階の共有状態を持つ．

また，コピーの存在範囲に合わせて，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・コントローラは，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最小化する．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，真にクラスタ間の処理が必要である場合に限定される．

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 FCFS 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，3.3 節でトランザクションの処理順序について論じる．

3.1 トランザクションの流れ

トランザクションに関わるクラスタは，以下のように分類される．既に述べたように，ラインのディレクトリを持つクラスタを home という．システム内で唯一の有効な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，要求パケットがまず 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，コピーの状態によっては，② と ③ は省略できる．

また，JUMP-1 では，パケットの流れは MBP Core によって制御されるのでプログラマブルであるが，本稿では簡単のため，三角通信などを考えず，上記のように，一旦 home を経由するものを考える．

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 initiator 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，① 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．

競合が発生した場合，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方法を採ると処理が複雑になるので，普通トランザクションの逐次化を行う．すなわち，home において，未応答のトランザクションを記録することによって競合を検出し，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，特に，無効化トランザクションの「相撃ち」への対応が難しい．

競合に「負け」た場合には，当該ラインへに対する ④ home からの応答が返る前に，別のクラスタが開始した「勝った」トランザクションの ② home からの無効化要求が届く．この場合，一旦その無効化要求を受け入れて，自コピーを無効化する必要がある．そうしないと，デッドロックに陥る．

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,「勝っ」たトランザクションを受け入れた際に，自分が行った書き込みの内容が失われてしまうからである．

「相撃ち」には，再試行によって対応するのが普通であろう．すなわち，書き込みを再試行できるだけの情報をライト・バッファに保存しておき,「負け」た場合には,「勝っ」た側の無効化要求を受け入れて自コピーを無効化した後に，再び書き込みからやり直すのである．

さて,「負け」なかったことを通知するためにわざわざ専用のパケットを用いないとすると，それが判明するのは 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．ライト・バッファは，その時まで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，ライト・バッファの数による制約が生じる．JUMP-1 では各プロセッサごとに 3 つ，Cenju-4 では 4 つ[9]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ライト・バッファが空くのを待つ必要があり，書き込みのレイテンシが表面化することになる．このライト・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・モデルの効果が薄れてしまう．

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，無効化型のような特別の配慮をする必要がなく，プロトコル自体は無効化型より簡単に設計することができる．ネットワークが FIFO 性を備えている場合には，トランザクションの逐次化を行う必要すらない．

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☆.

更新型プロトコルの場合には，更新要求パケットが ② home から renter に転送される際に，initiator にも送り返す必要がある．これは競合が発生した場合にもメモリの最終状態を逐次的な状態にするためであり，競合に「負け」た場合には，自分の送出した更新要求パケットが ② home から戻ってきたときに，そのパケット内の情報から自コピーを正しく更新することができる．

したがって更新型では，更新要求パケット送出後，直ちにライト・バッファを解放してよい．その場合，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 の 2 次キャッシュ・コントローラは，更新要求の送出後は直ちにバッファを解放する方法を採っており，実際上無制限の更新要求を同時に未完了にすることができるように設計されている

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する問題には，デッドロックとレイテンシがある．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がある．その場合，ネットワークへの出口が塞がっていると，処理を進めることができない．ここでただ単に出口が空くのを待つと，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) 十分長のバッファ の 3 つが考えられる．以下，それぞれについて説明する．

☆ したがって，更新パケットによって無効化を実現する方法が考えられる．

(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．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には 4 本のチャネルを用意すれば，デッドロックを防止することができる．

トランザクションに起因するデッドロックに加えて，トポロジに起因するデッドロックの防止も同時に考慮しなければならない．トポロジのために n 本のチャネルが必要だとすると，最悪 $4 \times n$ 本ものチャネルが必要になる．それ以下で済む方法は知られていない．

DASH では，それぞれがデッドロック・フリーである 2 つの個別のネットワークを用意し，それぞれを要求用と応答用にあてるという構成を持つ[10]が，それでもデッドロックを防止するには不十分である．

(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．

前節で述べたように，無効化型プロトコルでは，無効化の「相撃ち」に対処するため送信後もライト・バッファを解放することができない．また，読み出しに対しては，読み出しの情報を保持しておくリード・バッファが必須である．したがって無効化型プロトコルを採用する計算機では，比較的安価に再試行を実装することができるため，採用例が多い．

しかし再試行には，以下の 2 つの問題がある．1 つはスターベーションであり，もう 1 つは前節で述べたのと同様の資源制約の問題である．

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

また再試行を行うためには，当然のことながら，再試行を行えるだけの情報を送信側に残しておく必要がある．これは，前節で述べた無制限の要求を同時に未完了にできるという更新要求の利点を消してしまう．

(3) 十分長のバッファ

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．もちろん，スターベーションも発生しない．

Cenju-4 は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[9]．

なお，Alewife[11]では，OS がパケットを主記憶に待避する方法を示している．

JUMP-1 におけるデッドロックの対処法

JUMP-1 では，以下に述べる理由により，(1) チャネルの多重化 と (2) 再試行 を採用することができない：

(1) チャネルの多重化
ハードウェア・コストが過大になるため，ネットワークは必要なチャネルを備えていない．

(2) 再試行
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る必要があるが，プロトコルが非常に複雑になる．
再試行を行うための情報を送信側に残しておく必要があり，同時に未完了にできる要求の数に制約がないという更新要求の利点を消してしまう．

したがって JUMP-1 では，Cenju-4 と同様，(3) 十分長のバッファ を採用する．

ただし 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ない．そのため，待避/復帰をハードウェアのみによって行うことは難しく，MBP Core のプログラムによって対応する必要がある．

3.3.2 レイテンシ

5 章で述べるが，通常のラインの処理は 100 サイクル程度で終了する．しかし，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：

(1) 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ず，renter ごとにユニキャストした方がよい場合がある．

(2) 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の終了まで後続の処理は待たされる．

(3) ページ単位の処理 2.3.2 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．

(4) 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]．

これらのうち，(3) と (4) は，JUMP-1 に特有であるが，(1) と (2) 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，その観点からは到着順に処理すればよい．

ただし，平均レイテンシを短縮するため，できればShort-Job-First で処理することが望ましい．すなわち，長い時間がかかる処理の最中に後続の要求パケットが到着した場合には，長い時間がかかる処理を中断し，後続のパケットを先に処理する方がよい．

なお，(2) 競合 に対しては，その対処方法によってはレイテンシ以外の問題が発生することがあるため，注意が必要である．以下では，それについてもう少し詳しく述べる．

競合とレイテンシ

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．

そのためDASH などでは，B には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[10]が，やはり，スターベーションが発生する．

またCenju-4 は，スターベーションを防止するため，前項で述べたのと同様の十分長のバッファを用意して，B を含む（一部例外を除く）すべての要求を待避する[9]．しかしこのバッファは FIFO であるため，ネットワークの混雑を緩和する効果があるものの，レイテンシの改善に対しては直接的な効果はない．

処理順序とマルチ・プログラミング環境

また，これらの長い時間がかかる処理の順序は，マルチ・プログラミング環境で特に問題となる．FCFS で処理した場合，競合や 3 次キャッシュのリプレースを頻繁に引き起こすような悪質なプログラムによって，別のプログラムの処理速度，ひいては，システム全体の性能が著しく低下する危険性がある．長い時間がかかる処理を後回しにすることには，この問題を緩和する効果がある．

4. MBP Core プログラム

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，プログラムの設計方針についてまとめた後，4.2 節でその実装について述べる．

4.1 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 FCFS 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：

十分長バッファ 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（一定時間待って）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を試みたときに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」たトランザクションの終了を待つ必要がある．

レイテンシの短縮 以下が可能であることが望ましい：

(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時に中断し，後回しにする．また，長い時間がかかる処理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．

(2) 競合が起こった時にも，競合していない後続のパケットを処理する．

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，中断を必要とする処理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してしまう．

また，短い時間で終了するパケットの処理の間では，高度なスケジューリングを行わず，FCFS で実行する方がよいと推測される．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，送

信バッファの塞がる確率が減り，理論上はレイテンシが短縮される．しかし，そのために横取りなどを行うと，やはりオーバーヘッドが大きくなり，かえって性能が悪化する可能性が高い．

そこで MBP Core プログラムでは，以下のようにスケジューリングを実装する．

まず，パケットを 2 種類に分ける．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，処理時間が短く，かつ，中断することなく処理可能なパケットに対しては，スレッドを生成せず，手続き的に実行するのである．以降では，前者を **Long-lived パケット**，後者を **Short-lived パケット** と呼ぶことにする．

Long-lived パケットに対するスレッド間では通常の OS と同様，Round-Robin などでスケジューリングを行い，Long-lived パケットと Short-lived パケットの間では Short-Job-First でスケジューリングを行う．すなわち，Long-lived パケットの処理の実行中に Short-lived パケットが到着した場合には，Long-lived パケットの処理を中断し，Short-lived パケットの処理を先に実行する．

Short-lived パケットの処理は，ちょうど古典的な UNIX のカーネルのように，手続き的に実行すればよい．Short-lived パケットの処理は，パケット到着時に実行されていた Long-lived パケットの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する．なお無負荷時には，アイドル・スレッドが実行されているものとする．このことによって，Short-lived パケットに対しては，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することができる．

4.2 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．また，アドレス変換を高速化するために，ソフトウェア TLB を用意する．

以下，4.2.1 節と 4.2.2 節でで PTT とソフトウェア TLB について説明し，最後に 4.2.3 節で Short-lived パケットの処理についてまとめる．

4.2.1 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 の役割は，以下のようにして競合に対応することである．

home が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．

PTT では，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，競合発生時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．

4.2.2 ソフトウェア TLB

2.3.2 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．なお，2.2で述べたように，MBP Core では，アドレスのハッシュ値は 1 命令で得ることができる．

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．

4.2.3 Short-lived パケットの処理

Short-lived パケットの処理は，以下のように進められる：

(1) 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．Short-lived 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 Long-lived パケットに対するスレッド，または，アイドル・スレッドの実行中である．

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が，そのような場所を見つけるのは一般には困難である．適切な場所がない場合には，レスポンス・タイムが悪化する．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/復帰も省略することができる．

(2) 実行準備

Short-lived パケットの処理では，実行途中での中断を省略するため，実行開始以前に実行のための環境を整えておく必要がある．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，(一定時間待った上で) 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．

(3) 処理の終了

パケット処理が終了した後は，新たなパケット到着をチェックする．パケットが到着していればその処理に移行し，していなければ割り込み時に実行されていた Long-lived パケットのスレッドが再開される．

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 評価環境

評価は，実機の設計時に用いた VHDL 記述のシミュレーションによって行った．この VHDL 記述では，JUMP-1 のために設計された LSI—2 次キャッシュ・コントローラ，RDT ルータ，そして，MBP-light の部分は，それを論理合成して実機の LSI を得たものである．その他の汎用の LSI に関しては，仕様を基にサイクル・レベルでの動作が実物と一致するように記述したものである．したがって，以下で述べる評価結果は実機のものと同一である．

Core プログラムは，基本的には C 言語を，性能上クリティカルな部分に関してはアセンブリ言語を用いて記述した．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ものを用いた．

5.2 基本的トランザクションのレイテンシ

クラスタ間の読み出し要求と無効化要求に対して，そのレイテンシを計測した．その結果を**表 2** に示す．

表中 () 内は，Core プログラム/それ以外のハードウェア の処理サイクル数を表す．

①〜④は，3.1で述べたトランザクションの 4 つのフェイズを示す．例えば，①→② は，要求パケットが ① home に届いてから，② renter に送られる までの MBP Core の処理を表す．

読み出し/無効化の各 2 列のうち，左側の列は ② と ③ が省略できた場合，右側はできなかった場合である．

クラスタ内 HW とは，要素プロセッサの命令パイプラインがロード/ストア命令を実行してから MBP Core に割り込みがかかるまでの時間と，MBP Core が応答パケットの転送を MMC に依頼してから要素プロセッサの命令パイプラインが再び動き出すまでの時間の合計である．

クラスタ間 HW とは，クラスタ間ネットワーク RDT におけるパケットの転送時間を表す．

要求パケットが到着したとき 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．

5.3 home に対する読み出しトランザクションのレイテンシ

本節以降では，表 2 中 第 1 列に示した，読み出し要求において ② と ③ が省略できた場合，すなわち，読み出し要求が home でヒットした場合について述べる．

まず，このトランザクションのレイテンシの内訳を，**表 3** に示す．表に示すように，このトランザクションのレイテンシは 469 サイクル，50MHz 動作時には約 9.38μsec となる．このうち Core の処理時間は 288 サイクルで，ソフトウェア・オーバーヘッド，すなわち，Core の処理時間のそれ以外のハードウェア部分に対する割合は 159.1%となる．

5.4 高速化技術の評価

本節では，ソフトウェア TLB と，Short-lived パケットに対してスレッドを生成しないことの効果について述べる．

ソフトウェア TLB の効果 TLB を用いない場合，処理 →① と ①→④ において，それぞれ以下のサイクルが余分に必要となる：

- →①：
  - 物理から大域仮想へのアドレス変換 . 64 サイクル
  - home の検索 . . . . . 43 サイクル
- ①→④：
  - 大域仮想から物理へのアドレス変換 153 サイクル
  - ディレクトリの検索 . . . . . 43 サイクル

TLB にヒットした場合には，処理 →① と ①→④

	読み出し		無効化	
→①	107 (93/ 14)	107 (93/ 14)	107 (93/ 14)	107 (93/ 14)
①→②	— (—/ —)	119 (119/ 0)	— (—/ —)	108 (108/ 0)
②→③	— (—/ —)	67 (50/ 17)	— (—/ —)	104 (104/ 0)
③→④	148 (131/ 17)	94 (80/ 14)	120 (106/ 14)	147 (130/ 17)
④→	81 (64/ 17)	116 (116/ 0)	81 (64/ 17)	81 (64/ 17)
クラスタ内HW	45 (0/ 45)	90 (0/ 90)	45 (0/ 45)	90 (0/ 90)
クラスタ間HW	88 (0/ 88)	176 (176/176)	88 (0/ 88)	176 (0/176)
合計	469 (288/181)	850 (52/328)	441 (263/178)	813 (499/314)

表 2 基本的トランザクションのレイテンシ
Table 2 Latency of basic transactions

のそれぞれにおいて，34 サイクルで処理することができる．トランザクション全体では，235 サイクルの高速化が達成されている．

スレッド生成省略の効果 処理 →①，①→④，および，④→ のそれぞれに対してスレッドを生成した場合，home に対する読み出しトランザクションのレイテンシは 614 サイクルとなる．スレッドの生成を省略することによって，145 サイクルの高速化が達成されている．

5.5 MBP Core の評価

本節では，汎用の RISC プロセッサをコアとして用いる場合と比較することによって，MBP Core のアーキテクチャの評価を行う．前項までと同様，home に対する読み出しトランザクションのレイテンシを比較する．

処理内容		サイクル数
→①	送信バッファの確認	8 (8/ 0)
	MMC からのパケット転送	24 (10/ 14)
	アドレス変換，home の検索	34 (34/ 0)
	パケット種 判別	10 (10/ 0)
	ヘッダ作成	31 (31/ 0)
	小計	107 (93/ 14)
①→④	送信バッファの確認	13 (13/ 0)
	PTT，ディレクトリ検索	34 (34/ 0)
	アドレス変換	29 (29/ 0)
	パケット種 判別	10 (10/ 0)
	主記憶読み出し	27 (10/ 17)
	ヘッダ作成	35 (35/ 0)
	小計	148 (131/ 17)
④→	送信バッファの確認	8 (8/ 0)
	PTT 検索，アドレス変換	34 (34/ 0)
	パケット種 判別	10 (10/ 0)
	ヘッダ作成，送信	10 (10/ 0)
	MMC へのパケット転送	19 (2/ 17)
	小計	81 (64/ 17)
クラスタ内HW		45 (0/ 45)
クラスタ間HW		90 (0/ 90)
合計		469 (288/181)

表 3 読み出し要求処理のレイテンシ
Table 3 Latency to service a read request

比較にあたってまず，パケット・ヘッダの取扱に関する本質的でないオーバーヘッドを除外することを考える．クラスタ・バスのプロトコル設計時には，プロセッサによる扱いは想定されていなかったため，パケット・ヘッダの各フィールドはバイト境界に整列していない．そのために，MBP Core がパケット・ヘッダのフィールドを扱う際には，余分なロード/ストア，マスク，シフト操作が必要となっている．また，MBP Core ではなく，汎用 RISC プロセッサを用いても同様の操作は必要になる．このオーバーヘッドは，プロトコルの設計を注意深く行えば除外できるものであり，プロセッサの比較をする上では本質的ではない．したがって以下ではこれを除外する．フィールドがバイト境界に整列した MBP Core のモデルを **MBP Core+** と呼ぶことにする．このことによって home に対する読み出しトランザクションのレイテンシは 20 サイクル短縮され，449 サイクルとなる．

比較対象として，以下のようなモデルを考える．命令/データ・キャッシュを持つ標準的な 32b スカラー RISC プロセッサをコアとして用いる．MBP Core の PBR に相当するパケット・バッファは，データ・キャッシュと並列に置かれ，ロード/ストア命令によってデータ・キャッシュと同じ速度でアクセスできるものとする．また，パケット・バッファ，データ・キャッシュとのデータ・パスは 64b あり，倍長語のロード/ストア命令が 1 サイクルで実行できるものとする．以下このモデルを **RISC** と呼ぶ．

RISC に対する MBP Core+アーキテクチャの得失は，以下のようにまとめられる：

良い点

PBR RISC では，ロード/ストア命令によってパケット・バッファにアクセスするため，PBR–PBR 間転送命令はロード + ストアの 2 命令で，PBR–GPR，PBR–即値間の演算命令はロード + 演算+ ストアの 3 命令で実行することになる．

しかし，PBR–GPR，PBR–即値間の演算命令は，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場合のマスク操作に専ら用いられており，Core+では全く用いられなかった．

トランザクション全体では，PBR–PBR 間転送命令を 9 回実行し，9 サイクルしか短縮されない．

ハッシュ RISC の命令セットでハッシュ値を求めるには 7 サイクルかかり，1 回につき 6 サイクル短縮される．

トランザクション全体では，4 回分，24 サイクル短縮される．

悪い点

16b アーキテクチャ Core は 16b 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．

トランザクション全体では，30 サイクル悪化する．

I/D 非分離 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アクセスを行うと，速度が低下する．

トランザクション全体では，25 サイクル悪化する．

RISC モデルの home に対する読み出しトランザクションのレイテンシは，Core+モデルより $(30+25)-(9+24)=22$ サイクル短縮され，427 サイクルとなる．Core のアーキテクチャは，RISC には及ばないものの，当初の目標どおり，ゲート数の少なさを特殊機能によってある程度カバーしているといえる．

最後に，MBP Core+アーキテクチャを，32b 化，I/D 分離化したモデルを考える．このモデルの home に対する読み出しトランザクションのレイテンシは，RISC モデルに対して $9+24=32$ サイクル短縮され，394 サイクルとなる．この場合，ハードウェアのレイテンシに対するソフトウェア・オーバーヘッドの割合は 117.7%となる．

トランザクション全体に対するこれらのモデルの差は大きくはない．現在なら，IP コアなどを利用することによって比較的容易に RISC モデルを実現することができ，特殊なアーキテクチャを新規開発する意味は薄くなっている．

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の短縮を図る一方で，クラスタ間の処理は 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，ハードウェアのレイテンシに対するソフトウェア・オーバーヘッドの割合は 159.1%になった．また，Core を 32b 化，I/D 分離化することによって 117.7%まで削減できることが分かった．

このデータは，ノード間処理をすべてハードウェアで行い，ソフトウェアで処理していた部分の処理時間がハードウェア化することでたとえ 0 になると仮定しても，レイテンシは 1/2 程度にしかならないことを示している．ソフトウェア・オーバーヘッドは，決して十分に小さいとは言えないが，許容できる範囲にあると言える．

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，今となって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処理速度が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，将来的に有効になっていくと結論づけることができる．

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．

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．

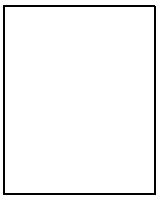
また本研究の一部は，文部省科学研究費補助金 重点領域研究 (1)「超並列ハードウェア・アーキテクチャの研究」，試験研究 (A)「超並列計算機のプロトタイプ開発と設計」，基盤研究 (B)(2) #12480072，同 #12558027，および，並列・分散処理研究推進機構，カテゴリー 1，グループ 3 による．

参考文献

1) Goshima, M., Mori, S., Nakashima, H. and Tomita, S.: The Intelligent Cache Controller of a Massively Parallel Processor JUMP-1, *IWIA'97, Int'l Workshop on Innovative Architecture for Future Generation High-*

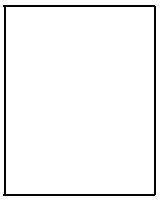
Performance Processors and Systems, pp. 116–124 (1997).
2) 五島正裕, 森眞一郎, 富田眞治: Virtual Queue : 超並列計算機向きメッセージ通信機構, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1399–1408 (1996).
3) 佐藤充, 天野英治, 安生健一郎, 周東福強, 西宏章, 工藤知宏, 山本淳二, 平木敬: 超並列マシン JUMP-1 のための分散共有メモリ管理プロセッサ, *JSPP'97*, pp. 265–272 (1997).
4) 中條拓伯, 大谷智, 小畑正貴, 金田悠紀夫: 超並列計算機 JUMP-1 の入出力サブシステムにおける I/O ネットワーク, 情報処理学会論文誌, Vol. 39, No. 6, pp. 1801–1808 (1998).
5) 楊愚魯, 天野英晴, 柴村英智, 末吉敏則: 超並列計算機向き結合網: RDT, 電子情報通信学会論文誌, Vol. J78-D-I, No. 2, pp. 118–128 (1995).
6) SPARC International, inc.: *The SPARC Architecture Manual Version 8* (1992).
7) Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *ICPP'88*, pp. 94–101 (1988).
8) 西村克信, 工藤知宏, 西宏章, 楊愚魯, 天野英晴: 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).
9) 細見岳生, 加納健, 中村真章, 広瀬哲也, 中田登志之: 並列計算機 Cenju-4 の分散共有メモリ機構, *JSPP'99*, pp. 15–22 (1999).
10) Gupta, A., Weber, W. D. and Mowry, T.: Reducing Memory and Traffic Requirement for Scalable Directory-Based Cache Coherence Scheme, *ICPP'90*, pp. 312–321 (1990).
11) Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J. D., Lim, B.-H., Mackenzie, K. and Yeung, D.: The MIT Alewife Machine: Architecture and Performance, *ISCA'95* (1995).

(平成 12 年 5 月 12 日受付)
(平成 12 年 8 月 29 日採録)



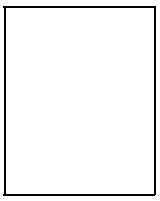
五島　正裕（正会員）

1968 年生. 1992 年 京都大学工学部情報工学科卒業. 1994 年 同大学大学院工学研究科情報工学専攻修士課程修了. 同年より日本学術振興会特別研究員. 1996 年京都大学大学院工学研究科情報工学専攻博士後期課程退学, 同年より同大学工学部助手. 1998 年 同大学大学院情報学研究科助手. 高性能計算機システムの研究に従事.



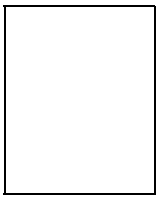
斎藤　康二

1977 年生. 1996 年 京都大学工学部情報工学科入学. 2000 年 同学卒業, 同年から三洋電機株式会社に勤務.



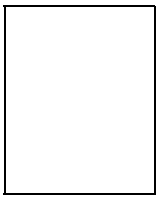
小西　将人

1976 年生. 1992 年 綾部市立八田中学卒業. 1995 年 京都府立綾部高校卒業. 1999 年 京都大学工学部情報学科卒業. 同年より同大学大学院情報学研究科修士課程に進学. 並列計算機アーキテクチャの研究に従事.



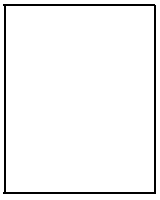
秤谷　雅史（学生会員）

1975 年生. 1991 年 枚方市立招提中学卒業. 1994 年大阪府立四条畷高校卒業. 1998 年 京都大学工学部情報学科卒業. 2000 年 同大学大学院情報学研究科修士課程修了, 同年からトヨタ自動車株式会社に勤務.



森　眞一郎（正会員）

1963 年生. 1987 年 熊本大学工学部電子工学科卒業. 1989 年 九州大学大学院総合理工学研究科情報システム学専攻修士課程修了. 1992 年 九州大学大学院総合理工学研究科情報システム学専攻博士課程単位取得退学. 同年京都大学工学部助手. 1995 年 同助教授. 1998 年 同大学大学院情報学研究科助教授. 工学博士. 並列/分散処理, 計算機アーキテクチャの研究に従事. IEEE, ACM 各会員.



富田　眞治（正会員）

1945 年生. 1973 年 京都大学大学院博士課程修了, 工学博士. 同年京都大学工学部情報工学教室助手. 1978 年 同助教授. 1986 年 九州大学大学院総合理工学研究科教授. 1981 年 京都大学工学部情報工学科教授. 1998 年 同大学大学院情報学研究科教授. 計算機アーキテクチャ, 並列計算機システムに興味を持つ. 著書「並列計算機構成論」,「並列処理マシン」,「コンピュータアーキテクチャ I」など. 電子情報通信学会, IEEE, ACM 各会員.