並列計算機 JUMP-1 における
分散共有メモリ管理プログラム

小西 将人

平成 13 年 2 月 9 日

内容梗概

並列計算機 JUMP-1 は物理的構成にあ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 ではメモリシステムも階層化されており、主記憶レベルの DRAM をリモートクラスタの主記憶に対する 3 次キャッシュとして利用する。この 3 次キャッシュでは、そ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に、ミス時のレイテンシに対する要求は低い。そこで JUMP-1 では、3 次キャッシュヒットはハードウェアで高速に処理し、ミスヒット時のクラスタ間処理を専用プロセッサ上で動作するソフトウェアで柔軟に処理を行うアプローチを採る。本稿ではこの DSM 管理プログラムの実装について述べる。

クラスタ間における DSM 管理では、デッドロックの防止や平均レイテンシの短複雑なスケジューリングを行うことが必要となる。

DSM 管理プログラムは、基本的には到着したパケットに対してその処理を行うスレッドを生成し、パケット処理順序の問題をスレッド間のスケジューリングの問題として解決する。しかし、到着する全てのパケットに対してスレッドを生成すると、そのオーバーヘッドが問題となる。そこで、到着した時点で処理が可能なパケットに対してはスレッドを生成せず、手続き的に処理を行うことでオーバーヘッドの削減を図った。

実機上で、実装した DSM 管理プログラムを動作させ 3 次キャッシュミス時のレイテンシは 508 サイクルになるとの結果を得た。この処理におけるソフトウェアオーバーヘッドは 156.6%、Core の 32b 化、I/D 分離化により 105.0%まで削減できることがわかった。またシステム全体の平均レイテンシを計算したところ、クラスタ間処理を全てハードウェアで処理を行うと仮定しても、4〜6%程度の改善しか図れず、ソフトウェアでクラスタ間処理を行うことは有効であるとの結論を得た。

Abstract

The JUMP-1 Multiprocessor has hierarchical, clustered architecture reflecting its physical configuration.

JUMP-1, which has a hierarchical memory system, uses a part of the main memory as the third level cache for remote memory. At this third cache a high hit ratio is expected for its large size, so low latency is not strictly required when miss-hit happens. Therefore, JUMP-1 takes a approach that while inner-cluster consistency controll is performed by hardware rapidly, inter-cluster controll is performed by a program on a special-purpose processor. In this report, we descibe the implementation of this Distributed Shared Memory(DSM) management program.

To avoid deadlock and reduce average latency,this inter-cluster DSM management program is required to schedule operations for arrived packets. This schedule is complex.

To satisfy this request,the program fundamentally creates a thread for an arrived packet and schdules threads. However,overhead of creating all arrived packet is problem. To reduce this overhead, we implement the program that doesn't create a thread for a packet that can be operated at the time it arrives.

We make this implemented DSM management program run on a real JUMP-1 system to measure that the latency of remote main memory read operations is 508 cycles. And that in this opration, software overhead is 156.6% and this overhead can be reduced to 105.0% if the processor is 32bit architecture with instruction cache apart from data cache. And by calculating the average memory access latency with such results, we know that using hard-wired inter-cluster DSM

controller improves at only 4-6% as compared with software control. So we conclude that it is available that inter-cluster DSM control is performed by software.

目 次

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。

本稿では、このMBP Core上のDSM管理プログラムについて述べる。

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 Served (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。

本稿では、まず2でJUMP-1について概説した後、3で、DSM管理におけるパケットの処理順序の制約について論じる。そして4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、DSM管理プログラムについて説明する。最後に5で、開発したプログラムの評価結果について述べる。

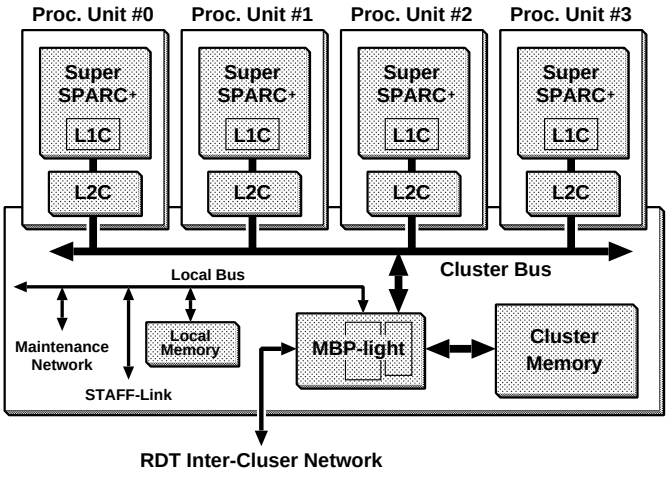

図1: JUMP-1のクラスタ

2 JUMP-1の概要

本章では，まず2.1節でJUMP-1全体の構成を説明し，2.2節でJUMP-1 DSMの概要を述べる。最後に2.3節でJUMP-1 DSMの中心的な役割を果たすMBP-lightについて述べる。

2.1 物理的構成

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。複数クラスタをクラスタ間ネットワークで接続することで、システム全体を構成する。

図1にクラスタの構成を示す。

1つのクラスタは、主に4つのプロセッサユニットと、メモリユニットから成る。4つのプロセッサユニットとメモリユニットは、クラスタバスと呼ばれる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、1次キャッシュ内蔵の要素プロセッサSuperSPARC+と、外部2次キャッシュ及び2次キャッシュコントローラ[1]から成る。各キャッシュのパラメータを表2.1に示す。

メモリユニットは主記憶であるクラスタメモリと、MBP-light[2]からなる。

MBP-lightは主に4つのポートを持ち、それぞれ、ク

ラスタバス、クラスタメモリ，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[3]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 は主に、前章で述べたクラスタ間処理を行うコアプロセッサ MBP Core と、クラスタメモリの制御を行う **MMC**(Main Memory Controller) から構成される。MBP-light については、2.3節で詳述する。

クラスタメモリは 16MByte の容量を持つ。クラスタメモリは、各クラスタに分散された主記憶であるとともに、その一部をリモートクラスタに対するキャッシュとして利用する。クラスタメモリには、キャッシュライン (32Byte) 単位に SRAM で構成された有効性を示すタグが設けられている。このキャッシュについては 2.2節で詳述する。

クラスタ間ネットワークは、**RDT(Recursive Diagonal Torus)**[4] 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 4 段まで再帰的に詰み上げた、やはり階層的な構造を持つ。

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 (以降、これを Ack と呼ぶ) の収集に役立つ。

最後にクラスタの実物の写真を図 2に示す。

中央より下半分がクラスタボード、上半分が 4 枚のプロセッサカードである。クラスタボードは下部のコネクタによって RDT ネットワークであるバックプレーンに接続される。SuperSPARC+は上方部に位置するヒートシンクが装着されたチップである。その下の正方形のチップが 2 次キャッシュとキャッシュコントローラである。

表 1: キャッシュのパラメータ

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

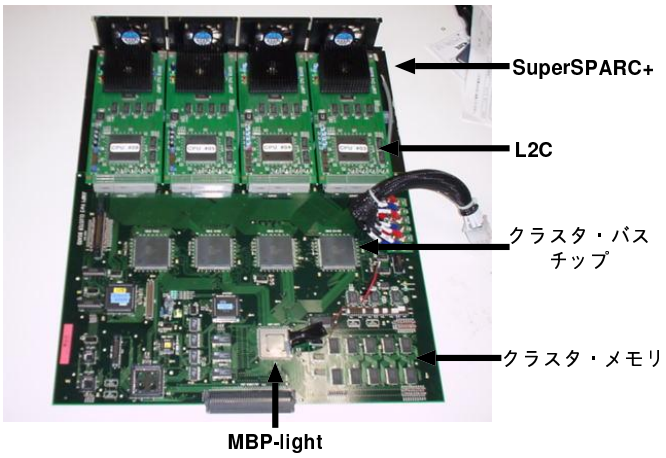

図 2: クラスタの写真

2.2 JUMP-1 DSM

JUMP-1 では、クラスタメモリの一部、例えば半分程度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ことで、平均メモリアクセスレイテンシの削減を図る。要素プロセッサは、SuperSPARC+内蔵 1 次キャッシュと、外部 2 次キャッシュを持つことから、このキャッシュは 4 つの要素プロセッサで共有される 3 次キャッシュにあたる。

この大容量 3 次キャッシュでは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く抑えることができる。

JUMP-1 におけるキャッシュの一貫性制御は、クラスタ内に関してはハードウェアで高速で行う一方、クラスタ間では MBP Core 上で動作するソフトウェアで柔軟に行う方式を採っている。

以下、2.2.1項でアドレス体系についてまとめた後、2.2.2項でキャッシュ一貫性制御について詳述する。

2.2.1 アドレス体系

3 次キャッシュのアドレス体系は SVM(Shared Virtual Memory)[5] に準ずる。SVM ではキャッシングはページ単位でなされる。リモートクラスタにあるオリジナルページをキャッシングするには、自身のクラスタメモリ上の 3 次キャシュ領域にページ枠を確保し、オリジナルページをコピーする。SuperSPARC+の MMU では、ページサイズとして 4KB、256KB、4MB を利用することができる。図 3にキャッシングの様子を示す。

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いページ枠を確保し、ページテーブルの設定を行う。この時ページ枠を確保するのみか、データを全てコピーするかは戦略による。以降のアクセスはページフォルトにはならず、通常の一貫性制御を通して仮想ページへのアクセスがなされる。

クラスタ内では、通常の仮想記憶システムを用いることができる。要素プロセッサ内の仮想アドレスから物理アドレスへの変換は、アクセスされる物理アドレスがオリジナル/コピーに関わらず、MMU で高速に変換することができる。クラスタ内キャッシングは物理アドレスを基に行う。

SVM では、1 つの仮想ページに対し、クラスタ毎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大域仮想アドレスを必要とする。これをネットアドレスと呼ぶ。

JUMP-1 で行われるアドレス変換の様子を図4に示す。

図4に示したように、クラスタ間で一貫性制御を行う際、パケット送信時には物理アドレスからネットアドレス、受信時にはその逆の変換が必要となる。これは MBP Core によって行われる。物理アドレスとネットアドレスの対応は自由であるが、このような大域仮想アドレスは、仮想アドレスにプロセス ID を付加したものとするのが普通である。この場合にはアドレス変換には、逆引き/正引きページテーブルを参照する必要がある。

2.2.2 一貫性制御

JUMP-1 では、false sharing を軽減するために、ページ単位ではなく、32Byte のライン単位で一貫性制御を行う。そのため、クラスタメモリにはライン単位で**stag** と

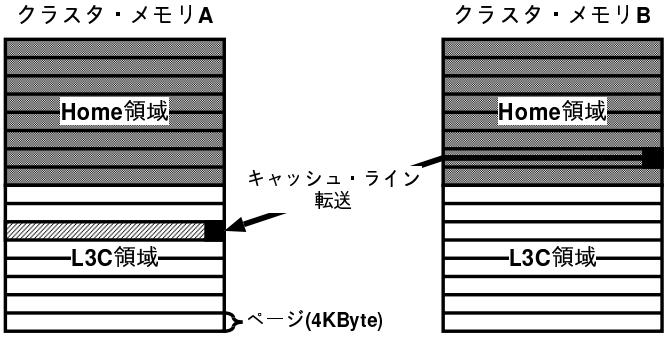

図 3: 3 次キャッシュにおけるキャッシング

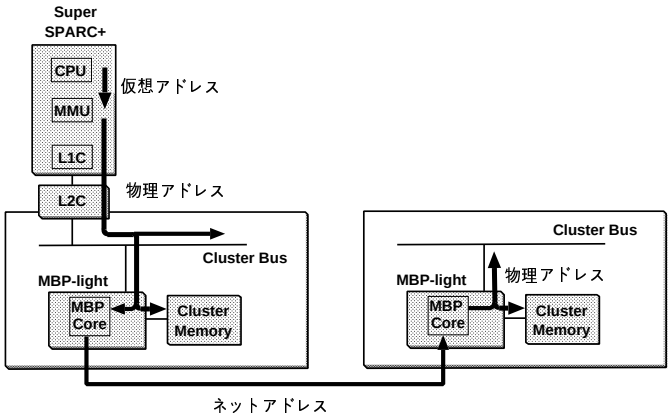

図 4: アドレス変換

呼ばれるタグが付加されている。

一貫性制御は、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分散ディレクトリ方式で行う。

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 [4]。

ディレクトリの、クラスタへのマッピングは自由であるが、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行う場合には、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、無効化型、更新型及びそ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　ラインの状態として、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 つの共有状態を持つ。

JUMP-1 では、以下のようにして、メモリアクセスに伴う一貫性維持の操作が、できるだけ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、

スヌープに対して状態を応答し、クラスタメモリアクセスの頻度を軽減する。キャッシュコントローラが処理できない場合に限り、MMC がクラスタメモリアクセスを行う。クラスタ内に閉じた処理になる場合には MMC がハードウェア的に処理を行い、クラスタ間処理が必要な場合には、MBP Core に処理を依頼する。したがって一貫性制御を行う上での MBP Core の役割は、クラスタ間の処理が必要である場合に限定される。

2.3 MBP-light

MBP-light は、JUMP-1 の DSM 管理における核となる専用の LSI である。

2.3.1 MBP-light の構成

図 5に MBP-light の構成と、周辺との接続の様子を示す。

前項で述べたように MMC は、2 次キャッシュコントローラが処理できなかったパケットに関してクラスタメモリアクセスを行う。まず stag を読み出し、クラスタ内で閉じるアクセスなのか、クラスタ間処理が必要なのか判定を行う。前者であれば、MMC がハードウェア的に処理を行う。後者であった場合、MBP Core に処理を受け渡す。MBP Core は、その上で動作するプログラムでクラスタ間処理を行う。

RDT Interface は、主に RDT とのパケットの送受信を行う他、マルチキャストパケットに対する Ack の生成と収集機能を備える [2]。

この節では、以後MBP Core に関して詳しく述べる。クラスタ間処理については、3、4で詳述する。

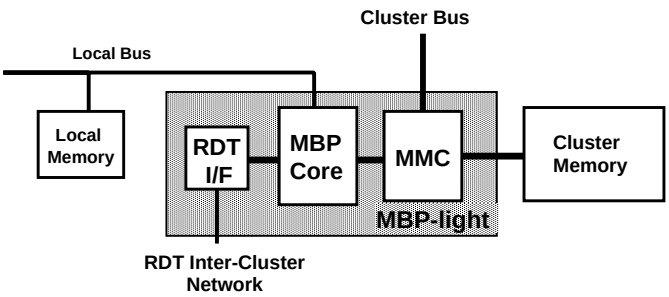

図 5: MBP-light

2.3.2 MBP Core

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。

MBP Core は命令長 21bit、データ長 16bit のスカラー RISC プロセッサである。16 本の 16bit 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。主記憶として、21bit × 64K の空間を持つ 1 サイクルでアクセス可能な外付けの SRAM **ローカルメモリ**を持つ (図5参照)。記憶階層はシンプルであり、キャッシュ、命令バッファ、ストアバッファ等は持たない。このような構成のため、ロード/ストア実行時には 1 サイクルのストールが発生する。

DSM 管理プロセッサとしての MBP Core の特徴を以下に挙げる。

- クラスタバスを介さないクラスタメモリアクセス
- パケット到着による割り込み発生機構
- パケット格納バッファ
- 特殊命令

以下これらについて述べる。

クラスタメモリアクセス　MBP Core は 3 次キャッシュの一貫性制御を行うために、クラスタメモリとそれに付随する stag にアクセスする必要がある。MBP Core は MMC を介することによってクラスタバスを介さずに、これらにアクセスすることが可能である。

パケット到着による割り込み　MBP Core は、MMC からのパケット処理依頼及び RDT からパケットが到着すると割り込みを発生させる機構を備える。また割り込み許可/禁止命令、多重割り込みを可能とする命令セットを持つ。これは 4で述べるマルチスレッド化したプログラムの実装に利用する。

PBR　MBP-light は **PBR** (Packet Buffer Register) と呼ぶ、パケット格納バッファを持つ。図 6にその模式図を示す。

PBR の語構成は 8bit×8B を 1 フリットとして、汎用が 64 フリット、RDT との送受信用にそれぞれ 8 フリット×3 パケットとなっている。

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、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+4bit 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、8bit、16bit 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　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 PBR の先頭及び送信用 PBR の末尾 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 のパケット送受信命令によって実行される。

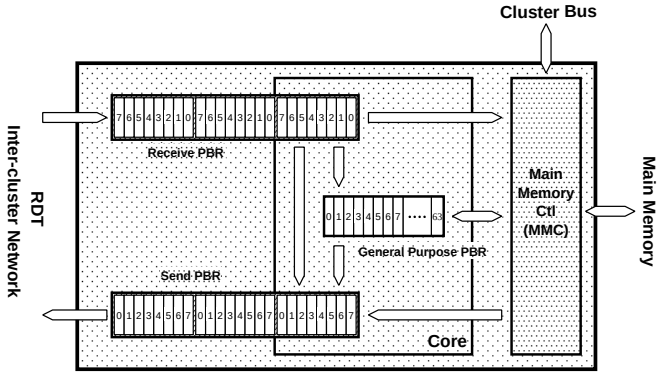

図 6: PBR

RDT から到着するパケットは受信用 PBR の先頭に直接格納される。送信用 PBR に置かれたパケットは、送信命令によってポインタが切り代わると、以後そのパケットの送信は RDT Interface に任される。

一方、クラスタバスとのパケットの送受信は、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。パケットのコピーは専用の 1 命令で実行される。このコピーには数十サイクルかかり、コピーに使用している PBR はその間使用することができない。

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クラスタメモリ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。

また MMC に対してクラスタメモリへのアクセスを依頼する際も、クラスタバスとの送受信と同様の操作が必要となっている。

特殊命令　PBR は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を持つ。この命令はアドレスをキーとする表を管理する上で効果的であり、通常の RISC 命令セットならば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。

3　DSM 管理

MBP Core の役割はクラスタ間にまたがる DSM の管理である。本章では MBP Core 上で動作する DSM 管理プログラムが満たすべき要件について述べる。

メモリ・アクセスに伴うクラスタ間のパケットの転送及び一連の状態更新をトランザクションと呼ぶ。MBP Core の役割はクラスタ間でのキャッシュの一貫性制御のためにトランザクションを処理することである。

トランザクションの処理は、基本的に到着したパケットに対してデータアレイとディレクトリの更新を行い、必要ならばパケットを送出することでなされる。ただしトランザクションの処理順序は、単に到着した順序で行えばよいわけではない。

まず 3.1節でトランザクションの流れについてまとめ、3.2節で競合と呼ばれる状態について述べる。さらに 3.3節でトランザクションの処理順序について議論する。

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次のように分類される。2.2.2項で述べたように、ラインのディレクトリを持ち、クラスタ間の共有状態を管理するクラスタをHome と呼ぶ。システム内で唯一の有効なラインを所持するクラスタを**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するクラスタを**Initiator** と呼ぶ。

簡単のため、ラインの状態が Global shared である場合には、Home は必ず有効なデータを持つこととする。

パケットの流れはMBP Core のプログラム次第であるが、簡単のため、三角通信を行わずに、応答も必ずHome を経由するものとする。パケットの基本的な流れは次のようになる：①Initiator から Home に、②Home から Owner/Renter に要求が転送される。③Owner/Renter から Home に応答が返され、④Home から Initiator に応答が転送される。

当然共有状態によっては、②③は省略されることもある。

トランザクションの例として (a) 読み出し要求と、(b) 無効化型書き込み要求におけるパケットの流れと各クラスタでのラインの状態、すなわち stag の変化を図7に示す。

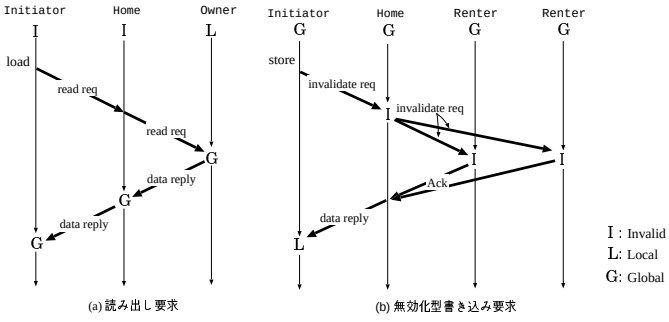

図 7: パケットの流れ

(a) 読み出し要求　Home に有効なデータが存在しない場合の読み出し要求のトランザクションの流れは以下のようになる。

1. Initiator は Home に読み出し要求パケットを送信する。
2. Home はOwner に読み出し要求パケットを転送する。
3. Owner はラインの状態を Global shared とし、Home にデータ付きの応答を返す。
4. Home はデータを自クラスタメモリに書き戻し、ラインの状態を Global とする。さらにディレクトリを更新する。Home は Initiator に応答を転送される。
5. Initiator はデータを自クラスタメモリに書き戻しラインの状態を Global とする。

(b) 無効化型書き込み要求　複数クラスタで共有されているデータに対する、無効化型書き込みトランザクションの流れは以下のようになる。

1. Initiator は Home に無効化要求パケットを送信する。
2. Home は自クラスタのラインを無効化し、Home から Renter に要求パケットが転送される。
3. Renter はラインの状態を Invalid とし、Home に Ack 応答を返す。
4. Home は全ての Renter から Ack を受け取れば、ディレクトリを更新しデータ付きの応答を Initiator に送信する。
5. 応答を受けた Initiator はラインの状態を Local shared とする。

3.2 競合

同一ラインに対するトランザクションが、異なるクラスタからほぼ同時に開始されることがある。この時 Home では、先行トランザクションの応答パケットが到着しないうちに、後続続要求パケットが到着することになる。この状態を競合と呼ぶ。

先行トランザクションの応答を待たずに、後続パケットの処理を始め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 では、応答待ちのトランザクションの情報を記憶しておき、競合を検出する。競合が検出された場合には、先行トランザクションの応答が返るまで、競合を起こした後続パケットの処理を何らかの方法で遅延させる。

無効化型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注意を払う必要がある。

図8左に相撃ちが発生している状況を示す。2 つの異なるクラスタがほぼ同時に同一ラインに対して無効化型書き込みトランザクションを開始している。実線矢印が Initiator 1 から、点線が Initiator 2 からのトランザクションを示す。Home には Initiator 1 からの要求が先に

届いており、その後に Home に届く 2 からの要求は、競合を起こし、何らかの方法で遅延させているものとする。

競合に負けた場合、すなわち Initiator 2 には、当該ラインに対する Home からの応答が返る前に、Initiator 1 から開始されたトランザクションの無効化要求が到着している。この場合、この時点で Initiator 2 は無効化要求を受け入れ、自コピーを無効化し、応答を返す必要がある。そうせずに、Initiator 2 が自分の開始したトランザクションが終了するまで、Initiator 1 からの要求を受け入れないとすると、デッドロックに陥る。

この時点で既に、Initiator 2 がラインの状態を不可逆的に更新していれば、書き込んだデータは失われてしまう。また Initiator 1 にも、後に Initiator 2 からの無効化要求が届き、Initiator 1 の書き込んだデータも失われシステム内に有効なデータが存在しなくなる。

したがって、無効化型書き込みでは、ラインの状態を不可逆的に更新するのは、競合に負けなかったことが判明した後でなければならない。

この対処には、負けなかったことが判明するまで、ライトバッファに書き込み内容を保存しておく方法が取られる。負けなかったことが分かるのは、Home から応答パケットが返された時であるので、それまで書き込み情報をバッファに保存しておく必要がある。

したがって、ライトバッファの数によって、同時に未完了にできる書き込み要求の数に制約が生じることになる。JUMP-1 では各プロセッサ毎に 3 つまでしか同時に未完了にできない。それを越える数の書き込みを行った際には、1 つ目の無効化が完了しライトバッファが解放されるまで待つことになり、ライトレイテンシが表面化することになる。

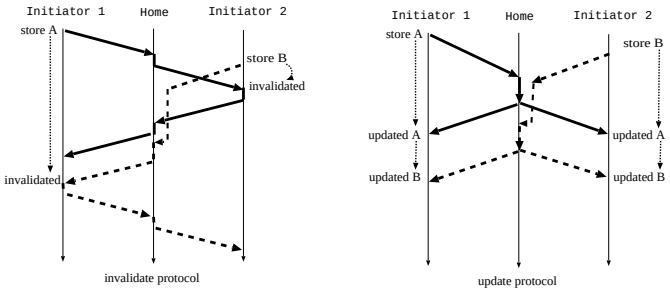

図 8: 競合

更新型トランザクションの競合　一方、更新型書き込みでは、更新要求パケット自体に書き込み内容が保存されているので、特別な配慮が必要ない。

図 8右側に更新型トランザクションの競合の様子を示す。

競合に負け、他クラスタからの更新要求によって自コピーが更新された場合でも、自分が送信した更新要求パケットが Home から戻ってきたときに、パケット内の情報を用いて自コピーを正しく更新することができる。

したがって、パケット送出後、直ちにライトバッファを解放することができる。この場合、バッファ資源によって同時に未完了にできる要求パケット数が制約を受けることがなく、レイテンシが表面化することがない。緩和されたメモリ・モデルの利点を十分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、無制限の更新要求を同時に未完了にすることができる。

3.3　トランザクションの処理順序

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。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。

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。

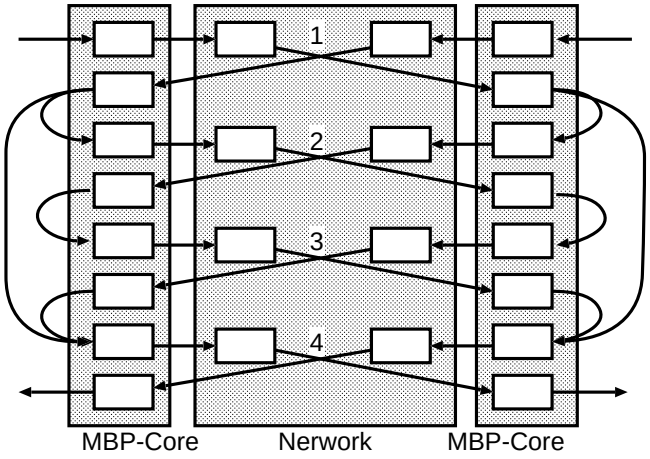

図 9: チャネル

その場合の資源要求の様子を図9に示す。図に示されるように、資源要求には閉路がないため、デッドロックは発生しない。

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 [6]。これは9に示した4つのチャネルのうち、1と2、3と4を1つのチャネルとして兼用することに相当し、その際閉路が形成される。したがって、こ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。

再試行によってデッドロックを防止するには、再試行を要求するパケットによって新たな資源の要求関係が生じてはならない。すなわち、再試行を要求するパケット自体は必ず受理できる必要があるが、これは以下のようにして容易に実装することができる。要求パケットを送信した時に、その送信バッファを解放せずにとっておき、再試行要求パケットが届いたときには、対応する送信バッファを未送信の状態に巻き戻せばよい。このようにすれば、新たな資源要求を導入することなく、再試行要求パケットを必ず受理できるようにすることが可能である。

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。

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、普通は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方法が採られる。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。デッドロックの検出には、楽観的なものから悲観的なものまでさまざまな方法が考えられる。

このような方法を採れば、十分長のバッファは、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 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 [7]。

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。

   また、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。

更新要求の利点を生かすためには、このバッファは無限長となり。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

MBP Core は通常のライン処理と比較して、非常に長い時間がかかるものが存在する。このような長いトランザクション処理が MBP Core の処理を占有することによって、後続の通常ライン処理が処理されないことになると、平均レイテンシの悪化につながる。

また、3.2節で述べたように、競合が発生した場合には、先行する処理 A が終了するまで、競合する後続処理 B は待たされる。しかし、B のために A を終了を待つことのみに MBP Core を占有されると、さらに後に届く競合しないパケット C が処理されないことになり、やはり平均レイテンシの悪化を招く。

以下では、長いトランザクションと競合に関して、平均レイテンシ短縮のためにパケット処理順序における注意点を述べる。

長いトランザクション 長いトランザクションの例は、ページの書き戻し処理がある。2.2.1で述べたように、ページフォールト時に 3 次キャッシュが溢れ新たなページ枠が確保できない場合には、リプレース処理を行う必要がある。ページ単位の処理には、通常のキャッシュライン単位の処理と比較すると明らかに長い処理時間を要する。

平均レイテンシの短縮ためには、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、後続パケットを先に処理する方がよい。

競合 競合については、対処方法によってレイテンシ以外の問題が発生することがある。できれば、競合した B を待たせたままで、後続の競合しない C を先に処理することが望ましい。この対処には再試行、十分長バッファが考えられる。

前者では B に再試行を要求しおいて、C の処理を行う。しかし前項で述べたように、スターベーション及びライトバッファの資源制約の問題が残る。DASH では、この方法が取られている [6]。

十分長バッファでは、B をバッファに退避しておき、C を先に処理する方法が考えられる。このバッファは、前項で述べた十分長バッファと同様に、システム内に存在可能な要求パケットをすべて収容できる容量が必要である。JUMP-1 では、無限長バッファとなり、やはり MBP Core プログラムによって対応する必要がある。

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 [7]。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 DSM 管理プログラム

本章では，前章での議論を踏まえて実装した，MBP Core 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述べ、以降その実装について述べる。

4.1 DSM 管理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：

デッドロックの回避 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、その処理を中断し、後続のパケットを受け付けバッファ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、通常ライン処理の後回しにする。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。

これらパケット処理順序に対する要求は、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する。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、競合を起こさず、また送信バッファが塞がることもなく短時間で終了する。このようなパケットに対して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、基本的に以下のような方針でスケジューリングを行う。

まずパケットを 2 種類に分ける。処理時間が長い、あるいは、処理中に中断する可能性ある、あるいは、到着時に実行することが不可能なパケットに対してはスレッドを生成する。一方、これ以外のパケットに関してはスレッドを生成せずに到着時に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。

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、手続き的に実行する。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。

DSM 管理プログラムは、到着したパケットに対して Short-lived パケットと Long-lived パケットの管理を行うカーネルと、パケット処理部から構成される。

以降 4.2節でカーネルの概要について述べた後、4.3 節で Core プログラムの持つデータ構造についてまとめ、4.4節でカーネルの動作の流れを述べる。

4.5ではパケット処理部、特にページ単位の処理に関して詳しく述べる。

4.2 カーネルの概要

カーネルの基本的な動作は、到着したパケットに対して Short-lived パケットと Long-lived パケットのどちらであるかを判断し、前者であればそのまま手続き的に処理を行い、Long-lived パケットであればスレッドを生成、スケジューリング後実行することである。4.4節の図 11に処理の流れを示す。詳しい動作については 4.4節で述べる。

この節では、4.2.1項でパケットの到着の検知方法について、4.2.2項で Short-lived パケットと Long-lived パ

ケットの具体的な判別方法とスケジューリングについて述べる。最後に4.2.3項で到着したパケットの状態遷移をまとめる。

4.2.1 パケットの到着

パケットの到着を検出する機構として、

1. 受信バッファのポーリング
2. パケットの到着による割込み

の2つが考えられる。MBP Coreではどちらの方法も実装が可能である。

適切な場所でポーリングを行うことで、コンテクストの退避/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ことは困難である。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間隔でポーリングを行う必要があり、現実的ではない。

一方、割り込みによる実装のオーバーヘッドはコンテクストの保存であるが、これは実際には関数呼び出し程度のコストしかかからない。特にアイドルスレッド実行中であった場合にはコンテクストの退避は必要ない。

従って、今回はパケットの到着を割り込みによって検出することとする。

4.2.2 Long-lived パケットと Short-lived パケット

この項では、Long-lived パケットと Short-lived パケットの判別とスケジューリングについて詳述する。

前節の処理順序の制約から、到着するパケットを以下のように分類する。

1. 処理時間が長いもの
2. 競合を起こし、処理自体が始められないもの
3. 必要な送信バッファの空きを、処理前に確認することが不可能なもの
4. 3. は可能であるが、チェックした結果塞がっていたもの
5. 上のいずれでもないもの

5. に関しては、Short-lived パケットとして到着時に手続き的に処理を行うこととする。Core プログラムはパケット到着時に、1〜4 の条件を、処理前にチェックし Short-lived パケットとなるか否かを判定する。

1. と 3. に関してはパケット種で判断することが可能である。2. に関しては、4.3節で述べる PTT と呼ばれる、アドレスをキーとしたハッシュテーブルを用いて調べる。4. に関しては、パケット種から必要な送信バッファの種類を判断し、そのバッファの空きを確認することでなされる。

以下では、Long-lived パケットとしてスレッドを生成する 1〜4、それぞれの処理に関して述べる。

1. 処理時間が長いものは平均レイテンシの観点から、処理中にパケットが到着したときには中断し、Short-lived パケットを優先させて処理する方が望ましい。したがって、スレッドを生成し、割り込み可で処理を行い、パケット到着時に割り込まれるようにする。

2. 競合を起こしたものは、先行処理が終了するまで処理自体を始めることができない。この時、後続の競合しないパケットは処理できることが望ましいため、スレッドを生成して待たせることで、後続のパケットを受け付ける。先行処理が終了すると、実行可能状態となる。

3. は複数のパケットを同一方向に送信するような処理を行うパケットである。Core から確認することのできる送信バッファはパケット 1 つ分であり、このようなパケットに対して処理前に、中断なく処理することを保証できない。したがってスレッドを生成後処理を始め、処理中に送信バッファが塞がっていれば中断させるようにする。

4. 殆どのパケットは、その処理において送信するパケットは1つである。そのようなパケットに対し、必要な送信バッファのチェックを行った結果が塞がっていた場合である。このパケットに対しても 3 と同じ方針で実行しても問題はない。しかし一旦送信バッファが確保できれば中断なく処理することが可能であることから、スレッドを生成し待たせておき、送信バッファの確保ができてから一気に実行してしまう方が良い。そのためスレッド生成後は、レディキューに登録し、スケジューラが送信バッファの空きを確認した時点でディスパッチを行うことにする。

以降ではスケジューリングの方針を述べる。

1. に関してはレイテンシの観点から、処理中は割り込みを許し、Short-lived パケットを優先して実行する。

しかし 2・4 に関しては、競合が解消され、また送信バッファの確保を処理前に行うことができれば、処理中に中断することなく、また処理自体は短時間で終了する。

Long-lived パケットとしてスレッドが生成されたものには、長時間かかる処理と、短時間で終了するものがある、前者を単に長いスレッド、後者を短いスレッドと呼ぶことにする。

スケジューリングは Short-Job-First で行うのが望ましいことから、長いスレッドと短いスレッドの間のスケジューリングは、短いスレッドを優先して実行する。また長いスレッドは割り込みを許可して実行し、新たにパケットが到着すると中断する。新たに到着したのが Short-lived パケットであれば、そのまま手続き的に実行を行うことで、Short-Job-First を実現する。

また長いスレッドが複数ある場合は、タイマで切り替えて実行する。

実行可能状態となった短いスレッドは、その処理に必要な送信バッファの確保ができた時点で実行する。スケジューラは各スレッドに対して、必要とされている送信バッファの状態を調査し、空きが確認された時点でそのスレッドをディスパッチする。

スケジューラは実行可能状態にあるスレッドが存在していると、送信バッファの状態を調査しつつループする。このとき新たなパケットの到着を受け付けなければデッドロックに陥る。したがって、スケジューリング中の割り込みを許可する。このとき常に割り込みを許可すると、スターベーションが発生し、実行可能状態にあるスレッドが実行されない可能性がある。そのため復帰直後は、割り込みを許可せずスケジューリングし、一定数ループが回った後に再び割り込み許可とする。

実行可能状態のスレッドが存在しなければ、スケジューラはアイドルスレッドを実行する。

4.2.3 パケットの状態遷移

Short-lived パケットは到着後直ちに実行され、終了する。Long-lived パケットに対するスレッドの状態には、実行可能状態、実行中、待ち状態 の 3 つの状態がある。

各状態の遷移を図 10に示す。

図中点線は、Short-lived パケットの流れであり、到着直後に実行され、実行は中断されることなく終了する。

前述したように競合が検出されると、スレッドが生成され先行処理の終了を待つ。このとき待ち状態となる。先行処理が終了すれば、Wakeup され、レディキューに登録される。

また処理時間が長い、もしくは必要な送信バッファが確保できなかった場合には、スレッドが生成され実行可能状態となる。

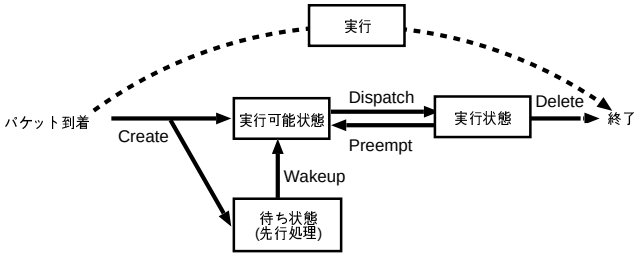

図 10: スレッドの状態遷移

長いスレッドは、パケットの到着によって横取りされる。また、このようなスレッドが複数ある場合には、タイムアウトによって別のスレッドに横取りされる。

横取りが発生するのは長いスレッドのみであり、短いスレッドは割り込み不可で実行され終了する。

4.3 データ構造

Core プログラムの持つ主なデータ構造は以下のものである。

1. スレッドテーブル
2. レディキュー
3. Pending Transaction Table(PTT)
4. ソフトウェア TLB

本節ではこれらについて詳しく述べる。

スレッドテーブル Long-lived パケットに対して作成されたスレッドの情報を登録する。通常の OS と同様に、スタックポインタやプログラムカウンタ等のメンバを持つ。これ以外にパケットバッファへのポインタ、PTT へのポインタを持つ。

レディキュー スレッドテーブルのリストから構成される、無限の深さを持つキューであり、実行可能状態にあるスレッドが登録される。長いスレッド用に 1 本と、短いスレッド用には送信バッファ毎に用意される。

4.2節で述べたように、到着したパケットが送信バッファが塞がっていることで処理されなかった場合、スレッドが生成される。このとき、その送信バッファに対応したレディキューに登録される。したがって、このキューが 3.3.1項で述べたデッドロックを回避するための十分長バッファの役割を果たす。

Pending Transaction Table Pending Transaction Table(以下PTT) は、ライン単位のアドレスをキーとするハッシュテーブルである。未完了のパケットに関する情報を管理し、主に競合の検出を行うために使われる。

PTT はアドレスとスレッドテーブルへのポインタを持つ構造体が、リストで繋がれた構成を持つ。

Home が要求パケットを受信した時には、まずPTTを検索する。同一ラインに対する登録が既に存在すれば、競合が発生していることがわかる。競合が発生すれば、パケットをローカルメモリ上に退避し、スレッドを作成する。PTT の対応エントリに登録され、先行パケットの終了を待たせる。

先行パケットの応答が返されトランザクションが完了すれば、対応するエントリを削除する。この際、競合を起こしていたパケットが登録されていれば先頭の1つを実行可能状態にする。

PTT ではライン毎にキューイングがなされ、このキューが3.3.2項で述べた十分長バッファにあたる。ライン毎にキューイングであるため、競合発生時にも別のラインに対するパケットは処理することができる。

またPTT は競合の検出以外に、要求パケットの送信元の情報を保持している。この情報は応答到着時に、その応答の転送先を知るために使われる。Home 以外のクラスタでは、この用途にのみ使用される。

ソフトウェア TLB Core プログラムは、ディレクトリと Home 検索テーブル及び正引き/逆引きページテーブルにアクセスする必要がある。ディレクトリと Home 検索テーブルは容量上ローカルメモリに置くことが出来ずクラスタメモリ上に置かれており、また両ページテーブルは OS が整備するためやはり主記憶であるクラスタメモリに置かれる。

2.3節で述べたように、Core からクラスタメモリへのアクセスには、MMC を介する必要があり時間がかかる。そこでこれらのテーブルへの参照を高速化するために、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、各テーブルの内容をキャッシングしてから利用する。なお、MBP Core では、アドレスのハッシュ値は1命令で求めることができる。

Home 検索テーブルとページテーブルは、いずれもページ番号をキーとする検索である。従ってこれらの TLB を統合し、アドレス変換と Home の検索を同時に行う。

またディレクトリの TLB と PTT は、ライン単位でのアドレスをキーとする検索である。そこでこれらも統合し、同時に検索することを可能とする。PTT の構造体は、キーとなるアドレスのみメンバとして共有し、競合検出のためのメンバとディレクトリのメンバを合わせ持つことになる。Home 検索テーブルとページテーブルの TLB 統合とは異り、未完了トランザクションの登録とディレクトリの登録は別々に行われる。このときキーとなるアドレスが異るものを登録しようとする場合があるが、競合を検出することは必須であるため、未完了トランザクションの登録を優先させる。一方、未完了トランザクションの登録が削除される際は、ディレクトリの登録は残しておき、検索時にはディレクトリにヒットし、かつ、競合が発生していない状況に対して最適化を行った。

4.4 処理の流れ

図 11にカーネルの処理の流れを示す。図中の点線矢印は、割り込みを表す。

カーネルはパケット到着時の処理を行うパケットハンドラと、スケジューラに分かれる。

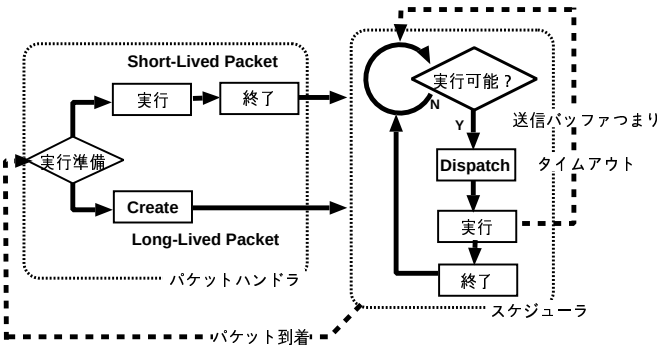

図 11: カーネルの処理の流れ

4.4.1 パケットハンドラ

前項で述べたように、パケットハンドラはShort-lived パケットと Long-lived パケットの判別を行う。そしてShort-lived パケットならばそのまま実行を行い、Long-lived パケットならばスレッドを生成する。

以下パケットハンドラ処理の流れを述べる。

パケット到着

パケットの到着は割り込みによって知らされる。

実行中のスレッドが横取りされ、コンテクストの退避が行われる。ただし、アイドルスレッド実行中であった場合にはこの処理は省略される。

実行準備

パケット実行に必要な情報を準備し、同時に4.2節で述べた方針でLong-livedパケットかShort-livedパケットかの判別を行う。

具体的には、アドレス変換、Homeの検索、パケット種の判別、ディレクトリの検索、競合の検出、送信バッファの確認を行う。

当然Homeでなければ、ディレクトリの検索と競合の検出は省かれる。

このとき、4.3節で述べたように各種TLBは統合しているため、アドレス変換とHomeの検索、ディレクトリの検索と競合の検出は同時に実行できる。

パケット種から、処理時間の長いトランザクション、もしくは送信バッファの確認ができるものであるかを調査することができる。これらの場合、スレッドを生成し、レディーキューに登録する。また競合が検知された場合にはPTTに登録し先行処理の終了を待たせる。必要な送信バッファが詰まっていた場合は、そのバッファに対応するレディキューに登録する。これらLong-livedパケットであった場合には、パケットハンドラの処理は終了し、直前に実行されていたスレッドが再開される。

Short-livedパケットであると判断された場合には、そのまま手続き的に実行してしまう。

実行

パケット処理ルーチンが呼ばれる。

この時点で、パケット処理は中断なく行うことが可能であることが保証されており、また処理のために必要な情報は全て揃っている。

終了

パケット処理の結果、応答が必要な要求パケットならば未完了トランザクションとしてPTTに登録される。

また応答パケットであった場合には、PTTの対応エントリが削除される。この時競合していたパケットが登録されていれば、先頭の1つを実行可能とする。

これらの処理が終了すれば直前に実行されていたスレッドが再開される。

4.4.2 スケジューラ

スケジューラは、レディキューに登録された実行可能状態にあるスレッドを選択実行する。

短いスレッド用のキューは、送信バッファ毎に用意され、登録されているスレッドは実行に際してその送信バッファを必要とするものである。スケジューラはバッファの調査を行い、空きが確認されれば先頭の1個をディスパッチする。

長いスレッド間ではタイマで割り込みをかけることで、処理を切り替える。

4.5 パケット処理部

前節のカーネルの動作の流れの中で、「実行」にあたる。

パケット処理には、通常ライン処理と、それとは同列に扱いにくいページ単位の処理がある。

4.5.1 通常ライン処理

この処理が始まる時点で、ディレクトリ、Homeの位置、ネット・物理アドレス等必要な情報は全て得られている。

ここで行うべきことは、パケットの種類とディレトリの状態に従って、ディクレトリとクラスタメモリのデータラインを更新し、パケットを3.1節の流れに従って送信することである。

以下ではパケットの送信について述べる。丸付き番号は3.1節のフェーズ番号と一致する。

ディレクトリはキャッシュライン毎に、当該ラインの有効なコピーを持つクラスタの情報を縮約階層ビットマップの形式で保持している。②HomeからOwner/Renterにパケットを送信する際には、これをそのままRDTパケットヘッダに挿入して送信すれば良い。

Homeの位置もやはり縮約階層ビットマップ形式で保持する。これは①③Initiator/Owner/RenterからHomeへのパケット転送に使われる。

ただし、④HomeからInitiatorへの応答パケット転送の場合は、他の転送と異りテーブル検索等でInitiatorの位置を求めることができない。①でInitiatorから受け取った要求パケットのヘッダを解析することで求めることも可能であるが、時間がかかる。そのため今回は、①InitiatorからHomeに要求パケットを転送する際に、やはり縮約階層ビットマップで表されたInitiatorの位置情報をパケットに付加して送信することとした。

この①で使用するInitiatorの位置情報は、Home検索テーブル内に、Homeの位置情報と合わせて登録しておき、Home検索の際に同時に求まるようにした。

4.5.2 ページ単位の処理

3次キャッシュが溢れた場合、リプレース処理を行い新たなページ枠を確保する必要がある。このとき、リプレース対象となったページ中でDirtyなラインをHomeに書き戻す必要がある。3次キャッシュ溢れの検知はOSによって行われ、MBP Coreにページ書き戻し要求が送られる。

要求を受けとったCoreは、単に当該ページ内でDirtyなラインを検索して、次々と書き戻しパケットをHomeに送信すればいいわけではない。このままでは書き戻すクラスタとHomeとの間で、同一アドレスの通常ライン処理と書き戻しパケットがすれ違う可能性がある。この状況まで考慮して実装を行うと、通常ライン処理が複雑になる。

そこで、先にHomeに通知し、書き戻し処理が完了するまでは、当該ページに関する通常トランザクションを止める方法が考えられる。通知を受け取ったHomeは、以後受け取る当該ページに関する要求パケットを止めておき、既に未完了状態であるトランザクションの終了が確認できた時点で、Initiatorにパケットを送信し、書き戻しの許可を与える。

ただし、書き戻し対象にないラインに関する要求パケットまで止める必要はない。また、既に書き戻されたラインに関しても、すべてのラインの書き戻し終了まで待つ必要はない。

書き戻し対象のラインについてのみ、以後到着する要求パケットを止め、また書き戻されたラインについては、書き戻し直後に通常トランザクションが実行可能となることが望ましい。そこで、Homeにおける未完了トランザクションの終了確認と、要求パケットの実行抑止にPTTを利用する。

次のような処理がなされるパケットを考えると都合が良い：ディレクトリを調べ、InitiatorがOwnerであれば、すなわち書き戻しの対象ラインであれば、書き戻しパケット待ちとしてPTTに登録される。それ以外の場合はディレクトリの共有状態からInitiatorを外し終了する。

このパケット処理がなされると、書き戻しが必要なラインに関してのみ、以後到着する要求パケットは競合を起こし待たせることとなる。それ以外のラインは書き戻しを待つことはない。また書き戻しパケットを受け取り、書き戻しの処理が終了した時点でこのPTTエントリは削除され、止められていた要求パケットの処理は実行可能となる。

書き戻し通知パケットを受け取ったHomeは、このようなパケットを、当該ページ内の全ラインに関して受け取ったものとし、各ライン毎に処理を試みる。具体的には、通常ライン処理と同じくPTTを検索し、競合の検出を行う。競合が検知されれば、未完了トランザクションが存在することを意味している。その際、このパケットをPTTに登録する。競合を起こしていなければ、そのまま上記のような処理がなされる。したがって、このパケットが処理された時点で、未完了トランザクションは終了していることが保証される。

全てのラインに対して処理が終了すると、書き戻しの必要なラインに関してのみPTTに登録が残っている状態となる。従って、以降このラインに関するトランザクションは止められることになる。HomeはInitiatorに対して書き戻しを許可するパケットを送信する。以後、Initiatorが自クラスタでDirtyなラインを読み出して次々とHomeに送信すれば良い。

5 評価

この章では、前章で実装したDSM管理プログラムを実機上で動作させ、評価を行う。5.1節ではトランザクションの処理時間を計測し、得られた結果から5.2節で考察を行う。

5.1 トランザクションの処理時間

この節ではクラスタ間トランザクションのうち基本的な読み出し要求と無効化型書き込み要求の、実機上での計測結果を述べる。

5.1.1 評価環境

計測にあたってシステムは無負荷状態とした。すなわち、各クラスタにパケットが到着したときMBP Coreはアイドルスレッド実行中であり、また送信バッファが塞がっていることもない。この時、到着したパケットは全てShort-livedパケットとして処理される。また各種TLBはすべてヒットするとした。

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、gcc-2.8.1 を MBP Core 向けにポーティングしたものを用いた。

また今回はネットアドレスとして、仮想アドレスを用いることとした。

5.1.2 読み出し要求

計測結果を表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節で述べたトランザクションの 4 つのフェーズを示す。例えば、①→② は、要求パケットが ① Home に届いてから、② Owner に向けて送信する までの MBP Core の処理を表す。

左側の列は Home の主記憶でヒットし② と ③ が省略できた場合、右側は Home でミスし Owner が応える場合である。Home の主記憶でヒットするとは、ディレクトリ検索の結果、共有状態が global shared であり、主記憶に有効なデータが存在すると判断した場合である。一方、Owner はクラスタメモリ上に有効なデータが存在することは確認できないため、クラスタバスに要求を転送する必要がある。表中③→ は Home からのパケットを受け取ってからクラスタバスへの転送、→④はクラスタバスから応答を受け取ってから Home に送信するまでの処理を表す。

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。Owner が応える場合は、これに Core が要求をクラスタバスに送信してから、その応答が Core に割り込みをかけるまでの時間が加わる。

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。

各クラスタでの Short-lived パケットの処理に 100〜150 サイクルかかる結果となっている。

Home、Owner が応える場合のそれぞれおける Core の処理時間は 310、609 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 156.6%、151.5% となる。

5.1.3 無効化型書き込み要求

無効化型書き込み要求の処理時間を計測した。この場合、Home は共有ノードに対して無効化要求を送信し、Ack を集める必要がある (3.1節の図7(b) 参照)。2.3 節で述べたように MBP-light のハードウェアの Ack 生成機構、収集機構を利用することができる。Ack 生成、収集それぞれに対しこの機構を利用した場合及びそれを利用せずに、ソフトウェアで行った場合の 4 通りの方法を用い計測を行った。

計測結果を表3に示す。示しているのは、トランザクションにかかる合計サイクル数である。() 内は Home において無効化のマルチキャストを行ってから Ack 収集が完了するまでのサイクル数である。

各列は以下の場合である。

- (a) ハードウェアで生成・収集
- (b) ハードウェアで生成、ソフトウェアで収集
- (c) ソフトウェアで生成、ハードウェアで収集
- (d) ソフトウェアで生成・収集

ハードウェアが Ack の収集を行う場合 (a)(c) には、Renter 数によらずほぼ一定の時間で Ack の収集が完了していることがわかる。一方、ソフトウェアで収集を行う場合 (b)(d) は、Renter 数が増える毎に、40 サイクル程度時間が増している。これは、前者では収集が完了してから Core に割り込みがかかる一方で、後者では Ack が到着する度に Core に割り込みがかかり、ソフトウェアで Ack を数える必要があるためである。Renter 数がさらに増加したときは、トランザクション全体の処理時間を考えると無視できなくなると推測される。したがって Ack の収集はハードウェアでサポートする方が良いと考えられる。

一方、Ack の生成については (a)(c) の比較から、ソフトウェアで収集した場合と比較して、ハードウェアで

表 2: 読み出し要求処理時間

	Home read	Owner read
→①	109 (92/ 17)	109 (92/ 17)
①→②	— (—/ —)	110 (110/ 0)
②→	— (—/ —)	105 (88/ 17)
→③	— (—/ —)	128 (94/ 34)
③→④	142 (125/ 17)	149 (132/ 17)
④→	110 (93/ 17)	110 (93/ 17)
クラスタ内HW	45 (0/ 45)	92 (0/ 92)
クラスタ間HW	102 (0/102)	208 (0/208)
合計	508 (310/198)	1011 (609/402)

生成した場合には 90 サイクル程度、トランザクション全体では 10% 程度の高速化がなされていることがわかる。

5.1.4 Home に対する読み出しトランザクション処理時間の内訳

以降では、Home に対する読み出しトランザクションの処理時間をもとに高速化手法の評価を行う。そのために Home に対する読み出しトランザクションの処理時間の内訳を表 4 に示しておく。

4.3節で述べたようにアドレス変換と Home の検索 TLB、及びディレクトリ TLB と PTT は統合されているため、これらは同時に求められている。

また④→における PTT の検索は、4.3節で述べたように要求パケットの送信元、この場合はプロセッサ番号を得るために行われる。

5.1.5 高速化手法の評価

本項では、前項で計測した Home に対する読み出しトランザクションの処理時間を用いて、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評価を行う。

ソフトウェア TLB ソフトウェア TLB を用いる処理は、ネットと物理の間のアドレス変換、Home の検索、ディレクトリの検索となる。

ソフトウェア TLB の効果を検証するため、これを用意しない場合のプログラムを記述し、命令数を数えることで、上記それぞれの処理に要するサイクル数を見積もった。結果を以下に示す。

- ネットから物理へのアドレス変換 123
- 物理からネットへのアドレス変換 58
- Home の検索 48
- ディレクトリの検索 64

一方、Home に対する読み出しトランザクションで TLB を利用しヒットした場合、表 4 より処理→①のアドレス変換に 33 サイクル、①→④ と④→のアドレス変換に 41 サイクルかかる結果となっている。また Home の検索は、アドレス変換と統合されており、同時に求められる。ディレクトリの検索は、PTT と統合し、やはり同時に求めることができる。競合の検出のため、TLB を利用するしないに関わらず、PTT の検索は必要であるためディレクトリ検索に必要なサイクル数は 0 と考える。

したがって Home に対する読み出しトランザクション全体では、ソフトウェア TLB を利用することで、$(123-41)\times 2+(58-32)+48+64=302$ サイクルの高速化が達成されていることになる。

表 3: 無効化型書き込み要求処理時間

	(a)	(b)	(c)	(d)
1 Renter	764 (178)	771 (211)	853 (267)	825 (265)
2 Renter	771 (185)	819 (259)	861 (275)	872 (312)
3 Renter	771 (185)	873 (313)	861 (275)	909 (349)

表 4: Home に対する読み出し要求処理のレイテンシ

処理内容		サイクル数
→①	MMC からのパケット転送	28 (11/ 17)
	アドレス変換，home の検索	33 (33/ 0)
	パケット種判別	12 (12/ 0)
	送信バッファの確認	7 (7/ 0)
	ヘッダ作成	30 (30/ 0)
	小計	109 (92/ 17)
①→④	アドレス変換	41 (41/ 0)
	PTT，ディレクトリ検索	20 (20/ 0)
	パケット種判別	12 (12/ 0)
	送信バッファの確認	7 (7/ 0)
	主記憶読み出し	32 (15/ 17)
	ヘッダ作成	30 (30/ 0)
	小計	142 (125/ 17)
④→	アドレス変換	41 (41/ 0)
	PTT 検索	22 (22/ 0)
	パケット種判別	12 (12/ 0)
	送信バッファの確認	7 (7/ 0)
	ヘッダ作成	9 (9/ 0)
	MMC へのパケット転送	19 (2/ 17)
	小計	110 (93/ 17)
クラスタ内 HW		45 (0/ 45)
クラスタ間 HW		102 (0/102)
合計		508 (310/198)

スレッド生成省略の効果 Home に対する読み出しトランザクションの処理 →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を考える。

スレッド生成の操作は、パケットの退避、スタックの割り当て、スレッドテーブルへの登録、レディキューへの登録である。これらの処理に要する時間の合計は、282 サイクルとなった。このうち 160 サイクルはパケットをローカルメモリに退避する操作であり、半分以上を占めている。

したがって Home に対する読み出しトランザクションの場合、到着したパケットに対し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全体では 836 サイクル増加することになる。さらに、スケジュー

ラによってディスパッチされるまでの時間が加わることになり、Short-lived パケットに対するスレッド生成省略は効果的であるといえる。

5.2 考察

この節では、前節の Home に対する読み出しトランザクションの結果をもとに、Core の評価と JUMP-1 のメモリシステムに関して考察する。

5.2.1 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。

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、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。このオーバーヘッドは本質的ではないため、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ているモデルを **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 49 サイクル短縮され、459 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のみ用いられており、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 サイクル短縮される。

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、5 回分、30 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、32 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、21 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 $(32+21)-(9+30)=14$ サイクル短縮され、445 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、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 $9+30=39$ サイクル短縮され、406 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 105.0%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。

5.2.2 JUMP-1 のメモリシステム

本節では JUMP-1 のメモリシステムについて考察する。

5.1項の結果から、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約 150%となった。また 5.2.1項から、Home に対する読み出しトランザクションの場合には、Core の 32b 化、ID 分離化することで 105.0%まで削減できることが分かった。

ノード間処理を全て布線論理で処理することを考える。このとき、ソフトウェアオーバーヘッド 105.0% という値は、たとえハードウェア化によりソフトウェアで処理していた部分の処理時間が 0 になると仮定してもレイテンシは半分程度にしかならないこと示している。

以下では、Home に対する読み出しトランザクションの処理時間として次の 3 つの値を用いて検証を行う：5.1 項で計測した (1)508 サイクル、5.2.1項で求めた Core の 32b 化、ID 分離化したモデルでの (2)406 サイクル、すべてハードウェアで処理することでソフトウェアの処理時間が 0 になると仮定した場合の (3)198 サイクル。

階層的なタイリング [8] を施して、各階層におけるキャッシュヒット率の向上を図った行列積を、JUMP-1 で計算した場合を考える。

行列積の各階層でのキャッシュヒット率を表2.1のキャッシュパラメータから求め、Home に対する読み出しトランザクションの処理時間が (1)(2)(3) それぞれの場合の読み出しの平均レイテンシを計算した。

5に結果を示す。各行は、(a)3 次キャッシュサイズを 8MB とした場合、(b)3 次キャッシュヒット率を 100%と仮定した場合である。また行列積に限らず (c) キャッシュヒット率が悪い場合の結果も示す。

(a) のようにキャッシュヒット率が高い場合では、(1)(2) のソフトウェアで処理する場合と (3) ハードウェアで処理した場合とでは殆ど差がない。また (b)3 次キャッシュヒット率を 100% と仮定した場合と比較しても同様である。(c) の場合、(3) ハードウェアで処理しても (1)(2) と比較してそれぞれ 6.18%、4.23% 程度しか改善されないことになる。

6 終わりに

本稿では JUMP-1 におけるクラスタ間 DSM 管理プログラムの実装と、それを用いた性能評価の結果について述べた。

JUMP-1 では、リモートクラスタの主記憶に対し大容量 3 次キャッシュを利用し平均レイテンシの短縮を図る。また DSM 管理は、クラスタ内はハードウェアで高速に行い、クラスタ間はソフトウェアで柔軟に行うというアプローチを採る。

クラスタ間処理を行う DSM 管理プログラムは、トランザクションの処理順序に対する制約を満たすためにマルチスレッド化し、処理順序の制約をスレッド間のスケジューリングの問題として対処した。また、スレッド化のために、通常トランザクションの処理時間が増加しないよう注意を払った。

このプログラムを実機上で動作させ、3 次キャッシュミス時のメモリアクセスレイテンシを計測した。結果、Home に対する読み出しの場合 508 サイクルかかる結果となった。また、この処理におけるソフトウェアオーバーヘッドは 156.6%となり、Core の 32b 化、I/D 分離化により 105.0% まで削減できることが分かった。このデータを用いて、JUMP-1 のクラスタ間処理をソフトウェアで行う方式の検証を行ったところ、全てハードウェアで行う場合と比べても見劣りしないとの結論を得た。

また JUMP-1 の実装は .5〜.4μm のテクノロジの時代のものであり、本稿のデータは、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ことが期待される。

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考える。

謝辞

本研究の機会を与えてくださり、適切な御指導を賜わった富田 眞治教授に深甚な謝意を表します。

また、貴重なご助言をいただいた森 眞一郎助教授、五島 正裕助手に深く感謝致します。

さらに，共同研究者である額田 匡則 氏をはじめとして，日頃から御鞭撻下さったコンピュータ工学講座計算機アーキテクチャ分野の諸兄に感謝致します。

また、日頃から貴重な御意見を頂いている文部省重点領域研究共同研究者各位に感謝致します。

表 5: 読み出し平均レイテンシ

	キャッシュヒット率 (%)			平均レイテンシ		
	1 次	2 次	3 次	(1)	(2)	(3)
(a)	98.340	99.756	99.902	1.06739	1.06739	1.06737
(b)	98.340	99.756	100.0			1.06737
(c)	90.0	93.0	95.0	1.75565	1.71995	1.64715

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] 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).

[3] 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).

[4] 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).

[5] Li, K.: IVY: A Shared Virtual Memory System for Parallel Computing, *ICPP'88*, pp. 94–101 (1988).

[6] Gupta, A., Weber, W. D. and Mowry, T.: Reducing Memory and Traffic Requirement for Scalable Directory-Based Cache Coherence Scheme, *Proc. Int'l Conf. on Parallel Processing (ICPP'90)*, pp. 312–321 (1990).

[7] 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).

[8] 津田健, 山本孝伸, 田中利彦, 五島正裕, 森眞一郎, 富田眞治: メモリ・アクセスの局所性を最適化するループ再構成手法, 情処研報 99–ARC–132, 99–OS–80, 99–HPC–75, pp. 133–138 (1999).