博 士 論 文

# 高性能並列分散処理環境向け<br>ネットワークインタフェースコントローラ<br>Martiniの実装と評価

2006年度

慶應義塾大学大学院理工学研究科

渡邊 幸之介

## 論文要旨

市販の PC をノードとし，高性能なネットワークでノード間を相互接続して並列分散処理を行うクラスタコンピューティングは，安価で実用的な計算資源として，従来の大型機に代わるハイパフォーマンスコンピューティングの主要なプラットフォームとなりつつある．一般に，ハイエンドなクラスタコンピューティングは，ラックなどに高密度に集積した PC 間を Myrinet に代表される System Area Network (SAN) や高性能な Ethernet などを用いて相互接続した環境で行われる．

RHiNET は，オフィスなどにおいて日常業務で用いられている PC の余剰計算力を利用して，このようなハイエンドなクラスタコンピューティングを行うことを目的に開発された独自ネットワークである．RHiNET では，SAN に匹敵する高い通信性能と Ethernet のような接続性を両立させることで，このようなクラスタコンピューティングを実現する．

Martini は，この RHiNET 用に開発されたネットワークインタフェースコントローラである．Martini は，高い通信性能を実現するために，Remote Direct Memory Access (RDMA) を用いた通信機構をハードウェアで提供する．また，Martini は On-the-fly 通信機構と呼ばれる低遅延なパケット送出機構や，乗っ取り機構と呼ばれる協調処理機構などの実験的な機構を多数備える．

本研究では，Martini のコアロジック部および周辺のソフトウェアを中心に実装し，これを用いたシステムの構築・評価を行った．実機上で基本通信性能を評価した結果，Martini は 2 ノード間のメモリコピーにおいて 470Mbyte/sec の双方向スループットと 1.74$\mu$sec の最小レイテンシを示した．レイテンシに関しては最新鋭の SAN に匹敵する値を実現しており，RDMA をハードウェア実装したことが効果的であることが確認された．また，乗っ取り機構の有効性を検証するために，これを用いた通信機構を新たに実装した．評価の結果，乗っ取り機構の利用により，ソフトウェア単体で通信処理を行う場合に比べ高い通信性能が得られることが確認された．

並列分散処理システム下での Martini の評価は，RHiNET 上に既存のクラスタシステムソフトウェアである SCore を移植することで行った．SCore の低レベル通信ライブラリである PM はメッセージ通信を必要とするが，メッセージ通信は Martini のハードウェアに実装されていない．そこで，上位レイヤを用いて，Martini の RDMA を利用したメッセージ通信の実装を行った．評価の結果，16 ノード規模の環境においてアプリケーションレベルで台数に応じた性能向上が見られた．一方で，Martini が単純な RDMA を用いた通信しかハードウェアで提供していないことにより，システムが大規模化した際に，メッセージ通信のレイテンシが増大してしまうという問題点が明らかになった．

## Abstract

Today, a cluster computing has become a mainstream of high-performance computing platform as cost-effective and practical resource and is replacing conventional super computers. In general, a high-end cluster computing is deployed on PCs gathered in a small space like racks and they are connected to each other by System Area Network (SAN), such as Myrinet, or high-performance Ethernet.

RHiNET is an original network to provide such a high-end cluster computing by utilizing surplus computation power of PCs used in offices for daily jobs. RHiNET achieves the high-end cluster computing by supporting both SAN-like high-performance communication and Ethernet-like connectivity.

Martini is a network interface controller developed for RHiNET. To obtain high-performance communication, Martini provides simple RDMA-based communication schemes by hardware. Also, Martini has some experimental features such as low-latency packet sending mechanisms called "On-the-fly (OTF)," a new methodology for cooperation of hardware and software called "Taking Over (TO)." In the research, core hardware logic of Martini and related low-level software has been implemented. Also, a system using Martini has been built up and evaluated. The results of basic performance evaluation show that Martini achieves 470Mbyte/s maximum bidirectional throughput and 1.74 $\mu$sec minimum latency. The latency of Martini is comparable with other cutting-edge network controllers and an advantage of hardware implemented RDMA is shown. Also, to validate an effectiveness of TO mechanism, a new communication scheme which uses it has been implemented. The evaluation results indicate that TO mechanism makes communication performance better than pure software processing.

Performance of Martini under the system has been evaluated by porting existing cluster system software called "SCore." A low-level communication library of SCore, called "PM," requires message communication but is not provided by Martini. Thus, the message communication has been implemented by software using RDMA-based communication schemes of Martini. As the evaluation results on 16-node RHiNET-2 system, speed-ups were achieved according to number of nodes on an application level. Also, it was found that the latency of message communication is enlarged when the size of system becomes large. This is mainly caused by the implementation with simple RDMA-based communication schemes of Martini.

# 目 次

# 表 目 次

# 図 目 次

# 第1章　緒論

## 1.1　RHiNETプロジェクトの背景

近年のパーソナルコンピュータ (PC) の性能向上や低価格化は目ざましく，多くの企業や教育機関はサーバや個人の端末として多数の PC を導入している．これら個々の PC は，従来の大型計算機の 1 プロセッサユニット (PU) と同程度の高い処理能力を持つが，その多くは演算能力を持て余していると考えられる．たとえば，事務系の業務で用いられる端末であれば，業務時間帯であっても電子メールや文書作成などの比較的軽量な処理しか行わず，業務時間外の夜間や休日ともなると停止状態となり，演算能力を一切利用しなくなるのが一般的である．すなわち，数十台の PC が導入されているオフィスや教室などの空間には従来の数十 PU 構成の大型機に匹敵する膨大な計算力が潜在していながらも，その多くが有効活用されていないことになる．

近年では，このような余剰計算資源に着目し，これを有効活用することを目指した様々な粗粒度の分散コンピューティングが提案されている．電波望遠鏡で観測された宇宙からの電波を個人の PC の余剰計算力を利用して解析を行うことで地球外知的生命体の存在を探査する SETI@home[1] は，よく知られた粗粒度分散コンピューティングの成功例である．また，近年では，遠隔地の多数の計算資源を統合管理し，余剰計算力を用いて巨大なコンピュータシステムを実現する GRID コンピューティング [2] に関する研究が盛んに行われている．このような分散コンピューティングでは，各 PC は単体の計算機として扱われ，単体で独立して完結する問題を処理することになる．そのため，個々の PC の処理能力を超えるような大規模な問題は，独立性の高い小規模な問題に分割しない限り扱うことができない．

一方で，PC クラスタ上で行われている分散コンピューティングは，大型機のように単一の計算機として利用可能な環境を提供し，PC 単体では処理できないような大規模な問題を扱うことが可能である．PC クラスタは，ラック内などの狭い空間に集積した PC をノードとして用い，ノード間をネットワークで相互接続することで構築する，高い処理能力や信頼性を低いコストで実現する並列分散処理環境である．このような並列分散処理環境を，オフィスなどに分散配置された PC を用いて実現する場合，ノード間の接続に用いるネットワークが問題となる．

通常，科学演算などに用いるハイパフォーマンスコンピューティング向けの高性能 PC クラスタは，PC を System Area Network (SAN) と呼ばれる Myrinet[3] などの高性能なネットワークで相互接続することで構築する．SAN は，信頼性の高いリンクやカットスルールーティングに対応したスイッチなどを用いることで Ethernet などの安価なネットワークに比べ高い通信性能を提供する．SAN を用いてオフィスなどの空間に分散配置されている PC を相互接続することができれば，これら PC をノードとして PC クラスタと同様の並列分散処理環境が実現できると考えられるが，SAN は元々ラックなどに集積された PC を相互接続することを用途として設計されているネットワークであるため，最大リンク長などの制限が厳しく，物理的に接続することが困難である．

一方，コストを最重視した安価な構成の PC クラスタでは，Fast Ethernet や Gigabit Ethernet(GbE) などを用いて PC 間を接続する場合が多い．このようなクラスタシステムの中でも，Linux などのオープンなコンポーネントを利用して構築したものは特にベオウルフ型クラスタ [4] と呼ばれ，

手軽に導入可能な安価な計算資源として広く利用されている．ベオウルフ型のクラスタではノード間の通信に TCP/IP を利用するのが一般的であるが，TCP/IP はプロトコルスタックが複雑で通信処理におけるソフトウェアオーバヘッドが大きい．一般に，PC クラスタの並列処理の性能は通信オーバヘッドの影響を強く受ける傾向にある [5] ため，Ethernet を用いたクラスタでは，より高い処理性能を実現すべく PM/Ethernet[6][7] や GAMMA[8] などの TCP/IP に代わるクラスタコンピューティングに特化した軽量通信プロトコルも広く利用されている．

しかしながら，Ethernet は SAN と比べるとネットワークの信頼性が低く，ネットワークインタフェースも SAN のものよりも低機能なものが多いため，軽量な通信プロトコルを用いた場合でもホスト上での通信プロトコル処理によって SAN よりも通信オーバヘッドやレイテンシが大きくなりやすく，SAN を用いた PC クラスタに比べて性能面で劣りがちである．また，Ethernet では，スパニングツリープロトコルによってスイッチ間の論理トポロジが木構造に制限されてしまうため，スイッチ間に複数のパスを設けたとしてもこれらを有効活用することができず，複数パスを許容する SAN と比べて bi-section bandwidth を向上させにくく，SAN を用いた PC クラスタに比べてスケーラビリティ面で劣るという問題もある．このようなことから，Ethernet を用いてオフィスなどに分散配置された PC を相互接続した場合，SAN を用いたクラスタと同等の処理性能を実現することは難しく，余剰計算力を効率的に利用できないおそれがある．

## 1.2 LASN と RHiNET

新情報処理開発機構 (Real World Computing Partnership (RWCP)) を中心とした研究プロジェクトでは，オフィスなどに多数導入されている PC の余剰計算力を PC クラスタと同様の並列コンピューティングによって有効活用することを目的として，Ethernet のような LAN 環境での利用に適していながらも SAN に匹敵する高い通信性能や信頼性を提供する，Local Area System Network (LASN) と呼ばれるネットワーククラスを新たに提唱した [9]．LASN を用いることで，オフィスの端末などの分散配置された PC をノードとして利用した場合でも，SAN を用いた PC クラスタと同様に並列分散処理において高い処理能力を実現できるものと考えられる．図 1.1に LASN のイメージ図を示す．

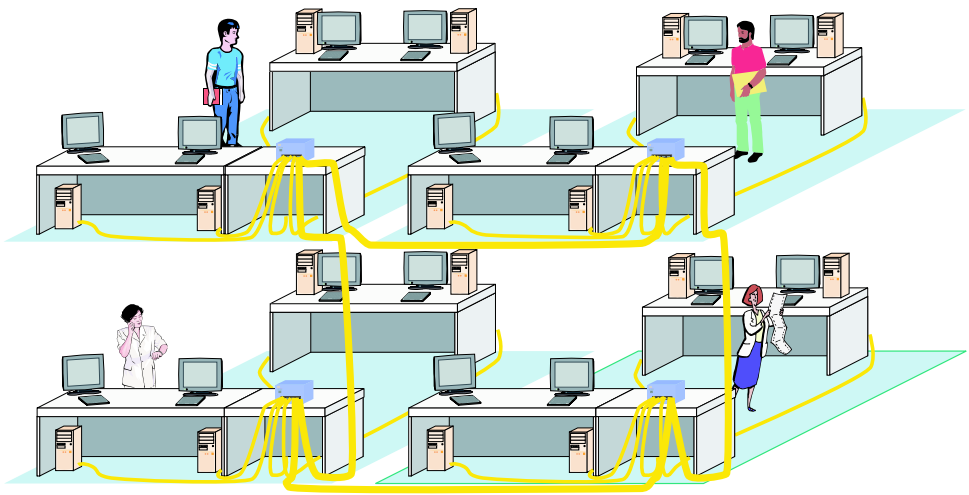


図 1.1 LASN のイメージ図

このようなネットワーククラスに属するネットワークとして，前出の RWCP を中心としたプロジェクトでは **R**WCP **Hi**gh Performance **Net**work (RHiNET) と呼ばれるネットワークを提案し，開発を行った [9][10]．以下，同プロジェクトを RHiNET プロジェクトと称する．

RHiNET は，独自のスイッチと光媒体のリンクを用いることで，数百 m から 1km 程度の最大リンク長，任意のトポロジでのデッドロックフリーなパケット転送，カットスルー方式による低遅延なスイッチング，ハードウェアによる通信信頼性の保証などの特徴を備えた結合網を提供するネットワークである．また，RHiNET では，ノード間の通信においてこのような結合網の性能を十分に活かせるように独自のネットワークインタフェースを用い，ノード間のメモリコピーを基本とする通信機構をハードウェアで実現することで，従来の SAN のネットワークインタフェースと同等以上の通信性能の実現を目指す．

## 1.3 RHiNET プロジェクトの経緯

RHiNET プロジェクトは，RWCP の掲げる “シームレス並列分散コンピューティング環境” の一環として，コンピュータ間でメモリやファイルシステムなどの計算資源を透過的に共有し，スケーラブルで柔軟な，耐故障性を備えたシステム [11] を実現するためのネットワークアーキテクチャを目指すプロジェクトとして 1997 年に発足した(注 1)．

1998 年頃よりハードウェアの開発が本格的に開始し [12][13]，翌 1999 年にはプロトタイプのスイッチである RHiNET-1/SW (発表当時は MLC-1 Network Router)[14] とネットワークインタフェース RHiNET-1/NI (発表当時は MLC-1) [15][16] の実装が行われた．RHiNET-1/SW は当初から Application Specific Integrated Circuit (ASIC) 実装されていたが，RHiNET-1/NI は各種プロトコルの試行実験を行うことを考えコントローラ部が Complex Programmable Logic Device (CPLD) 上に実装されており，PCI バスコントローラや光リンク側の制御を行うリンクコントローラは外部チップとして独立していた．また同年には，RHiNET-1/SW を強化した RHiNET-2/SW の実装が行われ [17]，翌 2000 年には RHiNET-2/SW とアプローチの異なる 3 世代目のスイッチである RHiNET-3/SW の開発も開始された [18].

ネットワークインタフェースに関しては，2000 年頃に，RHiNET-2/SW の検証を目的として，RHiNET-1/NI をリファインして RHiNET-2/SW に対応させた RHiNET-2/NI0 が開発された．また，同年にはこれと並行して ASIC 実装のネットワークインタフェースコントローラである Martini の開発も開始された．Martini は，RHiNET-2/SW や RHiNET-3/SW に対応するネットワークインタフェースコントローラである．Martini の開発にあたり，RHiNET プロジェクトでは RHiNET のネットワークインタフェースの設計方針を従来の RHiNET-1/NI や RHiNET-2/NI0 から大きく変更した．そのため，Martini は，これらの設計資産を流用せず，新たに一から設計し直されることになった．また，プロジェクトの事情などから Martini にはメモリバスを利用してホスト PC に接続するためのインタフェースなどの実験的な機能がいくつか搭載されることになった．Martini の開発には RWCP，慶應義塾大学 (慶大)，日立インフォメーションテクノロジー (日立 IT) が参加した．

2001 年には，2 度に渡り Martini がチップ化された．最初のチップは PCI インタフェース部が満足に動作せず，ほとんどネットワークインタフェースとして機能しなかった．その後，設計の修正を行った 2 度目のチップは，動作はしたものの，PCI コントローラに不具合が残存しており，連続してデータ転送などを行うことができないなど，アプリケーションレベルでの性能評価には

(注 1)プロジェクト発足当初，ネットワークは RHiNET ではなく Memory-based Light-weight Communication mechanism (MLC) と呼ばれていた．

程遠い状況であった．また，同年には RHiNET-3/SW の実機もできあがったが，スキュー調整のための外部チップである Deskew-LSI[19] に不具合があったため，光リンクを用いた通信さえも行うことができなかった．

2002 年の 3 月には，プロジェクトの母体であった RWCP が解散し，ハードウェアなどの新規開発が停止となった．この時点で RHiNET-3/SW や Martini は満足に動作せず，RHiNET-2/SW は RHiNET-2/NI0 と組み合せることで動作はするものの，安定性に欠け，また RHiNET-2/NI0 自体が低性能であったため [20]，RHiNET の目標のひとつである高性能な通信の実現には程遠い状況であった．RWCP 解散後は，RHiNET 関連の資産や研究を慶大が引き継ぎ，日立 IT と共同して同年秋に 3 度目のチップ化が行われた．この年，RHiNET-2/SW を用いた 64 ノード構成の RHiNET-2 クラスタが慶大に構築され，PM/RHiNET の実装により RHiNET-2 上で SCore クラスタシステムソフトウェア [21][22] が動作するようになった．これにより，ようやく本格的なアプリケーション実行環境が実現する運びとなった．

2003 年以降，RHiNET-2 システムはルーティングやトポロジなどの実機評価 [23][24] や実アプリケーションによる評価 [25]，並列プログラミングの教材などに用いられた．また，その傍ら，ライブラリのチューニングや安定化作業などが行われた．ネットワークインタフェースが満足に動作するようになったことで，スイッチに本格的な負荷をかけられるようになり，その結果，Martini や RHiNET-2/SW のこれまで知られていなかった不具合などが新たに発見された．特に，ネットワークが中程度に混雑した場合に，フロー制御に失敗し，スイッチでパケットが詰まったり消失したりしてしまう RHiNET-2/SW の不具合の発見によって，大規模なアプリケーションを安定して動作させることが極めて困難であることが明らかになった [26].

2005 年になると，維持管理面の問題で RHiNET クラスタは解体され，以後は小規模な環境での評価が続けられた．

## 1.4 Martini における筆者の貢献

筆者は 2000 年に Martini の開発に加わった．筆者が参加した時点で，既に Martini のアーキテクチャの大まかな方針は決定していた．筆者は Martini を構成するブロックのうち，オンチッププロセッサとプロトコル処理を行うハードウェアモジュールを中心とした部分の開発に関わり，シミュレーションによる論理検証や設計上の不具合の調査と修正，性能の予備評価などを担当した．また，オンチッププロセッサによるソフトウェア処理の性能を改善するために，乗っ取り機構と呼ばれるハードウェア・ソフトウェアの協調処理を新たに提案し，プロトコル処理を行う複数のハードウェアモジュールに対してこれを組み込むことを行った [27][28].

2002 年に RWCP が解散した後は，筆者はまず 3 度目のチップ化のための論理検証やテストベクタ作成などのチップ実装作業を担当し，その後，3 度目のチップ化で完成した Martini (Martini 3rd) を用いたシステムの構築を行った．先に述べたように，これより以前にチップ化された Martini には PCI コントローラを中心に不具合が多数あり，PC に接続して運用した場合，全力で稼働させることができず，ハードウェアが本来備えるべき能力を抑えた上での小規模かつ部分的な性能評価しか行うことができなかった．また，このようなことから，Martini 3rd が完成した時点ではソフトウェア面の整備がほとんどされておらず，実用的な環境で評価を行うにあたり，ファームウェアやデバイスドライバ，ユーザライブラリなどの整備を行う必要があった．このような状況に対し，筆者は Martini のハードウェアの不足分を補う低レベルソフトウェアライブラリの設計・実

装を行い，実機上で実用的なネットワークインタフェースとして利用可能な環境の構築を行った [29]．さらに，これらを利用して，実機上でシステムを構築し，これまで十分に行われてこなかった実機上での Martini の基本通信性能の評価とその解析を行い，その有効性や問題点を明らかにした [30]．また，Martini の設計段階で提案した乗っ取り発機構に関して，これを利用した VPUSH と呼ばれる通信機構の実装を行い，その有効性を示した [27][28]．

その後は，不安定要因の調査や不具合回避のためのライブラリの調整などを行い，システムレベルでの評価などを行った．その際，既存のクラスタシステムソフトウェアである SCore を RHiNET 上に移植することで並列分散処理システムを実現することにしたが，SCore の移植で必要となる低レベルな通信ライブラリである PM [7] は Martini が提供していないメッセージ通信を必要とするため，Martini の基本通信機構を用いた PM/RHiNET と呼ばれるメッセージ通信の実装を新たに行った．また，その後，スケーラビリティ面で問題のある PM/RHiNET の設計を見直し，PM/RHiNET-VP と呼ばれる VPUSH を利用したメッセージ通信を新たに提案・実装した．

## 1.5 本稿の構成

本稿では，RHiNET プロジェクトにおいて Martini を中心に筆者の関わった部分の詳細を示し，これらより得られた知見についてまとめる．

以降，まず第 2章において本研究の母体である RHiNET プロジェクトについて述べ，続く第 3章において本研究で扱う内容と関連性の深い技術・研究について示す．

次に，第 4章から第 6章にかけて，本研究において設計・実装を行った Martini の通信処理部および Martini 向けの低レベルソフトウェアライブラリについて示し，これを利用した Martini の実機上での基本性能の評価に関して述べる．まず第 4章にて実装のベースとなる Martini そのもののアーキテクチャの設計・実装について述べ，次に第 5章で低レベルソフトウェアライブラリの設計および実装について詳細を述べる．その後，第 6章において，実機上で行った Martini の基本性能の評価を示し，基本通信機能の有効性や問題点に関して検討する．

続いて，第 7章では，本研究において筆者が提案・実装したハードウェア・ソフトウェア協調処理機構である乗っ取り機構について詳細を述べ，その応用の実装や評価についてまとめる．

その後，第 8章にて，Martini 向けの PM のメッセージ通信の 2 種類の実装およびこれらを用いたアプリケーションレベルでの性能評価の結果を示す．

最後に，9章にて本研究によって得られた知見などについてまとめる．

本稿の第 4章から第 8章までの各章で述べる内容の関係を図 1.2に示す．また，本研究の要点を表 1.1にまとめて示す．

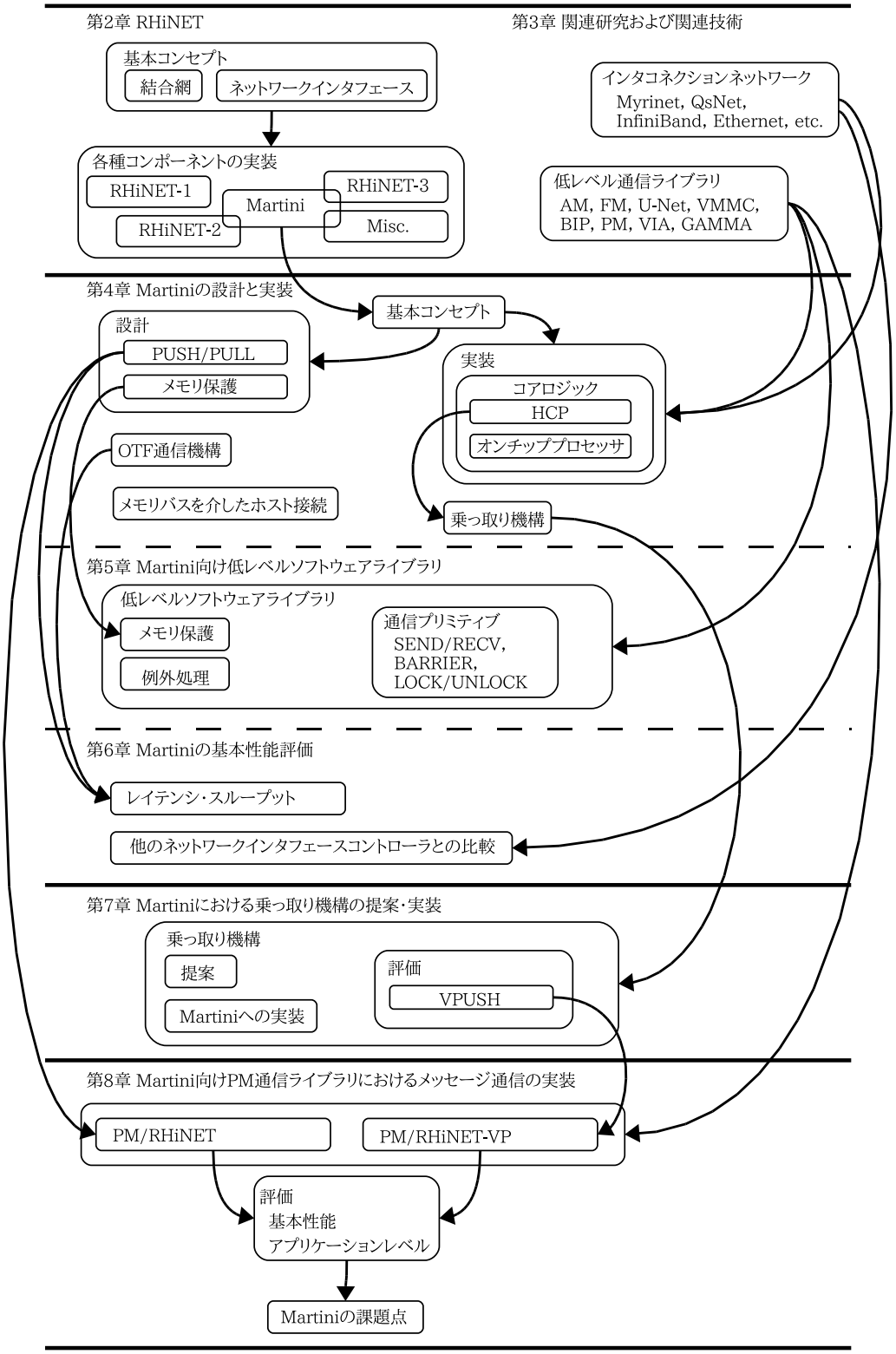

第2章 RHiNET
第3章 関連研究および関連技術
基本コンセプト
結合網
ネットワークインタフェース
インタコネクションネットワーク
Myrinet, QsNet,
InfiniBand, Ethernet, etc.
各種コンポーネントの実装
RHiNET-1
RHiNET-2
Martini
RHiNET-3
Misc.
低レベル通信ライブラリ
AM, FM, U-Net, VMMC,
BIP, PM, VIA, GAMMA
第4章 Martiniの設計と実装
基本コンセプト
設計
PUSH/PULL
メモリ保護
実装
コアロジック
HCP
オンチッププロセッサ
OTF通信機構
メモリバスを介したホスト接続
乗っ取り機構
第5章 Martini向け低レベルソフトウェアライブラリ
低レベルソフトウェアライブラリ
メモリ保護
例外処理
通信プリミティブ
SEND/RECV,
BARRIER,
LOCK/UNLOCK
第6章 Martiniの基本性能評価
レイテンシ・スループット
他のネットワークインタフェースコントローラとの比較
第7章 Martiniにおける乗っ取り機構の提案・実装
乗っ取り機構
提案
Martiniへの実装
評価
VPUSH
第8章 Martini向けPM通信ライブラリにおけるメッセージ通信の実装
PM/RHiNET
PM/RHiNET-VP
評価
基本性能
アプリケーションレベル
Martiniの課題点

図 1.2　章間の関係

表 1.1　本研究の要点

| | | | |
|---|---|---|---|
| Martini の設計・実装・評価 | 4–6章 | 目的 | プロセス間通信に高い通信性能を低オーバヘッドで提供するネットワークインタフェースコントローラ Martini の開発 |
| | | 取り組み | 基本通信機能である RDMA を用いたリモートメモリライト・リードをハードウェアで実装し，それ以外の処理をソフトウェア実装 |
| | | 成果 | 1.75$\mu$sec の，最新のネットワークインタフェースに匹敵する低いレイテンシおよび 470Mbyte/s の 64bit/66MHz の PCI バスの性能を十分に活かした双方向スループットを実現した |
| 乗っ取り機構の提案・実装 | 7章 | 目的 | ハードウェアと同じような処理を行うことの多い Martini のソフトウェア処理を，ソフトウェア処理時に遊休状態となるハードウェアを活用して効率化 |
| | | 取り組み | ハードウェアをモジュール単位で安全に停止させてソフトウェアの制御下に置く乗っ取り機構を提案し，これを Martini の一部のモジュールに対して適用 |
| | | 成果 | ソフトウェアによる例外処理や通信プロトコル処理が実際に効率化することが確認された |
| Martini 向けの PM のメッセージ通信の実装 | 8章 | 目的 | クラスタシステムソフトウェア SCore を稼働させるために，SCore が必要とする PM 通信ライブラリのメッセージ通信の機能を Martini 上で実現 |
| | | 取り組み | Martini のリモートメモリライト・リードやオンチッププロセッサを利用してメッセージ通信を実装 |
| | | 成果 | アプリケーションベンチマークで 16 ノード構成までならノード数に応じた性能向上示すことが確認されたが，ノード数が増大した場合に現状の Martini では通信性能の低下を避けられないことが確認された |

# 第2章　RHiNET

本研究の主題であるネットワークインタフェースコントローラ Martini は，第 2 世代および第 3 世代の RHiNET 向けのネットワークインタフェースコントローラとして開発した Application Specific Integrated Circuit (ASIC) である．Martini の大まかな設計方針は RHiNET プロジェクト全体のコンセプトに基づいて決定されている．本章では，Martini における各種機能や設計の背景を明らかにするために，RHiNET プロジェクト全体に関する基本的なコンセプトおよび RHiNET プロジェクトにおいて実装された各種コンポーネントについて概要を示す．

なお，1.3節でも述べたように，RHiNET プロジェクト自体は 1997 年より始動しており，これらコンセプトはその時点での状況を元に提案・確立されたものである．また，筆者自身は，以下で述べるコンセプトの提案や確立には関わっておらず，これらの確立に至る過程でどのような技術検討や議論が行われたのかについては明らかでない．

## 2.1　RHiNET の結合網

1章で述べたように，RHiNET プロジェクトにおいて提案された LASN は，以下の 2 点を満たすネットワークである．

- オフィスなどのフロアレベルの範囲に不規則に分散配置された PC 間の接続
- SAN と同程度の高い通信性能

このような LASN の要件に対して，RHiNET では，以下に示すアプローチで結合網を構築して対応する．

- 光リンクの採用

  オフィスなどの空間に分散配置された PC 間を接続するためには，少なくとも数百メートル程度の伝送距離に対応することが要求される．また，SAN と同程度の通信性能が必要となることから，通信媒体は数 Gbps クラスの高いビットレートでのデータ転送に対応したものでなければならない．このような要求を満たす通信媒体として，RHiNET では光リンクを採用する．

- 縮約構造化チャネル法によるデッドロックの回避

  一般に，ネットワーク上でのデッドロックの検出や，デッドロックしたパケットの破棄に伴う再送処理はコストが大きく通信性能を悪化させる．SAN では，循環構造を回避する経路選択 (デッドロックフリールーティング) を行うことでデッドロックを回避することが行われている．しかし，RHiNET の想定するオフィス内などの規則的なスイッチ間接続が難しい環境では，イレギュラーネットワークが構築される可能性が高く，このようなネットワーク

にデッドロックフリールーティングを導入した場合，非最短経路の利用やトラフィックの偏りの発生などの問題が生じ，ネットワークの性能を十分活かせなくなる恐れがある．

構造化チャネル法 (Structured Buffer Pool)[31] はネットワークの最大直径程度の数の仮想チャネルを設け，スイッチを経由するたびにパケットの転送に利用する仮想チャネルの値を大きくしていくことで，任意の論理トポロジ上で最短経路でデッドロックを伴わないルーティングを実現する方式である．この方法では，ルーティング時に経由するスイッチ数がハードウェアで用意されている仮想チャネル数以下である限り循環依存が生じないため，デッドロックフリーであることが保証される．

RHiNET では，この構造化チャネル法を改善した縮約構造化チャネル法 [32] を新たに提案し，スイッチに実装する．縮約構造化チャネル法は，“ネットワークにおいてスイッチ間接続リンクを 3 つ以上持たないスイッチではチャネル番号を増やさなくともデッドロックフリーが保証される” という理論に基づき，スイッチ間接続リンク数が 2 以下のスイッチではパケットの使用するチャネルの値を大きくしないことで，必要な仮想チャネル数の節約を行う方式である．

- バーチャルカットスルースイッチング

  Ethernet のスイッチやルータで用いられている Store-and-Forward 方式のパケット転送は，スイッチにおいて，入力されたパケットがすべてスイッチ内のバッファに収まるまで待ってから宛先へ向けた転送が開始する方式であるため，スイッチを経由するたびに大きな通信遅延が生じてしまう．

  これに対し，SAN などの高性能なスイッチは，バーチャルカットスルー方式のパケット転送を採用している．バーチャルカットスルー方式のパケット転送では，パケットの入力が開始した時点でパケットヘッダの解析を行い，パケット本体の到着を待たずに宛先へ向けた転送を開始する．RHiNET においても他の高性能なネットワークと同様に，スイッチでのパケット転送による通信遅延を削減するためにバーチャルカットスルースイッチングを採用する．

- ネットワークレベルでの通信信頼性の保証

  下位の通信層の信頼性が低い場合，上位レイヤによるパケットの並べ替えや再送処理などが必要となるが，これらはノード上で通信オーバヘッドとして現れる．RHiNET は，ネットワークの下位レイヤにおいて，ハードウェアで通信の信頼性を保証することで，上位レイヤでのパケットの並べ替えや再送処理などを不要とし，このような通信オーバヘッドを回避する．

## 2.2 RHiNET のネットワークインタフェース

RHiNET では，独自開発のスイッチと光リンクの採用により，通信性能や信頼性が高く長距離リンクや任意のトポロジに対応した結合網を提供する．しかし，結合網上の通信性能がいくら高くても，この結合網に接続されるノード PC の通信オーバヘッドが大きく結合網の通信性能を十分に活用できないようでは意味がない．

RHiNET の結合網は信頼性が高いため，ホスト PC がパケットのエラー検出や訂正，再送処理，並べ替えなどを行う必要がなく，このような処理に伴うホストオーバヘッドは回避できる．しかしながら，アプリケーション間の通信においてボトルネックとなるカーネルの呼び出しや転送デー

タのメモリ間コピー，データ受信時の割込みによるオーバヘッドなどは結合網の工夫では回避することができない．そこで，RHiNET では，独自のネットワークインタフェースを用い，ユーザレベル通信およびゼロコピー通信に対応した通信処理をハードウェアで提供することで，ノード PC 上のアプリケーション間の通信に高い通信性能を提供する．

### 2.2.1 ユーザレベル通信・ゼロコピー通信

RHiNET のネットワークインタフェースは，既存のクラスタ向け低レベル通信ライブラリの多くで導入されているユーザレベル通信 [33] [34] [35] およびゼロコピー通信 [36] を提供する．

- ユーザレベル通信

  ユーザレベル通信は，ネットワークインタフェースのメモリなどをユーザ空間に直接マップし，ユーザプロセスが通信要求を書き込むことで通信処理を起動する通信方式である．

  一般的に，TCP/IP などは，ユーザが通信処理を起動するにあたり，OS の提供するシステムコールを呼び出す実装となっている．これにより，通信処理に OS が介在することになるため，不正なパケット発行の防止やハードウェアの仮想化・多重化などが実現するが，一方で，ユーザが通信処理を起動するたびにシステムコールによるコンテキストの切り替えが発生するため，ホストのオーバヘッドや通信レイテンシが増大するという問題がある．ユーザレベル通信を用いることで，通信における OS の介在が不要となるため，これら問題が解消する．

- ゼロコピー通信

  ゼロコピー通信は，ネットワークインタフェースが直接ユーザプロセスのメモリ領域を読み書きすることで，データの送受信処理におけるメモリ間のデータコピーを回避する通信方式である．

  通常，Ethernet などのネットワークインタフェースでは，メモリ保護などの観点からネットワークインタフェースがアクセスする通信バッファはカーネルメモリ上に確保される．ユーザプロセスがデータを送信する場合，ユーザメモリ上の送信データは OS などによって一旦カーネルメモリ上の通信バッファにコピーされ，その上でネットワークインタフェースによって DMA で読み出される．また，ユーザプロセスがデータを受信する場合も，ネットワークインタフェースはまず受信データをカーネルメモリ上の通信バッファに DMA で書き込み，その上で OS などによって通信バッファのデータがユーザプロセスの受信領域にコピーされる．このような場合，送信側と受信側でそれぞれ 1 回ずつメモリ間コピーが発生することになるが，メモリ間のデータコピーは CPU によって行われるため，ホストの通信オーバヘッドの大きな要因となる．また，メモリ間コピーはメモリバスを占有してしまうため，ネットワークインタフェースのメモリへのアクセスとパイプライン動作させることもできずプロセス間の通信におけるスループットを制限する要因ともなる．ゼロコピー通信を導入することで，カーネル上の通信バッファとユーザメモリとの間でのデータのコピーがなくなるため，これらの問題が解消する．

ユーザレベル通信およびゼロコピー通信を実現するには，ネットワークインタフェースやカーネルなどで以下の機能を提供する必要がある．

- ユーザレベルでアクセス可能な I/O 領域

ユーザレベル通信では，ネットワークインタフェースのメモリやレジスタなどをユーザプロセス自身のアドレス空間にマップし，これに対して通信要求を直接書き込むことで通信処理を起動する．その際，ユーザプロセスにネットワークインタフェースのすべてのレジスタをマップしてしまうと，ユーザがネットワークインタフェースの動作設定のためのレジスタなどにアクセスできるようになってしまい危険である．そこで，ネットワークインタフェースにユーザプロセス向けの通信要求用の I/O 領域を別途設け，ユーザプロセスが制御用のレジスタをマップしなくても通信要求を発行できるようにする必要がある．

また，ユーザレベル通信を利用することで，OS によるネットワークインタフェースの多重化が行えなくなるため，複数のユーザプロセスが同時にネットワークインタフェースを利用する場合に備え，ユーザプロセス向けの I/O 領域はユーザプロセス間の通信要求の干渉を防止可能な構造とすべきである．

- 通信領域のピンダウン・アンピンダウン

  ゼロコピー通信では，ネットワークインタフェースはユーザのアドレス空間上に確保された通信領域に対応する物理メモリに対して DMA でアクセスを行うが，一般的な OS では，ユーザが確保したメモリ領域に対して常に物理メモリが割当てられている保証はない．たとえば，`malloc` などのシステムコールを用いてメモリを確保した場合，メモリにデータの書き込みが行われるまで物理メモリが割当てられない可能性がある．また，物理メモリの不足などが発生すると，ユーザのメモリがページアウトを起こして物理メモリを解放し，外部記憶装置に退避されてしまう可能性もある．

  このような事態の発生によってネットワークインタフェースがユーザメモリに対して DMA を行うことができなくなることを防ぐために，ユーザメモリ上の通信領域を確実に物理メモリ上に固定するピンダウン処理および固定を解除するアンピンダウン処理が必要となる．

- 仮想 – 物理アドレス変換

  DMA を行う場合，ネットワークインタフェースは DMA 対象となる領域の物理アドレスを知る必要があるが，ユーザレベル通信では通信処理の要求をユーザレベルで発行することになるため，通信領域は仮想アドレスで指定することになる(注 1)．そのため，ネットワークインタフェースは，ユーザからの通信要求を受け付けた際に，ユーザ指定の通信バッファの仮想アドレスに対応する物理アドレスを獲得する必要がある．このアドレス変換をホストに割込みをかけて OS を呼び出して行うのは本末転倒であることから，ネットワークインタフェース内部で仮想 – 物理アドレス変換を行うべきである．

- 不正な通信の防止

  ユーザレベル通信を行う場合，ユーザが通信処理を起動するため，OS を利用してユーザからの通信要求を検閲して不正なパケットの送信などを防止することができない．

  ゼロコピー通信ではリモートのユーザプロセスの通信領域に対して直接データの書き込みや読み出しを行うが，不正な通信要求を発行することで任意のリモートのユーザプロセスのメモリ領域にアクセスできてしまうと，リモートノードでのメモリ保護が破綻してしまう．

(注 1) デバイスドライバなどにカーネル空間のページテーブルにアクセスする機能を設けることで，ユーザが通信に先立って自身の通信領域の物理アドレスを取得することが可能となるが，これを用いてネットワークインタフェースに指示を出すようにしてしまうと，ユーザに任意の物理ページへのアクセスを許容することになり OS のメモリ保護機構が破綻してしまう．

リモートの任意の領域へのアクセスは，リモートにおけるネットワークインタフェースのアドレス変換機構を用いて防ぐことができるが，プロセス間の通信に関しては別途ポリシーを決め，不正なプロセス間通信を防ぐための仕組みをネットワークインタフェース上に用意しなければならない．

#### 通信の保護ポリシー

RHiNET では，ネットワークインタフェースを利用するプロセスの間で無制限に通信を許可せずに，同一の並列ジョブに属する並列プロセスの間でのみ通信を許可するポリシーとする．これにより，他の並列ジョブに属するプロセスに対する通信要求の発行を防止する機構をネットワークインタフェースに設ける必要が生じる．なお，本稿における “並列ジョブ” とは，“あるユーザによってシステム上での実行が指示された並列処理” を指す．並列ジョブは，並列して動作する複数のプロセスの集合体としてシステム上に存在する．

一方，同じ並列ジョブに属するプロセス間の通信に関してはすべてユーザの責任のもとで保護を行うものとし，ネットワークインタフェースによる保護は提供しない．したがって，存在しないプロセスに対する通信要求の発行などをネットワークインタフェースで防止するような仕組みは設けないものとする．

### 2.2.2 基本通信処理のハードウェア実装

RHiNET のネットワークインタフェースは，基本的な通信処理を，不正なパケット受信時の処理などの一部の例外的な状況を除いて完全にハードウェア実装して上位レイヤに提供する．これにより，Myrinet のようにソフトウェアでプロトコル処理を行う場合に比べてネットワークインタフェース上での通信処理効率を高め，より高性能な通信を実現する．

#### RHiNET のプリミティブ

一般に，ユーザがライブラリなどを介して呼び出し可能な通信処理の最小単位はプリミティブと呼ばれる．RHiNET のネットワークインタフェースが提供すべきプリミティブの一例を以下に示す．

- PUSH: リモートメモリライト
- PULL: リモートメモリリード
- ISEND/IRECV: 非ブロッキング型のメッセージ通信
- MCAST: マルチキャスト
- LOCK/UNLOCK: プロセス間での排他制御処理
- BARRIER: プロセス間での同期処理

RHiNET の通信モデルは，リモートプロセスとローカルプロセスとの間のメモリ間コピーである Remote Memory Access (RMA) を通信処理の基本とする．したがって，これらの中では PUSH

と PULL が最も重要なプリミティブとなる．また，PUSH，PULL，ISEND/IRECV，MCAST などのプリミティブでは，メモリ上の連続したデータの単純な転送だけでなく，TWIN/DIFF 方式で抽出した差分のみの転送 [37] や，ユーザ指定のビットマップやストライドによる部分転送などの，分散共有メモリ型のシステムの構築を支援するモードを選択して利用することも可能とする．

RHiNET のプリミティブは関数呼び出しなどの形でユーザが明示的に起動するモデルを採用する．ネットワークインタフェースは，ユーザレベルでプリミティブを起動できるよう，プリミティブ起動用の I/O 領域を用意する必要がある．

## 2.3 RHiNET の実装

以下では RHiNET プロジェクトにおいて実装された，RHiNET の各種コンポーネントについて述べる．

### 2.3.1 RHiNET-1

RHiNET-1 は RHiNET のプロトタイプとして最初に実装されたネットワークである．RHiNET-1 は，専用スイッチ RHiNET-1/SW[38] と専用ネットワークインタフェース RHiNET-1/NI[16] で構成され，これらの間を光リンクで接続する．

#### RHiNET-1/SW

RHiNET-1/SW は，富士通社の 0.35$\mu$m プロセスの CMOS エンベデッドアレイによる 1 チップの ASIC スイッチ LSI と大容量の外部 SRAM で構成される．RHiNET-1/SW は，外部 SRAM をパケットバッファとして用い，チップ内部のメモリでこれをキャッシュする仮想チャネルキャッシュ方式 [32] を採用している．また，チップ内に 8×8 のクロスバを内蔵し，光インタコネクションモジュールを 8 組接続可能である．フロー制御には Stop-and-Go 方式を採用し，各リンクの速度は 1.33G+1.33Gbps となっている．

#### RHiNET-1/NI

RHiNET-1/NI は，RHiNET-1/SW に接続可能なネットワークインタフェースである．32bit/33MHz の PCI バスを介したホストとの接続に対応し，PCI コントローラは QuickLogic 社の Field Programmable Gate Array (FPGA) 上に実装されている．また，スイッチとの間の接続に用いるリンクコントローラも同様に QuickLogic 社の FPGA を用いて実装されており，これらの間の制御はリコンフィギャラブルデバイスである ALTERA 社の Complex Programmable Logic Device (CPLD) 上に実装されたコントローラによって行われる．

RHiNET-1/NI では，低レイテンシで高スループットの通信を実現するために多数の基本通信命令をネットワークインタフェース上でハードワイヤード実装している．

#### RHiNET-1 の評価

RHiNET-1 では PUSH や PULL といった基本的な通信プリミティブにおいて，4Kbyte 転送時のノード間のスループットが 35Mbyte/s，16byte 転送時の RTT が 25$\mu$sec という性能が得られた [39]. スループットに着目すると，この値は PCI バスや光リンクのデータ転送能力に比べて非常に低い値であり，RHiNET の要求に対して不十分である．十分なスループットが得られない理由には，コントローラが実装されている CPLD のデバイス能力や，ボード設計などの問題から，コントローラ部の動作周波数が PCI の動作周波数である 33MHz に到達できなかったことが挙げられる．

### 2.3.2 RHiNET-2

RHiNET-2 は，第 2 世代の RHiNET であり，専用スイッチ RHiNET-2/SW，専用ネットワークインタフェース RHiNET-2/NI および光リンクで構成される．

RHiNET-2 では，PC における PCI バスなどの汎用 I/O バスの高性能化に備え，RHiNET-1 では 1.33G+1.33Gbps であったリンク速度を 8G+8Gbps(注 2)にまで強化している．

また，RHiNET-2 ではエラー発生率が低くスキューも小さい高品質な光リンクを用い，すべてのフリットに対して Error Correcting Code (ECC) を付加することで，リンクをエラーフリーな状態とし，パケットの破損などによる上位の通信層での再送処理を不要としている．さらに，パケットの再送がなくなることで，2 点間でのルーティングに固定の経路を用いる限りパケットの到着順序が入れ替わることがなくなるため，ノードにおける受信後のパケットの並び替えも不要としている．このように，RHiNET-2 では信頼性の高い通信路を物理層で実現することで，上位層の負担を軽減する点が大きな特徴となっている．

#### RHiNET-2/SW

RHiNET-2/SW は 0.18$\mu$m プロセスの CMOS エンベッデッドアレイで構成される 1 チップスイッチである．RHiNET-1/SW の設計をベースとしているため，基本的な部分は RHiNET-1/SW と同様であるが，ポートごとに 1G+1Gbps，2G+2Gbps，8G+8Gbps の速度の異なるリンクを混合して利用可能な点が異なる．また，チップ内部に大容量の SRAM を持つようになったことで RHiNET-1/SW で用いていた外部メモリを不要とし，さらに大容量のスラックバッファを確保したことで最大で 200m 程度のリンク長に対応する．ルーティング方式には，宛先に応じて出力ポートが決まる固定ルーティングを採用している．

図 2.1に RHiNET-2/SW の外観を，図 2.2に RHiNET-2/SW の基板を示す．

#### RHiNET-2/NI

RHiNET-2/NI は，64bit/66MHz の PCI バスを介してホストに接続されるネットワークインタフェースである．ネットワークインタフェース上にネットワークインタフェースコントローラ Martini，SDRAM，RHiNET-2 の光リンクに対応した光インタコネクションモジュールを備える．

Martini は単体で RHiNET-2 および後述の RHiNET-3 の両方のリンクに対応するネットワークイ

(注 2)エラー訂正のための Error Correcting Code (ECC) やフリット識別子などが付加されるため，ノード間の理論上の最大ビットレートは 4/5 の 6.4G+6.4Gbps となる．

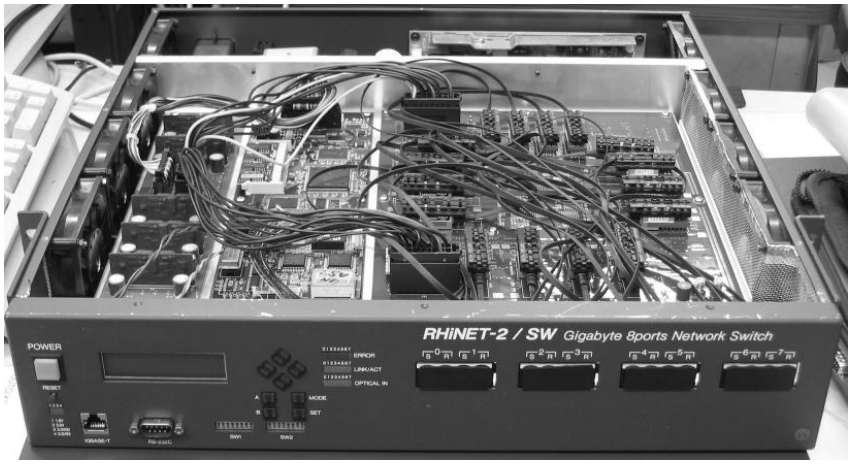



図 2.1　RHiNET-2/SW の外観

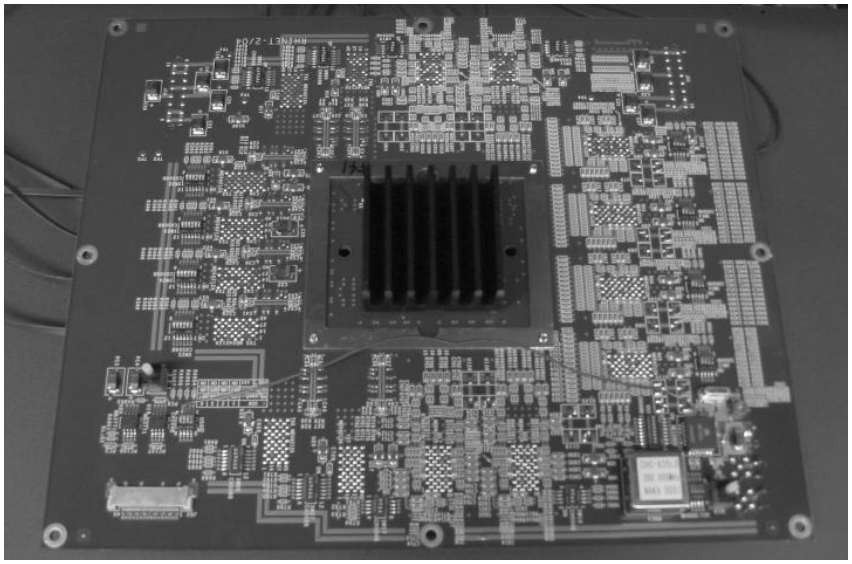


図 2.2　RHiNET-2/SW の基板

ンタフェースコントローラである(注 3)．64bit/66MHz 規格の PCI バスを介してホスト PC に接続可能な他，DIMM スロットを介したホスト PC への接続にも対応している．

Martini は，RHiNET-1 のコントローラと異なり ASIC 実装されている．また，ハードウェアで提供するプリミティブは PUSH と PULL のみに限定し，それ以外のプリミティブは上位レイヤを組み合せて実現する方針をとっている．Martini に関する詳細は 4章にて述べる．

図 2.3に RHiNET-2/NI の外観を示す．

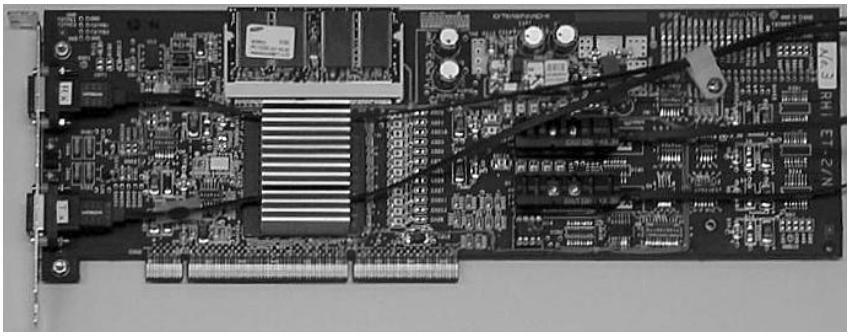


図 2.3　RHiNET-2/NI の外観

### 2.3.3　RHiNET-3

RHiNET-3[40] は，第 3 世代の RHiNET であり，専用スイッチ RHiNET-3/SW，専用ネットワークインタフェース RHiNET-3/NI および光リンクで構成される．

RHiNET-3 は，RHiNET-2 に比べてより実用的な LASN の実現を目標としたネットワークである．RHiNET-1 や RHiNET-2 で用いられていた Stop-and-Go 方式のフロー制御をやめ，代わりに伝送フリット単位で Hop-by-Hop でのクレジットベースのフロー制御を行うことで，大容量のスラックバッファを不要とし，LASN の想定する 1km 程度のリンク長に対応する．また，これとあわせて Hop-by-Hop でのフリット単位での Cyclic Redundancy Check (CRC) によるエラー検出とシーケンス番号を用いた再送機構をハードウェアで実現することで，安価で標準的な光リンクを用いた通信にも対応する．安価な光リンクを用いた場合，スキューが問題となるが，これについては Deskew-LSI[19] と呼ばれる専用の LSI をスイッチやネットワークインタフェースに搭載することでスキュー調整を行う．

#### RHiNET-3/SW

RHiNET-3/SW[41] は 0.14$\mu$m CMOS エンベッデッドアレイ ASIC で構成される 1 チップスイッチであり，10+10Gbps (注 4)のリンクに対応する．

RHiNET-3/SW は RHiNET-3 の特徴であるクレジットベース方式のフロー制御やハードウェアによるエラー検出および再送機構を搭載する．また，ルーティング方式としては，RHiNET-2/SW と同様に宛先による固定ルーティングに対応する他，ソースルーティングにも対応している．

(注 3)NEC 社の Optical-interconnection Intellectual Property (OIP) と呼ばれる技術を用いたスイッチにも対応している．
(注 4)CRC やシーケンス番号の負荷により 64bit のフリットが 80bit に拡張されて転送されるため，ノード間の理論上の最大ビットレートは 4/5 の 8G+8Gbps となる．

図 2.4に RHiNET-3/SW の外観を，図 2.5に RHiNET-3/SW の基板を示す．

図 2.4　RHiNET-3/SW の外観

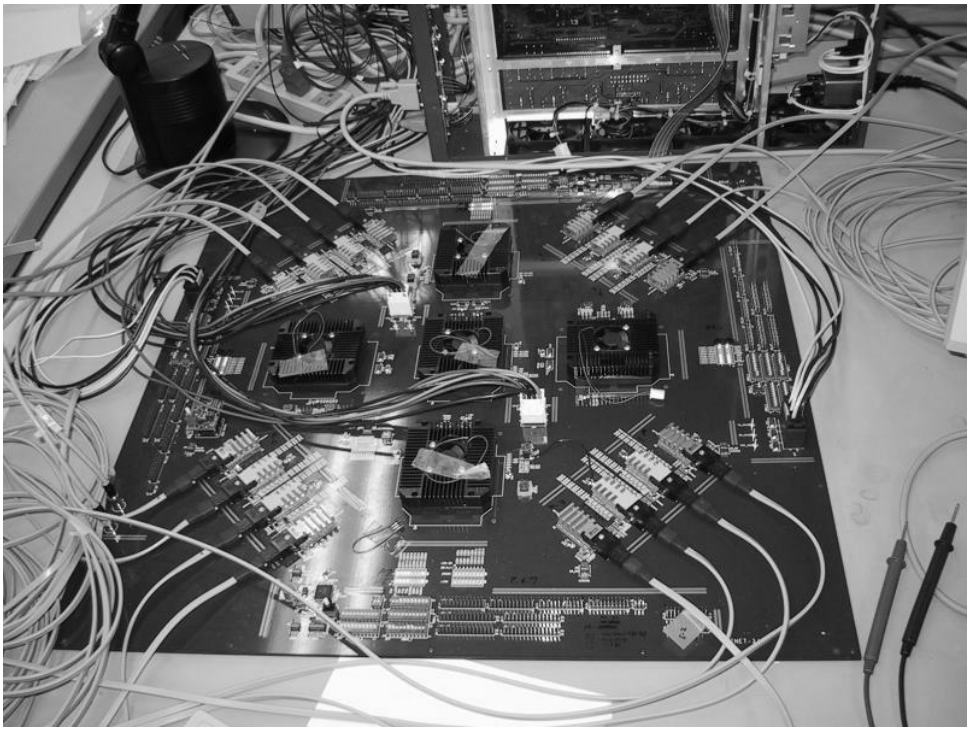


図 2.5　RHiNET-3/SW の基板

**RHiNET-3/NI**

RHiNET-3/NI は，RHiNET-2/NI と同様に，64bit/66MHz の PCI バスを介してホストに接続されるネットワークインタフェースである．

ネットワークインタフェース上に Deskew-LSI を搭載している点および光インタコネクションモジュールが RHiNET-3 の光リンクに対応したものとなっている点を除くと，RHiNET-2/NI とほぼ同様の構成なっており，ネットワークインタフェースコントローラも RHiNET-2/NI と同じく Martini

を搭載する．

図 2.6に RHiNET-3/NI の外観を示す．

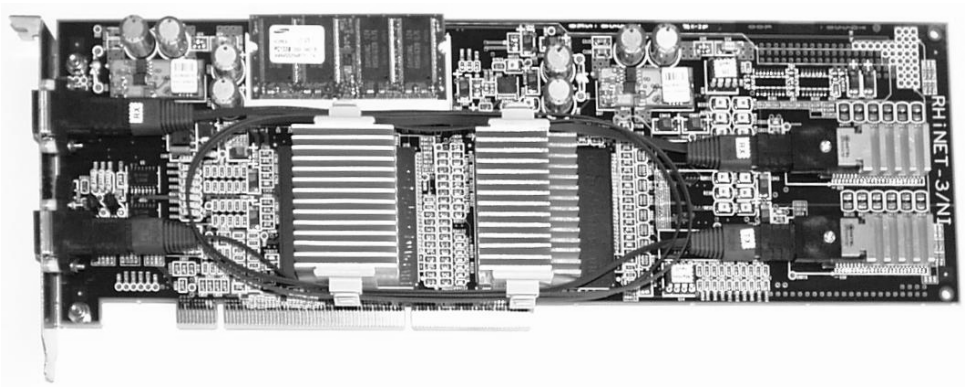



図 2.6　RHiNET-3/NI の外観

### 2.3.4　その他の RHiNET 関連する実装

**RHiNET-2/NI0**

RHiNET-2/SW は RHiNET-2/NI に先行して開発が行われたため，開発過程で動作検証を行うためのネットワークインタフェースが必要となった．RHiNET-2/SW は基本的に RHiNET-1/SW のコンセプトを踏襲していることから，RHiNET-2/SW の開発段階での検証用に RHiNET-1/NI を元にした RHiNET-2/NI0 と呼ばれるネットワークインタフェースが開発された．RHiNET-2/NI0 は RHiNET-1/NI と同様に 32bit/33MHz の PCI バスを介してホストに接続される．図 2.7に RHiNET-2/NI0 の外観を示す．

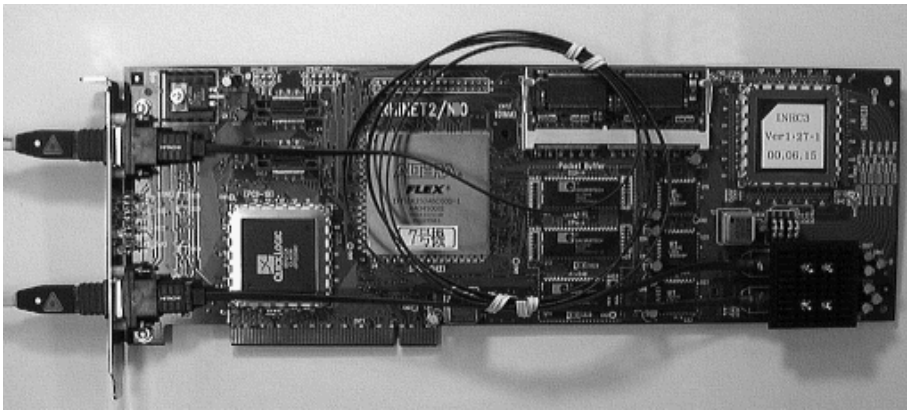


図 2.7　RHiNET-2/NI0 の外観

RHiNET-2/NI0 のコントローラ部には，RHiNET-1/NI に完全に実装できなかった TWIN/DIFF 方式に対応した PUSH や PULL，LOCK/UNLOCK などのプリミティブがハードウェア実装された [42]．しかしながら，コントローラ部は，動作周波数向上のための最適化を行っても PCI の動作周

波数である 33MHz に到達することができず，スループットは 80Mbyte/s，片道通信遅延は 7$\mu$sec という性能しか得られなかった [20].

**DIMMnet-1**

DIMMnet-1[43] は PC133 の DIMM スロットを介してホストに接続されるネットワークインタフェースである．チップ開発コストなどの問題から，DIMMnet-1 では独立してネットワークインタフェースコントローラを開発せず，ネットワークインタフェースコントローラに Martini を用い，結合網として RHiNET-2 や RHiNET-3 を用いる構成とした．その結果，Martini に対して DIMMnet-1 で要求される機能を実装する必要が生じ，メモリバスに接続するためのインタフェースコントローラなどが搭載されることになった．また，これを機に，Martini は On-the-fly (OTF) 通信機構 [44] と呼ばれる PIO ベースのパケット送出機構を搭載することになった.

DIMMnet-1 の外観を図 2.8に示す.

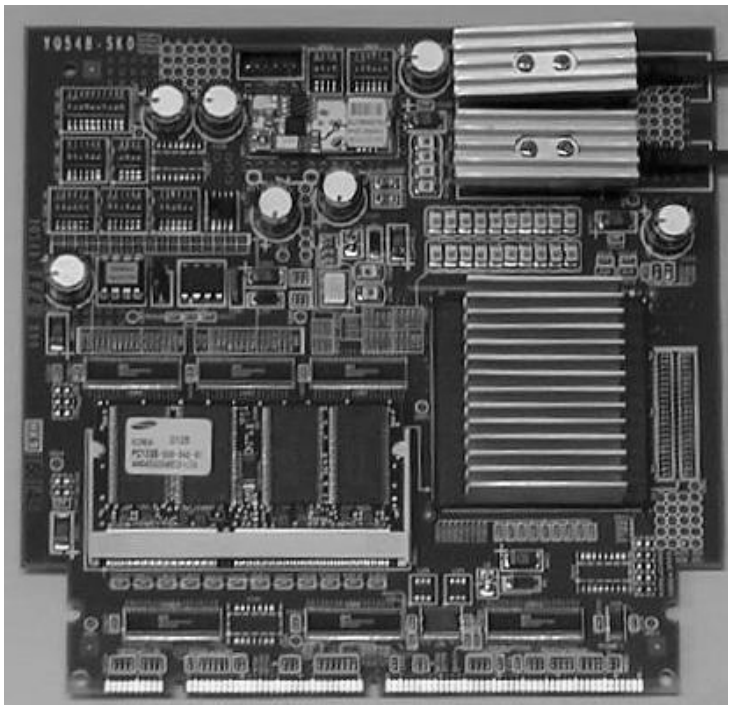


図 2.8　DIMMnet-1 の外観

**RHiNET-NI/NEC**

RHiNET-NI/NEC は，OIP-SW[45] と呼ばれる，新情報処理開発機構光インターコネクション NEC 研究室で開発された LSI 用光インタフェースを Intellectual Property(IP) 化した Optical-interconnection Intellectual Property (OIP) 技術 [46] を採用したスイッチを評価することを目的に開発されたネットワークインタフェースである.

64bit/66MHz の PCI バスを介してホストに接続され，コントローラ部に Martini を搭載する.

## 2.4 まとめ

RHiNET では，ハードウェアを積極的に導入して通信処理にソフトウェア処理が介在することを極力避けることで，高い性能と信頼性を備えた結合網を実現し，高性能かつ低オーバヘッドなプロセス間通信を提供する．信頼性の高い結合網を設けることは既存のクラスタ向けインタコネクションネットワークでも行われているが，プロセス間の通信処理まで含めてハードウェアで提供する点は既存のクラスタ向けインタコネクションネットワークや Ethernet に見られない RHiNET の大きな特徴の一つであると言える．このようなコンセプトに基づいたネットワークを実現することで，RHiNET は LASN の要件を満たし，さらに既存の SAN を上回る性能を実現することを目指す．

また，本研究の主題である Martini は，試作である RHiNET-1 以外のほぼすべてのコンポーネントと関係したネットワークインタフェースコントローラであり，RHiNET プロジェクトおよび他の周辺プロジェクトにおいて重要な位置づけにあると言える．Martini には，多様な要求を満たし，幅広い用途で高い性能を発揮するような設計・実装が求められる．

# 第3章　関連研究および関連技術

本章では，本研究において参照した，既存のクラスタで用いられているインタコネクションネットワークのアーキテクチャおよびそれらを利用した通信プロトコルの実装方式について概要をまとめる．

## 3.1　クラスタ向けインタコネクションネットワーク

以下では，クラスタ用途で研究・開発されたインタコネクションネットワークおよび現在クラスタにおいて広く用いられている汎用的なインタコネクションネットワークについて，代表的なものを挙げ，概要を述べる．

### 3.1.1　Myrinet

Myricom 社 [47] の Myrinet[3] は Caltech Mosaic C [48] および Mosaic C で用いられた USC/ISI ATOMIC LAN [49] の成果を元に開発されたクラスタ向けインタコネクションネットワークである．Myrinet は専用スイッチ，専用ネットワークインタフェースおよびそれらの間を接続するリンクで構成される．

Myrinet のスイッチは，カットスルー方式でパケットのスイッチングを行うクロスバスイッチであり，8×8 や 16×16 のクロスバスイッチをバックプレーンを介して多段接続し，Fat-Tree や Clos 網と呼ばれるトポロジの結合網を構築してノード間を接続する．このような結合網上でノード側でソースルーティングによる経路選択を行うことで，トラフィックの分散や経路の冗長化を実現する．16×16 のクロスバスイッチを組み合せて Fat-Tree を構築し，128 ノードの接続に対応した Myrinet の結合網を図 3.1に示す．

Myrinet のネットワークインタフェースは LANai(注 1)と呼ばれるネットワークインタフェースコントローラと大容量の SRAM を搭載する．LANai は内部に 32bit の RISC プロセッサを持ち，ネットワークインタフェース上でのプロトコル処理は RISC プロセッサ上で実行される Myrinet Control Program (MCP) と呼ばれるファームウェアによって実現される．SRAM は通信バッファなどに用いる．また，LANai 外部の専用コントローラ(注 2)によって，ホスト PC 上の物理メモリやネットワークとの間での DMA 転送が提供されている．図 3.2に，第 3 世代の Myrinet である Myrinet-2000 用のネットワークインタフェースの構成を示す．図の中央には，LANai9[50] と呼ばれるコントローラが位置している．

Myrinet は信頼性の高いリンクを用いており元々のエラー発生率は低い．また，CRC を用いたエラー検出を提供している．

---

(注 1)最近のものは “Lanai” と表記が変更されている．

(注 2)最近の Myrinet のネットワークインタフェースではこれらは LANai に統合されている．

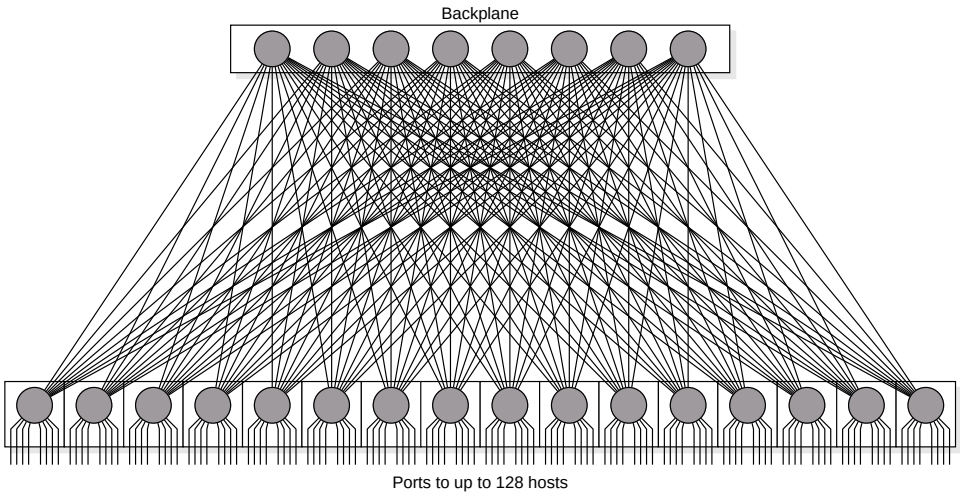



図 3.1　16×16 のクロスバスイッチを多段結合して Fat-Tree を構築した Myrinet の結合網

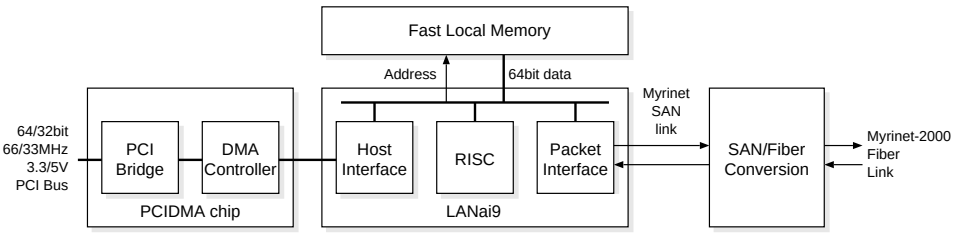



図 3.2　Myrinet-2000 用のネットワークインタフェースの構成

#### 第 1 世代および第 2 世代の Myrinet

1994 年に登場した最初の Myrinet は，Sun Microsystems 社のワークステーションをホストとしてサポートしており，ネットワークインタフェースは SPARC 向けのバスである SBus を介してホストと接続可能であった．リンク速度は 0.64G+0.64Gbps であった．

1990 年代後半に登場した第 2 世代の Myrinet では，リンク速度が 1.28G+1.28Gbps に強化された．ネットワークインタフェースには 33MHz 動作の LANai (LANai 4) と最大 1Mbyte の SRAM が搭載され，SBus に加えて新たに 32bit/33MHz の PCI バスを介したホスト接続への対応が行われた．また，後に 66MHz 動作の LANai (LANai 7[51]) を搭載した 64bit/66MHz PCI バス対応のネットワークインタフェースも登場した．

この頃の Myrinet のリンク媒体には銅線が用いられており，SAN モードと呼ばれる接続方式では最大 3m，LAN モードと呼ばれる接続でも最大 10m と，Ethernet などの LAN と比べてリンク長に厳しい制限が存在していた．

#### Myrinet-2000

2000 年頃に登場した第 3 世代の Myrinet は Myrinet-2000 と呼ばれ，現在主流となっている．Myrinet-2000 では，リンク速度が 2.0G+2.0Gbps に向上した．当初はネットワークインタフェースとして 64bit/66MHz 規格の PCI バスに対応したもの (最大 200MHz 動作の LANai (LANai 9[50]) を搭載) が提供されていたが，現在は 64bit/133MHz の PCI-X バスに接続可能なもの (最大 333MHz 動作の Lanai (Lanai X[52]) を搭載) が提供されている．PCI-X バスの転送能力と比べた場合，リンク速度は低いが，この問題を回避するためにネットワークインタフェース上のポート数を 2 ポートに増やしてノード間のデータ転送速度の強化を図ったデュアルポート方式のネットワークインタフェースも提供されている．リンクの媒体は光ファイバ (50/125 マルチモードファイバ) が標準となっており，最大で 200m まで延長可能となっている．

#### Myrinet のソフトウェア環境

Myrinet 向けのソフトウェア環境については，Myricom 社により低レベル通信ライブラリである GM[53][54] が提供されており，GM を利用した MPI や TCP/IP の実装が用意されている．また，Lanai X を搭載したネットワークインタフェース向けに Myrinet Express (MX)[55] と呼ばれる，より低遅延な通信を提供する通信ライブラリが用意されている．

なお，Myrinet ではネットワークインタフェースの仕様が公開されており，開発環境が提供されていることから，PM/Myrinet[56][7] などの独自の通信プロトコルが開発されており，研究用途にも広く利用されている [57].

#### Myri-10G

Myri-10G[58] は 10Gigabit Ethernet (10GbE) との相互運用が可能な Myricom 社による新しいインタコネクションネットワークである．IEEE 802.3ak や IEEE 802.3ae といった 10GbE と同じ物理層の規格を採用しており，スイッチやネットワークインタフェースのリンクの媒体には 10GBase-CX4 規格の銅線や 10GBase-R 系列の光ファイバの利用が可能である．Myri-10G のリンク速度は 10G+10Gbps であるが，ネットワークインタフェースは PCI Express x8 規格のバスを介した接続

に対応しているため，ホストとの間は 16G+16Gbps の全二重 I/O 接続となり，リンクに対して十分なデータ転送性能を提供可能な構成となっている．

Myri-10G のネットワークインタフェースは，従来の Myrinet と同様に，300MHz 以上のクロックで動作する Lanai (Lanai Z8E) を搭載し，ソフトウェアによりプロトコル処理を行う．MX で提供される Lanai 上のファームウェアで，データリンクレベルで Myrinet と 10GbE の両方のプロトコルに対応することができるため，Myri-10G のネットワークインタフェースは 10GbE のネットワークインタフェースとしても利用可能である．

### 3.1.2 QsNet

Quadrics 社 [59] の QsNet[60][61] は，ASCI Q-machine[62] と呼ばれる SMP クラスタ向けに開発されたクラスタ向けのインタコネクションネットワークである．QsNet は専用のスイッチ，ネットワークインタフェースおよびそれらの間を接続するリンクで構成される．

QsNet のスイッチは Elite と呼ばれる ASIC を搭載している．Elite は 4 進 Fat-Tree トポロジを基本とするフルクロスバスイッチである．Elite はソースルーティングに対応し，パケットの先頭についたタグの並びに応じてルーティングを行う．タグは任意の出力ポートの集合を指すことができ，これを利用したハードウェアマルチキャストもサポートされている．パケットは Wormhole 方式で転送され，送信元ノードと宛先ノードの間の経路は，宛先ノードがパケットを受け取ってから送信元ノードに確認応答 (Ack) を返すまで維持される．

QsNet のネットワークインタフェースは，Elan と呼ばれるネットワークインタフェースコントローラと大容量の外部メモリを搭載する．Elan は内部に DMA 要求やパケット処理などの通信処理を専門に行うプロセッサと，MPI などの上位の通信プロトコルを実装するのに用いる RISC プロセッサ (スレッドプロセッサ) の 2 つのプロセッサを内蔵する．また，Elan は MMU も内蔵しており，Elan 内部の処理で用いられる仮想アドレスを，ホスト上のメモリやネットワークインタフェース上の外部メモリの物理アドレスに高速に変換できるようになっている．Elan 上のメモリをユーザのメモリ空間にマップすることも可能であり，この領域を通信バッファや DMA ディスクリプタを置くのに用いる．その他，Elan はプロセッサ用のキャッシュやリンク接続用のロジックなどを内蔵している．

#### 第 1 世代の QsNet

1998 年に発表された第 1 世代の QsNet は，Elan3 と呼ばれるネットワークインタフェースコントローラと Elite3 と呼ばれるスイッチで構成される．Elan3 は，マイクロコードプロセッサと呼ばれる 32bit の通信処理専用プロセッサと，上位プロトコル処理用の 32bit のスレッドプロセッサを持つ．いずれのプロセッサも 100MHz で動作し，スレッドプロセッサは 8Kbyte のキャッシュを内蔵する．Elan3 を搭載したネットワークインタフェースは 64bit/66MHz の PCI バスに接続され，ネットワークインタフェース上には 64Mbyte の ECC つき SDRAM を搭載する．図 3.3に Elan3 のブロック図を示す．

Elite3 は，16×8 のフルクロスバスイッチ (入力ポートはチャネルごとに独立) であり，3.2G+3.2Gbps のリンク速度に対応する．

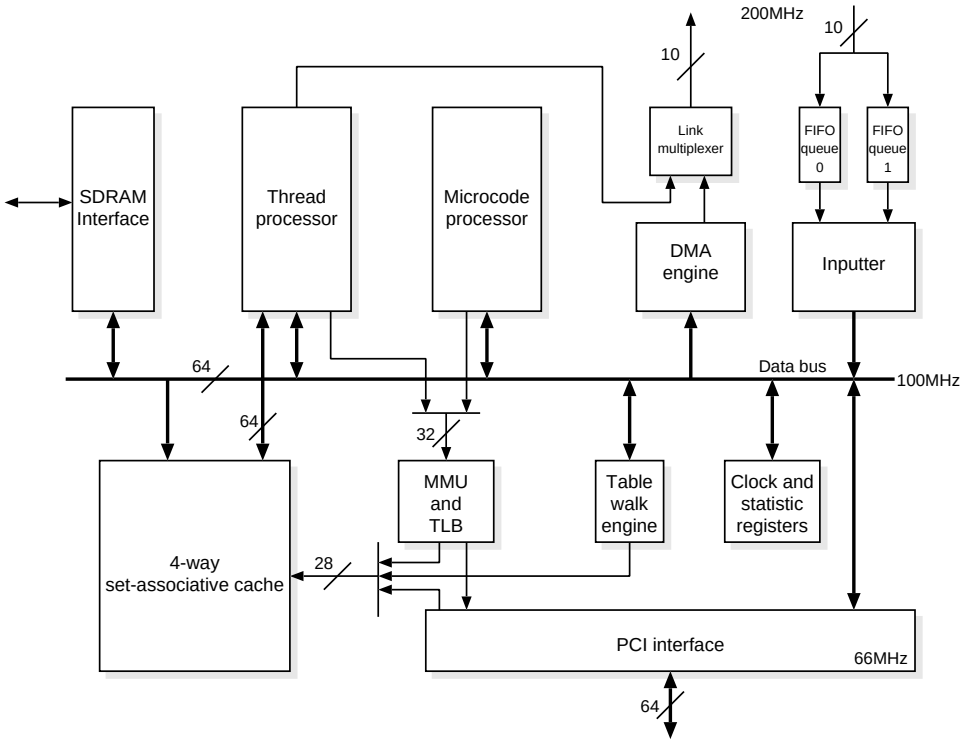



図 3.3　Elan3 の内部ブロック図

**QsNet II**

最新の QsNet である QsNet II は Elan4 と呼ばれるネットワークインタフェースコントローラと Elite4 と呼ばれるスイッチで構成される．Elan4 では通信処理用のプロセッサおよびスレッドプロセッサが 64bit アーキテクチャに拡張されており，動作周波数も 200MHz に向上している．スレッドプロセッサ用のキャッシュもデータキャッシュ32Kbyte，命令キャッシュ16Kbyte と，大幅に強化されている．また，Small Transaction ENgine (STEN) と呼ばれるサイズの小さいパケットの生成に特化したモジュールを搭載し，より低レイテンシな通信が可能になっている．Elan4 を搭載したネットワークインタフェースは 64bit/133MHz の PCI-X バスに接続される．ネットワークインタフェース上には 64Mbyte の ECC つき DDR SDRAM を搭載する．図 3.4に Elan4 の内部ブロック図を示す．

Elite4 は 13G+13Gbps(実効データ転送速度は 10.6G+10.6Gbps) のリンク速度および最大 100m のリンク長に対応する．

**QsNet のソフトウェア環境**

QsNet では，Quadrics 社によって Elanlib[63] と呼ばれる独自のソフトウェアライブラリが提供されており，その上で動作する MPI-2 などの標準的な並列プログラミング環境も用意されている．

また，Quadrics 社よりデバイスドライバや qsnetlibs と呼ばれるユーザライブラリなどがオープンソースで提供されているため，ユーザが Elan 上のプログラムを独自に開発することも可能となっている．

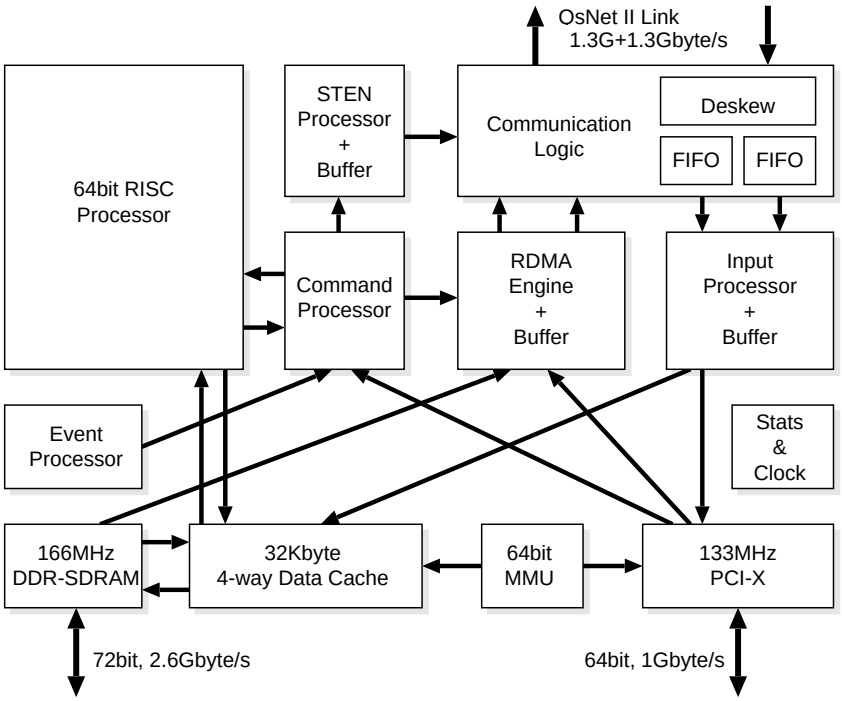



図 3.4 Elan4 の内部ブロック図

### 3.1.3 InfiniBand

InfiniBand は，サーバ I/O やサーバ間の通信に RAS(Reliability，Availability，Scalability) を提供することを目的に，InfiniBand Trade Association (IBTA)[64] によって策定されたアーキテクチャである．最初の規格として，InfiniBand Architecture Specification Release 1.0 [65] が 2000 年に策定されている．

InfiniBand はスイッチベースのファブリック構造を採用し，元々は PCI バスなどの汎用 I/O バスに代わるノード内部のチップ間接続およびノード間の接続に用いることを想定したアーキテクチャであったが，ノード内部のチップ間接続用に後発の PCI Express が普及したため，現在では PC クラスタやストレージ接続などの分野を中心に普及している．

#### InfiniBand のプロトコル

InfiniBand では，物理層，データリンク層，ネットワーク層およびトランスポート層の各通信層についてプロトコルが定められている．

- 物理層

  InfiniBand の物理層では片方向 2 本のシリアル差動信号方式で伝送を行い，双方向の 4 本単位を基本構成 (1X) として利用する．1X における実効データ転送速度は 2Gbps となる．媒体としては銅線や光ファイバの他，プリント基板も利用可能であり，光ファイバを利用した場合は，数百 m クラスのリンクを実現できる．

  また，信号線を 4 本束ねて使用する 4X や 12 本束ねて使用する 12X が規格として定義さ

れており，それぞれ 10Gbps，30Gbps のデータ転送速度を提供する．2004 年に策定された InfiniBand Architecture Specification Release 1.2 [66] では DDR と QDR の動作モードが新たに追加され，最大で 120Gbps のデータ転送速度が規定されている．

- データリンク層

  InfiniBand のデータリンクレベルのパケットには，通常のデータ転送に用いるデータパケットと，リンクの管理などに用いる管理パケットが存在する．ネットワーク内のスイッチングは Local ID (LID) と呼ばれる識別子によって行われる．フロー制御には Point-to-Point のクレジットベース方式を採用し，エラー検出には CRC が用いられる．

  スイッチではルーティングテーブルを用いたルーティングを行う．スイッチにおいてマルチキャストを行うことも可能である．

- ネットワーク層

  InfiniBand では，一意な LID でアクセス可能なネットワークをサブネットとして扱い，IP のように InfiniBand 用のルータを介してサブネット同士を接続することが可能である．

- トランスポート層

  ノード上のアプリケーション間では，ノード間で Queue Pair (QP) と呼ばれる送受信用のキューを用いた通信が提供される．QP を用いた通信では，Ack/Nack プロトコルによりノード間でのパケット到達保証を行う．

  また，QP を用いた通信では，コネクション指向通信とデータグラム型の通信が規定されており，ユーザは各々について，パケットの到達保証と順序保証が行われる信頼性のある通信モードと，不正なパケットの破棄のみが行われ再送処理などは行われない信頼性のない通信モードを選択して利用することができる．QP を用いないデータグラム型の通信モードも提供されている．

#### InfiniBand のハードウェア構成

InfiniBand は，Channel Adapter (CA) と呼ばれるネットワークインタフェース (ノード間接続に用いる Host Channel Adapter (HCA) およびストレージなどのデバイスとの接続に用いる Target Channel Adapter (TCA))，スイッチおよびサブネット間でのパケット転送を行うルータで構成される．

CA を搭載したノードとスイッチで構成されるサブネット内には，サブネットの管理を行う Subnet Manager が存在する必要があり，サブネット内の CA やスイッチのどれか一つがこれを担当することになる．また，各 CA やスイッチはサブネットマネージャと通信を行うための Subnet Manager Agent (SMA) を搭載する必要がある．

現在，Mellanox 社 [67] などの複数のベンダが InfiniBand 規格に準拠したスイッチや CA を開発・販売している．

#### InfiniBand のソフトウェア環境

InfiniBand の仕様では，Virtual Interface Architecture[68] と呼ばれる規格を拡張した verb と呼ばれる CA の提供すべき機能のみが定められている．具体的な通信 API の定義は各ベンダに委ねられ

ており，Mellanox 社による VAPI[69] などが規定されている．VAPI は，メッセージ通信と RMA 型の通信の両方をサポートし，InfiniBand で規定されているトランスポートサービスのうち，RC(信頼性のあるコネクション指向通信) と UD(信頼性のないデータグラム型の通信) が提供されている．また，オハイオ州立大学によって MVAPICH[70][71] と呼ばれる VAPI の上で動作する MPI が実装されている．

### 3.1.4 Ethernet

Ethernet は LAN に分類されるネットワークであるが，コストパフォーマンスに優れていることから，クラスタ用のインタコネクションネットワークとしても広く利用されている．1990 年代後半は，100Mbps のリンク速度の Fast Ethernet が主流であったが，現在では 1Gbps のリンク速度の GbE が主流となっている．また，10Gbps のリンク速度を持つ 10GbE 製品も登場しているが，現時点ではネットワークインタフェースやスイッチのポート単価が極めて高く，コストパフォーマンス面で GbE に比べ大きく劣る．

Ethernet では，TCP/IP を利用した MPICH[72] などの多数のプログラミング環境が利用可能であるが，TCP/IP はプロトコル処理が複雑であるため，通信処理のオーバヘッドが大きいという問題点がある．特に 10GbE などでは，ホスト CPU の性能が十分でない場合，通信処理だけでホストの処理能力の大半が消費されてしまうなどの問題が生じる．

また，PM/Ethernet[6] や GAMMA[8] などの Ethernet 向けの並列分散処理向けの軽量通信ライブラリも存在する．これらを用いることで，TCP/IP に比べ通信処理のオーバヘッドを軽減することができる．しかしながら，Ethernet のネットワークインタフェースの多くは OS を介したアクセスを想定した設計となっているため，SAN において一般的なユーザレベル・ゼロコピー通信を実装することは難しく，これら軽量通信プロトコルを用いても SAN と同等の低オーバヘッドな通信を実現することは難しい．

これに対し，最近では Chelsio Communications 社 [73] や Neterion 社 [74] などによって 10GbE を中心に TCP Offload Engine (TOE) を搭載したネットワークインタフェースが開発されており，TCP/IP 使用時のホストオーバヘッドの低減が図られている．また，RDMA Consortium によって iWARP と呼ばれる TCP/IP 上での RDMA の利用 (RDMA over TCP/IP) に関する規格が策定されており [75]，NetEffect 社 [76] などがこれに対応した 10GbE のネットワークインタフェースの開発を行っている．これらにより，Ethernet は Myrinet などの SAN との性能差が縮まる傾向にある [77].

また，一般的な Ethernet のスイッチでは論理トポロジが木構造に制限され，循環構造をとる論理トポロジを構築できないため，SAN に比べて経路分散が行いにくく，トラフィックの集中が発生して大規模化した際に高い bi-section bandwidth を得られないという問題がある．これに対し，最近では既存の VLAN 技術を用いることで SAN と同様の多様な論理トポロジを実現し，この問題を回避する手法が提案されている [78] [79][80].

### 3.1.5 その他のクラスタ向けインタコネクションネットワーク

#### Giganet

Giganet 社の Giganet[81] は，専用のネットワークインタフェースとスイッチで構成されるクラスタ向けインタコネクションネットワークである．

ネットワークインタフェースは後述する Virtual Interfaces Architecture (VIA)[68] をハードウェ

ア実装しており，コネクション型通信を提供する．ホストとの接続は 64bit/33MHz の PCI バスに対応する．また，スイッチは Asynchronous Transfer Mode (ATM) の技術を採用しており，VIA のエンドポイントである Virtual Interface (VI) を ATM の仮想チャネルに直接対応づけ，VI 間のメッセージを ATM Adoption Layer (AAL) 5 にカプセル化して転送する．1.25G+1.25Gbps のリンク速度に対応する．

**ServerNet**

ServerNet (TNet)[82][83] は，Tandem 社によって 1995 年に発表された，プロセッサ同士の接続や I/O デバイスの接続に用いることを目的としたインタコネクションネットワークである．プロセッサインタフェースやバスインタフェースはそれぞれ 2 つのポートを備え，2 つの独立したスイッチ網を利用することで冗長性を上げ，耐故障性を確保する．また，ノード間でのリモートライトやリモートリードを提供する．

後継の ServerNet II は VIA をハードウェア実装したネットワークインタフェースと 1G+1Gbps のリンク速度に対応した 12 ポートのスイッチで構成される．リンクの媒体には光ファイバがサポートされており，最大で 150m 程度のリンク長に対応する．また，最新の ServerNet である ServerNet III では，リンク速度は 2.5G+2.5Gbps に達している．

Tandem 社が後に Compaq 社に買収されたことにより，ServerNet 自体は Compaq 社 (現 HP 社) のクラスタ製品内部で主に使用されている．また，ServerNet のアーキテクチャは現在の InfiniBand 規格の元となった．

**Scalable Coherent Interface (SCI)**

Scalable Coherent Interface (SCI)[84] は 1992 年に標準化されたインタコネクションネットワーク規格であり，Point-to-Point のインタフェースとパケットプロトコルを定義している．元々バスを用いたマルチプロセッサシステムの問題を解消することを目的に開発が行われたことから，キャッシュコヒーレンシに対応した分散共有メモリ向けプロトコルが定義されており，メモリアクセスのためのバス要求をインタフェースでパケット化してリモートに転送し，リモートでこれを処理するといったことが行われる．また，メッセージパッシングのためのパケットプロトコルも定義されている．スイッチを介した接続も可能であるが，Point-to-Point 接続によるリング状のトポロジを基本としており，リングを組み合せて実現するトーラスが一般的に用いられる．

SCI の実装としては，現在，Dolphin 社 [85] によって PCI 接続のネットワークインタフェースや 8×8 のクロスバスイッチなどが提供されている．ネットワークインタフェースは最大 3 次元のトーラスにまで対応する．最新のネットワークインタフェースは PCI Express を介してホストに接続され，10G+10Gbps のリンク速度に対応する．

**Memory Channel**

Memory Channel[86] [87] は Digital Equipment Corporation (DEC) 社によって開発されたクラスタ向けネットワークである．Memory Channel ではノード間の通信に送信用と受信用の 2 つのメモリページを用いる．受信用のページは，物理メモリを割当て，スワップアウトしないようにピンダウンしておく．一方，送信用のページは，ネットワークインタフェースにマップする．送信ページへの書き込みアクセスは，ネットワークインタフェースによりリモートへパケットとして

転送され，リモートノードの受信用のページに対して書き込まれる．これにより，クラスタ間にまたがる広域のアドレス空間を実現することができる．

Memory Channel のネットワークインタフェースは PCI バスを介してホストに接続される．

### 3.1.6 まとめ

スーパーコンピュータの世界的ランキングである TOP500[88] の 2006 年 11 月の統計によると，世界上位 500 台のうち，クラスタシステムは 361 台にのぼり，さらにそれらのインタコネクションネットワークは Myrinet，QsNet，InfiniBand および GbE で占められている．一方，数年前までは Giganet や Dolphin 社の SCI を用いたクラスタも数台ずつランクインしていたが，現在ではこれらは圏外となっている．実際，これらをインタコネクションネットワークに用いたクラスタに関する文献は非常に少ない．Giganet は 1Gbps 以上のリンク速度を提供するが，VIA をハードウェア実装したことによりリソース面で制限が生じ，最大 256 ノードまでの接続しかできないというスケーラビリティ面での制限がある．同種のネットワークで新たにクラスタを構築するのであれば，よりリンク速度が高く，製品の選択の幅が広い InfiniBand を選択するのであろう．また，SCI はノード間を直接リング状のトポロジで接続する形態を前提としたネットワークであるため，拡張性などの面で不利である．

また，これらインタコネクションネットワークのうち，InfiniBand と Myrinet については，いずれも光リンクを用いることで数百 m のリンク長に対応し，数 Gbps クラスのリンク速度を提供していることから，技術的には LASN の結合網として利用可能であると考えられる．また，Ethernet に関しても VLAN 技術による bi-section bandwidth の問題の解消により，上位プロトコル次第では SAN に匹敵する性能が得られるものと考えられる．このようなことから，今日であれば LASN を実現するために結合網から新規に実装を行う必要はなく，必要であればネットワークインタフェースのみを開発することに注力するのが得策であろう．しかしながら，LASN が提唱された 1997 年頃には，InfiniBand 規格自体がまだ存在せず，Myrinet はリンク長が最大でも 10m 程度に制限されており，LASN としてそのまま利用可能な技術はなかった．そのため，RHiNET において結合網を新規に開発したことは，当時としては妥当な選択であったと言える．

## 3.2 クラスタ向け低レベル通信ライブラリ

これまで述べたインタコネクションネットワークの多くは企業を中心に開発が行われており，ユーザアプリケーションや上位通信ライブラリがネットワークの性能を十分に活用できるよう，独自の低レベル通信ライブラリがベンダによって提供されている．一方，既存のインタコネクションネットワークを利用してベンダ提供の低レベルな通信ライブラリよりも高い通信性能を実現し，クラスタ全体の処理能力を高めることを目的に，様々な低レベル通信プロトコルの研究が行われている．以下では，これらプロトコルに関する研究に関して代表的なものを挙げ，概要を述べる．

### 3.2.1 Active Messages (AM)

Active Messages (AM)[89] は 1990 年代初頭に California 大学 Berkeley 校 (UCB) のグループによって提案された通信モデルである．元々は分散共有メモリ型の計算機の性能向上を目的に，CM-5 や nCUBE/2 向けに開発が行われ，後に Intel Paragon や IBM SP-2，Meiko CS-2 などにも実装さ

れた．

AM では，パケット (メッセージ) のヘッダにリモートのユーザプログラム中のメッセージの受信処理関数 (ハンドラ) のポインタを保持し，リモートでメッセージが受信されると直ちにヘッダのポインタを参照して受信処理関数が呼び出され，受信データが処理される．初期の AM では，通信モデルとしてリモートライト (PUT) およびリモートリード (GET) が提供されていた．

クラスタ向けに実装された AM としては，Medusa 社の FDDI を介して HP 9000/735 ワークステーションを接続して構築したクラスタ向けのもの (HPAM[90]) や，SPARCstation 20s を ATM で相互接続して構築したクラスタ向けのもの (SSAM[91]) がある．

HPAM では，信頼性の低いネットワーク上で効率的に通信を行うために，AM の通信モデルを強化し，Request–Reply 型のモデルを導入している．Request–Reply 型のモデルでは，Request と Reply の 2 種類のメッセージを用いて通信を行い，Request メッセージに対応するハンドラは Reply メッセージを送信するためだけにネットワークを利用し，Reply メッセージに対応するハンドラはネットワークを使用しないよう制限を設けることを行う．

また，HPAM では，FDDI ネットワークインタフェースをすべての通信プロセスに対してマップし，スケジューリングデーモンが通信プロセスのうち一つだけがアクティブになるよう管理することでデバイスへのプロセス間のアクセス競合を防ぐ．その際，非アクティブなプロセスへのメッセージ到着に対応するためキューを別途設ける．メッセージのやりとりは，同一の並列ジョブに属する並列プロセス間でのみに制限されるよう，ジョブごとに割振ったキーをメッセージに付加することで行う．

#### Active Messages 2.0 (AM-II)

100 台規模の Sun 社の UltraSPARC 搭載ワークステーションを Myrinet で相互接続して構築した NOW クラスタ [33] では，AM を改良した Active Messages 2.0 (AM-II) [92] が導入された．

AM-II では，メッセージのやりとりにエンドポイントと呼ばれる概念を導入することで，これまで同一の並列ジョブに属する並列プロセス間のみに限定されていたメッセージのやりとりの範囲を任意のプロセス間に拡大している．また，AM-II のハンドラは，テーブルで管理され，関数ポインタを通知せずにテーブル上のインデックス番号で指定を行う．

エンドポイントは仮想化されたネットワークインタフェースであり，各プロセスは複数のエンドポイントを生成して利用することが可能である．個々のエンドポイントにはタグがつけられており，タグが一致するエンドポイント間でのみメッセージのやりとりが許可される．タグ自体は 64bit のランダムな値を用いており，またプロセスは任意のタイミングでエンドポイントのタグを変更することができる．各エンドポイントには Request メッセージを格納するための送受信キュー，Reply メッセージを格納するための送受信キュー，ハンドラテーブルおよびサイズの大きなメッセージを受信するためのバッファが割り当てられている．エンドポイントは，通信プロセスとネットワークインタフェース上の LANai の双方からアクセスされることになるため，LANai からアクセスしやすいようにネットワークインタフェースのメモリ上に確保される．プロセスがエンドポイントを確保した場合，このメモリ上の領域がプロセスのアドレス空間にマップされることになる．

AM-II では，メッセージの大きさにより 3 種類の通信方式を切り替えることで効率的な通信処理を図っている．短いメッセージの場合，送信側のプロセスはエンドポイントに直接データを書き込み，受信側のプロセスは PIO でエンドポイントに格納されたデータを読み出す．一方，中程度のサイズのデータのメッセージを送る場合，送信側は一旦プロセスの領域からプロセスのアド

レス空間にマップされたカーネル内のヒープにメッセージをコピーし，エンドポイントにはメッセージに必要となる情報のみを格納する．送信時にネットワークインタフェースが DMA でヒープからデータを読み出し，メッセージとして送出する．受信時には，LANai はエンドポイントにヘッダ情報のみを格納し，データはカーネルのヒープに DMA でコピーされる．さらにサイズの大きなメッセージの場合は，バルク転送となり，送信側は中程度のサイズと同様にカーネルヒープを介して送信し，受信側ではカーネルヒープからさらにプロセスのメモリ領域上に確保したバッファにコピーが行われる．

### 3.2.2 Fast Messages (FM)

FM は Illinois 大学によって開発された PC クラスタ向け通信ライブラリである．AM と同様，元々は Cray T3D 上での分散共有メモリ向けに開発されたが，後に Sun 社の SPARCstation を Myrinet で相互接続したワークステーションクラスタ上に移植された [35][93].

FM は，ペイロードが短い (4 ワード以下) メッセージの送信，ペイロードが長い (4 ワード超) メッセージの送信およびメッセージの受信の 3 つのプログラミングインタフェースを提供する．

各メッセージは AM と同様にハンドラのポインタを持つが，AM と異なり Request–Reply 型の通信モデルは用いていない．また，AM と異なり，メッセージを受信しても直ちにメッセージの受信ハンドラが呼び出されず，LANai がこれを一旦ネットワークインタフェースのメモリ上のキューに格納し，受信側のプロセスが明示的に受信関数 (`FM_extract`) を呼び出すまでハンドラによる処理が行われない．

FM では，Myrinet を利用することで信頼性の高い通信路が利用できるため，タイムアウト処理や再送処理，並び替えといった処理は行わず，フロー制御によるパケット破棄防止と，キューを用いた順序性の維持のみをプロトコルで実現する．

FM ではネットワークインタフェース上のメモリに送信用のメッセージキューと受信用のメッセージキューを確保し，これをホスト上のプロセスの空間にマップする．また，ホストメモリ上にも，大容量の受信用のキューを確保し，ピンダウンしておく．LANai は，送信キューとネットワークを監視し，送信キューにホストからメッセージが書き込まれていればネットワークへ DMA で送出する．ホストからの送信キューへのデータの格納は PIO で行う．一方，ネットワークにメッセージが到着していれば，DMA でネットワークから受信キューに転送を行う．その後，受信キュー上のメッセージはホストメモリ上のキューに DMA 転送される．

フロー制御には，End-to-End のクレジットベース方式を採用している．受信側で，受信キューの中身が不要になった際に，送信側にクレジットを送る．

FM では，ホスト CPU からネットワークインタフェースにデータを格納する際に PIO を利用することになるため，スループットはネットワークインタフェースへの PIO アクセス時の最大性能に制限されてしまう．

#### Fast Messages 2.x (FM 2.x)

これまで述べた FM の実装では，上位レイヤプロトコルで `MPI_recv` のような受信関数が呼ばれると，`FM_extract` が呼ばれ，受信キューのすべての内容が一気に処理されてしまう．その際，受信キューのデータのうち，受信関数の処理対象外のものは一旦受信バッファにコピーされ，その後，再度受信関数が呼ばれた際にあらためて上位レイヤ用のバッファにコピーが行われる．そ

のため，受信時にメモリコピーが頻発してしまうという問題がある．

このような問題を解決すべく，送受信メッセージをバイトストリーム化して処理する FM 2.x[94] が実装された．FM 2.x では，`FM_extract` の際に 1 度に処理する最大サイズを指定することで，メッセージを必要な分だけ処理可能とし，メモリコピーの回数を削減している．

### 3.2.3 U-Net

U-Net は Cornell 大学によって開始された研究プロジェクトであり，クラスタシステムにおけるユーザレベル通信の実装を目標としている．最初の実装は，SPARCstation を ATM で相互接続したクラスタシステム上で行われた (U-Net/ATM) [34]．また，後にノードとして 133MHz の Pentium 搭載 PC を Fast Ethernet で接続したクラスタシステム向けにも実装された (U-Net/FE)[95].

U-Net では，ネットワークインタフェースを仮想化・多重化し，エンドポイントと呼ばれる形でアプリケーションプロセスに対して提供する．U-Net のエンドポイントは，メッセージデータ本体を保持するバッファ，その空き領域を管理するフリーキューおよび送受信メッセージに関するディスクリプタを格納・保持する送信キューと受信キューで構成される．

ユーザプロセスは，ネットワークを利用するにあたり，まずエンドポイントを 1 つ以上生成する．エンドポイントを介してメッセージを送るには，まず送信データをバッファ上に構築し，送信キューに送信メッセージに関するディスクリプタを格納する．U-Net のレイヤはディスクリプタを元にネットワークへのメッセージの送出処理を行い，完了後，ディスクリプタに対して完了を通知するフラグを設定する．一方，ネットワークから到着したメッセージは，U-Net のレイヤによって処理され，データが適切なエンドポイントのバッファに格納され，受信メッセージに関するディスクリプタが受信キューに格納される．ただし，サイズの小さいメッセージは，バッファに置かれずに直接受信キューに格納される．メッセージの到着については，プロセスが定期的に受信キューをポーリングして検出を行う以外に，シグナルなどを利用して U-Net のレイヤから通知を受けることも可能である．

宛先エンドポイントの識別はタグを利用して行う．プロセス間で通信チャネルを構築する際に利用するタグを U-Net のレイヤに登録しておくことで，送出されるメッセージにタグが付与される．タグを用いることで，同じ並列ジョブに属するプロセス間だけでなく任意のプロセス間での通信が可能となる．

データのコピー処理については，メッセージのデータをカーネル内で一旦バッファする base-level と，ネットワークインタフェースがユーザ領域との間で直接データコピーを行うゼロコピー通信を提供する direct-access の 2 つの方式が定義されている．

#### U-Net/ATM

ATM を利用した U-Net の実装である U-Net/ATM は，ネットワークインタフェースに Fore Systems 社の SBA-200 を用いている．SBA-200 はボード上に Intel 社の RISC プロセッサである i960 (25MHz 動作) および 256Kbyte のメモリを搭載しているため，SBA-200 を用いた U-Net のレイヤはネットワークインタフェース上のファームウェアによって提供される(注 3)．U-Net/ATM では，メッセージ格納用のバッファをホスト上の物理メモリに確保し，ピンダウンした上で，i960 の DMA 空間

(注 3)初期の実装ではネットワークインタフェースに Fore Systems 社の SBA-100 が用いられており，U-Net の機能はカーネルのレイヤで提供されていた．

にマップする．また，受信キューは，ホスト上のプロセスがポーリングでアクセスを行いやすようにホストメモリ上に確保する．一方，送信キューとフリーキューに関しては，i960 が DMA を用いずにポーリングできるのが望ましいことから，ネットワークインタフェース上のメモリに確保する．

#### U-Net/FE

Fast Ethernet を用いた U-Net の実装である U-Net/FE は，ネットワークインタフェースに DEC 社の DC21140 を用いている．DC21140 は，フレーム転送用の DMA エンジンと Ethernet 向けの CRC 処理機構を持つ．DMA エンジンへの指示は，ホスト上のメモリ領域を指すディスクリプタによって行われ，ディスクリプタは送信用と受信用に分かれてリング状に管理される．このような DC21140 の構造は，ユーザが直接アクセスするのには適さないため，DC21140 向けの U-Net のレイヤはカーネルレベルで実装されている．

U-Net/FE では，プロセスはエンドポイントの送信キューに送信メッセージのディスクリプタを格納してカーネルを呼び出す．カーネルの U-Net のレイヤは，エンドポイントの送信キューを読み，対応する DC21140 用のディスクリプタを生成して DC21140 の送信用のリングに格納する．送信用のリングに格納されているディスクリプタは，Ethernet のヘッダ部分はカーネル領域のバッファ上に構築されたものを指すが，ペイロードについてはユーザ領域の U-Net 用のバッファ上のアドレスを指し，送信時のメモリ間コピーを排除する実装となっている．Ethernet のパケットが到着すると，DC21140 は，受信リングの DC21140 用のディスクリプタが指しているカーネル領域上の受信バッファにこれを転送し，ホストに対して割込みをかける．割込みを受けると，タグの参照が行われ，対応するエンドポイントのバッファに対してメッセージのコピーが行われ，エンドポイントの受信キューにディスクリプタが格納される．

#### U-Net/MM

U-Net/MM[96] は，従来の U-Net を改良した実装であり，TLB を新たに導入することで，あらかじめ通信バッファとして確保した特定の領域ではなくアプリケーション領域上の任意のアドレスを DMA で利用可能としている．U-Net/MM は SBA-200 と DC21140 向けに実装されている．

### 3.2.4 Virtual Memory-Mapped Communication (VMMC)

Virtual Memory-Mapped Communication (VMMC) [97] は，Princeton 大学の SHRIMP プロジェクト [98][99] の一環で開発されたネットワークインタフェース向けの通信システムである．VMMC では，送信側がネットワークを介して受信側のメモリ領域を自身のアドレス空間にマップし，この領域を介して通信を行う．

受信側はあらかじめ通信用にマップされるべき領域を export しておく．送信側はこれを import して，アドレス空間上に確保した destination proxy space と呼ばれる領域にマップする(注 4)．export された領域を destination proxy space へマップすることで，受信側の export している領域に対してデータを送信する権利が送信側にあるかどうかを確認することができる．また，この領域を介し

(注 4)destination proxy に対応するメモリや I/O があるわけではなく，後述の deliberate update と呼ばれるモード関数呼び出しでリモートの受信バッファを指定するのに用いられる．

てデータを送ることで，受信側がまだバッファを確保する前の段階であっても，アプリケーションが送信要求を行うことが可能となる．

VMMC では，deliberate update と automatic update の 2 種類の通信モードが定義されている．deliberate update は，送信側で明示的に関数を呼び出すことで，任意の領域上のデータを import 済みの受信側のバッファに転送するモードである．送信先は destination proxy space 内のアドレスを用いて指定する．一方，automatic update は，送信側のローカルのアドレス空間への書き込みアクセスによって，受信側のバッファへデータ転送を行うモードである．atomic update では，転送起動までのレイテンシが隠蔽される．また特殊な通信関数が必要とされないため，既存の分散共有メモリモデルを用いたコードを容易に移植することが可能となる．これを実現するために，既に import してある受信バッファをユーザの仮想アドレス空間にマップすることを行う．

VMMC では，データ転送完了後に受信プロセス側でユーザレベルのハンドラを起動させることで受信通知を行うことができる．deliberate update の場合，送信時に通知を行うかどうかを指定する．automatic update の場合は，受信バッファのマップ時にこれを指定する．一方，受信プロセスは，export した領域ごとにハンドラを指定することができる．UNIX のシグナルのように，通知をブロックすることも可能であるが，シグナルと異なり複数の通知はキューされる．

VMMC はハードウェアとの協調によって実現される．頻度の高いデータ転送はハードウェアで実現し，import/export 処理などは VMMC 専用のデーモンを導入し，ソフトウェアで行う．

VMMC が実装された SHRIMP は，Pentium を搭載した PC を，EISA バスを介して Intel Paragon の結合網を用いて相互接続したクラスタシステムである．ネットワークインタフェースには SHRIMP-I[98] [99] と SHRIMP-II[100] の 2 つの実装がある．

SHRIMP-I は deliberate update のみに対応し，送信処理はシステムコールを用いて呼び出される．受信側では，受信領域を物理メモリにピンダウンしておき，その物理アドレスを送信側のデーモンに通知しておく．送信側で，受信領域へのデータの転送が行われると，ヘッダに書き込み先の物理アドレスが書かれたパケットが送られる．これを用いて，受信側ではネットワークインタフェースから DMA でホストメモリに対して書き込みが行われる．

SHRIMP-II では，SHRIMP-I と同様の転送方式に加え，automatic update にも対応する．また，ユーザレベルで通信を行うためのプロテクションも提供する．受信バッファは，送信側の仮想アドレス空間にマップされ，automatic update はバスのスヌープで実現する．deliberate update も仮想アドレス空間にマップされた I/O へのアクセスで実現する．

**VMMC on Myrinet**

VMMC は後に Myrinet と Ethernet を組み合せたシステム上にも移植されている [101]．このシステムでは deliberate update のみをサポートする．データ通信には Myrinet を用い，Ethernet はデーモン間の通信に利用する．Myrinet のネットワークインタフェースのメモリ上にキューを構築し，これをユーザのアドレス空間に直接マップする．サイズの小さなメッセージを送る際は，メッセージをキューに直接書き込む．一方，サイズの大きなメッセージを送る場合は，メッセージの場所を指すアドレスをキューに書き込み，LANai がホスト上のメモリから読み出して送出する．これを実現するために，ネットワークインタフェースのメモリ上に TLB を設け，ファームウェア上で仮想アドレスを物理アドレスに変換する．また，ネットワークインタフェースのメモリ上には destination proxy space のアドレスを変換するための変換テーブルも置かれる．

**VMMC-2**

VMMC-2[102] は VMMC に対してユーザ管理 TLB (UTLB)，転送リダイレクトおよびデータリンク層での信頼性の確保の 3 つの機能を新たに設けることで性能の改善を図った通信システムである．

VMMC-2 では Myrinet を結合網として利用するが，VMMC on Myrinet と異なり，import/export 処理を MCP に処理させることで VMMC デーモンを廃止し，Ethernet を不要としている．

UTLB はプロセスのピンダウンされた全ページの物理アドレスを持つ TLB であり，ドライバのメモリ上に保持される．通信開始の際は，仮想ページの識別子がネットワークインタフェースに渡され，VMMC-2 のライブラリは UTLB を参照して物理アドレスを得る．

転送リダイレクトは，最終的な受信データの格納先を送信側で指定せずにデータを送り，受信側であらかじめ指示されている受信先へデータを書き込むデータ転送方式である．受信側で，受信先が指定されていない場合は一旦デフォルトのバッファに蓄えられ，後に受信先が与えられると，受信先へのデータのコピーが行われる．

VMMC-2 では，通信の信頼性はネットワークインタフェース間の再送処理によって実現する．再送バッファは Ack を受け取るまで内容を保持する．受信側では，途中のパケットが失われたことを検知したら，後続のパケットはすべて破棄する．これにより順序性を維持する．

### 3.2.5 BIP

BIP[103] は Lyon 大学で開発された低レベル通信ライブラリである．ネットワークには Myrinet を用いるが，Myrinet 以外のネットワークインタフェースでもファームウェアと DMA エンジンがあれば実現可能な設計となっている．

BIP は上位に MPI や IP などのプロトコルレイヤを構築して利用することを想定し，高度な通信処理は上位レイヤで提供する方針で開発されている．

基本的な通信モデルとしてメッセージ通信を提供する．メッセージ通信はノンブロッキング型の呼び出しにも対応するが，1 度にペンディングできる Send/Recv は最大 1 個までに制限されている．

フロー制御に関しては，BIP のレベルではハードウェアの機能に依存したフロー制御のみをサポートする．そのため，BIP 上の MPI の実装である MPI-BIP ではクレジットベースのフロー制御を行っている．また，通信信頼性に関しては，メッセージにシーケンス番号をつけ，CRC のチェックを行うことでメッセージの欠落や破損を検出する．ただし再送などのリカバリ処理は提供せず，対応を上位レイヤに委ねる．このように，BIP 自体は非常にシンプルな機能のみを提供する．

さらに，BIP では複数のアプリケーション間でネットワークインタフェースを共用することを想定せず，また，ホストは並列処理専用とみなし，他のプロセスの動作への影響を考慮せずにビジーループを多用する実装となっている．

Myrinet の転送では，送信側と受信側の双方で，ホストメモリとネットワークインタフェース上のメモリの間の DMA 転送およびネットワークインタフェース上のメモリとネットワーク間の DMA 転送が行われる．そこで，BIP ではサイズの大きなメッセージを複数回に分けて送ることで，これら 4 つの DMA をパイプライン動作させ，ネットワークの利用効率を向上させている．一方，サイズの小さいメッセージについては，メッセージ本体を直接ネットワークインタフェース上のメモリに書き込んで転送する．受信側では，事前に確保したキューに受信される．

### 3.2.6 PM

PM[56] は新情報処理開発機構 (RWCP) によって開発されたクラスタ向けの低レベル通信ライブラリである．上位に SCore[21][22] と呼ばれるクラスタシステムソフトウェアを構築することを念頭に開発されており，メッセージ通信と Remote Memory Access (RMA) の 2 種類の通信モデルを提供する．PM ではレイテンシを削減するためにポーリングを多用する実装となっており，上位レイヤで積極的に受信関数を呼んでメッセージを受信することが要求される．PM はまず Myrinet 向けに実装が行われ，後に Ethernet や InfiniBand などにも実装された．

#### PM 1.0

Myrinet 向けの最初の PM の実装 (PM 1.0)[56] では，メッセージ通信のみが実装されている．

PM 1.0 では Myrinet 上でのスループットを上げるために，送信側でホストから SRAM への DMA 転送が開始した直後に SRAM からネットワークへの DMA 転送を開始する Immediate Sending と呼ばれる機構を導入している．一方，受信側では CRC のチェックがあるため，同様の処理を行うことができない．そこで受信側ではダブルバッファリングを用いてスループットを上げている．

フロー制御には Modified ACK/NACK と呼ばれる方式が導入されている．Modified ACK/NACK では，メッセージにシーケンス番号をつけ，送信側でメッセージをバッファした上で送出し，受信側から ACK が返ってきた場合のみこれを解放する．受信側でメッセージが受信できなくなった場合は，再送を希望するメッセージのシーケンス番号を沿えて NACK を返し，それより大きなシーケンス番号のメッセージはすべて破棄する．送信側では，NACK が返ってきたら，NACK で要求されている番号から再度送信を行う．Modified ACK/NACK を用いることで，スケーラビリティに優れ，順序性を維持したフロー制御が可能となる．また，これを利用することでネットワーク上に未受信のメッセージがないことを保証できるため，コンテキストスイッチが可能となる．

#### PM 1.2

PM 1.2[104] では，PM 1.0 にゼロコピー通信の機能が新たに追加されたことで，RMA がサポートされている．

PM 1.2 のゼロコピー通信には，Pin-down Cache と呼ばれるメモリ管理機構が導入されている．Pin-down Cache では，ピンダウン要求に応じて物理メモリ上に固定した領域をアンピンダウン要求で解除せずに物理メモリ上に保持したままにする．これにより，同じ領域に対して再びピンダウン要求があった際に物理メモリへの固定処理を省略できるため，ピンダウン処理のオーバヘッドを低減できる．なお，物理メモリ上に固定されたままの領域が無制限に増えるのを防ぐために，ピンダウンしたままの領域が一定数を超えた場合は，既にアンピンダウン要求がなされている領域の物理メモリへの固定を解除する．

#### GigaE PM

GigaE PM[105] は，プロセッサを搭載し，ファームウェアの改変が可能な GbE のネットワークインタフェースを対象とした PM の実装であり，Essential 社の EC-440-SF を実装対象としている．

GbE は Myrinet と異なり，ネットワークレベルで通信信頼性を提供しておらず，パケットの消失の可能性がある．そこで，GigaE PM では GO back N プロトコルを導入している．GO back N

では，送信側は i 番目から i+N 番目のメッセージを，ACK を待たずに送る．受信側は，データを受け取るたびに，受信したシーケンス番号のついた ACK を返す．送信側では，i 番目の ACK を受け取ったら，i+N+1 番目までのデータの送信が可能となる．送信側が一定時間 i 番目の ACK を受け取れなかった場合，メッセージの消失か ACK の消失が考えられるため，i 番目から再送を行う．そのため，送信側では ACK を受け取るまでデータを保持しておく必要がある．一方，受信側では，次に i 番目が受信されるべきところで i より大きいシーケンス番号のメッセージが受信された場合，データが途中で欠落したと判断して i よりシーケンス番号の大きいメッセージをすべて破棄した上で LOSE メッセージを送信側に送る．TCP/IP におけるスライディングウィンドウと良く似た仕組みであるが，パケットが欠落した際に，欠落したパケットより後のものをすべて再送する点が異なる．

フロー制御は STOP and GO で行っており，受信バッファがあふれそうになった場合，STOP メッセージを送り，十分に空きができたら GO を送る．

EC-440-SF では，ネットワークインタフェース上のメモリに対してアクセスする際にコントロールレジスタを介する必要がある．そのため，ユーザプロセスが直接ネットワークインタフェースのメモリ上に通信起動用のディスクリプタを置く場合，コントロールレジスタをアクセスする必要が生じるが，これはユーザプロセスに対して他の特権レジスタへのアクセス権限も与えてしまうことになる．そこで，GigaE PM では，ネットワークインタフェースへのディスクリプタの書き込みはカーネルを呼び出して行うものとする．一方，EC-440-SF 上の一部の領域についてはコントロールレジスタを介さずにアクセス可能であるため，受信メッセージ用のディスクリプタをこの領域上に置くことで，ユーザプロセスからカーネル呼び出しを用いない受信検出を実現する．なお，実際に送受信データを置くバッファは，ユーザメモリ上に確保し，ピンダウンしておく．

MTU は 1468byte に制限し，PM の単一のメッセージを複数のフレームに分割して送ることはしない．

**PMv2**

PMv2[7] は，これまで実装されてきた PM を発展させ，Ethernet や Myrinet，SMP における共有メモリ (shmem) などの複数のネットワークインタフェースの同時利用に対応した実装である．これにより，SMP 構成のノードでは，リモートノードとの間の通信には Myrinet を用い，同一ノード上の別プロセッサ上のプロセスとの間の通信には共有メモリを用いるといった処理を，上位レイヤが意識することなく実現できる．また，Ethernet などでは RDMA によるデータ転送などが実現できない場合があることから，PMv2 ではメッセージ通信によるデータ転送を基本関数とし，メモリのピンダウン処理や RMA については，実装は任意と定めている．

**PM/Ethernet**

PM/Ethernet[6] は，Ethernet 向けの PMv2 の実装である．

Ethernet 向けの PM の実装には GigaE PM があるが，GigaE PM ではプロセッサを搭載した GbE のネットワークインタフェースを利用して，ファームウェアを改変することで PM を実現しているため，プロセッサを搭載していない一般的な GbE のネットワークインタフェースでは同じアプローチで PM を実装することができない．そこで，多くの種類のネットワークインタフェースに対応し，クラスタ向け通信ライブラリとして十分に高い性能を提供可能な GigaE PM II[106] (後の PM/Ethernet) の開発が行われた．

GigaE PM II 自体は Packet Engines 社の G-NIC II をネットワークインタフェースとして利用した実装となっているが，多くのネットワークインタフェースに対応できるよう，既存の Ethernet のデバイスドライバを利用し，その上に PM の機能を実現する実装となっている．メッセージの受信関数が呼ばれた際に，デバイスドライバの割込みハンドラを直接呼び出すことで，積極的にメッセージの受信を行う．

#### PM/InfiniBand-FJ

PM/InfiniBand-FJ[107] は，富士通社の InfiniBand の HCA 向けに実装された PM である．

InfiniBand-FJ では，InfiniBand のトランスポート層で提供される信頼性のあるコネクション指向通信の上に PM の通信機構を実現している．

### 3.2.7 Virtual Interface Architecture (VIA)

VIA[68] はユーザレベル通信の標準化を目的に，Compaq 社や Intel 社，Microsoft 社などを中心に 1997 年に策定された通信モデルである．

VIA は，これまで述べたクラスタ向けの通信ライブラリから様々なアイディアを採り入れている．

VIA ではコネクション型の通信を基本としており，通信に先立って Virtual Interface (VI) を生成する．各 VI はリモートのプロセス上の特定の VI に接続され，これを用いてリモートプロセスとの間にコネクションを形成する．

各 VI は送信キューと受信キューを持ち，通信要求に必要となる情報を持ったディスクリプタを格納することで通信処理を発行する．メッセージを送信する場合，プロセスは送信メッセージに関する情報が書かれたディスクリプタを送信キューに入れる．一方，メッセージを受信する場合は，メッセージの受信先に関する情報が書かれたディスクリプタを受信キューに入れる．送受信のいずれのキューも，ドアベル方式でネットワークインタフェースに通知を行う．

ネットワークインタフェースは，ホストからの要求の処理を完了すると，ディスクリプタに対して完了を示すフラグをセットする．プロセスは，ディスクリプタの処理完了を示すフラグをポーリングするか，あるいは割込みを利用してネットワークインタフェースからの通知を受けることで処理の完了を検出する．また，それ以外に Completion Queue (CQ) を利用することもできる．CQ は，処理が完了したディスクリプタを指すポインタのキューであり，ネットワークインタフェースは，ディスクリプタの要求処理が完了すると，処理が完了したディスクリプタを指すポインタを CQ に格納する．プロセスは，ポーリングや割込みによって CQ の変化を検出することができる．また，すべての VI の処理完了を単一の CQ で管理することや同じ VI の各キューの処理完了を異なる CQ で管理するなど，各キューを CQ に柔軟に対応づけすることができる．CQ への対応付けは VI が作られたときに行われる．

VIA のモデルは，VI と CQ に加え，VI Provider (VIP) と VI Consumer (VIC) で構成される．VIP は VI を提供するネットワークインタフェースとカーネルエージェント (デバイスドライバ) で構成される．一方，VIC は VI の利用者であり，通常 VI を使用するアプリケーションプログラムとユーザエージェント (ユーザライブラリ) で構成される．ユーザエージェントは VI をアクセスするための関数や，VIP とのインタフェースとなる関数などを提供する．ユーザアプリケーションが VI を作るときはユーザエージェントが VIP の関数を呼び出し，リソースの確保などを行う．カーネルエージェントは，これ以外に VI の削除やコネクションの管理，CQ の生成や削除，プロセス

への割込み，メモリ登録，エラーハンドリングなどを行う．それ以外の処理はすべてユーザレベルで行うことになる．

VIA はメッセージ通信 (Send/Recv) 型の通信モデルと RDMA 型の通信モデルの両方を提供する．Send/Recv 型のモデルでは，受信側は事前に受信データをどこにいれるかを示したディスクリプタを受信キューに入れておくことで行う．その際，コネクション間でのフロー制御は VIC のレイヤで提供する．

一方 RDMA 型のモデルは，リモートリードもリモートライトも送信元のアドレスと受信先のアドレスの両方を指定する特殊な Send である．リモートライトでは，読み出し元を複数のバッファのリストにして，ローカルの分散したデータをリモートの一続きのバッファに書き込むことができる．また，リモートリードでは，読み出し先を複数のバッファのリストとして，リモートの一続きのバッファから読み出したデータをローカルの分散したバッファに書き込むことができる．いずれの場合も，RDMA 要求を発行するローカル側は，リモート側のバッファのアドレスを事前に知る必要がある．その代わり，受信側ではディスクリプタを受信キューに入れる必要がなく，通信終了時の通知も行われない．データ転送に用いる領域は，物理メモリにピンダウンして，物理アドレスをカーネルエージェント内のページテーブルに登録しておく．領域の再利用や解放は VIC の責任で行う．

Giganet や ServerNet II は VIA を直接ハードウェアで実装している．また，VIA の通信機構はソフトウェアによって提供することも可能であり，Berkeley VIA[108] のように Myrinet 上に実装したものや，Modular VIA (M-VIA)[109] のように Ethernet のネットワークインタフェースを用いて実装したものなどがある．M-VIA では，キューはホスト上のメモリに構築し，ネットワークインタフェース上のメモリでドアベルを実現する．

### 3.2.8 Genoa Active Message MAchine (GAMMA)

Genoa Active Message MAchine (GAMMA)[110] [111] は Genoa 大学で開発された Ethernet 向けの軽量通信ライブラリである．元々Fast Ethernet 用に開発され，後に GbE 向けに実装が行われた．以下ではまず 3Com 社の 3c595 および 3c905 向けに実装された Fast Ethernet 向けの GAMMA について述べる．

GAMMA の通信モデルは，送信側でメッセージハンドラを指定し，受信側でメッセージ受信後ハンドラが呼び出されて受信データの処理が行われる，AM の通信モデルを元にしている．

各プロセスは Active Port [112] と呼ばれる複数のポートを持ち，これを通じてメッセージの送受信を行う．送信処理は，プロセスが GAMMA のデバイスドライバ内の関数を呼び出すことで起動する．各メッセージは，デバイスドライバによって適宜分割され，連続した Ethernet のフレームとしてリモートに送信される．その際，ユーザ領域上のバッファからネットワークインタフェースの FIFO に対して直接 DMA でデータの転送を行うため，カーネルによるバッファリングは行われない．フレームのヘッダは，あらかじめポートをバインドする際に計算しておく．最後のフレームの送信が済んだらシステムコールから戻る．

一方，受信側では，通信に先立ってユーザ関数であるハンドラと受信バッファをポートに対してバインドしておく．受信バッファは，受信したメッセージをハンドラが起動するまで保持するのに用いる領域であり，メッセージが到着すると割込みに応じて GAMMA のドライバが動き，DMA を用いてこの領域に対して受信データを転送する．そのため，受信バッファは事前にピンダウンしておく必要がある．最後のフレームが受信されたら，GAMMA のドライバはポートに対応する

ハンドラを呼び出し，直ちにデータの処理を行う．その際，ドライバが一時的にコンテキストを切り替え，受信側のプロセスのコンテキスト上でハンドラが実行されるようにする．通常，プロセス内のハンドラは，メッセージをメインのスレッド内の領域に必要に応じて退避し，メインのスレッドにメッセージの到着を通知した上で，次のメッセージの受信用の領域をポートに対してバインドする．

このように，GAMMA では送信側と受信側の双方においてメッセージのゼロコピー通信が行われる．

GAMMA は，CRC によるエラー検出と通知を行うが，再送などのエラー回復やフロー制御は提供せず，エラーへの対処は上位アプリケーションの判断に委ねられる．また，GAMMA の通信レイヤは，大部分が Linux のカーネルレベルで実装されており，残りの一部はユーザライブラリに実装されている．

#### GAMMA on DEC 2114x

GAMMA は元々3Com 社の 3c595 および 3c905 向けに実装されていたが，ネットワークインタフェースの入手が困難となったなどの理由から，新たに descriptor-based DMA (DBDMA) と呼ばれる機構に対応した DEC 社の DC21140 向けに移植された [113]．DBDMA では，ネットワークインタフェースはホストメモリとネットワークの間の DMA を，ホストメモリ上に事前に用意した DMA ディスクリプタを用いて行う．これまでの GAMMA の実装では，ホスト CPU がネットワークインタフェースの DMA を直接起動し，DMA 開始後，CPU は次の DMA を始めるには今の DMA が完了するのを待たなければならない．そのため，長いメッセージを送る場合，CPU がブロックされてしまうという問題があった．DBDMA を用いた GAMMA では，送信側は要求されたメッセージを複数に分割し，分割した数分のディスクリプタをキューにいれて送り出す．その後は，ネットワークインタフェースがキューからディスクリプタを読み出して DMA 転送を行うため，ディスクリプタをキューに格納した時点でプロセスに戻るノンブロッキング送信が可能となる．

一方，DBDMA における受信処理では，ネットワークインタフェースは，受信データをホスト上ディスクリプタの指すバッファに対して DMA で書き込み，それが完了した上でホストに割込みをかける仕様となっている．割込みが発生した時点で受信データがホストメモリ上に存在することになるため，受信したデータをユーザプロセスに渡す際にメモリ間コピーが発生することになり，従来の実装で実現されていたメッセージのゼロコピー通信が不可能となる．

また，GAMMA では GO back N 方式で再送制御を行うことで順序性を維持するが，この方式ではパケットが消失した場合の再送量が大きいためネットワークの混雑が発生してしまう．GAMMA の想定する Ethernet 環境では，パケットの消失はネットワークでのエラーによって引き起こされることは稀であり，ほとんどの場合，受信バッファのオーバーフローによるパケットの破棄が原因である．DBDMA を用いた実装では，スイッチが IEEE 802.3x のフロー制御に対応していない場合にも受信バッファのオーバーフローが発生しないよう，クレジットベースのフロー制御が新たに追加されている．

#### GAMMA on GbE

GbE 向けの GAMMA は，当初 Packet Engines 社の G-NIC II と Netgear 社の GA620 に対して実装が行われた [8]．GA620 は NIC 上にプロセッサを持ちプログラマブルな環境を提供しているが

GAMMA はこれを利用しない実装となっている．

GbE 向けの GAMMA のプロトコルは，基本的に DC21140 向けの実装と大きな違いはない．GAMMA のプロトコル自体は特定のネットワークインタフェースの機能に依存しない作りとなっているため，Basic Interface for Network Device Drivers (BIND$^2$) と呼ばれる割込みやディスクリプタの扱いなどの個々のネットワークインタフェースで処理が異なる部分をまとめた小さなコードセットを用意し，既存のデバイスドライバを GAMMA に対応できるよう修正することで一般的な GbE のネットワークインタフェースに対して移植を行うことができる．現在では Intel 社の PRO/1000 シリーズや，Broadcom 社の Tigon3 チップなどがサポートされている．

### 3.2.9 まとめ

これまで述べたクラスタ向けの低レベル通信ライブラリは，Remote procedure call (RPC) 型の通信モデル，Send/Recv のメッセージ通信型の通信モデルおよび Put/Get などの RMA 型の通信モデルのいずれか，あるいは複数を組み合せたものを提供している．また，ユーザレベルでの通信呼び出しや通信時のユーザメモリ – カーネルメモリ間のメモリコピーを排除するゼロコピー通信などが取り入れられているものも多い．これら通信ライブラリのほとんどは 1990 年代後半に提案されたものであるが，PM や GAMMA などは現在でもそのままの形で最新の OS やハードウェアに追従し，利用されている．

2.3.2節で簡単に触れたが，RHiNET-2 のネットワークインタフェースコントローラである Martini は，PUSH および PULL のみをハードウェアで提供し，それ以外の通信処理はソフトウェアで実装するという方針をとっている．さらに Martini では PUSH でリモートプロセスのメモリにメッセージを書き込んだ際に，書き込み完了後に相手に割込みをかけるなどの通知手段が用意されておらず，受信側は，積極的にメモリをポーリングすることでメッセージの受信を検出することが前提となっている．このようなネットワークインタフェース上に構築されている低レベルな通信ライブラリは，これまで述べた中にない．Martini を用いた低レベルソフトウェアライブラリで上位にメッセージ通信などの通信モデルを提供する場合，高性能なハードウェアによる通信処理を積極活用するならば，これまで述べた低レベル通信ライブラリとは全く異なる通信機構の実装が必要となると考えられる．

# 第4章　Martiniの設計と実装

本章では，本研究の主題である Martini に関して，設計および実装の詳細を示し，構成や特徴に関して明らかにする．また，これにより，以降の章で述べる本研究における各提案や実装の背景を明確にする．

なお，本章で示す Martini の設計および実装について，筆者は乗っ取り機構を中心とした通信処理部分の設計・実装に関わっているものの，PUSH・PULL のプロトコル策定やメモリ保護方式の提案などに関しては携わっておらず，一部については RHiNET 全体のコンセプト同様，どのような経緯でこれらが決定したのか明らかではない．

## 4.1　Martini開発の経緯と設計の基本方針

1章や 2章で述べたように，Martini は RHiNET プロジェクトで開発されたネットワークインタフェースコントローラである．Martini は，元々RHiNET-2 および RHiNET-3 向けの一般的な PCI バス接続型のネットワークインタフェースコントローラとして開発される予定であったが，関連プロジェクトの要請などからメモリスロット装着型ネットワークインタフェースのプロトタイプや，OIP-SW の検証目的のネットワークインタフェースコントローラとしての機能も搭載することになり，多機能・多目的の ASIC として実装されることとなった．

また，RHiNET プロジェクトは，RHiNET-1/NI や RHiNET-2/NI0 の開発を通じて判明した性能向上の難しさ [42] などから，RHiNET-2 以降の RHiNET におけるネットワークインタフェースコントローラの実装コンセプトを見直し，これまでの “多様なプリミティブをすべてハードウェアで提供する” というものから，“単純で使用頻度の高いプリミティブのみをハードウェアで実装する” というものに大幅に方針転換を行った．Martini はこの基本方針に基づいて設計がなされている．

## 4.2　ノード間のメモリコピー

以下では，Martini がハードウェアで提供する基本通信機能であるノード間のメモリコピー，すなわち PUSH・PULL の設計と実装に関して述べる．これら機能の設計および実装は，RWCP および筆者の属する慶大のグループによって行われた．

### 4.2.1　通信プリミティブ PUSH・PULL

基本方針に従い，Martini はノード間のメモリコピーを行う PUSH と PULL の 2 種類のプリミティブのみをハードウェア実装する．まず，これらプリミティブのプロトコルについて述べる．

PUSH はローカルプロセスのメモリ上の連続したデータをリモートプロセスの指定の領域にコピーする機能を提供する．PUSH 要求が発行されると，Martini は指定されたサイズのデータをローカルノード上のローカルプロセスのメモリの指定領域から DMA で読み出し，書き込み先の領域

などが書かれたヘッダをつけて，push パケットとしてリモートノードに対して送出する．リモートノードでは，push パケットを受信すると，ヘッダに書かれたリモートプロセスの書き込み先アドレスに対して DMA でペイロードを書き込み，DMA 完了後に，必要に応じて PUSH 要求発行側のノードに確認応答として ack パケットを返す．ack パケットは，push パケットとほぼ同様に処理されるパケットであり，あらかじめ PUSH 要求を発行した際に指定したローカルノード上の指定メモリ領域に対して完了通知フラグを書き込む．

図 4.1は PUSH においてローカルプロセスのメモリの内容がリモートプロセスのメモリにコピーされるまでのデータの流れを示している．ホスト CPU からの書き込みアクセスで PUSH 要求が発行されると Martini は要求を解析し，ホストメモリからデータを DMA で読み出す (1)(2)．そして Martini はパケットヘッダとデータを結合してパケットとして送出する (3)．リモートノードは受信したパケットを解析し (4)，CPU に割込みをかけることなくデータをホストメモリに DMA で書き込む (5)．

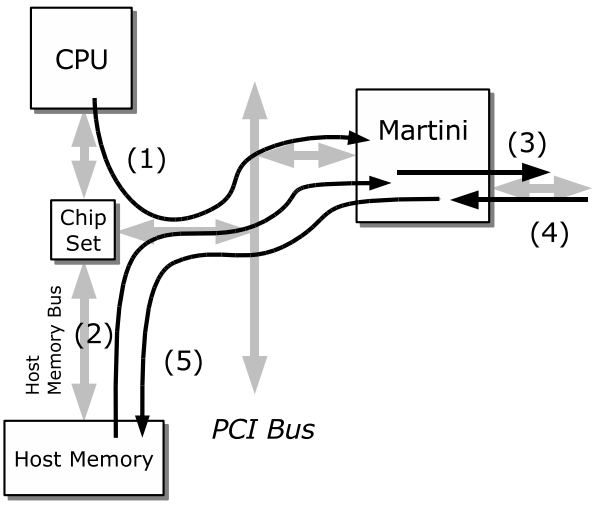



図 4.1　PUSH のデータの流れ

一方，PULL はリモートプロセスのメモリ上の連続したデータをローカルプロセスの指定の領域にコピーする機能を提供する．PULL 要求が発行されると，リモートプロセスの読み出し領域やローカルプロセスのデータの書き込み先が書かれたヘッダのみで構成される pull 要求パケットがリモートノードに送出される．リモートノードでは pull 要求パケットを受け取ると，ヘッダを読み取り，リモートプロセスの指定アドレスから DMA でデータを読み出し，ヘッダを付加して pull 応答パケットとしてローカルノードに返信する．pull 応答パケットを受け取ったローカルノードは，push パケットを受け取った場合とほぼ同様の手順でこれを受信処理し，ヘッダのアドレスで指定されるローカルプロセスの書き込み先に DMA でデータを書き込む．その後，PULL プリミティブ発行時にデータ受信領域とは別に指定したローカルプロセスのメモリ領域に対して，PULL 要求の完了を通知するフラグを書き込む．

PUSH および PULL では，転送指定サイズが RHiNET-2 や RHiNET-3 の Maximum Transmission Unit (MTU) である 2Kbyte を超える場合，複数のパケットに分割した転送が行われるが，通信要求を発行するユーザはこれを意識する必要はない．また，データのコピー元およびコピー先のアドレスは byte 単位で自由に指定可能であり，アラインに関する制約は存在しない．

### 4.2.2 ユーザレベル通信への対応と排他制御の回避

ノードが SMP 構成の場合などは，単一のノード上に複数の並列プロセスが同時に存在し，それぞれが同時にネットワークインタフェースにアクセスする可能性が考えられる．一般的な Ethernet のネットワークインタフェースなどで I/O 領域が 1 つしかない場合，OS がこれを仮想化・多重化してユーザプロセスに提供することで排他制御を実現するが，これに伴い通信オーバヘッドが生じてしまう．一方，Myrinet を用いたユーザレベル通信に対応した通信ライブラリでは，ネットワークインタフェースの SRAM 上の異なる範囲を個々のプロセスのアドレス空間にマップし，この SRAM への書き込みを介してネットワークインタフェースへの通信要求の通知などを行う設計とすることで，OS によるプロセス間の排他制御を不要としている．Martini でも，同様の手法をとることで排他制御を回避する．

Martini では，プリミティブ要求専用の小さなメモリ領域を複数用意し，ネットワークインタフェースを利用するプロセスのメモリ空間にカーネルの機能を利用してマップすることで，ユーザプロセスが Martini 内のメモリに直接要求を書き込めるようにした．その際，これら領域を，一般的なプロセッサの OS 上で利用可能なページサイズである 4Kbyte 単位にアラインして I/O アドレス空間上に用意することで，複数のプロセスに対して 1 個単位で領域をマップできるようにした．これにより，ある領域をマップした際に，同じページ内に別の領域が含まれることを回避し，あるプロセスが，別のプロセスが使用している領域をアクセスできないようにすることができる．ユーザプロセスからネットワークインタフェースの I/O 領域の一部分だけをアクセスできるようになることから，Martini ではこの領域を “Window” と称する．

また，Window の実体は Martini 内部のメモリであるため，たとえ複数のプロセスがそれぞれの Window に対して交互に 1 ワードずつ要求に必要なパラメータの書き込みを行ったとしても，書き込まれた内容は単に各 Window の実体であるメモリに書き込まれるだけであり，Martini に対する要求が上書きなどにより破壊されてしまうことはない．

このような機構により，OS による I/O の多重化・仮想化を用いたユーザプロセス間での排他制御が不要となる．

なお，各 Window に書き込まれた内容は，その後 Window の特定のアドレスにアクセスすることで Martini に渡され，処理が開始する設計となっている．Window の特定のアドレスに対してアクセスすることで Martini に Window に書いた要求を伝える処理を “キック” と呼ぶ．Martini はキックを検知した後，キックに用いられたアドレスや Window のメモリ内容を参照して，Window を介してユーザプロセスから発行された要求を判断し，それに応じた処理を開始する．

### 4.2.3 TLB によるアドレス変換

2.2.1節で述べたように，RHiNET のネットワークインタフェースにはアドレス変換機構が必要となる．そこで，Martini に対して，内部に Physical Address Translation Look-aside Buffer (PATLB) と呼ばれる TLB を設け，ユーザから指定された仮想アドレスを高速に物理アドレスに変換可能な機能を設けた．

また，PATLB のエントリがミスヒットした際のリプレースメント処理は，Martini の基本方針に従いハードウェア実装せず，ホスト上のデバイスドライバや Martini のオンチッププロセッサ上で動作するファームウェアで行えるような設計とした．

### 4.2.4 ローカルホストとリモートホストにおけるメモリ保護

ユーザレベル通信では，通信バッファはユーザプロセスによって指定されるため，OS のプロトコルスタックを用いたアドレスの検証を行うことができない．そのため，ネットワークインタフェースは OS に代わってメモリ保護機構を提供しなければならない．そこで，Martini のメモリアクセスを以下のポリシーで制限するものとし，設計を行った．

- プリミティブを発行したプロセスのピンダウンされたメモリ領域へのアクセスを許可
- プリミティブを発行したプロセスと同じ並列ジョブに参加しているリモートプロセスのピンダウンされたメモリ領域へのアクセスを許可
- それ以外のメモリ領域に対するアクセスはすべて禁止

Martini では，プロセス ID の管理方法が異なる可能性のある OS 間での通信に対応し，また同一並列ジョブ内のプロセスを確実に識別するために，OS によって提供されるプロセス ID の代わりに次の 2 つの値を導入し，これらを用いてメモリ保護を実現する．

- Process Group ID (PGID)

  各並列ジョブを識別するためのユニークな ID
- Process ID (PID)

  同一の並列ジョブに属している並列プロセスを識別するためのユニークな ID

図 4.2 は，ネットワークインタフェースに Martini を用いたクラスタシステムにおけるプロセス実行の流れの一例である．ユーザが並列ジョブの実行をシステムに対して要求すると，マスタタスクマネージャ(並列処理システム全体を管理するプロセス) が並列ジョブに対してユニークな PGID を割当て，並列プロセスの実行要求とそのプロセスに割当てる PID を，PGID とともに各ノード上のタスクマネージャ(スレーブタスクマネージャ) に対して送る (1)．各ノード上のスレーブタスクマネージャは，未使用のいくつかの Window をプロセス用に予約し (2)，Process Group ID Table (PGIDTBL) と呼ばれる Window と PID および PGID の対応を管理するテーブルの，予約した Window に対応するエントリに PGID と PID を書き込む (3)．PGIDTBL に適切な PGID と PID を書き込んだ後，スレーブタスクマネージャは並列プロセスを実行し (4)，プロセスは予約されている Window を自身のメモリ領域にマップする (5)．プロセスが Window を介して通信要求を発行すると，Martini は Window の ID を元に PGIDTBL を参照して PGID と PID を取得する．

このように PGIDTBL のエントリをスレーブタスクマネージャなどの特権プロセスのみが設定できるように制限し，ハードウェアが通信時にこれを直接参照することで，PID や PGID の詐称を防止する．

PATLB を参照する際のキーとして，仮想アドレスに PGID と PID を以下のように組み合わせて用いると，Martini のポリシーに沿ったメモリ保護が実現することになる．

- ローカルのメモリ領域にアクセスする場合，プリミティブを発行したプロセスの PGID と PID を用いてローカルの PATLB を参照する
- リモートのメモリ領域にアクセスする場合，プリミティブを発行したプロセスの PGID とリモートプロセスの PID でリモートの PATLB を参照する

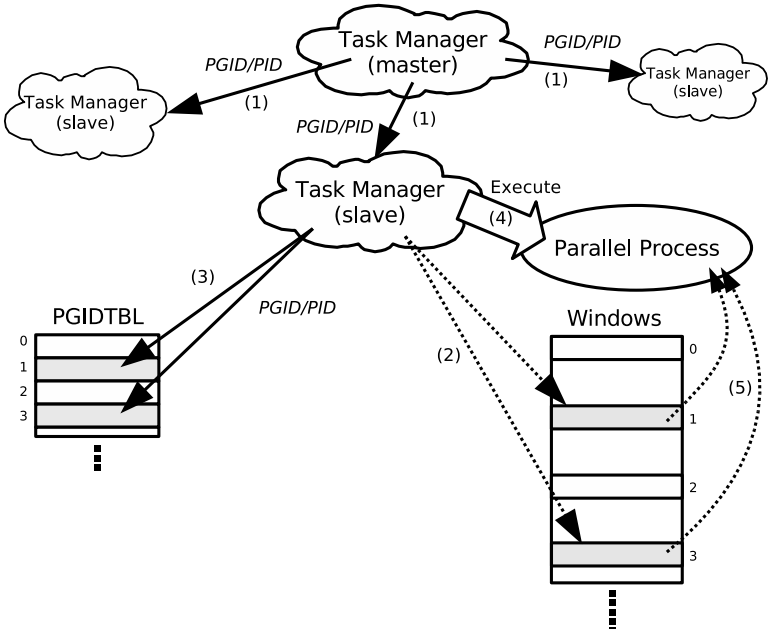



図 4.2　RHiNET を用いたシステムにおけるプロセス実行の一例

ただし，この場合，PATLB に不正な物理アドレスが登録されていないことを何らかの形で保証する必要がある．

### 4.2.5　リモートアドレスの抽象化

PUSH や PULL は，メッセージ通信と異なり，リモートライト・リードの対象となるリモートの領域を送信側で指定する通信モデルである．そのため，プリミティブを発行するプロセスは，通信に先立ち他のプロセスの変数や配列などのデータが配置されている領域のアドレスを知る必要がある．

しかしながら，リモートのデータが配置されている領域の仮想アドレスは，同一のプログラムを実行した場合でも動的に確保した場合などにプロセス間で異なる値となる可能性があるため，リモートのデータを指し示すのに仮想アドレスを使用する場合，ユーザプログラムは各データのアドレスを他の並列プロセスとの間で事前に教えあう必要が生じる．これはプログラムを複雑化し，また，実行開始時に多くの通信を必要とする．

そこで，これを回避するために，リモートの通信領域を指すための識別子として，OS によって提供される仮想アドレスの代わりに Segment ID (SID) と呼ばれる識別子を導入することにした．SID はポインタ配列に対するインデックスのような存在であり，各 SID はプロセス領域の通信バッファとして PUSH の書き込み先や PULL の読み出し元として用いるデータ領域の先頭を指す．SID とそれに対応する仮想アドレスをプロセス実行時に通信処理に先立って Martini 上に登録しておき，Martini が push パケットや pull 要求パケットを受信した際に，パケットのヘッダに書かれた対象プロセスの SID を仮想アドレスに変換し，それをさらに PATLB を介して物理アドレスに変換することで，SID を用いたリモートの仮想アドレスの抽象化が可能となる．さらにパケットヘッダにオフセット値を持たせ，SID から仮想アドレスを得た後，これにオフセット値を加えた上で物

理アドレスを得るようにすることで，配列などのデータ構造を扱いやすくすることができる．SID を用いることで，プログラマはリモートの領域をプログラミングの段階で指定可能となり，プロセス間での仮想アドレスの交換が不要となる．

このような SID から仮想アドレスへの変換は，push パケットや pull 要求パケットを受信するたびに発生するので，これを高速に行うために Martini 内部に SID–仮想アドレス変換用の TLB である Remove Virtual Address TLB (RVATLB) を設けることとした．

ローカルホストで PUSH 要求が発行されてからパケットが送出されるまでのアドレス変換を含む処理の流れの例を図 4.3に示す．図 4.3の例では，4Kbyte 単位にアラインされた Window のうち，ユーザのアドレス空間には ID が 1 のものがマップされている．ユーザが Window に，通信要求に必要となる各種パラメータを書き込み，キックを行うと，Martini 内部で Window の内容の処理が開始する．Martini は，まず Window の ID を元に PGIDTBL を参照し，Window を利用しているプロセスの PGID と PID を取り出す．次に，この値と，ユーザから指定のあった送信データの領域の仮想ページ番号をキーとして PATLB を参照し，対応する物理ページ番号を得る．これに，ユーザ指定の仮想アドレスのページ内オフセットを加え，通信データの置かれた領域の物理アドレスを得る．Martini は DMA を用いて，この物理アドレスの領域からデータを読み出し，別途作成しておいたパケットヘッダと接合して push パケットとしてネットワークへ送出する．

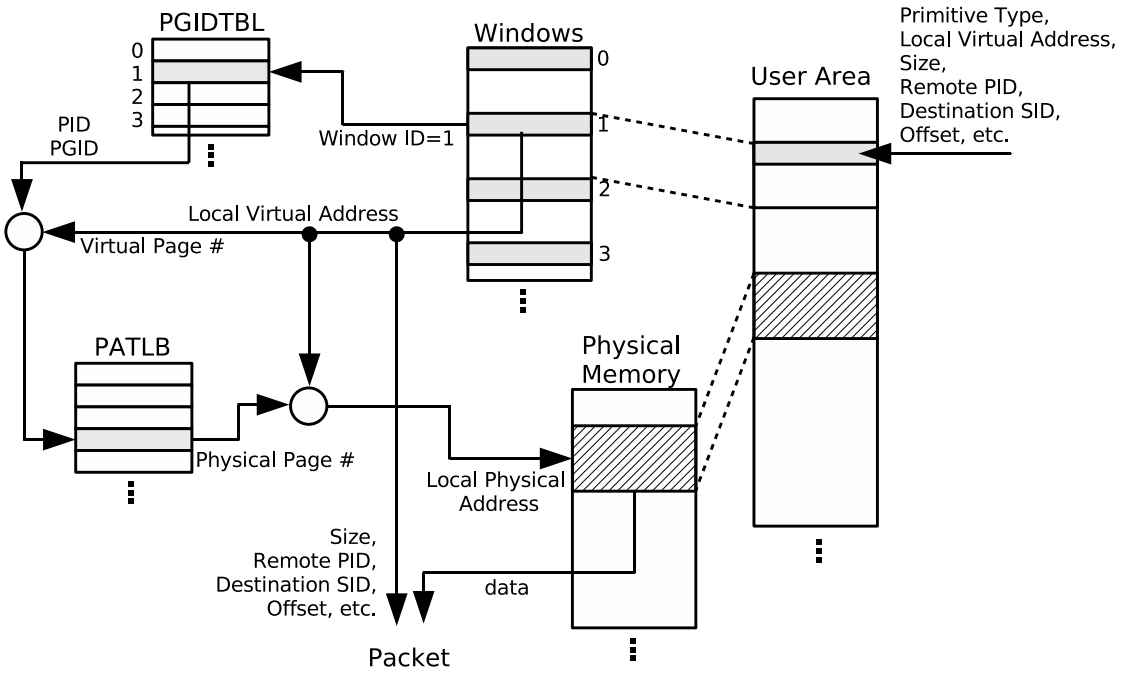



図 4.3　ローカル側でのアドレス変換

一方，リモートノードにおいて，push パケットを受信してから，データがリモートメモリに書き込まれるまでのアドレス変換の流れの例を図 4.4に示す．Martini は，まず受信した push パケットのヘッダに書かれている宛先プロセスの PID，PGID および書き込み先領域が含まれるセグメントの SID をキーとして RVATLB を参照する．RVATLB からは，SID に対応した宛先ユーザプロセスのセグメントの開始仮想アドレスが得られる．これに，同じくパケットヘッダに書かれた SID からのオフセット値を加え，書き込み先領域の仮想アドレスを生成する．この仮想アドレスのペー

ジ番号と，先ほど利用した宛先プロセスの PID と PGID をキーとして，今度は PATLB を参照し，書き込み先の物理ページ番号を取得する．この物理ページ番号に，仮想アドレスのページ内オフセットを加えることで，書き込み先の物理アドレスを取得する．取得した物理アドレスの領域に，Martini は DMA でデータを書き込む．

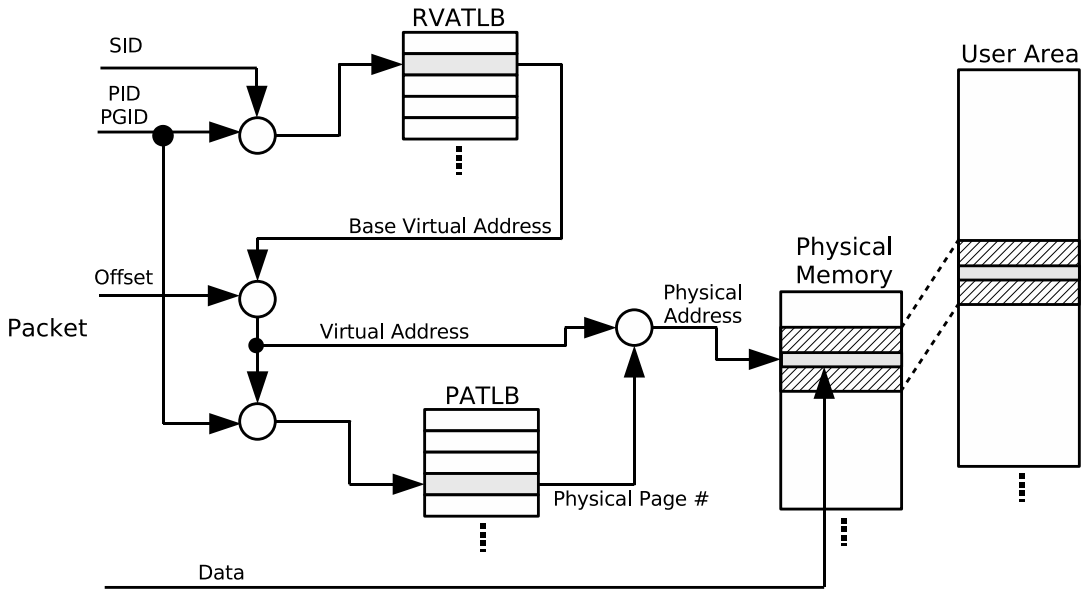



図 4.4　リモート側でのアドレス変換

## 4.3　PIO ベースのパケット生成

DMA 転送は，転送開始までのセットアップコストが大きいため，PUSH や PULL のような RDMA を用いた通信ではサイズの小さいメッセージの転送においてレイテンシが大きくなりやすい．これを削減するために，Martini では Block On-the-fly (BOTF) と Atomic On-the-fly (AOTF) の 2 種類の On-the-fly (OTF) 通信機構と呼ばれる Programmed I/O (PIO) ベースのパケット送出機構を実装した [44]．どちらの OTF 通信機構もユーザレベルで利用可能であり，メモリ保護を維持したまま任意のヘッダを持ったパケットを，DMA を用いずに送り出すことができる．これを用いて push パケットを発行することで，PUSH を用いた場合に発生する送信側での DMA セットアップコストを回避でき，メッセージサイズが小さい場合にレイテンシを低く抑えることが可能となる．以下でこれら 2 つの OTF 通信機構について述べる．

### 4.3.1　BOTF

BOTF は Window に書き込まれた内容をそのままパケットとしてネットワークへ送出する PIO ベースのパケット送信機構である．ユーザプログラムは，ヘッダとペイロードにより構成されるパケットイメージ全体を構築し，PUSH や PULL のパラメータを書く代わりにこれを Window に書き込む．そして Window 領域の BOTF 用の特殊なアドレスに書き込みアクセスを行うことで，書

き込まれたデータの BOTF での送出を Martini に要求する．

BOTF では，任意のパケットヘッダのパケットを送出することができる．しかしながら，ユーザが任意のヘッダを自在に指定できてしまうとメモリ保護が破綻してしまう．そこで，Martini は，BOTF によるパケット送出の際，パケットヘッダの PIGD の部分のみ PGIDTBL を参照して得た Window の ID に対応する値で上書きして送り出す設計とした．これにより，PGID だけは詐称できなくなり，他の並列ジョブのプロセスのメモリ領域に対して干渉することを防ぐことが可能となる．

### 4.3.2 AOTF

AOTF は Martini に対して 1DW(注 1) のデータを書き込むことで，パケットを送出する PIO ベースのパケット送信機構である．パケットのヘッダは事前に Martini 上に登録しておき，これに書き込まれたデータをペイロードとして接合して送出する．BOTF と異なり，ペイロードのサイズは 1DW に制限されるが，事前に複数種類のパケットヘッダを Martini の AOTF 用のバッファ(Header Buffer) に登録しておくことで，多様なパケットを 1DW の書き込みだけで送出することが可能となる．

AOTF には，Window とは異なる，複数ページに渡る AOTF 専用の I/O 領域を用いる．この領域に対するデータの書き込みを検出すると，Martini は書き込みアクセスがあった I/O 領域のアドレスのページ番号を元にパケットヘッダを選択し，これを書き込まれたデータと結合してパケットとして送り出す．その際，書き込まれたアドレスから Header Buffer 上のヘッダ登録位置を参照するためのアドレス変換には，AOTF 専用の TLB(Header TLB) を用いる．

AOTF では，レイテンシを最小限に抑えるために，パケット送出の際に PGIDTBL の参照などは行わないが，ユーザが事前に Martini にデバイスドライバなどを介してパケットヘッダを登録する際にヘッダの検閲を行うことでメモリ保護を維持する．

図 4.5は AOTF におけるパケット生成の様子を示している．AOTF の I/O 領域に書き込みが行われると，Martini は書き込みアドレスのページ番号の下位 16bit を用いて Header TLB を参照し，Header Buffer のアドレスを得る．このアドレスを元に Header Buffer からヘッダを取得し，書き込まれたデータを結合してパケットを構築して送出する．

なお，図には示していないが，ホストからの書き込みアクセスに対して AOTF によるパケット発行が遅い場合を考慮して，連続した AOTF 要求に対応できるよう，AOTF の I/O 領域への書き込みは Martini 内でキューイングする．

AOTF では，事前に push パケットのヘッダをヘッダバッファに登録しておくことで，AOTF 領域に対する 1 回の書き込みで push パケットを送出することができる．その際，書き込んだ先のアドレスの下位 12bit で，ヘッダバッファに登録してあるパケットヘッダの SID からのオフセット値を上書きすることができる．これにより，SID がリモートのページの先頭を指している場合，VMMC[97] の automatic update や Memory Channel[87] のように，リモートのユーザ領域のページに対して，そのページがあたかもローカルにあるかのような透過的な書き込みアクセスが可能となる．

(注 1)1DW は 64bit.

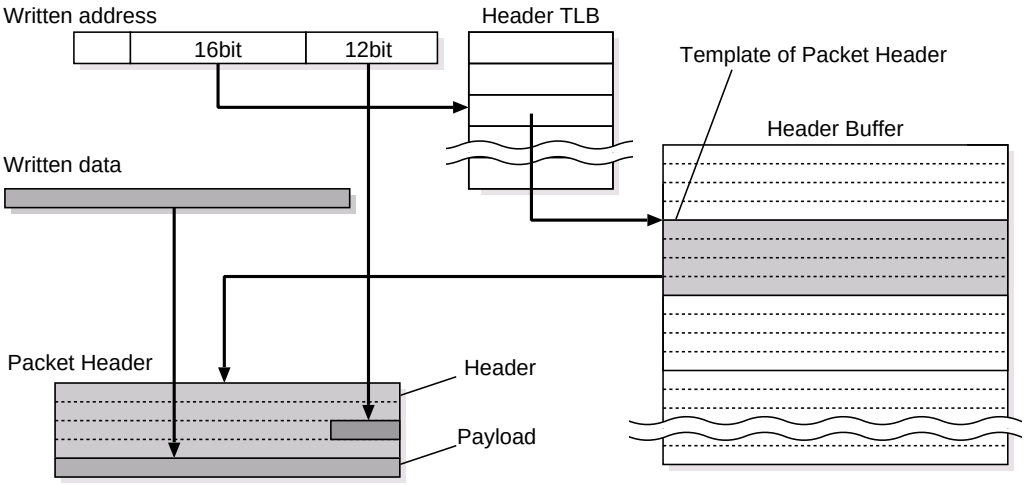



図 4.5　AOTF によるパケットの生成

## 4.4　乗っ取り機構

本研究において新規に提案を行った乗っ取り機構 [27][28] はハードウェアの一部のモジュールのステートやレジスタを，オンチッププロセッサから自由にアクセス可能とすることで，オンチッププロセッサとの間でステートレベルでの協調処理を実現する機構である．

Martini では，PUSH・PULL の 2 つのプリミティブは完全にハードウェアによって処理されるが，それ以外の通信処理や PATLB のミスヒットなどが発生した場合，ハードウェアは処理を続行できずに停止し，オンチッププロセッサなどがソフトウェアでこれに対処することになる．このようなソフトウェア処理では，部分的にハードウェアと同内容の処理を伴う場合がある．また，このような状況下では，ハードウェア自体は処理を続行できずに停止していることから，ハードウェアのリソースは丸々空いていることになる．そこで，停止しているハードウェアの空きリソースをソフトウェアから利用することで，ソフトウェア処理の効率を上げることができると考え，本研究ではこのようなリソースの利用を実現するための仕組みとして乗っ取り機構を提案し，Martini に対して実装を行った．

乗っ取り機構に関する詳細は，別途 7章にて述べる．

## 4.5　メモリバスを介したホスト接続

DIMM スロットを介してホスト PC に接続するための仕組みは，RWCP と東京農工大学を中心とした研究グループによって提案がなされ，日立 IT によって実装が行われた．これに関する各種機能の詳細は文献 [43] にて述べられている．

## 4.6　Martini の実装

### 4.6.1　Martini の構成

Martini のブロック図を図 4.6に示す．

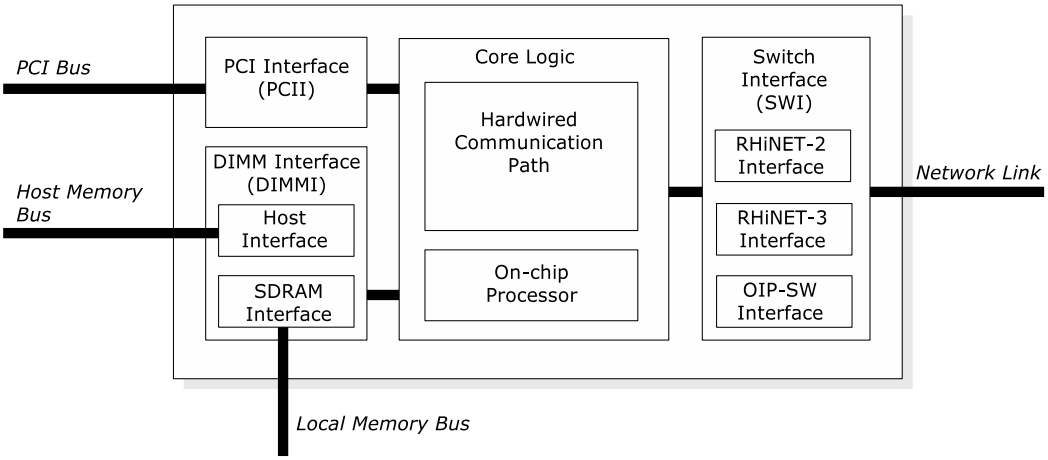



図 4.6　Martini のブロック図

Switch Interface (SWI) はクロックレートの変換やパケットの符号処理，フロー制御，仮想チャネルの管理などのネットワークのリンク関連の処理を行う．SWI は，Martini が接続可能な RHiNET-2/SW，RHiNET-3/SW および OIP-SW の 3 種類のリンクプロトコルに対応する．

PCI Interface(PCII) は Martini を PCI バスに接続する際に用いられ，DIMM Interface(DIMMI) は Martini をメモリバスに接続する際に用いられる．また，DIMMI は，外部 SDRAM へのインタフェースも搭載している．PCII は 32bit/33MHz から 64bit/66MHz までの PCI バスに対応する．DIMMI は PC100 および PC133 規格の SDRAM に対応する．

コアロジック (図中の NIC Core Logic) は Martini の中核部分であり，アドレス変換やパケット構築などの PUSH・PULL プリミティブと関わりの深い多くの機能がここで処理される．オンチッププロセッサもコアロジック内に位置する．

これら各構成要素のうち，スイッチと共通部分の多い SWI は RWCP によって，また，DIMMI および PCII に関しては日立 IT によって実装が行われた．一方，コアロジックに関しては，RWCP と筆者ら慶大のグループとで共同で実装を行った．以下ではコアロジック部を中心に実装を述べる．

### 4.6.2　コアロジックの構造

Martini のコアロジックは Hardwired Communication Path (HCP) と呼ばれるハードワイヤードロジックと，オンチッププロセッサで構成される．

HCP は，ホストなどからのパケット発行要求への対応を行う Initiator Controller (ICONT)，ネットワークからの到着パケットの処理を行う Remote Controller (RCONT)，各モジュール間の DMA 転送を制御する DMA Controller (DMAC) および PATLB の，4 つのブロックに分けて実装を行った．図 4.7にこれらブロック間のつながりを示す．

#### Initiator Controller

ICONT は Window と Send Controller で構成される (図 4.8)．4.2.2節でも述べたように，Window はユーザからの要求を受け付けるための，ユーザ空間にマップされるメモリであり，ユーザプロ

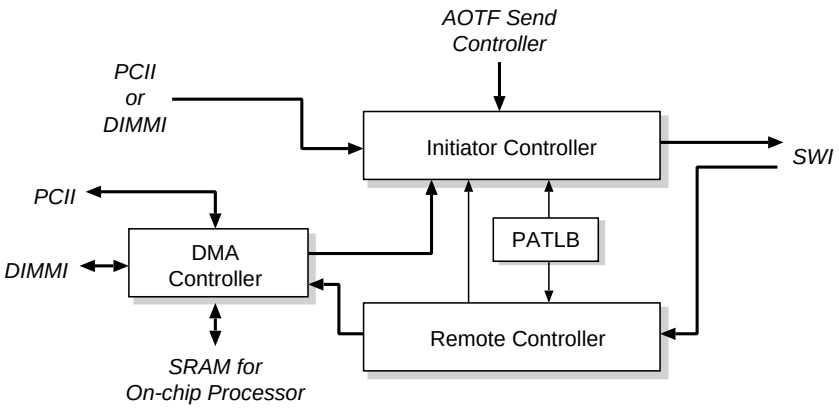



図 4.7　HCP のブロック図

セスが PUSH や PULL，BOTF によるパケット送出などの要求を行う際に利用される．Window がキックされると，Window に付随するコントローラによって要求のパラメータ (BOTF の場合はパケットイメージ) が FIFO を経て Send Controller に転送される．

Send Controller 内部は Initiator と Replier の 2 つのモジュールで構成される．Initiator は PUSH・PULL の 2 種類のプリミティブと BOTF の処理を行い，Replier は RCONT 側からの ack パケットや pull 応答パケットの生成要求の処理を行う．これら 2 つのモジュールの機能はほぼ同一であるが，デッドロックを回避するために，異なるモジュールとして実装した．

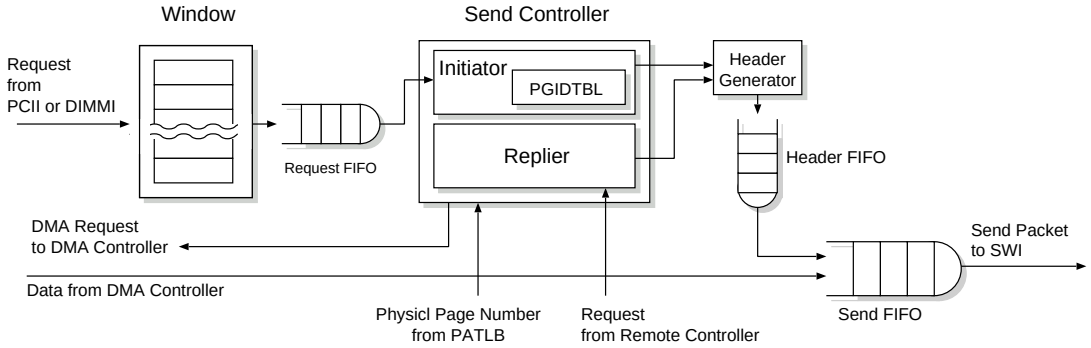



図 4.8　Initiator Controller のブロック図

Window を介して発行された要求は，Initiator に転送される．Initiator は，まず要求の発行に用いられた Window の ID より PGIDTBL を参照して，Window に対応した PGID と PID を取得する．

もし，要求が PUSH か PULL であった場合，PGIDTBL から読み出した PGID と Window に書かれているパラメータを元に Header Generator でパケットヘッダを生成する．これらヘッダを FIFO (Header FIFO) に格納し，要求が PUSH の場合は，DMA で読み出されるペイロードとの待ち合わせを行う．要求が PULL の場合は，ヘッダの待ち合わせをせずに，そのまま送信用の FIFO (Send FIFO) に転送する．

要求が PUSH であった場合，Initiator はヘッダ生成と同時に PATLB を参照してローカルプロセスの通信領域の物理アドレスを取得しておく．その後，DMA 要求を DMA コントローラの要求

FIFO に格納し，ペイロードとなるデータがホストから読み出されたら，ヘッダ FIFO のヘッダと結合して Send FIFO を経由して送出する．

PUSH で送信を要求されたデータサイズが大きく，RHiNET での MTU(2Kbyte) を超えるような場合は，Initiator が適宜分割を行い，push パケットを複数回送出する．また，PUSH で送出するデータが格納されている領域が複数ページにまたがっている場合，Initiator は繰り返し PATLB にアクセスし，連続して DMA 要求を発行する．このとき，DMA の要求発行と次のパケットのヘッダ生成処理との間に依存がないことから，これらはパイプライン動作する設計としてある．

一方，要求が BOTF であった場合，Initiator は PGIDTBL より得た PGID で，Window から読み出したパケットイメージのヘッダの PGID が格納されるフィールドを上書きし，できあがったパケットイメージを Send FIFO 経由で送出する．

ホストからの要求がこれら以外のハードウェア非対応のものであった場合は，Initiator は例外を発生し，オンチッププロセッサなどに割込みをかけて処理を依頼する．この仕組みを利用することで，ユーザプロセスから直接オンチッププロセッサの処理を呼び出すことができ，PUSH・PULL 以外の通信プリミティブに関してもユーザレベルでの要求発行を実現できる．

Window に書かれた内容の処理が完了したら，Martini は，Window に書いた要求の処理が完了し，次の処理の要求の発行が可能になったことをホストに通知する．通知は，あらかじめ Window ごとに指定されたホスト上のメモリ領域に対する DMA によるフラグ (continue flag (contflag)) の書き込みで行う．contflag の書き込み先は，事前にホストから PGIDTB に付随する Continue Flag Table (CONTTBL) と呼ばれるテーブルに格納しておき，PGIDTBL と同様に Window の ID で参照することでアドレスを得る．その際，CONTTBL には完了フラグの書き込み先の物理アドレスが登録されているものとし，アドレス変換をせずにこの値を完了通知の際の DMA 転送の要求に直接用いる．これは，contflag に用いる領域はホストプロセスから見ると Window に付随する I/O 領域の一部と考えることができ，Window に要求を発行するたびにアドレスが変更になるような使い方は想定されないためである．

### Remote Controller

RCONT は処理が複雑であることから，パケットヘッダの解析を主に行う Receiver Front-end (RFend) と呼ばれるフロントエンド部と，仮想–物理アドレス変換や DMA 要求発行などを行う Receiver Back-end (RBend) と呼ばれるバックエンド部に分けて実装した (図 4.8).

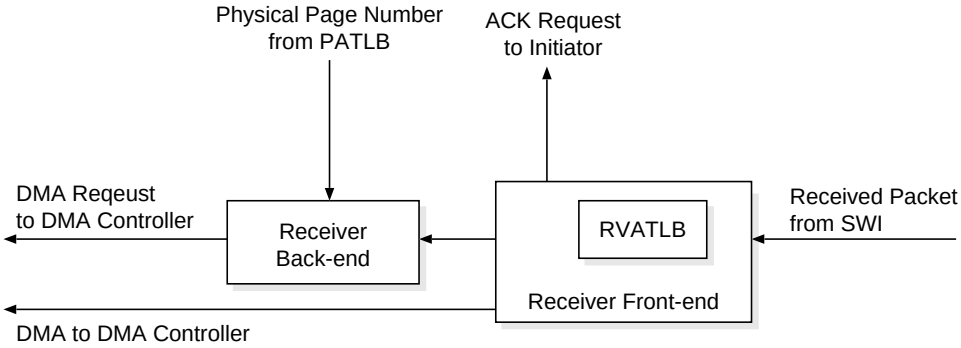



図 4.9 Remote Controller のブロック図

ネットワークから到着したパケットは，SWI を経由して RCONT に転送される．まず RFend にて，転送されたパケットのヘッダの解析を行う．

受信したパケットが push パケットであった場合，RFend は RVATLB を参照してパケットヘッダから読み取った SID を書き込み先領域の仮想アドレスに変換し，他のヘッダのパラメータとともに RBend に渡す．一方，受信したパケットが ack パケットや pull 応答パケットであった場合，パケットヘッダに書かれているアドレスはプリミティブ発行時に指定された仮想アドレスなので，RFend はアドレス変換を行わずに，各種ヘッダのパラメータを RBend に渡す．RBend は，RFend から渡されたデータ書き込み先の仮想アドレスを PATLB を参照して物理アドレスに変換し，SWI の受信バッファから，データ書き込み先の物理アドレスへの DMA 転送要求を DMA コントローラに発行する．さらに，push パケットのヘッダに確認応答を要求するフラグが設定されていた場合，ack パケットを生成する必要があるので，RFend は push パケットに対応する DMA が完了するのを待ち，その後ヘッダに書かれている ack 書き込み先のアドレスなどとともに ack パケットの送出を ICONT の Replier に要求する．なお，ユーザが送受信アドレスのアラインを気にせず任意のアドレス間の転送を実現できるよう，Martini では受信時の DMA 転送の最中に RFend 内の J-joint[114] と呼ばれるモジュールでアラインメントの調整を行う実装となっている．

一方，受信したパケットが pull 要求パケットであった場合，RFend は RVATLB を参照してパケットヘッダの SID を読み出し対象となる領域の仮想アドレスに変換した後，他のヘッダのパラメータとともに pull 応答パケットの発行要求を ICONT の Replier に伝える．Replier は，pull 応答パケットの発行要求を受け付けると，Initiator が PUSH 要求を処理するのとほぼ同様の手順で DMA 要求発行などを行い，パケットを送出する．

到着したパケットのタイプが PUSH や PULL によって生成されるもの以外のハードウェア非対応なものであった場合，RFend は例外を発生し，オンチッププロセッサなどに割込みをかけて処理を依頼する．これを利用することで，BOTF などを利用して発行した独自形式のパケットを，ホスト CPU やオンチッププロセッサでソフトウェア処理することが可能となる．

#### PATLB

PATLB は ICONT と RCONT によって共有されるため，これらモジュールから独立したモジュールとして実装した．

PATLB は 2-way セットアソシアティブの構造とし，エントリ数は全体で 2048 とした．64bit のアドレス変換にも対応しており，ホストが 64bit OS の場合は，1024 エントリのダイレクトマップの構造となる．ミスヒットが発生すると，ICONT や RCONT が例外を発生し，オンチッププロセッサかホスト CPU に対して割込みをかけ，リプレースメント処理を依頼する．

なお，RVATLB の構造も PATLB とほぼ同様であるが，RVATLB は RFend からしか参照されないため，独立したモジュールとして実装せず，RCONT 内部に実装を行った．

#### DMA Controller (DMAC)

DMAC は PCII，DIMMI，内蔵 SRAM および SWI の送受信 FIFO の各モジュール間の DMA 転送の制御に対応し，これによりホストメモリ，外部 SDRAM，オンチッププロセッサ用内蔵 SRAM およびネットワークの任意の組み合わせでの DMA 転送を実現する．図 4.10に DMAC の接続を示す．また図 4.11に DMAC の内部構造を示す．

DMAC では転送対象に 1)PCII，2)DIMMI，3)SRAM，4) 送受信バッファという固定的な優先順

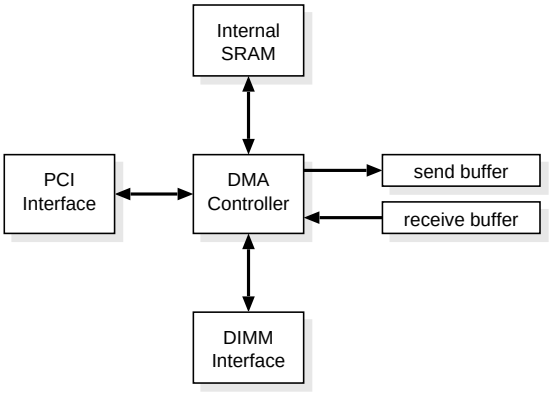



図 4.10　DMA Controller の転送対象

位をつけ，制御を行うこととした．この優先順位は DMA の要求を出した際に，実際に DMA が実行されるまでに必要とする時間に従って決定した．転送は常に優先度の高い方 (これをマスタと呼ぶ) のデータ転送が実際に開始されてから，優先度の低い方 (これをスレーブと呼ぶ) に対して要求を出すことによって 2 つの対象間で DMA を行う構造となっている．このような構造をとることで，各転送対象を占有する時間を最小限に抑え，DMA 転送を効率よく行うとともに，重ならない対象間の DMA の同時実行を可能とする．

また，各転送対象に対して要求を出すモジュールと実際にデータ転送を行うモジュールを独立して持ち，DMA 要求・転送の動作をパイプライン化する事で DMA の高速実行を実現する．図 4.11中の PCI DMA Request Arbiter，DIMM DMA Request Arbiter，SRAM DMA Request Arbiter はそれぞれ PCII，DIMMI，内蔵 SRAM との DMA の要求のアービトレーションを行う．

PCI DMAC は PCII と，DIMM DMAC は DIMMI と，SRAM DMAC は内蔵 SRAM とのデータ転送を制御し，DMA 転送の対象に応じて動的にパスの切り替えを行う．送受信バッファへの DMA 要求およびデータ転送制御は ICONT と RCONT が行うため，そのためのアービタや DMAC は存在せず，PCI DMAC，DIMM DMAC，SRAM DMAC との間で直接データ転送を行う．

### オンチッププロセッサ

Martini のオンチッププロセッサは，32bit の MPIS R3000 命令互換の RISC プロセッサであり，Martini 向けに新規に実装を行った．5 段のパイプラインのシンプルな構造で，命令の拡張などは行っていない．命令メモリとデータメモリに Martini 内部の SRAM を用い，それぞれの容量を 128Kbyte とした．

Martini では，内部のデータの処理単位がパケットの 1 フリットに相当する 64bit となるため，各ハードウェアモジュールのレジスタは 64bit 幅となっている．これを 32bit のオンチッププロセッサから利用するために，オンチッププロセッサ周辺にバッファやシフタなどを持つメモリコントローラを設けた．また，各ハードウェアモジュールからの割込みに柔軟に対応するために，複雑なマッピングやマスクに対応した割込みコントローラを設けた．

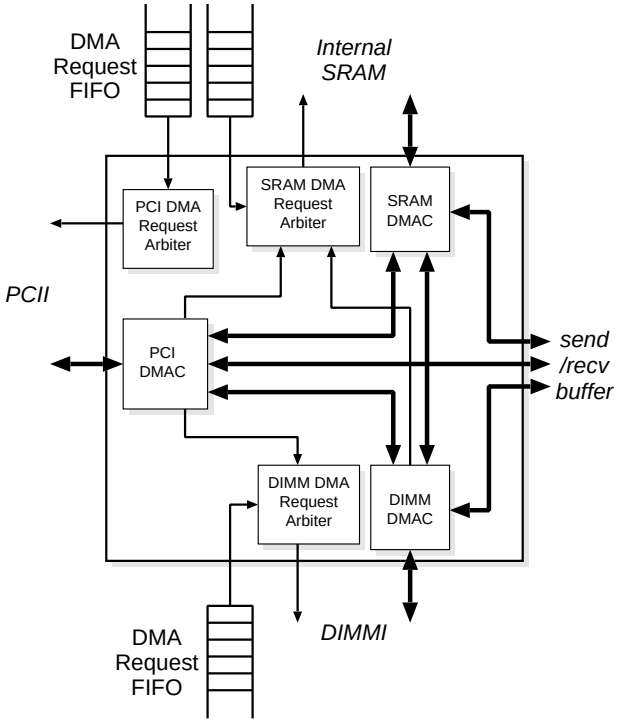



図 4.11　DMA Controller ブロック図

### 4.6.3　Martini のチップ実装

表 4.1 に Martini チップの諸元を示す．Martini のチップ実装には，日立デバイス開発センタ [115] の 0.14$\mu$m CMOS エンベデッドアレイテクノロジを用いた．複数の I/O コントローラを内蔵している関係で，内部は複数種類のクロックで動作する構造となっている．コアロジックは PCI バスと同じ 66MHz で動作する．

表 4.1　Martini の諸元

| Item | Value |
|---|---|
| Design Rule | 0.14$\mu$m |
| Die Size | 272.91mm$^2$ |
| Total Memory Size | 620Kbyte |
| I/O Frequency | |
| RHiNET-2 | 800MHz |
| RHiNET-3 | 1.25GHz |
| OIP-SW | 250MHz |
| Core Frequency | |
| Core Logic | 66MHz |
| DIMMI | 133MHz |
| SWI | 125MHz |
| Package | 784BGA |
| Power Consumption | |
| MAX | 14.9W |
| TYP | 11.3W |

Martini のゲート数の内訳を表 4.2に示す．Martini 全体では，回路規模は NAND 換算で 2,221k ゲートとなった．処理の複雑な HCP 部が回路の約半分を占めていることがわかる．

Martini のメモリ容量の内訳を表 4.3に示す．Martini の搭載する内蔵メモリの総容量は 620Kbyte となった．オンチッププロセッサの命令・データメモリが半数近くを占めている．

Martini のチップ内のレイアウトを図 4.12 に示す．網がけされている領域は内蔵メモリを示している．図より Martini の面積の半分近くがメモリによって占められていることがわかる．

表 4.2　Martini のゲート数の内訳

| Name | Gates |
|---|---|
| PCII | 116k |
| DIMMI | 170k |
| Core Logic | |
| Hardwired Communication Path | 1,190k |
| Initiator Controller | (593k) |
| Remote Controller | (66k) |
| PATLB | (15k) |
| DMA Controller | (300k) |
| AOTF Communication Processor | (202k) |
| Misc. | (14k) |
| On-chip Processor | 143k |
| SWI | 387k |
| Misc. | 215k |
| Total | 2,221k |
| After Layout | 3,624k |

表 4.3　Martini のメモリ容量の内訳

| Name | Size (byte) |
|---|---|
| SWI | 20K |
| Core Logic | |
| HCP | 168K |
| On-chip Processor | 288K |
| Misc. | 144K |

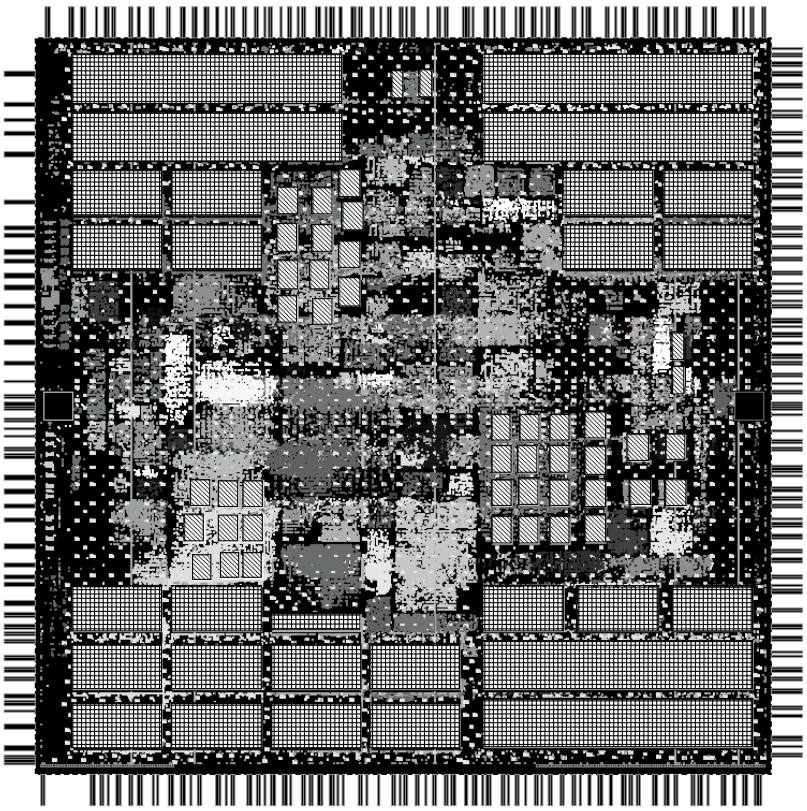


図 4.12　Martini のレイアウト

# 第5章　Martini向け<br>低レベルソフトウェアライブラリ

本章では，本研究において実機上で Martini を用いたシステムを構築するにあたり必要となり，筆者が実装を行った低レベルソフトウェアライブラリの設計および実装について詳細を述べる．本章で示すメモリ保護機構は，4章で述べた Martini のハードウェアによるメモリ保護の仕組みを補完する．

## 5.1　Martiniにおける低レベルソフトウェアライブラリの必要性

4章で述べたように，Martini のハードウェアはメモリ保護を伴うユーザレベル通信に対応した設計となっているが，実際にメモリ保護を実現するには，OS やその他の低レベルなソフトウェアによるサポートが必要となる．また，Martini のハードウェアは，例外が発生した場合，オンチッププロセッサやホスト CPU に対して割込みをかける設計となっているため，この対処を行うソフトウェアも必要となる．

また，Martini は，ハードウェア単体で PUSH および PULL の 2 種類の通信プリミティブを提供するが，ユーザレベルで PUSH と PULL のみを用いて Message Passing Interface (MPI) などの上位の通信ライブラリや分散 OS などを実装する場合，処理によってはポーリングが頻発したりメモリを大量に消費するプロセスが必要になるなど，効率的な実装を行えない事態が発生する可能性がある．そのため，あらかじめ Martini 上のオンチッププロセッサによる処理などを組み合せ，ユーザレベルで実装するよりも効率的に動作する PUSH と PULL 以外のプリミティブを用意しておくのが望ましい．

このような背景から，本研究における Martini を用いた各種実装や評価の基盤となる低レベルソフトウェアライブラリの実装を行った．以下ではこれらの設計・実装について述べる．

なお，Martini は特定の OS に依存せず，64bit アーキテクチャの PC にも対応する実装となっているが，本研究において実装したライブラリは，x86 アーキテクチャの Linux での利用を前提としている．また，本ライブラリは PCI バスを介してホスト CPU に接続された Martini で用いることを想定しており，DIMMnet-1 での利用は考慮していない．

## 5.2　ソフトウェアの階層

Myrinet などのオンチッププロセッサを搭載するネットワークインタフェースを用いたクラスタ向け通信ライブラリの多くは，以下の 3 つのソフトウェア層を利用して実装されている．

- ネットワークインタフェースのプロセッサ上で動作するファームウェア
- カーネル内部に組み込まれて動作するデバイスドライバ

- 並列プログラムなどと直接リンクして実行されるユーザライブラリ

Martini 用の低レベルソフトウェアライブラリも，これら 3 つの層に渡って実装を行った．各レイヤに関する概要を以下で述べる．

- ファームウェア

  ファームウェアは Martini のオンチッププロセッサ上で動作するソフトウェアである．ホスト CPU と完全に並列動作することが可能であるため，ファームウェアに処理を実装することでホスト CPU でのオーバヘッドの発生を防ぐことができる．しかし，一方で，Martini のオンチッププロセッサの処理能力はホスト CPU に比べて大幅に低いことから，利用頻度の高いプリミティブなどの処理をファームウェアのみで実装してしまうと，十分な通信性能が得られなくなる可能性がある．

  Martini では，オンチッププロセッサにも専用の Window が用意されており，ホスト上の並列プロセスと同様に Window を介して PUSH や PULL のプリミティブを発行したり BOTF でパケットを送出したりすることが可能である．また，DMA コントローラや HCP のパケット送受信用の FIFO などに対してアクセスできるため，より低いレベルでのプロトコル処理をソフトウェアで実現することも可能である．

- デバイスドライバ

  デバイスドライバはカーネル内に組み込まれ，カーネルモードで動作するソフトウェア層であり，システムコールなどのカーネルの提供する機能の拡張や，特権レジスタへのアクセスの検査などを実装するのに適している．

  Martini では，データのブロック単位でのバッファリングやソケット型のインタフェースの提供などが不要であることから，OS に対してはキャラクタデバイスとして認識させ，ユーザプロセスからはデバイスを `open` し，`mmap` や `ioctl` を用いてアクセスすることでデバイスドライバ内の関数を呼び出すことにした．

- ユーザライブラリ

  ユーザライブラリは，上位の通信ライブラリや分散 OS の管理プログラムなどと直接リンクして呼び出されるソフトウェア層である．ユーザライブラリは，ユーザに対してデバイスドライバやファームウェア，ハードウェアなどの下位のレイヤが提供する複雑な手順を必要とする機能を，簡単な API の呼び出しの形に抽象化して提供する．また，これらを組み合せたより高度な通信処理を提供するなど，上位レイヤの実装における負担を軽減するのに用いる．

## 5.3 メモリ保護

以下では Martini の低レベルソフトウェアライブラリによるメモリ保護機構の実装について述べる．

### 5.3.1 ページテーブル

Martini では，ネットワークインタフェース上で仮想アドレスから物理アドレスの変換を行う際に PATLB を用いるが，ユーザプロセスが不正な仮想アドレスを伴った PUSH や PULL の通信要求を発行した場合，PATLB でミスヒットが発生する．また，正常な仮想アドレスからの変換においても，PATLB 上にすべての仮想アドレスと物理アドレスの対応を同時に保持することができないため，同様にミスヒットが生じる可能性がある．

PATLB のミスヒットが発生した場合，正しい物理アドレスをプロセスのページテーブルから取得して，PATLB のリプレースメント処理を行う必要がある．PATLB のミスヒットが発生すると，ハードウェアモジュールは例外を発生し，オンチッププロセッサやホスト CPU に対して割込みの形で報告を行うが，5.4節にて述べるように，例外処理はすべてファームウェア上で処理する方針を採用したことから，PATLB のリプレースメント処理もファームウェアで実現することとした．

ファームウェアからアクセスしやすく，大容量の領域を確保できるよう，Martini が利用するページテーブルは，ネットワークインタフェースの SDRAM 上に構築することにした．PATLB がミスヒットした場合，ファームウェアは SDRAM 上のページテーブルから DMA 転送を行うことで物理アドレスを取得してリプレースメント処理を行う．

ページテーブルをネットワークインタフェース側で保持するのは，Martini がアクセスできる領域を，事前にピンダウン処理がなされ物理メモリ上に存在することが保証されている仮想アドレス空間に限定するためである．ホスト上でのピンダウン・アンピンダウン処理にあわせて SDRAM 上のページテーブルを更新することで，これを保証することができる．

### 5.3.2 ピンダウン・アンピンダウン処理

2.2.1節で述べたように，PUSH や PULL などで DMA 対象となるユーザプロセスのメモリ領域は，通信に先立ってピンダウン処理を行っておく必要がある．

また，5.3.1節で述べたように，PATLB やネットワークインタフェース上のページテーブルのエントリは，ホスト上でのピンダウン・アンピンダウン処理と連動して更新されることを保証する必要がある．特に，ホスト上でアンピンダウン処理を行って物理メモリへの固定が保証されなくなったエントリが，その後も PATLB やネットワークインタフェース上のページテーブルに残存していると，Martini がピンダウンされていないメモリ領域に対してアクセスできてしまうことになり，最悪の場合，別プロセスに再割当てされた物理ページに対してアクセスできてしまうなど，メモリ保護が破綻してしまう可能性がある．そのため，アンピンダウンによってその領域が物理メモリ上に存在していることが保証できなくなる前に，PATLB などのエントリを無効化することを確実に行わなければならない．

このことから，ピンダウンと Martini 側へのページテーブルエントリの登録およびアンピンダウンと Martini 側のページテーブルエントリの無効化は，それぞれひとまとめにして行うのが望ましい．

#### ピンダウン・アンピンダウン処理の実装

ピンダウン・アンピンダウン処理はカーネルの管理するページテーブルにアクセスし，ページテーブルの各エントリのフラグを直接書き換えることで実現可能であるが，Linux がこのようなフラグに対する直接操作を想定しているかどうかは不明であり，またフラグをセットしたままプロ

セスが終了してしまうとメモリが正常に解放されずカーネルで不具合が生じる可能性がある．そこで，安全で確実にピンダウン処理が行えるよう，Martini 向けの通信ライブラリでのピンダウン処理には，OS の提供する mlock・munlock システムコールを利用する実装とした．

Linux では，mlock を呼び出すと，ユーザプロセスの連続した仮想ページ領域をまとめて管理するカーネル内の vm_area 構造体がピンダウンした領域とそれ以外の領域に分割され，ピンダウンした領域を管理する vm_area 構造体にスワップアウトを禁止するフラグがセットされ，ピンダウン処理が完了する．この場合，ピンダウンを解除する munlock システムコールを呼ばずにプロセスが終了しても，カーネルが自動的にメモリのアンピンダウン処理を行う．

しかしながら，mlock・munlock システムコールをそのまま用いてページのピンダウン・アンピンダウンを行う場合，ユーザの責任のもとでページテーブルへの登録・無効化をあわせて呼び出す必要が生じ，これらがひとまとめに行われることを保証できないという問題が発生する．mlock によってピンダウンした後に，ネットワークインタフェース側のページテーブルへの登録を行わなかった場合は単に Martini が DMA に失敗するだけ済むが，PATLB やネットワークインタフェースのページテーブルの無効化を行う前に munlock を呼び出すことができてしまうと，対象となる通信領域が物理メモリ上に存在することが保証できなくなってしまうため，ホストの OS の提供するメモリ保護が破綻してしまうという問題がある．また，mlock は特権が必要なシステムコールであるため，ユーザ権限で実行されている並列プロセスからそのままでは利用できないという点も問題となる．

そこで，Martini の低レベルソフトウェアライブラリではデバイスドライバ内で mlock・munlock システムコールに対するフックを用意し，デバイスドライバがロードされた際に mlock・munlock システムコールに前処理および後処理を追加することで，これらの問題を解決することにした．

図 5.1に mlock および munlock システムコールに対するフックの処理の流れを示す．

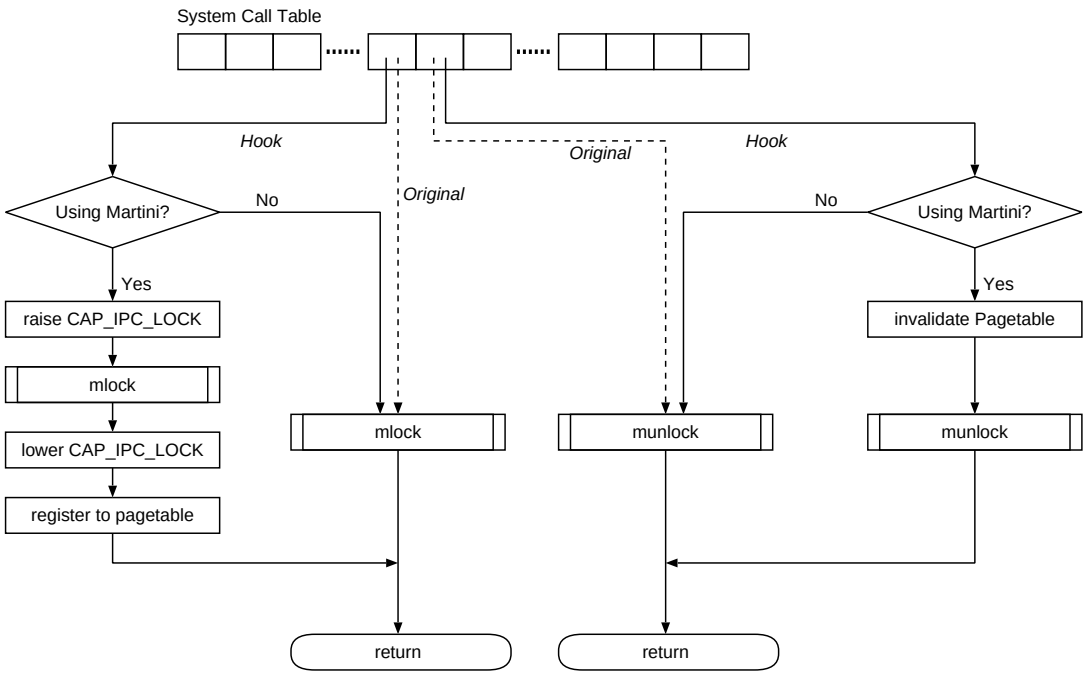



図 5.1　mlock/munlock システムコールへのフック

mlock システムコールに対するフックでは，まず mlock を呼んだプロセスが Martini を利用しているプロセスかどうかを確認し，そうであればプロセスに対して一時的に mlock を許可する権限である CAP_IPC_LOCK を付与し，オリジナルの mlock を呼び出す．プロセスが Martini を利用しているかどうかの確認は，事前に設定されているプロセスの管理構造体のフラグを参照して行う．オリジナルの mlock から戻ったら，プロセスの CAP_IPC_LOCK を解除し，続いてネットワークインタフェース上のページテーブルへの登録を行い，処理を完了する．一方，munlock システムコールに対するフックでは，まず munlock を呼んだプロセスが Martini を利用しているプロセスであった場合，先に PATLB とネットワークインタフェース上のページテーブルの無効化処理を行った上で，元の munlock を呼び出す．mlock・munlock を呼び出したプロセスが Martini を利用していないプロセスであった場合，どちらの場合もそのまま元の mlock・munlock を呼び出して処理を完了する．

この実装では，プロセス終了時に mlock したままの領域が残存していた場合，メモリ領域のアンピンダウン自体は行われるが，PATLB やネットワークインタフェース側のページテーブルに不正な値が残ってしまうという問題がある．この問題は，プロセス終了時にデバイスファイルが close された際にページテーブルの無効化処理を行うことで回避することにした．

以上の方法で，ページテーブルと確実に連動し，メモリ保護を破綻させないピンダウン・アンピンダウン処理を実現した．なお，この実装では，ピンダウン処理に Martini とのデータのやりとりが含まれるため処理に時間を要してしまうが，これは通信に用いる領域を事前にまとめて確保したり，あらかじめピンダウンした領域を，通信終了後にすぐにアンピンダウンせずに再利用するなどの工夫をアプリケーション側で行うことで隠蔽できる [104].

### 5.3.3 contflag 領域

4.6.2節において述べたように，Martini は，Window を通じて発行された処理が完了すると，CONTTBL が指すホストメモリ上の領域に対して DMA で Window が再利用可能となったことを示す contflag を書き込む．その際，CONTTBL のエントリは物理アドレスとして Martini に処理される．

contflag が書き込まれる領域 (contflag 領域) は，ユーザプロセスが Window の空きを確認するために頻繁にアクセスすることになるため，PUSH や PULL で用いる通信領域などと同様にユーザプロセスから直接ポーリングできることが望ましい．そこで，contflag 領域をデバイスドライバ内のメモリ上に確保し，Window と同様にユーザプロセスに対してマップすることにした．

contflag 領域は各 Window に一対一対応する領域であることから，デバイスドライバ内に Window の個数分のページを確保し，各ページの先頭部分の物理アドレスを contflag 領域として CONTTBL に登録しておく．contflag 領域自体は Window に付随するリソースであることから，ユーザからデバイスドライバに対して Window のマップ要求があった際に，Window に加えて contflag 領域をユーザプロセスにマップする．

### 5.3.4 プロセス登録機能

4.2.4節で述べたように，Martini の提供するメモリ保護機構は，PGIDTBL に登録された PID と PGID が正しい値であることを前提とした設計となっている．そこで，この PGIDTBL への PID と PGID の登録を，デバイスドライバを通じてのみ可能なように制限し，さらにこのデバイスドライバ内の PID と PGID の登録処理を呼び出すことが可能なプロセスを，並列プロセスの実行を管

理する特別なプロセス (管理プロセス) に限定することで，PID と PGID の値の正当性の保証を実現した．

まず，OS 上の Martini のデバイスファイルをマイナー番号を変えるなどして複数設け，デバイスドライバ内では複数のデバイスファイルのうち，特定のものからのみ `ioctl` 経由で PID と PGID の登録処理が呼び出せるように制限する．その上で，OS 側で PID と PGID の登録処理の呼び出しが可能なデバイスファイルに対して一般ユーザ権限のプロセスがアクセスできないように，特定のユーザ所有のものとし，パーミッションの制限を行う．これらにより，一般ユーザ権限で実行される並列プロセスが PGID や PID を書き換えてしまうことを防ぐ．

並列プロセスにどの Window を割当てるかは，分散 OS などの管理プロセスが決定することを想定している．管理プロセスは並列プロセス実行要求を受け付けると，`fork` などを行い，子プロセスを生成する．子プロセスは PGIDTBL への登録を行うことができるデバイスを `open` し，並列プロセスに割当てる Window の ID と自らのプロセス ID [(注 1)]をデバイスドライバに登録する．その際，デバイスドライバ内で，子プロセスのプロセス管理構造体に，そのプロセスが並列プロセスであることを示すフラグと，そのプロセスが使用する Window の ID をセットしておく．登録後，子プロセスは `exec` で並列プロセスを実行する．並列プロセスが Window 獲得のためにデバイスを `open` して `mmap` システムコールを呼ぶと，デバイスドライバ内で，そのプロセスの管理構造体を確認し，Martini を利用した並列プロセスであるかどうかの認証を行う．デバイスドライバは，認証に成功した場合に限り，その並列プロセスに対してあらかじめ管理プロセスが指定した Window をマップする．この処理により，管理プロセスによって許可されたプロセスのみが Window を利用できる環境を実現する．

### 5.3.5 SID テーブル

SID から仮想アドレスを得るのに用いる TLB である RVATLB に関しても，ページテーブルと同様に，ミスヒットに対応すべくネットワークインタフェース上にすべての SID と仮想アドレスの対応を持ったテーブル (SID テーブル) を設ける．

PATLB と異なり，RVATLB が不正な値や無効な値を返したとしても，そこから得た仮想アドレスが PATLB でアドレス変換できなければ DMA 転送が行われることはないことから，SID テーブルの内容はユーザの責任で管理すれば十分である．このことから，SID テーブルのエントリについては，デバイスドライバを通じてピンダウンの有無を厳格に確認するなどの処理は行わない実装とした．

## 5.4 例外処理

Martini では，アクセスレジスタを介して，ホスト CPU からもオンチッププロセッサと同様に各モジュールのレジスタにアクセス可能な設計となっている．そのため，例外処理をはじめとするファームウェアで実装可能な処理は，PCI バスコントローラのリセットなどの一部の処理を除いてホスト上のデバイスドライバを用いても実現することができる．

TLB のミスヒットやプロテクション違反などは基本的に発生頻度が低いと考えられることから，デバイスドライバとファームウェアのどちらで実装しても全体の通信性能への影響は少ないものと考えられるが，ホスト CPU のオーバヘッドの緩和などを考え，例外処理はファームウェア上で

[(注 1)]Martini が扱う PID ではなく，OS が管理するプロセス ID.

実装することにした．

Martini の各ハードウェアモジュールは，例外が発生すると割込み線をアサートする．割込み線は割込みコントローラを介してオンチッププロセッサに伝わる．Martini のファームウェアでは，割込み処理前後のレジスタ退避および復帰処理にそれぞれ数十サイクルを要する．発生頻度の低い例外処理のみが目的であれば，この遅延は大きな問題ではないが，Window を介したハードウェア非対応の通信処理要求の発行やネットワークからのハードウェア非対応のパケットの到着など，ソフトウェア提供の通信プリミティブを実装する場合，このレジスタ操作の遅延は通信性能に影響することになるため，わずかでも削減できるのが望ましい．そこで，これらの割込みを発生する可能性のある Initiator Controller と Remote Controller に関しては，割込みコントローラで割込み信号をマスクし，割込みコントローラのレジスタを常時ポーリングして例外発生を検出することで応答性を向上させる実装とした．

## 5.5 ソフトウェア実装の通信プリミティブ

低レベルソフトウェアライブラリでは，Martini がハードウェアで提供する PUSH・PULL 以外のプリミティブとして以下に挙げるものを実装した．

- メッセージ通信 (SEND/RECV)
- 排他制御 (LOCK/UNLOCK)
- バリア同期 (BARRIER)

以下ではこれらの実装について述べる．

### 5.5.1 メッセージ通信 SEND/RECV

SEND/RECV は送信側と受信側の双方で明示的に送受信関数を呼び出すことでデータのやりとりを行うメッセージ通信型の通信プリミティブである．SEND により特定プロセスに対するメッセージの送信を行い，RECV で特定プロセスからのメッセージの受信を行う．

メッセージ通信型の通信モデルでは，実際のデータの転送が抽象化されているため，ユーザはネットワークの構成を気にせずプログラミングを行うことが可能である．PC クラスタ向けの軽量通信ライブラリではメッセージ通信型の通信モデルを提供しているものが多く，既存のライブラリやアプリケーションの移植を考えた場合，SEND/RECV に対する上位レイヤからの需要は大きいと考えられる．そこでメッセージ通信型の通信機能をソフトウェアで実装することとした．SEND/RECV は，上位に MPI などの通信ライブラリを実装した場合，頻繁に呼び出されるプリミティブとなることから，低オーバヘッドで高い通信性能が得られるよう，ハードウェア実装されている PUSH・PULL を組み合わせ，極力これらに近い性能が得られるようユーザライブラリ上で実装した．また，リソースの制限などを考慮し，実際のデータ転送に PUSH を用いたものと PULL を用いたものの 2 方式を提案・実装した．

### データ転送に PUSH を用いた SEND/RECV の実装

PUSH を用いた実装では，SEND 要求側が PUSH で RECV 要求側にデータを転送し，その上で別途 PUSH を用いてデータを送信したことの通知を行う．RECV 要求側には，データ本体を受信するメッセージバッファ(Message Buffer) と，データの Valid や長さなどの個々のメッセージの情報を保持するディスクリプタテーブル (Descriptor Table) を設ける．メッセージバッファは一定サイズのブロックに分割して管理し，バッファ管理テーブルやディスクリプタテーブルは分割後した数と同数のエントリを備える．また，SEND 要求側にはリモートのメッセージバッファの使用状況を管理するバッファ管理テーブル (Buffer Management Table) を設ける．これらバッファやテーブルのエントリは，それぞれ通信する可能性がある全プロセス分用意する．

図 5.2に，PUSH を用いた SNED/RECV の実装において，プロセス 0 がプロセス 1 にメッセージを SEND し，プロセス 1 がそれを RECV で受信するまでの流れを示す．

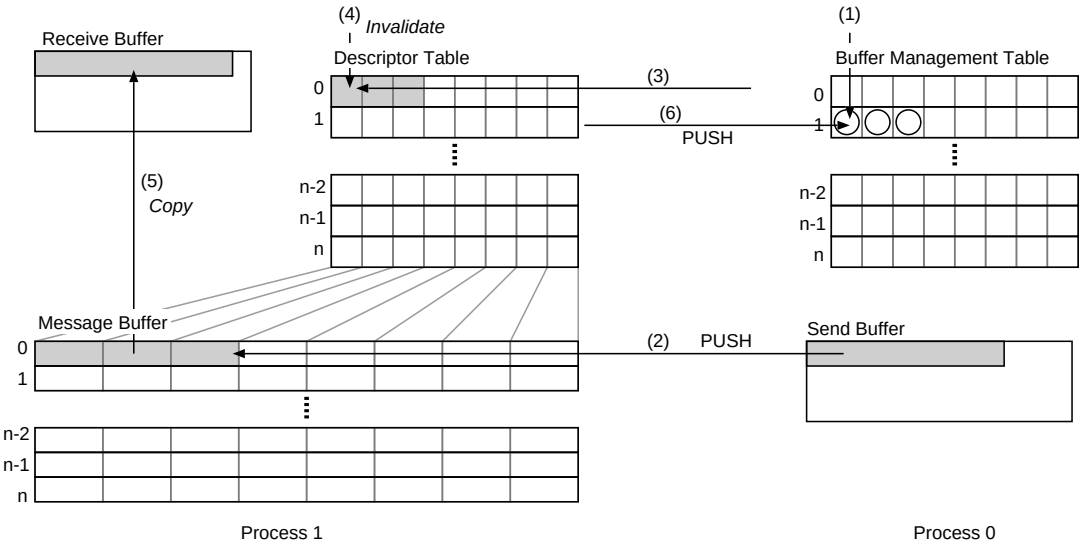



図 5.2　PUSH を用いて実装した SEND/RECV

まず，SEND 要求側は，SEND が呼ばれるとバッファ管理テーブルを参照して RECV 要求側のプロセスのメッセージバッファの使用状況を確認する．RECV 要求側のメッセージバッファに空きが存在する場合，バッファ管理テーブルにバッファ使用中を示すフラグを書き (1)，PUSH を用いて送信データが置かれている領域 (図中の Send Buffer) から RECV 要求側のメッセージバッファの空き領域へ直接データを書き込む (2)．その後，続けて送信したメッセージに関する情報をディスクリプタテーブルに対して PUSH を用いて書き込む (3)．ディスクリプタテーブルは，メッセージバッファの分割数と同じ数のブロックに分割されており，送信したデータ本体の先頭が書かれたメッセージバッファのブロックに対応するエントリにのみ，メッセージ情報が書き込まれる．

RECV 要求側は，RECV 関数が呼ばれると，送信元のプロセスに対応するディスクリプタテーブルをポーリングし，メッセージの到着を確認する．ディスクリプタテーブルに有効なメッセージ情報が書き込まれていることを確認し，メッセージの情報を読み取ったら，メッセージバッファから受信メッセージをコピーするなどの処理を行い (4)，メッセージバッファの受信メッセージが格納されていた領域に対応するディスクリプタテーブルのエントリを無効化する (5)．最後に，SEND 要求側のバッファ管理テーブル内のバッファの有効性を示すフラグを，PUSH を用いて消去する

(6).

#### データ転送に PULL を用いた SEND/RECV の実装

データ転送に PULL を用いた実装では，SEND 要求側は送信データを用意して RECV 要求側にデータの所在のみを PUSH で通知し，RECV 要求側は送信データを PULL で SEND 要求側から読み出すことで取り込む．そのため，RECV 要求側に SEND 要求側で用意した送信データの所在に関する情報を受信するためのディスクリプタテーブル (Descriptor Table) と，PULL でデータを取り込むのに用いる受信バッファ(Receive Buffer) を設ける．また，SEND 側には，RECV 側の受信キューの空き情報を管理する受信管理テーブル (Receive Management Table) と，PULL が完了するまで送信データを保持するのに用いる送信バッファ(Send Buffer) を設ける．

図 5.3に PULL を用いて実装した SEND/RECV において，プロセス 0 がプロセス 1 にメッセージを SEND し，プロセス 1 がそれを RECV で受信するまでの流れを示す．

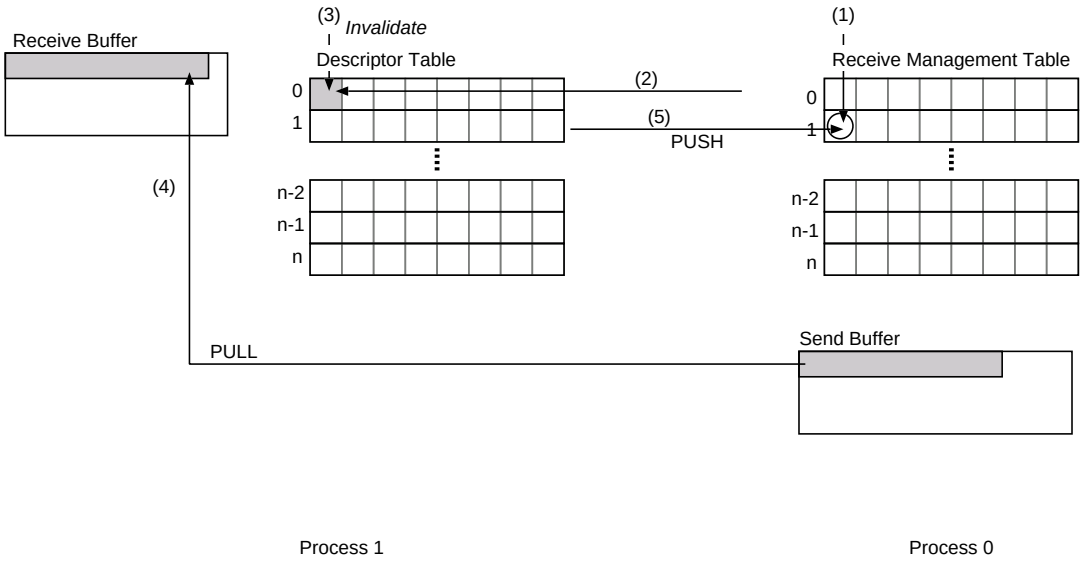



図 5.3　PULL を用いて実装した SEND/RECV

SEND 要求側は受信管理テーブルを通じてリモートのディスクリプタテーブルに空きがあることを確認したら，受信管理テーブルの空き領域にディスクリプタテーブルのエントリが使用中であることを示すフラグを書き込み (1)，フラグを書き込んだ部分に対応する RECV 側のディスクリプタテーブルに対して送信バッファに置いたメッセージの情報を PUSH で書き込む (2).

RECV 要求側は，RECV 関数が呼ばれると，メッセージの到着を確認するためにディスクリプタテーブルをポーリングする．メッセージの到着が確認できたら，ディスクリプタテーブルの中身を無効化し (3)，ディスクリプタテーブルに書かれている SEND 要求側の送信バッファから，ローカルの受信バッファに PULL でデータを読み出す (4)．その後，SEND 要求側の受信管理テーブル内のフラグを PUSH でクリアする (5)．SEND 要求側は，受信管理テーブルのエントリを確認して送信バッファが再利用可能かどうかを判断する．

**2 種類の SEND/RECV の実装の比較**

PUSH を用いた実装では，メッセージバッファを通信相手となるプロセス数分用意しなければならないため，メモリの消費量が大きくなりやすいという問題点がある．これに対し，PULL を用いた実装では，受信バッファを相手プロセス数分用意する必要がなく，リソースの消費を抑えられるが，一方で PUSH による確認が行われた上で PULL によるデータ転送が行われるため，サイズの小さいメッセージの場合，レイテンシの影響が大きくなると考えられる．

そこで，PUSH を用いた実装は中程度のサイズのメッセージ用に，PULL を用いた実装はサイズの大きなメッセージ用に，それぞれを使い分けることで，双方の問題を解決する．また，PUSH を用いた実装においても，サイズの小さいメッセージについては，メッセージを直接受信キューに書き込む実装とすることでより低レイテンシな通信を実現する．

### 5.5.2 排他制御 LOCK/UNLOCK

LOCK/UNLOCK は，同一並列ジョブに属する並列プロセス間で排他制御を行うためのロック機構を提供する通信プリミティブである．LOCK を呼び出してロックを獲得することで安全にクリティカルセクションを実行し，クリティカルセクションから抜ける際に UNLOCK でロックを解放する．

プロセス間のロック機構をホスト上のユーザライブラリで実装する場合，並列プロセスグループ内で管理プロセスを専用に設けて常にリモートプロセスからのロック要求をポーリングして待機するか，割込みなどを利用してロック要求を受け付ける必要があるが，これらはいずれもコストやオーバヘッドが大きく非効率的である．そこで，LOCK/UNLOCK は処理の大部分をファームウェア上に実装することとした．

低レベルソフトウェアライブラリの提供する LOCK/UNLOCK は，キューベースド・スピンロックとする．キューベースド・スピンロックは，ロックを要求したプロセスで待ち行列を形成し，先頭から順にロックが与えられる方式のロック機構である．代表プロセスによって最後に誰がロックを要求したのかを管理し，新たなロックの要求を代表プロセスを介して最後にロックを要求したプロセスに伝えることでこれを実現する．ロックを要求してからロックを獲得するまでの間は処理がブロックするが，ロック要求からロック獲得まで高々3 個のメッセージが行き交うだけで済むためネットワークの混雑を回避でき，また確実に要求した順にロックに成功することから飢餓状態に陥ることがないという特徴がある．

図 5.4にキューベースド・スピンロックの例を示す．

図 5.4では，プロセス M がロックの管理を行う代表プロセスであり，プロセス S1 と S2 がその他のロックを要求する可能性のあるプロセスである．S2 が直前にロックを要求して獲得し，S1 はこれからロックを要求しようとしている．

まず，ロックを要求する場合，代表プロセスに対してロック要求 (Lock Request) を発行する．図では S1 がロックを要求し，M に対してロック要求を発行している (1).

代表プロセスは，ロック要求を受け付けると，最後にロックを要求したプロセスに対してロック予約 (Lock Reserve) を発行し，自身で管理している最後にロックを要求したプロセスに関する情報を更新する．図ではプロセス M が，最後にロックを要求したプロセスである S2 に対してロック予約を発行し (2)，その後最後にロックを要求したプロセスを S2 から S1 に更新している (3).

ロック予約を受け取ったプロセスは，ロックを要求中であるかロックを獲得した状態である場合，ロック予約に書かれた予約情報を保持し，ロックが解放された時点で予約プロセスに対して

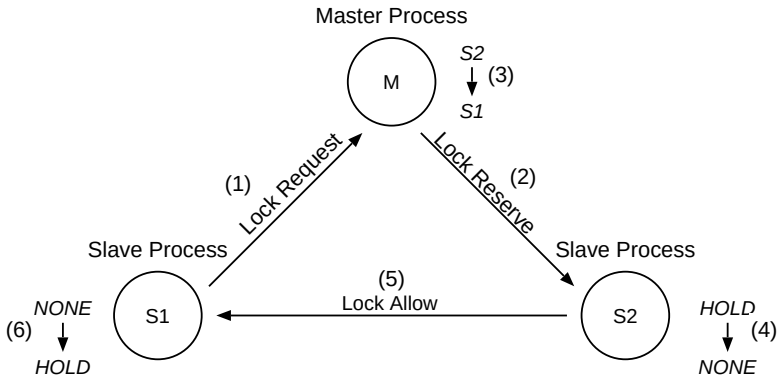



図 5.4　キューベースド・スピンロックの例

ロック成功 (Lock Allow) を通知する．また，ロック予約を受け取ったプロセスが既にロックを解放した後であった場合，そのプロセスは即座に予約プロセスに対してロック成功を通知する．図の例では，S2 はロックを解放する前にロック予約情報を受け取るため，S2 は S1 に関するロック予約の情報を保持し，ロック解除後に S1 に対してロック成功を通知している (4)(5)(6).

**LOCK/UNLOCK の実装**

Martini を用いた LOCK/UNLOCK の実装では，各プロセスはロック獲得をポーリングして待つことから，ホストメモリ上の領域を用いてロック獲得通知を行うのが望ましい．そこで，ロックの獲得状態を示す領域を，ホストメモリ上に確保し，事前にピンダウンして利用することとした．これにより，ロック獲得の通知を push パケットで行えるようになる．

一方，それ以外のロック処理に必要となる情報 (最後にロックを要求したプロセスやロックを予約しているプロセスに関する情報など) は，ファームウェアでの処理で用いられるため，ネットワークインタフェース上の SDRAM 上に保持する．これらの値は，Window を介してファームウェアに通知することで初期化する．

ロック要求とロック予約については，受信と同時にファームウェアが処理を開始できるよう，PUSH や PULL とは異なる独自のパケットを用いる実装とした．

また，LOCK/UNLOCK では，複数のロックを同時に実現できるよう，ID でロックを使い分けられるような実装とした．並列プロセスは，各ロック変数の初期化の段階で，ID ごとにどのプロセスが代表プロセスなのかを指定する．

### 5.5.3　バリア同期 BARRIER

BARRIER は，並列ジョブを構成する任意のプロセス間で同期処理を行うための通信プリミティブである．あるプロセスが BARRIER を呼び出すと，バリア同期に参加するすべてのプロセスが BARRIER を呼び出すまで処理がブロックされる．

一般に，バリア同期の実装には，プロセス間で木構造を構築し，根に向かってバリア参加のメッセージを収集した後，葉に向かってバリア解除のメッセージを伝達するツリー方式のものと，2 プロセス間でのメッセージの交換を繰り返すことで全体の同期をとるバタフライ方式のものがある．

バタフライ方式の方が短時間でバリア同期が完了するが，RHiNET-2 では論理トポロジに偏りが発生する可能性があるため，ネットワーク構造にあわせて木構造を構築することで最適化を図りやすいツリー方式を用いることにした．

**BARRIER の実装**

バリア同期に参加するプロセスは，BARRIER を要求してから解除を待つまでの間ブロックされるのが通常であることから，プロセスがバリア参加要求発行からバリア解除までの間にポーリングを行ってもオーバヘッドとならない．そこで，バリア同期への参加や解除の通知は，ホスト上であらかじめ指定した領域へのフラグの PUSH をユーザライブラリ内で行うことで実現する実装とした．

バリアの解除の通知は，バリア参加プロセスに対して push パケットをマルチキャストすることで行う．RHiNET-2/SW はハードウェアマルチキャスト機構を備えているが，これはヘッダも含めて完全に同じパケットを複数のノードに送ることしかできないため，受信プロセスを明示的に指定しなければならない Martini の PUSH と組み合わせて利用することができず，受信処理にファームウェアによる処理が介在することになるため，頻繁に同期をとるような処理の場合待ち時間が増大してしまう．そこで BARRIER では，PUSH によるユニキャストを繰り返すことでマルチキャストを実現することにした．ユニキャストの相手や順序については，木構造となるように，事前にバリア同期を行うプロセスの集合に ID を割り当てて登録する際に静的に決定しておく．この木構造のルートとなるプロセスから子となるプロセスに向かってマルチキャストを繰り返すことで，バリア参加プロセス全体に対するマルチキャストを実現する．

一方，バリアへの参加要求は，マルチキャスト時の通知の順序を逆行して，段階的に収集すればよい．そこで，バリア要求の通知は，プロセスによる木構造での子から親へのフラグの PUSH として実装した．

BARRIER が発行されると，各プロセスはバリアの木構造の中で自身の子となるプロセスからのバリア要求を，メモリをポーリングして待つ．子がいないか，もしくは子からのバリア要求が出揃った場合は，親プロセスの通知領域に対してフラグを PUSH する．ルートとなるプロセスがすべての子からのバリア参加通知を検出したら，今度は子に対してバリア解除を PUSH で通知する．各プロセスは，親となるプロセスからのバリア解除をポーリングして待ち，親からのバリア解除を検出したら，子に対して順に PUSH を発行し，バリア解除を通知する．各プロセスは，自身のすべての子に対してバリア解除を通知した時点で，バリア同期完了とし，BARRIER 関数から戻る．

# 第6章　Martiniの基本性能評価

Martini は，従来，シミュレーションによる限定された評価しか行われてこなかった．そのため，完全にハードウェア実装された RMA や OTF 通信など，これまでのネットワークインタフェースには見られない新しい取り組みについて，アーキテクチャは示されていながらも実際の通信における効果や問題点などについては十分に評価・検討がなされていない状況であった．本研究ではこれらについて，実機上での基本通信性能の評価を示し，その分析結果について述べ，これらの結果より現在の Martini のプロトコル処理部分のアーキテクチャが抱える問題点を明らかにした．本章ではこれらについて述べる．

## 6.1　基本性能の評価

### 6.1.1　評価環境

以下に示す評価結果は，特に記載がない限り実機を用いて測定した値である．本章の実機評価は，RHiNET-2/NI を搭載した 2 台のホスト PC を直結して行った．評価に用いたホスト PC の諸元を表 6.1に示す．この環境では，Martini は 64bit/66MHz の PCI バスを介してホスト PC と接続されていることになる．

表 6.1　ホスト PC の諸元

| | |
|---|---|
| CPU | Pentium III 933MHz × 2 |
| Memory | PC133 SDRAM 512M |
| Chip Set | ServerWorks ServerSet III LE |
| PCI | 64bit/66MHz |
| OS | RedHat Linux 7.2 with kernel 2.4.21 |

なお，本評価で用いた RHiNET-2/NI は，RHiNET-2/SW の製造上の不具合の回避を目的に，リンク速度を 3/4 の 6G+6Gbps に落としたものを使用している．RHiNET-2 では，元々64bit のフリットに対して，フリットの種別を識別するための情報を 2bit 付加し，さらに 14bit の ECC を付加して合計 80bit 幅に拡張して送出する．そのため，この場合のスループットの理論上の最大値は 6G+6Gbps の 64/80 である 4.8G+4.8Gbps となる．

### 6.1.2　レイテンシの評価

まず，通信レイテンシの評価として，リモートメモリライトおよびリモートメモリリードのレイテンシを示す．

リモートメモリライトのレイテンシは，PUSH，BOTF および AOTF を用いて Ping-Pong を行うことで Round Trip Time (RTT) を計測し，それを 2 で割ることで求めた．なお，BOTF では Window に push パケットのイメージそのものを書き込むことで push パケットを送出してリモートメモリライトを行った．また，AOTF では，あらかじめ Martini 上の Header Buffer に push パケットのヘッダを登録しておき，AOTF 用の I/O 領域に書き込みアクセスを行うことで push パケットを送出してリモートメモリライトを行った．

リモートメモリリードのレイテンシは，PULL および BOTF を用いて pull 要求パケットを発行し，読み出したデータがローカルメモリに書き込まれたことを検出するまでの時間を計測することで求めた．BOTF を用いた測定では，pull 要求パケットのイメージそのものを Window に書き込むことでリモートリードを行った．

いずれの場合も，受信領域の末尾をポーリングすることでデータ書き込み完了を検出し，測定を行った．

### リモートメモリライトにおける通信レイテンシ

リモートメモリライトのレイテンシの測定結果を図 6.1に示す．

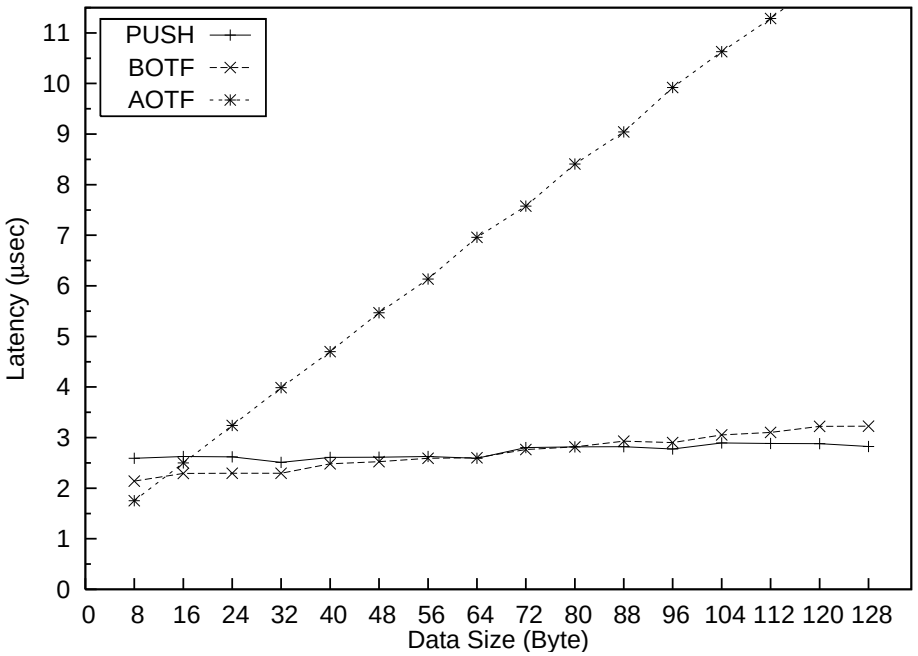



図 6.1　リモートメモリライトのレイテンシ

図の X 軸はリモートライトのサイズを，Y 軸はその際のレイテンシを示している．“PUSH” は PUSH を用いてリモートメモリライトを行った際のレイテンシを，“BOTF” と “AOTF” はそれぞれ BOTF 機構および AOTF 機構を利用して push パケットを送出した場合のレイテンシを示している．

AOTF を用いて 8byte の push パケットを送出した場合に，レイテンシは 1.75$\mu$sec と，最小値を示した．サイズが 16byte から 64byte の間では，BOTF を用いた push パケットの送出が最小レイテンシを示し，それ以上では PUSH のレイテンシが最小となった．PUSH は大きなデータを転送し

た際に高スループットを実現するよう設計されているため，パケット送出までの処理時間に DMA 転送のセットアップに要する時間が含まれる．そのため，サイズの小さいデータを送信する場合，送信側で DMA を行わない AOTF や BOTF に比べレイテンシが大きくなってしまう．

表 6.2 は PUSH，BOTF および AOTF による 8byte のリモートライトの処理時間の内訳を示している．値の一部は RTL シミュレーション [116] により導出している．

表 6.2　リモートライトの処理時間内訳 (単位: $\mu$sec)

| | PUSH | BOTF | AOTF |
|---|---|---|---|
| Martini が要求受け付け | 0.72 | 0.79 | 0.39 |
| リモートライトパケット送出開始 | 1.48 | 1.14 | 0.74 |
| リモートにパケット到着 | 1.49 | 1.15 | 0.75 |
| リモートでパケット処理開始 | 1.73 | 1.39 | 0.99 |
| PCI トランザクション開始 | 2.06 | 1.72 | 1.32 |
| リモートライト完了 | 2.48 | 2.14 | 1.74 |

表 6.2を見ると，通信要求がホスト上のプロセスで発行されてから Martini が通信要求を受け付けて処理を開始するまでの時間と，リモートでの PCI トランザクションが開始してからリモートライトが完了するまでの時間とで，レイテンシの約半分を占めていることがわかる．

AOTF は 1DW の書き込みで通信が起動されるが，評価環境では 1DW の書き込みでも，Martini が要求を受け付けるまでに 0.39$\mu$sec 要している．これは，評価環境では Martini はホストに PCI バスを介して接続されているため，連続したアドレスへの書き込みを 1 度のトランザクションにまとめるなどの目的でホストからの書き込みがチップセット内の PCI コントローラで一旦バッファされた上で Martini に送られるためだと考えられる．PCI バスを使用している以上，この分のレイテンシを縮めることは難しい．

また，受信側における，PCI トランザクション開始後のレイテンシについても，ホストメモリへの DMA 転送が必要となるため，Martini 側の工夫によりこれ以上レイテンシを縮めることは難しい．そのため，通信レイテンシを削減するには，ネットワークインタフェースコントローラの性能向上だけでなく，バスやチップセットなどの性能向上が必要となると考えられる．

### リモートメモリリードにおける通信レイテンシ

リモートメモリリードのレイテンシを図 6.2に示す．図中の “PULL” は PULL によるリモートメモリリードのレイテンシを示しており，“BOTF” は BOTF を用いて pull 要求パケットを送出して行ったリモートメモリリードのレイテンシを示している．

図より，BOTF を用いた場合の方が，PULL の発行を要求した場合に比べて 0.18$\mu$sec 短い時間で pull 要求パケットを送出できていることがわかる．PULL の発行にも，BOTF による pull 要求パケットの発行にも，Window に対して等しく 6DW データを書き込む必要がある．このことから，このレイテンシの差は，Martini 内部でのパケット構築までに要する時間の差によるものと考えられる．

表 6.3 に，PULL と BOTF で 8byte のリモートメモリリードを行った際のレイテンシの内訳を示す．PUSH の場合と同様，値の一部は RTL シミュレーションにより導出している．

PULL を用いて pull 要求パケットを発行した場合と，BOTF で pull 要求パケットを発行した場

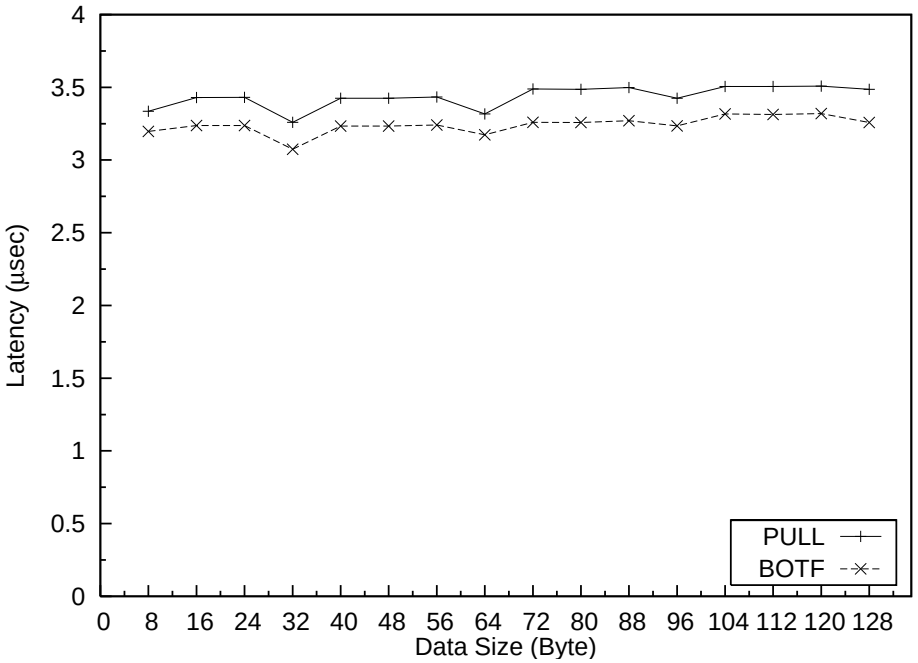



図 6.2　リモートメモリリードのレイテンシ

表 6.3　リモートリードの処理時間内訳 (単位: $\mu$sec)

| | PUSH | BOTF |
|---|---|---|
| Martini が要求受け付け | 0.72 | 0.72 |
| リモートリードパケット送出開始 | 1.25 | 1.07 |
| リモートにパケット到着 | 1.26 | 1.08 |
| リモートで PCI トランザクション開始 | 1.97 | 1.79 |
| 応答パケット送出開始 | 2.26 | 2.08 |
| ローカルにパケット到着 | 2.27 | 2.09 |
| ローカルで PCI トランザクション開始 | 2.84 | 2.66 |
| リモートリード完了 | 3.26 | 3.08 |

合とを比べると，どちらもホスト上から Martini に対して設定すべき情報量は等しいことから通信処理におけるホストのオーバヘッドは同等となる．pull 要求パケットが発行されるまでの処理時間については，BOTF の方が短時間であることから，PULL の要求時の処理のような，ユーザからの要求を受けてヘッダのみの短いパケットを送出するような処理は，ヘッダ生成に複雑な処理を必要としない限り個別にハードウェア実装せずに BOTF のような仕組みで発行するものと定めるのが性能面およびハードウェア量の面で効率的であると考えられる．

### 6.1.3 スループットの評価

次に，PUSH を用いたリモートメモリライトによるスループットを，片方向通信と双方向通信のそれぞれの場合について示す．

片方向通信におけるスループットは PUSH を連続要求して，途中一定時間内に送出できたデータサイズを元に導出している．一方，双方向通信におけるスループットは，同様に両側のノードからデータを送りあい，途中一定時間内に送出できたデータサイズを合計して求めている．

評価の結果を 図 6.3 に示す．図において，X 軸は 1 度の PUSH の要求で転送指示を出すデータサイズを示しており，Y 軸はその際のスループットを示している．

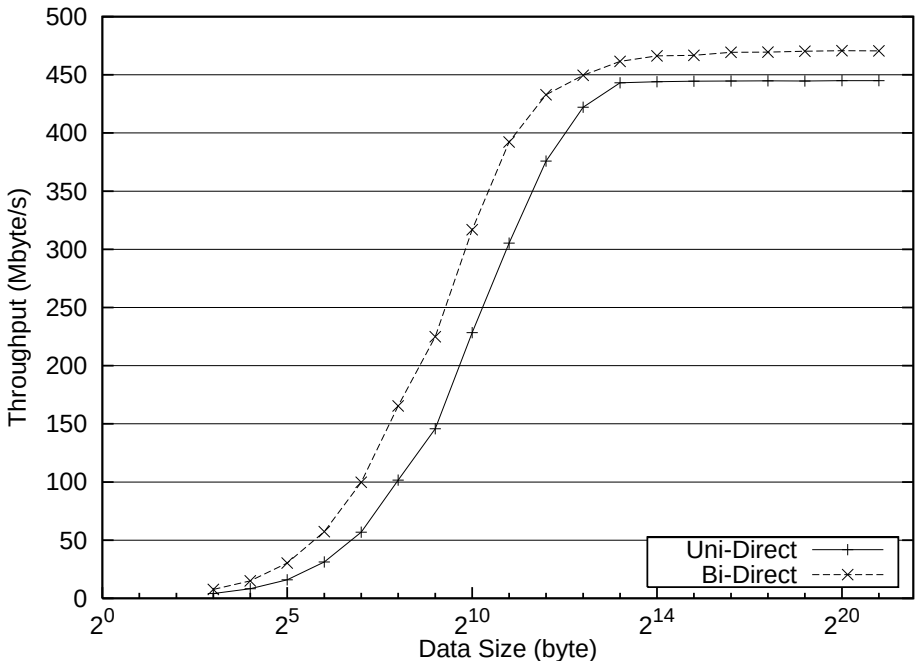



図 6.3 リモートメモリライトのスループット

片方向通信におけるスループットは最大で 444Mbyte/s，双方向通信におけるスループットは最大で 470Mbyte/s となった．評価で用いた 64bit/66MHz の PCI バスはデータ転送速度が 528Mbyte/s の半二重のバスである．評価の結果の片方向通信時のスループットは PCI バスのデータ転送速度の 84%，双方向通信時のスループットは 89%の速度となっていることから，Martini はバスを効率的に利用できていると言える．

ここで，ノード間のデータ転送のボトルネックを明らかにするために，受信側の Martini の SWI

において，受信したパケットを処理せず破棄し続けた場合のスループットの測定もあわせて行った．結果を図 6.4 に示す．

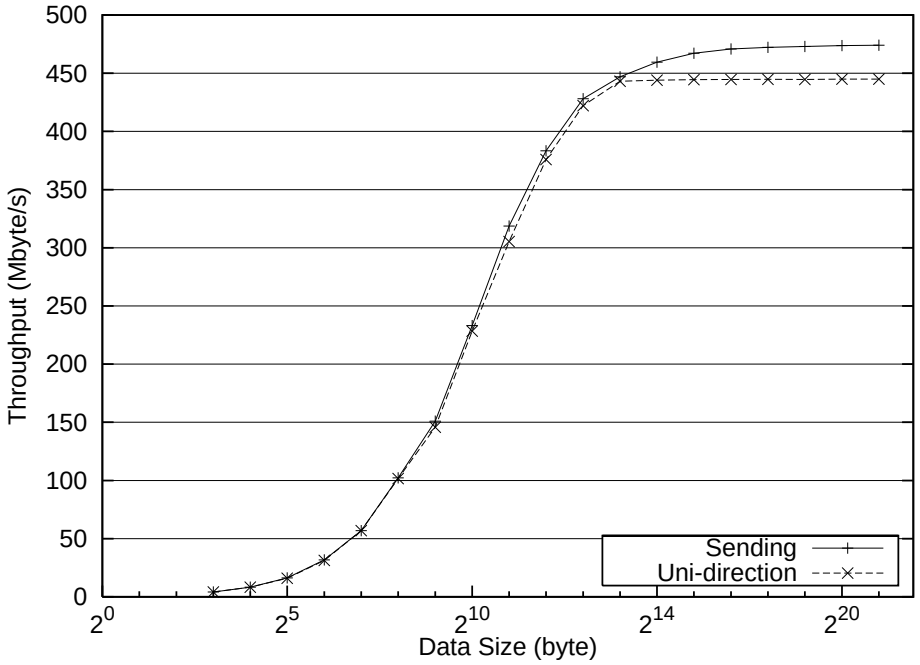



図 6.4　受信側でパケットを破棄する場合のスループット

図中の “Sending” が，受信側でパケットを破棄し続けた場合のスループットである．図より，受信側で受信パケットを捨てた場合の方が高いスループットを示していることから，通常の片方向通信の測定では，送信側の送信能力に余裕があり，受信側によって最大値が制限されていることがわかる．

これは，Martini において，受信側でパケットをパイプライン処理できないのが原因であると考えられる．パケットの受信処理を行う RCONT は RFend と RBend で構成され，RFend がパケットヘッダを解析し，RBend が DMA 転送の要求を発行する．これらは 2 モジュールに分かれているものの，SWI の受信パケットのバッファが FIFO 構造となっているため，RFend は直前に処理したパケットのペイロードが DMA コントローラによって完全に読み出されるまで次のパケットの処理を開始できず，パイプライン動作することができない．これにより，受信側での DMA 転送の発行間隔が空いてしまい，スループットが制限されてしまうことになる．

この問題を解決する方法として，SWI における受信パケットのダブルバッファリングが考えられる．到着したパケットを交互に違うバッファで受信することで，RFend と RBend で独立したバッファに対してアクセス可能となり，RBend で DMA 転送が始まる前に RFend で後続のパケットのヘッダの解析を開始することができる．

### スループットの立ち上がり

サイズの小さいデータのやりとりが多い並列分散処理環境では，ノード間通信におけるスループットのピーク値以外に，スループットのカーブの立ち上がりの早さも重視される．すなわち，立

ち上がりが早いネットワークでは，サイズの小さいデータを転送した場合においても高いスループットが得られることを意味する．これを判断する指標の 1 つに，スループットがピーク値の半分に達した時点での 1 度の通信関数呼び出しで送っているデータサイズ (以下 $Size_{half}$) がある．図 6.3より，Martini は $Size_{half}$ に 1024byte で到達していることがわかる．この値をより小さくするには，Martini における送受信処理の改善が必要であると考えられるが，図 6.4を見ると，1 度の転送サイズが 1024byte 以下の場合，受信側の SWI でのパケットの破棄の有無によるスループットの差はほとんどないことがわかる．これはメッセージサイズが小さくなると，前のパケットの受信が完了してから次のパケットが到着するまでの間隔が広がるため，前のパケットの受信処理が次のパケットが到着する前に完了してしまうことによる．このことから，仮に受信側の性能が改善しても，メッセージサイズが小さいところではスループットはほとんど向上しないものと予想される．

一方，送信側には，$Size_{half}$ をより小さくする上で改善する余地があると考えられる．送信側では，PUSH のデータサイズが RHiNET-2 の MTU(2Kbyte) より大きい場合や，転送領域がページ境界をまたぐ場合，1 回の PUSH 要求によって複数回の DMA 転送が発行される．その際，DMA 対象領域のアドレス計算と DMA 転送はパイプライン実行され，前の DMA 転送が処理されている間に，次の DMA 転送のアドレスが計算されて DMAC のキューに格納される．そのため DMA 転送が密な間隔で実行され，パケットが密な間隔で送出されることになる．

これに対し，PUSH で送信するデータサイズが MTU 以下でページ境界をまたがないような場合，1 度の PUSH 要求で DMA 転送は 1 回しか行われない．PUSH プリミティブの要求は Window を介して Martini に渡されるが，ホストプロセスが Window に書き込めるのは，直前の要求の処理が完了して Window が再度キックできるようになったことを示す contflag が検出できてからであり，contflag はホストメモリからの送信データの DMA 読み出しが完了した後に設定される．一方，評価で用いたプログラムでは単一の Window を利用して PUSH 要求を発行しているため，ホストメモリが直前に発行した PUSH 要求に対応する DMA 転送を行っている間に，次の PUSH 要求を発行することができない．このような理由から，送信するデータサイズが小さい場合にパケット送出の間隔が空いてしまっていると考えられる．

この問題は，プロセスに複数の Window を割当て，連続して PUSH を要求する際にこれら順番に利用することで改善可能であると考えられる．複数の Window を利用した場合，前に発行した PUSH 要求の処理を行っている間に後続の PUSH 要求の Window への書き込みを済ませることができるため，contflag がセットされるのを待つよりも DMA 転送の間隔を短くすることができる．

## 6.2 他のネットワークインタフェースとの性能比較

表 6.4 は，Martini と，現在多くの PC クラスタで利用されているネットワーク [88] のネットワークインタフェースコントローラ (Myrinet-2000[3]，QsNet II[60][61]，InfiniBand および GbE) を比較したものである．表中の評価結果は評価環境が不揃いであるため公平な比較にはならないが，これらは厳密な比較ではなく大まかな傾向を確認するための指針として掲載している．

### 6.2.1 スループットの比較

Martini のスループットのピーク値は他のネットワークインタフェースコントローラに比べて大幅に低い．この差は，主にホストとの接続に用いているバスの性能差によるものである．すなわ

ち，Martini は 64bit/66MHz の PCI バスにあわせて設計されているのに対し，最新のネットワークインタフェースコントローラは 64bit/133MHz の PCI-X バスや PCI Express などにあわせた設計となっているためである．

### 6.2.2 レイテンシの比較

レイテンシに関しては，PUSH を利用した場合の値を見ても，他のネットワークインタフェースコントローラと同程度の値となっていることがわかる．これは，ハードウェア実装された送受信処理ロジックや AOTF による PCI アクセスレイテンシの最小化によると考えられる．PCI-X や PCI Express を用いた場合でも，バスのアクセスレイテンシの大幅改善が難しいことから，AOTF のような PCI のトランザクションそのものを減らす通信機構は通信レイテンシを低減させる上で効果的であると考えられる．

文献 [117] によると, Elan4 によるリモートライトプロトコルのチップ内のレイテンシは 0.24$\mu$sec であるのに対し，Martini はチップ内部での処理に 0.92$\mu$sec 要している (表 6.2)．Martini のコアクロックは 66MHz でしかなく，この値は他の最新鋭のネットワークインタフェースコントローラに比べて低い値である．Elan4 に関しては動作周波数が明らかでないが，オンチッププロセッサの周波数が 200MHz であることから Martini よりもはるかに高い周波数で動作しており，動作周波数の差が，チップ内のレイテンシの差の大きな要因となっているものと予想される．

### 6.2.3 スループットの立ち上がりの比較

表 6.4における “$Size_{half}$” の行は，6.1.3節にて述べたスループットがピーク値の半分に達する際のデータ転送サイズを示している．LANai 2XP を除いて，これらの値は参考とした文献の片方向通信のスループットのグラフから読み取っている．他のネットワークインタフェースと比較すると，$Size_{half}$ に関しては，Martini は中程度の値を示しており，平均的な立ち上がりを示していることがわかる．

表 6.4　Martini と他の最新のネットワークインタフェースコントローラの比較

| | Martini<br>RHiNET-2/NI | LANai 2XP [118][119]<br>M3F2-PCIXE | Elan4 [117]<br>QM500 | MT25208 [120]<br>InfiniHost III Ex | T110 [121][77] |
|---|---|---|---|---|---|
| ベンダ | | Myricom | Quadrics | Mellanox | Chelsio |
| ネットワーク | RHiNET-2 | Myrinet-2000 | QsNet II | InfiniBand | 10GbE |
| 基本通信処理 | Hardware | Firmware | Hardware | Hardware | Hardware |
| プロトコルオフローディング | No | Yes | Yes | No | Yes |
| コアクロック | 66MHz | 333MHz | N/A | N/A | N/A |
| ホストインタフェース | PCI<br>(64bit/66MHz) | PCI-X<br>(64bit/133MHz) | PCI-X<br>(64bit/133MHz) | PCI Express<br>(8X) | PCI-X<br>(64bit/133MHz) |
| リンク性能 (実効データ転送速度) | 600Mbyte/s×2 | 250Mbyte/s×2×2 | 1.06Gbyte/s×2 | 1Gbyte/s×2 (4X) | 1.25Gbyte/s |
| パケット転送方式 | VCT | VCT | Wormhole | VCT | SAF |
| トポロジ | Any | Any | Fat Tree | Any | Any |
| ホスト CPU | Pentium III<br>933MHz×2<br>ServerSet III LE | Opteron 1.8GHz×2<br>AMD-8131<br>MX-2G 1.0 | Itanium2<br>Intel E8870 | Xeon 3.4GHz×2 | Opteron 2.2GHz<br><br>with TOE |
| 接続方式 | back-to-back | 1 Myrinet switch<br>(0.5$\mu$sec) | N/A | InfiniScale | back-to-back |
| レイテンシ | 1.74$\mu$s | 2.6$\mu$s | 1.6$\mu$s | 3.8$\mu$s | 8.9$\mu$s |
| スループット (片方向)<br>($Size_{half}$) | 444Mbyte/s<br>(1Kbyte) | 495Mbyte/s<br>(300byte) | 911Mbyte/s<br>(4K - 8Kbyte) | 972Mbyte/s<br>(1K - 2Kbyte) | 972Mbyte/s<br>(256 - 512byte) |
| スループット (双方向) | 470Mbyte/s | 912Mbyte/s | 900Mbyte/s | 1932Mbyte/s | N/A |

# 第7章　Martiniにおける乗っ取り機構の提案・実装

本章では，Martini の開発において筆者が提案･実装した協調処理機構である乗っ取り機構 [27][28] について詳細を述べる．

## 7.1　乗っ取り機構の提案の背景

近年，汎用的なプロセッサ，メモリ，専用ハードウェア，入出力インタフェースなどの，システムを構成する要素を単一チップに集約したシステム LSI の開発が盛んである．一般的なシステム LSI は，図 7.1に示すような各構成要素がバスで接続された形態を基本としており，特にオンチッププロセッサと専用ハードウェアで処理を分担する構成となっているものが多い．近年では，オンチッププロセッサと専用ハードウェアの処理の分担を，設計の段階で明確に切り分けておくことで，ハードウェアとソフトウェアの開発を同時に進める協調設計がマルチメディア向けチップなどの開発を中心に広く行われている [122][123].

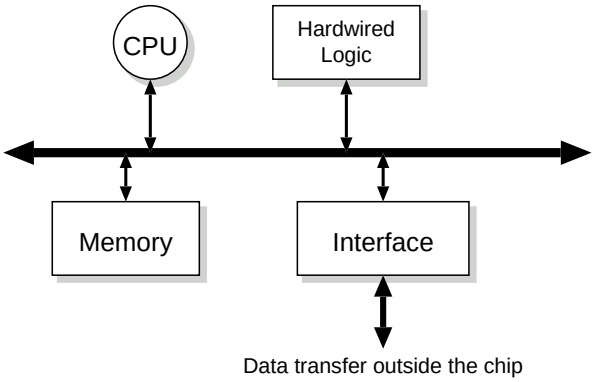



図 7.1　一般的なシステム LSI の接続モデル

Martini もまた，通信処理専用ハードウェアやメモリ，各種インタフェースに加えオンチップメモリを内蔵するシステム LSI である．しかしながら，Martini ではマルチメディア向けチップなどと異なり，最初からハードウェア単独で基本的な通信処理を提供するよう設計されており，オンチッププロセッサは常時稼働しているわけではなく，TLB のミスヒットやプロテクション違反の発生，ハードウェア実装されていない通信要求やパケットの処理の必要が生じたときのみ動作する構成をとっている．オンチッププロセッサが処理を行っている間は，ハードウェアは一切の処理をオンチッププロセッサに委ねていることになるので，停止した状態となる．すなわち，Martini では，普段はハードウェアのみが動作し，ハードウェアが処理を続行できない状況が発生した場

合に限り，代わりにオンチッププロセッサのみが動作することになる．

また，Martini のオンチッププロセッサが担当すべき処理にはハードウェアの提供する処理と類似するものが多いと考えられる．たとえば，Martini において，push パケットの受信時に RBend で PATLB のミスヒットによる例外が発生した場合，オンチッププロセッサはページテーブルを参照して PATLB のリプレースメントを行った後，DMA コントローラに対する転送要求の発行や DMA 完了後の Replier への ack パケットの発行要求などを行う必要が生じるが，リプレースメントの後の処理はハードウェアが行っている処理とまったく同じである．また，オンチッププロセッサを利用して通信処理を実装する場合でも，パケットのヘッダの解析や TLB を介したアドレスの変換などは，ハードウェア実装されている通信処理と共通の処理となるはずである．よって，オンチッププロセッサがハードウェア処理と同じ処理を行う必要がある場合に，ソフトウェアから既存のハードウェアの機能を部分的に利用することができると，ソフトウェア処理の効率を高めることができるものと考えられる．

このような背景から，本研究では，Martini の設計・実装の過程において “乗っ取り機構” と呼ばれるハードウェア・ソフトウェア協調処理の方式を新たに提案し，これを Martini のハードウェアモジュールに対して実装した．

## 7.2 乗っ取り機構

乗っ取り機構とは，システム LSI の専用ハードウェアを構成する個々のモジュールについて，そのステートや内部のレジスタの値をオンチッププロセッサが自由に変更できるように設計しておくことで，オンチッププロセッサとの間での柔軟な協調処理を実現する機構である．これを用いることで，ハードウェア処理の部分的なソフトウェア処理による置き換えやソフトウェア処理の一部でのハードウェアの使用などが実現する．

一般的なシステム LSI では，あるモジュールが例外を発生した場合，専用ハードウェア全体もしくは例外を発生したブロックが丸ごと停止した上で，オンチッププロセッサが例外の要因を取り除き，そのまま例外を発生したモジュールの代わりに残りの処理を行った上で全体のハードウェア処理を再開させる．これに対し，乗っ取り機構を利用した場合，個々のモジュールは独立にオンチッププロセッサから制御可能となるため，例外を発生したモジュールのみが停止し，そのモジュールと依存関係のない他のモジュールは例外処理中も並列に動作し続けることができる．加えて，オンチッププロセッサがモジュールのステートを自由に操作可能となるため，例外の原因を取り除いた後，例外を発生したモジュールを例外発生の直前の状態に強制的に遷移させるなどしてハードウェア処理を再開させることで，残りの処理のソフトウェアによる代行が不要となる．

また，乗っ取り機構を利用することで，モジュールの特定のステート間の処理をソフトウェア処理に置き換えることができる．ハードウェアの提供する基本通信処理と類似した通信処理をソフトウェアで実装する場合，このような部分的な処理の置き換えを行うことで，ソフトウェアによる処理を最小限に抑え，ハードウェアを活用した処理が可能となる．

このように，乗っ取り機構を用いることで，ソフトウェア単体での処理に比べ，処理の効率化を図ることができる．

## 7.3 ハードウェアモジュールへの乗っ取り機構の実装

既存のハードウェアモジュールに乗っ取り機構を実装するには，ハードウェアモジュールに以下の 3 項目を実装する必要がある．

- 停止状態
- 停止状態への移行手段
- 停止状態下での制御機構

### 7.3.1 停止状態

停止状態とは，モジュールがステートの遷移を停止して内部レジスタなどのリソースをオンチッププロセッサに解放している状態を指す．乗っ取り機構は，モジュールが例外発生やオンチッププロセッサからの要求などによって停止状態となることで成り立つ．

乗っ取りの対象となるモジュールにおいて，例外などの発生により処理が停止する可能性のあるステートが複数ある場合，個々のステートに停止状態になるような機構を設けるよりも，自発的に他のステートに遷移せず，一切のリソースを要求しないステートを新たに設け，例外発生やオンチッププロセッサからの要求によって乗っ取られた状態となる際に，自発的にこのステートへ遷移する設計とした方が実装が容易である．以下では，このような停止状態のために設けたステートを “サスペンデッドステート” と呼ぶ (図 7.2)．あるステートからサスペンデッドステートに遷移する際に，リソースを解放する機構を加えることで，停止状態を実現できる．

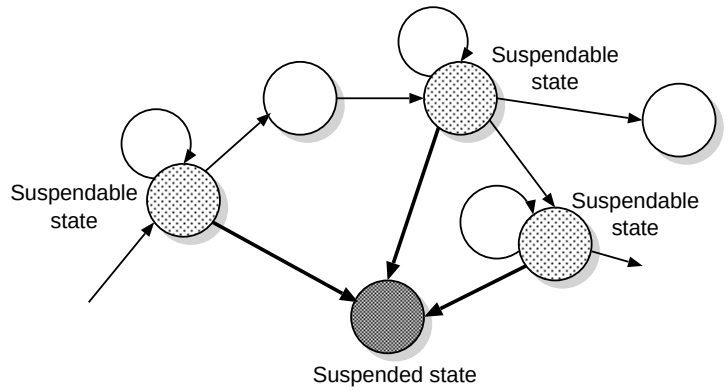



図 7.2　サスペンデッドステート

また，乗っ取り機構では，ソフトウェア処理が完了した後，モジュールがサスペンデッドステートから元のステートに安全に復帰し，ハードウェア処理を再開できる必要がある．これを実現するには，停止状態になったモジュールに入力を与えたり，そのモジュールの出力を利用するような依存のある他のモジュールを必要に応じて待機させたりするような構造にしなければならない．

モジュールがサスペンデッドステートへ遷移し，その後オンチッププロセッサによってステートを戻された後も，正常にハードウェア処理を再開できるようなステートを，以下では “サスペンダブル” なステートと呼ぶ．図 7.2の “Suspendable State” はサスペンダブルなステートである．

乗っ取り対象となるモジュールのステート $S$ がサスペンダブルであるには，以下の条件を満たす必要がある．

- 乗っ取り対象モジュールと依存のある周辺モジュールが，有限ステート遷移した後に遷移を停止し，ステート $S$ の次のステートにおいて発生する信号を待ち続ける．
- 依存のある周辺モジュールのステートをオンチッププロセッサが認識できるか，乗っ取り対象のモジュールがステート $S$ に滞在している間の周辺モジュールのステートが一意に定まる．

以下ではサスペンダブルなステートについて例を示しつつ述べる．

まず，図 7.3に示すような，乗っ取り対象となるモジュール Y が，周辺モジュール X と依存関係にある場合を考える．それぞれのモジュールは，順序機械 MX(IX, OX, SX, $\sigma_x$, $\lambda_x$) および MY(IY, OY, SY, $\sigma_y$, $\lambda_y$) で表される．ここで I は入力集合，O は出力集合，S は状態集合，$\sigma$ は状態遷移関数，$\lambda$ は出力関数である．

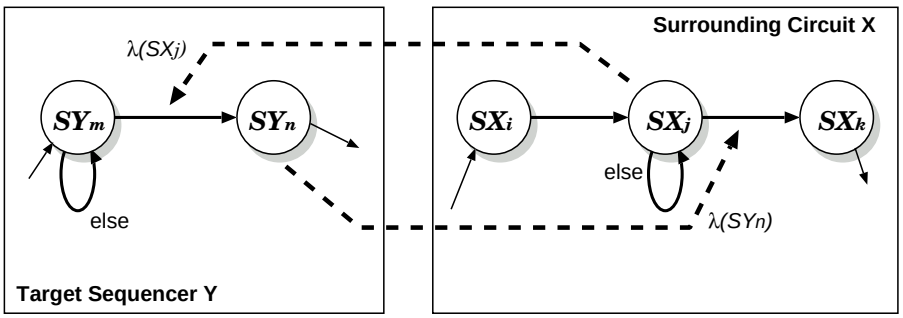



図 7.3　サスペンダブルなステートの例

X は，状態 $SX_i$ から $SX_j$ に遷移する際に，Y にハンドシェイク要求出力信号 $\lambda(SX_j)$ を送信し，その後 Y が $SY_n$ に遷移することで出されるアクノリッジ信号 $\lambda(SY_n)$ を受け取らない限り，次の状態 $SX_k$ に遷移しないものとする．このような場合，Y が $SY_m$ からサスペンデッドステートに遷移しても，X は $SX_j$ から先のステートへ進めずに待機することになる．このため，Y がサスペンデッドステートからの復帰時に $SY_m$ から $SY_n$ に遷移することで，X は正常にハードウェア処理を再開することができる．よって，Y において $SY_m$ はサスペンダブルなステートであると言える．

すなわち，以下の条件を満たす場合に，状態 $SY_m$ はサスペンダブルであると言える．

**周辺モジュール X について：**

$$\sigma(SX_j, \lambda(SY_n)) = SX_k$$

$$\sigma(SX_j, \overline{\lambda(SY_n)}) = SX_j$$

**乗っ取り対象モジュール Y について：**

$$\sigma(SY_m, \lambda(SX_j)) = SY_n$$

$$\sigma(SY_m, \overline{\lambda(SX_j)}) = SY_m$$

ただし，ここで $\overline{O}$ は $O$ の補集合を表すものとする．

一方，Y が $SY_n$ に遷移したかどうかにかかわらず X のステートが $\lambda(SX_j)$ を出力した後に $SX_j$ から別の状態に遷移してしまう場合，Y が $SY_m$ から サスペンデッドステートに移行している間に X の処理が進んでしまう．従って，復帰後に Y が正常なハードウェア処理を再開できない可能性が生じてしまう．

このような状況でも，以下の条件を満たす場合，Y は安全に処理を再開できる．

- X は $SX_j$ から数ステート先の特定のステート $SX_p$ で待機し，Y が $SY_n$ に遷移したことを示す信号 $\lambda(SY_n)$ の入力がない限りその先のステートへは遷移しない
- オンチッププロセッサが， X の現在のステートを知ることができる

この例の場合，Y が $SY_m$ からサスペンデッドステートへ遷移した後，オンチッププロセッサは X が $SX_p$ まで遷移したことを確認した上で Y を サスペンデッドステートから $SY_n$ に遷移させることができるため，安全に処理を再開できることになる．すなわち，$SY_m$ はサスペンダブルなステートとなる．

この場合の $SY_m$ がサスペンダブルとなる条件は以下の通りである．

**周辺モジュール X について：**

$$\sigma(SX_p, \lambda(SY_n)) = SX_q$$

$$\sigma(SX_p, \overline{\lambda(SY_n)}) = SX_p$$

例として，図 7.4に示すような状況を考える．図 7.4では X はハンドシェイク要求信号 $\lambda(SX_j)$ を出した後，Y からのアクノリッジ信号を待たずに $SX_j$ から先のステートに進んでしまう．ただし，X は $SX_p$ で Y からのアクノリッジ信号 $\lambda(SY_n)$ を待つものとする．

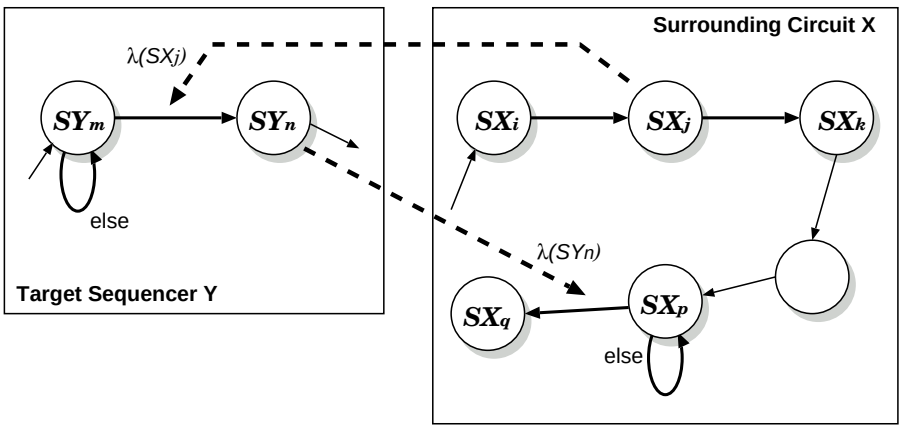



図 7.4　条件つきでサスペンダブルとなるステートの例

この場合，オンチッププロセッサは，Y が $SY_m$ からサスペンデッドステートに遷移した後，復帰する際に， X が $SX_p$ まで到達しているのを確認した上で，Y のステートを $SY_m$ にせずに $SY_n$ に設定する．このようにすることで，X，Y 共にハードウェア処理を正常に再開することが可能となり，$SY_m$ はサスペンダブルなステートとなる．

### 7.3.2 停止状態への移行手段

乗っ取り機構では，モジュールが停止状態となる要因として，以下の 2 つを想定している．

- モジュール自身による停止
- オンチッププロセッサからの要求によるモジュールの停止

前者には，TLB のミスヒットなどでモジュールがハードウェア処理を続行できない状態に陥った際に，ソフトウェアに処理の続きを依頼するケースが該当する．この場合，ハードウェアモジュールが例外などを検出した段階で，リソースを解放し，サスペンデッドステートへ遷移する設計とすればよい．

一方，後者には，ソフトウェア処理でハードウェアモジュールの機能を部分的に利用するケースなどが該当する．この場合，ソフトウェア処理によってモジュールのステートをサスペンデッドステート に変更すればよいが，モジュールのステートをソフトウェア側が任意のタイミングで強制的にサスペンデッドステートに変更してしまうと，そのモジュールのすべてのステートがサスペンダブルでない場合，非サスペンダブルなステートから直接停止状態となり，後にハードウェア処理を正常に再開できなくなる可能性が生じる．この問題は，モジュール側がソフトウェアからの要求に応じて，サスペンダブルなステートに達したことを確認してからサスペンデッドステートに遷移するような設計とすれば回避することができる．

### 7.3.3 停止状態下での制御機構

乗っ取り機構では，オンチッププロセッサは，各モジュールが停止状態になったことを確認した上で制御下に置き，代わりの処理を行う．これには以下の仕組みが必要となる．

- モジュールが停止状態になったことをオンチッププロセッサに通知するための仕組み
- 停止状態になったモジュールのステートマシンやレジスタにオンチッププロセッサがアクセスするための仕組み
- オンチッププロセッサが停止状態のきっかけを判別するための仕組み

停止状態の通知には，一般的に専用ハードウェアからの例外発生の通知と同様にオンチッププロセッサに対する割込みを用いる方法が考えられる．

また，停止状態になった各モジュールのステートマシンやレジスタへアクセスするための仕組みの実現には，オンチッププロセッサから乗っ取り対象となるすべてのモジュールに対して制御用のバスを配線すればよい．

停止状態への遷移のきっかけは，停止の原因を示すレジスタをモジュール内に新たに設けることでオンチッププロセッサが認識可能となる．あるいは，停止状態となる直前のステートがわかればハードウェアモジュールが処理を続行できなくなった理由が一意に定まるのであれば，直前に滞在していたステートを記録するレジスタを新たに設けるだけでもよい．

## 7.4 Martini への乗っ取り機構の実装

Martini では内部の通信処理を行う送信部と受信部が独立した構造となっており，いずれも細かなモジュールで構成されている．そのため，個々のモジュールは独立して動作可能であり，乗っ取り機構に対応させやすい．

Martini を構成する各モジュールのうち，HCP を構成する各モジュールは，TLB のミスヒットやハードウェア処理できないパケットの受信などの要因で例外を発生する可能性がある．また，HCP の ICONT および RCONT が提供する機能は，ソフトウェアによる通信処理の実装において利用可能なものが多い．そこで，Martini では，ICONT 内の Initiator と Replier および RCONT 内の RFend と RBend の 4 モジュールについて，モジュールごとに独立した割込み信号を設け，プロセッサのバスと接続し，さらに直前のステートを保持するレジスタを新規に追加することで，乗っ取り機構を適用した．なお，HCP にはこれ以外に DMAC や PATLB などが含まれ，また，ICONT や RCONT にはその他のモジュールも含まれるが，その多くは例外の発生を伴わないものであったり，伴ったとしても発生する例外の処理が要因を除去するだけで完了するものであったりするなど理由から，乗っ取り機構の実装を行っていない．また，HCP 外部の通信処理を行うモジュールとして，AOTF Send Controller があるが，このモジュールは HCP とは独立して開発されたため，乗っ取り機構には対応していない．

### 7.4.1 例外処理

先に述べた ICONT 内の Initiator と Replier および RCONT 内の RFend と RBend の 4 つのモジュールの発生する可能性のある例外を表 7.1に示す．

表 7.1 例外の種類と発生モジュール

| 例外の種類 | 発生モジュール |
|---|---|
| 特殊通信要求発生 | Initiator |
| プロテクション違反 | Initiator |
| PATLB ミスヒット | Initiator<br>Replier<br>RBend |
| 特殊パケット受信 | RFend |
| RVATLB ミスヒット | RFend |
| タイムアウト | 全モジュール |

各モジュールにおいて，例外を発生する可能性のあるステートはすべてサスペンダブルなステートとなるよう設計した．

また，これらのモジュールに，ソフトウェア側からサスペンデッドステートへの遷移を要求するための手段として，あらかじめ指定したステートへ遷移したらモジュール自らがサスペンデッドステートへ遷移するブレークポイント機構を設けた．ブレークポイント機構をとることで，任意のサスペンダブルなステートでモジュールを停止状態とすることができるため，単にサスペンデッドステートへの遷移を要求するだけの場合に比べ，ソフトウェアからより柔軟に利用することが可能となる．なお，本来であればブレークポイントはサスペンダブルなステートにのみ設定

可能とすべきものであるが，Martini では実験的な目的ですべてのステートに対してブレークポイントを設定できる設計とした．当然，非サスペンダブルなステートに対してブレークポイントを設定した場合，ソフトウェア処理後に停止状態から正常に処理を再開できない事態が生じる可能性がある．

### 7.4.2 乗っ取り機構のモジュールへの実装の具体例

以下では，Initiator と RFend を例に，Martini への乗っ取り機構の実装について述べる．

#### Initiator

図 7.5に Initiator の状態遷移図を示す．

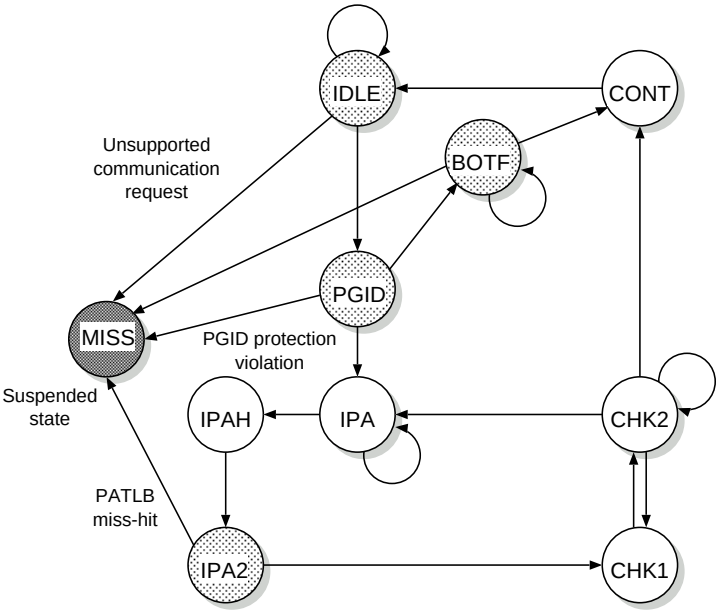



図 7.5 Initiator の状態遷移図

Initiator は，ホストから Window に通信要求が書き込まれると処理を開始する．ホストからの要求が PUSH であった場合は，Window の ID を用いて PGIDTBL を参照して対応する PGID と PID を取得し (PGID)，通信領域として用いるホストメモリの物理アドレスの取得 (IPA–IPA2) を行った後，DMA 要求を発行し (CHK1 および CHK2)，CONTTBL を参照して Window に書き込まれた処理が完了したことをホストに通知して処理を完了する (CONT).

図 7.5において網がけとなっているステートは例外を発生する可能性のあるステートであり，サスペンダブルに設計されている．オンチッププロセッサは，Initiator からの割込みを検出すると，Initiator 内のレジスタから Initiator が サスペンデッドステート に遷移する直前のステートを読み出し，それを元に乗っ取り要求の原因を判断して処理を行う．以下に，Initiator におけるサスペンデッドステート に遷移する直前のステートに対応する処理の一例を示す．

**IDLE** Window に書き込まれた内容が PUSH もしくは PULL の要求，および BOTF によるパケッ

ト発行のいずれでもなかったことを示す．オンチッププロセッサは Window に書き込まれた内容を元に，ソフトウェアによる要求処理を行うことになる．すなわちこれを利用することで，ユーザプロセスからファームウェアの処理を呼び出すことができる．

**PGID** 無効な Window から要求が発行されたことを示す．オンチッププロセッサは現在 Initiator が保持している通信要求をキャンセルし，Initiator のステートを IDLE に設定することで，次の通信要求の処理を開始する．

**IPA2** PUSH を処理する際，送信領域の物理アドレスを取得する段階で PATLB がミスヒットしたことを示す．オンチッププロセッサは PATLB のミスヒットしたエントリをリプレースメントした上で，Initiator のステートを IPA に設定し，PUSH プリミティブの処理を再開する．

### RFend

図 7.6に RFend の状態遷移図を示す．

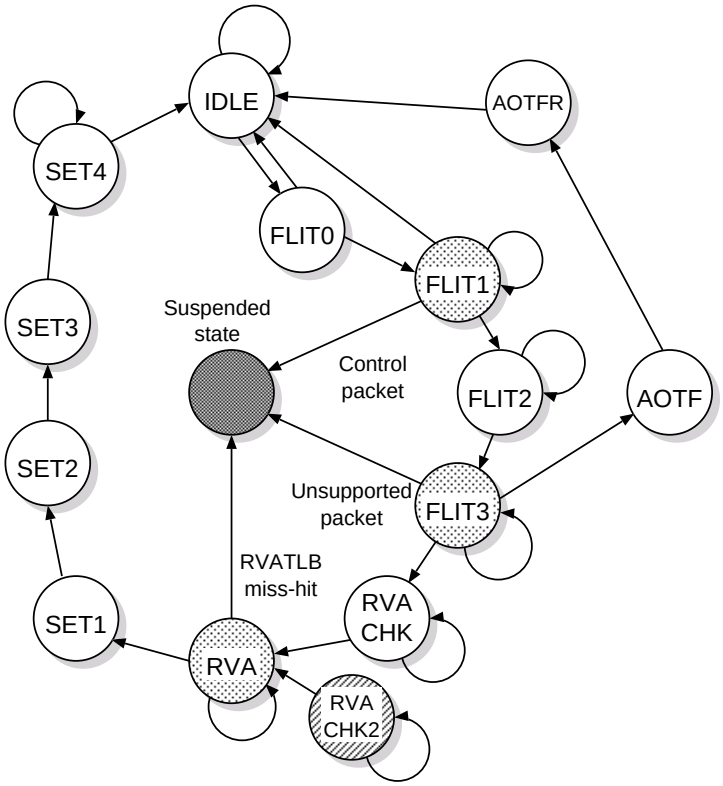



図 7.6　RFend の状態遷移

RFend はネットワークからのパケットの到着により処理を開始する．まずパケットヘッダの解析を行い (FLIT0–FLIT3)，次に SID を仮想アドレスに変換するアドレス変換を行う (RVACHK および RVA)．その後，ヘッダの解析結果や変換後のアドレスを RBend や Replier へ渡す (SET1–SET4)．受信したパケットが AOTF による特殊なパケットであった場合，低遅延で受信するために別ステートで処理を行う (AOTF および AOTFR)．

なお，図中の RVACHK2 は，RVATLB のリプレースメントを完了した後に，ハードウェア処理を安全に再開できるよう新規に追加したステートである．

図の網がけのステートはサスペンダブルに設計されている．以下に，RFend におけるサスペンデッドステートに遷移する直前のステートとそれに対応する処理の一例を示す．

**FLIT1** ネットワーク制御用の特殊パケットが受信されたことを示す．オンチッププロセッサはパケットに対応した処理を行い，その後，RFend のステートを IDLE に設定することで次のパケットの受信に備える．

**FLIT3** ハードウェア処理できないパケットが受信されたことを示す．オンチッププロセッサはパケットに対応するソフトウェア処理を行う．すなわちこれを利用することで，独自フォーマットのパケットを導入し，リモートでソフトウェア処理させることが可能となる．

**RVA** RVATLB でミスヒットが発生したことを示す．オンチッププロセッサは何らかの方法で RVATLB のミスヒットしたエントリをリプレースメントした上で，RFend のステートを RVACHK2 に設定し，ハードウェアによるパケット受信処理を再開する．

## 7.5 乗っ取り機構の評価

以下では Martini 上で，乗っ取り機構を利用した処理について評価を行い，その有効性について検討する．

### 7.5.1 評価環境

乗っ取り機構の評価は，実機を用いて行った．ただし，実機では計測不可能な一部の評価については，RTL シミュレーション [116] を用いた．

評価に用いた実機は，RHiNET-2/SW に対し，RHiNET-2/NI を搭載したノード PC を複数台接続した構成とした．ノード PC の仕様を表 7.2に示す．

表 7.2 乗っ取り機構の評価で用いたノード PC の仕様

| | |
|---|---|
| CPU | Intel Pentium III 933MHz × 2 (SMP) |
| Chipset | Serverworks Serverset III HE-SL |
| Memory | PC133 SDRAM 512Mbyte |
| PCI bus | 64bit/66MHz |
| OS | RedHat Linux 7.2 (kernel 2.4.21) |

評価で用いたリンクの最大データ伝送能力は 6章で述べた通り，4.8G+4.8Gbps となっている．

### 7.5.2 例外処理

乗っ取り機構を導入することで，例外を発生したモジュールのみを停止させ，無関係なモジュールは動作させたままオンチッププロセッサによる例外処理を行うことが可能となる．また，オンチッププロセッサが例外の要因を取り除いた後，モジュールのステートを例外発生前のステート

に設定することで，モジュールによるハードウェア処理を例外発生前の段階から再開させることが可能となる．以下では，これらの有効性を確認するために行った評価について示す．

### 例外処理の他のモジュールへの影響

まず，送信側もしくは受信側のどちらか一方のモジュールで例外が発生した際の，もう一方の処理能力への影響についての評価を行った．評価では，Martini を搭載したノード PC を 3 台用い，図 7.7に示す 5 通りのパターンで一定サイズの PUSH を連続して発行した際の，送信ノードでのスループットの合計を測定した．1 度の PUSH で送出するデータサイズは 128byte，512byte，2Kbyte，8Kbyte，32Kbyte，128Kbyte の 6 通りについて測定した．

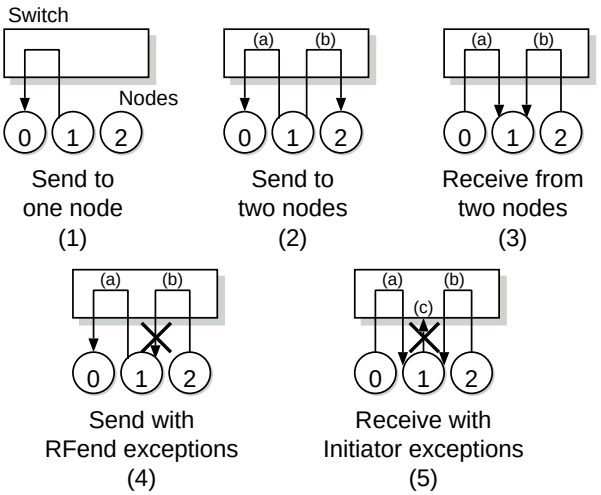



図 7.7　測定データ転送パターン

図 7.7の (1) は，3 ノード中 2 ノードのみを用いて，一方が PUSH によるデータ送出を続け，もう一方がそれを受信し続けるパターンである．(2) は 1 ノードがその他 2 ノードに対して交互にデータ送出を行うパターンであり，(3) は逆に 2 ノードが 1 ノードに対して同時にデータ送出を行うパターンである．(2) では受信のみを行うノードが分散されることからノードのデータ送信能力の上限を見ることができ，また (3) では送信ノードが複数存在することからノードのデータ受信能力の上限を見ることができる．(1) の結果を (2) および (3) の結果と比較することで，まず 1 対 1 通信におけるスループットが送信処理と受信処理のいずれにより制限されているかがわかる．

(4) は，(1) と同じ状況で，さらにノード 2 が送信ノード (ノード 1) に対してデータを送出し続けるパターンである．ただし，ノード 2 から送出されるパケット (図中 (b)) はノード 1 の RFend のステート RVA で必ず例外を発生させ，オンチッププロセッサにより読み捨てられる．また (5) は，(3) と同じ状況で，さらにノード 1 が PUSH 要求 (図中 (c)) を発行し続けるパターンである．ただし，この要求は必ず Initiator のステート IPA2 において例外を発生させ，オンチッププロセッサによってキャンセルされる．なお，(5) については，比較のためにノード 1 で例外が発生した後，通信要求をキャンセルせずに Initiator のステートを IDLE へ戻すようにした場合 (以下 (5’)) についても測定を行った．この場合，ノード 1 の Initiator は 1 度目の PUSH 要求で IDLE–PGID–IPA–IPA2–MISS の遷移を繰り返し，例外を発生し続けることになる．(4) および (5) の結果を (1) および (3) の結果と比較することで，例外処理が，例外と無関係なモジュールへ及ぼす影響を確認することがで

きる．

このようなパターンで通信を行い，各パターンでの送信側のノードのスループットの総和を求めた．結果を図 7.8に示す．ただし，例外処理と無関係な部分についての結果を見るために，例外を発生するものに関しては総和を求めず，(4) については (a) におけるスループットを，(5) および (5’) については (a) と (b) のスループットの和を求めた．

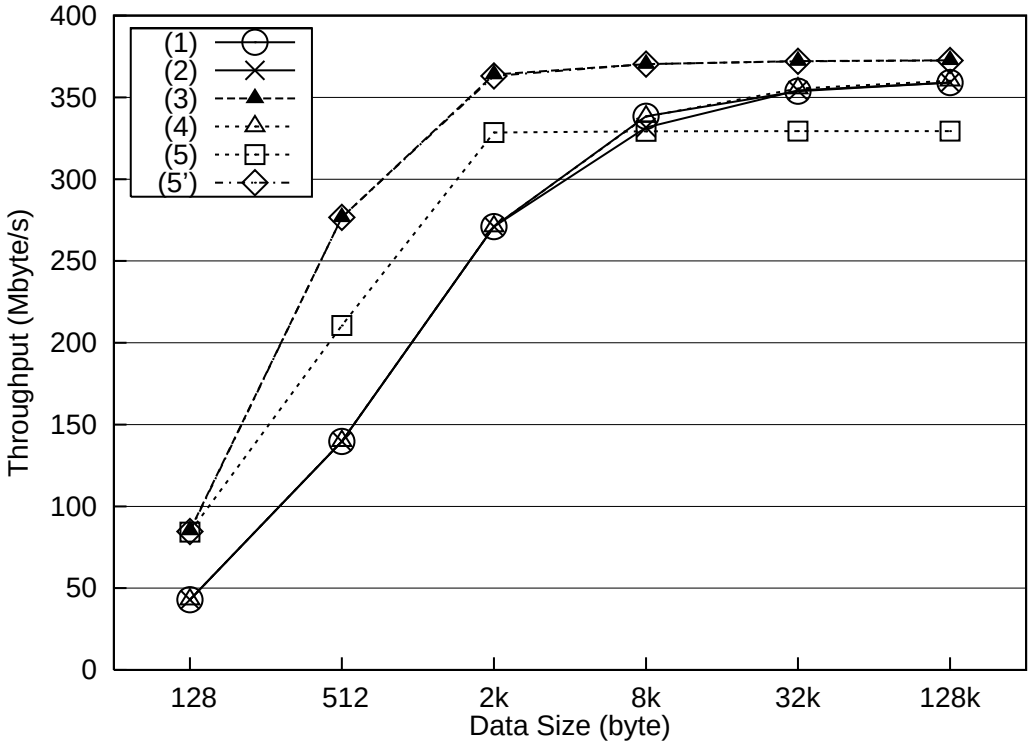



図 7.8　乗っ取り処理時のスループット

図 7.8より，(1) の結果は (2) の結果とほぼ一致し，(3) よりも低い値であることから，ここでの 1 対 1 通信時のスループットはノードのデータ送信能力の上限により制限されていることがわかる．

(4) ではノード 1 の Martini 上の RFend において例外が多発し，常時オンチッププロセッサが到着パケットを読み捨てる処理を行っているが，その間のノード 1 からノード 0 へのスループットは (1) の結果とほぼ一致している．このことから，ノード 1 の送信処理を行うモジュールは RFend での例外処理の影響を受けていないということがわかる．

(5) ではノード 1 の Initiator において例外が多発し，常時オンチッププロセッサが通信要求をキャンセルする処理を行っている．この場合，他の 2 ノードから受信されるデータのスループットはデータサイズ 2Kbyte 以上でほぼ一定値となってしまい，全体的に (3) に比べて低い．一方，(5’) の結果を見ると，Initiator で例外処理のみが繰り返されている場合は (3) と全く等しいスループットが得られている．これより，(5) が (3) に比べて低い値を示しているのは Initiator での例外処理の多発が直接の原因でないことがわかる．乗っ取り機構と直接関係ないため詳細について触れないが，(5) でスループットが制限されているのは，ノード 1 での PUSH 要求の密な発行による PCI バスの混雑が原因である．

(3) および (5’) の結果より，送信側もしくは受信側での乗っ取り機構による例外処理は，もう一

方の例外を発生していない側の処理能力に影響しないことがわかる．

### 例外処理後のハードウェア処理の再開

以下では，例外処理を行った後，例外発生モジュールを例外発生前のステートへ遷移させることでハードウェア処理を再開させることの効果について評価する．

評価では，2 ノード間での PUSH による Ping-Pong において，双方の Initiator でデータ送出時に PATLB のミスヒットが発生した際に，PATLB のリプレースメントを行った後，

- Initiator のステートを IPA に戻しハードウェアによる PUSH 処理を再開させた場合 (HARD)
- パケットヘッダ生成や DMA 要求などの処理をすべてソフトウェアで続けた場合 (SOFT)

のそれぞれについて，RTT の測定を行った．

結果を図 7.9に示す．参考値として，PATLB のミスヒットがない場合の RTT (NOEX) もあわせて示す．

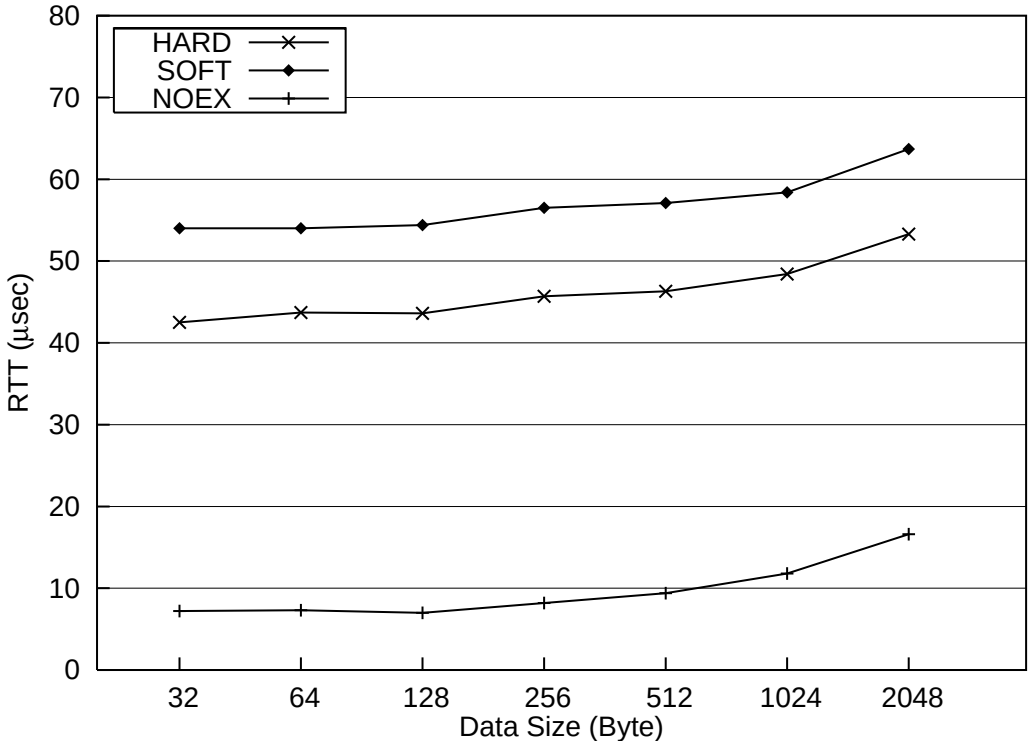



図 7.9　PATLB ミスヒット発生時の RTT

各データサイズにおける RTT は HARD の方が SOFT よりも常に約 10$\mu$sec 小さい値を示しており，また，データサイズによるこれらの値の変化は NOEX の値の変化とほぼ同一である．このことから，データサイズ増加時の RTT の増加は転送データ量に起因するものと考えられ，HARD と SOFT の RTT の差は Initiator における PATLB のリプレースメント後の PUSH 処理を，すべてソフトウェアで実行した際のオーバヘッドによるものであると言える．ここでは，乗っ取り機構を利用

してハードウェア処理を再開させることで，32byte 転送時に約 21%，2048byte 転送時に約 16%の処理時間の低減を実現している．

### 7.5.3 ソフトウェアによる通信処理

乗っ取り機構によるプロトコル処理の効率化の効果を確認するために，評価用の通信プリミティブとして VPUSH を実装した．VPUSH は，複数のプロセスから PUSH によって送られたデータを，受信側で動的に決定した任意のアドレスに書き込む通信機構である．

#### VPUSH 機構

Martini の PUSH では，書き込み先の領域は PUSH を要求したプロセスにより SID を用いて指定され，SID と受信領域の仮想アドレスの対応は受信側のネットワークインタフェース上の RVATLB で管理される．Martini は，受信のたびに RVATLB の値を増減させるような機構を持たないため，ソフトウェアで書き換えを行わない限り SID に対応する仮想アドレスは固定のままとなる．そのため，PUSH による書き込みが行われた際に，受信側で動的に受信先の領域を決定するような仕組みを実現するには，受信処理をハードウェアで行わずにオンチッププロセッサで行う必要がある．

VPUSH は，PUSH の受信処理をオンチッププロセッサで行うことで受信側での PUSH の受信先のアドレスを任意の値に動的に決定可能とする機構である．VPUSH において，受信時に書き込み先のアドレスを決定する以外の処理は，PUSH の受信処理とほぼ共通であることから，乗っ取り機構を利用することで処理の効率化をはかることができると考えられる．

#### VPUSH の概要

今回実装を行った VPUSH では，メッセージ通信の実装などの応用を想定し，ホストメモリ上にリング状の受信バッファを設け，到着順にデータが先頭から格納されるような設計とした．

受信バッファには，ホストが受信領域のどこまで処理したのかを示す解放ポインタと，どこまでが有効なデータであるかを示す受信ポインタが付随する．これらを用いて受信バッファの使用状況の管理を行う．

#### VPUSH の実装

VPUSH では，受信バッファに付随するポインタのうち，受信ポインタはホストがデータを処理した時点で更新し，Martini のオンチッププロセッサがホスト上の受信バッファが不足した際に必要に応じてこれをポーリングすることから，Martini 内のレジスタ上に設けることにした．一方解放ポインタは，受信のたびにオンチッププロセッサが更新を行うが，ホスト上のプロセスがデータ到着をポーリングする際に頻繁に読み出しを行うため，Martini 内のレジスタに置いてしまうと PCI へのアクセスが頻発し転送効率が下がる．そこで解放ポインタはホストメモリ上に設け，オンチッププロセッサからは DMA を用いて更新を行うことにした．

また，VPUSH の受信処理は，オンチッププロセッサのソフトウェアが行うが，パケット到着の段階からソフトウェア処理を開始してしまうとパケットヘッダの解析をソフトウェアで行う必要が生じてしまい効率が悪い．そこで，VPUSH が PUSH とほとんど同じ仕組みであり，受信処理

には PUSH のパケットヘッダに含まれる情報があれば十分であることに着目して，パケットヘッダの解析をハードウェアで行うことにした．RFend は，ステートが RVA に達した時点でヘッダ解析がほぼ完了した状態となる．そこで，VPUSH では，通常の push パケットを利用し，RVA で例外が発生するように，受信時に必ず RVATLB でミスヒットが発生する SID を指定して PUSH 要求を発行する実装とした．

RFend がステート RVA からサスペンデッドステートである MISS へ遷移すると，オンチッププロセッサは受信先のアドレスを計算し，RFend 内の SID からのオフセット値が書かれたレジスタの内容を受信先アドレスに更新した上で，RVATLB を参照しないようにレジスタの設定を変更する．その後の処理は通常の push パケットを受信した際の処理と全く同一であるため，ソフトウェア処理を一旦完了し，RFend のステートを RVA に設定することで，ハードウェア処理を再開させる．すなわち，push パケットの受信処理において，本来であればハードウェアで RVATLB を参照して受信領域の仮想アドレスを得る部分を，ソフトウェアでエミュレーションし，異なるアドレスを与えたことになる．

以上で push パケットの受信先のアドレスを，受信時に任意のものに変更可能となるが，VPUSH では，ホストの受信領域に完全にデータを転送し終えた後に解放ポインタを更新する処理をソフトウェアで行う必要があるため，再びソフトウェア処理に戻ることができるよう RFend にブレークポイントをセットしておかなければならない．しかし，RFend には RVA 以降 IDLE までのステートにサスペンダブルなステートがないためブレークポイントをセットすることができない．そこで，応答パケット生成モジュールの Replier にブレークポイントをセットする．Replier による応答パケット転送処理は DMA 転送が完了した直後に開始するため，ここにブレークポイントを設定しておくことで，後でソフトウェア処理に復帰できるようになり，ホストへの DMA 転送完了直後に確実に解放ポインタを更新することが可能となる．

VPUSH は，以上のようにソフトウェア処理の間に部分的にハードウェアによる処理を織り交ぜることで実装されている．なお，VPUSH における，送信側が 1 度の通信要求で送信可能なデータサイズの上限は RHiNET-2 の MTU と同じ 2048byte としている．これは，受信時に複数ホストから PUSH された転送データがインタリーブした状態でバッファに格納されてしまうことを防ぐためである．

#### VPUSH の処理の流れ

VPUSH の流れを図 7.10に示す.

送信側は，受信側で例外を発生する SID を伴った PUSH 要求を Martini に書き込み (1)，push パケットを発行する (2)．受信側では，このパケットを受信すると，RFend が RVA まで遷移したところで停止状態となり，オンチッププロセッサに乗っ取られた状態となる (3).

オンチッププロセッサは，RFend 内の受信アドレスを指すレジスタの値を Martini 側で管理する受信ポインタの値で上書きし (4)，同時に Replier に対してブレークポイントを設定する (5)．その後，受信処理を行うモジュールのステートを書き換えて RVA から処理を再開させる (6)(7).

データの受信処理が完了すると，Replier がブレークポイントで停止し (8)，オンチッププロセッサに乗っ取られた状態となるので，通信完了通知としてホスト上の受信ポインタの値を DMA で更新する (9).

受信側のホスト上のプロセスは，解放ポインタと受信ポインタを比較し，差がある場合にメッセージが到着しているものと判断して受信バッファのデータを処理する．処理が済んだ後は，解放ポインタを更新する．

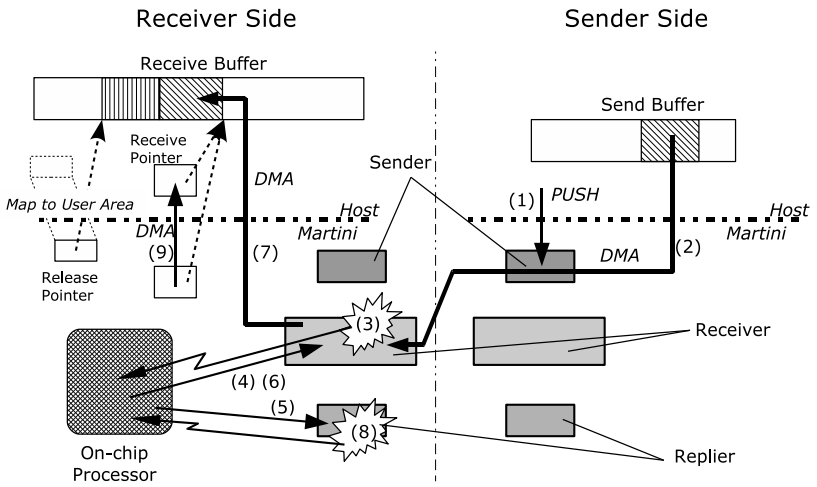



図 7.10　VPUSH の流れ

**VPUSH のスループット**

VPUSH の基本的な性能として，2 ノード間での VPUSH のスループットを測定した．評価は，例外処理の評価で用いたものと同一の環境で行った．

図 7.11に VPUSH の送信パケットのデータサイズとスループットの関係を示す (VPUSH)．また，オンチッププロセッサが RFend を乗っ取った後の処理をすべてソフトウェアで行った場合のスループット (VPUSH/Software) と通常の PUSH のスループット (PUSH) もあわせて示す．

VPUSH のスループットは転送データサイズが 2048byte の時に 180.3Mbyte/s に達した．同データサイズの転送を完全にソフトウェアで処理した場合のスループットは 78.6Mbyte/s であったことから，ハードウェアとの協調処理を行うことでソフトウェアのみで処理するよりも大幅に高い処理能力を実現できていることがわかる．

図 7.12は，RFend がデータサイズ 2048byte，512byte，8byte の VPUSH パケットの到着を検出して処理を開始した直後からオンチッププロセッサが Replier の例外処理を完了するまでの，VPUSH に関連したモジュールのステートや処理内容の変化を示している．なお，実機で各モジュールの細かな処理内容を測定することは難しいため，測定には RTL シミュレーションを用いた．

図 7.12の “CPU” はオンチッププロセッサの処理内容を示している．“main” は各ハードウェアモジュールからの乗っ取り要求 (割込み線) の変化をポーリングしている状態であり，“rfend handler” は RFend を，“replier handler” は Replier をそれぞれ乗っ取った際のソフトウェア処理を示している．

また，RFend，RBend，Replier について，“MISS” はサスペンデッドステートに滞在している状態 (すなわち停止状態) を示し，“WAIT” は DMA の完了待ち状態を，黒塗りの部分はモジュールがステートを遷移しながら処理を行っている時間帯を示している．DMA は DMA 転送が開始してから完了するまでの時間帯を示している．

図 7.12-(a) より，2048byte の VPUSH パケットを受信した際の処理時間の約 60%をオンチッププロセッサの処理が占めていることがわかる．この時間は約 6.6$\mu$s に相当し，リソースの競合などが生じない限りパケット長によらず一定である．

図 7.12-(a) の DMA 転送が行われている部分に着目すると，DMA 転送が完了するよりも前にソフトウェア処理の rfend handler が完了しており，オンチッププロセッサは “DMA 転送の完了を

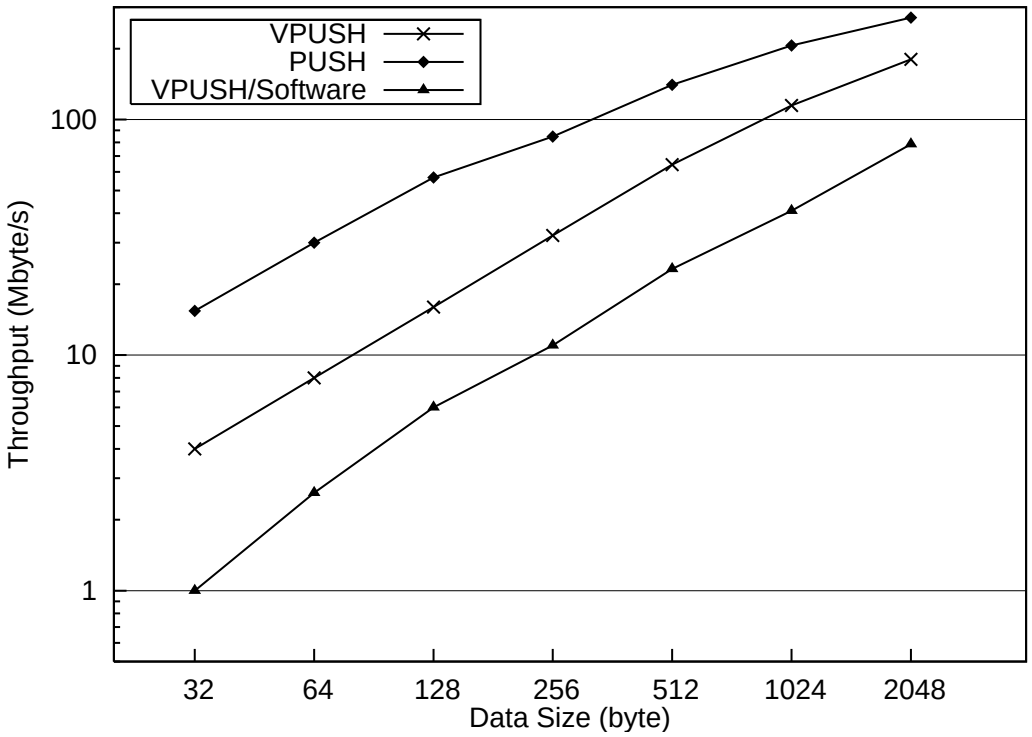



図 7.11　VPUSH のスループット

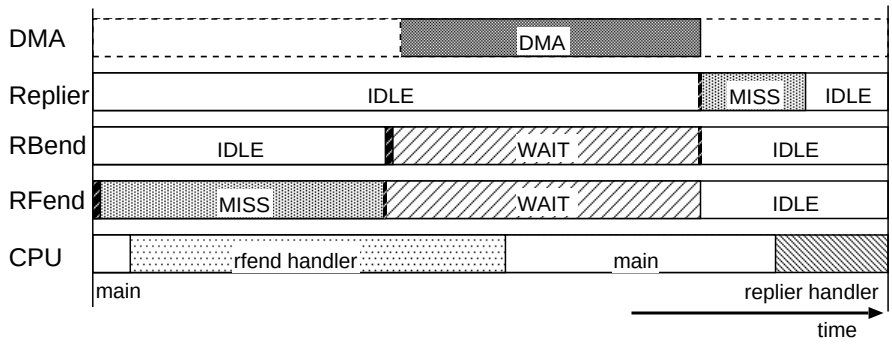
待っている Replier からの割込みを待つ” という状態になっていることがわかる．一方，8byte 転送時の図 7.12-(c) を見ると，ソフトウェアが rfend handler を実行している間に DMA 転送が完了しており，オンチッププロセッサは RFend の例外処理後，無駄なループをせずに即座に Replier の例外処理へと移っている．ここで，512byte 転送時の図 7.12-(b) を見ると，rfend handler と DMA 転送がほぼ同時に完了していることがわかる．このことから，512byte 以下の場合には，パケット処理時間がオンチッププロセッサの処理に支配され一定となり，512byte 以上の場合には，処理時間にパケット長により変化する DMA 完了待ち時間が加わることがわかる．

### 7.5.4　乗っ取り機構の実装によるハードウェア増加

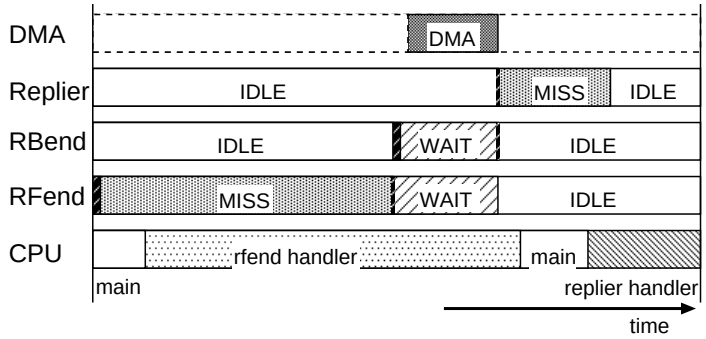
乗っ取り機構を実装するためには，オンチッププロセッサなどからモジュールに対して個別にバスを設け，モジュール内にサスペンデッドステートを設ける必要がある．また，それ以外にもモジュール内に外部からレジスタの値を読み書きするためのバスなどが必要となるため，ハードウェア規模は若干増大し，遅延も加わることになる．

バスの配線に関しては，Martini の場合，乗っ取り機構を実装したことでオンチッププロセッサからの書き込みに 37 段，読み出しに 59 段分のゲート遅延が加わった．そのため，バスの途中にラッチを加え，書き込みに 2 クロック，読み出しに 3 クロック要する構造とした．

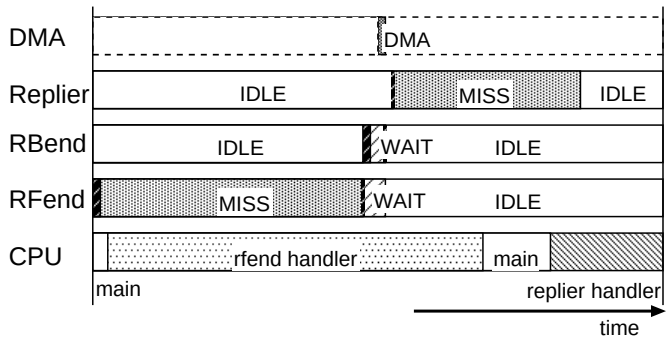
また，乗っ取り機構を追加することによるハードウェア量の増加について評価するために，Replier を通常のものと乗っ取り機構に関する記述を除外したものとでそれぞれ論理合成を行い，ベーシックセル数の比較を行った．結果を表 7.3に示す．




(a) 2048byte 受信時




(b) 512byte 受信時




(c) 8byte 受信時

図 7.12　VPUSH 処理時間の内訳

表 7.3より，Replier において乗っ取り機構を追加した場合のハードウェア量の増加は約 5.5%程度であることがわかる．

表 7.3　乗っ取り機構の有無に伴う Replier のハードウェア量の変化

| 乗っ取り機構あり | 乗っ取り機構なし |
|---|---|
| 9898 | 9384 |

### 7.5.5　乗っ取り機構に関する考察

ハードウェアに乗っ取り機構を実装するにあたり，すべてのステートをサスペンダブルとすることは設計が複雑化するため現実的ではない．そのため，どのステートをサスペンダブルに設計するかの判断が必要となる．少なくとも例外を発生する可能性があるステートはハードウェア設計の段階で明らかであるため，サスペンダブルな設計とすべきである．一方，乗っ取り機構を利用してソフトウェアからハードウェアの機能を用いる場合，元々例外の発生のないステートからモジュールを停止状態に遷移させた方が都合がよい場合も存在する．

VPUSH は乗っ取り機構を利用して実装したが，Replier の ack パケット発行処理を行う部分はサスペンダブルな設計となっていないため，DMA 転送完了を検出するために Replier をサスペンドさせた後，Replier を元のステートに戻して ack パケットを生成させることができず，Replier に渡された ACK 生成要求をキャンセルしてステートを IDLE に戻さざるを得ないなど，いくつか制約を伴っている．乗っ取り機構を利用したソフトウェア処理をより少ない制約で実現するには，ハードウェアの設計段階でソフトウェア処理に必要とされる機能についてある程度検討し，必要に応じて一部のステートをサスペンダブルな設計としておく必要がある．

また，乗っ取り機構は他のネットワークコントローラやストレージコントローラなどの専用ハードウェアが処理の中心で，オンチッププロセッサは補助的な処理を行うような構成のシステム LSI においても，Martini と同様の効果を得ることができると考えられる．

# 第8章　Martini向けPM通信ライブラリにおけるメッセージ通信の実装

RHiNET-2 を用いたクラスタ上でさらなる評価を進める上で，クラスタシステムソフトウェアの導入による実用的な並列分散処理環境の構築が必要となった．そこで稼働実績と移植性の高い既存のクラスタシステムソフトウェアである SCore[21][22] を導入することとした．本章の前半では，RHiNET-2 への SCore の導入に際して必要となった，Martini 向けの PM 通信ライブラリ [7] の設計・実装について，メッセージ通信の機能を中心に述べる．

また，実装した 2 種類の Martini 向け PM 通信ライブラリを利用して，実際に RHiNET-2 を用いたクラスタ上に SCore システムを導入し，MPI レベルでの性能評価およびアプリケーションレベルでの性能評価をあわせて行った．本章の後半では，これらについて示し，結果に関して考察を行う．

## 8.1　PM通信ライブラリの実装の背景

RHiNET に限らず，SAN や Ethernet などのインタコネクションネットワークを用いて構築したクラスタを用いて実用的な並列分散処理システムを実現するには，ノードに対するジョブの割付けや制御などを行う分散オペレーティングシステムや，MPI や OpenMP のようなユーザに馴染み深い並列プログラミング環境などを別途提供する必要がある．しかしながら，このような分散オペレーティングシステムや並列プログラミング環境を新規に実装して安定したシステムを実現するのは，膨大なコストを要し現実的ではない．

本研究では評価用に RHiNET-2 を用いた 64 ノード構成のクラスタを構築したが，これらに対応する分散オペレーティングシステムや並列プログラミング環境は十分に整備されていなかったため，これまで並列分散処理システムとして既存の実アプリケーションなどを用いた評価を十分に行うことができなかった．そこで，本研究では，稼働実績が高く，オープンソース体制で開発が進められているクラスタシステムソフトウェア SCore に着目し，RHiNET-2 を用いたクラスタ上にこれを導入することで実用的な並列分散処理システムを構築することとした．

SCore は，低レベル通信ライブラリである PM 上に構築される構造となっており，新規のネットワークで SCore を利用する場合，そのネットワークに対応した PM を実装することで，そのネットワーク上で SCore の提供する分散オペレーティングシステムや並列プログラミング環境を利用することが可能となる．したがって，RHiNET-2 を用いたクラスタ上で SCore を利用するには，RHiNET-2 向けに PM を実装しなければならない．

3.2節でも述べたように，PM はメッセージ通信と RMA 型の通信の 2 種類の通信モデルを提供する通信ライブラリであるが，これらのうち，特にメッセージ通信の実装は必須とされており，SCore の上位通信ライブラリの多くは PM のメッセージ通信の機能に依存している．これに対し，Martini は，RMA 型の通信である PUSH と PULL のみをハードウェアで提供しており，メッセー

ジ通信に相当する通信処理は別途ソフトウェアで実装する必要がある．

5.5.1節にて示したように，本研究で Martini 向けに実装を行った低レベルソフトウェアライブラリでは，メッセージ通信に相当する通信プリミティブとして SEND/RECV を提供しているが，Martini 向けに PM を実装するにあたりこれを利用せず改めて実装を行った．これは，既存の SEND/RECV よりも，より PM に最適化されたメッセージ通信を実現することと，RHiNET-2/SW の不具合(注 1)を回避してシステムを安定稼働させるためである．

## 8.2 SCore と PM 通信ライブラリ

SCore は，新情報処理開発機構で開発されたオープンソースのクラスタシステムソフトウェアであり，MPI ライブラリである MPICH[72] を移植した MPICH-SCore や分散共有メモリシステム SCASH[124] などの並列プログラミング環境を提供する．また，グローバルオペレーティングシステム SCore-D により，クラスタ上での並列プログラムの実行や並列ジョブのスケジューリング，クラスタのリソース管理などを実現する．これら SCore の提供する機能は，低レベル通信ライブラリである PM の通信機能を用いて実装されている．

### 8.2.1 PM 通信ライブラリ

PM は，ノード間の通信モデルとしてメッセージ通信と RMA 型の通信の 2 種類の通信モデルを定義している．これらのうち，MPICH-SCore や SCore-D はメッセージ通信を利用して実装されていることから，SCore の導入にはメッセージ通信の実装が必須となる．

PM のメッセージ通信は，以下の 4 つの API を基本とする．

- `pmGetSendBuffer`: 送信バッファ確保
- `pmSend`: メッセージ送信
- `pmReceive`: メッセージ受信
- `pmReleaseReceiveBuffer`: 受信バッファ解放

PM のメッセージ通信では，送信側では宛先ノードを指定してメッセージの送信を行い，受信側では送信元に関係なく到着順にメッセージを受信する．

送信側のプロセスは，`pmGetSendBuffer` 関数を呼ぶことで送信バッファの開始アドレスを取得し，得られたバッファに送信データを書き込んだ上で `pmSend` を呼んで送信を行う．送信の際は，宛先としてノード識別子のみを指定する．これら 2 つの関数は必ず交互に呼び出さなければならない．

一方，受信側のプロセスは，`pmReceive` 関数を呼ぶことで，メッセージが到着していた場合に受信メッセージが格納されている受信バッファの先頭アドレスを取得する．この時点でパケットヘッダに書かれた送信元に関する情報は読み取られず，メッセージは単純に到着順に取り出される．受信メッセージ内のデータを上位のライブラリなどが取り出すなどして処理を行った後は，

(注 1)複数ポート間で大量のパケットが同時に行き交うと，タイミングによってフロー制御に失敗してしまうことがある．これにより，ハードウェアの性能を最大限引き出した状態でアプリケーションレベルでの性能評価を行うことが困難となっている [26].

pmReleaseReceiveBuffer を呼んでバッファを解放する．これら 2 つの関数も，必ず交互に呼び出す必要がある．

## 8.3 PM/RHiNET

Martini 向けの PM を実装するにあたり，RMA に関しては，Martini が提供する PUSH および PULL をほぼそのまま用いて実装することができる．一方，メッセージ通信に関しては，8.1節にて述べたように，ソフトウェアで別途実装しなければならない．

ソフトウェアで実装を行う場合，Martini 向け低レベルソフトウェアライブラリで提供している SEND/RECV と同様に，ユーザレベルで PUSH および PULL を利用することで実装する方法と，ファームウェア上でメッセージ通信そのものもしくはそれを補助する通信機構を設け，ユーザレベルでこれを利用して実装する方法とが考えられる．6章で示したように PUSH や PULL の基本通信性能は高く，一方で Martini ではオンチッププロセッサの性能が低いことから，前者の手法を採用した方がメッセージ通信において高い通信性能が得られるものと考えられる．そこで，本研究では，まず SEND/RECV と同様に，ユーザレベルで PUSH および PULL を直接利用してメッセージ通信を実現する PM/RHiNET を実装した．以下では PM/RHiNET のメッセージ通信の実装について述べる．

### 8.3.1 PM/RHiNET の実装

#### メッセージの到着順処理の実現

PM では，同一ノードから連続して到着したメッセージは，受信時に先に到着したものから取り出されなければならないが，異なるノードからのメッセージ間では，どちらが先に到着したのかを保証する必要はない．そこで，PM/RHiNET では，SEND/RECV の実装と同様に，個々のノードに送信元のノードの数分の受信領域を設け，受信領域の使用状況を送信側で管理し，PUSH によって送信したメッセージが必ず送信順に受信領域に格納されるようにした．また，SEND/RECV では，受信関数を呼び出す際にどのノードから到着したメッセージを受信するのかを指定するため，受信確認を行うためにポーリングする領域は一意に定まったが，PM/RHiNET では受信関数で任意のノードからのメッセージの到着を検出しなければならない．そこで，受信関数が呼ばれた際には，メッセージが到着しているバッファが見つかるまで各ノードに対応する受信バッファを順番にアクセスして回る実装とした．

#### PUSH によるメッセージの転送

SEND/RECV と同様に，メッセージ本体の転送には，送信側が PUSH で直接メッセージを受信側に送る方式と，PUSH でメッセージの位置情報だけを送り，実際の転送は受信側が受信時に送信側から PULL するという方式とが考えられる．5.5.1節で述べたように，メッセージ通信の機能単体で見た場合，両者は一長一短であるが，PM で用いる場合，PULL を用いた実装では MPICH-SCore で，MPI_Gather のような多数のノードからのメッセージを受信する処理において性能が極端に低下するという問題が生じる．これは，複数ノードからメッセージがほぼ同時に到着し，それらを一気に受信しなければならない場合に，毎回 PULL を要求してからデータが受信し終わるまで待たなければならないためである．先に各ノードに対応する PULL をまとめて発行しておき，後

から PULL の完了確認をして回ることでこの問題は緩和できるが，その場合，上位レイヤの実装に変更を加えなければならなくなる．このような理由から，PM/RHiNET では PULL を用いずに，送信側から PUSH でメッセージ本体を転送する方式とした．

### 受信領域の管理

PM/RHiNET では，個々のノードでメッセージ受信領域を送信元のノードの数だけ設け，受信領域の使用状況を送信側で管理する．受信領域上のデータをどこまで処理したのかという情報は受信側で更新されるが，受信側で受信処理を行う度にこれを送信側に通知するのは無駄が多い．なぜなら，受信側のプロセスが受信領域のどこまでを処理して解放したのかを送信側のプロセスが知る必要があるのは，送信側が認識している受信側の受信領域の空きが不足したときだけだからである．このことから，PM/RHiNET では，受信領域の解放情報を受信側から送信側に積極的に通知せず，送信側が認識している受信側の受信領域の空き容量が不足した場合にのみ，この値を送信側が PULL で読み出して最新のものに更新する実装とした．

また，SEND/RECV と異なり，メッセージ本体を受信する受信バッファは詰めて利用せず，RHiNET-2 の MTU に合わせて必ず 2Kbyte 単位にアラインしたアドレスから受信するように利用する実装とした．これは，サイズの小さいメッセージを受信した際に，メッセージがページ境界をまたいでしまうことで，DMA が 2 回に分かれてレイテンシが増大してしまうのを防ぐためである．

### データ到着の検出

PUSH には，受信データの書き込みが行われたことを受信側のホストに通知する仕組みがない．そのため，受信側では，PUSH によってデータが書き込まれたことを検出するには，データの変化をポーリングする必要がある．

PUSH を用いた SEND/RECV では，ディスクリプタテーブルと呼ばれる受信メッセージに関する情報を格納する領域を設け，送信側が受信側のメッセージバッファに対してメッセージ本体を PUSH で書き込んだ後，ディスクリプタテーブルに対してディスクリプタを別途 PUSH で書き込み(注 2)，受信側はディスクリプタテーブルをポーリングすることでメッセージの到着の検出を行う実装としていた．この場合，ディスクリプタテーブルに収まらないサイズのメッセージを送るには PUSH を必ず 2 回発行しなければならないため，レイテンシの増大やパケット数の増加によるネットワークの混雑という問題を伴う．

そこで，PM/RHiNET では，メッセージを PUSH する際に，送信バッファ上のメッセージ本体にヘッダとトレイラを付加し，これらにディスクリプタに相当する情報を埋め込んで PUSH で送出する実装とした．受信側では受信バッファの末尾，すなわち PUSH によって新たなメッセージが書き込まれる領域の先頭をポーリングすることで，新規メッセージのヘッダを検出し，その後ヘッダに書かれたサイズを元にトレイラの位置を求めてポーリングすることでメッセージ全体がメッセージバッファに格納されたことを検出することになる．これにより，メッセージを送る際の PUSH の回数を 1 回に減らすことができ，無駄なパケットの削減ができる．ただし，この方式では，ヘッダおよびトレイラ検出のために常にメッセージバッファの空き領域が初期化されている必要があることから，`pmReleaseReceiveBuffer` が呼ばれた際に，あわせてメッセージバッファの内容をゼロに初期化しなければならない．

(注 2) メッセージがディスクリプタテーブルに収まるサイズの場合はメッセージを直接ディスクリプタテーブルに PUSH して書き込む．

### メッセージ通信の処理の流れ

PM/RHiNET におけるメッセージ通信の処理の流れの例を図 8.1に示す．図では，ノード 2 がノード 3 へメッセージを送信している．なお，図の各バッファには，メッセージが到着順に左から格納される．

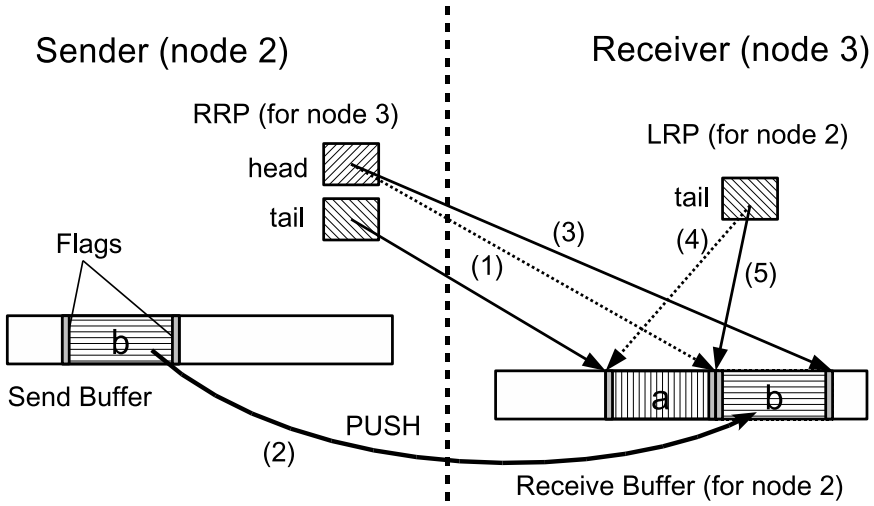



図 8.1　PM/RHiNET のメッセージ転送

PM/RHiNET では，各ノードは，全リモートノード数分の受信バッファと単一の送信バッファをユーザ空間上に確保する．これらをリングバッファとして利用するため，バッファごとにポインタを持つ．ここで，自ノードの各受信バッファの使用状況を管理するポインタを Local Receive Pointer (LRP) と称する．また，メッセージを送信順に受信側に格納できるよう，送信側にも各リモートノードの受信バッファの使用状況を管理するポインタを導入する．これを Remote Receive Pointer (RRP) と称する．

バッファの空き容量の管理は送信側で行うため，LRP はリングバッファ上の有効なメッセージの末尾 (最も古い未受信のメッセージの先頭部分) を指す tail ポインタのみを持ち，RRP は tail ポインタに加えて有効なメッセージの先頭 (次にメッセージを格納すべき領域の開始部分) を指す head ポインタを持つ．

送信側が `pmGetSendBuffer` によってバッファ領域を確保して `pmSend` を呼ぶと，リモートノードの受信バッファの head で示される領域に PUSH でメッセージが転送される．図の例ではノード 2 の送信バッファの `b` の領域に格納されたメッセージを，RRP の head で示されるノード 3 の受信バッファに PUSH し (1)(2)，その後 PUSH したサイズ分 head を進めている (3).

一方，受信側では，`pmReceive` が呼ばれると，LRP の tail の指す各受信バッファの末尾にアクセスしてメッセージの受信を確認する．メッセージが到着していればヘッダの値を元に受信領域のアドレスとサイズを返す．図の例では，tail の指す先に，以前に書き込まれたメッセージ `a` が格納されており (4)，これが受信されることになる．その後，`pmReleaseReceiveBuffer` が呼ばれバッファが解放される際に LRP の tail を更新する (5).

受信側は，この時点では LRP の tail の値を送信側に通知せず，送信側で RRP の head と tail から求まる受信側のバッファの空き容量が不足した場合にのみ，PULL を用いて LRP の tail の値を

RRP の tail に読み出して最新の情報に更新する．

図 8.2に RRP の tail の更新の様子を示す．

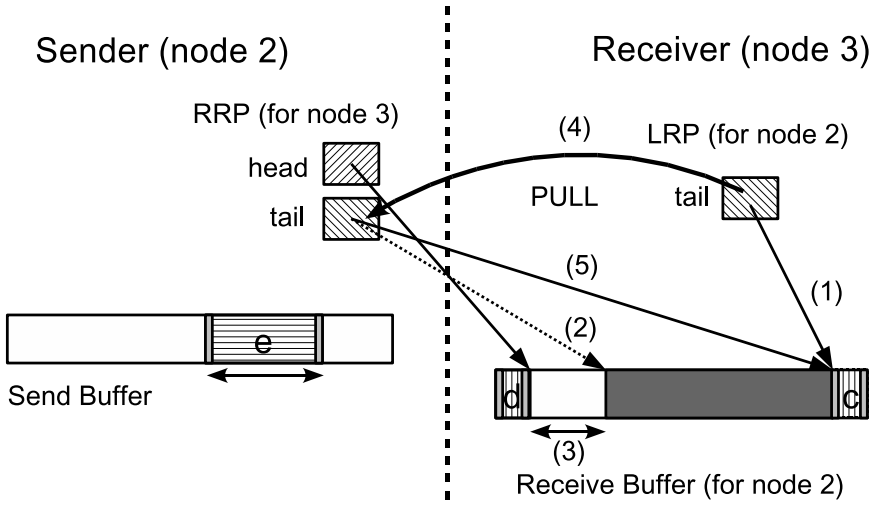



図 8.2　PULL による tail ポインタの更新

図 8.2では，ノード 3 の，ノード 2 に対応する受信バッファには，LRP の tail で示される領域からメッセージが格納されている (1)．一方，ノード 2 の持つ RRP の tail の値は，更新がなされていないため値が古く，ノード 2 は図中の領域 c よりも左の領域にも未受信のメッセージが格納されていると認識している (2)．ここで，ノード 2 がメッセージ e を送信するにあたり，RRP の head と tail の差から求まるノード 3 の受信バッファの空き領域 (3) が不足するとわかると，ノード 2 は PULL を用いてノード 3 の LRP の tail の値を自身の RRP の tail の領域に読み出し，最新の情報に更新する (4)(5)．これにより，ノード 2 は，ノード 3 に十分な受信側に空き領域が存在することを認識し，メッセージの PUSH が可能となる．

### 8.3.2　PM/RHiNET の特徴と問題点

PM/RHiNET ではメッセージの転送に PUSH を用いているため，Martini のハードウェアによる通信処理を活かすことができ，高い基本通信性能を発揮することが期待される．

しかし，一方で，すべてのノードが送信ノードごとに十分な大きさの受信領域を確保しなければならないため，通信ノード数の増加に比例して受信バッファとして必要なメモリが増加してしまうという問題がある(注 3)．また，受信データが到着していない場合，1 度の `pmReceive` の呼び出しですべての受信領域の受信状況を確認して回ることになるため，ノード数が増大した場合に受信のオーバヘッドが大きくなってしまうことが予想される．この問題を検証するために，実機を用いた測定を行った．

(注 3)この点に関しては SEND/RECV も同様である．

### 評価環境

評価は，64 ノード構成の RHiNET-2 クラスタ (図 8.3) のうち，16 台のノード PC を，完全結合した 4 台の RHiNET-2/SW に対してそれぞれ 4 台ずつ接続することで構築したクラスタ上で行った．評価に用いたノード PC の構成は 7.5節において表 7.2で示したものと同様である．

また，PM については，SCore バージョン 5.8.2 に付属のものを元に実装を行った．PM/RHiNET の各ノードに対応する個々の受信バッファのサイズは 128Kbyte とし，すべてページ境界にアラインした配置とした．スループットおよびレイテンシの評価では，SCore に付属のテストプログラムである `pmtest` を用いた．

図 8.3　RHiNET-2 クラスタ

### ノード数のレイテンシに対する影響の評価

まず，ノード数が増えた場合に，メッセージの到着検出に要するオーバヘッドが受信レイテンシにどの程度影響するかを評価するため，全体のノード数を 2，4，8，16 と変化させた際の 2 ノード間のピンポン転送時の RTT を測定した．ピンポン転送の RTT は，2 ノードが互いに相手からのメッセージを受信後，直ちに返送する処理を繰り返すことで測定している．結果を図 8.4に示す．

全体のノード数が 8 までは，RTT は 2 ノードの場合と比べてほとんど変化がないが，16 ノード時に約 1.8$\mu$sec が大きくなっている．この差は，受信バッファの確認処理の際の，ホスト CPU の L2 キャッシュや TLB におけるミスヒットによるオーバヘッドに起因すると考えられる．

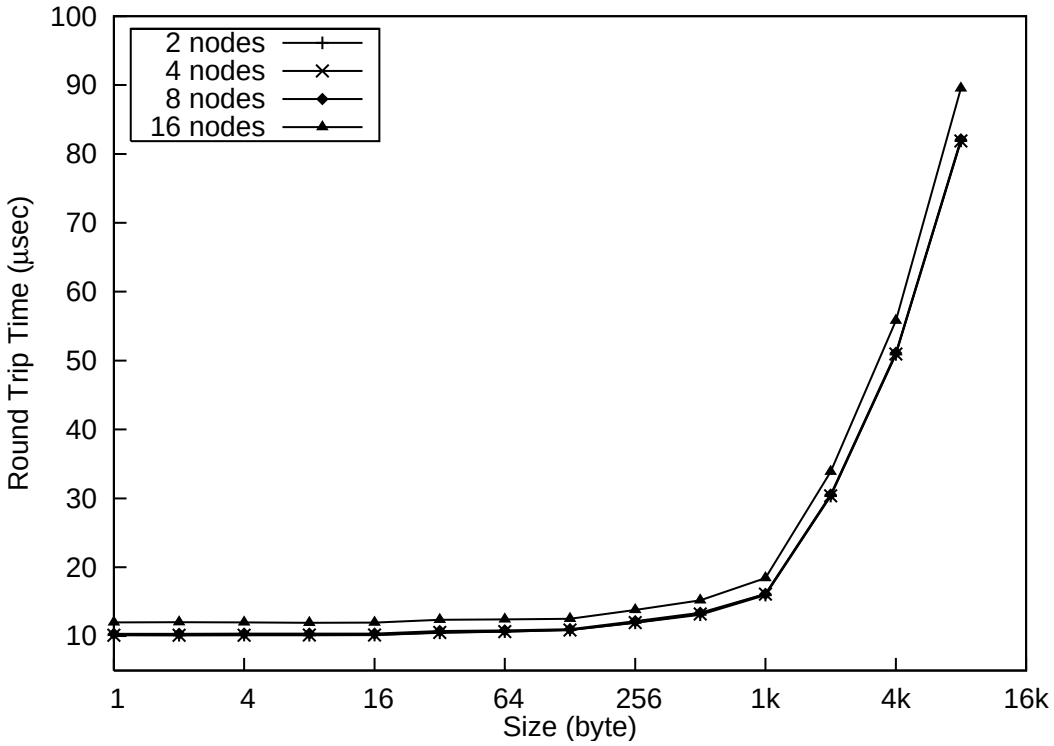



図 8.4　PM/RHiNET におけるノード数増加の RTT への影響

先に述べたように，PM/RHiNET では，PUSH の書き込み先の領域は必ず 2Kbyte 単位にアラインしたアドレスとなる．そのため，メッセージ到着のためにポーリングする領域は 2Kbyte にアラインされたアドレスとなり，キャッシュ上でのポーリング対象領域のラインが衝突する確率はランダムなアドレスをポーリングした場合と比べて大幅に高くなる．また，受信バッファの確認の際に広範囲にアクセスを行うことで TLB が汚染されてしまうため，メッセージ受信後のアプリケーションの実行性能にも影響するものと考えられる．

そこで，メッセージ到着確認によるメモリアクセスが性能に与える影響を確認するために，同一のホスト PC 上で，受信バッファに見立てたバッファを複数用意し，各々に対して順番にアクセスしてすべてに対するアクセスが完了するまでに要する時間の測定を行った．結果を図 8.5に示す．

各受信バッファのサイズは，評価で用いた PM/RHiNET と同様に 128Kbyte とし，ページ境界にアラインするよう配置した．測定では，メッセージ到着検出処理として，バッファ上の 2Kbyte にアラインしたアドレスから 4byte のデータの読み出しを行った．図中の “Head” は，特定のノードとの間でのみメッセージ通信が行われている状況を想定して，各受信バッファの先頭アドレスをアクセス対象として順番にアクセスして回った際の所要時間を示している．一方，“Random” は，各ノードからある程度メッセージが到着し，受信バッファが適度に埋まった状況を想定して，個々の受信バッファ上の 2Kbyte 単位にアラインされたアドレスからランダムに選択した一つを順番にアクセスして回った際の所要時間を示している．Head ではバッファ数 64 以上で，Random ではバッファ数 128 以上で，それぞれバッファ確認に要する時間が大幅に増大していることがわかる．Random の場合，アクセス対象はページ先頭か，ページ先頭 + 2*Kbyte* となるのに対し，Head ではアクセス対象はページ先頭のみとなるため，Head は Random に比べてアクセス対象のキャッ

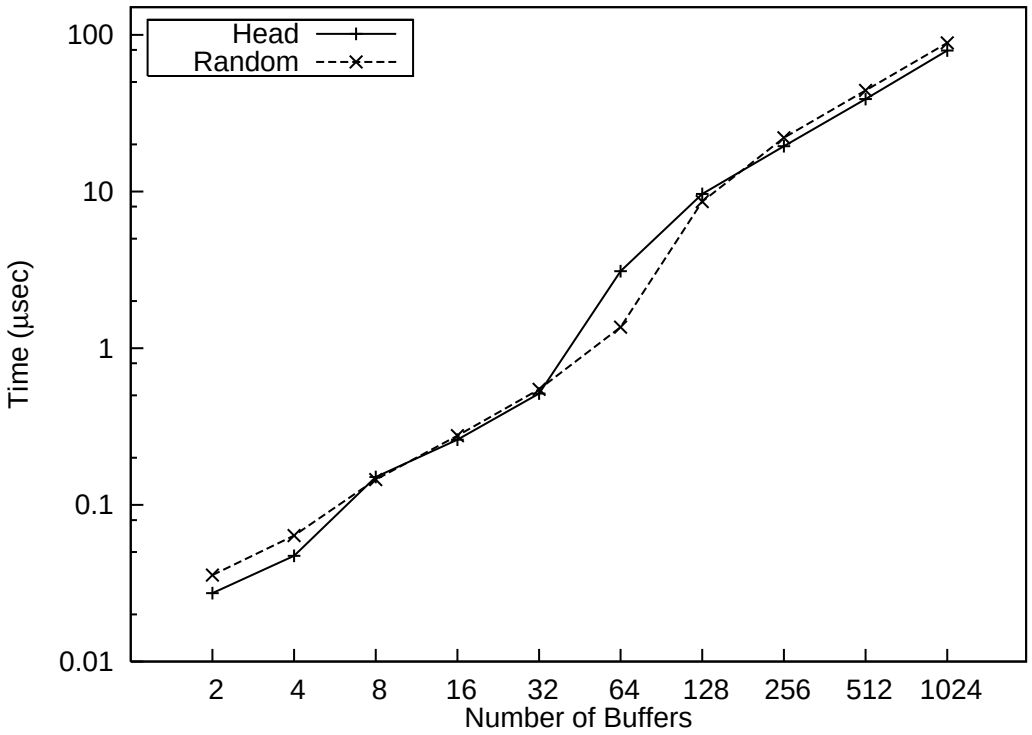



図 8.5　メッセージ到着検出時のバッファアクセスの所要時間

シュラインの衝突確率が 2 倍となる．バッファ数 64 で両者の値に差が生じているのはこのためである．バッファ数が 128 以上になると，Random と Head の間でキャッシュラインの衝突の頻度に大きな差がなくなることからほぼ同一のアクセス時間を示すようになっており，128 ノードの場合少なくとも 9$\mu$sec 程度，それ以上ではノード数の増加に比例して受信処理のオーバヘッドが増大している．この結果より，PM/RHiNET を用いた場合，ノード数の増大に伴いメッセージ到着確認時間が RTT の数倍に達し，上位通信ライブラリを組み合わせた通信処理の性能に大きく影響を及ぼすことが予想される．

なお，図 8.5に示した結果では，バッファ数が 16 の場合，図 8.4で示した結果と異なり，メッセージ到着検出に伴うオーバヘッドはほとんど発生していない．これは，図 8.5の測定ではメモリアクセスのみを連続して行っているため，メッセージ到着検出後に別の処理を行う PM/RHiNET を用いたベンチマークに比べ，キャッシュや TLB のミスヒットが発生しにくいことによると考えられる．

### ノード数のスループットに対する影響の評価

次に，全体のノード数を 2，4，8，16 と変化させた際の 2 ノード間のバースト転送におけるスループットの測定を行った．スループットは，送信側のノードが受信側ノードに対して連続してメッセージを送信し，受信側のノードは受信のみを行うことで測定をしている．結果を図 8.6に示す．

スループットは，転送サイズが小さいうちは，2 ノードの場合も 16 ノードの場合もほとんど差

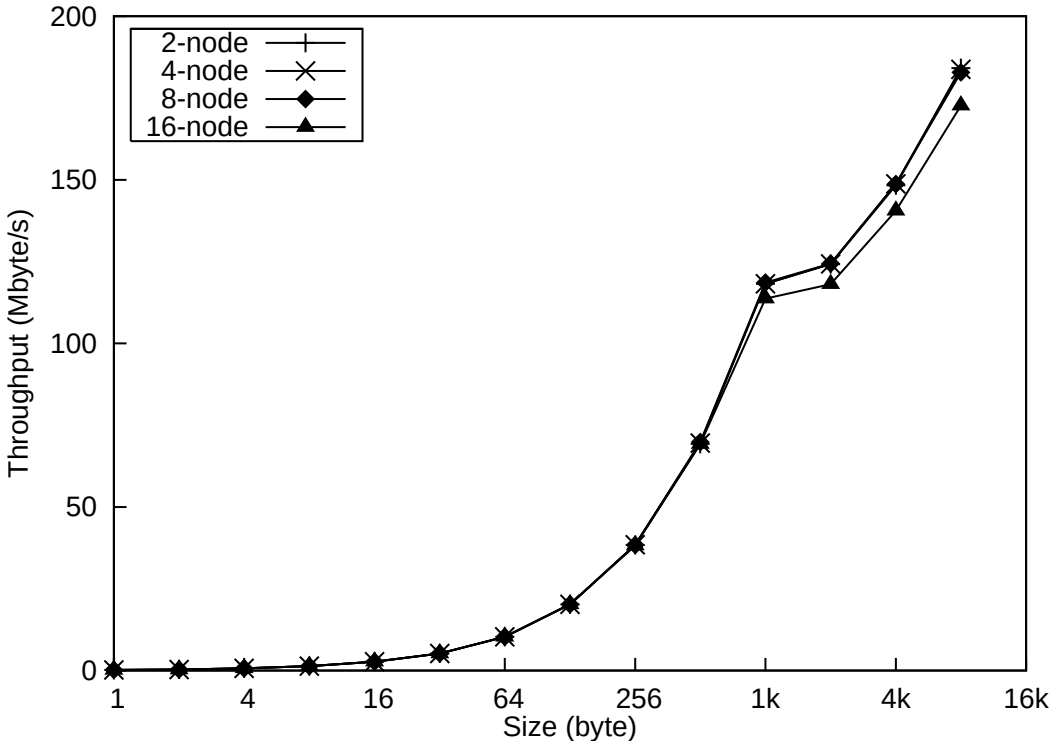



図 8.6　PM/RHiNET におけるノード数増加のスループットへの影響

がないが，転送サイズが 1Kbyte 以上では，16 ノードの場合のみわずかに低くなっている．

メッセージサイズが小さい場合，メッセージ受信関数が呼ばれてメッセージを受信してから再度メッセージ受信関数が呼ばれるまでの間に次のメッセージが到着するため，無駄な受信バッファの参照がほとんど生じずノード数の影響が出にくい．これに対し，メッセージサイズが 1Kbyte 以上の場合，ホスト上で受信バッファの先頭に PM のメッセージのヘッダが新たに書かれたことを確認した直後に，メッセージのトレイラが格納される領域をアクセスしても，受信データのホストメモリへの DMA 転送が完了していないため，データ末尾の検出ができないという状況が発生する．その際，PM/RHiNET は一旦受信をあきらめ，他のノードからのメッセージ到着確認を行うが，図 8.4に示したように，16 ノードの場合，8 ノードの場合に比べて受信バッファ全体を確認して回るのに時間を要する．これがスループットの低下につながっているものと考えられる．

## 8.4　PM/RHiNET-VP

これまで述べたように，PM/RHiNET の実装では，メッセージの到着検出のためのバッファの巡回処理により実際に通信性能が悪化することが明らかとなった．この問題は，16 ノード程度の小規模なクラスタでは影響が小さいが，数百台規模のノード数となった場合，上位アプリケーションの実行性能に対して深刻な影響を及ぼすものと考えられる．そこで，この問題を解決するために，新たに PM/RHiNET-VP と呼ばれる RHiNET-2 向けの PM を実装した．

PM/RHiNET-VP では，メッセージ通信における受信処理に 7.5.3節で述べた VPUSH を利用する．VPUSH では，PUSH により送出されたデータを受信側が任意の領域に格納することができ，

7.5.3節ではメッセージを到着順にホストメモリ上のバッファに格納する例を示した．この処理は，PM のメッセージ通信の受信処理でほぼそのままの形で利用することができる．メッセージを一続きのキューに受信することで PM/RHiNET で問題となったメモリ消費量の節約とメッセージ到着検出時のオーバヘッドの削減が実現し，ノード数が増えた場合にもホストのメッセージ受信検出によるオーバヘッドで 2 ノード間の通信性能が低下することはなくなるものと考えられる．

### 8.4.1 PM/RHiNET-VP の実装

PM/RHiNET-VP では，PM の関数呼び出しを，7.5.3節の図 7.10で示した VPUSH の各処理に対応させることで実装した．

送信側のプロセスは，まず `pmGetSendBuffer` でピンダウンされた送信バッファを確保し，ここに送信するメッセージを書き込む．次に `pmSend` で，先ほど確保した領域のデータを VPUSH 用の SID をつけて宛先ノードに対して PUSH で送出する．

受信側では `pmReceive` が呼ばれた際に，解放ポインタと受信ポインタの比較をし，メッセージの到着を検出する．両者に差がある場合，メッセージが到着していることになるため，メッセージからサイズ情報を読み取り，未受信のメッセージの先頭のアドレスをプロセスに渡す．その後，`pmReleaseReceiveBuffer` でバッファを解放する際に，解放ポインタを更新する．

### 8.4.2 PM/RHiNET-VP の特徴と問題点

PM/RHiNET-VP では，単一の受信領域に全プロセスからのメッセージを受信できるため受信領域を端から詰めて無駄なく利用することができ，プロセス数が増えた場合にも受信領域の容量をノード数分増やす必要がない．また，メッセージの受信は受信ポインタと解放ポインタの差を取るだけで検出できることから，全受信領域を見て回る必要のあった PM/RHiNET に比べ，`pmReceive` 発行時のホストの処理オーバヘッドを大幅に低減できることになる．

しかし，一方で，受信処理にソフトウェア処理が介在することから，受信処理のレイテンシが大きくなってしまうという問題が予想される．また，VPUSH の都合上，PM のメッセージサイズを RHiNET-2 の MTU 以下に制限しなければならない点や Replier が機能しなくなるため pull 応答パケットの発行ができなくなり PM の RMA 機構を実装することができないといった点など，制限が多く含まれる．これらにより，サイズの大きなメッセージの転送においてスループットを上げにくいと考えられる．

### 8.4.3 PM/RHiNET-VP の機能通信性能の評価

8.3.2節で用いた評価環境と同様の環境で，以下の 2 項目に関して PM/RHiNET-VP の 2 ノード間の基本通信性能の評価を行った．また，比較のため，PM/RHiNET に関しても同じ条件で評価を行った．

- ピンポン転送時のメッセージ通信の RTT
- バースト転送時のメッセージ通信のスループット

いずれも測定では SCore に付属の `pmtest` を用いた．また，メッセージ到着検出によるオーバヘッドを含まないよう，全体のノード数は 2 とした．

**PM のメッセージ通信の RTT**

PM/RHiNET-VP と PM/RHiNET の RTT の測定結果を図 8.7に示す.

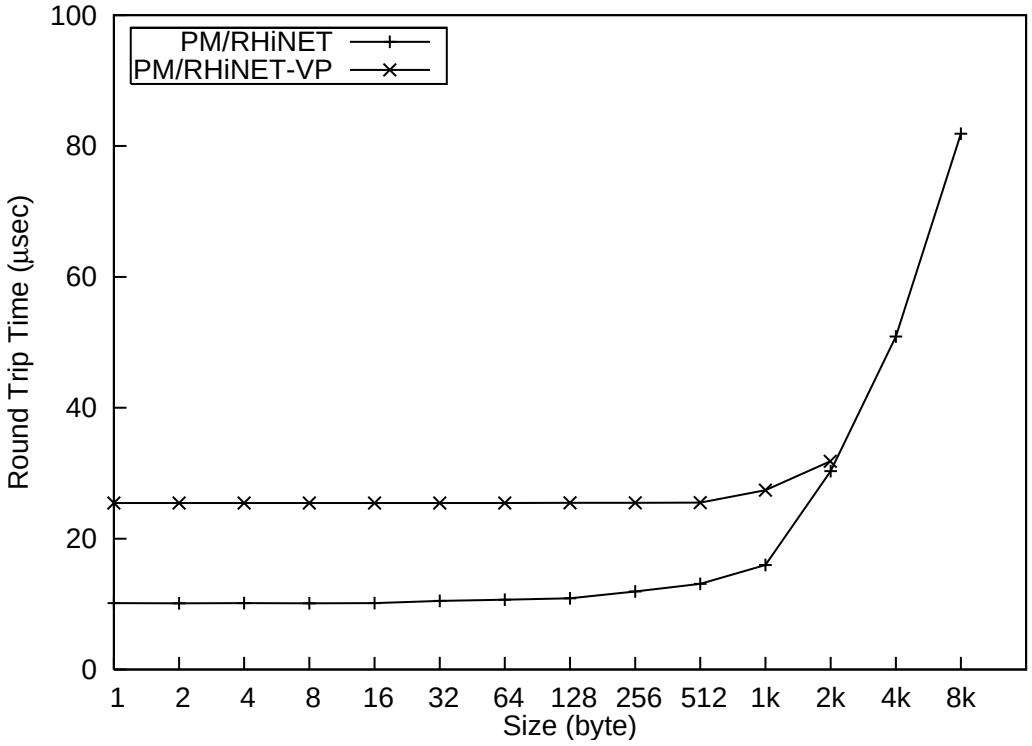



図 8.7　PM/RHiNET-VP と PM/RHiNET のメッセージ通信の RTT

最小 RTT は，PM/ RHiNET-VP では 25.5$\mu$sec，PM/RHiNET で 10.1$\mu$sec となり，両者を比較すると，1Kbyte 以下では PM/RHiNET-VP の RTT の方が大きな値を示していることがわかる．

PM/RHiNET の RTT は，メッセージサイズが 1Kbyte と 2Kbyte の間で急激に増加している．これは，PM/RHiNET に対して，RHiNET-2/SW の不具合を発生させないように通信量を抑えるチューニングが施されているためである．PM/RHiNET では，メッセージサイズが 2040byte を超えた時点でメッセージを複数回の PUSH 要求で送り出すようにし，さらに次のメッセージを PUSH で送出する前に，push パケットに対する ack パケットの到着を必ず待つことで，ネットワークの混雑を低下させ，RHiNET-2/SW の不具合を回避している．

これに対し，2040byte 以下のメッセージについては，ピンポンにおける ping と pong のいずれも単一の push パケットで済み，pong に相当する push パケットが ack パケットの直後に到着するため，次の ping に相当する push パケットを送信するまでに ack パケットを待つ必要は生じない．よって，2040byte 以下のメッセージサイズの PM/RHiNET の RTT は，RHiNET-2/SW の不具合を回避するためのチューニングの影響を受けていないと言える．

一方，PM/RHiNET-VP の RTT は，メッセージサイズが 512byte 以下ではほぼ一定しており，メッセージサイズに応じたレイテンシの増加は見られない．これは，PM/RHiNET-VP で用いている VPUSH において，ハードウェア処理による DMA 転送などと並列に実行されるソフトウェア処理が転送サイズによらず一定の処理時間を要するためである．

**PM のメッセージ通信のスループット**

PM/RHiNET-VP と PM/RHiNET のスループットの測定結果を図 8.8に示す．測定は，PM/RHiNET では 8184byte まで，PM/RHiNET-VP では 2040byte までと，それぞれで送ることのできるメッセージサイズの最大値まで行った．

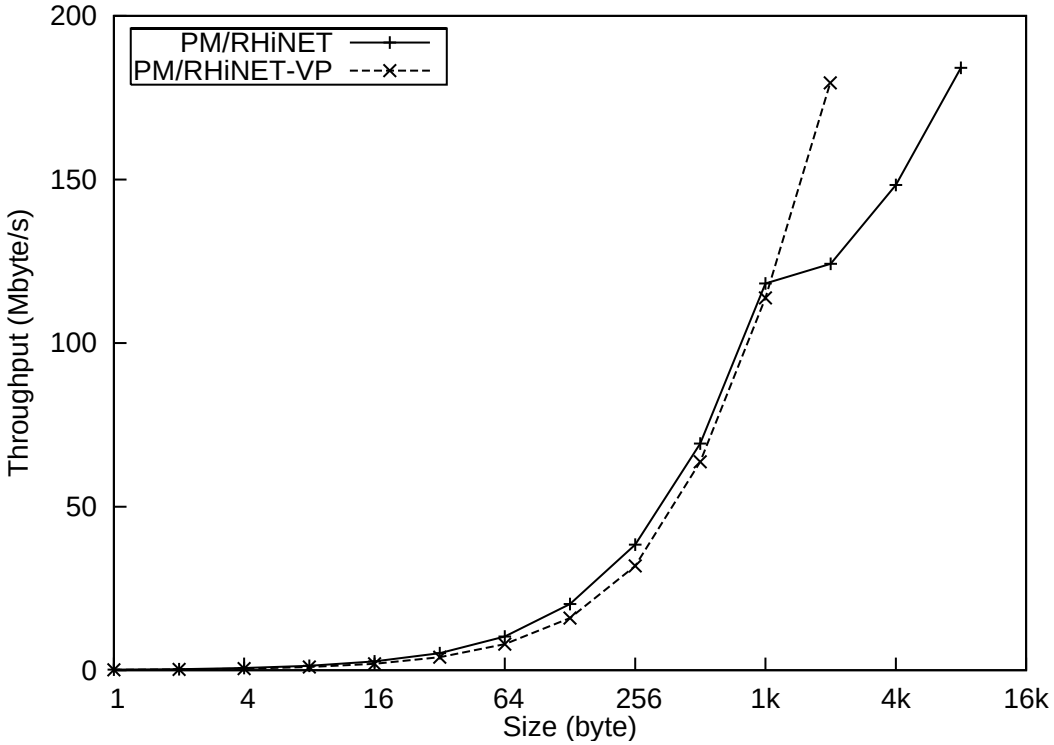



図 8.8　バースト転送時のメッセージ通信のスループット

バースト転送時のスループットは，1024byte までは PM/RHiNET と PM/RHiNET-VP との間で差が小さく，RTT とは異なる傾向を示している．これは，PM/RHiNET では，8.4.3節で述べたように直前に送信した push パケットに対応する ack パケットの到着を確認してからでないと次の push パケットを送り出せないという制約が課されているために，バースト転送時にパケット送信の間隔が空いてしまうことに起因する．

## 8.5　MPI レベルでの基本通信性能

PM/RHiNET および PM/RHiNET-VP の実装について，それぞれ上位レイヤから利用した際にどの程度の性能が得られるかを検討するために，MPI レベルでの基本通信性能の評価を行った．

評価は 8.3.2節で用いたものと同一の環境で行った．MPI には SCore-5.8.2 付属の MPICH-SCore を用い，MPI 用のベンチマークには Pallas MPI Benchmarks (PMB) V2.2.1[125] を用いた．

**MPI レベルでのスループット**

図 8.9に，PMB_Sendrecv ベンチマークにより測定したスループットを示す．PM/RHiNET は RMA に対応しているが，安定動作させるため，通常は RMA を無効にしている．図 8.9にの “Zerocopy” は RMA を有効にした場合の参考値として示してある．

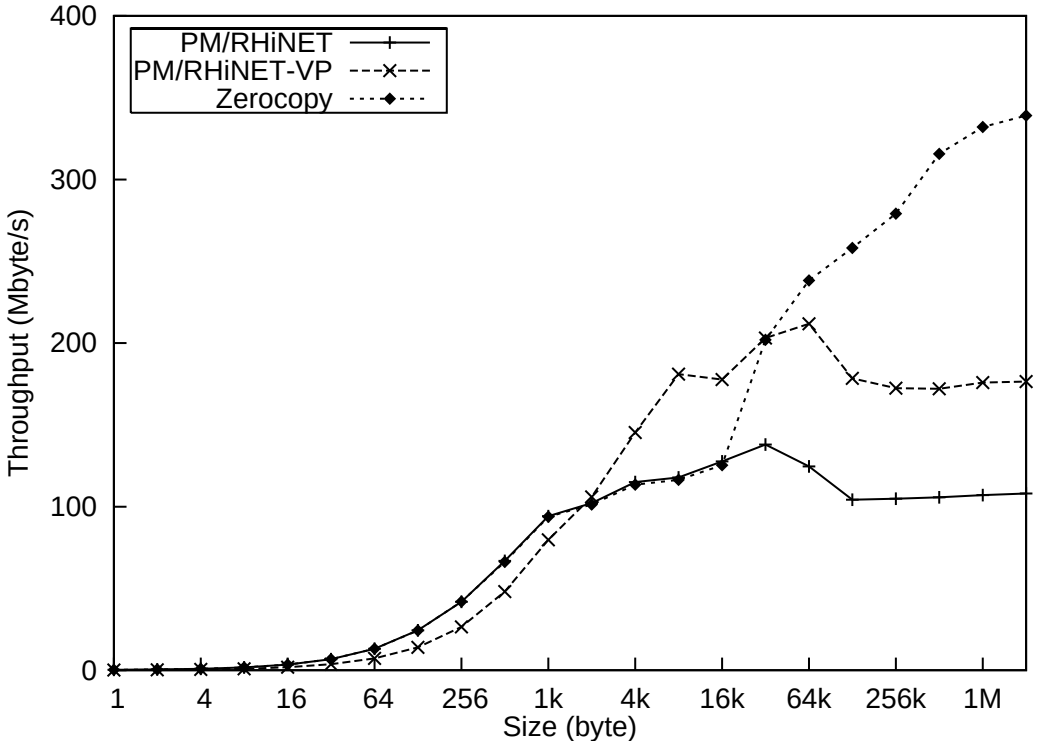



図 8.9　MPI のスループット

PM/RHiNET を用いた場合の MPI レベルでのスループットは，PM/RHiNET の性能の傾向がそのまま表れており，2Kbyte を境に向上が緩やかになっている．また，PM/RHiNET と PM/RHiNET-VP のどちらを用いた場合でも，スループットは 32Kbyte 付近でピークに達し，それより大きいメッセージサイズでは横ばいとなっている．このスループットのピークは，評価で用いたベンチマークプログラムである PMB において，`MPI_Sendrecv` を呼ぶ際に送受信バッファとして指定する領域の多くが CPU の L2 キャッシュに収まってしまっていることに起因している．PMB_Sendrecv では，`MPI_Sendrecv` を複数回連続発行して計測を行うが，その際，送受信バッファとして毎イテレーション同一の領域を繰り返し用いている．そのため，送受信のメッセージサイズが小さい場合は，送受信バッファが L2 キャッシュに収まりやすくなり MPI の API 呼び出し後に MPICH 内部で行われるメモリ間コピーが高速に完了するが，メッセージサイズが大きくなるとメモリ間コピーにおいてキャッシュラインのリプレースメントが多発して，速度が低下する．

なお，RMA 機構を有効にした場合，16Kbyte を境に MPICH-SCore 内部での通信方式が Eager から Rendezvous に切り替わり，16Kbyte より大きいサイズでは RMA を用いてメッセージ転送を行うようになるため，メモリ間コピーに伴う L2 キャッシュのリプレースメント処理によるスループット低下の影響は見られず，300Mbyte/sec 以上の高いスループットを示すようになる．

**MPI レベルでの通信レイテンシ**

次に，PMB_PingPong ベンチマークによるレイテンシの測定結果を図 8.10に示す.

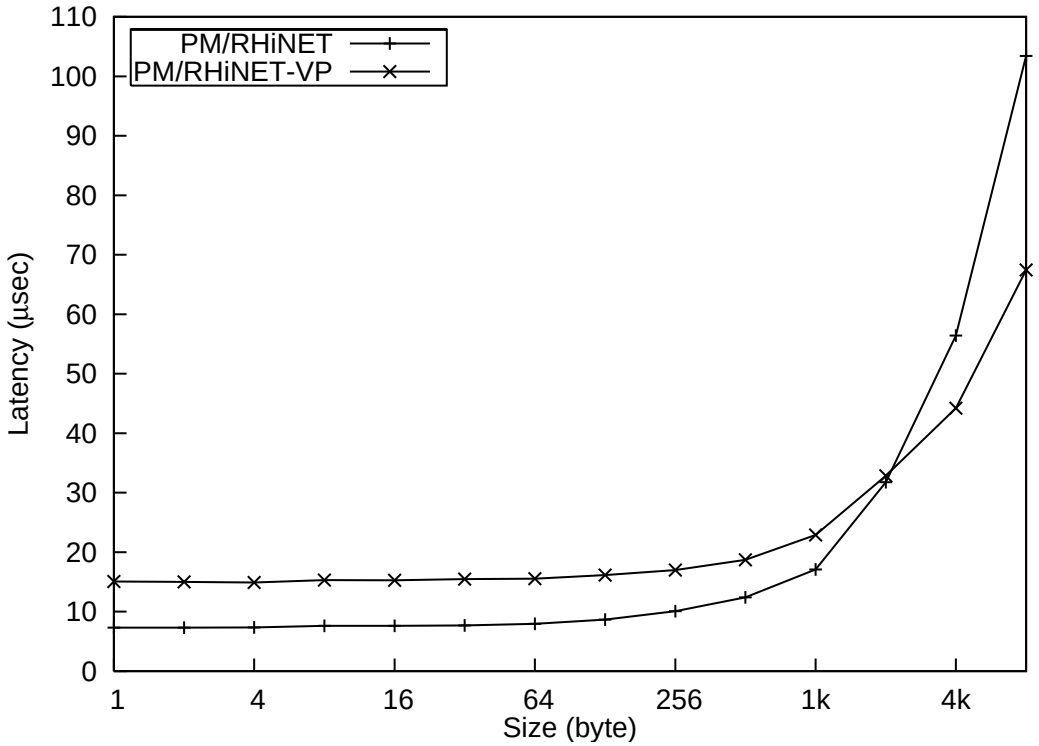



図 8.10　MPI のレイテンシ

図より，メッセージサイズが 2Kbyte 以下では PM/RHiNET を用いた場合の方が PM/RHiNET-VP を用いた場合に比べ低いレイテンシを示し，それ以上の転送サイズでは PM/RHiNET-VP の方が低いレイテンシを示していることがわかる．2Kbyte 以下の場合については図 8.7で示した PM レベルでの RTT と同様の傾向であるが，これは MPICH-SCore が通常の Send および Recv の実装に PM のメッセージ通信をほぼそのまま利用しているためである．一方，メッセージサイズが 2Kbyte 以上の場合，PM/RHiNET では PUSH を 2Kbyte 単位で発行し，次の PUSH を発行する前に毎回 ack パケットの受信を待つ実装となっているため，pmSend の呼び出しが 1 回であったとしても内部でデータサイズを 2Kbyte で割った回数分 ack パケットを待つ処理が加わり，2Kbyte ごとにレイテンシが大幅に増大してしまう．これに対し，PM/RHiNET-VP でも PM 自体の MTU が 2Kbyte に制限されているため，MPI レベルでの 2Kbyte 以上のデータ転送においては PM のメッセージ通信が複数回呼ばれることになるが，VPUSH 自体が ack パケットを発行しない実装となっており，送信側では PUSH 要求を連続して発行できるため，受信側ではパケットを 1 つ受信処理した後，すぐに次のパケットの受信を開始できる．これらより，PM/RHiNET-VP を用いた場合，単純にパケットの受信処理時間に加えて 2Kbyte ごとに VPUSH の受信処理分のレイテンシが増すことになるが，PM/RHiNET を用いた場合，それより大きなパケット往復分のレイテンシが 2Kbyte ごとに加わることから，2Kbyte 以上の転送では PM/RHiNET-VP を用いた場合の方が低レイテンシとなっている．

## 8.6 アプリケーション実行性能

前章で示した MPI レベルの性能が，実アプリケーションに対してどのように影響するかを確認するために，アプリケーションベンチマークを用いた性能評価を行った．評価環境は前章と同一のものを用いた．

評価対象のアプリケーションベンチマークには，MPI を用いて並列化されている NAS 並列ベンチマーク [126] [127] バージョン 2.3 を用い，BT，CG，EP，FT，IS，LU，MG，SP の 8 つのベンチマーク項目について 3 種類の問題サイズ (Class S，Class W，Class A) で評価を行った．各アプリケーションは gcc および g77 バージョン 2.95.3 でコンパイルし，コンパイルオプションは`-O3 -funroll-loops -fomit-frame-pointer -march=i686` (Fortran で書かれたベンチマークではさらに `-malign-double` を追加) とした．

各アプリケーションの性能値 (Mop/s) を図 8.11から図 8.18に示す．なお，安定して動作せず測定が行えなかった項目については，データが欠落している．

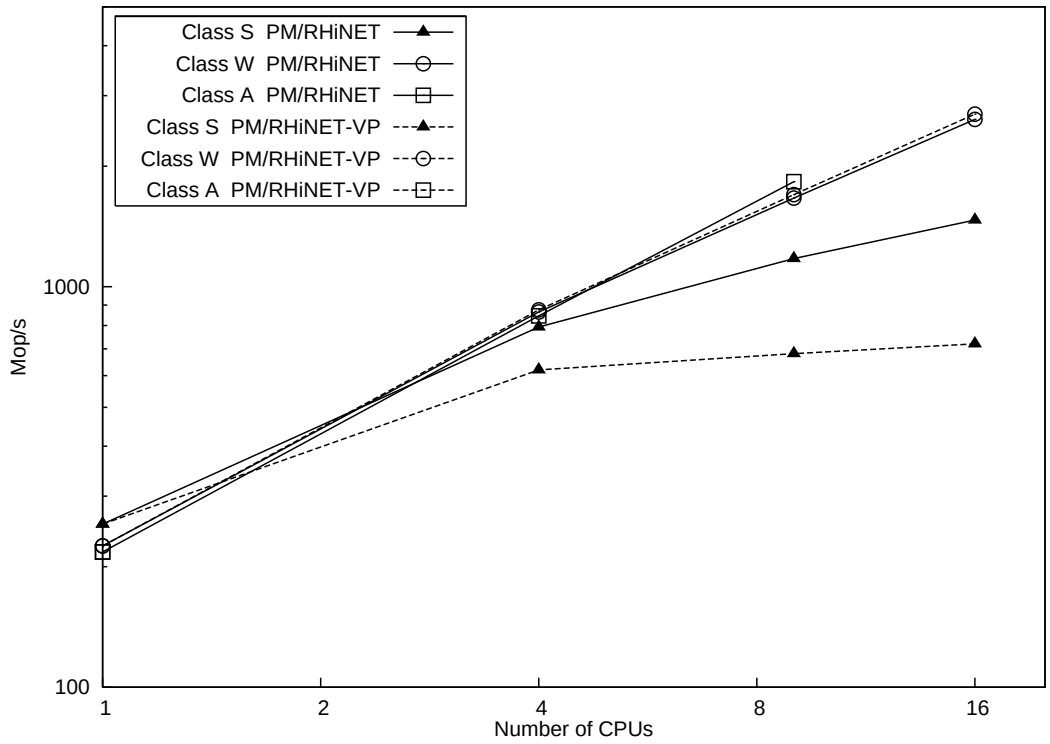



図 8.11 BT の実行結果

計算中に通信を行わないため理想的な性能向上を示す EP を除くと，Class S ではノード数の増加に応じた性能向上が見られないアプリケーションが多く見受けられる．これは，Class S は問題サイズが小さく，並列化による性能向上を得にくいためである．また，PM/RHiNET を用いた場合と PM/RHiNET-VP を用いた場合とで比較すると，いずれのアプリケーションにおいても PM/RHiNET-VP を用いた場合の方が低い性能を示す傾向にあるが，これは問題サイズが小さいことで通信レイテンシの影響が出やすいことに起因している．

一方，問題サイズのより大きな Class W や Class A に着目すると，いずれのベンチマークもノード数の増加に応じた性能向上が見られることがわかる．特に Class A では，PM/RHiNET を用い

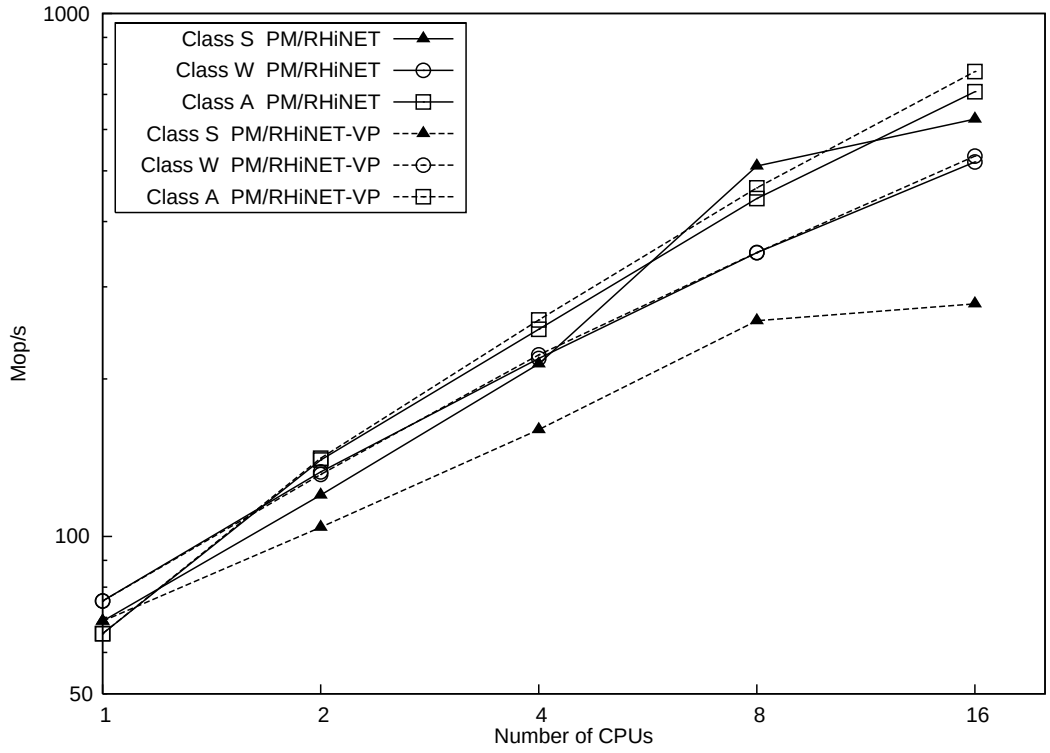



図 8.12　CG の実行結果

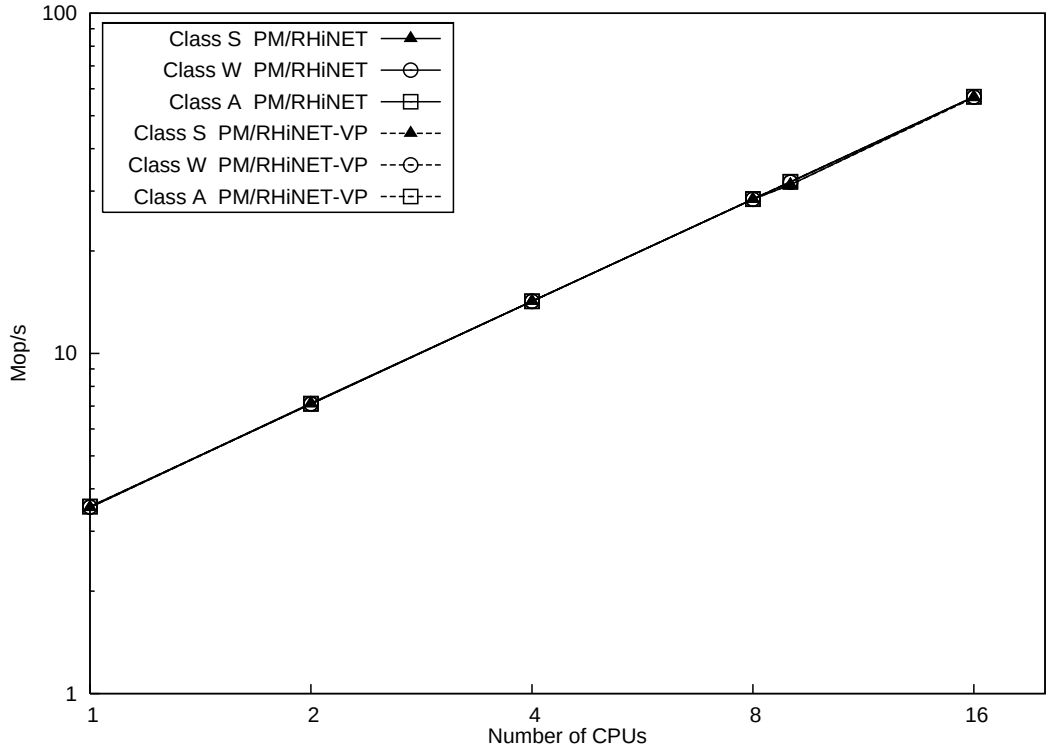



図 8.13　EP の実行結果

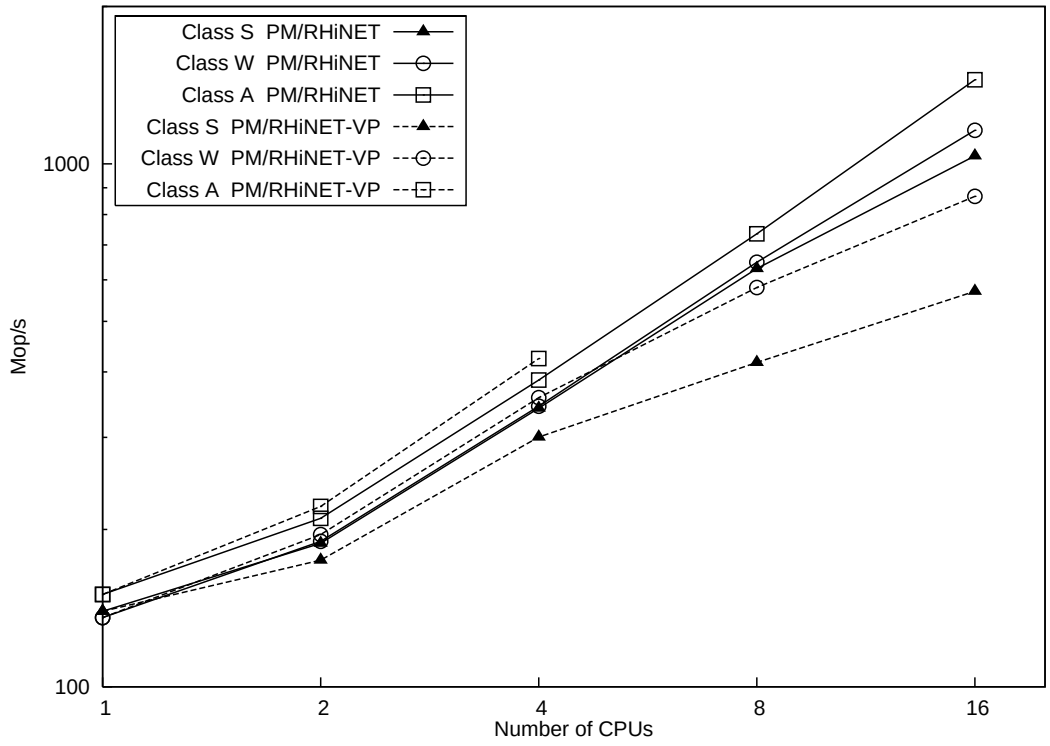



図 8.14 FT の実行結果

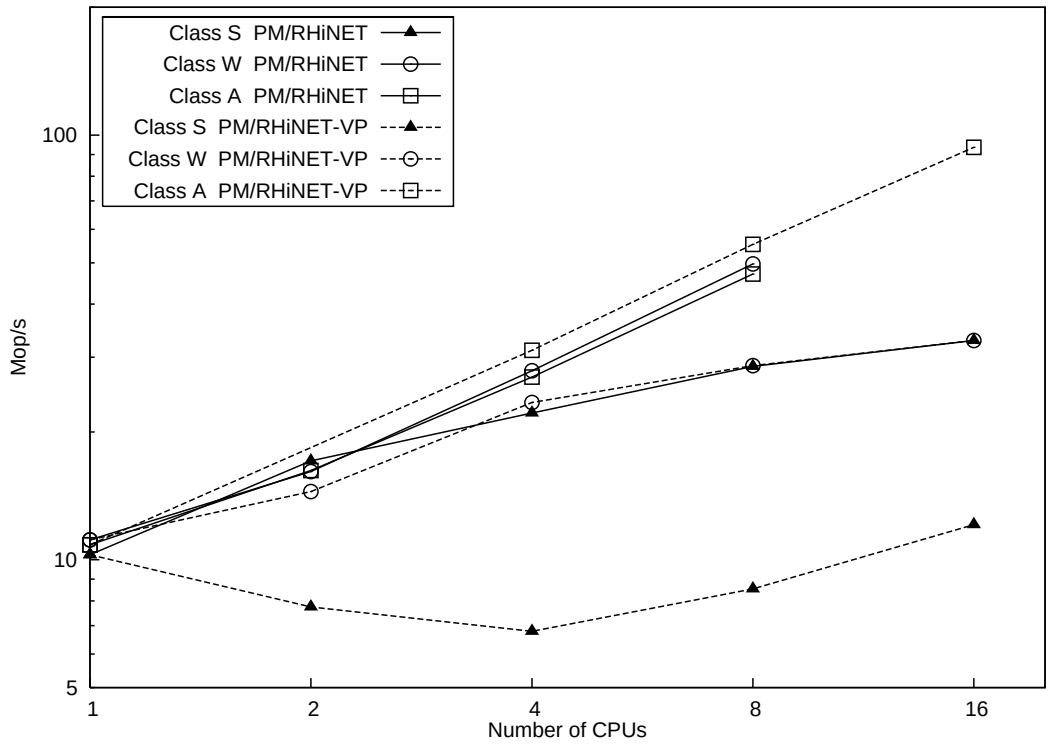



図 8.15 IS の実行結果

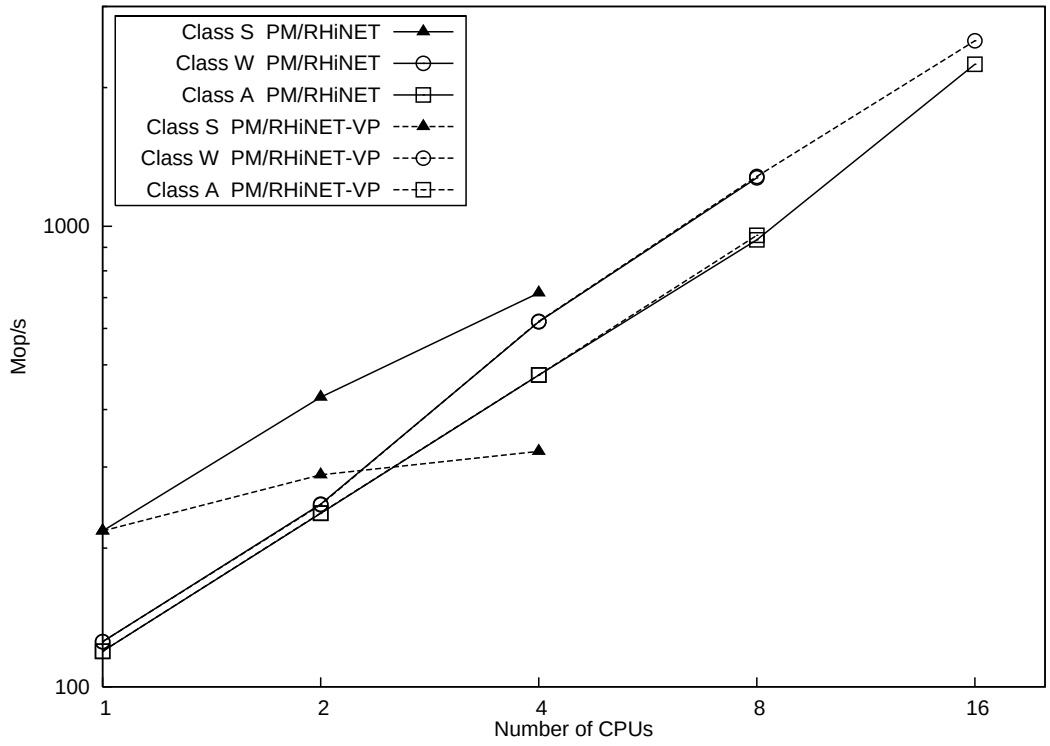



図 8.16　LU の実行結果

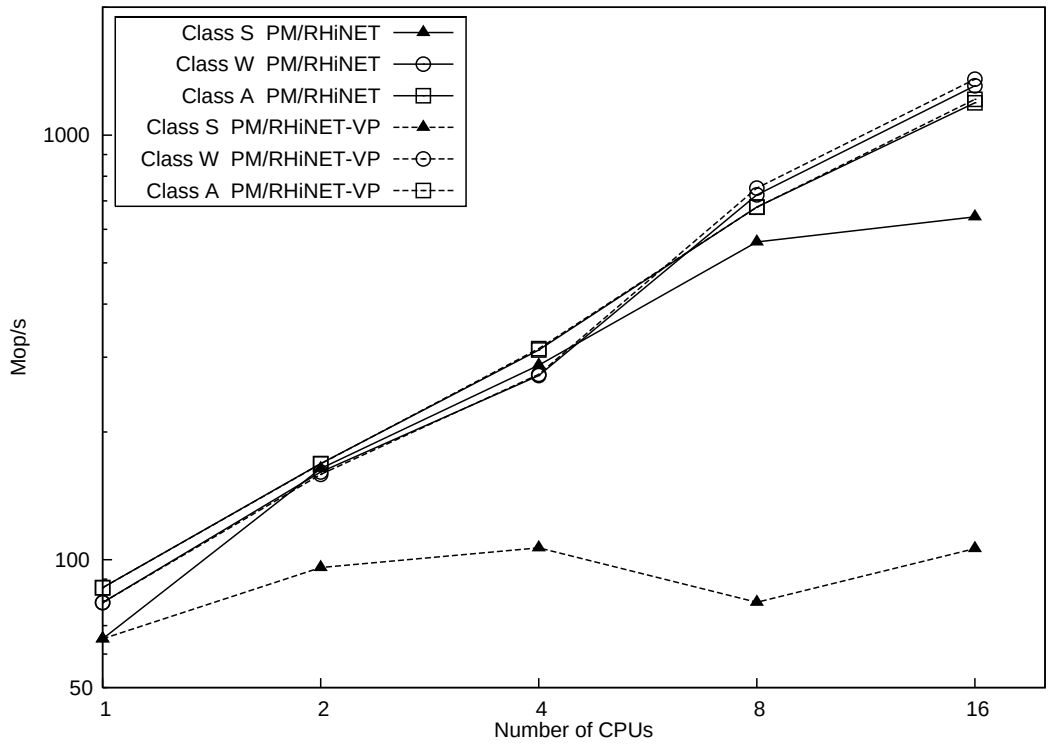



図 8.17　MG の実行結果

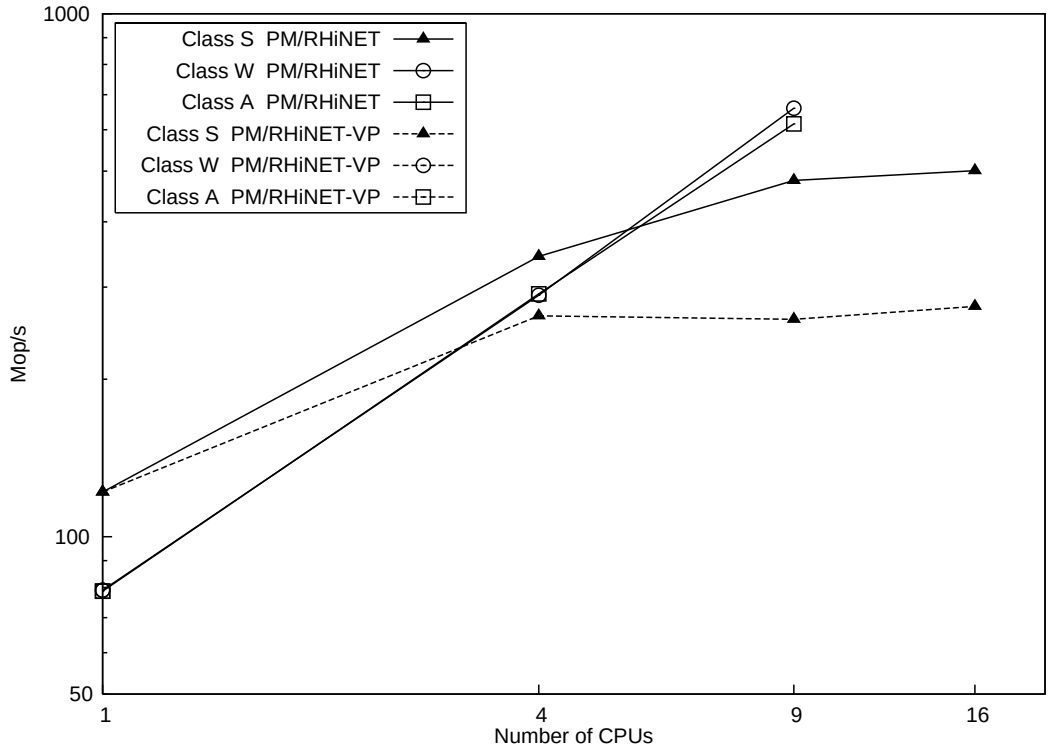



図 8.18 SP の実行結果

た場合と PM/RHiNET-VP を用いた場合とで比較すると，PM/RHiNET-VP の方が PM/RHiNET を上回る性能を示しており，ノード数が増えるにつれ差が広がる傾向にある．中でも IS は，ノード数が少ない場合でも差が顕著である．MPI レベルのベンチマーク結果でも示したように，サイズの大きなデータ転送では PM/RHiNET-VP の方が PM/RHiNET よりも高いスループットを示している．IS では他のベンチマークに比べてサイズの大きなデータをまとめて転送することが多いため，PM/RHiNET よりも高いスループットを提供可能な PM/RHiNET-VP の方が高い性能を実現できていることがわかる．ただし，PM/RHiNET は安定動作のためにスループットを抑制する制限が課されているため，ここでの Class W や Class A のアプリケーションのベンチマーク結果を元に，PM/RHiNET と PM/RHiNET-VP の設計上の優劣を比較することはできない．

## 8.7 Martini におけるメッセージ通信の実装に関する考察

### 8.7.1 メッセージ通信の性能改善案

これまでに示した評価の結果より，PM/RHiNET-VP はシステムの規模に関係なくレイテンシは一定だが値が大きく，一方で PM/RHiNET はシステムが小規模なうちは，レイテンシは非常に小さいが，大規模化するにつれノード数に比例してレイテンシは増大していくことが予想される．

PM/RHiNET のノード数増加時のレイテンシ増加の問題は，5.5.1節で述べた SEND/RECV プリミティブにおけるディスクリプタテーブルのようなバッファを受信バッファと別に用意することで，メッセージ受信確認のためのアクセス範囲を集中させることができるため，ある程度緩和す

ることができると考えられる．しかしながら，この手法では，先に述べたようなレイテンシの増大やパケット数の増加によるネットワークの混雑という問題を伴う他，メッセージ到着の検出にノード数分のメモリアクセスが必要となる問題や，ノード数分の受信領域を確保しなければならないという PM/RHiNET の問題については解決しない．

この問題を根本的に解決するには，Martini のハードウェア機能を強化し，PM/RHiNET-VP における VPUSH の性能を向上させればよい．VPUSH の性能向上には，オンチッププロセッサの処理能力を向上させる方法と，VPUSH そのものをハードウェア実装する方法が考えられる．

VPUSH の受信処理のうち，ハードウェアが待機状態でオンチッププロセッサのソフトウェアのみが処理を行っている時間は約 7.4$\mu$sec であることが RTL シミュレーションより求まっている．これより，オンチッププロセッサの処理能力を向上させ，仮に VPUSH におけるソフトウェア処理時間が半減できたとすると，PM/RHiNET-VP の RTT を 7.4$\mu$sec 小さくできることになる．ただし，この場合でも RTT は 18.1$\mu$sec 程度であり，PM/RHiNET を用いた場合の 2 ノード時の RTT と比べると倍近く大きい．

一方，VPUSH そのものをハードウェア実装するには，受信バッファの空きを判定する処理と，受信のための DMA 要求発行後に受信先のアドレスを更新しつつホストに対して受信通知を行う機構を push パケットの受信処理に追加する必要がある．RHiNET プロジェクトで派生した DIMMnet-1 の後継プロジェクトである DIMMnet-2[128][129] では，VPUSH に似た，より高機能な IPUSH[130] と呼ばれる機構がハードウェア実装されている．IPUSH も通常の RDMA の受信処理を拡張して実現されているが，処理時間の増大は受信領域の空き容量の判定部分で 4 サイクルだけで済んでいる．Martini でも同様に数サイクルの遅延増加で実装可能であると考えられる．

また，受信通知に要する遅延時間は，受信側で push パケットのデータ受信のための DMA 要求が発行されてから実際にホストメモリに書き込まれ，それが CPU から検出されるまでの時間とほぼ等しいものと考えられる．この時間は，実機評価および RTL シミュレーションより，評価で用いた環境では約 1.0$\mu$sec となることがわかっている．この時間はパケットの受信処理時間の増加分に比べて十分に大きい．よって，VPUSH をハードウェア化することで，受信処理時間は通常の push パケットの受信処理時間に比べて多く見積っても約 1.0$\mu$sec の増加で済むと考えられ，PM/RHiNET-VP のレイテンシは 2 ノードでの PM/RHiNET のレイテンシに対して約 1.0$\mu$sec 増となる．PM/RHiNET-VP の RTT が 16 ノードの場合の PM/RHiNET の RTT とほぼ等しくなることから，VPUSH をハードウェア実装した場合，16 ノードより大きな規模であれば性能面で PM/RHiNET を上回ると考えられる．

また，VPUSH をハードウェア実装する場合のハードウェア量については，受信バッファのアドレスやサイズ，受信通知用の領域のアドレスなどを保持するレジスタが数ワード分新たに必要となるが，メッセージ本体の受信処理や受信通知の処理は既存の push パケットの受信処理機構をほぼそのまま利用できるため，増加はわずかであると予想される．Martini において VPUSH がハードウェア実装されていた場合，メッセージ通信のレイテンシは 16 ノード時の PM/RHiNET のレイテンシと同程度となると見積られることから，NAS 並列ベンチマークの Class S の結果は 16 ノードでほぼ同等，2 ノードから 8 ノードまでは図 8.11から図 8.18に示した PM/RHiNET を用いた場合の結果をやや下回る結果となると予想される．また，VPUSH をハードウェア実装することで PM/RHiNET-VP においても乗っ取り機構による実装上の制約がなくなるため，PULL が利用できるようになり，PM で RMA 機構を提供することが可能となる．そのためメッセージサイズが 16Kbyte 以上の際のスループットは RMA を使用することで図 8.9で示した Zerocopy と同様の値を得ることができる．NAS 並列ベンチマークにおいても，問題サイズの大きな Class A を中心に現状の PM/RHiNET-VP を用いた結果よりもさらに高い性能を得られると予想される．

### 8.7.2 メッセージ通信の実装より明らかになった Martini の課題

RHiNET-2 では，ネットワークインタフェースに対して PUSH と PULL という 2 つの単純な RDMA 通信機構のみをハードウェア実装し，これを上位レイヤから利用することで複雑な通信処理においても高い性能を実現できるものと考え Martini が開発された．Martini を搭載したネットワークインタフェースは，RHiNET-2/SW を介した 2 ノード間の通信による評価では，他の SAN のネットワークインタフェースと同程度の高性能な基本通信性能を示した [131] が，実際にシステムを構築して並列分散処理環境を実装してみると，ネットワークインタフェースの提供する機能が不十分なために大規模化した際に通信性能が悪化してしまうことが判明した．このような事態を防ぐために，並列分散処理向けのネットワークインタフェースにおいては，複雑な通信処理は上位レイヤを組み合せて実現する場合であっても，単純な RDMA だけでなく，VPUSH のような単純ながらもメッセージ通信の効率的な実装を支援する通信機能を十分検討しハードウェア実装すべきである．

# 第9章　結論

## 9.1　本研究のまとめ

本研究では，RHiNET 用のネットワークインタフェースコントローラ Martini に関して，そのプロトコル処理部分および Martini の利用に必要となる各種ソフトウェアの設計･実装を行い，Martini の基本機能の評価および Martini を利用した並列分散処理システムである RHiNET-2 システムの構築・評価までを行った．

RHiNET の提供する高い通信性能を活用し，ノード PC 上のプロセス間に高い通信性能を低オーバヘッドで提供するために，Martini はリモートメモリライトおよびリモートメモリリードをハードウェア実装し，OS のメモリ保護を破綻させることなくこれをユーザレベルで直接起動可能な実装となっている．また，Martini には，サイズの小さなパケットを低レイテンシで送出するための AOTF/BOTF 通信機構や，メモリバス装着型ネットワークの有効性を検討するための PC133 メモリバス用を介したホスト PC への接続などの，実験的な試みがいくつか導入されている．

本研究ではこのような Martini において，プロトコル処理部分を中心とした設計・実装およびオンチッププロセッサを用いた論理検証などを行った．その際，乗っ取り機構と呼ばれるオンチッププロセッサと専用ハードウェア部分による協調処理の提案を行い，これを実装した．

また，実機上で Martini を用いたシステムを構築するにあたり必要となった低レベルソフトウェアライブラリの設計および実装を行った．低レベルソフトウェアライブラリは Martini のオンチッププロセッサ上で動作するファームウェアおよびホスト上で動作するデバイスドライバとユーザライブラリで構成される．これらソフトウェアレイヤ上に，Martini がハードウェアで提供するメモリ保護機構を補完する機能，PATLB のミスヒットなどへの対処および Martini がハードウェアで提供しない通信機構プリミティブである SEND/RECV，LOCK/UNLOCK，BARRIER などの実装を行った．

実機を用いた基本性能の評価の結果，Martini は，64bit/66MHz の PCI バスを介してホストに接続した場合に，470Mbyte/s の双方向スループットと 1.74$\mu$sec のレイテンシを提供することがわかった．最新鋭の SAN や Ethernet のネットワークインタフェースと比較した場合，スループットに関しては大きく劣るものの，レイテンシについては同程度の値となっており，AOTF 通信機構のような PCI のトランザクションを最小限に抑えるパケット送出機構やハードウェア実装されたパケットの送受信機構が効果的であることが示された．

また，Martini 開発時に提案・実装を行った乗っ取り機構について，評価を行った．乗っ取り機構の導入により，例外の要因を取り除いた後の処理を，ソフトウェアで行わずにハードウェアによって続行させることで，例外処理の効率化を図れることが確認された．また，乗っ取り機構を利用してソフトウェア処理にハードウェア処理を組み込んだ応用例として VPUSH を実装し，評価を行ったところ，通常の PUSH パケットを処理するためのハードウェアを利用することで，単純にすべてをソフトウェアで処理した場合に比べて倍以上のスループットを実現することが確認された．乗っ取り機構の実装はバスの配線と数%程度のロジックの増大で実現できることから，乗っ

取り機構はネットワークプロセッサのようなシステム LSI において低い実装コストで効果的が得られる協調処理の手段であると言える．

さらに，本研究では Martini を利用した並列分散処理システムである RHiNET-2 システムを構築し，通信ライブラリの性能やアプリケーションレベルでの通信性能の評価を行った．RHiNET-2 システムの構築にあたり，上位の分散オペレーティングシステムやプログラミング環境として高い稼働実績と移植性を持つ SCore を導入することとし，SCore が必要とする低レベル通信ライブラリである PM の RHiNET-2 向けの実装を行った．PM ではメッセージ通信が必要となるが，Martin はこれをハードウェアで提供しないため，ホスト上のソフトウェアから Martini の PUSH・PULL を利用してこれを実現する PM/RHiNET と，VPUSH を利用することでこれを実現する PM/RHNET-VP の 2 種類の PM を実装した．

PM レベルでの基本通信性能を評価した結果，PM/RHiNET は PM で必要となるメッセージ通信を高い性能で実現できることがわかったが，スケーラビリティの面で問題があることが判明した．一方，PM/RHiNET-VP にはスケーラビリティの問題は伴わないが，VPUSH を利用するため，メッセージの受信処理のレイテンシが PM/RHiNET に比べて倍以上大きいことがわかった．

MPI レベルでの基本通信性能の評価においても，同様に PM/RHiNET の方が PM/RHiNET-VP に比べ低いレイテンシを示した．16 ノード構成の並列分散処理システムを構築して行った NAS 並列ベンチマークによるアプリケーションレベルでの性能評価では，問題サイズの大きい Class A や Class W のアプリケーションについては概ね台数に応じた性能向上が見られたが，問題サイズの小さい Class S のアプリケーションでは 4 ノード程度までしか台数効果が得られないことがわかった．特に，問題サイズが小さい場合，メッセージ通信のレイテンシが性能に影響しやすいため，PM/RHiNET を用いた場合と PM/RHiNET-VP を用いた場合で比較すると，PM/RHiNET を用いた場合の方が圧倒的に高い性能を示した．

PM/RHiNET におけるスケーラビリティの問題や PM/RHiNET-VP のレイテンシの問題の解消手段の検討を行った結果，現状の Martini ではこれを解決することは難しく，最も効果的な解決策は VPUSH をハードウェアで実装することであることがわかった．

## 9.2 おわりに

Martini の最大の特徴である PUSH・PULL の完全なハードウェアによる提供は，高い基本通信性能を示し，特にレイテンシについては，プロセスや動作周波数で優る最新鋭の他のネットワークインタフェースコントローラに匹敵する性能を実現した．その一方で，このような低遅延な通信を実際の並列システムにおいて需要の高いメッセージ通信などの上位プロトコルで有効利用することは難しく，上位の通信モデルによってはスケーラビリティ面で問題が生じることから，もし Martini の設計を今後発展させるようなことがあれば，今一度ネットワークインタフェースの提供する機能面の見直しが必要であると考えられる．

Martini に搭載された実験的な機構の多くは，その有効性が示され，またそこから得られた知見を元に次の研究へと発展している．たとえば Martini で得られた知見は，DIMMnet-1 の後継プロジェクトである DIMMnet-2 プロジェクトに活かされている．特に DIMMnet-2 においてメッセージ通信を支援する IPUSH と呼ばれる VPUSH をさらに発展させた通信機能がハードウェア実装されており，その有効性が評価されている．

RHiNET プロジェクトが開始した当時と状況が変わり，今日では，InfiniBand の登場や Ethernet

の高性能化などにより，スイッチを独自に開発しなくとも LASN の結合網が現実的なものとなりつつある．今後，独自のネットワークインタフェースを開発する場合は，このような標準化された結合網を用いるのが合理的であろう．本研究の成果が，今後の並列分散処理環境向けネットワークの発展に貢献できれば幸いである．

# 謝辞

本研究の機会を与えてくださり，なんと 7 年間もの間，絶えず手厚くご指導くださった慶應義塾大学 理工学部 天野 英晴 教授に深く感謝します．今頃になって，ようやく研究らしい研究の進め方がわかってきた気がします．ちょっと遅すぎましたね．なかなか期待に応えることのできない不肖の弟子ですいません．長いことご指導頂き，本当に有難うございました．

本研究をまとめるにあたり，貴重なご意見をくださった慶應義塾大学 理工学部 笹瀬 巌 教授，寺岡 文男 教授，山崎 信行 助教授に深く感謝します．年末年始の慌しい中，丁寧に査読して頂き有難うございました．年賀状に書かれたメッセージは励みになりました．

本研究を進めるにあたり，多大なご助言をくださった産業技術総合研究所 工藤 知宏 氏，株式会社 日立製作所 中央研究所 山本 淳二 氏，慶應義塾大学 理工学部 西 宏章 専任講師，株式会社 東芝 研究開発センター 田邊 昇 氏，株式会社 シナジェテック 清水 敏行 氏をはじめとする新情報処理開発機構関係者の皆様に深く感謝します．色々と納得がいかないバグや仕様に悩まされたこともありましたが，そもそも RHiNET プロジェクトがなければこの研究はなかったわけで，大きなプロジェクトの末席に加えて頂けたことを大変有難く思っています．

Martini のチップ実装でお世話になった日立情報通信エンジニアリング株式会社 今城 英樹 氏，大杉 浩三 氏をはじめとする旧日立 IT 関係者の皆様に深く感謝します．あの 3 度目のチップ実装がなければ，修士論文の時点でまったく別のテーマに取り組むことになっていたかもしれません．

本研究を共に行った日本電気株式会社 土屋 潤一郎 氏，ソニー株式会社 伊豆 直之 氏，慶應義塾大学 大学院 理工学研究科 後期博士課程 大塚 智宏 氏，北村 聡 氏に深く感謝します．いつもいっぱいいっぱいでご迷惑をお掛けしてばかりだった気がします．本当にお世話になりました．

日頃より公私共にお世話になった慶應義塾大学 理工学部 情報工学科 天野研究室の現役生ならびに卒業生の皆様に深く感謝します．いつも散らかしていてすいません．私物はそろそろ撤去します．

何やら上に不自然な空間がありますが，最後に，不規則な生活を繰り返している自分を半ば呆れながらも見捨てずに支えてくれた家族に心より感謝します．本当に有難うございました．

2007 年 2 月 提出期限前日の朝
矢上キャンパス 26-107 にて
渡邊 幸之介

# 参考文献


[1] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, Vol. 45, No. 11, pp. 56–61, 2002.

[2] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, Vol. 15, No. 3, pp. 200–222, 2001.

[3] Nanette J. Boden, Denny Cohen, Robert E. Felderman, Alan E. Kulawik, Charies L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet - A gigabit per second local area network. *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, 1995.

[4] Thomas L. Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP 1995)*, pp. 11–14, Aug. 1995.

[5] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, pp. 85–97, Jun. 1997.

[6] Shinji Sumimoto, Hiroshi Tezuka, Atsushi Hori, Hiroshi Harada, Toshiyuki Takahashi, and Yutaka Ishikawa. High Performance Communication using a Commodity Network for Cluster Systems. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pp. 139–146, Aug. 2000.

[7] Toshiyuki Takahashi, Shinji Sumimoto, Atsushi Hori, Hiroshi Harada, and Yutaka Ishikawa. PM2: High Performance Communication Middleware for Heterogeneous Network Environment. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pp. 52–53, Nov. 2000.

[8] Giuseppe Ciaccio. Messaging on Gigabit Ethernet: Some experiments with GAMMA and other Systems. In *Proceedings of the International Parallel and Distributed Processinig Symposium*, pp. 1624–1631, Apr. 2001.

[9] Tomohiro Kudoh, Shinji Nishimura, Junji Yamamoto, Hiroaki Nishi, Osamu Tatebe, and Hideharu Amano. RHiNET: A network for high performance parallel processing using locally distributed computers. In *Proceedings of the 1999 International Workshop on Innovative Architecture (IWIA99)*, pp. 69–73, Nov. 1999.

[10] Tomohiro Kudoh, Noboru Tanabe, Junji Yamamoto, and Hiroaki Nishi. RHiNET: A network for high performance parallel computing using locally distributed computers. Technical report, RWC Technical Report TR-99-002, Jan. 2000.

[11] 佐藤三久, 石川裕, 工藤知宏, 島田潤一. 光インタコネクションネットワークを用いたコンピューテングクラスタの構想. 情報処理学会研究報告 ARC-122, Feb. 1997.

[12] 西宏章, 藤知宏, 天野英晴. 軽量メモリベース通信用ネットワークルータ. 電子情報通信学会技術研究報告 CPSY98-1, pp. 1–8, Apr. 1998.

[13] 周東福強, 山本淳二, 西宏章, 天野英晴, 工藤知宏. 軽量メモリベース通信のためのネットワークインタフェース. 情報処理学会研究報告 1997-ARC-128, pp. 103–108, Mar. 1998.

[14] 西宏章, 多昌廣治, 天野英晴, 工藤知宏. コモディティPC を用いた並列処理のためのネットワークルータアーキテクチャ. 電子情報通信学会技術研究報告 CPSY98-158, pp. 57–64, Jan. 1999.

[15] 横山知典. MLC-1 システムにおけるネットワークインタフェースの実装. 卒業論文, 慶應義塾大学理工学部, 1999.

[16] 山本淳二, 工藤知宏, 宮脇達朗, 坂光彦, 清水敏行, 横山知典, 天野英晴. コモディティPC を用いた並列処理のための通信機構について. 情報処理学会研究報告 1998-ARC-132, pp. 115–120, Mar. 1999.

[17] 西宏章, 多昌廣治, 西村信治, 天野英晴, 工藤知宏. 64Gbps のスループットを持つワンチップネットワークスイッチ RHiNET-2/SW. 電子情報通信学会技術研究報告 CPSY99-76, pp. 25–32, Oct. 1999.

[18] 西宏章, 上野龍一郎, 多昌廣治, 稲沢悟, 西村信治, 工藤知宏, 天野英晴. LASN 用 10Gbps/port 8x8 ネットワークスイッチ: RHiNET-3/SW. 情報処理学会研究報告 2000-ARC-140, pp. 13–18, Nov. 2000.

[19] Hiroaki Nishi, Junji Yamamoto, Kozo Ohsugi, Katsuyoshi Harasawa, and Shinji Nishimura. Deskew-LSI for 10-Gbit/s parallel optical links in RHiNET-3 system. In *COOL Chips V An International Symposium on Low-Power and High-Speed Chips Vol.I Proceedings*, pp. 37–46, Apr. 2002.

[20] Naoyuki Izu, Tomonori Yokoyama, Junichiro Tsuchiya, Konosuke Watanabe, and Hideharu Amano. RHiNET/NI: A reconfigurable network interface for cluster computing. In *12th International Conference on Field Programmable Logic and Application*, Sep. 2002.

[21] Yutaka Ishikawa, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Toshiyuki Takahashi, Francis O'Carroll, and Hiroshi Harada. RWC PC Cluster II and SCore Cluster System Software – High Performance Linux Cluster. In *Proceedings of the 5th Annual Linux Expo*, pp. 55–62, May 1999.

[22] PC クラスタコンソーシアム. `http://www.pccluster.org/`.

[23] Michihiro Koibuchi, Konosuke Watanabe, Kenichi Kono, Akiya Jouraku, and Hideharu Amano. Performance Evaluation of Routing Algorithms in RHiNET-2 Cluster. In *Proceedings of IEEE International Conference on Cluster Computing*, pp. 395–402, Dec. 2003.

[24] Michihiro Koibuchi, Konosuke Watanabe, Tomohiro Otsuka, and Hideharu Amano. Performance Evaluation of Deterministic Routings, Multicasts, and Topologies on RHiNET-2 Cluster. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 16, No. 8, pp. 747–759, Aug. 2005.

[25] 大門優, 松尾亜紀子, 大塚智宏, 渡邊幸之介, 天野英晴. 反応を伴った圧縮性流体計算による RHiNET-2 の評価. 電子情報通信学会技術研究報告 CPSY2003-22, pp. 19–24, Aug. 2003.

[26] 大塚智宏, 渡邊幸之介, 北村聡, 鯉渕道紘, 山本淳二, 西宏章, 工藤知宏, 天野英晴. RHiNET プロジェクトの最終報告. 情報処理学会研究報告 2004-ARC-158, pp. 31–36, May 2004.

[27] Konosuke Watanabe, Hideharu Amano, Junji Yamamoto, Jun ichiro Tsuchiya, and Tomohiro Kudoh. Taking over mechanism: a cooperation methodology of Hardware and Software in Network Controllers. In *International Workshop on Synthesis and System Integration of Mixed Information Techonologies (SASIMI2003)*, pp. 386–393, Apr. 2003.

[28] 渡邊幸之介, 大塚智宏, 天野英晴. ネットワークインタフェース用コントローラチップ Martini における乗っ取り機構の実装と評価. 情報処理学会論文誌コンピューティングシステム, Vol. 45, No. SIG11 (ACS7), pp. 393–407, Oct. 2004.

[29] 渡邊幸之介. RHiNET-2 システムの実装と評価. 修士論文, 慶應義塾大学大学院理工学研究科開放環境科学専攻, 2003.

[30] Konosuke Watanabe, Tomohiro Otsuka, Junichiro Tsuchiya, Hiroaki Nishi, Junji Yamamoto, Noboru Tanabe, Tomohiro Kudoh, and Hideharu Amano. Martini: A Network Interface Controller Chip for High Performance Computing with Distributed PCs. *IEEE Transactions on Parallel and Distributed Systems*, 2007. 掲載予定.

[31] Paul J. Schweitzer Philip M. Merlin. Deadlock Avoidance in Store-and-Forward Networks. *IEEE Transaction on Communications*, Vol. COM-28, No. 3, pp. 345–354, Mar. 1980.

[32] 西宏章, 多昌廣治, 工藤知宏, 天野英晴. 仮想チャネルキャッシュを持つルータの構成と性能. 並列処理シンポジウム JSPP'99 論文集, pp. 71–78, Jun. 1999.

[33] Brent N. Chun, Alan M. Mainwaring, and David E. Culler. Virtual Network Transport Protocols for Myrinet. *IEEE Micro*, Vol. 18, No. 1, 1998.

[34] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 40–53, Dec. 1995.

[35] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, p. 55.1, Dec. 1995.

[36] Jonathan M. Smith and C. Brendan S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, Vol. 7, No. 4, pp. 44–52, Jul. 1993.

[37] Christina Amza, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–271, Feb. 1997.

[38] 西宏章, 工藤知宏, 天野英晴. 軽量メモリベース通信用ネットワークルータ. 電子情報通信学会技術研究報告 CPSY98-1, pp. 1–8, Apr. 1998.

[39] 土屋潤一郎. RHiNET-1 におけるネットワークインタフェースの設計と実装. 卒業論文, 慶應義塾大学理工学部, 2000.

[40] Shinji Nishimura, Tomohiro Kudoh, Hiroaki Nishi, Junji Yamamoto, Katsuyoshi Harasawa, Nobuhiro Matsudaira, Shigeto Akutsu, Koji Tasho, and Hideharu Amano. RHiNET-3/SW: an 80-Gbit/s high-speed network switch for distributed parallel computing. In *Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, pp. 119–123, Aug. 2001.

[41] 西宏章, 多昌廣治, 稲沢悟, 西村信治, 工藤知宏, 天野英晴. RHiNET スイッチ RHiNET-2,3/SW. 情報処理学会研究報告 2001-ARC-144, pp. 55–60, Jul. 2001.

[42] Tomonori Yokoyama, Naoyuki Izu, Jun ichiro Tsuchiya, Konosuke Watanabe, Hideharu Amano, and Tomohiro Kudoh. Design and implementation of RHiNET-2/NI0: a reconfigurable network interface for cluster computing. *IEICE Transaction*, Vol. E86-D, No. 5, pp. 789–795, 2003.

[43] Noboru Tanabe, Yoshihiro Hamada, Hironori Nakajo, Hideki Imashiro, Junji Yamamoto, Tomohiro Kudoh, and Hideharu Amano. Low latency communication on DIMMnet-1 network interface plugged into a DIMM slot. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, pp. 9–14, Sep, 2002.

[44] Noboru Tanabe, Junji Yamamoto, Hiroaki Nishi, Tomohiro Kudoh, Yoshihiro Hamada, Hironori Nakajo, and Hideharu Amano. On-the-fly sending: a low latency high bandwidth message transfer mechanism. In *Proceedings of the 2000 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '00)*, pp. 186–193, Dec. 2000.

[45] Takashi Yoshikawa and Hiroshi Matsuoka. Optical Interconnections for Parallel and Distributed Computing. *Proceedings of the IEEE*, Vol. 88, No. 6, pp. 849–855, Jun. 2000.

[46] Takashi Yoshikawa, Ichiro Hatakeyama, KazunoriMiyoshi, Kazuhiko Kurata, Junichi Sasaki, Nobuharu Kami, Takara Sugimoto, Muneo Fukaishi, Kazuuki Nakamura, Kei Tanaka, Hiroaki Nishi, and Tomohiro Kudoh. Optical-interconnection as an IP macro of a CMOS Library. In *Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, pp. 31–35, Aug. 2001.

[47] Myricom, Inc. `http://www.myri.com/`.

[48] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, and Wen-King Su. The Design of the Caltech Mosaic C Multicomputer. In *Proceeding of the 1993 symposium on Research on integrated systems*, pp. 1–22, 1993.

[49] Robert Felderman, Annette DeSchon, Danny Cohen, and Gregory Finn. ATOMIC: A High-Speed Local Communication Architecture. *Journal of High Speed Networks*, Vol. 3, No. 1, pp. 1–29, 1994.

[50] Myricom, Inc. *LANai 9*, June 2000.

[51] Myricom, Inc. *LANai 7*, June 1999.

[52] Myricom, Inc. *Lanai X*, rev. 1.1 edition, July 2003.

[53] Myricom, Inc. *The GM Message Passing System*, Jul. 2000.

[54] Myricom, Inc. *GM Reference Manual 1.6.3*, Nov. 2002.

[55] Myricom, Inc. *Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet*, version 1.0 edition, Jul. 2005.

[56] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, pp. 708–717, Apr. 1997.

[57] Raoul A.F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Micro*, Vol. 31, No. 11, pp. 53–60, 1998.

[58] Myricom, Inc. Low-latency 10-gigabit ethernet, Oct. 2006.

[59] Quadrics, Inc. `http://www.quadrics.com/`.

[60] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Proceedings of the The Ninth Symposium on High Performance Interconnects (HOTI '01)*, pp. 125–130, Aug. 2001.

[61] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, Vol. 22, No. 1, pp. 46–57, 2002.

[62] Fabrizio Petrini, Juan Fernandez, Eitan Frachtenberg, and Salvador Coll. Scalable Collective Communication on the ASCI Q Machine. In *Proceedings of the 11th Symposium on High Performance Interconnects*, pp. 54–59, Aug. 2003.

[63] Quadrics Ltd. *Elan Programming Manual*, Apr. 2005.

[64] InfiniBand Trade Association. `http://www.infinibandta.org/`.

[65] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.0*, Oct. 2000.

[66] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.2*, Oct. 2004.

[67] Mellanox Technologies, Inc. http://www.mellanox.com/.

[68] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, Vol. 18, No. 2, pp. 66–76, 1998.

[69] Mellanox Technologies, Inc. *Mellanox IB-Verbs API (VAPI)*, 2001.

[70] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. *International Journal of Parallel Programming*, Vol. 32, No. 3, pp. 167–198, Jun. 2004.

[71] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Q. Gao, and Dhabaleswar K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06) - Volume 00*, pp. 43–48, May 2006.

[72] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, Vol. 22, No. 6, pp. 789–828, Sep. 1996.

[73] Chelsio Communications. http://www.chelsio.com/.

[74] Neterion, Inc. http://www.neterion.com/.

[75] RDMA Consortium. Architectural Specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org.

[76] NetEffect. http://www.neteffect.com/.

[77] Pavan Balaji, Wu chun Feng, and Dhabaleswar K. Panda. Bridging the Ethernet-Ethernot Performance Gap. *IEEE Micro*, Vol. 26, No. 3, pp. 24–40, 2006.

[78] Tomohiro Kudoh, Hiroshi Tezuka, Motohiko Matsuda, Yuetsu Kodama, Osamu Tatebe, and Satoshi Sekiguchi. VLAN-based Routing: Multi-path L2 Ethernet Network for HPC Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster2004)*, Sep. 2004.

[79] Shin'ichi Miura, Takayuki Okamoto, Taisuke Boku, Mitsuhisa Sato, and Daisuke Takahashi. Low-cost High-bandwidth Tree Network for PC Clusters based on Tagged-VLAN Technology. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN 2005)*, pp. 84–93, Dec. 2005.

[80] Tomohiro Otsuka, Michihiro Koibuchi, Tomohiro Kudoh, and Hideharu Amano. A Switch-tagged VLAN Routing Methodology for PC Clusters with Ethernet. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP-06)*, pp. 479–486, Aug. 2006.

[81] Jenwei Hsieh, Tau Leng, Victor Mashayekhi, and Reza Rooholamini. Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, pp. 53–54, Nov. 2000.

[82] Robert W. Horst. TNet: A Reliable System Area Network. *IEEE Micro*, Vol. 15, No. 1, pp. 37–45, 1995.

[83] William E. Baker, Robert W. Horst, David P. Sonnier, and William J. Watson. A flexible ServerNet-based fault-tolerant architecture. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pp. 2–11, Jun. 1995.

[84] Davib B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, Vol. 12, No. 1, pp. 10–22, 1992.

[85] Dolphin Interconnect Solutions, Inc. `http://www.dolphinics.com/`.

[86] Richard B. Gillett. Memory Channel Network for PCI. *IEEE Micro*, Vol. 16, No. 1, pp. 12–18, 1996.

[87] Richard Gillett and Richard Kaufmann. Using the Memory Channel Network. *IEEE Micro*, Vol. 17, No. 1, pp. 19–25, 1997.

[88] TOP500.Org. Top 500 Supercomputer Sites. `http://www.top500.org/`.

[89] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. International Conference on Computer Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 256–266, May 1992.

[90] Richard P. Martin. HPAM: an active message layer for a network of hp workstations. In *Proceedings of Hot Interconnects II*, pp. 40–58, Aug. 1994.

[91] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, Vol. 15, No. 1, pp. 46–53, 1995.

[92] Alan Mainwaring and David Culler. Active Message Applications Programming Interface. Technical report, University of California at Berkeley, Oct. 1996.

[93] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, Vol. 5, No. 2, pp. 60–72, 1997.

[94] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, pp. 10–20, Jul. 1998.

[95] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Low-Latency Communication over Fast Ethernet. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, Vol. I, pp. 187–194, Aug. 1996.

[96] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of the 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997.

[97] Cezary Dubnicki, Liviu Iftode, Edward Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 1996 International Parallel Processing Symposium*, pp. 372–381, Apr. 1996.

[98] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pp. 142–153, Apr. 1994.

[99] Edward W. Felten, Richard D. Alpert, Angelos Bilas, Matthias A. Blumrich, Douglas W. Clark, and Stefanos Damianakis. Early Experience with Message-Passing on the SHRIMP Multicomputer. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 296–307, May 1996.

[100] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi, and Robert A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 330–341, Jun. 1998.

[101] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and implementation of virtual memory-mapped communication on Myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, pp. 388–396, Apr. 1997.

[102] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos Damianakis, and Kai Li. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Proceedings of the 1997 IEEE Symposium on High Performance Interconnects (HOT Interconnects V)*, Aug. 1997.

[103] Loic Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98), IPPS/SPDP'98*, pp. 472–485, Apr. 1998.

[104] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, pp. 308–314, Apr. 1998.

[105] Shinji Sumimoto, Hiroshi Tezuka, Atsushi Hori, Hiroshi Harada, Toshiyuki Takahashi, and Yutaka Ishikawa. The Design and Evaluation of High Performance Communication using a Gigabit Ethernet. In *Proceedings of the 13th international conference on Supercomputing*, pp. 243–250, Jun. 1999.

[106] 住元真司, 堀敦史, 手塚宏史, 原田浩, 高橋俊行, 石川裕. GigaE PM II : Gigabit Ethernet による高速通信ライブラリの設計. 情報処理学会研究報告 1999-ARC-134, pp. 61–66, Aug. 1999.

[107] 住元真司, 成瀬彰, 久門耕一, 細江広治, 清水俊幸. PM/InfiniBand-FJ: InfiniBand を用いた大規模 PC クラスタ向け高性能通信機構の設計. 先進的計算基盤システムシンポジウム SACSIS2004 論文集, pp. 373–380, May 2004.

[108] Philip Buonadonna, Andrew Geweke, and David Culler. An implementation and analysis of the virtual interface architecture. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, Nov. 1998.

[109] Patrick Bozeman and Bill Saphir. A Modular High Performance Implementation of the Virtual Interface Architecture. In *Proceedings of the 1999 USENIX Annual Technical Conference, Extreme Linux Workshop #2*, Jun. 1999.

[110] Giovanni Chiola and Giuseppe Ciaccio. GAMMA: Architecture, Programming Interface and Preliminary Benchmarking. Technical report, Technical Report DISI-TR-96-22, Nov. 1996.

[111] Giuseppe Ciaccio. Optimal Communication Performance on Fast Ethernet with GAMMA. In *Proceedings of the Workshop PC-NOW, IPPS/SPDP'98*, pp. 534–548, Apr. 1998.

[112] Giovanni Chiola and Giuseppe Ciaccio. Active Ports: A Performance-oriented Operating System Support to Fast LAN Communications. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par'98)*, pp. 620–624, Sep. 1998.

[113] Giovanni Chiola, Giuseppe Ciaccio, Luigi V. Mancini, and Pierluigi Rotondo. GAMMA on DEC 2114x with Efficient Flow Control. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, Vol. 1, pp. 2337–2343, Jun. 1999.

[114] 渡邊幸之介, 土屋潤一郎, 天野英晴. ミスアラインメントを補正するモジュール間転送用インタフェース. 電子情報通信学会技術研究報告 VLD2002-117, Nov. 2002.

[115] Hitachi, Ltd. Micro Device Division. `http://www.hitachi.co.jp/Div/mdd/english/index.html`.

[116] 山本淳二, 渡邊幸之介, 宮脇達朗, 西宏章, 工藤知宏, 天野英晴. PLI を用いたネットワークインタフェースコントローラとホストプログラムの協調シミュレーション. 情報処理学会研究報告 2001-ARC-145, pp. 73–78, Nov. 2001.

[117] Jon Beecroft, David Addison, David Hewson, Moray McLaren, Duncan Roweth, Fabrizio Petrini, and Jarek Nieplocha. QsNetII: Defining High-Performance Network Design. *IEEE Micro*, Vol. 25, No. 4, pp. 34–47, 2005.

[118] Myricom, Inc. Myricom Custom-VLSI Chips. `http://www.myri.com/vlsi/`.

[119] Myricom, Inc. Myrinet Peformance. `http://www.myri.com/myrinet/performance/`.

[120] Jiuxing Liu, Amith Mamidala, Abhinav Vishnu, and Dhabaleswar K. Panda. Evaluating InfiniBand Performance with PCI Express. *IEEE Micro*, Vol. 25, No. 1, pp. 20–29, 2005.

[121] Wu chun Feng, Pavan Balaji, Chris Baron, Laxmi N. Bhuyan, and Dhabaleswar K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the 13th Symposium on High Performance Interconnects*, pp. 58–63, Aug. 2005.

[122] Yuan Xie, Hua Lin, Zhao Wu, and W Wolf. CAD Techniques for Multimedia System Design. In *Proceedings of the Ninth Workshop on Synthesis and System Integration of MIxed Technologies (SASIMI 2000)*, pp. 81–87, Apr. 2000.

[123] Yuan Xie and Wayne Wolf. Co-synthesis with custom ASICs. In *Proceedings of the 2000 conference on Asia South Pacific design automatio(ASP-DAC 2000)*, pp. 129–133, Jan. 2000.

[124] Hiroshi Harada, Yutaka Ishikawa, Atsushi Hori, Hiroshi Tezuka, Shinji Sumimoto, and Toshiyuki Takahashi. Dynamic home node reallocation on software distributed shared memory. In *Proceedings of the Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region (HPC-ASIA 2000)*, pp. 158–163, May 2000.

[125] Pallas GmbH. Pallas MPI Benchmarks. `http://www.pallas.com/e/products/pmb/`.

[126] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical report, NAS Technical Report NAS-95-020, 1995.

[127] William Saphir, Rob F. Van der Wijngaart, Alex Woo, and Maurice Yarrow. New Implementations and Results for the NAS Parallel Benchmarks 2. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997*, Mar. 1997.

[128] 田邊昇, 濱田芳博, 三橋彰浩, 中條拓伯, 天野英晴. メモリスロット装着型ネットワークインタフェース DIMMnet-2 の構想. 情報処理学会研究報告 2002-ARC-152, pp. 61–66, Mar. 2003.

[129] 北村聡, 濱田芳博, 宮部保雄, 伊澤徹, 宮代具隆, 田邊昇, 中條拓伯, 天野英晴. DIMMnet-2 ネットワークインタフェースコントローラの設計と実装. 情報処理学会論文誌コンピューティングシステム, Vol. 46, No. SIG12 (ACS11), pp. 13–26, May. 2005.

[130] 北村聡, 宮部保雄, 中條拓伯, 田邊昇, 天野英晴. メッセージパッシングモデルを支援するパケット受信機構の DIMMnet-2 への実装と評価. 情報処理学会論文誌コンピューティングシステム, Vol. 47, No. SIG12 (ACS15), pp. 59–73, Sep. 2006.

[131] Konosuke Watanabe, Tomohiro Otsuka, Jun ichiro Tsuchiya, Tomohiro Kudoh, and Hideharu Amano. Performance Evaluation of RHiNET-2/NI: A Network Interface for Distributed Parallel Computing Systems. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pp. 318–325, May 2003.

# 論文目録

## 本研究に関する論文

### 公刊論文


1. 山本 淳二, 渡邊 幸之介, 土屋 潤一郎, 原田 浩, 今城 英樹, 寺川 博昭, 西 宏章, 田邊 昇, 上嶋 利明, 工藤 知宏, 天野 英晴, “高性能計算をサポートするネットワークインタフェース用コントローラチップ Martini”, 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム Vol.43 No.SIG6 (HPS-5-018), pp.122–133, Sep. 2002

2. 渡邊 幸之介, 大塚 智宏, 天野 英晴, “ネットワークインタフェース用コントローラチップ Martini における乗っ取り機構の実装と評価”, 情報処理学会論文誌コンピューティングシステム vol.45 SIG11(ACS7), pp.393–407, Oct. 2004

3. 鯉渕 道紘, 渡邊 幸之介, 大塚 智宏, 天野 英晴, “RHiNET-2 クラスタを用いたシステムエリアネットワーク向け トポロジの実機評価”, 情報処理学会論文誌コンピューティングシステム Vol.45 SIG11(ACS 7), pp.420–431, Oct. 2004

4. 鯉渕 道紘, 渡邊 幸之介, 大塚 智宏, 上樂 明也, 天野 英晴, “RHiNET-2 クラスタを用いたデッドロックフリー 固定ルーティングの実機評価”, 情報処理学会論文誌コンピューティングシステム Vol.45 SIG11(ACS 7), pp.432–444, Oct. 2004

5. 鯉渕 道紘, 大塚 智宏, 渡邊 幸之介, 天野 英晴, “RHiNET-2 クラスタにおけるユニキャストを基にしたマルチキャストアルゴリズムの評価”, 電子情報通信学会論文誌 D-I VOL.J88-D-I No.4, pp.791–799, Apr. 2005

6. Michihiro Koibuchi, Konosuke Watanabe, Tomohiro Otsuka, Hideharu Amano, “Performance Evaluation of Deterministic Routings, Multicasts, and Topologies on RHiNET-2 Cluster”, IEEE Transactions on Parallel and Distributed Systems vol.16 no.8, pp.747–759, Aug. 2005

7. Konosuke Watanabe, Tomohiro Otsuka, Junichiro Tsuchiya, Hiroaki Nishi, Junji Yamamoto, Noboru Tanabe, Tomohiro Kudoh, Hideharu Amano, “Martini: A Network Interface Controller Chip for High Performance Computing with Distributed PCs”, IEEE Transactions on Parallel and Distributed System (掲載予定)

8. 渡邊 幸之介, 大塚 智宏, 天野 英晴, “並列分散処理環境 RHiNET-2 システム の実装と評価”, 電子情報通信学会論文誌 (掲載予定)

## 国際会議

1. Konosuke Watanabe, Junji Yamamoto, Jun-ichiro Tsuchiya, Noboru Tanabe, Hiroaki Nishi, Tomohiro Kudoh, Hideharu Amano, “Preliminary Evaluation of Martini: a Novel Network Interface Controller Chip for Cluster-based Parallel Processing”, Proceedings of the IASTED International Multi-Conference on Applied Informatics (AI2002), pp.390–395, Feb. 2002

2. Tomohiro Otsuka, Konosuke Watanabe, Jun-ichiro Tsuchiya, Hiroshi Harada, Junji Yamamoto, Hiroaki Nishi, Tomohiro Kudoh, Hideharu Amano, “Performance Evaluation of a Prototype of RHiNET-2: A Network-based Distributed Parallel Computing System”, Proceedings of the IASTED International Multi-Conference on Applied Informatics (AI2003), pp.738–743, Jan. 2003

3. Konosuke Watanabe, Hideharu Amano, Junji Yamamoto, Jun-ichiro Tsuchiya, Tomohiro Otsuka, Tomohiro Kudoh, “Taking over mechanism: a Cooperation Methodology of Hardware and Software in Network Controllers”, Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2003), pp.386–393, Apr. 2003

4. Konosuke Watanabe, Tomohiro Otsuka, Jun-ichiro Tsuchiya, Hiroshi Harada, Junji Yamamoto, Hiroaki Nishi, Tomohiro Kudoh, Hideharu Amano, “Performance Evaluation of RHiNET-2/NI: A Network Interface for Distributed Parallel Computing Systems”, Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), pp.318–325, May. 2003

5. Michihiro Koibuchi, Akiya Jouraku, Konosuke Watanabe, Hideharu Amano, “Descending Layers Routing: A Deadlock-Free Deterministic Routing using Virtual Channels in System Area Networks with Irregular Topologies”, Proceedings of the International Conference on Parallel Processing (ICPP’03), pp.527–536, Oct. 2003

6. Akira Kitamura, Konosuke Watanabe, Tomohiro Otsuka, Hideharu Amano, “The evaluation of dynamic load balancing algorithm on RHiNET-2”, Parallel and Distributed Computing and Systems (PDCS 2003), pp.262–267, Nov. 2003

7. Michihiro Koibuchi, Konosuke Watanabe, Kenichi Kono, Akiya Jouraku, Hideharu Amano, “Performance Evaluation of Routing Algorithms in RHiNET-2 Cluster”, Proceedings of IEEE International Conference on Cluster Computing, pp.395–402, Dec. 2003

## 研究会ほか

1. 山本 淳二, 田邊 昇, 西 宏章, 土屋 潤一郎, 渡邊 幸之介, 今城 英樹, 上嶋 利明, 金野 英俊, 寺川 博昭, 慶光院 利映, 工藤 知宏, 天野 英晴, “高速性と柔軟性を併せ持つネットワークインタフェース用チップ: Martini”, 情報処理学会研究報告 2000-ARC-140, pp.19–24, Nov. 2000

2. 山本 淳二, 土屋 潤一郎, 寺川 博昭, 田邊 昇, 渡邊 幸之介, 今城 英樹, 西 宏章, 工藤 知宏, “RHiNET の概要と Martini の設計／実装”, 情報処理学会研究報告 2001-ARC-144, pp.37–42, Jul. 2001

3. 渡邊 幸之介, 山本 淳二, 土屋 潤一郎, 田邊 昇, 西 宏章, 今城 英樹, 寺川 博昭, 上嶋 利明, 天野 英晴, “RHiNET/MEMOnet ネットワークインタフェース用コントローラチップ Martini の予備評価”, 情報処理学会研究報告 2001-ARC-144, pp.49–54, Jul. 2001

4. 渡邊 幸之介, 山本 淳二, 土屋 潤一郎, 今城 英樹, 寺川 博昭, 西 宏章, 田邊 昇, 上嶋 利明, 工藤 知宏, 天野 英晴, “ネットワークインタフェースコントローラチップ Martini の転送機構”, 並列処理シンポジウム JSPP2002 論文集, pp.157–158, May 2002

5. 渡邊 幸之介, 土屋 潤一郎, 天野 英晴, “ミスアラインメントを補正するモジュール間転送用インタフェース”, 電子情報通信学会技術研究報告 VLD2002-117, pp.211–214, Nov. 2002

6. 山本 淳二, 渡邊 幸之介, 宮脇 達朗, 西 宏章, 工藤 知宏, 天野 英晴, “PLI を用いたネットワークインタフェースコントローラとホストプログラムの協調シミュレーション”, 情報処理学会研究報告 2001-ARC-145, pp.73–78, Nov. 2001

7. 天野 英晴, 山本 淳二, 渡邊 幸之介, 土屋 潤一郎, 金子 直人, 工藤 知宏, “クラスタコンピュータ用ネットワークインタフェースチップ Martini における代行処理機構”, 電子情報通信学会技術研究報告 CPSY2001-54, pp.23–30, Oct. 2001

8. 原田 浩, 山本 淳二, 土屋 潤一郎, 渡邊 幸之介, 天野 英晴, 工藤 知宏, 石川 裕, “RHiNET の高速通信ライブラリ PMv2 による評価”, 情報処理学会研究報告 2001-ARC-147/2001-HPC-089, pp.145–150, Mar. 2002

9. 山本 淳二, 渡邊 幸之介, 土屋 潤一郎, 原田 浩, 今城 英樹, 寺川 博昭, 西 宏章, 田邊 昇, 上嶋 利明, 工藤 知宏, 天野 英晴, “高性能計算をサポートするネットワークインタフェース用コントローラチップ Martini”, 並列処理シンポジウム JSPP2002 論文集, pp.35–42, May 2002

10. 大塚 智宏, 渡邊 幸之介, 土屋 潤一郎, 原田 浩, 山本 淳二, 西 宏章, 工藤 知宏, 天野 英晴, “RHiNET ネットワークインタフェースの性能評価”, 電子情報通信学会技術研究報告 CPSY2002-44, pp.23–28, Aug. 2002

11. 北村 聡, 天野 英晴, 渡邊 幸之介, 大塚 智宏, “PC クラスタ用ネットワーク RHiNET-2 上における動的負荷分散アルゴリズムの評価”, 情報処理学会研究報告 2002-ARC-152, pp.73–78, Mar. 2003

12. 大塚 智宏, 渡邊 幸之介, 北村 聡, 原田 浩, 山本 淳二, 西 宏章, 工藤 知宏, 天野 英晴, “分散並列処理用ネットワーク RHiNET-2 の性能評価”, 先進的計算基盤システムシンポジウム SACSIS2003 論文集, pp.45–52, May. 2003

13. 鯉渕 道紘, 渡邊 幸之介, 河野 賢一, 上樂 明也, 天野 英晴, “RHiNET-2 クラスタを用いたルーティングアルゴリズムの実機評価”, 電子情報通信学会技術研究報告 CPSY2003-13 pp.43–48, Aug. 2003

14. 大門 優, 松尾 亜紀子, 大塚 智宏, 渡邊 幸之介, 天野 英晴, “反応を伴った圧縮性流体計算による RHiNET-2 の評価”, 電子情報通信学会技術研究報告 CPSY2003-22, pp.19–24, Aug. 2003

15. 鯉渕 道紘, 大塚 智宏, 渡邊 幸之介, 天野 英晴, “RHiNET-2 クラスタにおけるユニキャストを基にしたマルチキャストアルゴリズムの評価”, 情報処理学会研究報告 2004-EVA-8, pp.25–30, Mar. 2004

16. 大塚 智宏, 渡邊 幸之介, 北村 聡, 鯉渕 道紘, 山本 淳二, 西 宏章, 工藤 知宏, 天野 英晴, “RHiNETプロジェクトの最終報告”, 情報処理学会研究報告 2004-ARC-158, pp.31–36, May 2004

17. 鯉渕 道紘, 渡邊 幸之介, 大塚 智宏, 上樂 明也, 天野 英晴, “RHiNET-2 クラスタを用いたシステムエリアネットワーク向けトポロジの実機評価”, 先進的計算基盤システムシンポジウム SACSIS2004 論文集, pp.381–388, May 2004

### その他

1. 渡邊 幸之介, 土屋 潤一郎, 天野 英晴, “J-joint: ミスアラインメントを補正するモジュール間転送用インタフェース”, 第 4 回 LSI IP デザイン・アワード 完成表彰・開発助成部門

## その他の論文

### 公刊論文

1. Tomonori Yokoyama, Naoyuki Izu, Jun-ichiro Tsuchiya, Konosuke Watanabe, Hideharu Amano, Tomohiro Kudoh, “Design and implementation of RHiNET-2/NI0: a reconfigurable network interface for cluster computing”, IEICE Transaction Vol.E86-D, No.5, pp.789-795, May 2003

### 国際会議

1. Naoyuki Izu, Tomonori Yokoyama, Junichiro Tsuchiya, Konosuke Watanabe, Hideharu Amano, “RHiNET/NI: A reconfigurable network interface for cluster computing”, 12th International Conference on Field Programmable Logic and Application, Sep. 2002

2. Akira Kitamura, Yoshihiro Hamada, Yasuo Miyabe, Tetsu Izawa, Tomotaka Miyasiro, Konosuke Watanabe, Tomohiro Otsuka, Noboru Tanabe, Hironori Nakajo, Hideharu Amano, “Evaluation of Network Interface Controller on DIMMnet-2 Prototype Board”, The 6th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT2005), pp.778-780, Dec. 2005

3. Tetsu Izawa, Konosuke Watanabe, Akira Kitamura, Yasuo Miyabe, Tomotaka Miyasiro, Hideharu Amano, “Cooperative Simulation Environment of Hardware Plugged into a DIMM slot”, Proceedings of International Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI2006), pp.78-84, Apr. 2006

### 研究会ほか

1. 金子 直人, 宇野 正樹, 渡邊 幸之介, 山本 淳二, 工藤 知宏, 天野 英晴, “ハードウエア設計のマルチコンテキスト化手法”, 電子情報通信学会技術研究報告 2001-SLDM-103, pp.147–152, Nov. 2001

2. 天野 英晴, 金子 直人, 渡邊 幸之介, 宇野 正樹, 土屋 潤一郎, “マルチコンテキスト FPGA を用いたネットワークコントローラ”, 電子情報通信学会技術研究報告 2001-SLDM-104, pp.17–24, Jan. 2002

3. 北村 聡, 伊豆 直之, 田邊 昇, 濱田 芳博, 中條 拓伯, 渡邊 幸之介, 大塚 智宏, 天野 英晴, “DIMMnet-2 ネットワークインタフェースボードの試作”, 情報処理学会研究報告 2004-ARC-159, pp.151–156, Jul. 2004

4. 北村 聡, 伊豆 直之, 伊沢 徹, 宮代 具隆, 宮部 保雄, 渡邊 幸之介, 大塚 智宏, 濱田 芳博, 田邊 昇, 中條 拓伯, 天野 英晴, “FPGA を用いたメモリスロット装着型ネットワークインタフェースの設計”, 第 12 回 FPGA/PLD Design Conference ユーザ・プレゼンテーション, pp.13–20, Jan. 2005

5. 伊澤 徹, 渡邊 幸之介, 北村 聡, 宮部 保雄, 宮代 具隆, 天野 英晴, “メモリスロット装着型ハードウェアの評価検証環境の構築”, 情報処理学会研究報告 2005-ARC-163, pp.1–6, May 2005