複数OSを活用した
高コストパフォーマンスを実現する
ストリームサーバアーキテクチャの提案

Low Cost - High Performance
Streaming Server Architecture
Making Use of Multiple Operating Systems

東京大学大学院

学際情報学府学際情報学専攻

総合分析情報学コース

坂村研究室

竹内 理

目 次

第1章　はじめに 4

第2章　外付け I/O エンジンアーキテクチャ 7

2.1　解決したい課題 8

2.2　解決方式の概要 9

2.2.1　市販のストリームサーバのモジュール構成 9

2.2.2　外付け I/O エンジン方式の概要 11

2.2.3　実現方法の検討 12

2.3　HiTactix の概要 16

2.3.1　基本機構 17

2.3.2　GDSL 17

2.4　実装の概要 23

2.4.1　全体構成 23

2.4.2　改変内容の概要 26

2.4.3　接続プロトコルの概要 27

2.5　提案方式の評価 29

2.5.1　改変ソースコード量の評価 29

2.5.2　配信性能の評価実験 31

2.5.3　配信タイミング保証性能の評価実験 36

2.5.4　配信レート保証性能の評価実験 38

2.6　提案方式の考察 40

2.7 本章のまとめ 41

第3章 軽量仮想計算機モニタを利用したOSデバッガ **42**

3.1 解決したい課題 42

3.2 軽量仮想計算機モニタを用いたOSデバッグの基本機能 46

3.2.1 提案デバッグ方式概要 46

3.2.2 部分ハードウェアエミュレーション機能の実装方式 49

3.2.3 軽量メモリ領域保護機能の実装方式 54

3.2.4 実装上の問題点 63

3.2.5 性能評価 68

3.2.6 本節のまとめ 76

3.3 軽量仮想計算機モニタを利用したOSデバッガ向けのログ&リプレイ機能の実装方式 76

3.3.1 ロギング&リプレイ機能 76

3.3.2 ロギング&リプレイ機能の実現上の課題 79

3.3.3 課題の解決方式 82

3.3.4 実装の詳細 89

3.3.5 性能評価 92

3.3.6 考察 100

3.3.7 本節のまとめ 103

第4章 コストパフォーマンスの評価 **105**

4.1 評価方法 105

4.1.1 評価対象 105

4.1.2 評価内容 106

4.2 評価結果 107

4.2.1 定性的な評価結果 107

4.2.2 定量的な評価結果 109

第 5 章 関連研究 116

5.1 外付け I/O エンジン方式の関連研究 116

5.2 軽量仮想計算機モニタを利用した OS デバッガの関連研究 117

第 6 章 まとめ 122

第1章　はじめに

ブロードバンドネットワークの普及と共に，大規模な商用ストリーミングサービスが出現しはじめた [1]．商用ストリーミングサービスの本格普及には，ユーザからのコンテンツ収入よりも安いコストでのサービス提供を可能にする配信基盤が必須となる [2]．この配信基盤の重要な一構成要素がストリーム配信サーバであるため，サービス事業者も，ストリーム配信サーバに対し，多数のユーザに，安価に，かつ安定品質でコンテンツ配信サービスを提供できるコストパフォーマンスの高さを求めるようになってきている．

一般に，コストパフォーマンスの高いストリーム配信サーバは，以下の要件を充足する必要がある．

1. 高い配信性能
2. 高い QoS 保証性能（安定レートでのコンテンツ配信能力）
3. 低いハードウェアコスト
4. 低い運用管理コスト（ハウジング代，電力消費代，故障ハード部品交換費用等）
5. 低い初期ソフトウェア開発コスト
6. 低いソフトウェア保守コスト（カスタマイズ，拡張機能開発，バグフィックスコスト）
7. 高いソフトウェア開発効率

しかしながら，上記商用ストリーミングサービスに使用されているストリーム配信サーバは以下のどちらかのアーキテクチャにより構成されており，上記要件の充足が難しい．

- 汎用 OS 上にストリーム配信サーバアプリケーションを搭載 [3, 4, 5]
- 筆者らが研究開発した HiTactix[6, 7] をはじめとする，専用 OS 上にストリーム配信サーバアプリケーションを搭載 [8, 9, 10]

前者のアーキテクチャを持つストリーム配信サーバは，コンテンツ配信のための I/O を汎用 OS が実行しており，高い配信性能，高い QoS 保証性能の達成が難しい [11, 12]．それに伴い，多数のユーザにサービスを提供するために必要なサーバ台数が大きくなり，ハードウェアコストの増大や運用管理コストの増大（ラック占有スペース増大に伴うハウジングコスト増大，電量消費量増大，ハード部品数増大に伴う故障ハード部品数の増大及び部品交換保守費用増大）を招く．

一方，後者のアーキテクチャを持つストリーム配信サーバは，専用 OS 上に，ストリーム配信サーバアプリケーションだけでなく，ストリーム配信サーバアプリケーションの管理機能を実装するために必要な管理ツール（SNMP プロトコル，crontab，パスワード管理機能等），OS デバッグ用ソフトウェア等の新規実装も必要となり，初期ソフトウェア開発コストが高くなる．また，専用 OS 上に構築されているデバッグ環境も低機能であるため，ソフトウェア開発効率が低い．この結果，カスタマイズ，機能拡張（バージョンアップ），バグフィックスなど，頻繁に行わねばならないソフトウェア保守のコスト向上を招く．

このような課題を解決し，ストリーム配信サーバのコストパフォーマンスの飛躍的な改善を図るためには，汎用 OS と複数 OS を活用したストリームサーバアーキテクチャを新規に提案し，両 OS の利点を兼ね備えさせることが解決策の一つとして考えられる．また，特に専用 OS 上のソフトウェアの開発効率を向上させるため，当該ソフトウェアの開発効率向上を可能にする開発環境も必要になる．本研究では，これらの課題の解決を目指し，以下を行う．

- 汎用 OS 搭載のストリーム配信サーバと専用 OS 搭載のストリーム配信サーバの利点を兼ね備えた「外付け I/O エンジンアーキテクチャ」の提案，及び当該アーキテクチャに基づくストリーム配信サーバの試作と，その性能向上効果や開発コード量削減効果などの定量的な評価 [13].

- 上記アーキテクチャに基づいても，専用 OS，及び専用 OS 上のアプリケーションの開発が必要となる．この開発効率を改善することを目的とした「軽量仮想計算機モニタを利用した OS デバッガ」の提案と試作．さらに，当該 OS デバッガが，現実的な CPU 負荷で開発効率改善を達成できることの実機での性能評価を通じた確認 [14, 15, 16].
- 「外付け I/O エンジンアーキテクチャ」「軽量仮想計算機モニタを用いた OS デバッガ」を用いることにより達成できるコストパフォーマンス改善の評価.

「外付け I/O エンジンアーキテクチャ」は，専用 OS が持つ高い配信性能，QoS 保証性能を引き出すことで，ハードウェアコスト，運用コストの低減を図る．また，汎用 OS が提供する管理ツールを有効活用することで初期ソフトウェア開発コストを低減する．さらに，ストリーム配信サーバアプリケーションの大部分を汎用 OS 上で実装することでソフトウェア保守コストの低減を実現する．すなわち，上記コストパフォーマンスの高いストリーム配信サーバの要件 1〜6 を充足する．また,「軽量仮想計算機モニタを利用した OS デバッガ」は，要件 7 のソフトウェア開発効率の向上を実現すると共に，OS ごとに OS デバッグ用ソフトウェアを実装する必要をなくし，要件 5 の実現にも寄与する．

以下，2 章では,「外付け I/O エンジンアーキテクチャ」により解決したい課題，解決のためのアプローチ，提案内容の概要，試作内容の概要について説明する．3 章では,「軽量仮想計算機モニタを利用した OS デバッガ」により解決したい課題，解決のためのアプローチ，提案内容の概要，試作内容の概要について述べる．4 章にて，これらを使用してストリーム配信サーバを構築することで期待できるコストパフォーマンスの改善の評価結果について説明し，最後に 5 章で，関連研究について言及する．

第2章　外付けI/Oエンジンアーキテクチャ

外付けI/Oエンジンアーキテクチャは,

1. 高い配信性能
2. 高いQoS保証性能
3. 低いハードウェアコスト
4. 低い運用管理コスト
5. 低い初期ソフトウェア開発コスト
6. 低いソフトウェア保守コスト

を実現することを目的に筆者らが設計した，汎用OS搭載のストリーム配信サーバと専用OS搭載のストリーム配信サーバの利点を兼ね備えたストリームサーバのアーキテクチャである．本章では，まず，2.1節にて，本アーキテクチャにより解決したい課題を定義する．2.2節にて，提案する課題解決方式の概要について説明する．2.3節にて，本アーキテクチャにおける専用OSとして筆者らが設計したHiTactixの概要について説明する．2.4節で本方式に基づいて筆者らが実装したストリーム配信サーバの実装の概要について説明する．さらに，2.5節にて，コストパフォーマンスの評価に必要となる，基本性能と開発コード量の基本評価結果について述べる．最後に2.6節にて，提案方式の利害得失についての考察を行う．

2.1 解決したい課題

「外付け I/O エンジンアーキテクチャ」は，汎用 OS 搭載の市販ストリーム配信サーバと機能互換性を維持しながら，HiTactix の様な専用 OS を利用した高性能，かつ配信品質を保証したストリーム配信を少ない初期開発工数で実現することを目的とする．この目的達成のため，既存の市販ストリーム配信サーバと HiTactix の様な専用 OS を搭載したサーバ（外付け I/O エンジン）を連動させる．既存の市販ストリーム配信サーバには，外付け I/O エンジンと連動するための改変のみを加える．一方，外付け I/O エンジンが，従来市販ストリーム配信サーバにより行なわれていたストリーム配信を代理実行するために，従来市販ストリーム配信サーバの配信機能部分のみを専用 OS 上に部分移植する．

このような連動により，市販ストリーム配信サーバとの機能互換性を維持しながらも，専用 OS を利用して，ストリーム配信の性能向上，及び配信品質保証が実現できる．また，これに伴い，大規模な商用ストリーム配信サービスの提供に必要なハードウェアコストや運用管理コストを低減できる．また，上記実現のために必要な初期開発工数は，既存の市販ストリームサーバの改変工数と，専用 OS 上で動作するストリーム配信機能の部分移植工数のみに抑えられる．また，ストリーム配信以外の機能（管理機能など）はすべて汎用 OS 上に実装されたままになるため，ソフトウェア保守のための開発も多くが汎用 OS 上のアプリケーション改変で済み，ソフトウェア保守コストも低減できる．

さらに「外付け I/O エンジンアーキテクチャ」の提案の妥当性を検証するため，Darwin ストリームサーバ[1] を「外付け I/O エンジン方式」に対応させるべく改変を加え，HiTactix 搭載の外付け I/O エンジンと連動させる．そして，両サーバの連動のために必要となる開発工数が大きくならないこと，及び連動オーバヘッドに起因するストリーム配信性能の劣化が大きく発生せず，HiTactix が提供する配信性能を引き出せることを定量的に検証する．

[1] QuickTime や MPEG4 フォーマットのストリーム配信が可能なストリームサーバ.

2.2 解決方式の概要

本節では，前節で述べた課題を解決するために著者らが新規に提案した「外付けI/Oエンジン方式」，すなわち，既存の市販ストリーム配信サーバにわずかな改変を加えるだけで，当該サーバのストリーム配信性能の向上，及びストリーム配信の品質保証機能の追加を可能にするストリームサーバの構成方式について述べる．

まず，市販のストリーム配信サーバの典型的なモジュール構成について述べる．次に，外付けI/Oエンジン方式の概要について述べる．最後にその実現方法の検討結果について説明する．

2.2.1 市販のストリームサーバのモジュール構成

市販ストリームサーバの一般的なモジュール構成を図2.1に示す．市販ストリームサーバは，配信要求処理モジュール，管理要求処理モジュール，配信制御モジュール，配信ドライバモジュールからなる．

配信要求処理モジュールは，クライアントから配信開始／停止要求を受信する．配信開始要求受信時には，管理要求処理モジュールに対して認証／課金要求の発行を行う．また，セッション情報（現在ストリーム配信を行っているクライアントとの間で確立されているコネクションに関する情報）の更新後，当該クライアントに対する配信開始／停止要求を配信制御モジュールに対して発行する．

管理要求処理モジュールは，配信要求処理モジュールからの認証／課金要求を受信すると，必要に応じ外部データベース・管理サーバ等と連動しながら当該要求に応じた処理を実行する．また，外部の管理サーバからモニタ要求を受信すると，配信ドライバモジュールが設定するモニタ情報（CPU負荷情報，配信レート情報等）を参照しつつ，当該情報を外部の管理サーバに返送する．

配信制御モジュールは周期的に駆動する（通常100ミリ秒程度の周期で駆動する）モジュールで，セッション情報に記載されているクライアントに対して，自モジュール駆動周期分の配信要求を配信ドライバモジュールに対して発行する．ストリームデータの最後まで配信が完了した等の理由で，配信すべきストリームデータが存在しない状態が一定時間以上継続している場合，セッショ

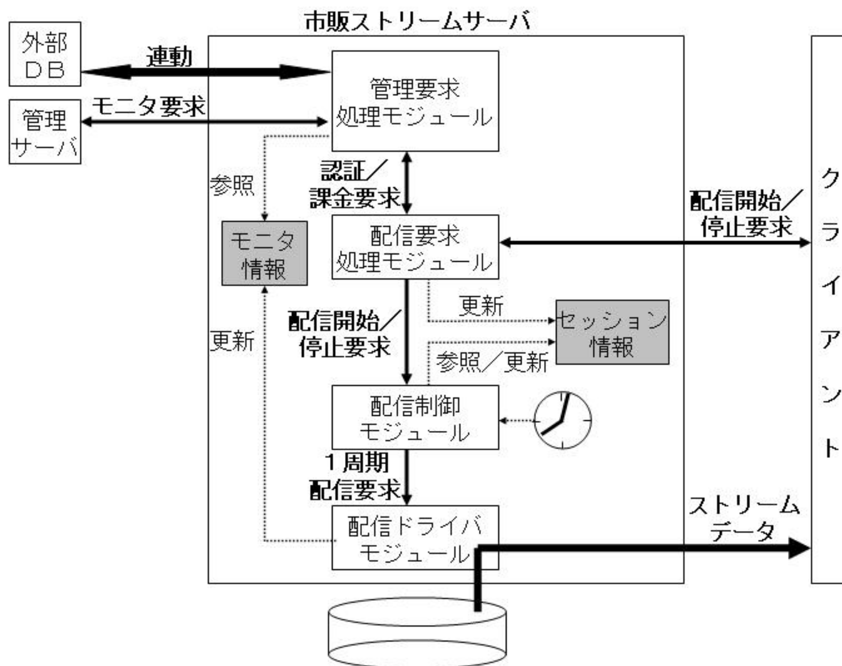

図 2.1: 市販ストリームサーバのモジュール構成

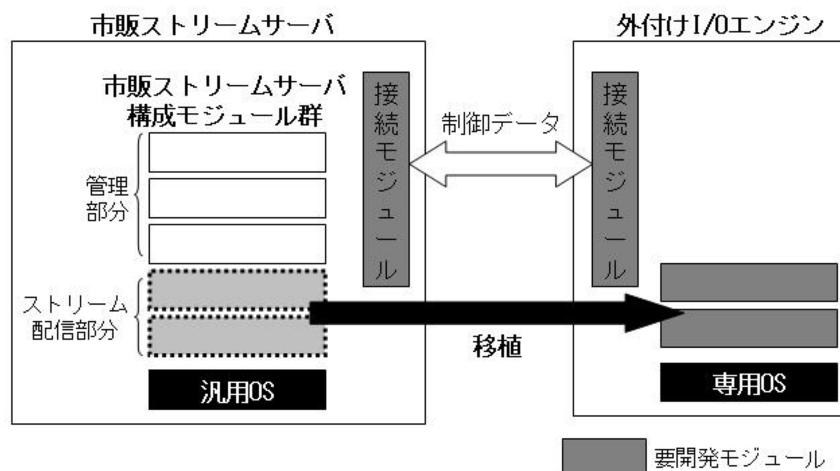

図 2.2: 外付け I/O エンジン方式を用いたストリーム配信サーバの構成

ン情報を更新する．配信要求処理モジュールは定期的に（通常 30 秒に 1 度程度）セッション情報をチェックし，上記更新処理を検知したら，当該クライアントに対する配信処理の強制終了処理，具体的には，クライアントとの間に形成されているコネクションの強制切断処理等を実行する．

配信ドライバモジュールは，配信制御モジュールから配信制御モジュール駆動周期分の配信要求を受信すると，当該周期分のストリームデータをディスクから読み込み，ネットワーク経由でクライアントに対して読み込んだデータを配信する．ストリームデータのフォーマット（QuickTime, MPEG4 等）を解釈し，当該周期に相当するデータのバイト数を決定する処理も本モジュールが行う．さらに，必要に応じてモニタ情報を更新する．

2.2.2 外付け I/O エンジン方式の概要

外付け I/O エンジン方式は，市販ストリーム配信サーバの配信性能向上と配信品質保証機能の追加を，機能互換性を維持しつつも，少ない開発工数で実現するストリーム配信サーバの構成方式である．

外付け I/O エンジン方式を用いたストリーム配信サーバの構成を図 2.2 に示す．本方式では，市

販ストリーム配信サーバと機能互換なサーバを，汎用 OS を搭載した市販ストリーム配信サーバと，HiTactix の様な専用 OS を搭載した外付け I/O エンジンを連動させることにより実現する．

本方式では，市販ストリーム配信サーバの構成モジュールを，管理部分とストリーム配信部分に分割し，専用 OS 上にはストリーム配信部分のみを移植する．さらに，両サーバ間で制御データを送受するために，接続モジュールを両サーバ上に新規実装する．

本方式では，既存の市販ストリーム配信サーバで行っていたストリーム配信処理を外付け I/O エンジンに代理実行させる．外付け I/O エンジンは，専用 OS を利用して，高性能かつ配信品質を保証したストリーム配信を実現する．

また，本方式実現のために必要となる開発は，接続モジュールの新規実装と，ストリーム配信部分の専用 OS への移植のみに抑えられ，専用 OS 上に市販ストリームサーバの全構成モジュールを移植する従来方式と比して，開発工数の低減を期待できる．

また，ストリーム配信サーバと連携する外部データベース・管理サーバが異なるたびに頻繁にカスタマイズや機能拡張が必要となる管理要求処理モジュールが汎用 OS 上で実装されているため，ソフトウェア保守コストの低減も期待できる．専用 OS 上に実装されている配信処理は，ストリーム配信サーバで使用しているストリーミングプロトコルやストリーミングフォーマットが変更されない限りバージョンアップの必要がなく，この点でもソフトウェア保守コストの上昇の懸念を抑えられる．

2.2.3　実現方法の検討

HiTactix を用いて外付け I/O エンジンを実現する方法は，2.2.1 節で述べた構成モジュールのうちどのモジュールを HiTactix に移植するかにより，図 2.3 に示す 3 方法が考えられる．

方法 1 は，配信要求処理モジュール，配信制御モジュール，配信ドライバモジュールを HiTactix に移植する方法である．この方法では，外付け I/O エンジンがクライアントからの配信開始／停止要求を受信し，かつ，当該要求に応じたストリーム配信を実行する．認証／課金処理や管理サーバからのモニタ要求の処理のみを市販ストリーム配信サーバが行う．モニタ情報は外付け I/O エ

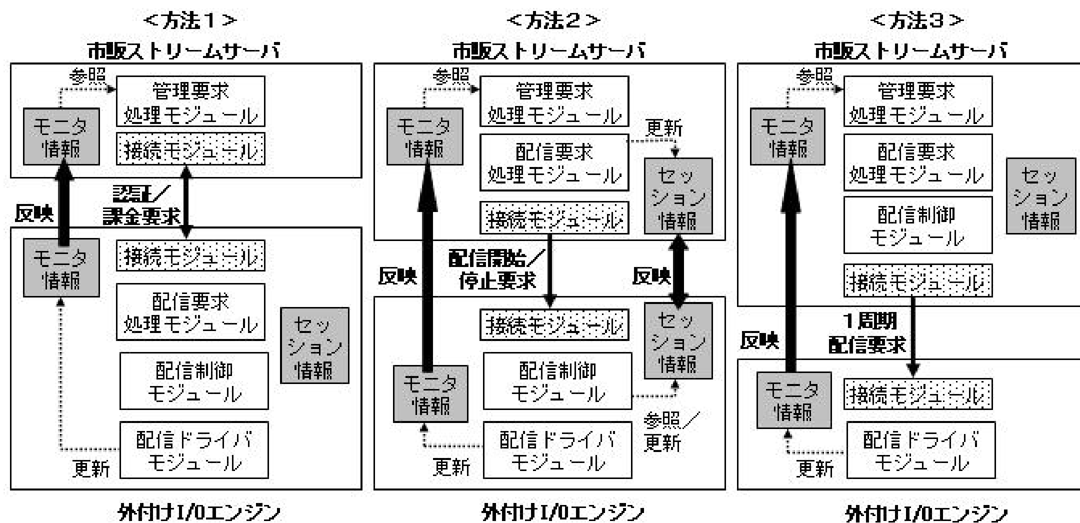

図 2.3: 外付け I/O エンジンの実現方法

ンジン側で更新するため，外付け I/O エンジンが保持するモニタ情報の内容を定期的（通常 1 秒に 1 度程度）に市販ストリーム配信サーバ側に反映させ，同期をとる必要がある．

方法 2 は，配信制御モジュール，配信ドライバモジュールを HiTactix に移植する方法である．この方法では，市販ストリーム配信サーバがクライアントからの配信開始／停止要求を受信する．当該要求を受信すると，市販ストリーム配信サーバは外付け I/O エンジンに要求を転送する．外付け I/O エンジンは，要求に応じたストリーム配信を代理実行する．この方法では，モニタ情報以外にセッション情報も，市販ストリーム配信サーバ，外付け I/O エンジンの双方で保持する．そのため，上記モニタ情報の同期の他に，セッション情報の同期が必要になる．セッション情報の同期のため，定期的に（30 秒に 1 度程度），各セッションにおけるストリーム配信が正常に行われているか否かを，市販ストリーム配信サーバから外付け I/O エンジンに問い合わせる．そして，外付け I/O エンジン側で配信完了等の理由でセッション情報の更新を行っていれば，市販ストリーム配信サーバ側にもその更新を反映させる．

方法 3 は，配信ドライバモジュールのみを HiTactix に移植する方法である．この方法では，クライアントからの配信開始／停止要求の受信の他に，配信タイミングの制御も市販ストリーム配

表 2.1: 外付け I/O エンジンの実現方法の比較

	方法 1	方法 2	方法 3
連動オーバヘッド （通信回数/秒）	○ （$1+N/300$）	○ （$1+N/30+N/50$）	× （$1+N/30+10N$）
配信品質保証	○ （HiTactix による配信制御）	○ （HiTactix による配信制御）	× （汎用 OS による配信制御）
スケーラビリティ	× （1URL に 1I/O エンジン）	○ （1URL に複数 I/O エンジン）	○ （1URL に複数 I/O エンジン）
開発工数	×	△	○

信サーバが行う．外付け I/O エンジンは，このタイミング制御に応じてストリームデータの I/O 処理を実行する．モニタ情報のみ上記方法で同期をとる必要がある．

上記 3 方法の優劣を，以下の観点から比較した．

- 連動オーバヘッド（ストリーム配信性能劣化の可能性）
- 配信品質保証の容易性
- スケーラビリティ
- 開発工数

比較結果を表 2.1 に示す．

まず，連動オーバヘッドによるストリーム配信性能劣化の可能性についての比較結果を示す．連動オーバヘッドは，連動に伴う市販ストリームサーバと外付け I/O エンジンとの間で発生し得る通信回数により比較した．発生し得る通信としては，

- モニタ情報同期のための通信
- セッション情報同期のための通信
- モジュール間での要求受け渡しのための通信

が考えられる．

先述した通り，モニタ情報同期のための通信はどの方法でも発生し，その頻度は1秒につき1回程度である．セッション情報同期のための通信は方法2でのみ発生し，その頻度は，高々30秒につき N 回程度（但し N は，外付けI/Oエンジンの同時配信ストリーム数を表す）である．

モジュール間での要求受け渡しのための通信は，方法1では，配信要求処理モジュールと管理要求処理モジュール間における認証／課金処理要求に伴い発生する．認証／課金処理要求はクライアントからの配信開始要求の到達に伴い発生する．配信開始要求の到達頻度は，高々ストリームデータの長さにつき N 回である．この比較では，ストリームデータの平均の長さを5分（300秒）とし，高々300秒につき N 回の頻度で発生すると仮定した．方法2では，配信要求処理モジュールと配信制御モジュール間における配信開始／停止要求に伴い発生する．2.4節で後述するように，Darwinストリームサーバでは，配信開始／停止要求に伴い6回の制御メッセージの通信を行う．すなわち，配信要求処理モジュールと配信制御モジュール間における配信開始／停止要求の通信頻度は高々300秒につき $6N$ 回である．方法3では，配信制御モジュールと配信ドライバモジュールとの間の1周期分の配信要求に伴い発生する．この頻度は，高々100ミリ秒につき N 回程度である．

以上より，1秒あたりに発生し得る最大の通信回数は，高々表2.1に記載した回数程度になると予測できる．N が1000程度であっても，方法1，方法2では，通信回数は1秒あたり5〜55回程度で抑えられ，このオーバヘッドはストリームデータの配信に伴う通信（Ethernet経由で1.5KBのデータ送信を1Gbps程度の速度を行っていると仮定すると，1秒あたり80000回以上）と比べると無視し得る．それに対して方式3では，通信回数が10000回を超え，ストリームデータの配信に伴う通信の10%以上を占め得る．

次に，配信品質保証の容易性の比較結果を示す．一般に，QuickTimeやMPEG4等のストリームデータには配信時刻情報（タイムスタンプ）が格納されており，ストリームサーバは，この情報に厳密に従ったタイミングで配信を行う．方法1，方法2では，配信ドライバモジュールの駆動をHiTactix上に実装された配信制御モジュールが実行する．それに対して，方法3ではこの制御を汎用OS上に実装された配信制御モジュールが行う．次節で述べる通り，HiTactixには高精度

(8 ミリ秒）で配信タイミングの制御を行う GDSL（Generic Data Streaming Library）と呼ぶライブラリが存在する．一方，汎用 OS には通常このような高精度な配信タイミングの制御機能がない．すなわち，配信品質保証に関しては，方法 1，方法 2 が方法 3 より優れていると言える．

次に，スケーラビリティについての比較結果を示す．ここで言う「スケーラビリティ」とは，クライアントがアクセスする際に使用する URL 一つにつき，最大配信セッション数をどの程度まで増やせるかを意味する．方法 1 では，外付け I/O エンジンの IP アドレスを持つ URL をクライアントが指定するため，一つの I/O エンジンの最大同時配信セッション数以上の同時配信を行えない．それに対して，方法 2，3 では，市販ストリーム配信サーバの IP アドレスを持つ URL をクライアントが指定するため，市販ストリーム配信サーバと連携する外付け I/O エンジンを増やしていけば，スケーラブルに最大配信セッション数を増やせる．

最後に，外付け I/O エンジンを実現するために必要となる開発工数についての比較結果を示す．接続モジュールの開発工数はどの方法でも大きな差はないため，HiTactix に移植するモジュール数が少ない方法 3，方法 2，方法 1 の順に開発工数は小さくなる．

以上の考察から，著者らは，方法 2 を用いて HiTactix 搭載の外付け I/O エンジンを実現することにした．開発工数についてのみ方法 3 より劣るが，HiTactix において，配信制御モジュール，配信ドライバモジュールを少ない開発工数で実装することを支援する GDSL というライブラリを提供することにより，その課題を一部解決した．GDSL の詳細については次節で述べる．

2.3 HiTactix の概要

本節では，HiTactix の概要について示す．HiTactix は，外付け I/O エンジンにおける高性能かつ配信品質を保証したストリーム配信を実現するために必要となる基本機構を保持する．この基本機構を応用し，2.2 節で述べた「配信制御モジュール」「配信ドライバモジュール」を HiTactix 上に少ない開発工数で実現可能とするために，著者らは，GDSL と呼ぶライブラリを新規に HiTactix 内部に実装した．以下，HiTactix が保持する基本機構と GDSL について説明し，HiTactix を用いることで外付け I/O エンジンの開発が容易になることを明らかにする．

2.3.1　基本機構

HiTactix は，高性能かつ配信品質を保証したストリーム配信を実現するために必要不可欠となる以下の基本機能を持つ．

高速 I/O 機構 HiTactix は，他の高速 I/O を指向する専用 OS と同様，ゼロコピーでネットワーク I/O 及びディスク I/O を実現する機構を持つ．さらに，アプリケーションが複数の非同期 I/O 要求を 1 つのシステムコールで一括して発行可能とする機能も持ち，I/O 完了通知オーバヘッドも低く抑えられる [17].

アイソクロナススケジューラ HiTactix は，アイソクロナススケジューラと呼ぶタイマ駆動型のスケジューラを持つ．アイソクロナススケジューラは，図 2.4 に示す様に，周期駆動が必要なスレッドの CPU 割り当て時間を予めテーブルを用いて予約する．予約した時間にわたり確実に当該スレッドが CPU を占有できるように，HiTactix はカーネル内部の排他制御や割り込み処理の管理を行っている [18].

サイクリックディスクスケジューリング HiTactix は，サイクリックディスクスケジューリングと呼ぶディスクスケジューラを持ち，指定したストリームのディスク I/O レートを保証できる．この I/O レート保証は，図 2.5 に示す様に，ディスク I/O レートを保証すべき高優先度ストリームのディスク I/O の実行時間を予め予約し，他の低優先度ストリームのディスク I/O による実行遅延が大きく発生しないことを保証する．このスケジューリングを応用すると，例えばディスクが高負荷になった場合にも，低優先度ストリームのディスク I/O レートを落とすことで，高優先度ストリームのディスク I/O レートを保証し続けることが可能になる [19].

2.3.2　GDSL

GDSL は，2.2 節で述べた配信制御モジュール，配信ドライバモジュールを少ない開発工数で実現することを目的に HiTactix 内部に新規実装したライブラリである．以下，本ライブラリの概要と実装の詳細について説明する．

図 2.4: アイソクロナススケジューラ

図 2.5: サイクリックディスクスケジューラ

2.3.2.1　GDSL の概要

GDSL は，デバイス間のストリームデータ転送処理を実行する際に必要となる I/O スケジューリング，I/O 要求発行処理，バッファリング処理をアプリケーションに代わって実行するライブラリである．アプリケーションは I/O の実行内容とその実行タイミングを GDSL に対して与える処理のみを行う．GDSL は自動的に指定されたストリームデータの転送処理を，指定されたタイミングに従い，低オーバヘッドで実行する．また，必要に応じてストリームデータの転送レート保証の有り／無しを制御することもできる．

GDSL を用いることで，高精度なタイミング制御機能を持つ「配信制御モジュール」の実現が容易になる．アプリケーションで I/O の実行タイミング指示を与えるだけで，GDSL が自動的に高精度な I/O スケジューリングを実現する．また，GDSL を用いることで，様々なストリームデータフォーマットや配信形態（ディスクからのオンデマンド配信，ネットワークから到達するストリームデータのライブ配信等）に対応した「配信ドライバモジュール」を少ない開発工数で実現できる．ストリームデータフォーマットや配信形態に非依存で必要となる処理は GDSL で，ストリームデータフォーマットや配信形態に依存する処理はアプリケーションで実現する様に GDSL は設計されており,「配信ドライバモジュール」開発者がこのレイヤ設計を行う必要を無くしている．

GDSL を用いたストリームデータの転送処理の概要を図 2.6 に示す．

アプリケーションは，GDSL の提供する source peer ／ destination peer ／ link と呼ぶカーネル資源を利用しながら，ディスクやネットワーク等のデバイス間におけるストリームデータ転送を実現する．source peer ／ destination peer はストリームデータの読み出し先／書き込み先となるファイルまたはソケットごとに生成し，当該ファイルやソケットに対する I/O 要求を発行する．link は，peer 間でのストリームデータ受け渡しのためのバッファキューイング領域を保持する．

デバイス間におけるストリームデータの転送のために必要となる I/O の内容は，アプリケーションが peer に対応させて登録する「I/O 要求発行モジュール」により制御する．I/O 要求発行モジュールは，各 peer がファイルシステムやプロトコルスタックに発行すべき I/O 要求の内容（I/O 先となるファイル名，ファイルオフセット，送信先アドレス，I/O サイズ，I/O すべきデータの指定等）

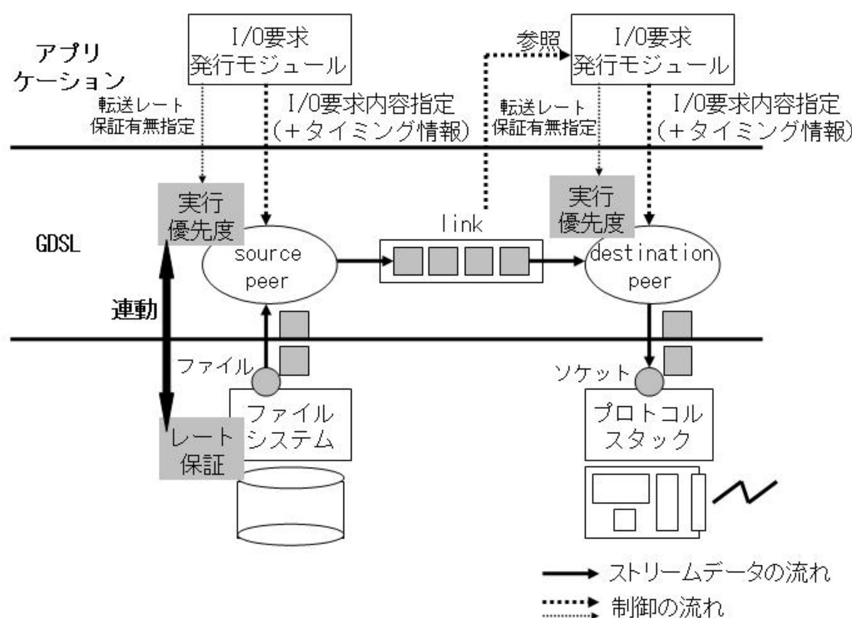

図 2.6: GDSL を用いたストリームデータの転送処理

を指定する．destination peer に登録された「I/O 要求発行モジュール」は，この要求内容を決定する際に，link にキューイングされているストリームデータを参照することもできる．

また，peer による I/O 要求発行タイミングの制御は,「I/O 要求発行モジュール」が I/O 要求内容を指定する際に，併せて I/O 要求発行タイミング情報を与えることにより行う．また，アプリケーションが peer を生成する際に，転送レート保証の有り／無しの指定を可能にすることにより，I/O 要求発行の優先度の制御も可能にしている．転送レート保証有りの peer による I/O の要求発行は，転送レート保証無しの peer による I/O の要求発行より優先して実行される．また，ファイルに対応する peer に転送レート保証有りを指定した場合，その読み出し／書き込みレートもサイクリックディスクスケジューリングを用いて保証される．

GDSL を用いることにより，I/O 要求発行モジュールの開発のみで，HiTactix 上の「配信制御モジュール」「配信ドライバモジュール」を実現可能になる.「配信ドライバモジュール」におけるサポートストリームデータフォーマットや配信形態の追加も，GDSL の提供機能を追加せずに実

現できる．

2.3.2.2　GDSL の実装

GDSL では，I/O 要求発行モジュールからの I/O 要求発行内容の受取りを，当該モジュールをアップコールすることにより実現する．また，I/O 要求発行モジュールからの I/O 要求発行タイミング情報の受取りは，当該モジュールのアップコールからリターンの際に，次回アップコール時刻をリターン値として受け取ることにより実現している．

GDSL では，上記 I/O 要求発行モジュールのアップコールの実行，及び当該モジュールから指示された I/O 要求の発行を，以下の 2 つのスレッドを用いて実現している．

- アイソクロナススケジューラにより，周期駆動と 1 周期あたりの CPU 割り当て時間が保証されたリアルタイム I/O 実行スレッド
- インターバルタイマを用いて周期駆動する非リアルタイム I/O 実行スレッド

リアルタイム I/O 実行スレッドは転送レート保証有りの peer を用いた I/O 要求発行を，非リアルタイム I/O 実行スレッドは転送レート保証無しの peer を用いた I/O 要求発行を行う．両者は I/O 要求発行のための CPU 時間の割り当て優先順位のみが異なる．両スレッドは，共に以下に示す処理を周期的に繰り返す．

1. 各 peer に登録されている I/O 要求発行モジュールを指定された時刻にアップコールする．I/O 実行スレッドは，アップコール時刻管理のため,「I/O 実行スケジューリングテーブル」を保持している．この詳細はすぐ後で述べる．
2. アップコールの結果，I/O 要求発行内容の指示を受けとる．また，次のアップコール時刻も併せて受けとる．
3. ファイルまたはソケットに対して,（2）で指示された内容に基づき，I/O 要求を HiTactix の提供する非同期 I/O インタフェースを用いて一括発行する．

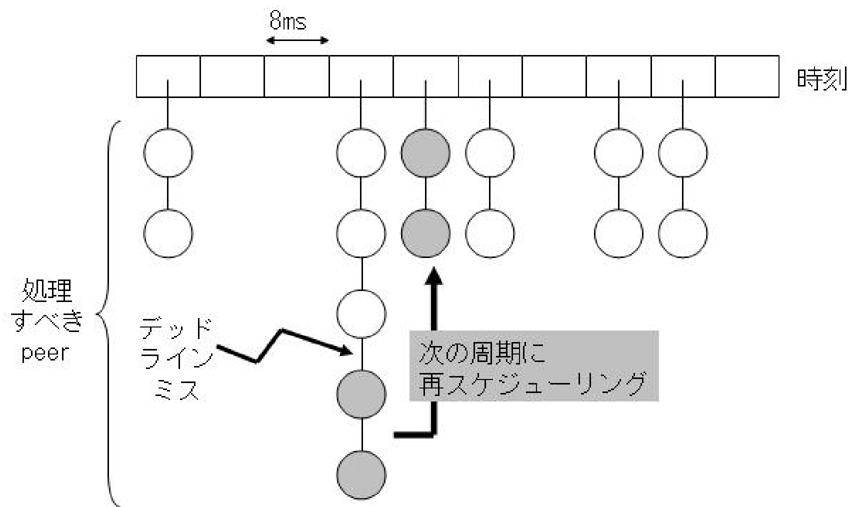

図 2.7: I/O 実行スケジューリングテーブル

4. 前の周期で発行した I/O 完了をチェックする．source peer に対して発行した I/O が完了していたら，当該 I/O の結果読み込んだストリームデータを link にバッファリングする．

5. （2）で指示されたアップコール時刻情報に基づき,「I/O 実行スケジューリングテーブル」を更新する．

「I/O 実行スケジューリングテーブル」の概要を図 2.7 に示す.「I/O 実行スケジューリングテーブル」は I/O 実行スレッドの駆動周期（現在の実装では 8 ミリ秒）ごとに，上記処理を行うべき peer が登録されている．ある周期に処理すべき peer が集中し，1 周期内ですべての peer の処理が完了しなかった場合，残った peer の処理は次の周期に実行する．この結果，I/O 実行要求モジュールが要求した I/O 要求発行タイミングと実際の I/O 要求発行タイミングとの間に誤差が発生し得る．この誤差の大きさがどの程度になるかについては，2.5 節で定量的に評価する．

図 2.8: 改変前の Darwin ストリームサーバのモジュール構成

2.4　実装の概要

本節では，外付け I/O エンジン方式に対応すべく著者らが Darwin ストリームサーバに加えた改変内容の詳細について述べる．そして，少ないソフトウェア開発工数で，Darwin ストリームサーバの機能互換性を維持しつつ，外付け I/O エンジン方式に対応可能であることを明らかにする．

2.4.1　全体構成

改変前の Darwin ストリームサーバのモジュール構成を図 2.8 に示す．

Darwin ストリームサーバはバージョンアップ作業を容易にするため，階層構造をしている．下位層に位置する基本モジュールは，上位層を追加実装するためのインタフェースを備えている．現在，このインタフェースを用いて，上位層に管理制御モジュールや QTSS ファイルモジュール等

が実装されている．

基本モジュールには，2.2 節で述べたストリームサーバの構成モジュールのうち，管理要求処理モジュール，配信要求処理モジュール，配信制御モジュール，配信ドライバモジュールの一部（ネットワーク送信部分のみ）に相当する部分が含まれる．管理制御モジュールには，管理要求処理モジュール相当部分が含まれる．QTSS ファイルモジュールには配信ドライバモジュール（ファイル読み込み部分のみ）相当部分が含まれる．

配信ドライバモジュール相当部分は，基本モジュールと QTSS ファイルモジュールに分割して存在し，その詳細動作は以下の通りになる．

1. 配信制御モジュールが配信ドライバモジュール（ファイル読み込み部分）に対して，1 周期分のストリームデータの送信要求を発行し，1 周期分のストリームデータのファイルからの読み込みを要求する．
2. 配信ドライバモジュール（ファイル読み込み部分）は要求されたストリームデータの読み込みを実行した後，配信ドライバモジュール（ネットワーク送信部分）に対して，当該データの送信を要求する．併せて配信制御モジュールに対して，配信ドライバモジュール（ファイル読み込み部分）の次回駆動時刻（通常，1 周期時間，すなわち 100 ミリ秒後）を通知する．
3. 配信ドライバモジュール（ネットワーク送信部分）は，要求されたストリームデータの送信を実行する．
4. 配信制御モジュールが指定された駆動時刻に配信ドライバモジュール（ファイル読み込み部分）を再駆動する．(1）に戻る．

ストリームサーバは，将来的に，

- 管理機構の機能向上
- 配信可能なストリームデータフォーマットの追加
- ライブ配信サポート（サポートする配信形態の追加）

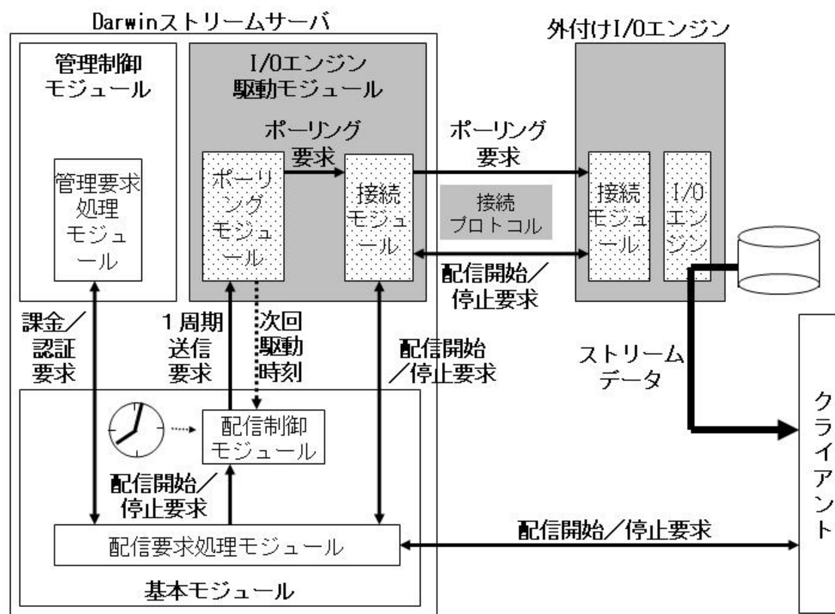

図 2.9: 改変後の Darwin ストリームサーバのモジュール構成

などの理由によりバージョンアップを行う[2]．

Darwin ストリームサーバでは，これらのバージョンアップ時には，基本モジュールの改変はほとんど不要である．管理機構の機能向上は，管理制御モジュールの改変により実現する．配信可能なストリームデータフォーマットの追加は，QTSS ファイルモジュールに相当するモジュールを新規に追加し，新規にサポートするストリームデータフォーマットの解釈，及び 1 周期分のストリームデータのファイルからの読み込みを可能にする．ライブ配信サポートは，QTSS ファイルモジュールに相当するモジュールを新規に追加し，ファイルからストリームデータを読み込むのではなく，ネットワークからストリームデータを読み込む.

2.4.2 改変内容の概要

今回の Darwin ストリームサーバの改変では，通常の Darwin ストリームサーバのバージョンアップ作業と同様，基本モジュールにはほとんど手を加えていない．図 2.9 に示す通り，上位層モジュールとして I/O エンジン駆動モジュールの新規実装を行ったのが主要な改変内容である．

I/O エンジン駆動モジュールは，接続モジュール，及びポーリングモジュールを含む．接続モジュールは，クライアントからの配信開始／停止要求の外付け I/O エンジンへの転送を行う．この転送のための接続プロトコルの詳細は次節で述べる．ポーリングモジュールはセッション情報，及びモニタ情報の同期を行うモジュールで，従来の配信ドライバモジュール（ファイル読み込み部分）を流用して実現している．

従来の配信ドライバモジュール（ファイル読み込み部分）と同様，ポーリングモジュールは配信制御モジュールから駆動される．しかし，配信ドライバモジュール（ファイル読み込み部分）とは異なり，ファイルからストリームデータを読み込まない．代わりに，セッションごとに外付け I/O エンジンが正常にストリームデータの配信を実行しているか否かのチェックを行う．このチェックは，接続モジュールに次節で述べる接続プロトコルを用いたポーリングを要求することにより実現している．正常に配信していなければ，セッション強制終了処理を実行する．配信制御モジュールに通知する次回駆動時刻を 30 秒後とすることで，ポーリングモジュールの駆動頻度を抑えている．同様の手法で，1 秒間隔でモニタ情報の同期も行っている．

外付け I/O エンジンには，接続モジュールの他に I/O エンジンモジュールを新規実装した．I/O エンジンモジュールとは，配信制御モジュール，配信ドライバーモジュール相当部分を GDSL 上に実装したモジュールである．

以上述べた通り，Darwin ストリームサーバを外付け I/O エンジン方式に対応させる際にも，Darwin ストリームサーバが管理するデータ構造や，QTSS ファイルモジュール以外のモジュールに改変を加える必要はほとんどなく，連動のために必要となる開発工数は大きくならない．より

[2] Darwin ストリームサーバがバージョン 3.3.8 からバージョン 4.0 にバージョンアップした際には，ここに示した 3 つがその主内容であった．

詳細な改変量の解析は 2.5.1 節で述べる．

2.4.3　接続プロトコルの概要

本節では，Darwin ストリームサーバと HiTactix 搭載の外付け I/O エンジンとの連動の際に使用する接続プロトコルの概要について述べる．

接続プロトコルは,

- 配信開始／停止要求を転送する機能
- セッション情報／モニタ情報の同期をとる機能

を持つ．これらの機能はすべて，Darwin ストリームサーバから外付け I/O エンジンに対して要求制御メッセージを送信し，外付け I/O エンジンが Darwin ストリームサーバに応答制御メッセージを返送することにより実現している．

配信開始／停止要求を転送する際に使用する制御メッセージの一覧を表 2.2 に示す．本制御メッセージの種類は，サーバ-クライアント間で RTSP[20] プロトコルに従い送受される制御メッセージに対応して存在する．

配信要求処理モジュールは，Play 以外の RTSP 制御メッセージ（配信開始／停止要求）を受信すると，接続モジュールに対して，対応する接続プロトコルの制御メッセージの外付け I/O エンジンへの送信を要求する．接続モジュールは外付け I/O エンジンからその要求の実行結果を応答として受信する．当該実行結果は配信要求処理モジュールを介してクライアントに返送される．

配信要求処理モジュールが，Play を RTSP 制御メッセージとして受信すると，まず，StartSession の送信を接続モジュールに要求する．上記と同様な手順でクライアントに StartSession の実行結果を返送した後，接続モジュールに対して RunSession の送信を要求し，外付け I/O エンジンによるストリーム配信を開始させる．この制御により，クライアントへの Play 要求の実行結果の返送が，ストリーム配信開始よりも前に行われることを保証する．

Darwin ストリームサーバと外付け I/O エンジンとの間では，クライアントからの配信開始／停

表 2.2: 接続プロトコルの制御メッセージ一覧

メッセージ名	対応する RTSP メッセージ	I/O エンジンへの要求内容
CreateSession	Describe	新規セッション情報の生成. ストリームデータの情報（ストリームデータフォーマット等）の返送.
SetupSession	Setup	ストリーム配信に必要なカーネル資源（peer，link 等）の生成.
StartSession	Play	ストリーム配信の開始準備（ファイルオフセット現在値等の配信制御データ初期化）.
RunSession	Play	ストリーム配信の開始.
StopSession	Pause	ストリーム配信の停止.
DestroySession	Teardown	ストリーム配信に使用したカーネル資源の解放. セッション情報の削除.

止要求到達に伴い，6 回の制御メッセージ送受信（外付け I/O エンジンへの要求送信と外付け I/O エンジンからの応答の受信）を行う．各制御メッセージの要求／応答には，要求／応答の種類の識別フィールド，各種要求実行時に必要となる情報（URL，クライアント IP アドレス等），実行結果が格納されている．その要求／応答のメッセージの大きさはそれぞれ最大で 296 バイト，272 バイトである．

セッション情報やモニタ情報の同期の際に使用する制御メッセージの一覧を表 2.3 に示す．GetEngineStatus は 1 秒につき 1 回，GetSessionStatus は 30 秒につき配信実行中のストリーム数だけ，Darwin ストリームサーバと外付け I/O エンジンとの間で送受される．GetEngeineStatus により，Darwin ストリームサーバは外付け I/O エンジンからモニタ情報（CPU 負荷，配信スループット等）を取得する．現在の実装では，本制御メッセージの要求は 12 バイトで，応答は 428 バイトである．また，GetSessionStatus により，セッションごとの配信状態（ファイルオフセットの現在値等）を取得する．本制御メッセージにより，配信が一定時間以上進行していないセッションを検知可能になる．現在の実装では，本制御メッセージの要求は 12 バイトで，応答は 44 バイトである．

本節で述べた制御メッセージの送受が，どの程度の外付け I/O エンジンの配信性能の劣化を招くかについての定量的な評価は，2.5 節で述べる．

表 2.3: 接続プロトコルの制御メッセージ一覧

メッセージ名	I/O エンジンへの要求内容
GetEngineStatus	モニタ情報（CPU 負荷，配信レート等）の返送
GetSessionStatus	セッションの配信状態（ファイルオフセットの現在値等）の返送

2.5 提案方式の評価

本節では，提案方式の評価結果について説明する．評価は，まず，Darwin ストリームサーバの改変のために要した改変ソースコード量の評価を行った．さらに，外付け I/O エンジン方式の定量的な性能評価のために，配信性能，配信タイミング保証性能，配信レート保証性能の 3 つについて行った．以下，それぞれの概要を説明する．

2.5.1 改変ソースコード量の評価

前節で示した改変で，Darwin ストリームサーバ／ HiTactix 上に新規実装，もしくは改変したソースコード量を表 2.4 に示す．表 2.4 に示す通り，今回の改変は，合計で 9.1K 行のソースコードの新規実装，改変，データ構造の追加を行った．改変対象となった Darwin ストリームサーバのコード量は 1.0K 行（QTSS ファイルモジュールのみ）であり，Darwin ストリームサーバ全体（66K 行）の 2%以下に抑えられた．

従来方式に従い，専用 OS 上に Darwin ストリームサーバの全構成モジュールを移植する場合，以下の開発工数が必要になると予想できる．

1. OS 依存モジュール[3] （15.0K 行）の HiTactix への移植
2. パスワード管理機能等の HiTactix に不足している機能の新規実装
3. 同期 I/O を用いて実現されているストリーム配信処理の高性能化

[3] Darwin ストリームサーバは，他 OS への移植を容易にするため，OS 依存モジュールの提供インタフェースを定義し，このインタフェース上に他モジュールを構築している．

表 2.4: 改変ソースコード量

モジュール名	改変対象量（K 行，使用言語）	改変量（K 行，使用言語）
接続モジュール／ポーリングモジュール（Darwin）	1.0（C++）	3.0（C++，改変）
接続モジュール（HiTactix）	–	1.0（C，新規実装）
I/O エンジン（HiTactix）	–	5.0（C，新規実装）
その他（Darwin）	–	0.1（C++，データ構造追加）
合計	1.0	9.1

上記（3）は，外付け I/O エンジン方式における HiTactix 上の I/O エンジンモジュールとほぼ同等の開発コード量（5.0K ステップ程度）で開発できると考えられる．また，外付け I/O エンジン方式における接続モジュール／ポーリングモジュールの実装工数は,（1）の工数の約 4/15 に抑えられる．外付け I/O エンジン方式を採用することにより，従来方式と比べて，上記（1）の工数の低減の他に,（2）の工数の分だけ工数低減が期待できる．

また，外付け I/O エンジン方式を用いると，将来，Darwin ストリームサーバがバージョンアップした場合においても，Darwin ストリームサーバや HiTactix 上のモジュールに大きな改変を必要としない．

例えば，管理機構の機能向上がなされても，接続モジュールや I/O エンジンモジュールの改変は不要である．サポートするストリームデータフォーマットの追加やライブ配信サポートがなされた場合には，HiTactix 上の I/O エンジンモジュールをバージョンアップする必要がある．しかし，HiTactix には GDSL が搭載されており，このバージョンアップに要する開発工数を低減できる．バージョンアップの際に追加実装すべきコード量は，高々現在の I/O エンジンモジュールのコード量，すなわち C 言語で 5.0K 行程度だと予測できる．

一方，従来方式では，Darwin ストリームサーバのバージョンアップ，特に管理機構の機能向上により，HiTactix 不足機能の新規実装がさらに必要になる可能性がある．

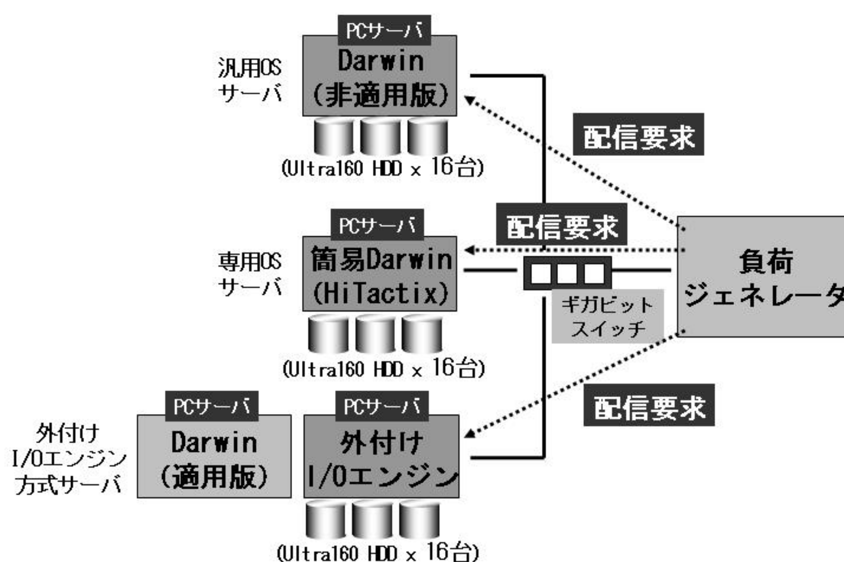

図 2.10: 配信性能評価実験システム

2.5.2 配信性能の評価実験

配信性能の評価実験では，外付け I/O エンジン方式を用いることで，従来方式で構築した専用 OS サーバ（専用 OS 上に市販ストリーム配信サーバ全体を移植して実現したサーバ）と比してどの程度の配信性能の劣化が発生し得るのか，また，汎用 OS サーバ（汎用 OS 上に構築されたストリーム配信サーバ）と比してどの程度の配信性能向上が期待できるのか，定量的に評価した．

配信性能の評価実験で用いた実験システムの構成を図 2.10 に示す．本実験では，汎用 OS サーバとして，改変を加えていない（外付け I/O 方式を適用していない）Darwin ストリームサーバを使用した．また，専用 OS サーバとして，HiTactix を搭載した簡易 Darwin ストリームサーバを使用した．簡易 Darwin ストリームサーバとは，外付け I/O エンジンに，Darwin ストリームサーバの配信要求処理モジュールの一部を HiTactix に追加移植することで実現したサーバである．外付け I/O エンジン方式サーバとして，前節の改変を加えた（外付け I/O エンジン方式に対応した）Darwin ストリームサーバと HiTactix 搭載の外付け I/O エンジンを使用した．

本実験で用いたハードウェアの仕様を表 2.5 に示す．本実験では，汎用 OS サーバ，専用 OS サー

表 2.5: ハードウェア仕様

サーバ種類	サーバ名	ハードウェア仕様
汎用 OS サーバ	Darwin ストリームサーバ （外付け I/O エンジン方式非適用版）	PC/AT 互換機（Pentium [4] III 1.26GHz 搭載） Gigabit Ethernet NIC（x2）（Alteon AceNIC チップ搭載） Ultra160 SCSI カード（x4）（LSILogic 53C1010 チップ搭載） Ultra160 HDD（x16）
専用 OS サーバ	簡易 Darwin ストリームサーバ	Darwin ストリームサーバ（外付け I/O エンジン方式非適用版） と同一構成
外付け I/O エンジン 方式サーバ	Darwin ストリームサーバ （外付け I/O エンジン方式適用版）	PC/AT 互換機（Pentium III 1.26GHz 搭載） Gigabit Ethernet NIC（x1）（Alteon AceNIC チップ搭載）
	外付け I/O エンジン	Darwin ストリームサーバ（外付け I/O エンジン方式非適用版） と同一構成

バは同一のハードウェア構成である．外付け I/O エンジン方式サーバは，上記ハードウェア構成に加えて，Darwin ストリームサーバ（外付け I/O エンジン方式適用版）用 PC サーバをもう 1 台追加している．

実験は，負荷ジェネレータから各サーバに対して可変のストリーム数（各ストリームはビットレート 1.42Mbps の QuickTime フォーマットのデータ）の配信要求を送信し，各サーバのストリーム配信レート及び配信実行中の CPU 負荷がどのように変動するかを測定した．但し，外付け I/O エンジン方式サーバは，外付け I/O エンジンの CPU 負荷を当該サーバの CPU 負荷として測定した．配信要求は，各サーバが保持する 16 台のハードディスクから均等に配信するように発行した．また，ディスクキャッシュを使用しないように，各配信要求はすべて異なるストリームデータの配信を要求した．

測定結果を図 2.11 及び図 2.12 に示す．各グラフの横軸は負荷ジェネレータが配信を要求したストリーム数を，縦軸は各サーバのストリーム配信レート及び CPU 負荷を表している．汎用 OS サーバは，配信要求ストリーム数が 220 を超えると，配信の異常終了が発生する（一定時間以上配信が実行できないストリームが生じはじめる）ため，測定ができなかった．そのため，配信要求ストリーム数が 220 ストリームを超えた時は，配信要求ストリーム数が 220 ストリームの時と同じ配信レートであると仮定した（図 2.11 の破線部分）．また，専用 OS サーバと外付け I/O エ

[4] Pentium は米国 Intel 社の登録商標です．

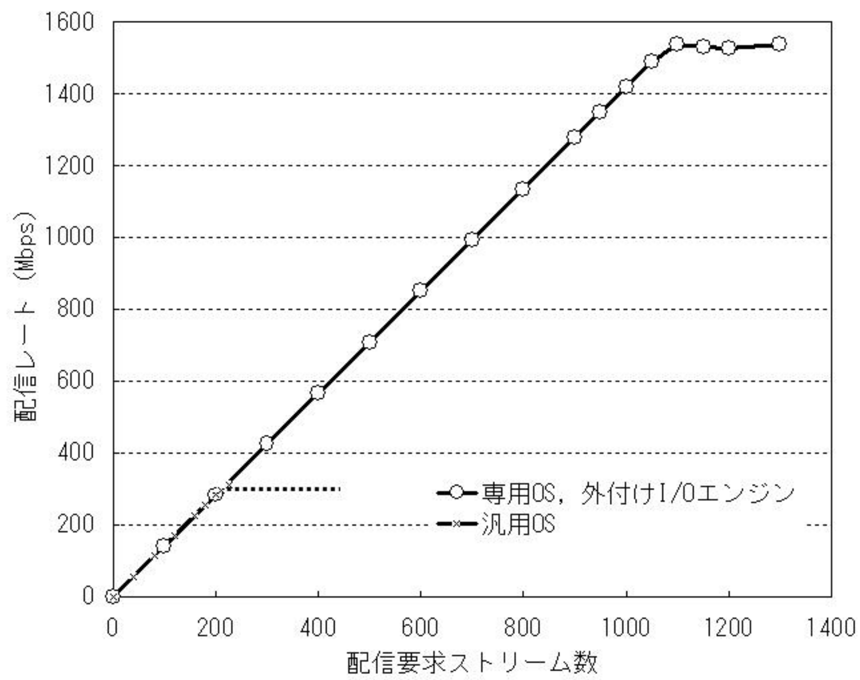

図 2.11: ストリーム配信レートの比較

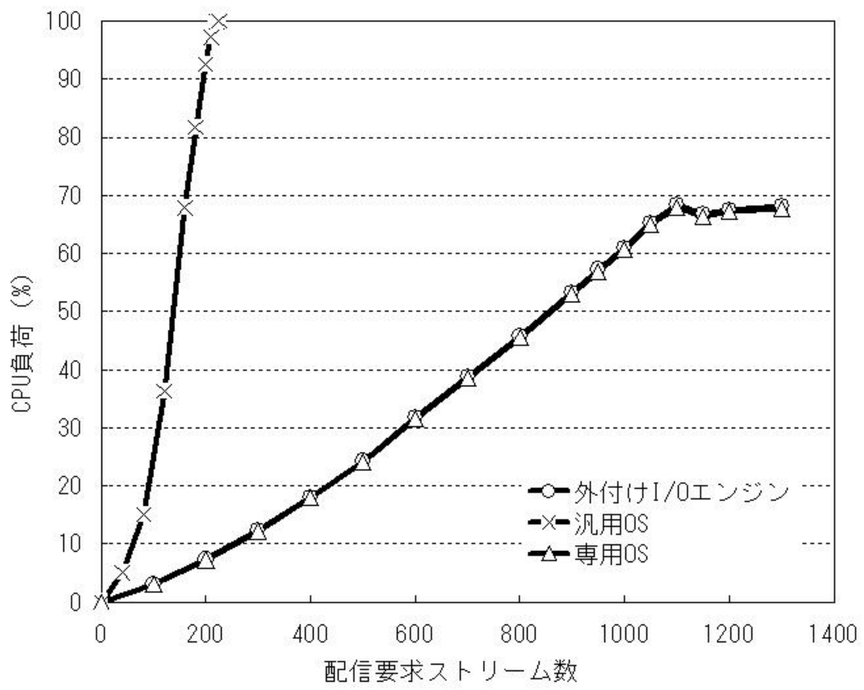

図 2.12: CPU 負荷の比較

ンジン方式サーバのストリーム配信レートの変動は全く一致するため，図 2.11 では両者を同一の実線で表している．

測定の結果，以下のことが明らかになった．

- 専用 OS サーバと比しても，外付け I/O エンジン方式サーバの CPU 負荷の増大は，平均で 0.49%に抑えられている．本実験では配信実行中の CPU 負荷を測定しているため，この配信性能の劣化は，2.2 節で述べたセッション情報の同期，モニタ情報の同期のために必要となる通信処理のみに起因している．短いストリームデータを配信した場合，配信開始／停止要求受け渡しのための通信処理による CPU 負荷増大も発生し得る．しかし，2.2 節で述べた通り，5 分程度の短いストリームデータを配信した場合でも，この通信の発生回数は 1 秒あたり $N/50$ 程度（N は同時配信ストリーム数）であり，セッション情報／モニタ情報の同期のために必要となる通信の発生回数 $1 + N/30$ と比して，同程度もしくはそれ以下である．そのため，外付け I/O エンジン方式における連動オーバヘッドに起因する配信性能の劣化は 1%以下に抑えられると予想できる．
- 汎用 OS サーバは，1 台のサーバを用いて，300Mbps 程度のストリーム配信性能を達成している．一方，外付け I/O エンジン方式サーバは，2 台のサーバを用いて 1.5Gbps 程度のストリーム配信を達成している．すなわち，外付け I/O エンジン方式の適用による 1 サーバあたりのストリーム配信性能向上の効果は 2.5 倍程度である．なお，外付け I/O エンジン方式においては，Darwin ストリームサーバ（外付け I/O エンジン適用版）にはほとんど負荷がかからないため，同時に稼働する外付け I/O エンジン方式を増やした場合，外付け I/O エンジンの台数に応じてスケーラブルにストリーム配信性能が向上すると期待できる．このため，外付け I/O エンジンの同時稼働台数を増やすと，1 サーバあたりのストリーム配信性能の向上の効果は，さらに増大すると予想できる．

なお，専用 OS サーバや外付け I/O エンジン方式サーバの場合，CPU 負荷が 100% に満たないにもかかわらず，ストリーム配信レートが増えなくなる．これは，サーバの PCI バスの帯域が飽

和しているためだと考えられ[5]，さらに広帯域な PCI バスを持つサーバを使用した場合，上記の 1 サーバあたりの配信性能向上の効果はさらに増大すると期待できる．

2.5.3　配信タイミング保証性能の評価実験

2.3 節で述べた通り，HiTactix 搭載の外付け I/O エンジンでは，GDSL を用いてストリーム配信のタイミング制御を行う．GDSL は，アプリケーション（I/O 実行要求発行モジュール）が指定する I/O 要求発行タイミングがある時刻に集中した場合，その一部の I/O 要求発行を次の周期に持ち越す可能性がある．そのため，厳密な配信タイミング保証ができない．この持ち越しによる配信タイミングの誤差がどの程度発生するか，図 2.10 に示した実験システムを用いて評価した．

本実験では，負荷ジェネレータから外付け I/O エンジン方式サーバに可変ストリーム数の配信要求を送信し，外付け I/O エンジンのパケットジッタがどのように変動するかを測定した．ここで言うパケットジッタとは，ストリームデータを格納する各パケットを送信すべき時刻と，実際の送信時刻との差を言う．送信すべき時刻は，QuickTime フォーマットのストリームデータに格納されている時間情報から算出でき，この時刻に基づきアプリケーションは配信タイミングを指定する．本実験では，I/O エンジンノードに 7 分間のストリームデータを配信させ，この 7 分間の最大と平均のパケットジッタを測定した．また，比較のため，同様の実験を汎用 OS サーバに対しても行った．

測定結果を図 2.13 に示す．グラフの横軸は，外付け I/O エンジン，または Darwin ストリームサーバ（外付け I/O エンジン非連動版）の CPU 負荷（配信要求ストリーム数を可変にした結果変動する）を示す．縦軸は，パケットジッタの大きさを示す．

測定の結果，以下のことが明らかになった．

GDSL を用いている外付け I/O エンジン方式サーバは，少なくとも，CPU 負荷が 60%以下であるならば，パケットジッタは平均で 3.5 ミリ秒，最大でも 7 ミリ秒〜13 ミリ秒で抑えられる．現

[5] 実験で使用した PC サーバが保持する PCI バスの理論性能は 532MB/秒である．1.5Gbps のストリーム配信を行った場合，この 70%を使用している．

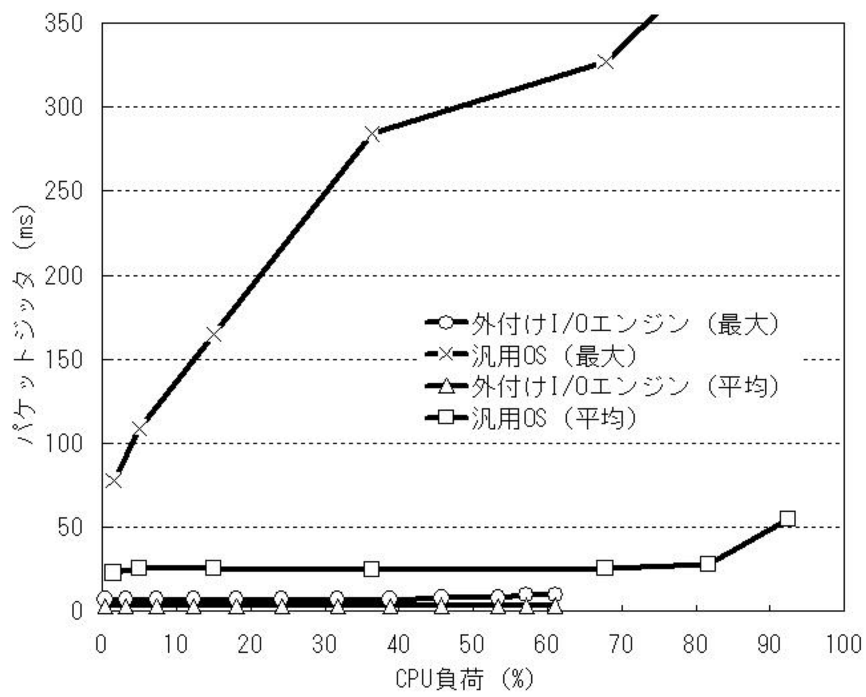

図 2.13: パケットジッタの比較

在の GDSL の実装では，8 ミリ秒を 1 周期として管理しているため，2 周期より大きい I/O 要求発行の持ち越しは発生していない．この最大パケットジッタの大きさは，汎用 OS サーバにおける平均のパケットジッタ（23 ミリ秒〜55 ミリ秒）の 1/2 以下，最大パケットジッタ（78 ミリ秒以上）の 1/11 以下である．

2.5.4　配信レート保証性能の評価実験

2.3 節で述べた通り，GDSL は，ストリームごとにデータの転送レート保証の有り／無しを制御できる．その結果，ディスクが過負荷状態になった場合においても，優先度の低い（転送レート保証の無い）ストリームのディスク読み込みレートを落とすことにより，優先度の高い（転送レート保証の有る）ストリームのデータを期待されるレートでディスクから読み込み，クライアントに配信できる．

外付け I/O エンジン方式を適用した Darwin ストリームサーバが，上記に示すストリーム配信制御を行えることを確認するため，以下に示す実験を図 2.10 の実験システムを用いて行った．

まず負荷ジェネレータから，1 ストリームの配信要求を外付け I/O エンジン方式サーバに対して送る．この 1 ストリームは，低い優先度での配信を要求する．次に，可変ストリーム数の配信要求を負荷ジェネレータから外付け I/O エンジン方式サーバに対して送る．これらのストリームは，高い優先度で配信するように要求する．なおこれらのストリームは，優先度の低いストリームと同じハードディスクからデータを読み出すことを要求した．優先度の高い配信を要求するストリーム数を変動させた場合，各優先度のストリームの配信レートがどのように変動するかを測定した．

測定結果を図 2.14 に示す．図 2.14 のグラフの横軸は優先度の高い配信を要求したストリーム数を，縦軸はストリーム配信レートの平均を表す．ストリーム配信レートは，優先度の高いストリームと優先度の低いストリームのそれぞれの平均値を測定した．

測定の結果，配信要求ストリーム数が 160 を超えてから期待通りに優先度の低いストリームの配信レートが落ち込むこと，及び優先度の高いストリームの配信レートは要求配信ストリーム数が 160 を超えても厳密に一定に保てることが確認できた．

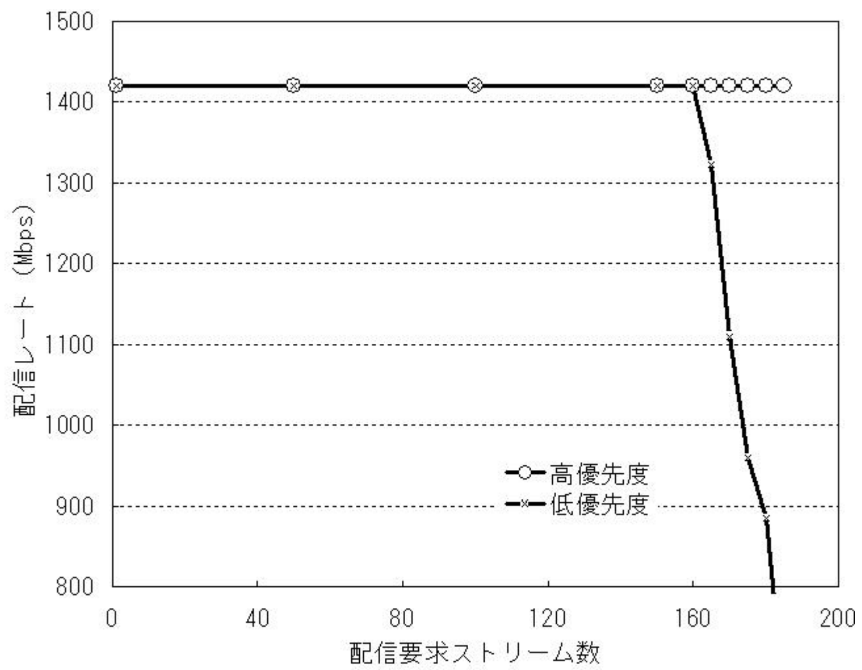

図 2.14: 配信優先度付けの効果

2.6 提案方式の考察

提案した方式は，以下の理由により，ストリーム配信サーバのコストパフォーマンスの改善を可能にする．

- 専用 OS 上にすべてのストリーム配信サーバを構築した時とほぼ同等の高い配信性能，QoS 保障性能を達成でき，結果として，大規模な商用ストリーム配信サービスの提供のために必要なハードウェアコストや運用管理コストを低減できる．

- 既存ストリーム配信サーバの完全互換のサーバを既存の市販ストリームサーバの改変工数と，専用 OS 上で動作するストリーム配信機能の部分移植工数のみで実現でき，初期ソフトウェア開発コストを低くできる．

- ストリーム配信以外の機能（管理機能など）はすべて汎用 OS 上に実装され，ソフトウェア保守のための開発も多くが汎用 OS 上のアプリケーション改変で済み，ソフトウェア保守コストも低減できる．

一方で，提案方式は，以下の制限があるため，本方式を適用できるストリーム配信サーバに限りがある．

- ストリーム配信サーバのストリーム配信処理において，クライアントからの配信開始／停止要求の送受を行うセッションと，クライアントへのストリーム配信を行うセッションが別々に定義されていることを前提としている．本前提が成立しないプロトコルにてストリーム配信処理を行うサーバには適用できない．例えば，RTSP などのプロトコルでストリーム配信を行うサーバには適用できるが，HTTP などのプロトコルで配信を行うサーバには適用できない．

- 本方式を適用するためには，既存のストリーム配信サーバのソースコードの改変が必要になる．そのため，ソースコードが開示されていないストリーム配信サーバには適用できない．

また，本方式には，以下に示す欠点もある．

- ストリーム配信サーバの構築には，最低でも 2 台のサーバが必要となる．そのため，従来の市販ストリーム配信サーバ 1 台でも実現できるような小規模ストリーム配信サーバに適用しても，ハードウェアコスト低減等は実現できない．

- ストリーム配信サーバのバージョンアップ時に，当該バージョンアップに追随するための開発工数が必要となる．管理機能の機能拡張等の多くのケースでは追加開発は必要ないが，配信スケジューリングの開始／停止インタフェースなどの改変などが行われたケースでは，多大な開発工数が必要になる．

2.7 本章のまとめ

本章では，既存の市販ストリーム配信サーバの機能互換性を維持しつつ，当該サーバの配信性能の向上，配信品質保証機能追加を少ない開発工数で実現するストリーム配信サーバの構成方式である外付け I/O エンジン方式を提案した．本方式では，既存の市販ストリーム配信サーバと HiTactix の様な専用 OS を搭載した外付け I/O エンジンを連動させ，既存の市販ストリームサーバのストリーム配信処理のみを外付け I/O エンジンに代行させる．

さらに本章では，Darwin ストリームサーバと HiTactix 搭載の外付け I/O エンジンを連動させた．この連動のために，HiTactix に，GDSL と呼ぶ低オーバヘッドでストリームデータの転送処理を行うライブラリを実装した．加えて，外付け I/O エンジン方式の定量的な評価を行った．評価の結果，上記連動は 9K 行程度のコード追加で実現できること，また，連動オーバヘッドによるストリーム配信性能の劣化は 1%以下に抑えられること，等を確認した．

第3章　軽量仮想計算機モニタを利用したOSデバッガ

軽量仮想計算機モニタを利用したOSデバッガは，外付けI/Oエンジン方式におけるソフトウェア（専用OSや専用OS上で動作するアプリケーション）の開発効率の改善を目的に設計・実装されたOSデバッガである．設計の際には，いかなるPCサーバ上で動作する専用OSにも適用できるように配慮することで，外付けI/Oエンジン方式における初期ソフトウェア開発コストの低減も併せて実現している．本章では，上記軽量仮想計算機モニタにおいて解決したい課題，課題解決方式の概要及び詳細，解決方式の評価結果について述べる．

3.1　解決したい課題

専用OSをPC/AT互換機上で開発するニーズが増大してきている．外付けI/Oエンジン方式を採用したストリーム配信サーバだけでなく，PC/AT互換機上に自社開発の独自OS（専用OS）を搭載し，I/O性能において差別化をはかるアプライアンスサーバ製品（Streamonix社のSX.Streamer[10]，Kesenna社のHigh Performance Network Driver[8]など）も近年数多く市場に出回ってきており，このような製品においても上記ニーズが存在する．

このような独自OSは，より高速なI/Oデバイス（10Gbps Ethernetや8Gbps FC HBAなど）や高速I/O支援機能を搭載した高機能I/Oデバイス（Intel社のI/O AT[21]仕様を充足したNICなど）が出現する度に，デバイスドライバの開発や，I/Oデバイス搭載機能を利用した新規I/O機能の開発を行う必要がある．また，いかなるPCサーバ上で動作する専用OSにも適用できるように配慮しなければ，特に外付けI/Oエンジン方式を適用したストリーム配信サーバの初期ソフ

トウェア開発コスト低減を実現できない．そのため，このような独自 OS のデバッグ環境は，様々な OS や I/O デバイスに対応可能で，かつ，高負荷 I/O 実行時の機能デバッグが効率的に行えることが望ましい．しかし，ICE 等のデバッグ用ハードウェアが提供されていない PC/AT 互換機では，従来，OS のデバッグ環境として，

1. 汎用 OS 上に構築されたハードウェアシミュレータ（または仮想計算機モニタ）及び当該ハードウェアシミュレータと連動するソフトウェアデバッガ [22, 23, 24]
2. ソフトウェアリモートデバッガ [25]
3. OS 内部に組み込まれた当該 OS 専用のデバッガ [26]

などを利用していた．これらはいずれも，A) デバッグ環境の安定稼働が保証できる，B) 様々な OS や I/O デバイスに追加開発なく適用できる，C) デバッグ時にも高速動作（特に高速な I/O 実行）が可能である，の 3 条件を同時に充足できない.（1）は，I/O デバイス（または I/O プロセッサ）や特権命令の厳密な動作シミュレーションをハードウェアシミュレータ（または仮想計算機モニタ）にて行うので，新規 I/O デバイスのエミュレータに大きな開発が必要になる上，I/O 性能も十分得られず，B），C）の条件を充足できない.（2）は，開発中の OS のバグに起因する異常動作によりリモートデバッグ動作（例えばシリアル通信）を阻害する可能性があるため A）の条件を充足できない．また,（3）は，OS が変わると多大な開発が必要となる上，OS とデバッガ間のメモリ保護等も実現できていないため，A），B）の条件を充足できない．

また，デバイスドライバなどの OS の開発においては，タイミングクリティカルなバグが多数発生し，このデバッグがいちばん難しい．このデバッグを支援するために，ログ&リプレイ機能 [27]（OS 実行履歴を保存しながら実行を行い，再現性の低いバグが顕在化した際に当該実行の再現を行う機能）が提案されているが，既存の当該機能の実現方式は，上記 (1) のデバッガを構成しているため，提案する OS デバッガに適用するためには，新規の実装方式が必要になる．

本章では，A) デバッグ環境の安定稼働が保証できる，B) 様々な OS や I/O デバイスに追加開発なく適用できる，C) デバッグ時にも高速動作（特に高速な I/O 実行）が可能である，の 3 条件を

同時に充足する，PC/AT 互換機上で動作する独自 OS のデバッグ方式として，軽量仮想計算機モニタを用いる方式を提案する．さらに，本軽量仮想計算機モニタにログ&リプレイ機能を実装するための実装方式を新規に提案する．あわせて，提案した軽量仮想計算機モニタとロギング& リプレイ機能の実装方式の性能評価を行い，提案方式が，従来よりも高速にデバッグ環境下での I/O を実行可能になること，及び現実的なオーバーヘッドでロギング&リプレイ機能を実現できること等を定量的に示す．

軽量仮想計算機モニタでは，リモートデバッグ機能を備える軽量仮想計算機モニタ上で開発中の独自 OS を動作させる．軽量仮想計算機モニタは，従来の仮想計算機モニタと同様に，実ハードウェアと同様のインタフェースを独自 OS に提供する．そのため，PC/AT 互換上で動作するいかなる独自 OS にも，本方式は適用可能である．

さらに，本軽量仮想計算機モニタは，従来の仮想計算機モニタと同様に，ハードウェア資源の仮想化を行う．しかしこの仮想化は，従来の様に，一つのハードウェア上に複数の OS を同時実行させることが目的ではない．仮想計算機モニタ上で動作する OS が実ハードウェア資源に直接アクセスすることを防ぐことで，たとえ OS が異常動作を行っても実ハードウェア資源の状態を正常に保ち，仮想計算機モニタ内に保持するリモートデバッグ機能を安定稼働させることが目的である．

本軽量仮想計算機モニタは，仮想化対象となるハードウェア資源を，リモートデバッグ機能の安定稼働の保証のために必要最小限な資源のみに絞る部分ハードウェアエミュレーション機能を持つ．独自 OS は上記仮想化対象以外の I/O デバイスに直接アクセスできるため，様々な I/O デバイスに対する I/O 処理を高速に実行できる．さらに，従来の仮想計算機モニタよりも低オーバヘッドで仮想計算機モニタと OS 間のメモリ保護を実現する軽量メモリ領域保護機能も併せて提供し，開発中の OS が異常動作しても，仮想計算機モニタが保持するリモートデバッグ機能が安定稼働することを保証する．

一方，ログ&リプレイ機能の新規実装方式では，上記軽量計算機モニタを用いた OS デバッガでのデバイスアクセス方式，すなわち，デバッグ対象の OS が NIC や HBA などのハードウェアに直接アクセスを行うことを仮定する．既存のロギング&リプレイ機能の実装方式は，すべての I/O

動作をエミュレートする既存のハードウェアシミュレータ（もしくは仮想計算機モニタ）を仮定しているため，その実装方式をそのまま適用することはできない．そこで，軽量仮想計算機モニタの機能拡張を行い，デバイスレジスタアクセスや DMA 転送などの履歴の保存・再生の実現することを目指し，軽量仮想計算機モニタの起動契機付与方式，及びデバッグ対象 OS の直接アクセス動作の軽量仮想計算機モニタによる追跡方式を新規に設計・実装する．

以下，本章では，上記軽量仮想計算機モニタを用いた OS デバッガの基本機能とログ&リプレイ機能の実装方式について，分けて説明する．

まず，3.2 節にて，軽量仮想計算機モニタを用いた OS デバッガの基本機能について述べる．3.2.1 節で，新規に提案する軽量仮想計算機モニタを用いた独自 OS デバッグ方式の概要について述べる．そして，提供する軽量仮想計算機モニタに，部分ハードウェアエミュレーション機能と軽量メモリ領域保護機能を備えることで，上記 A）〜C）の 3 条件の同時充足が可能であることを示す．次に 3.2.2 節と 3.2.3 節において，軽量仮想計算機モニタの特徴機能である部分ハードウェアエミュレーション機能と軽量メモリ領域保護機能の実装方式の概要について示す．次に，3.2.4 節において，部分ハードウェアエミュレーション機能と軽量メモリ領域保護機能を実装する際に直面した実装上の問題点について示す．3.2.5 節では，本 OS デバッグ方式の定量的な評価結果について述べる．

次に，ログ&リプレイ機能の実装方式に関して 3.3 節で述べる．3.3.1 節で，本研究で追加しようとしているロギング&リプレイ機能の概要について説明する．次に，3.3.2 節では，軽量仮想計算機モニタを用いた OS デバッガにロギング&リプレイ機能を実現するための課題を示し，デバイスレジスタアクセス履歴の保存・再生処理におけるデバイス依存処理の分離・極小化方式，及び DMA 転送履歴の保存・再生の全体実現方式の新規設計が必要であることを示す．3.3.3 節では，新規に提案する VM 層デバイスドライバを用いたデバイス依存処理の極小化・分離方式，及び DMA 転送履歴の保存・再生の実現方式について説明する．さらに，3.3.4 節では，本研究で提案する OS デバッガの実装の概要について説明する．3.3.5 節で性能評価結果を示し，提案方式での履歴の保存・再生で生じるオーバーヘッドを定量的に示す．最後に，3.3.6 節にて提案した OS デバッガの

利点/欠点と適用範囲を考察する．

3.2　軽量仮想計算機モニタを用いたOSデバッグの基本機能

3.2.1　提案デバッグ方式概要

本節では，本論文で新規に提案する軽量仮想計算機モニタを用いた独自 OS デバッグ方式の概要について述べる．まず，3.2.1.1 節にて，提案する OS デバッグ方式を適用可能なデバッグ対象，及びデバッグ範囲について述べる．次に，3.2.1.2 節にて，提案するデバッグ方式の方式概要について述べる．

3.2.1.1　デバッグ対象，範囲

提案するデバッグ方式は，独自 OS 及び独自 OS 上のアプリケーションをデバッグ対象とする．ここで言う「独自 OS」とは，ソースコードを入手可能である OS のことを指す．また，独自 OS としては，特に高速 I/O デバイスを介した高い I/O 性能を達成することを指向した OS に着目する．

また，本デバッグ方式のデバッグ範囲はスーパバイザモードで動作する独自 OS 及び独自 OS 上のアプリケーションのデバッグに限る．ユーザモードで動作する独自 OS 上のデバッガのデバッグについては考慮しない．

上記デバッグ対象，範囲を持つことで，例えば，独自 OS や独自 OS 上のアプリケーションにに新機能コードの追加をしており，その新機能コードのデバッグを行う，あるいは，独自 OS 向けの新規デバイスドライバを追加開発しており，既開発の（安定稼働している）アプリケーションを用いて新規デバイスドライバのデバッグをする，などの場合に提案デバッグ方式を利用可能になる．

3.2.1.2　方式概要

提案するデバッグ方式は，PC/AT 互換機上で動作する独自 OS の開発の際に，以下の条件を充足するデバッグ環境を提供することを目標としている．

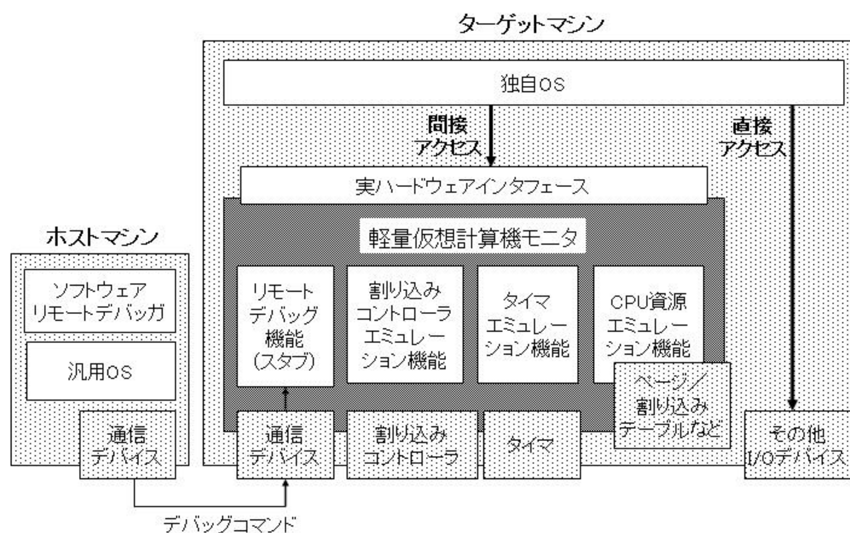

図 3.1: 提案デバッグ環境の構成

1. デバッグ環境の安定稼働が保証できる．

2. 様々な OS や I/O デバイスに追加開発なく適用できる．

3. デバッグ時にも高い I/O 性能で動作させることができる．

上記目的を達成するために，本デバッグ方式では，図 3.1 に示す構成を持つデバッグ環境を提供する．

本デバッグ環境は，従来のソフトウェアリモートデバッガを利用する際のデバッグ環境と類似した構成を持つ．デバッグ環境は，ホストマシンとターゲットマシンからなり，ホストマシン上ではリモートデバッグ機能を持つソフトウェアデバッガが動作する．ソフトウェアデバッガは，デバッグコマンド（ターゲットマシンのメモリ参照／更新，レジスタ参照／更新など）をユーザから受け取り，当該コマンドをターゲットマシンに転送する．

しかし，本方式は，独自 OS とは独立に開発された軽量仮想計算機モニタが予めターゲットマシンにデバッグ環境として組み込まれている点が従来のソフトウェアリモートデバッガと異なる．軽量仮想計算機モニタには，上記デバッグコマンドの受信，コマンドの実行，コマンド実行結果

の返信を行うリモートデバッグ機能を保持する．さらに，上記リモートデバッグ機能が利用するハードウェア資源の仮想化（エミュレーション）機能も提供する．なお，ここで言う「軽量」とは，デバッグ対象の独自 OS が I/O を実行する際に必要となるエミュレーション処理（I/O デバイスや割り込みコントローラなどのデバイスやページテーブル等の CPU 資源のエミュレーション）を低オーバヘッドで実現できることを言う．

軽量仮想計算機モニタは，リモートデバッグ機能が使用するハードウェア資源（割り込みコントローラ／タイマ／ページテーブル／割り込みハンドラテーブルなど）のみをエミュレートする部分ハードウェアエミュレーション機能を持つ．独自 OS は，当該資源に軽量仮想計算機モニタを介してアクセスする．そのため，独自 OS がバグに起因する異常動作を行っても，これらのハードウェア資源の状態を正常に保てる．しかし，それ以外のハードウェア資源，特に高速 I/O デバイス（SCSI コントローラ，Ethernet コントローラなど）へのアクセスは，独自 OS が直接行える．

仮想化されたハードウェア資源へのアクセスインタフェースは，実ハードウェア資源へのアクセスインタフェースと同様である．そのため，実ハードウェア上で動作する独自 OS は，大きな改変を加えることなく，本軽量仮想計算機モニタ上でも動作する．但し，実ハードウェア上で動作する独自 OS を本軽量仮想計算機モニタ上で動作させる際に，バグの混入の恐れの少ない簡単なコードの改変（アセンブリ言語で記述された命令列の機械的な置き換え等）は行っても良いことにした．PC/AT 互換機上で動作する従来の仮想計算機モニタは，バイナリで提供される複数の商用 OS を一つのマシン上で同時に利用することを目的としている．そのため，上記のようなコード改変は目的達成を不可能にし，あまり採用されていなかった．代わりに，仮想計算機モニタは，独自 OS 実行時にバイナリの動的な置き換えを行っていたが [28]，仮想計算機モニタ上で動作する独自 OS の実行性能の低下を招いていた．本仮想計算機モニタの目的は独自 OS のデバッグであるため，完全なバイナリ互換性の確保より，独自 OS の実行性能の確保を優先することにした．

また，IA-32 仕様 CPU が提供するページテーブルでは，各ページのアクセス保護を定義する際に 2 段階の特権レベルのみ（スーパバイザレベルとユーザレベル）が使用できる [29][1]．軽量仮想

[1] IA-32 仕様 CPU 自体は，レベル 0〜3 の 4 段階の特権レベルを提供している．しかし，ページテーブルを用いて各ページのアクセス保護を設定する際には，レベル 0〜2 の特権レベルは同一レベルとして扱われる．

計算機モニタに含まれるリモートデバッグ機能を安定稼働させるためには，軽量仮想計算機モニタ，独自 OS，独自 OS 上で動作するアプリケーションをそれぞれ別の特権レベルで動作させる必要がある．そのため，本軽量仮想計算機モニタでは，軽量仮想計算機モニタと独自 OS はスーパバイザレベルで，独自 OS 上で動作するアプリケーションはユーザレベルで動作させている．さらに，軽量仮想計算機モニタの動作領域への独自 OS からの不正アクセスを防止する軽量メモリ領域保護機能も提供し，たとえ独自 OS が，バグに起因した異常動作を行っても，リモートデバッグ機能が安定して稼働することを保証している．本軽量メモリ領域保護機能も，独自 OS 側にバグ混入の恐れの少ない簡単なコード改変（ページテーブル使用領域の仮想計算機モニタへの通知コードの追加等）を行うことにより，バイナリの動的置き換えをすることなく，保護を低オーバヘッドで実現している．なお,「軽量メモリ保護機能」の「軽量」も，メモリ保護実現のために必要なページテーブルのエミュレーション処理が低オーバヘッドで実現できることを示す.

上記デバッグ方式は，軽量メモリ保護機能の提供と，リモートデバッグ機能が利用するハードウェア資源の仮想化を行う部分エミュレーション機能により目標（1）を実現する．また，軽量仮想計算機モニタが実ハードウェアと同様のインタフェースを独自 OS に提供し，さらに独自 OS に様々な高速 I/O デバイスへの直接アクセスを許可することにより，目標（2）を実現する．さらに上記直接アクセスの許可は，目標（3）の実現にも寄与している．3.1 節で示した通り，デバッグ環境の安定稼働を保証する従来のデバッグ方式は，ハードウェアシミュレータ（または仮想計算機モニタ）を用いる方式だけであるが，提案方式はこの方式よりもデバッグ中の OS に提供する I/O 性能を改善する．I/O 性能改善の定量的な評価結果は 3.2.5 節で示す.

3.2.2　部分ハードウェアエミュレーション機能の実装方式

本節では，前節で述べた部分ハードウェアシミュレーション機能の実装方式について述べる．まず，3.2.2.1 節にて，エミュレーションの対象とすべきハードウェア資源について明らかにする．次に，3.2.2.2 節にて，部分ハードウェアエミュレーション機能にて上記ハードウェア資源のエミュレーションをどのように実装しているか，その概要を述べる.

3.2.2.1 エミュレーション対象のハードウェア資源

部分ハードウェアエミュレーション機能では，軽量仮想計算機モニタに含まれるリモートデバッグ機能が使用するハードウェア資源のみをエミュレートする．

リモートデバッグ機能は，外部デバイスとして，割り込みコントローラ（8259A チップ），タイマ（8253 チップ），ホストマシンとターゲットマシン間の通信を行う比較的低速な通信デバイス（現在の実装では IEEE1394 コントローラ）を使用する．現在の軽量仮想計算機モニタの実装では，割り込みコントローラとタイマのエミュレートを行っている．しかし，独自 OS からはホストマシンとターゲットマシン間の通信を行う通信デバイスへのアクセスを行わない（独自 OS は当該通信デバイスのデバイスドライバを保持しない）ことを前提とし，当該通信デバイスのエミュレートは行っていない．現在までに軽量仮想計算機モニタ上での動作を確認した HiTactix，μ ITRON 仕様 OS（TOPPERS/JSP）[30]，BSD/OS バージョン 4.3[31][2] は，この条件を満たしている[3]．

また，リモートデバッグ機能は独自 OS と同じ CPU を時分割で共有する．そのため，独自 OS が CPU を占有し動作し始めた後でも，必要な時にリモートデバッグ機能に制御が戻ることを保証する必要がある．具体的には，独自 OS が異常動作を行い例外が発生した場合にでも，確実に独自 OS 定義の例外ハンドラの起動前に軽量仮想計算機モニタ定義の例外ハンドラが起動し，リモートデバッグ機能を動作させる必要がある．リモートデバッグ機能の動作により，ホストマシン上で動作するソフトウェアデバッガに当該例外発生を通知可能になる．また，たとえ独自 OS が割り込み発生をマスクしつつ無限ループに陥った場合にでも，一定時間間隔に 1 回は軽量仮想計算機モニタ定義の割り込みハンドラに制御が移ることも保証しなければならない．ホストマシン上で動作するソフトウェアデバッガから発行されるブレーク要求に，リモートデバッグ機能が応答する必要があるためである．

---

[2] BSD/OS は米国 Windriver 社の登録商標です．

[3] 本条件を満たさない独自 OS の場合は，従来の仮想計算機モニタと同様に当該通信デバイスのエミュレートを行う必要がある．しかし，そのエミュレーションオーバヘッドが問題になる程，比較的低速な通信デバイスに高い負荷をかける独自 OS は，本論文では対象としない．また，当該通信デバイスのエミュレーションを行わなくても，デバッグ環境の安定稼働は保証できる．3.2.3 節で示す軽量メモリ領域保護機能を利用して，独自 OS から当該デバイスのレジスタ群へのアクセスを仮想計算機モニタは防止している．

この実現のために，軽量仮想計算機モニタは，割り込み／例外ハンドラ起動時に参照する CPU のハードウェア資源を仮想化する．具体的には，割り込み／例外ハンドラテーブル（IDT），セグメンテーションテーブル（GDT/LDT），ページテーブル，割り込み制御ビット（EFLAGS レジスタの IF ビット），コンテキスト格納領域（TSS）の仮想化を行っている．

3.2.2.2 実装方式の概要

前節で示したハードウェア資源のエミュレーションを実現するために，本軽量仮想計算機モニタでは，仮想計算機モニタを特権レベル 0 で，独自 OS を特権レベル 1 で，独自 OS 上のアプリケーションを特権レベル 3 で動作させる．本軽量仮想計算機モニタでは，実ハードウェア資源上で，独自 OS が特権レベル 0 で，独自 OS 上のアプリケーションを特権レベル 3 で走行することを仮定している．この仮定は，IA-32 仕様 CPU 上で動作する OS/2 以降のよく知られた OS すべてにおいて成立する [32].

割り込みコントローラとタイマ以外のデバイスを独自 OS から直接アクセスさせるために，軽量仮想計算機モニタはコンテキスト格納領域内の I/O 許可マップを設定する．割り込みコントローラとタイマデバイスの I/O ポートは特権レベル 0 からのみアクセス可能にするが，それ以外のポートはどの特権レベルからもアクセス可能にする．

独自 OS は，割り込みコントローラとタイマデバイスの I/O ポートにアクセスしようとすると，一般保護例外が発生する．例外発生を契機に，軽量仮想計算機モニタに制御が移り，割り込みコントローラまたはタイマエミュレーションモジュールが走行する．しかし，それ以外のデバイスの I/O ポートには，独自 OS は軽量仮想計算機モニタを介さずに直接アクセスする．また，PCI デバイス（但し，ホストマシンとターゲットマシン間の通信を行う通信デバイスは除く）のレジスタ群をマップするメモリ領域についても，独自 OS から例外を発生することなく当該メモリ領域にアクセス可能にすべく，軽量仮想計算機モニタはページテーブルの設定を行う．

また，割り込み／例外発生時に確実に軽量仮想計算機モニタ定義の割り込み／例外ハンドラが起動することを保証するために，CPU が提供するハードウェア資源は，表 3.1 に示す仮想化がな

される．

独自 OS（または独自 OS 上で動作するアプリケーション）走行中に割り込み／例外が発生した場合は，以下の手順で軽量仮想計算機モニタ定義の割り込み／例外ハンドラに制御が渡り，さらに必要に応じて独自 OS 定義の割り込み／例外ハンドラを起動する．

1. 割り込み／例外ハンドラテーブルのベースポインタ（IDTR）及びセグメントテーブルのベースポインタ（GDTR）はシャドウテーブルを指している．割り込み／例外が発生すると，軽量仮想計算機モニタ定義のハンドラが起動．コードセグメントも，モニタ動作用のセグメントに更新.

2. コンテキスト格納領域（TR）もシャドウを指しているため，スタックセグメントもシャドウの SS0 に格納されているモニタ動作用のセグメントに更新．スタックポインタは ESP0 に格納されている軽量仮想計算機モニタ定義の特権スタックへのポインタに更新.

3. 割り込み制御ビットは独自 OS（または独自 OS 上のアプリケーション）走行中は常に割り込み許可状態に設定．そのため，割り込み発生時にも，直ちに軽量仮想計算機モニタに制御が渡る．また，軽量仮想計算機モニタ定義の特権スタック領域，シャドウセグメントテーブル，シャドウ割り込み／例外ハンドラテーブル，シャドウコンテキスト格納領域，軽量仮想計算機モニタ定義の割り込み／例外ハンドラコードは 3.2.3 節に示す方法で物理常駐が保証されている．また，GDTR/IDTR/TR 等の更新はスーパーバイザレベルではあっても特権レベル 1 で動作する独自 OS からは行えないため，これらのレジスタの格納値も正常であることが保証される．そのため, (1) , (2) の処理途中に別な例外は発生し得ず，確実に割り込み／例外ハンドラが起動される．

4. リモートデバッグ機能，デバイスエミュレーション処理等の軽量仮想計算機モニタ内のモジュールが動作．発生した例外が，リモートデバッグ機能起動のための例外（デバッグ例外）やデバイスエミュレーションのための例外であった場合には処理を完了．それ以外は（5）以降の独自 OS への割り込み／例外通知処理を継続.

表 3.1: CPU 資源の仮想化方法の概要

資源名	仮想化方法
IDT	IDT のシャドウテーブルをモニタは保持. シャドウテーブルには，モニタ定義のハンドラを登録, 独自 OS 定義のハンドラを登録しない. モニタ定義のハンドラは，モニタ動作用のコード／メモリセグメントで動作.
GDT LDT	GDT/LDT のシャドウテーブルをモニタは保持. 独自 OS 動作用のコード／メモリセグメントディスクリプタは特権レベルを 1 に設定. モニタ動作用のコード／メモリセグメントディスクリプタを追加登録，特権レベルは 0 に設定. 独自 OS 定義のゲート／ TSS ディスクリプタは無効化.
ページテーブル	3.2.3 節を参照. シャドウページテーブルをモニタが保持. モニタ定義の特権スタック，GDT/IDT/TSS，ハンドラコードの物理常駐を保証.
IF ビット	ソフトウェアで代替ビットを保持 代替ビットに関わらず，独自 OS 実行中は割り込み許可.
TSS	TSS のシャドウをモニタは保持 モニタ定義の特権スタックセグメントセレクタ，スタックポインタをシャドウの SS0，ESP0 に登録. I/O 許可ビットマップを設定.

5. 割り込み発生時には，割り込み制御ビットの代替ビットを検査．代替ビットが割り込み禁止状態の場合には，独自 OS への割り込み通知をペンディング.

6. OS 定義の割り込み／例外テーブル，コンテキスト格納領域を参照．ソフトウェアにて，割り込み／例外発生に伴うスタック切り替え処理，命令ポインタの更新をエミュレート．独自 OS 定義の割り込み／例外ハンドラが起動する．

3.2.3 軽量メモリ領域保護機能の実装方式

3.2.1 節で述べた通り，軽量メモリ領域保護機能は，軽量仮想計算機モニタの動作領域への独自 OS からの不正アクセスを防止する機能である．本節では，この機能の実装方式について説明する．まず，3.2.3.1 節にて，本機能の実装方針について述べ，さらに，3.2.3.2 節と 3.2.3.3 節にて，本機能の実装方式の概要について述べる．

3.2.3.1 実装方針

軽量メモリ領域保護機能では，軽量仮想計算機モニタの動作領域をダイナミックマッピング領域とスタティックマッピング領域の 2 つの領域に分割して管理している．

ダイナミックマッピング領域とは，軽量仮想計算機モニタ走行中ばかりでなく，独自 OS 走行中にも，独自 OS が使用していない仮想空間領域にマッピングされるメモリ領域である．独自 OS が使用していない仮想空間領域は絶えず変化し得るので，独自 OS 走行中は，当該領域の仮想空間上のアドレスを動的に決定する．一方，スタティックマッピング領域は，独自 OS 走行中にはマッピングされず，軽量仮想計算機モニタ動作時にのみマッピングされる領域である．軽量仮想計算機モニタ走行中に使用する仮想空間の領域配置は独自 OS とは独立に決定できるため，スタティックマッピング領域の仮想空間上のアドレスは固定で良い．このように軽量仮想計算機モニタの動作領域を二つの領域に分割することで，マッピングアドレスを動的に更新する際に必要となる処理オーバヘッドの低減，必要な空き仮想空間領域量の低減，独自 OS からの不正な書き込みの防止対

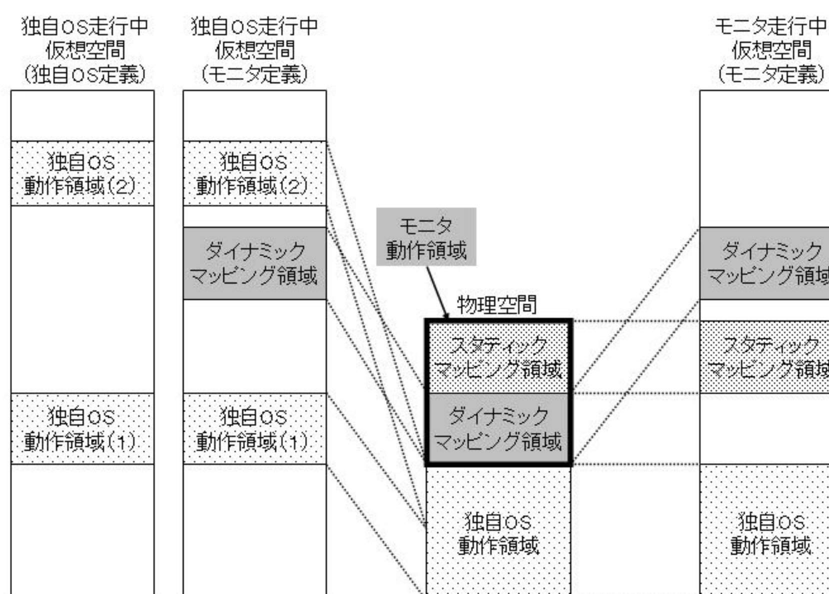

図 3.2: 仮想空間構成

象とすべきメモリ領域の低減，を実現している．

ダイナミックマッピング領域には，独自 OS の通常走行時や，軽量仮想計算機モニタ定義の割り込み／例外ハンドラの起動の際に参照する可能性がある軽量仮想計算機モニタのコード及びデータ構造（シャドウ割り込み／例外ハンドラテーブル等）が格納されている．一方，スタティックマッピング領域には，残りの軽量仮想計算機モニタのコード及びデータ構造が格納されている[4]．

図 3.2 に示す通り，独自 OS 走行中は，独自 OS 定義の仮想空間に，ダイナミックマッピング領域が追加マッピングされている．3.2.2.2 節で述べた通り，軽量仮想計算機モニタは，独自 OS が使用しているページテーブルのシャドウを保持する．シャドウページテーブルは，独自 OS が使用しているページテーブルの単純なコピーと，ダイナミックマッピング領域の追加ページマッピング情報からなる．独自 OS 走行中に，割り込み／例外が発生すると，ダイナミックマッピング領域内に格納されている軽量仮想計算機モニタ定義のコード（割り込み／例外ハンドラの入口コード）

[4] ホストマシンとターゲットマシン間の通信を行う通信デバイスのレジスタ群がマッピングされているメモリ領域もスタティックマッピング領域に含まれる．

が仮想空間の切替えを行い，スタティックマッピング領域に含まれるコード及びデータ構造にもアクセス可能になる．

ダイナミックマッピング領域は，独自 OS 走行中に使用する仮想空間の空き領域にマッピングされる．どの領域にマッピングされても走行可能とするために，当該マッピングの実行前に，ダイナミックマッピング領域内のコードやデータ構造に含まれるアドレス情報の更新を行う．また，ダイナミックマッピング領域内のデータ構造に，軽量仮想計算機モニタ動作時にもアクセス可能とすべく，軽量仮想計算機モニタ走行中に使用する仮想空間におけるダイナミックマッピング領域のページマッピング情報も更新する．

軽量メモリ領域保護機能では，以下の二つを保証することにより，独自 OS から軽量仮想計算機モニタのコードやデータ構造への不正アクセスを防止する．

- ダイナミックマッピング領域への不正書き込みを独自 OS が行うことを防止する．ここで言う「不正書き込み」とは，独自 OS の書き込みにより，軽量仮想計算機モニタ定義の割り込み／例外ハンドラが起動不可能になることを言う．
- ダイナミックマッピング領域及びスタッティックマッピング領域のページマッピング情報が，独自 OS 定義のページテーブルに加えられることを防止する．

上記それぞれの実装方式の概要については，次節及び次々節にて説明する．

3.2.3.2 ダイナミックマッピング領域保護方式

ダイナミックマッピング領域の構成を図 3.3 に示す．ダイナミックマッピング領域には，独自 OS の通常走行時や，軽量仮想計算機モニタ定義の割り込み／例外ハンドラの起動時に参照する可能性がある軽量仮想計算機モニタのコード及びデータ構造のすべて，具体的には，シャドウセグメンテーションテーブル（GDT/LDT のシャドウ），特権スタック領域（軽量仮想計算機モニタ定義の割り込み／例外ハンドラ起動時に使用），シャドウ割り込み／例外ハンドラテーブル（IDT のシャドウ），シャドウコンテキスト格納領域（TSS のシャドウ），軽量仮想計算機定義の割り込

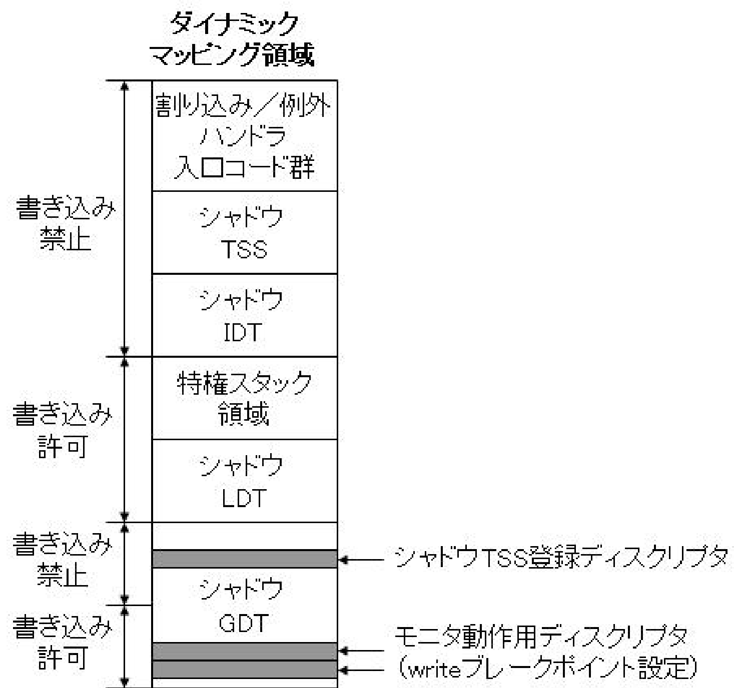

図 3.3: ダイナミックマッピング領域の構成

み／例外ハンドラの入口コード（起動すると仮想空間の切り替えを実行，その後にハンドラ本体を起動する）が格納されている．

独自 OS 走行時に使用する仮想空間では，ダイナミックマッピング領域は原則として書き込み禁止でマッピングされている．しかし，シャドウセグメンテーションテーブルと特権スタック領域に対しては，独自 OS が正常動作を行っている際にも書き込みが発生する．セグメントレジスタへのセレクタロードの際に，CPU は自動的に，シャドウセグメンテーションテーブルのディスクリプタのビジービットをオンにする．特権スタック領域には，割り込み／例外ハンドラ起動時に，現在のスタックセグメントセレクタ，スタックポインタ等の情報が格納される．

そのため，シャドウセグメンテーションテーブルの一部と特権スタック領域は，書き込みも許可してマッピングする代わりに，図 3.3 に示す方法にて，正常な軽量仮想計算機モニタ割り込み／例外ハンドラの起動を保証している．シャドウセグメンテーションテーブルに含まれる軽量仮想計算機モニタ動作用のセグメントディスクリプタに write ブレークポイントが設定されている[5]．また，シャドウコンテキスト格納領域の登録ディスクリプタは，書き込み禁止ページにマッピングされている．それ以外（シャドウセグメントテーブル内の独自 OS 動作用のセグメントディスクリプタや特権スタック領域）は書き込み許可ページにマッピングされているが，これらの領域は破壊されても，軽量仮想計算機モニタ定義の割り込み／例外ハンドラの正常起動を阻害することはない．

3.2.3.3 ページマッピングのチェック方式

軽量仮想計算機モニタは，ダイナミックマッピング領域及びスタッティックマッピング領域のページマッピング情報が，独自 OS 定義のページテーブルに加えられることを，以下の方法により防止する．

[5] 軽量仮想計算機モニタ定義の割り込み／例外ハンドラ起動に伴い，軽量仮想計算機モニタ動作用のディスクリプタのビジービットはオンに更新される．しかし，IA-32 仕様 CPU の仕様では，割り込み／例外の発生と共にブレークポイントはすべて無効となるため，本ビジービットの更新に伴い，デバッグ例外が発生することはない．また，IA-32 の仕様により特権レベル 1 で動作する独自 OS はブレークポイントを更新できず，独自 OS のバグにより，本 write ブレークポイントが無効化されることはない．

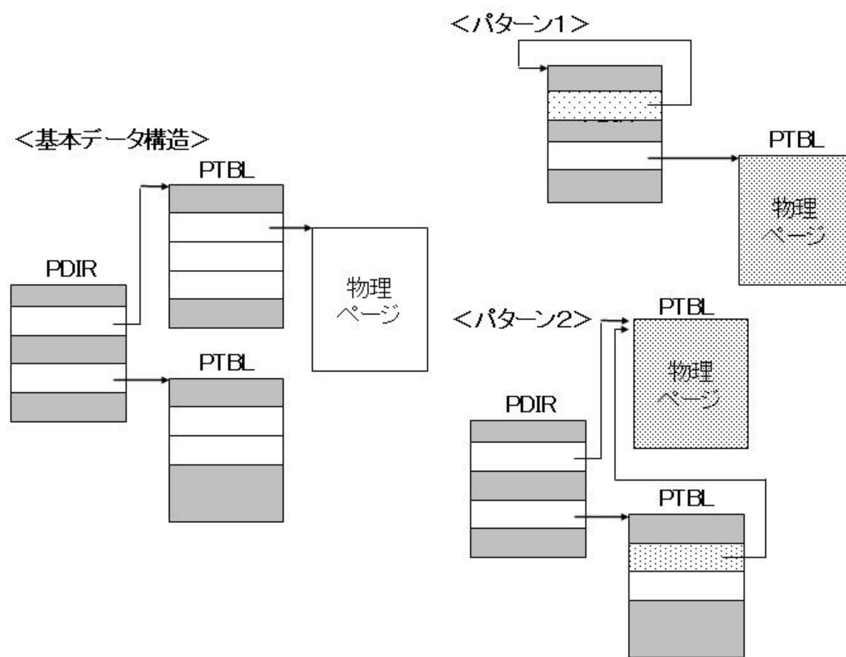

図 3.4: IA-32 のページテーブルの構成

- 独自 OS 定義のページテーブルとして使用されている物理メモリ領域へのデータ書き込みをページ不在例外として検出する．
- 上記ページ不在例外を契機に軽量仮想計算機モニタが動作する．軽量仮想計算機モニタは，書き込みにより，ダイナミックマッピング領域やスタティックマッピング領域へのマッピング情報が生成されないことを確認後，書き込み処理を実行する．

IA-32 仕様の CPU では，図 3.4 で示すような 2 段構造のページテーブルを保持する [29]．以下では，1 段目のページテーブルを PDIR，2 段目のページテーブルを PTBL と表記する．

IA-32 仕様の CPU が提供するページテーブを利用して PDIR/PTBL を仮想空間にマッピングする際には，以下の 2 つのいずれかの方法により行う．

1. PDIR のエントリの一つに自分自身の PDIR を指し示すエントリを追加する．

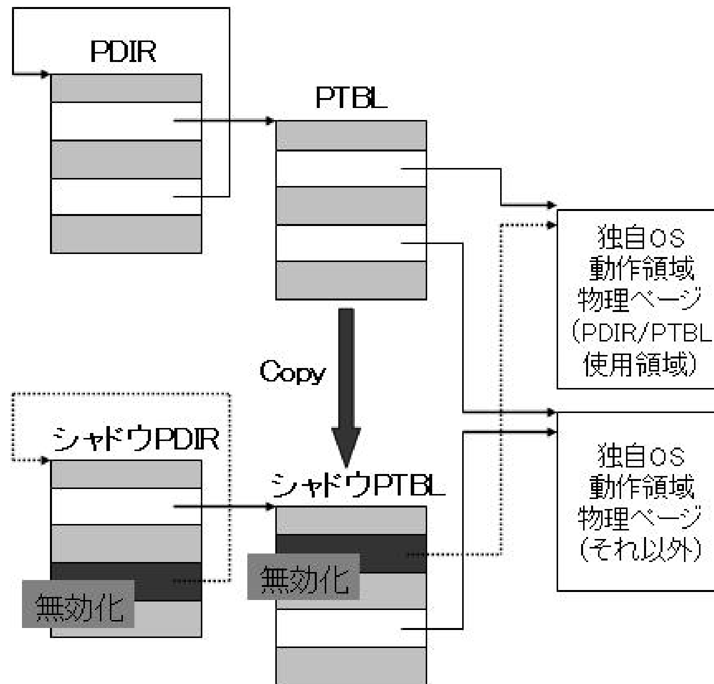

図 3.5: シャドウページテーブルのデータ構造

2. PTBL のエントリの一つに他の PDIR/PTBL を指し示すエントリを追加する．

（1）を行うと，PDIR の当該エントリに対応する仮想空間領域（4MB 分）に，現仮想空間で使用しているすべての PDIR/PTBL がマッピングされる．一方,（2）を行うと，PTBL の当該エントリに対応する仮想空間領域（4KB 分）に，当該エントリにより指し示されている PDIR/PTBL がマッピングされる．実装したページマッピングのチェック方式では，これらの PDIR/PTBL エントリを検索し，かつ対応する PDIR/PTBL のシャドウのエントリを無効化することで，独自 OS 定義のページテーブルとして使用されている物理メモリ領域へのデータ書き込みを検出している．

上記検出のため，軽量仮想計算機モニタは，図 3.5 に示すシャドウページテーブルを保持する．軽量仮想計算機モニタが作成するシャドウページテーブルは，PDIR のシャドウと PTBL のシャドウからなる．

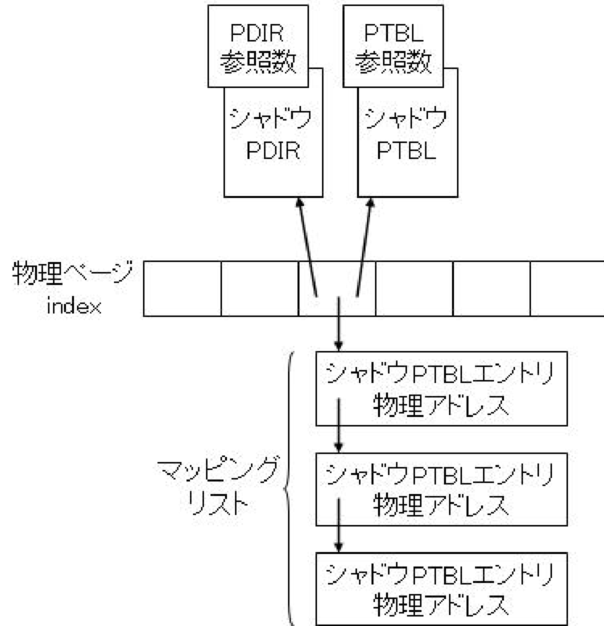

図 3.6: ページマッピング情報管理データ構造

PDIR のシャドウの各エントリは，独自 OS 定義の PTBL の物理アドレスを保持しない．代わりに，各 PTBL のシャドウの物理アドレスを保持する．但し，2 段目のページテーブルとして独自 OS 定義の PDIR を指し示しているエントリは上記（1）に対応するエントリと判定し無効化する．一方，PTBL のシャドウの各エントリには，独自 OS 定義の PTBL のエントリのコピーを保持する．ただし，すぐ後で述べる方法にて，上記（2）に対応するエントリを検出し，当該エントリはコピーを持たず無効化する．但し（2）で言っている「他の PDIR/PTBL」とは，現プロセスで使用している PDIR，PTBL だけに限らない．独自 OS にて生成されている全プロセスのいずれかで使用されている独自 OS 定義の PDIR，PTBL すべてを指す．使用する可能性のある独自 OS 定義の PDIR，PTBL へのいかなる書き込みも，その書き込み実行時に正当性をチェックすることで，仮想空間の切り替えの度に，新しい仮想空間のページマッピング情報の正当性をチェックすることを不要にしている．

使用する可能性のある独自 OS 定義の PDIR，PTBL のページマッピング情報は，図 3.6 に示すデータ構造を用いて管理する．本データ構造は，独自 OS 動作領域内の物理ページごとに，シャドウ PDIR，シャドウ PTBL，PDIR 参照数，PTBL 参照数，マッピングリストを持つ．PDIR 参照数は，当該物理ページを独自 OS 定義の PDIR として利用している仮想空間の数を示す．PTBL 参照数は，当該物理ページを 2 段目のページテーブルとして設定している独自 OS 定義の PDIR のエントリの数を示す．マッピングリストは，当該物理ページへのマッピング情報を持つシャドウ PTBL のエントリのアドレスを格納したリストである．

本データ構造により，新しいプロセスが生成され，独自 OS 定義の PDIR，PTBL として使用している物理ページが増えた場合に，当該ページをマップしており，上記（2）に対応する PTBL エントリのシャドウを高速に無効化できる．新しいプロセスが生成された場合におけるページマッピング情報の更新手順は以下の通りである．

1. 新規プロセスの独自 OS 定義の PDIR を参照し，各物理ページの PDIR，PTBL 参照数を増やす．PDIR/PTBL 参照数が 0 から 1 に更新された場合には，新たに当該物理ページに定義されている独自 OS 定義の PDIR/PTBL に対応するシャドウ PDIR/PTBL の初期化，具体的には使用物理メモリ領域の確保と，独自 OS 定義の PDIR/PTBL に基づいた当該領域の初期化を実施する．

2. 新規プロセスの独自 OS 定義の各 PTBL を参照し，マッピングリストを追加する．

3. （1）で PDIR/PTBL 参照数が 0 から 1 に更新された場合には，当該 PDIR/PTBL を指し示す独自 OS 定義の PTBL のエントリすべてを検出し，対応するシャドウを無効化する．具体的には，マッピングリストに登録されている各シャドウ PTBL のエントリを無効化する．

独自 OS 定義のプロセスが削除された場合にも，ページマッピング情報の更新を行い，不要なシャドウ PDIR，PTBL 用の物理メモリ領域の解放や，シャドウ PTBL エントリの有効化等を行う．

本ページマッピング情報の更新は，独自 OS 定義のプロセスの生成や削除を実行した際にのみ行われる．プロセス切り替えの際には，更新は必要ない．独自 OS 定義のプロセスの生成，削除をど

のように軽量仮想計算機モニタが認識するかについては，3.2.4.2 節で述べる．

3.2.4　実装上の問題点

本節では，まず 3.2.4.1 節と 3.2.4.2 節にて，3.2.2 節と 3.2.3 節で述べた部分ハードウェアエミュレーション機能，及び軽量メモリ領域保護機能を実装するにあたり直面した問題点とその解決方法について述べる．

本節で述べる実装上の問題点の多くは，独自 OS のソースコードの改変により解決している．3.2.1.1 節で述べた通り，提案している OS デバッグ方式では，ソースコードを入手可能な独自 OS をデバッグ対象としてるため，ソースコードの改変が必要であることは問題ない．しかし，改変に伴い，独自 OS にバグが混入する可能性が高まると問題になる．改変の複雑さや改変量に関する考察は 3.2.4.3 節で述べる．

3.2.4.1　部分ハードウェアエミュレーション機能の実装上の問題点

部分ハードウェアエミュレーション機能の実装を行うために，以下の 3 点の問題を解決する必要が生じた．

- 独自 OS が使用するコードセグメント／スタックセグメントセレクタの変換
- 割り込み制御ビットの参照／更新要求の捕捉
- 例外発生要因の特定

以下で，上記 3 点の問題の内容と解決策について述べる．

第 1 の問題点は，独自 OS が使用するコードセグメント／スタックセグメントセレクタの変換に関する問題である．独自 OS は，実ハードウェア上で動作する際には特権レベル 0 で走行するが，軽量仮想計算機モニタ上で動作する際には特権レベル 1 で走行する．そのため，軽量仮想計算機モニタ上で独自 OS が走行する際に使用するコードセグメント／スタックセグメントセレクタは，実ハードウェア上で動作する際と異なる．独自 OS にこの違いを隠蔽する必要がある．

上記違いを隠蔽するために，軽量仮想計算機モニタでは，以下を実行している．まず，シャドウのセグメントテーブル（GDT/LDT）では，独自 OS 定義のゲートディスクリプタ，TSS ディスクリプタをすべて無効にしている．ゲート通過時（独自 OS 上のアプリケーションからのシステムコール発行時等）やタスクスイッチ時に例外が発生し，軽量仮想計算機モニタが変換動作を行う．実際に独自 OS が特権レベル 1 で動作している場合でも，軽量仮想計算機モニタは特権レベル 0 のコードセグメント／スタックセグメントセレクタを独自 OS 定義の特権レベルスタックやコンテキスト格納領域にセーブする．また，ゲート通過後やコンテキストスイッチ後に特権レベル 0 のコードセグメント／スタックセグメントセレクタを持つ様に独自 OS が指示している場合でも，軽量仮想計算機モニタは特権レベル 1 のコードセグメント／スタックセグメントセレクタをセグメントレジスタに設定する．

さらに，スタックセグメントレジスタを汎用レジスタにロードする命令[6] を独自 OS が実行する際にも，軽量仮想計算機モニタは，レジスタ値の変換を行う必要がある．しかし，独自 OS が上記命令を実行しても例外は発生せず，軽量仮想計算機モニタは動作できない．そのため，独自 OS を軽量仮想計算機モニタ上で動作させる場合には，上記命令をトラップ命令[7] に置き換え，この変換を行うことにした．

第 2 の問題は，独自 OS から発行される割り込み制御ビット（EFLAGS レジスタの IF ビット）の参照／更新要求の捕捉に関する問題である．軽量仮想計算機モニタは，上記ビットをソフトウェアの代替ビットを用いてエミュレートする．独自 OS からの上記ビット参照／更新要求が発行された場合には，実ハードウェアの状態を参照／更新する代わりに，代替ビットの参照／更新を行う必要がある．

割り込み制御ビットの参照／更新は以下を契機に行われる．

1. cli/sti 命令の発行

[6] コードセグメントレジスタを汎用レジスタにロードすることはできない．

[7] 現在の実装では，使用するトラップ番号は軽量仮想計算機モニタが定義する固定値を使用している．もし，このトラップ番号を独自 OS が使用している場合には，独自 OS 実装者が定義できる構成ファイル等でこのトラップ番号を指定する機能が必要となる．また，独自 OS 実装者は構成ファイルに従って，独自 OS の改変を行わなければならない．

2. タスクスイッチ命令（ljmp 命令）の発行

3. 割り込み／例外ベクタの起動（EFLAGS レジスタの退避）

4. 割り込み／例外ベクタからのリターン命令（iret 命令）の発行（EFLAGS レジスタの回復）

5. pushf/popf 命令の発行

独自 OS が（1）～（3）を実行する際には，例外が発生，軽量仮想計算機モニタが動作して割り込み制御ビットの参照／更新処理のエミュレートを行う．しかし，独自 OS が（4），(5）を実行しても例外は発生せず，軽量仮想計算機モニタによるエミュレートが行えない．そのため，(4），(5）に関しても，スタックセグメントセレクタの汎用レジスタへのロード命令と同様，トラップ命令への置き換えを行うことにした．

第 3 の問題は，例外要因の特定に関する問題である．3.2.2.2 節で述べた様に，軽量仮想計算機モニタ定義の例外ハンドラは，例外発生要因がリモートデバッグ機能起動やデバイスエミュレーションに起因する例外であった場合には，独自 OS に対して例外発生通知を行ってはならない．

デバイスエミュレーションに起因する例外か否かは，例外を起こした命令列を解析することにより判別できる．しかし，リモートデバッグ機能を起動すべき例外か否かは，発生した例外が独自 OS のバグに起因しているか否かで判別すべきであり，厳密な判別は不可能である．現在の実装では，上記判別をデバッグ例外が発生したか否かにより行っている．独自 OS のバグに起因した例外であることを軽量仮想計算機モニタに明示的に通知するため，独自 OS には，カーネルパニック発生時にはデバッグ例外を発行するように命令列の追加が行われている．しかし，独自 OS のバグに起因しているがカーネルパニックルーチンに到達せず，デバッグ例外以外の例外が発生し，かつ独自 OS 定義の例外ハンドラの起動が無限に繰り返されることがある．この場合には，ホストマシン上のソフトウェアデバッガからのブレーク命令の発行により，上記ハンドラの起動を止めることにした．また，デバッグ例外を使用する独自 OS 上で動作するアプリケーション（アプリケーションデバッガなど）は，デバッグ例外が独自 OS に通知されないため使用不可能になるが，本軽量仮想計算機モニタのデバッグ対象外と仮定しているため，問題ないと判断した．

3.2.4.2　軽量メモリ領域保護機能の実装上の問題点

軽量メモリ領域保護機能の実装を行う際には，以下の 2 点の問題を解決する必要が生じた．

- 独自 OS が指示する DMA 転送に伴う軽量仮想計算機モニタ動作領域の破壊の防止．
- 軽量仮想計算機モニタによる独自 OS 定義のプロセス生成，削除の認識方法

以下で，上記 2 点の問題の内容と解決策について述べる．

第 1 の問題点は，独自 OS が指示する DMA 転送に伴う軽量仮想計算機モニタ動作領域の破壊の防止に関する問題である．軽量メモリ領域保護機能は，独自 OS 定義のページテーブルの正当性をチェックすることにより，軽量仮想計算機モニタ動作領域が独自 OS により破壊されることを防いでいる．しかし，本方式は CPU からメモリへの不正な書き込みは防げるものの，独自 OS が制御する高速 I/O デバイスからの DMA 転送に起因する不正な書き込みは防げない．現在の実装では，独自 OS に DMA 転送先領域の正当性をチェックするコードを挿入することにより，この問題を回避している．

第 2 の問題点は，軽量仮想計算機モニタによる独自 OS 定義のプロセス生成，削除の認識方法に関する問題である．独自 OS は軽量仮想計算機モニタに対して実ハードウェアと同様なインタフェースを用いてアクセスする．そのため，軽量仮想計算機モニタは，独自 OS 定義のプロセス生成／削除を認識することは困難である．現在の実装では，以下によりこの問題を回避している．まず，プロセスの生成に関しては，仮想空間の切り替え（プロセス切り替え）が発生する度に，切り替え先プロセスが PDIR として利用している物理ページの PDIR 参照数をチェックする．PDIR 参照数が 0 であれば，軽量仮想計算機モニタは，新規にプロセスが生成されたと判断し，3.2.3.3 節で述べたページマッピング情報の更新を行う．一方，プロセス削除に関しては，独自 OS のコードを改変することで，軽量仮想計算機モニタにその削除を認識させている．具体的には，独自 OS 定義の PDIR 領域の解放を行うコードの直前に，トラップ命令を発行するコードを追加している．

3.2.4.3　考察

3.2.4.1 節と 3.2.4.2 節で述べた解決策では，以下に示す独自 OS の改変が必要になる．

1. スタックセグメントレジスタを汎用レジスタにロードする命令のトラップ命令への置き換え
2. iret/pushf/popf 命令のトラップ命令への置き換え
3. カーネルパニック時にデバッグ例外発行命令を追加
4. DMA 転送領域の正当性をチェックするコードの追加
5. PDIR 領域の解放時にトラップ命令を追加

本節では，これらの改変が独自 OS のバグ混入を引き起こす可能性があるかを検証するため，その改変の複雑さや改変量について考察する．

まず, (1) , (2) については機械的なコードの置き換えで済むので，改変は容易である．また, (3) についても，たとえ正しい位置に命令追加できなくとも，デバッグ環境の安定稼働を阻害しない（正しくデバッグ例外が発生しなくとも，ホストマシンからブレーク命令を発行すればデバッグ可能である）ため，大きな問題にならない．

（4）については，独自 OS で扱っているすべての PCI デバイスドライバのソースコード改変が必要となりやや改変量が多くなる．また，各デバイスドライバごとに，DMA 転送アドレスを指定しているソースコードの部分を捜し出す必要があり，改変もやや複雑となる．しかし．DMA 転送領域を指定するソースコードの部分には，各独自 OS が準備している V to P のアドレス変換ルーチンが呼ばれることが殆んどであること，及び，独自 OS から認識不可能な高位アドレスに存在する軽量計算機モニタ動作領域を DMA 転送先として指定するバグの発生確率が大きくないこと，からこの改変も大きな問題なく実行できると判断している．

（5）については，改変量はトラップ命令の追加のみで済むため大きくないが，独自 OS のページテーブル関連ルーチンの解析が必要となり，改変がやや複雑になる．しかし，移植性の高さを保つことを考えて実装されている独自 OS は，ページテーブル関連ルーチンはコンパクトに実装され

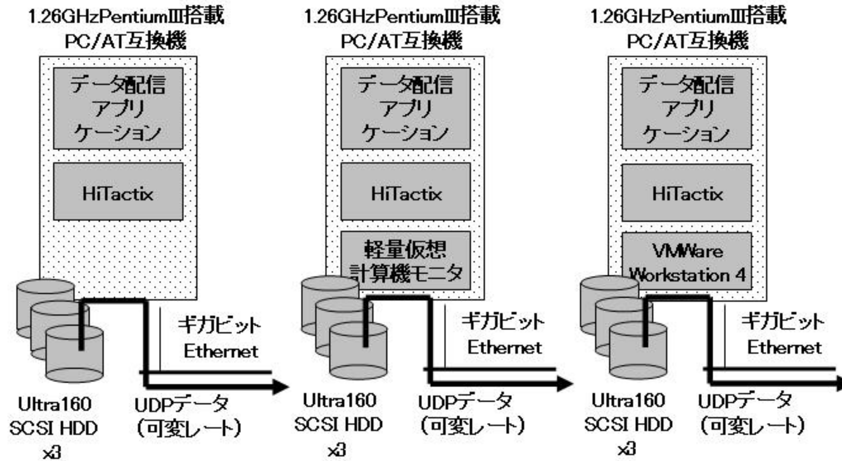

図 3.7: I/O 性能評価のための実験環境

ていることが殆んどであるため，この改変も大きな問題がないと判断している．例えば，BSD/OS バージョン 4.3 の場合，解析対象ルーチンが 2100 行程度であり，かつ改変量は 5 行であった．

3.2.5 性能評価

本節では提案した OS デバッグ方式で使用している軽量仮想計算機モニタの定量的な性能評価結果について説明する．性能評価は，軽量仮想計算機モニタ上で動作する独自 OS の I/O 性能と，プロセス生成／削除の性能について行った．以下では，上記のそれぞれについて，評価方法の概要と評価結果について説明する．

3.2.5.1 I/O 性能の評価

軽量仮想計算機モニタが提供する I/O 性能を評価するために用いた実験環境を図 3.7 に示す．

本実験では，PC/AT 互換機（PentiumIII[8] 1.26GHz 搭載）上に，実ハードウェア上で動作する HiTactix，軽量仮想計算機モニタ上で動作する HiTactix，Linux 版 VMware Workstaton 4[28,

[8] PentiumIII は米国 Intel 社の登録商標です．

33, 34][9] 上で動作する HiTactix を搭載した．VMware workstation 4 は，多様な高速 I/O デバイス上での動作をサポートする PC/AT 互換機向け既存仮想計算機モニタの代表例である．そして，上記の各 HiTactix 上で，データ配信アプリケーションを動作させた．データ配信アプリケーションは，3 つの Ultra160 SCSI ディスクから均等なレートで 2MB ずつデータを読み出し，当該データを 1MB ずつに分割してからギガビット Ethernet に対して UDP 送信する．上記データ配信処理の配信レートを変動させた場合における各 PC/AT 互換機の CPU 負荷の変動を測定し，軽量仮想計算機モニタは VMware Workstation 4 と比べてどの程度高速な I/O 性能を提供できるのか，及び，軽量仮想計算機モニタが提供する I/O 性能は実ハードウェアと比してどの程度低減するのか，について定量的に評価した．さらに，軽量仮想計算機モニタの仮想化オーバヘッド要因（I/O 性能低減要因）の解析も併せて行い，軽量仮想計算機モニタと VMware Workstation 4 との間の提供 I/O 性能に差がつく要因を推定した．仮想化オーバヘッドの測定は，軽量仮想計算機モニタのソースコードに，CPU クロックを用いた実行時間測定コードを追加することにより行った．

CPU 負荷の変動の評価結果を図 3.8 に示す．グラフの横軸はデータ配信レートを，縦軸は配信実行時の PC/AT 互換機の CPU 負荷を示している．各 PC/AT 互換機で消費している CPU 負荷の比率を算出するために，得られた結果から CPU 負荷が 100%に達する時の配信レートを予測し，その配信レートの大小を比較した．得られた配信レートは，実ハードウェアが 6092Mbps，軽量仮想計算機モニタが 1587Mbps，VMware が 294Mbps となった．すなわち，軽量仮想計算機モニタは，Hosted Virtual Machine Monitor architecture を持つ VMware Workstation 4 と比して 5.4 倍の I/O 性能を達成していること，しかし，実ハードウェアと比べると，その I/O 性能は，1/3.8 に低減していることがわかる．

また，650Mbps のレートでデータ配信を実行している際における軽量仮想計算機モニタの仮想化オーバヘッド要因の解析結果を表 3.2 に示す．この表は，各要因ごとに，オーバヘッド比率（独自 OS コード以外を実行している時間全体に対する当該要因処理を実行している時間の割合），実行時間（1 秒あたりの平均総実行時間），実行単価（1 回あたりの平均実行時間），実行回数（1 秒

[9] VMware workstation 4 は米国 VMware Inc. の登録商標です．

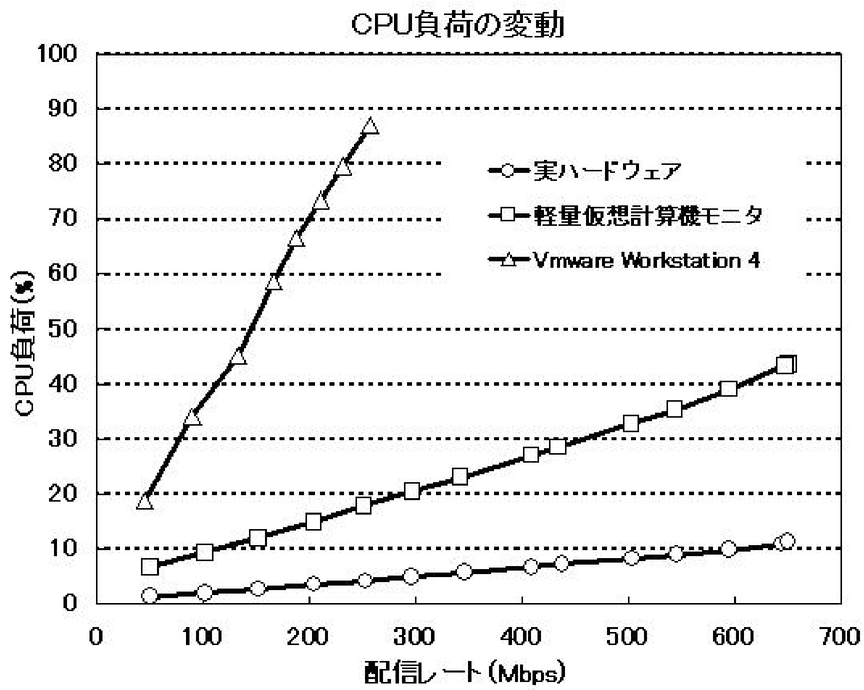

図 3.8: I/O 性能の評価結果

表 3.2: 仮想化オーバヘッド要因の解析

要因	オーバヘッド比率	実行時間 (ms/s)	実行単価 (μ s)	実行回数 (回/s)
モニタ定義例外／割り込みハンドラ起動	32.19%	105.2	0.234	448167
割り込みコントローラエミュレーション	38.50%	125.9	0.549	229076
例外／割り込み通知エミュレーション	11.02%	36.0	9.63	3737
hlt 命令エミュレーション	8.96%	29.3	0.137	213527
空間生成／削除エミュレーション	0.00%	–	–	–
ページテーブル更新エミュレーション	0.00%	–	–	–
その他	9.34%	30.5	–	–

あたりの平均実行回数）を表記している．この解析結果から，軽量仮想計算機モニタ定義の例外／割り込みハンドラの起動オーバヘッド（仮想空間切り替えを実行し，エミュレーション処理を行うハンドラ本体を起動するオーバヘッド），割り込みコントローラエミュレーション，独自 OS への例外／割り込み通知エミュレーション，hlt 命令のエミュレーション[10] がその主要オーバヘッドであることがわかる．独自 OS への例外／割り込み通知エミュレーションはその実行単価が大きいために，それ以外は実行回数が多いために大きなオーバヘッドになっている．一方，データ配信アプリケーション実行中には仮想空間の生成／削除やページテーブル更新は行われず，3.2.3.3 節で述べた処理は実行不要である．

割り込みコントローラエミュレーション，独自 OS への例外／割り込み通知エミュレーション，hlt 命令エミュレーションは VMware Workstation 4 でも実行の必要があり，同等のオーバヘッドが必要であると考えられる．VMware Workstation 4 が軽量仮想計算機モニタよりも低い I/O 性能のしか提供できない理由として，以下が考えられる．

- Host OS 動作中に割り込みが発生すると，仮想計算機モニタ定義の割り込みハンドラ起動のため，仮想空間だけでなく CPU の状態すべてを切り替える World Switch を行う．上記を仮想空間の切り替えだけで実行する軽量仮想計算機モニタと比して，オーバヘッドが大きくなる．

- I/O 要求の発行は独自 OS が行うが，I/O 処理は Host OS のデバイスドライバが実行する．そのため，独自 OS が I/O 要求発行後に Host OS への World Switch が必要になる．また，独自 OS から Host OS へのデータ受渡しのためのリマッピング処理等が必要になる．

- I/O 処理実行は Host OS が行う．たとえ独自 OS が低オーバヘッドで I/O 処理が実行できたとしても，Host OS の高い I/O 処理オーバヘッドが必ず余計にかかる．

[10] hlt 命令のエミュレーションでは，独自 OS のコードが hlt 命令を発行する度に例外を発生させ仮想計算機モニタを起動する．しかし，仮想計算機モニタは直ちにリターンするため，独自 OS は例外発生を繰り返す．割り込みが発生した場合にのみ，独自 OS の実行コンテキスト格納領域に含まれるインストラクションポインタ（EIP）の値を更新し，独自 OS の実行を先に進める．そのため，独自 OS が消費する CPU 負荷が低い程，hlt 命令エミュレーションの負荷は大きく計測される．

表 3.3: コンパイル実行時間の比較

環境	経過時間	合計 実行時間	ユーザ モード 実行時間	スーパバイザ モード 実行時間
実ハードウェア	123.27 秒	108.75 秒	99.50 秒	9.25 秒
軽量モニタ	185.27 秒	171.42 秒	104.03 秒	67.39 秒

- Ethernet コントローラ，ディスクコントローラの I/O デバイスエミュレーションを行っている．
- 実ハードウェアとの完全なバイナリ互換性を保証するために，独自 OS 実行時にバイナリの動的な置き換えが必要であり，このための処理オーバヘッドが余計にかかっている．

3.2.5.2 プロセス生成／削除性能の評価

プロセスの生成／削除を頻繁に繰り返す処理を独自 OS 上で実行させると，3.2.3.3 節で述べたページマッピング情報管理データ構造の更新処理のために，実行性能が低下する．上記更新処理がどの程度の実行性能の低下を招くかを定量的に評価するために，BSD/OS バージョン 4.3 上における，BSD/OS のカーネルソースコードのコンパイル処理を用いた性能評価を行った．カーネルのソースコードのコンパイルは，make，gcc，cpp，cc1，collect2，as，ld などの多数のアプリケーションの起動／終了に伴い，頻繁にプロセスの生成／削除を繰り返す．

本評価実験では，PC/AT 互換機（PentiumIII 1.26GHz 搭載）上に，実ハードウェア上で動作する BSD/OS，及び軽量仮想計算機モニタ上で動作する BSD/OS を搭載する．そしてこの BSD/OS 上で上記カーネルソースコードのコンパイル処理を実行し，両者の実行時間を比較，実行性能の低下の度合を定量的に測定した．また，併せて軽量仮想計算機モニタの仮想化オーバヘッドも測定し，実行性能低下要因の解析も行った．

実行時間の測定結果を表 3.3 に示す．この表では，実ハードウェア及び軽量仮想計算機モニタ上

表 3.4: 仮想化オーバヘッド要因の解析

要因	オーバヘッド比率	実行時間 (ms/s)	実行単価 (μs)	実行回数 (回/s)
モニタ定義例外／割り込みハンドラ起動	53.39%	180.6	0.900	200707
割り込みコントローラエミュレーション	12.01%	40.6	0.650	62421
例外／割り込み通知エミュレーション	6.11%	20.7	6.80	3043
hlt 命令エミュレーション	2.28%	7.8	0.189	40810
空間生成／削除エミュレーション	1.18%	4.0	14.1	284
ページテーブル更新エミュレーション	18.77%	63.5	0.774	81996
その他	6.26%	21.2	–	–

で上記コンパイル処理を実行させた場合における，経過時間（実行完了までに要した時間），合計実行時間（コンパイル処理が CPU を占有した時間の総計），ユーザモード実行時間（合計実行時間のうち，ユーザモードでコンパイル処理が走行していた時間の総計），システムモード実行時間（合計実行時間のうち，スーパバイザモードでコンパイル処理が走行していた時間の総計）を示している．なお，このコンパイル処理中には，2016 回ずつプロセスの生成／削除が行われている．1 秒あたりのプロセス生成／削除回数の平均は，実ハードウェアでは 1 秒あたり 16.4 回，軽量仮想計算機モニタでは 1 秒あたり 10.9 回である．この程度の頻度でプロセスの生成／削除を繰り返す処理を軽量仮想計算機モニタ上で実行させると，実ハードウェア上で実行させた場合と比して実行時間は約 57.6%増大する．この実行時間の増大は，スーパバイザモードで動作する BSD/OS カーネルコードの実行時間の増大が主要要因となっている．

上記コンパイル処理の仮想化オーバヘッドの測定結果を表 3.4 に示す．この測定結果から以下が

明らかになった．

- 仮想空間生成／削除に伴うページマッピング情報管理データ構造の更新処理オーバヘッドは，1 回あたりの平均で 183.5 μ秒程度である（表 3.4 に表記されている実行回数は，仮想空間切り替え時に，切り替え後の仮想空間が新規であるかチェックするための処理起動回数も含んでいる．実際に仮想空間の生成／削除を行っているのは，1 秒あたりでは，284 回のうち 21.8 回[11] だけである）．このオーバヘッドが大きくないのは，オンデマンドページングを行う BSD/OS では，プロセス生成直後に有効な PTBL エントリが少ないためだと考えられる．

- 逆に，ページテーブルの更新に伴うページマッピング情報管理データ構造の更新オーバヘッドは，実行単価は小さいが，実行回数が多いため，全体として大きなオーバヘッドになっている．この実行回数が多い理由は，プロセス生成後にオンデマンドページングに伴うページテーブル更新処理が多発すること，及び BSD/OS では，プロセス生成時に，新規プロセス用のページテーブルとして使用する物理メモリ領域を，一時的に現プロセスの仮想空間にマッピング後初期化する処理が多発するためだと考えられる．

- BSD/OS がページテーブルの更新，または仮想空間の生成／削除を行うと，軽量仮想計算機モニタ定義の例外ハンドラの起動と，エミュレーション処理が行われる．これらのオーバヘッドの総計が，ページマッピング情報管理データ構造の保持に必要なオーバヘッドの総計になる．このオーバヘッドの総計の全仮想化オーバヘッドに対する比率は，$53.39 \times (284 + 81996) \div 200707 + 1.18 + 18.77 = 41.83$ %に達する．すなわち，このデータ構造の保持のために，コンパイル処理の実行時間が $57.6 \times 41.83 \div 100 = 24.10$ %増大していると考えられる．

- 他の主要仮想化オーバヘッドは，割り込みコントローラや hlt 命令エミュレーション，及び上記エミュレーションに先立ち行われる計量仮想計算機モニタ定義の例外／割り込みハンドラ起動である．これらは，ページマッピング情報管理データ構造を持たなくとも必要なオー

[11] 1 秒あたり 10.9 回の生成と削除を行うため，総計で 21.8 回になる

バヘッドである．

3.2.6　本節のまとめ

本節では，デバッグ環境の安定稼働が保証できる，様々なOSやI/Oデバイスに大きな開発なく適用できる，デバッグ時にも高速動作（特に高いI/O性能）が可能である，の3条件を充足するPC/AT互換機上の独自OSデバッグ方式として軽量仮想計算機モニタを用いる方式を新規に提案した．さらに, 提案した軽量仮想計算機モニタにロギング&リプレイ機能を追加するための実装方式を提案した．

提案したデバッグ方式は，従来のソフトウェアリモートデバッグ方式の改良である．従来と異なり，ターゲットマシン上でリモートデバッグ機能も内部に持つ軽量仮想計算機モニタが動作する．軽量仮想計算機モニタは，仮想化対象となるハードウェア資源を，リモートデバッグ機能の安定稼働の保証のために必要な最小限の資源に絞る部分ハードウェアエミュレーション機能を持つ．さらに，低オーバヘッドで仮想計算機モニタと独自OS間のメモリ保護を実現する軽量メモリ領域保護機能も併せて提供する．この2機能により，上記3条件の充足が可能になった．

さらに，実装した軽量仮想計算機モニタの実行性能の評価を行った．その結果，Hosted Virtual Machine Monitorと比べて5.4倍のI/O性能は達成できること，及び1秒間に16.4回のプロセスの生成／削除を繰り返す処理を，実行時間の増大を57.6%に抑えて実現できることが明らかになった．

3.3　軽量仮想計算機モニタを利用したOSデバッガ向けのログ&リプレイ機能の実装方式

3.3.1　ロギング&リプレイ機能

本研究では，3.2節で説明した軽量仮想計算機モニタにロギング&リプレイ機能の追加を目指す．本節では，このロギング&リプレイ機能の概要と機能追加方針について説明する．

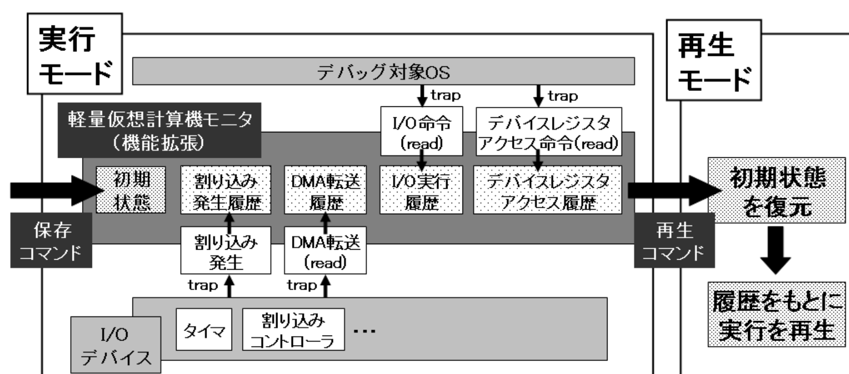

図 3.9: ロギング&リプレイ機能の概要

3.3.1.1 機能概要

ロギング&リプレイ機能とは，デバッグ対象の OS の実行中に，初期状態と実行履歴をログとして残しておき，OS のバグが顕在化した際に，初期状態の回復と当該実行履歴に従った命令レベルでの OS 実行の再生を行う機能である．このような実行再生をサポートすることで，タイミング依存の再現性の低いバグが発生した際にも，何度でも当該バグを再現可能となり，効率的な OS のデバッグが可能になる．

3.3.1.2 機能追加方針

本研究では，デバッグ対象の OS をシングル CPU 上で動作する OS に限定する．マルチコア CPU や複数 CPU 上で動作する OS のロギング&リプレイ機能の実現については今後の課題とする．本研究で実装するロギング&リプレイ機能の概要を図 3.9 に示す．

シングル CPU 上で動作する OS を命令レベルで再生するためには，初期状態と外部（I/O デバイス）からの入力履歴だけを OS 実行時に保存すれば良い．外部（I/O デバイス）への出力等は，初期状態からの命令再実行を再生時に行うだけで実行時と同様の実行を行えるため，履歴として保存する必要はない．本研究で実装するロギング&リプレイ機能では，以下の 4 種類の履歴を保存

する．CPU 能力以外の履歴保存帯域の制約（I/O 帯域制約等）をできる限り取り除くため，これらの履歴を主メモリ上に格納する方針をとる．

I/O ポートアクセス履歴 I/O ポートへの read 実行の結果読み込んだ値の履歴

デバイスレジスタアクセス履歴 PCI デバイスのデバイスレジスタへの read 実行の結果読み込んだ値の履歴

割り込み発生履歴 割り込み発生タイミングと，発生した割り込み番号の履歴

DMA 転送履歴 DMA write 転送の発生タイミングと，転送されたデータの履歴

本研究で実装する OS デバッガからのロギング&リプレイ機能の起動手順は以下になる．

1. ホストマシン上のソフトウェアデバッガから保存コマンドを実行する．この結果，ターゲットマシン上の OS の初期状態（メモリやレジスタのスナップショット）が保存される．
2. ホストマシン上のソフトウェアデバッガから実行継続コマンドを実行する．ターゲットマシン上の OS は実行モードで動作を開始する．この間，上記履歴を軽量仮想計算機モニタ内部に保存する．
3. タイミング依存のバグ等が発生した場合，ターゲットマシン上の OS は例外を発生し，実行を停止する．ホストマシン上のソフトウェアデバッガから回復コマンドを実行すると，OS のメモリ／レジスタ状態が初期状態に回復する．
4. ターゲットマシン上のソフトウェアデバッガから実行継続コマンドを実行する．今度は，OS は再生モードで動作を開始し,（2）で保存した履歴を参照しながら,（2）での実行を忠実に再現する．そして,（3）と同じタイミングで例外が発生し，実行が停止する．

3.3.2　ロギング&リプレイ機能の実現上の課題

本節では，3.3.1 節で述べたロギング&リプレイ機能を，3.2 節で述べた軽量仮想計算機モニタ上に実現する際の課題について述べる．

3.3.2.1　実現に向けた課題

3.1 節で述べた通り，本研究の目標は軽量仮想計算機モニタへのロギング&リプレイ機能の追加実装である．

また，3.3.1.2 節で述べた通り，仮想計算機モニタにロギング& リプレイ機能を実装するためには，I/O デバイスからの入力履歴（I/O ポートアクセス履歴，デバイスレジスタアクセス履歴，割り込み発生履歴，DMA 転送履歴）の保存・再生の実現ができれば良い．

従来の仮想計算機モニタを用いた I/O 処理は，仮想計算機モニタが I/O デバイスのエミュレータ（仮想 I/O デバイス）を保持し，OS のデバイスドライバ（以下，「OS ドライバ」と略す）は仮想 I/O デバイスに対して I/O 要求を発行する．仮想計算機モニタ（デバイスエミュレータ）が，物理デバイスを介した I/O 処理と，OS ドライバへの通知処理を行う．OS ドライバに対する受信データや I/O 完了の通知はすべて仮想計算機モニタが制御しているため，自モジュールの動作履歴の保存と当該履歴を用いた自モジュール動作の再生処理のみで履歴の保存・再生が実現できた．

しかし，軽量仮想計算機モニタを用いた I/O 処理は，OS ドライバが直接物理 I/O デバイスに要求を発行し，物理 I/O デバイスが受信データや I/O 完了通知を OS ドライバに対して行う．I/O 処理中に軽量仮想計算機モニタが動作する保証もない[12]．そのため，従来の履歴の保存・再生方式では，OS ドライバと物理 I/O デバイスとの間でやりとりされる動作履歴の保存・再生を軽量仮想計算機モニタが行うことはできず，新規の方式設計が必要になる．特に，以下の課題解決の可否の検討が必要となる．

動作契機付与　軽量仮想計算機モニタによる履歴の保存・再生には，OS ドライバと物理デバイス

[12] 従来方式と同様に軽量仮想計算機モニタ内部にデバイスエミュレータを保持させれば動作契機の保証はできるが，本 OS デバッガの目標（2）の充足はできなくなる．

との間のI/O処理実行中に当該モニタの動作契機が必要になる．この動作契機を確実に与える方式を設計する．

OSドライバの動作追跡 再生動作に必要十分な履歴を収集するために，OSドライバの動作を軽量仮想計算機モニタが追跡する機構も新規に設計する．例えば，NICからの受信処理の履歴を保存・再生するためには，OSドライバと物理I/Oデバイスとの間で共有しているデータ構造（NICの受信ディスクリプタ等）を軽量仮想計算機モニタがスヌープして，OSが受信したデータの内容や，OSがI/O完了通知を受けたタイミングの検出などをしなければならない．

デバイス依存処理の極小化・分離 OSドライバの動作追跡などはデバイス依存処理であり，軽量仮想計算機モニタ層にデバイス依存コードを実装しなければならない．しかし，本コードサイズや開発工数が大きくなると，本OSデバッガの目標（2）の充足が難しくなる．そのため，デバイス依存処理の極小化，及び，当該コードの分離による新規デバイスへの対応の容易化が必要となる．

以下の節では，3.3.1.2節であげた各履歴に関して，仮想計算機モニタによる履歴の保存や再生を行うための動作契機の付与方式や，OSドライバの動作追跡に基づいた履歴の保存・再生方式の設計・実装が容易であるか否か，及び各履歴の保存・再生にデバイス依存処理があるか否を検証する．

3.3.2.2 I/Oポートアクセス履歴の保存・再生

Intel VT[35]などのCPU機構を利用すれば，OSがI/Oポートへのアクセス命令を発行した際に仮想計算機モニタを起動でき，以下の方式でI/Oポートアクセス履歴の保存・再生をデバイス非依存の処理で容易に実現できる．実行モードにて起動された仮想計算機モニタは，当該アクセス命令を実行してI/Oポートからreadした値をレジスタに格納する．併せて，readされた値を履歴に保存する．また，再生モードでは，アクセス命令を実行する代わりに，保存した履歴から

read されるべき値を読み出しレジスタに格納する．

3.3.2.3　デバイスレジスタアクセス履歴の保存・再生

3.2 節で示した OS デバッガを用いた場合，NIC や HBA などの I/O デバイスレジスタへのアクセスは，OS が仮想計算機モニタを介さずに行う．PCI デバイスレジスタアクセスはメモリアクセス命令（mov 命令など）にて行われるため，デバイスレジスタのアクセス履歴の保存・再生には，軽量仮想計算機モニタの初期化時に当該デバイスレジスタ領域のページを不在化し，メモリアクセス命令実行時にページ例外を強制発行させれば良い．これにより軽量仮想計算機モニタの動作契機が保証され，I/O ポートと同様のアルゴリズムで履歴の保存・再生ができる．

しかし，デバイスレジスタアクセス領域を取得し，該当するページを不在化する処理は，PCI 構成空間から読み込んだデバイス ID の識別などの I/O デバイス依存の処理を含む．そのため，本履歴の保存・再生におけるデバイス依存処理の極小化と分離が課題となる．

3.3.2.4　割り込み発生履歴の保存・再生

Intel VT などの CPU 機構を利用すれば，実行モードにおいて割り込みが発生した際に仮想計算機モニタを起動でき，以下の方式で割り込み発生履歴の保存・再生をデバイス非依存の処理で容易に実現できる．実行モードにて起動された仮想計算機モニタは，発生した割り込み番号，割り込み発生時の命令ポインタ，累積実行命令カウンタの値を読み込み履歴に残す．再生モードでは，仮想計算機モニタは，割り込み禁止状態で OS を動作させる．そして，履歴に記載された命令ポインタにブレークポイントを設定し，当該命令実行時にブレークポイント例外により仮想計算機モニタを起動させる．起動された際に，累積実行命令カウンタが履歴に記載された値と同じであるか否かを判別し，同じであれば，履歴に記載されている番号の割り込み発生を OS に通知する．

3.3.2.5　DMA 転送履歴の保存・再生

通常の仮想計算機モニタは，OS が認識する DMA 転送そのものをソフトウェアがエミュレートするため，DMA 転送履歴の保存・再生は容易である．しかし，提案する OS デバッガを用いた場合，NIC や HBA などの I/O デバイスからの DMA 転送は，OS ドライバが仮想計算機モニタを介さずに制御する．そのため，提案する OS デバッガでは，OS が制御するハードウェア機構を用いて行われる DMA 転送状況を仮想計算機モニタがスヌープし，その結果を元に履歴を生成・保存する（もしくは，履歴をもとに DMA 転送を再生する）必要がある．また，このスヌープを行える契機は，OS によるデバイスレジスタアクセスと割り込み発生に限られる．そのため，OS ドライバの動作のモデル化と，当該モデルに基づく履歴の保存・再生機構の設計が必須となる．また，上記スヌープ処理はデバイス依存処理となるため，デバイス依存処理の極小化と分離が課題となる．

以上より，

1. デバイスレジスタアクセス履歴の保存・再生におけるデバイス依存処理の極小化・分離方式
2. DMA 転送履歴の保存・再生の全体実現方式（当該保存・再生処理における軽量仮想計算機モニタの動作契機付与方式，OS ドライバの動作追跡及び当該追跡結果を利用した履歴の保存・再生方式，デバイス依存処理の極小化・分離方式）

について詳細設計が課題となることがわかる．これらの課題の解決方式を次節で述べる．

3.3.3　課題の解決方式

本節では，3.3.2 節で述べたロギング&リプレイ機能の実現上の課題の解決方式として，デバイスレジスタアクセス履歴の保存・再生におけるデバイス依存処理の極小化・分離方式，及び DMA 転送履歴の保存・再生の実現方式を説明する．

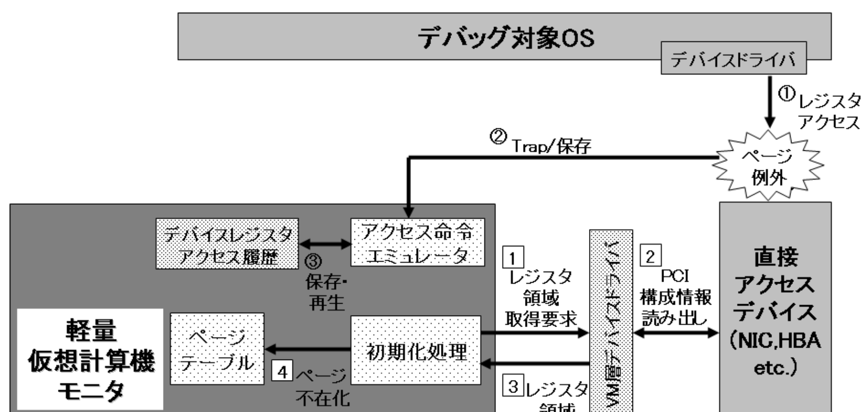

図 3.10: デバイスレジスタアクセス時の仮想計算機モニタ機能方式

3.3.3.1 デバイスレジスタアクセス履歴の保存・再生方式

3.3.2 節で述べた通り，デバイスレジスタアクセス履歴の保存・再生のために，軽量仮想計算機モニタの初期化時に PCI デバイスのレジスタ領域を不在にする．そして，OS ドライバがデバイスレジスタにアクセスした際に軽量仮想計算機モニタの動作契機を与え，軽量仮想計算機モニタが，I/O ポート履歴の保存・再生と同様の手法でデバイスレジスタアクセス履歴の保存・再生を行えるようにする．

上記履歴の保存・再生処理におけるデバイス依存処理の極小化と分離を行うため，図 3.10 に示す通り，VM 層デバイスドライバ（OS 層ドライバとは別のドライバ）を新規に実装し，当該ドライバにてデバイス依存処理を実現した．

初期化時のデバイスレジスタ領域不在化のために，PCI 構成情報を読み出し，デバイスレジスタ領域のアドレス取得を行う．本 PCI 構成情報取得処理はデバイス依存処理であるため，軽量仮想計算機モニタとの間に PCI 構成情報取得インタフェースを作成し，VM 層デバイスドライバにて処理本体を実装する．一方，履歴の保存・再生のための命令エミュレーションや read された履歴の保存処理などはデバイス非依存処理であるため，VM 層ドライバではなく仮想計算機モニタ内部に実装し，デバイス依存部分を極小化する．

前節で述べた通り，VM 層デバイスドライバのコード規模や開発工数が，本研究で実装する OS デバッガの汎用性に大きく影響する．この解析結果については 3.3.4 節で述べる．

また，上記方式によるデバイスレジスタアクセス履歴の保存・再生機能を提案する OS デバッガに追加することで，OS によるデバイスレジスタアクセスのたびの仮想計算機モニタ起動が余計に必要になり，OS の I/O 性能が大きく低減する懸念がある．I/O 性能劣化についての定量的な評価結果については 3.3.5 節で述べる．

3.3.3.2 DMA 転送履歴の保存・再生方式

3.3.2 節で述べた通り，DMA 転送履歴の保存・再生には，OS ドライバ（NIC ドライバ）のモデル化と，当該モデルに基づく履歴の保存・再生機構の設計が必要となる．特に NIC 受信時の DMA 転送は，OS ドライバの指示ではなく，OS ドライバとは独立のタイミングで DMA 転送が起動されるため，この設計が複雑になる．本節では，NIC の受信時の DMA 転送の保存・再生の実現方式について説明する．まず，提案する OS デバッガで仮定した NIC ドライバの動作モデルを説明した後に，当該モデルに基づいた履歴の保存・再生手順，及びその手順を実現するために VM 層デバイスドライバに実装すべきデバイス依存部分について説明する．

3.3.3.2.1 NIC ドライバのモデル化

提案する OS デバッガで想定している NIC ドライバの受信時の動作モデルを策定するため，筆者らが開発した HiTactix[6][7][13] で実装されている PCI バスに接続可能な NIC のドライバ実装コードの解析を行った．その結果，これらの NIC ドライバの受信処理は図 3.11 に示すフローでモデル化できることがわかった．以下のモデルの説明では，図 3.12 に記載されている受信ディスクリプタの使用を仮定する（HiTactix でデバイスドライバを実装している NIC は，本ディスクリプタと同じ，もしくは同等のデータ構造を利用している）．受信ディスクリプタは，DMA 転送先アドレス，サイズの他に，受信済みインデックスレジスタ，完了ビットを保持する．ディスクリプタのエントリは，OS ドライバが reader，NIC が writer の共有リングバッファになっている．OS ド

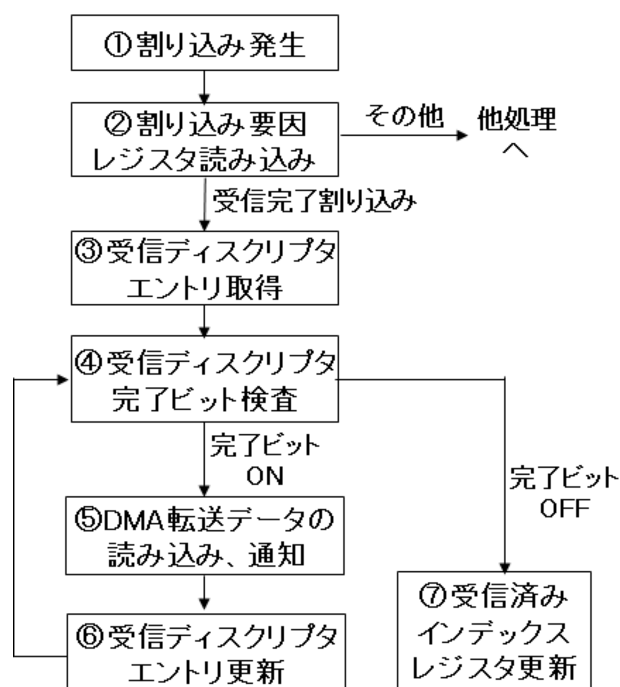

図 3.11: NIC ドライバ受信時の動作モデル

ライバは，受信処理を完了した最終エントリを受信済みインデックスレジスタに格納することで，NIC に read 状況を通知する．一方，NIC は，当該エントリへの受信データの DMA 転送が完了すると完了ビットを ON にすることで，OS ドライバに write 状況を通知する．

NIC ドライバの受信時の動作手順を以下に示す．

1. 割り込み発生により NIC ドライバが起動する．
2. 割り込み要因レジスタを読み込み，割り込み要因が受信完了かそれ以外であるかを判別する．受信完了であれば処理を継続する．
3. 受信ディスクリプタのエントリを順に参照し,（4）～（6）の処理を行う．
4. 完了ビットが立っていれば（5）に，立っていなければ（7）にジャンプする．
5. データ受信をドライバの上位層に通知する．

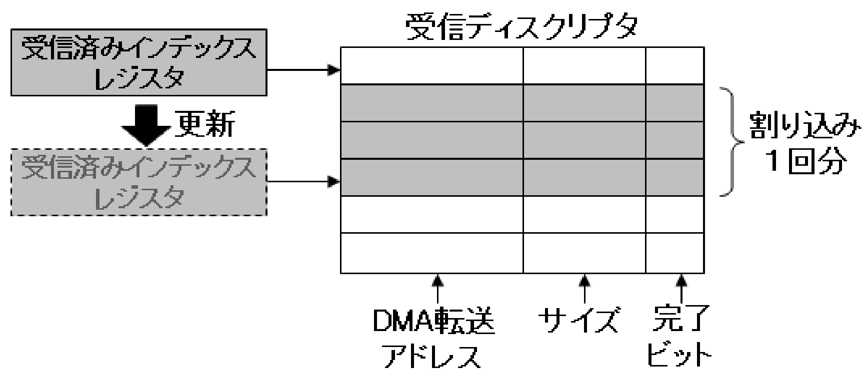

図 3.12: 受信ディスクリプタの構成

6. 新しい DMA 転送アドレス（受信バッファアドレス），サイズを設定する．完了ビットをクリアする．

7. 受信済みインデックスレジスタを更新する．

上記モデル化により，以下の 2 点が明らかになる．次節で述べる履歴の保存・再生方式では，この 2 点を仮定する．

- DMA 転送データは，割り込み要因レジスタの参照後，かつ，受信済みインデックスレジスタの更新前に読み込まれる．また，これら 2 つの処理の間には仮想計算機モニタの起動契機はない．すなわち，履歴の保存は受信済みインデックスレジスタ更新時に，1 回の割り込みで処理した DMA データの転送履歴をまとめて保存する必要がある．また，割り込み要因レジスタの参照時に，1 回の割り込みで処理した DMA データの転送をまとめてエミュレート（履歴を用いて再生）する必要がある．

- 受信ディスクリプタの更新内容と受信済みインデックスレジスタの更新履歴をたどることで，1 回の割り込み処理でデータ受信を通知した DMA 転送データのアドレス，サイズ群を取得できる．

3.3.3.2.2　履歴の保存・再生方式

DMA 転送履歴を用いた再生のために，以下を履歴に保存する．

- 割り込み要因レジスタの累積参照回数
- DMA 転送アドレス
- DMA 転送サイズ
- DMA 転送データ

これらの情報があれば，割り込み要因レジスタ参照を契機に起動した仮想計算機モニタが以下を実行すれば，DMA 転送のエミュレート（履歴を用いた再生）が実現できる．

1. 累積参照回数が一致する履歴のエントリ群を検索する[13]．
2. 上記エントリ群に格納されている DMA 転送アドレスに，格納されている DMA 転送データを，格納されているサイズ分だけコピーする（DMA 転送本体のエミュレート）．
3. 受信ディスクリプタエントリの DMA 転送サイズフィールドに履歴に格納されている値を設定する．併せて同エントリの完了ビットを設定する（受信ディスクリプタの更新のエミュレート）．

以下では，実行モードにおいて上記各情報を取得・保存する方法，及び当該方法の処理のうち VM 層デバイスドライバに実装すべきデバイス依存部分について説明する．

割り込み要因レジスタの累積参照回数 仮想計算機モニタは初期化時に割り込み要因レジスタアドレスを取得する．3.3.3.1 節で述べた方式により，OS が割り込み要因レジスタを参照した際

[13] 本エミュレーション方式では, 割り込み要因レジスタの参照直後に 1 回の割り込みで処理した DMA データの転送をまとめてエミュレートする．仮定した NIC ドライバの動作モデルでは,1 回の割り込み処理につき 1 度の割り込み要因レジスタの参照を行うため, 割り込み要因レジスタの累積参照回数から, 対応する DMA データの転送が処理された割り込み処理, すなわち DMA 転送のエミュレートの実行タイミングが特定できる．

に仮想計算機モニタを起動できる．仮想計算機モニタは，起動の際にページ例外発生アドレスと初期化時に取得したアドレスと比較することで，当該レジスタへの累積参照回数を管理する．保存時には，現在の累積参照回数を履歴にコピーする．

DMA 転送アドレス DMA 転送アドレスの取得及び履歴への保存は，受信済みインデックスレジスタ更新時に受信ディスクリプタを参照しても実現できない．3.3.3.1 節で示した通り，受信済みインデックスレジスタ更新（3.3.3.1 節のステップ 7）前に，OS ドライバは，対応するエントリの DMA 転送アドレスを新しいアドレスに更新（3.3.3.1 節のステップ 6）するためである．

提案する OS デバッガでは，仮想計算機モニタ内にシャドウ受信ディスクリプタ（DMA 転送アドレスフィールド部分のみ）と，シャドウ受信済みインデックスレジスタを保持する．軽量仮想計算機モニタは，初期化時に受信ディスクリプタと受信済みインデックスレジスタのコピーを作成する．

受信済みインデックスレジスタ更新時に，シャドウ受信済みインデックスレジスタと更新後の受信済みインデックスレジスタの値を比較し，対応する割り込みにて受信処理が行われた受信ディスクリプタエントリを取得する．そして，シャドウ受信ディスクリプタから，DMA 転送アドレスを取得し，履歴に保存すると共に，新しい DMA 転送アドレスをシャドウ受信ディスクリプタに設定する．

DMA 転送サイズ DMA 転送サイズも，DMA 転送アドレスの項で述べた理由で，受信済みインデックスレジスタ更新時に受信ディスクリプタを参照しても取得できない．代わりに，提案する OS デバッガでは，取得した DMA 転送アドレスに格納されている受信データのヘッダを解析し，受信データサイズを算出，算出した値を履歴に保存する．

DMA 転送データ 上記方式で取得した DMA 転送アドレス，DMA 転送サイズを元に主メモリ上に配置されている DMA 転送データを取得，履歴に保存する．

上記履歴の保存・再生方式において，デバイス依存部分を極小化するため，以下の方針で軽量仮想計算機モニタとVM層デバイスドライバのインタフェースを設計する．

- 割り込み要因レジスタの累積参照回数を取得するため，当該レジスタアドレス取得インタフェースを設ける．アドレス取得処理はVM層デバイスドライバで行うが，取得できたアドレスとページ例外アドレスとの比較や累積参照回数の管理は軽量仮想計算機モニタが行う．
- DMA転送アドレスを保存するため，受信済みインデックスレジスタアドレス・格納値の取得，ディスクリプタエントリの取得・更新など，シャドウ受信ディスクリプタとシャドウ受信済みインデックスレジスタの管理に必要なインタフェースを設ける．VM層デバイスドライバでは指定情報の取得処理のみを行い，シャドウの管理は軽量仮想計算機モニタが行う．

3.3.4 実装の詳細

本節では，実装したOSデバッガの構成，実装上の課題，VM層デバイスドライバの実装容易性の解析結果について示す．

3.3.4.1 構成

本研究で実装したシステムの構成を図3.13に示す．本システムで実装した軽量仮想計算機モニタは，Intel VT機構を持つCPU（Intel社Xeon）上で動作する．既存実装機能は，3.1節で述べた軽量仮想計算機モニタに記載した機能を実装している（但し，Intel VT機構を利用した本バージョンでは，軽量メモリ保護機能を実装していない）．そして，履歴保存機能と再生機能を追加で実装している．さらに，VM層ドライバとして，Intel社NICのドライバを実装している．本ドライバの実装容易性の解析結果は3.3.4.3節に示す．軽量仮想計算機モニタは，Intel社Xeon[14]と完全互換なハードウェアインタフェースをOSに提供している．但し，アドレス変換機構（及び軽量メモリ保護機能）のみはIntel VT機構でサポートされていないため実装していない．本軽量

[14] Xeonは米国Intel社の登録商標です

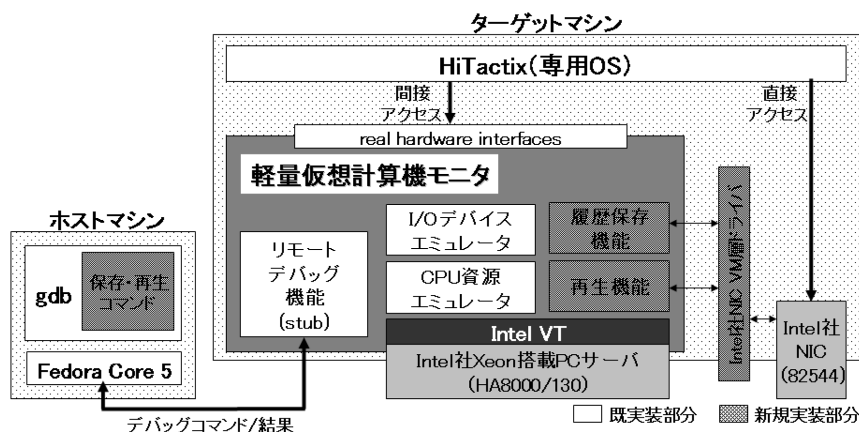

図 3.13: 実装システムの構成

仮想計算機モニタ上で動作する OS がページ変換機構を利用しないことを前提としている．現在の実装では，軽量仮想計算機モニタ上に，筆者らが開発した専用 OS HiTactix を動作させている．ターゲットマシン上では，Fedora Core 5 上で gdb（Gnu Debugger）を動作させている．gdb には，3.3.1.2 節で述べた保存・再生コマンドを追加で実装している．

3.3.4.2 実装上の課題

Intel 社 NIC の VM 層ドライバの実装にあたり，以下の二つの課題に直面した．

- 受信エラーの再生
- 送信完了通知のエミュレート

以下では，上記課題とその解決方式の概要について説明する．

3.3.4.2.1 受信エラーの再生

3.3.3.2.2 節で示した通り，提案した OS デバッガの軽量仮想計算機モニタは，受信ディスクリプタ更新のエミュレートのため，受信ディスクリプタエントリの DMA 転送サイズフィールド，完了

ビットの設定を行う．

しかし，Intel 社 NIC（及び他の多くの NIC）は，受信処理時にチェックサムエラー等の受信エラーが発生した際に，エラー発生を通知するエラーフィールドのビットを立てる．そのため，軽量仮想計算機モニタによる再生処理は，DMA 転送サイズフィールド，完了ビット以外に，エラーフィールド更新のエミュレートも行わねばならない．しかし，3.3.3.2.1 節で述べた NIC 動作モデルの割り込み要因レジスタ読み込み後に，NIC が本フィールドを更新する可能性がある（当該レジスタ読み込み後に NIC が行った受信処理にてエラーが発生した場合）．一方，受信済みインデックスレジスタ読み込み時には，ドライバは当該受信ディスクリプタエントリを再初期化した後であるため，実行モードにて軽量仮想計算機モニタは当該フィールドの内容を読み込み，履歴に保存することができない．

現在の実装では，エラー発生時に OS ドライバがディスクリプタエントリに設定されている DMA 転送アドレスを更新しないこと，及び OS ドライバは，通知されたエラーの種類により処理を変えないことを仮定している．そして，実行モードにおける受信済みインデックスレジスタ更新時に，受信ディスクリプタエントリの DMA 転送アドレスの更新が行われていないことを軽量仮想計算機モニタが検知した場合に，特定の種類の受信エラーが発生したと認識し，履歴に保存する．再生モードにて，当該履歴に従い特定の種類の受信エラー発生を通知するために受信ディスクリプタの更新を行う．

より厳密な受信エラーの再生には，OS ドライバとより密な連携が必要である．この実現は今後の課題とする．

3.3.4.2.2　送信完了通知のエミュレート

OS ドライバにより送信要求が発行されると，NIC は送信処理を開始する．そして，送信処理完了時に，NIC が OS ドライバに完了を通知する．本通知は，デバイスレジスタ（送信済みインデックスレジスタ）への値の設定により行われる場合と，送信ディスクリプタの完了ビット更新により行われる場合がある．

前者の場合は，OS ドライバによるデバイスレジスタアクセス時に仮想計算機モニタが起動できるため，実行モードにおけるデバイスレジスタのアクセス履歴の保存，及び再生モードにおけるアクセスの再現のみで，上記送信完了通知をエミュレートできる．しかし，後者の場合は，送信ディスクリプタ領域全体をページ不在にしない限り，完了ビットアクセス時に仮想計算機モニタの起動が行えず，送信完了通知のエミュレートを行えない．そして，送信ディスクリプタ領域全体のページ不在化は，他の DMA 転送アドレスフィールド設定等のたびにページ例外発生を引き起こし，大きな I/O 性能劣化を招く．

現在の実装では，VM 層デバイスドライバが，初期化処理時に，対象 NIC が上記どちらのモードで動作するかを軽量仮想計算機モニタに通知している．そして，軽量仮想計算機モニタは，上記モードによって送信完了通知のエミュレート方法を変える．実装した Intel 社 NIC の VM 層ドライバは後者のモードでの動作を選択しているため，送信性能の大きな劣化が発生している．本性能劣化の回避方法も今後の課題とする．

3.3.4.3　VM 層デバイスドライバの実装容易性

VM 層デバイスドライバは，NIC からの DMA 転送の履歴を保存・再生するために表 3.5 に示すインタフェースを提供する．どのインタフェースも，PCI 構成空間レジスタ，デバイスレジスタ，ディスクリプタ領域の読み込み，または更新の組み合わせのみで実現でき，内部状態管理は一切必要ない．

Intel 社 NIC の VM 層デバイスドライバを実装したところ，約 600 行（コメント行を含む）程度で実装できた．OS ドライバ約 14,000 行と比較すると，その 4.3% の追加コードの開発でロギング&リプレイ機能を利用可能になっており，大きな追加開発は必要ない．

3.3.5　性能評価

本節では，本研究で実装したロギング&リプレイ機能にて，履歴の保存・再生に要するオーバーヘッドを定量的に評価する．そして，提案方式による履歴の保存・再生がどの程度の現実的なオー

表 3.5: VM 層デバイスドライバの提供インタフェース

I/F 名	概要
Init	VM 層デバイスドライバの初期化処理を行う．PCI 構成空間から自デバイス ID を検索し，当該デバイスのデバイスレジスタ領域のリスト，割り込み要因レジスタのアドレス，受信済みインデックスレジスタアドレス，送信完了通知の動作モードをリターンする．
GetDescRange	送信/受信ディスクリプタ領域をリターンする．
GetRxIndexReg	受信済みインデックスレジスタの値をリターンする．
GetRxMaxIndex	受信済みインデックスレジスタの最大値（受信済みインデックスレジスタのエントリ数）をリターンする．
GetRxDmaAddr	指定インデックスを持つ受信ディスクリプタエントリの DMA 転送先アドレスをリターンする．
SetRxUpdateEntry	指定インデックスを持つ受信ディスクリプタエントリの DMA 転送サイズ，完了ビットフィールドを設定する．

バーヘッドで実現可能であるかを明らかにする．オーバーヘッドの定量評価は，履歴の保存・再生に要する CPU 負荷とメモリ消費量の 2 つについて行う．

3.3.5.1 CPU 負荷の評価

履歴の保存・再生に要する CPU 負荷を測定するため，図 3.14 に示す評価環境で評価を行った．本環境では, (1) 履歴の保存機能がない軽量仮想計算機モニタ, (2) 履歴の保存機能を持つ軽量仮想計算機モニタ, (3) VMware Workstation 6[33]（Linux 版）上で，我々が開発した専用 OS HiTactix, 及び UDP 受信プログラムを動作させる．そして，UDP 受信プログラムに対して，他のサーバ上で動作する UDP 送信プログラムから可変レートで UDP パケットを送信する．UDP パケットの送信レートを変動させた際のそれぞれの仮想計算機モニタが動作するサーバの CPU 負荷の変動を測定することで，ネットワーク受信処理の履歴保存に要する CPU 負荷の評価を行った[15]．また，履歴保存機能つき及び履歴保存機能なしの軽量仮想計算機モニタと従来の仮想計算機モニタのネットワーク受信性能の比較も行った．

測定結果を図 3.15 に示す．図中の横軸は UDP 送信プログラムによる UDP パケットの送信レー

[15] HiTactix ではアイドルスレッドが消費する CPU 時間を利用してシステム全体の CPU 使用率を算出する．アイドルスレッド走行中に軽量仮想計算機モニタが動作した場合，当該モニタが動作したことが HiTactix から認識できないため，当該モニタの走行時間もアイドルスレッドの走行時間の一部としてカウントされる．これによる測定誤差を防ぐため，本測定では，(1) アイドルスレッド走行時間の総計，(2) アイドルスレッド実行中の軽量仮想計算機モニタの走行時間の総計，(3) それ以外のスレッド実行中の軽量仮想計算機モニタの走行時間の総計を計測し，(1) から (2) を引いた時間をシステムがアイドル状態になっている時間の総計と判定した．

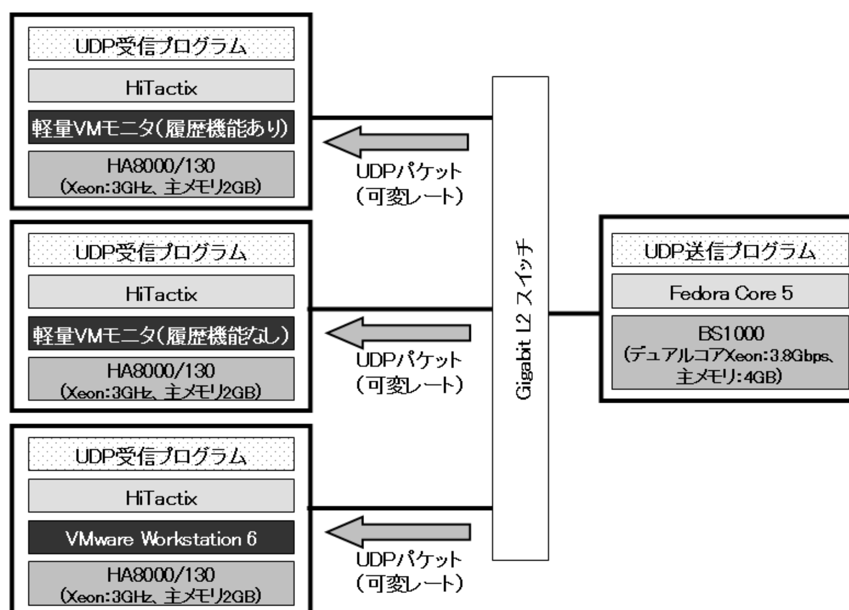

図 3.14: 評価環境

トを，縦軸がそれぞれの仮想計算機モニタが動作するサーバの CPU 負荷を示す．本結果から以下が明らかになった．

1. 履歴保存を行うことで，ネットワーク受信に要する CPU 負荷が約 3.5 倍に上昇する．

2. 履歴保存を行わない軽量仮想計算機モニタは，従来の仮想計算機モニタ（VMware）の約 38%の CPU 負荷でネットワークの受信処理が実現できる．一方，履歴保存を行う軽量仮想計算機モニタは，VMware よりネットワーク受信処理に要する CPU 負荷が約 38%上昇する．

上記（1）の要因を分析するため，履歴保存機能を持つ軽量仮想計算機モニタと持たないモニタを使用しつつ 400Mbps の UDP パケットを受信した場合について，軽量仮想計算機モニタ，HiTactix, UDP 受信プログラムが消費する CPU 使用率の内訳を測定した．測定は以下により行った．

- 軽量仮想計算機モニタの起動時と終了時に CPU クロックを取得するコードを埋め込み，計量仮想計算機モニタの起動要因ごとに，起動回数，及び消費 CPU 使用率を求めた．

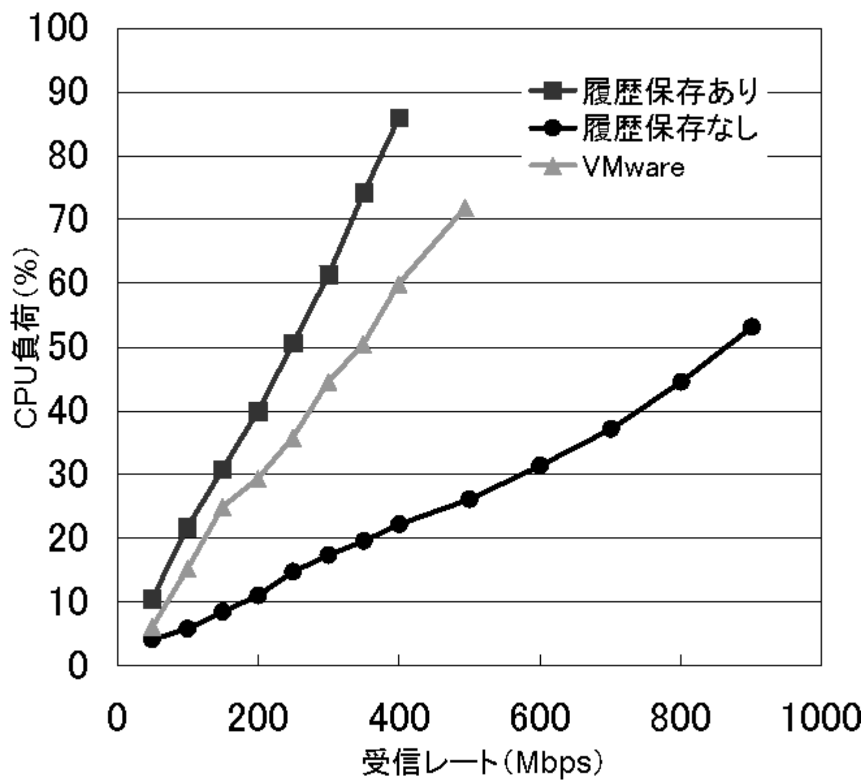

図 3.15: CPU 負荷の評価結果

- 上記計量仮想計算機モニタの起動/終了時に，Intel VT が提供する VMexit/VMresume 命令の実行，及び VMread/VMwrite 命令を用いたコンテキストの退避/回復も実行される．上記起動回数にこれらの命令の実行時間をかけ，消費 CPU 使用率を求めた（命令の実行時間はベンチマークプログラムを作成し実測した．VMexit/VMresume 命令合計で約 3.22 マイクロ秒，VMread/VMwrite 命令によるコンテキストの退避/回復合計で約 6.65 マイクロ秒の実行時間を消費するとの結果を得た）.

- 上記から軽量仮想計算機モニタの起動/走行/終了に要する CPU 使用率の総計が求まる．全体の CPU 使用率より上記 CPU 使用率の総計を引いた値を UDP 受信プログラム及び OS（HiTactix）が消費する CPU 使用率と推定した.

測定結果を表表 3.6 に示す．この結果より以下がわかった.

- デバイスメモリのアクセス履歴保存処理により，CPU 使用率が 56.44%上昇している．これは全体の CPU 使用率上昇量の約 89%に相当し，CPU 使用率上昇の主要因だと言える．なお，この 56.44%の上昇のうち，VMexit/VMresume/ VMread/VMwrite 命令実行により 33.98%（(3.22+6.65）マイクロ秒× 34430 回=339.8 ミリ秒)，DMA 転送データ履歴保存のためのコピー処理により 13.25% の CPU 使用率上昇が起こり，これらの処理時間を低減できなければ，本 CPU 使用率上昇を低減できない.

- 上記の他に，CPU クロックの読み出し履歴保存[16] により約 4.85%の CPU 使用率上昇が起こる．さらに，UDP 受信プログラム・HiTactix による CPU 使用率も約 2.59%上昇している．後者は，頻繁な VMexit/VMresume 命令実行によるキャッシュヒット率低下が要因と考えられる.

また，上記（2）の要因を推測するため，表 3.6 の測定結果から VMware による受信処理の CPU 使用率の内訳を以下の方法で推定した．推測結果も表 3.6 に追記してある.

[16] HiTactix の場合，OS 実行中に rdtsc 命令を実行する．実行モードで読み出した値を再生モードで読み出し可能にするため，軽量仮想計算機モニタは本命令の実行結果も履歴に保存している

表 3.6: CPU 使用率の内訳の測定結果

処理	履歴保存機能あり（実測）		履歴保存機能なし（実測）		VMware（推測）	
	起動回数（回/s）	CPU 使用率（%）	起動回数（回/s）	CPU 使用率（%）	起動回数（回/s）	CPU 使用率（%）
デバイスメモリアクセス&DMA 転送履歴保存	34430.0	56.44%	-	-	21282.2	26.70%
（うち DMA 転送履歴保存のためのデータコピー）	(33929.2)	(13.25%)	-	-	-	-
割り込み発生履歴保存・割り込み通知 emulation	1839.1	2.12%	1754.7	2.08%	1839.1	2.12%
I/O ポートアクセス履歴保存・ICU emulation	8153.8	8.62%	8305.7	8.97%	8153.8	8.62%
CPU クロック読み出し履歴保存	4966.4	4.85%	-	-	-	-
コンテクストスイッチ emulation	558.3	0.56%	900.5	0.89%	558.3	0.56%
OS ドライバ処理，UDP 受信プログラム	-	13.39%	-	10.80%	-	13.39%
Linux ドライバ，World Switch など	-	-	-	-	-	8.48%
合計		85.98%		22.73%		59.97%

- HiTactix は軽量仮想計算機モニタ上で動作する場合には Intel 社 NIC ドライバを，VMware 上で動作する場合には Lance ドライバを利用して受信処理を行う．前者は，1 回の割り込み要因処理につき（到達パケット数）＋1 回の，後者は 3 回のデバイスメモリアクセスが発生する．Lance ドライバが動作した場合のデバイスメモリアクセスエミュレーション処理の起動回数を推測するため，Intel 社の NIC ドライバを Lance ドライバと同様のデバイスメモリアクセス回数となるように改変し，デバイスメモリアクセスエミュレーション処理の起動回数を測定した．その結果 21282 回/秒の起動が発生した．これにより，デバイスメモリアクセスエミュレーションの消費 CPU 使用率は，$(56.44\text{-}13.25) \times 21282/34430{=}26.70\%$だと予測した（デバイスメモリアクセスの履歴保存に要する CPU 使用率は小さいため，本推測では無視した）.

- その他の各種エミュレーション処理は，軽量仮想計算機モニタの履歴保存処理とエミュレーション処理の合計と同等の CPU 使用率を消費すると推定した.

- 残った CPU 使用率が，world switch，バッファのリマッピング，Linux のデバイスドライバの実行により消費された CPU 使用率だと想定し，約 8.48% だと推測した.

上記推測結果より以下が明らかになった.

- 履歴保存なしの軽量仮想計算機モニタを用いた場合，VMware と比べ，約 26.70%の CPU 使

用率を消費するデバイスメモリアクセスエミュレーションが不要になる．これが全体の CPU 使用率低減量の約 72%を占め，CPU 低減の主要因になっている．

- 履歴保存ありの軽量仮想計算機モニタを用いた場合は，約 8.48%の world switch，バッファのリマッピング，Linux のデバイスドライバの動作が不要になるものの，DMA 転送履歴の保存にためのデータコピー処理，CPU クロック読み取り処理で約 18.10%の CPU 使用率を余計に消費し，全体の消費 CPU 使用率が上昇する．さらに本実験では，使用しているデバイスドライバの違いによりデバイスメモリアクセス回数が増えたため，さらなる CPU 使用率上昇が発生した．

3.3.5.2 メモリ消費量の評価

履歴の保存のために消費するメモリ消費量を評価するため，図 3.14 の評価環境にて，UDP 送信プログラムによる UDP パケットの送信レートを変動させた場合に，履歴保存機能付き軽量仮想計算機モニタが履歴保存のために消費するメモリ使用量の増加レートを測定した．

評価結果を図 3.16 に示す．図の横軸が UDP パケットの送信レートを，縦軸が履歴保存のためのメモリ使用量の増加レートを示す．メモリ使用量の測定は，全履歴を格納するためのメモリ使用量と，DMA 転送データの履歴を格納するためのメモリ使用量のそれぞれについて行った．その結果，全メモリ消費量の 98.5%が DMA 転送データの履歴保存に使用されていることがわかった．

提案した OS デバッガにて，長時間にわたる実行の履歴の保存（及びその履歴に基づく再生）を行うためには，上記メモリ消費量増加レートにて wirte 可能な大容量記憶媒体が必要になる．上記の結果より，ギガビットオーダーの write 性能を持つ大容量記憶媒体（HDD を多数搭載した RAID 装置など）を使用すれば，上記の履歴の保存・再生も可能になると予測できる．

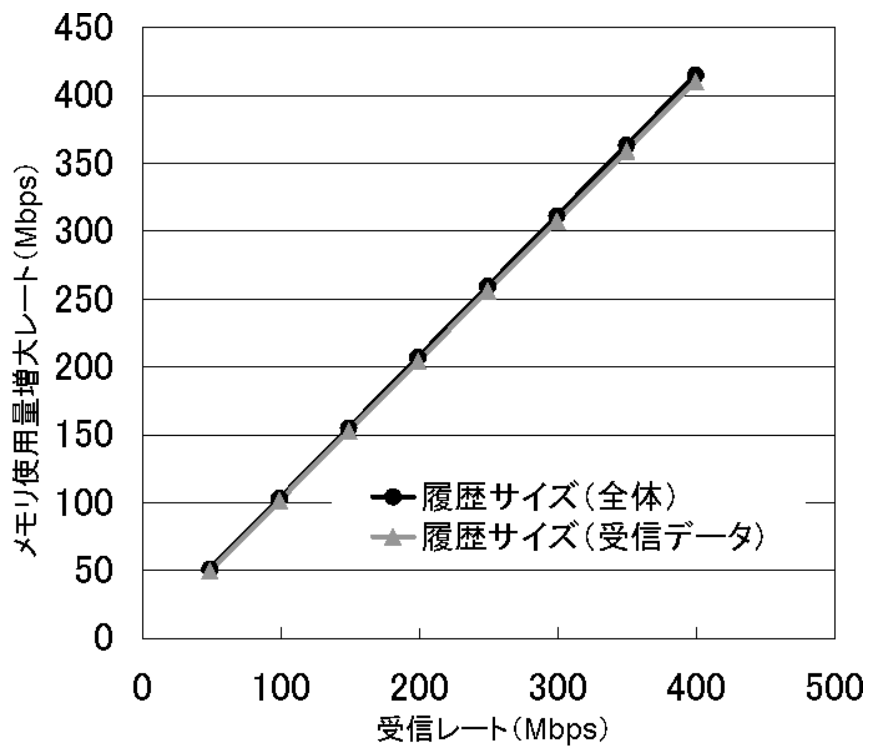

図 3.16: メモリ消費量の評価結果

3.3.6 考察

本節では，提案デバッガの利点と欠点，及び提案デバッガの I/O デバイス依存性についての考察を行う．

3.3.6.1 提案デバッガの利点/欠点

提案デバッガは，汎用性（いかなる I/O デバイスドライバの開発にも適用可能）と安定稼動を確保しつつ，ロギング&リプレイ機能を利用したデバッグ機構を提供する．特にロギング&リプレイ機能に関しては，物理 I/O デバイスから OS ドライバが受けた入力履歴を保存し，当該履歴の忠実な再生ができる点で従来のロギング&リプレイ機能とは異なる．この特徴のために，提案デバッガは以下の利点を提供できる．

- 対象 I/O デバイスに関わらず，再現性の低いバグ（例えば，特定の割り込み発生タイミング，特定の受信データなど限定的な条件下でのみ顕在化するバグやメモリ破壊）が顕在化した場合に，その原因の着実な追究が可能になる．
- OS ドライバと I/O デバイスの並列動作タイミングに起因するバグの着実な原因追求が可能になる．軽量仮想計算機モニタの場合，OS ドライバ実行中も I/O デバイスは並列動作し，ハードウェア状態を変える．一方，従来の仮想計算機モニタのハードウェアエミュレータは，この状態変更を OS ドライバ実行中に行う保証はない．そのため，例えば，OS ドライバの受信処理中に受信ディスクリプタの更新が発生した場合にのみ顕在化するバグなどの原因追求が可能になる．
- I/O デバイスのハードウェアエラー発生など，特別なハードウェア状態下でのみ顕在化するバグの原因追求が可能になる．通常，仮想計算機モニタのエミュレータはハードウェアの異常動作をエミュレートしない[17]．

[17] 但し，現在の実装では 3.3.4.2.1 に示す制限があるため，提案デバッガを用いてデバッグできるハードエラー処理には制限がある．

一方で提案デバッガは，履歴の保存・再生処理での大きなCPU負荷発生，I/Oドライバのモデル化，仮想計算機モニタ層の実装のI/Oデバイス依存性，などの前提を持っているため，以下の欠点を持つ．

- 軽量仮想計算機モニタ上で動作させる場合とさせない場合でOSドライバの実行タイミングが大きく変わる．そのため本モニタ上で顕在化しないバグがある．例えば，一定時間内に特定区間を走行した場合にのみ健在化するバグなどは顕在化しない可能性がある．
- 軽量仮想計算機モニタの履歴の保存・再生に大きなNICの受信性能低下をもたらす．その結果，ネットワーク帯域飽和より前にCPU飽和が発生し，OSドライバによるネットワーク帯域飽和時の異常処理のデバッグが困難になる．デバッグのためには，軽量仮想計算機モニタによる擬似ネットワーク負荷の生成など，追加のエミュレーション処理の実装が必要になる．
- 軽量仮想計算機モニタはOSドライバの動作をモデル化している．OSドライバが当該モデルに従わない動作を行うなどの重大なバグがあった場合，正しい履歴の保存・再生ができない．間違った履歴の再生は，OSドライバ実行中の例外発生としてデバッガは通知する．しかし，再生履歴の誤り，OSドライバのバグのどちらが例外発生要因であるかの特定は難しい．
- 提案デバッガを使用するためにはVM層デバイスドライバの開発が必要になる．VM層デバイスドライバのバグも，誤った履歴再生を引き起こす上に，バグ追求が難しい．

3.3.6.2 提案NIC受信モデルの適用可能性

軽量仮想計算機モニタを用いたOSデバッガは，NICの受信処理のモデル化を行い，DMA転送履歴の保存・再生を行う．本モデルの適用可否はNICが保持するDMA転送機能仕様とOSドライバの実装方式により決まる．本節では，本モデル化の適用範囲について考察し，本モデルが一般的なNICやOSドライバに広く適用可能であることを示す．

3.3.6.2.1　NIC の DMA 転送仕様の仮定事項と適用範囲

NIC の DMA 転送機能仕様に関して提案した受信処理モデルで仮定した事項について述べ，その仮定が一般的な NIC で成立することを示す．

リングバッファ状受信ディスクリプタ OS ドライバと NIC 間で共有リングバッファ状の受信ディスクリプタを持ち，当該ディスクリプタで DMA 転送先アドレス，サイズを通知を行うことを仮定している．本機能は，OS ドライバと NIC の並列動作を可能にし，高速ネットワークからの受信処理時のパケット取りこぼしを防ぐための必須機能であり，高速ネットワークに接続可能な NIC に一般的に搭載されている．

完了ビット DMA 転送が完了した受信ディスクリプタのエントリを完了ビット，もしくはそれに相当する手段で NIC から OS ドライバに通知することを仮定している．一般に本通知は, (1) 受信ディスクリプタのビット（完了ビット）, (2) デバイスレジスタ，のいずれかで行われる．後者の方式を持った NIC でも提案受信モデルの適用は容易である．VM 層ドライバの SetRxUpdateEntry インタフェースの実装において，受信ディスクリプタの更新の変わりにデバイスレジスタの更新を行うことで提案デバッガとの連携ができる．

受信インデックスレジスタ DMA 転送データの読み込みが完了したエントリを，受信インデックスレジスタ，もしくはそれに相当する手段により OS ドライバから NIC が受け取ることを仮定している．一般に本通知は, (1) 受信ディスクリプタのビット, (2) デバイスレジスタ（受信済みインデックスレジスタ），のいずれかで行われる．前者の方式を持った NIC でも提案受信モデルの適用は可能である．前者方式を持つ NIC は，一般に，新たな受信割り込みを受け付け可能になったことを OS ドライバからデバイスレジスタ更新により通知を受ける．そのため，VM 層デバイスドライバで，当該デバイスレジスタアドレスを Init インタフェースでリターンして登録すると共に，当該レジスタアクセス時に呼び出される GetRxIndexReg インタフェースで受信済みインデックスレジスタの代わりに受信ディスクリプタのビットを検査することで，提案デバッガとの連携が可能になる．

3.3.6.2.2　OS ドライバ実装方式の仮定事項と適用範囲

OS ドライバの実装方式に関して提案した受信処理モデルで仮定した事項について述べ，その仮定が一般的な OS ドライバで成立することを示す．

割り込み要因レジスタの読み出し契機 OS ドライバが，割り込み処理の先頭で割り込み要因レジスタを読み出すことを仮定している．NIC には割り込み要因が複数あり，OS ドライバがその要因別に処理を変える必要があるため，一般的な OS ドライバでこの仮定は成立する．

DMA 転送データの読み込み契機 OS ドライバが，割り込み要因レジスタの読み込み後，かつ受信済みインデックスレジスタの更新前に DMA 転送データを読み込み，かつディスクリプタの読み込みエントリを更新することを仮定している．OS ドライバの受信処理は，割り込み要因の読み込みにより起動されること，及び受信インデックスレジスタ更新前のディスクリプタエントリ更新を怠ると NIC による受信済みデータの上書きが発生しうることから，一般的な OS ドライバでこの仮定は成立する．

3.3.7　本節のまとめ

本節では，3.2 節で示した軽量仮想計算機モニタを用いた OS デバッガに，ログ&リプレイ機能を追加するため，その実装方式の提案をおこなった．

3.2 節で提案した軽量仮想計算機モニタを用いた OS デバッガでは，デバッグ対象の OS が NIC や HBA などのハードウェアに直接アクセスを行い，デバイスレジスタアクセスや DMA 転送時に仮想計算機モニタを介さない．ロギング&リプレイ機能の追加を行うためには，これらの履歴の保存・再生が必要になるため，軽量仮想計算機モニタの起動契機付与方式，及びデバッグ対象 OS による直接アクセス動作の追跡方式を新規に設計・実装した．また，これらの方式実現処理におけるデバイス依存処理を極小化・分離する方式も新規に設計・実装した．

起動契機付与方式，及び直接アクセス動作の追跡方式の設計は DMA 転送履歴の保存・再生の際に問題となる．本研究では，NIC 受信時の動作を中心に設計・実装を行った．NIC ドライバ動

作をモデル化した結果,（1）デバッグ対象 OS （NIC ドライバ）割り込み要因レジスタ参照時，受信インデックスレジスタ更新時に軽量仮想計算機モニタの起動契機を与える,（2）受信ディスクリプタ，受信済みインデックスレジスタの更新履歴の追跡により，1 回の割り込みにおいて受信処理された DMA 転送アドレス，サイズ，データ群の取得を行う方式にて履歴の保存・再生が可能であることがわかった．

デバイスレジスタアクセス及び DMA 転送履歴の保存・再生処理には I/O デバイスに依存する処理が含まれる．そこで，デバイス依存部分を VM 層デバイスドライバとして実装する方針をとった．さらに，これらの履歴の保存・再生処理内容を検証し，本ドライバ実装規模を最小化できるドライバインタフェースを設計してデバイス依存処理の極小化・分離を実現した．Intel 社の NIC 用の VM 層デバイスドライバを実装した結果，約 600 行にて実装可能であり，多様な I/O デバイスドライバの開発への本 OS デバッガの適用が容易であることがわかった．

実装した軽量仮想計算機モニタによるロギング&リプレイ機能の実用性を評価するため，ネットワーク受信処理の履歴保存に要する CPU 負荷と消費メモリ量の測定を行った．評価の結果，ロギング&リプレイ処理を行うと，当該処理を行わない場合に比べて CPU 負荷が約 3.5 倍に上昇するが，VMWare によるネットワーク受信処理と比較すると約 38%程度の CPU 負荷の上昇に抑えられることがわかった．また，メモリ消費量の約 98.5%は受信データ履歴の保存であり，ネットワーク受信レートと同等の帯域を確保できる大容量記憶装置があれば，長時間の履歴の保存・再生が可能になることがわかった．

第4章　コストパフォーマンスの評価

本章では，2 章で示した外付け I/O エンジンアーキテクチャや 3 章で示した軽量仮想計算機モニタを用いた OS デバッガにより，ストリーム配信サーバのコストパフォーマンスがどの程度改善できるかの評価を行う．まず，4.1 節にて評価方法について記述し，4.2 節には評価結果について示す．

4.1　評価方法

4.1.1　評価対象

本章の評価は，アーキテクチャとデバッグ方式が異なる表 4.1 の 4 タイプのストリーム配信サーバを対象とする．アーキテクチャは，従来からある 2 種類のアーキテクチャ，もしくは，本研究で提案している外付け I/O エンジンアーキテクチャを仮定する．また，使用デバッガは，タイプ A と B は従来方式で構築された OS デバッガ（タイプ A の場合は gdb のような既存アプリケーションデバッガ，タイプ B の場合は，使用する専用 OS に特化して開発された専用 OS デバッガ）を想定する．一方，タイプ C と D では，従来方式のデバッガの他に，本研究で提案している軽量仮想計算機モニタ上で動作する OS デバッガも想定する．

ストリーム配信サーバアプリケーションは，初期開発時には，Darwin ストリームサーバのような既存アプリケーションを移植する．しかし，その後は，サービス事業者の要望等に応じたカスタマイズや機能拡張を行うと仮定する．

表 4.1: 評価対象となるストリーム配信サーバのタイプ

サーバタイプ	アーキテクチャ	使用デバッガ
A	汎用 OS 上にストリーム配信 AP（管理機能・配信機能）を構築	既存アプリデバッガ
B	専用 OS 上にストリーム配信 AP（管理機能・配信機能）を構築	専用 OS デバッガ
C	外付け I/O エンジンアーキテクチャ（汎用 OS 上に管理機能，専用 OS 上に配信機能を構築）	専用ソフトデバッガ
D	外付け I/O エンジンアーキテクチャ（汎用 OS 上に管理機能，専用 OS 上に配信機能を構築）	軽量仮想計算機モニタ上 OS デバッガ

4.1.2 評価内容

本章では，前節で示したストリーム配信サーバのコストパフォーマンスを評価する．コストパフォーマンスの評価は，想定している規模のサービスを提供するために必要となるコストの比較により行う．

ここで言う「コスト」は，評価対象となるストリーム配信を 1 システム購入し，配信サービスを提供するために，3 年間（想定しているストリーム配信サーバのリプレースサイクル）でサービス提供業者が負担しなければならないコストの総計を言う．そして，このコストは，以下の合計と仮定する．

- ハードウェア購入コスト
- ソフトウェア購入コスト（初期ソフトウェア開発コストの捻出のために，ベンダがサービス提供業者に求める費用）
- 運用維持コスト（ハウジング代，電力消費代など）

- ハードウェア保守コスト（故障ハードウェア部品交換等のためのオンコール費用）
- ソフトウェア保守コスト（バグフィックス，バージョンアップ，カスタマイズ機能開発等のソフトウェア保守コストの捻出のために，ベンダがサービス提供業者に求める費用）

上記で示した各種コストは，ハードウェアで使用している部品価格や信頼性指標値，ソフトウェア開発者の人件費等の情報などの，コストを決定する各種パラメータが入手できない限り，正確な見積りができない．そこで，本章では，以下の 2 つの評価を行う．

1. 4.1.2 節で示した各種コストが，4.1.1 節で示した各タイプのサーバ間でどのように増減するかの定性的な評価.
2. コストを決定する各種パラメータとしてできる限り妥当な値を仮定し，当該仮定のもとでの，サービス提供コストの定量的な見積りと，各タイプのサーバごとの比較.

4.2 評価結果

4.2.1 定性的な評価結果

コストパフォーマンスの定性的な評価結果を表 4.2 に示す.

ハードウェア購入コスト サーバタイプ A は，汎用 OS 上で配信機能が動作するため，必要サーバ台数が多くなり，その分だけハードウェア購入コストは増大する．サーバタイプ B，C，D は，専用 OS 上で配信機能が動作する（2.5.2 節の評価結果によると，約 5 倍の配信性能向上を達成する）ため，当該コストを約 1/5 に低減できる．但し，サーバタイプ C,D は，汎用 OS が動作するサーバが 1 台多く必要となるため，サーバ B と比べるとわずかにコストが増大する．

ソフトウェア購入コスト 4.1.1 節で述べた通り，本章の評価では，初期開発時には，Darwin ストリームサーバのような既存アプリケーションを移植することを想定している．そのため，サー

バタイプ A では，この移植コストが殆んどかからず（流用可能），コストが低減できる．サーバタイプ B は，上記アプリケーションの全モジュールの移植の他に，OS デバッガ，管理ツールの新規開発が必要となるため，コストが大きくなる．一方，サーバタイプ C は，アプリケーションの一部モジュールの移植と OS デバッガの開発のみが必要となるため，サーバタイプ A と B の中間のコストとなる．サーバタイプ D はアプリケーションの一部モジュールの移植のみで済むので，サーバタイプ C よりはコストを抑えられるが，A よりはコストは大きくなる．

運用維持コスト ハードウェア購入コストと同様，必要サーバ台数が多い（その結果，ハウジングコストや電気代が増大する）ために，サーバタイプ A のコストが圧倒的に大きくなる．サーバタイプ B，C，D のコストは約 1/5 になるが，汎用 OS が動作するサーバが不要となる分，サーバタイプ B のコストの方が C,D に比べて小さくなる．

ハードウェア保守コスト ハードウェア購入コストと同様，必要サーバ台数が多いために，サーバタイプ A のコストが圧倒的に大きくなる．サーバタイプ B，C，D のコストは約 1/5 になるが，汎用 OS が動作するサーバが不要となる分，サーバタイプ B のコストの方が C,D に比べて小さくなる．

ソフトウェア保守コスト サーバタイプ A は，カスタマイズ機能，管理拡張機能，配信拡張機能をすべて汎用 OS 上で開発するため，コストを低減できる．サーバタイプ B はこれらすべてを専用 OS 上で開発するため，コストはいちばん大きい．サーバタイプ C，D は，配信拡張機能のみを専用 OS 上で開発し，それ以外を汎用 OS 上で開発するため，サーバタイプ A と B の中間のコストとなる．但し，開発効率を上げられる OS デバッガを使用している分，サーバタイプ D のコストの方が小さくなる．

表 4.2: 定性的なコストパフォーマンスの評価結果

サーバタイプ	A	B	C	D
ハードウェア 購入コスト	× 多数サーバ	◎ サーバ台数 1/5	○ サーバ台数 1/5+1 台	○ サーバ台数 1/5+1 台
ソフトウェア 購入コスト	◎ 既存 AP 流用可	× AP 全モジュール移植, &OS デバッガ, 管理ツール開発要	△ AP 一部モジュール移植 &OS デバッガ開発要	○ AP 一部モジュール移植要
運用維持 コスト	× 多数サーバ	◎ サーバ台数 1/5	○ サーバ台数 1/5+1 台	○ サーバ台数 1/5+1 台
ハードウェア 保守コスト	× 多数サーバ	◎ サーバ台数 1/5	○ サーバ台数 1/5+1 台	○ サーバ台数 1/5+1 台
ソフトウェア 保守コスト	◎ 汎用 OS 上の カスタマイズ, 管理・配信機能拡張	× 専用 OS 上の カスタマイズ, 管理・配信機能拡張	△ 汎用 OS 上の カスタマイズ, 管理機能拡張 専用 OS 上の 配信機能拡張	○ 汎用 OS 上の カスタマイズ, 管理機能拡張 専用 OS 上の 配信機能拡張 (+高効率 OS デバッガ)

4.2.2 定量的な評価結果

本節では，4.1.2 節で示した各種コスト及び配信性能の定量的な見積りにおいて仮定した各種パラメータと，当該パラメータを用いて算出した各種コストの見積り結果を示す．さらに，各サーバごとに各種コストの総計の比較と分析結果を示す．

4.2.2.1 仮定したパラメータ値

本見積もりで想定したパラメータ値の一覧を表 4.3 に示す．各パラメータの想定根拠を以下に列挙する．

配信性能 2.5.2 節の評価実験結果に基づき，サーバ 1 台あたりの配信性能は，専用 OS 搭載時，もしくは外付け I/O エンジンアーキテクチャ使用時（4.1.1 節で示したサーバタイプ B,C,D）の配信性能を 1.5Gbps，汎用 OS を搭載時（サーバタイプ A）の配信性能を 0.3Gbps と想定した．また，ピーク時の配信帯域は大規模のサービスを提供するサービス事業者と通常規模のサービスのサービスを提供する事業者の 2 種類を想定し，それぞれごとにコストパフォー

マンスを算出する．最大規模の顧客のピーク時配信帯域は，文献 [1] に記載の国内最大の配信規模である 300Gbps の 1/10 を想定した．通常顧客のピーク時配信帯域はさらにその 1/10 とした．

ハード価格 必要となる CPU 処理能力，I/O 帯域幅，ストレージ容量が大きいため，省電力・省スペースでありながら，CPU 周波数，メモリ容量，拡張性の高い PC サーバが用いられる．Dell 社の PowerEdge2950III などはその典型例であり，同モデルのインターネット販売価格（Quad Core Intel 社 Xeon E5405 2.0GHz 2 個搭載，4GB メモリ，OS なし，1TB SATA HDD4 台搭載）をもとに，想定値を算定した．本費用はハード保守費用も含んでいる．

ソフト価格 汎用 OS 上のソフトウェア開発費（K ステップ単価）は，文献 [36] の記載から，100 万円／ K ステップと想定した．一方，専用 OS 上のソフトウェア開発は，極めて熟練した技術者が必要であること，及び開発効率が落ちることから，その倍のコストがかかると想定した．本来なら，軽量仮想計算機モニタを利用した OS デバッガを使用した場合と，既存 OS デバッガを使用した場合とで，専用 OS 上のソフトウェア開発費に差が出るはずだが，この差の見積りが困難であるため，本仮定では，差がないとした．

運用維持価格 運用維持費用は，ハウジングサービスの利用料金と同等と仮定した．Domain キーパー社の提供価格 [37] をもとに，費用を想定した．

初期開発ステップ 2.5.1 節の評価結果に基づき，外付け I/O エンジン方式を用いた際（サーバタイプ C，D）の初期開発ステップ数は 9.1K ステップと想定した．同じく，同節の評価結果に基づき，専用 OS 上にストリーム配信サーバを構築した際には，2.5.1 節に示した（1）と（3）の開発コード量である 20.0K ステップの初期開発が必要だと想した．一方，汎用 OS 上にストリーム配信サーバを構築した場合には，既存のストリーム配信サーバをほぼ改変なく使用できると仮定し，1.0K ステップのみ初期開発が必要だと想定した．また，専用 OS 上で実装すべき管理ツール（サーバ B のみ必要）の開発ステップ数は，Darwin ストリームサーバの移植で必要となるパスワード管理機能，Web サーバ機能等の開発コード量の見積り量をもと

に想定した．一方，OS デバッガ（サーバ B のみ必要）の開発ステップ数は，gdbserver[38] のコード量をもとに想定した．

保守開発ステップ 新規プロトコル対応ステップ数，新規フォーマット対応ステップ数は，それぞれ Darwin ストリームサーバ上で既に実装されている RTSP プロトコル，及び HiTactix 上で既に実装されている Quick フォーマット配信機能のコード量をもとに想定した．一般に，管理機能拡張は，課金システムとの連携機能開発等の大規模開発が多いため，新規プロトコル対応と同等の開発規模だと想定した．一方，カスタマイズ機能は，顧客管理システムとの接続等の小規模開発が多いため，新規フォーマット対応と同等の開発規模だと想定した．新規プロトコルの対応頻度は，Darwin ストリームサーバの機能拡張実績（HTTP ストリーミングプロトコルに対応）をもとに想定した．一方，新規フォーマット対応は，Darwin ストリームサーバの過去の実績はないが，最大で新規プロトコルと同等の頻度がありうると想定した．一方，カスタマイズ機能開発は，通常顧客の数だけ発生するため，販売先システム数とほぼ同等とした．管理機能開発は，カスタマイズ機能開発頻度より小さいが，新規プロトコル対応頻度より大きいと想定し，1 回／1 年とした．

製品寿命 サーバリプレースサイクルは，Dell 社 PowerEdge2950III の標準サポート期間をもとに想定した．また，ソフトウェアの投資回収期間は，上記リプレースサイクルと同じと仮定した．また，販売先サービス事業者数は，沖電気の OkiMediaServer の販売計画（2 年間で 150 システム）[39] と同等のなるように想定した．

4.2.2.2 コストパフォーマンス比較結果

前節で仮定したパラメータをもとに，4.1.1 節で示した各サーバタイプのコストパフォーマンスを見積もった．見積り結果を図 4.1 に示す．

本評価において使用した，各コストの算出方法の概要を以下に示す．

ハードウェア購入・保守コスト ピーク時配信総帯域とサーバ配信性能から必要サーバ台数を算出，

表 4.3: 想定したパラメータ値

分類	パラメータ	値
配信性能	ピーク時配信総帯域	3Gbps または 30Gbps
	サーバ配信性能 （専用 OS 搭載時）	1.5Gbps
	サーバ配信性能 （汎用 OS 搭載時）	0.3Gbps
ハード価格 （保守費込み）	サーバ価格 （1 台あたり，2U）	70 万円
ソフト価格	ソフトウェア開発費 （汎用 OS 上，1K ステップあたり）	100 万円
	ソフトウェア開発費 （専用 OS 上，1K ステップあたり）	200 万円
運用維持価格	サーバ占有ラックユニット数	2U
	ハウジング費用（フルラック，40U）	30 万円/月
	ハウジング費用（1/2 ラック，19U）	18 万円/月
	ハウジング費用（1/4 ラック，9U）	12 万円/月
	ハウジング費用（1U）	1.8 万円/月
初期 開発ステップ	ストリーム配信サーバ AP （外付け I/O エンジン方式）	9.1K ステップ
	ストリーム配信サーバ AP （専用 OS 上）	20.0K ステップ
	ストリーム配信サーバ AP （汎用 OS 上）	1.0K ステップ
	管理ツール （専用 OS 上）	20.0K ステップ
	OS デバッガ移植 （専用 OS 上）	10.0K ステップ
保守 開発ステップ	新規プロトコル対応	11.0K ステップ
	新規プロトコル対応頻度	1 回/3 年
	新規フォーマット対応	5.0K ステップ
	新規フォーマット対応頻度	1 回/3 年
	管理機能拡張	11.0K ステップ
	管理機能拡張頻度	1 回/1 年
	カスタマイズ機能	5.0K ステップ
製品寿命	販売先サービス事業者数	30Gbps：年間 5 システム 300Gbps：3 年間 1 システム
	サーバリプレースサイクル	3 年
	ソフトウェア投資回収期間	製品販売開始から 3 年

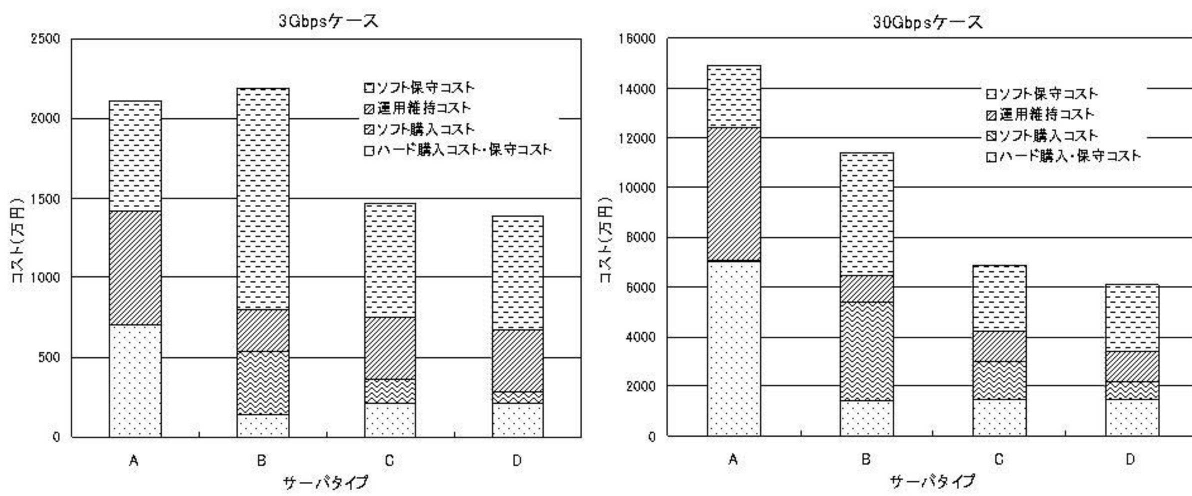

図 4.1: コストパフォーマンスの比較結果

サーバ価格（保守費込み）に当該サーバ台数を掛けて，ハードウェア購入・保守コストを計算する．

ソフトウェア購入コスト まず，ソフトウェア初期開発費用を，初期開発ステップ数にKステップあたりのソフトウェア開発費をかけて算出する．そして，ソフトウェア開発投資回収期間でのサーバの販売台数で割ることで，サーバ1台あたりに課金すべきソフトウェア価格が算出する．さらに，この価格に，必要サーバ台数を書けてソウトウェア購入コストとする．

運用維持コスト 必要サーバ台数にサーバ占有ラックユニットを掛けて，必要総ラックユニット数を算出，当該ラックユニット数分のサーバリプレイスサイクル期間中のハウジング費用の総計を運用維持コストとする．

ソフトウェア保守コスト 各種機能（新規プロトコル対応，新規フォーマット対応，管理機能拡張）の保守開発ステップ，開発頻度，Kステップあたりのソフトウェア開発費から，ソフトウェア投資回収期間中のソフトウェア保守にかかる総開発費用を算出，当該費用を同期間中のサーバ販売台数でわることで，サーバ1台あたりに課金すべきソフトウェア保守価格を算出する．この価格に，必要サーバ台数を書けてソウトウェア保守コストを算出する．但し，カスタマイズ機能についてのみ，当該機能開発に要する費用を，販売先となるサービス事業者に負担させるべく，カスタマイズ機能の開発ステップ数にKステップあたりのソフトウェア開発費に掛けた値を，上記ソフトウェア保守コストに上乗せする．

本評価結果から，以下が明らかになった．

- 外付けI/Oエンジン方式と軽量仮想計算機モニタを用いたOSデバッガを使用することで，汎用OS上にストリーム配信サーバアプリケーションを構築する方式より，総配信帯域が3Gbpsのケースで約34%のコスト低減が実現できる．また，総配信帯域が30Gbpsのケースで約59%のコスト低減が実現できる．これは，サーバ1台あたりの配信性能向上により，必要なサーバ台数が約1/5に減少，それに伴い，ハードウェア購入・保守コストや運用維持コストの低減が約1/5に低減するためである．ソフトウェア購入コストやソフトウェア保守コ

ストは，従来方式と比べてやや上昇するが，そのコスト上昇量は，上記低減量と比べると小さい．また，ソフトウェア購入・保守コストの総コストに占める割合は，総配信帯域が増えるほど小さくなることから，総配信帯域が増える程，外付け I/O エンジン方式と軽量仮想計算機モニタを用いた OS デバッガによるコスト低減効果は大きくなる．

- 外付け I/O エンジン方式と軽量仮想計算機モニタを用いた OS デバッガを使用することで，専用 OS 上にストリーム配信サーバアプリケーションを構築する方式より，総配信帯域が 3Gbps のケースで約 37%のコスト低減が実現できる．また，総配信帯域が 30Gbps のケースで約 47%のコスト低減が実現できる．これは，汎用 OS 上に管理機能を動作させることによるソフトウェア初期開発のコード量低減（約 82%低減），及びソフトウェア保守による開発コスト低減（約 49%低減）による．外付け I/O エンジン方式の場合，汎用 OS を動作させるサーバが 1 台余計に必要になるが，これによるハードウェア購入・保守コストや運用維持コスト向上の比率は，総配信帯域が大きくなる程小さくなり，結果として，外付け I/O エンジン方式と軽量仮想計算機モニタを用いた OS デバッガによるコスト低減効果は大きくなる．

- 軽量仮想計算機モニタを用いた OS デバッガのみよるコスト低減効果は 5.4%〜8.8%にとどまっている．これは，開発効率向上によるソフトウェア開発費の低減（結果として，ソフトウェア購入・保守コストの低減）の効果がないとパラメータ決定時に仮定したためである．実際のコスト低減効果は見積り値より大きいと考えられる．

第5章　関連研究

本章では，本論文で新規に提案した外付け I/O エンジン方式，及び軽量仮想計算機モニタを利用した OS デバッガの関連研究について述べる．

5.1　外付け I/O エンジン方式の関連研究

本稿で提案した外付け I/O エンジン方式のように，市販ストリームサーバとの機能互換性維持，少ない開発工数，高性能ストリーム配信，配信品質保証のすべてを同時に実現するストリームサーバを構築しようとする研究は，著者らの知る限りでは存在しない．

従来のストリームサーバは，

- 汎用 OS 上にストリームサーバを構築する．ストリーム配信性能を向上させる必要がある場合には，汎用 OS とインタフェースの互換性を維持しながらも I/O 性能の向上をはかる機能を使用する．
- 専用 OS 上にストリームサーバを構築する．専用 OS 内部に，高性能ストリーム配信機能や配信品質保証機能を組み込む．

のどちらかのアプローチで構築されていた．

前者のアプローチとしては，例えば Linux 等の汎用 OS と，Zero-Copy TCP[40] や I/O-Lite[41] 等の配信性能の向上機能を組み合わせることが考えられる．しかし，このアプローチでは配信品質保証の実現が難しい．何故ならば，配信品質の保証のためには，配信に必要な資源の予約宣言等の既存 OS のインタフェースでは提供されていない機能をアプリケーションが利用する必要があるためである．

一方，後者のアプローチとしては，splice[42]，RoadRunner[43, 44]，Nemesis[45]，Tiger Video File Server[46] 等が知られている．しかし，これらのアプローチは，開発工数の低減や市販ストリームサーバとの機能互換性維持について考慮していない．さらに，以下の点で本研究と差がある．

splice，RoadRunner は，デバイス間のデータ転送を高速に行う特別なパスを作成することにより，高性能なストリーム配信を実現する．しかし，このパスを経由してストリーム配信を行う場合，アプリケーションレベルのヘッダ（例えば，RTP プロトコル [47] を使用する場合は RTP ヘッダ）を挿入したり，ディスクから読み込んだデータをネットワークに送信するタイミングを制御することができない．HiTactix 搭載の外付け I/O エンジンでは，アプリケーションが自由に上記制御を行える．

Nemesis は，デバイスドライバ層に I/O スケジューリング機能，すなわち I/O 要求の実行順序を入れ換える機能を組み込むことにより，配信品質保証を実現する．しかし，I/O 要求発行のタイミングの制御については考慮されていないため，I/O 要求を発行するスレッドが厳密にスケジューリングされないことによる配信品質の劣化が発生し得る．HiTactix 搭載の外付け I/O エンジンは，「I/O 実行スケジューリングテーブル」を保持することにより，この配信品質の劣化を防いでいる．

Tiger Video File Server は，ディスク I/O やネットワーク I/O のスケジューリングテーブルを保持することにより，配信品質保証を実現する．しかし，ストリーム配信以外の負荷（例えば，ストリームデータ登録に伴うディスク負荷）がかかった場合を想定しておらず，この場合に配信品質の劣化が発生し得る．HiTactix 搭載の外付け I/O エンジンは，2.3 節で述べた通り，このような場合にも配信品質を保証する．

5.2 軽量仮想計算機モニタを利用した OS デバッガの関連研究

本 OS デバッガが目的としている効率的な OS のデバッグを目的とした関連研究やソフトウェアは数多く存在する．

3.1 節で示した kgdb[25] などのソフトウェアリモートデバッガ，kdb[26] などの OS 内部に組み込まれた当該 OS 専用のデバッガなどはその例である．しかし，これらのデバッガはデバッグ環境

の安定稼働を保証しておらず，OS のバグに起因する異常動作により，デバッグ継続が不可能になる可能性がある．本論文で提案した軽量仮想計算機モニタを使用した OS デバッグ方法では，上記デバッガと比して多少の OS の I/O 性能低減等のデメリットが発生するものの，代わりにデバイスエミュレーションや軽量メモリ領域保護機能等を利用してデバッグ環境の安定稼働の保証を提供している．

Temporal Debugger[22] は，汎用 OS 上に構築されたハードウェアシミュレータと連動するソフトウェアデバッガの一例である．通常のソースレベルのリモートデバッグだけでなく，汎用 OS の実行タイミング情報をユーザに提供している．そのため，デバッグ環境の安定稼働を保証するだけでなく，I/O 性能のエンハンス等も本デバッガを用いて行える．しかし，本デバッガの利用には完全なハードウェアシミュレータが必須であり，次々と新種の高速 I/O デバイスが利用可能になる PC/AT 互換機向け OS への適用が難しい．本論文ので提案した OS デバッグ方式は，部分的なハードウェアエミュレーションを行うことで，新種の高速 I/O デバイスへの適用を容易にしている．

また，本論文で提案した OS デバッグ方式は，仮想計算機モニタを低オーバヘッドで動作させることにより，デバッグ対象となる OS に高い実行性能（特に高い I/O 性能）を提供可能にしている．従来から，仮想計算機モニタを低オーバヘッドで動作させる研究は数多く存在する．

1960 年代より，System/370 をはじめとするメインフレームにおいて仮想計算機モニタを高速に動作させる数多く研究 [48, 49, 50, 51] はその代表例である．これらの研究における仮想計算機モニタは，実ハードウェアと完全なバイナリ互換性を保証しながら，複数 OS を一つのハードウェア上でできるだけ高速に並行実行させ，かつ，OS 間のアクセス保護も完全に保証している．しかしこれらの研究では，実ハードウェアに仮想化に必要な機能要件 [52] が予め備わっていることを前提としている．また，I/O デバイス等も多様な種類が存在することを仮定していないため，多様な I/O デバイスを少ない開発工数でサポートする必要性を想定していない．また，仮想計算機モニタの高速化のために，実ハードウェアに機能追加を行っても良いとしている．本論文で提案した軽量仮想計算機モニタは，実ハードウェアとの完全なバイナリ互換性を保証する必要はない，

複数OSを一つのハードウェア上で並行実行させる必要はない，と仮定する代わりに，仮想化に必要な機能を備えていないPC/AT互換機を実ハードウェアの仕様を変更せずに使用することを前提としている．また，多様なI/Oデバイスを少ない開発工数でサポートすることも実現している．

VMware Workstation[28, 33]は，PC/AT互換機上で動作する仮想計算機モニタである．Hosted Virtual Machine Architectureを持つことで，多様なI/Oデバイスをサポートできる．さらに，一つのハードウェア上での複数OSの並行実行や，実ハードウェアとの完全なバイナリ互換性の保証も実現している．本論文の軽量仮想計算機モニタは，複数OSの並行実行や完全なバイナリ互換性の保証をしない代わりに，3.2.5.1節で示した通り，デバッグ中のOSの高いI/O性能を実現している．また，VMware Workstationは，I/Oデバイスのエミュレーションにより I/O動作の正当性保証を行う．そのため，決められた独自OSのデバイスドライバしか動作させることができない．それに対して，軽量仮想計算機モニタは，DMA転送先領域の正当性を独自OS自身に保証させる代わりに，独自OSから高速I/Oデバイスへの直接アクセスを許している．そのため，軽量仮想計算機モニタ上で独自OSが持つ様々なデバイスドライバを動作させることができる．すなわち，多様なデバイスのドライバのデバッグに利用できる．

VMware ESX Server[34]もPC/AT互換機上で動作する仮想計算機モニタである．しかし，VMware ESX Serverはデバイスドライバを仮想計算機モニタ自身が持つため，多様なI/Oデバイスのサポートが困難である．軽量仮想計算機モニタは，多様なI/Oデバイスのサポートを容易に実現できる．

Denali[53]やXen[32]もPC/AT互換機上で動作する仮想計算機モニタである．しかし，これらは仮想計算機モニタを低オーバヘッドで動作させることを最優先課題とし，実ハードウェアと完全に一致する，あるいは実ハードウェアと近いインタフェースを提供することを目指していない．そのため，実ハードウェア上で動作するOSをこれらの仮想計算機モニタ上で動作させようとする際に，OSにバグが混入する可能性が高くなる．軽量仮想計算機モニタは，実ハードウェアとほぼ互換なインタフェースを提供することで，実ハードウェア上で動作するOSを当該モニタ上で動作させようとする際に必要となるOSの改変量をより少なくし，かつ改変の内容もより容易にしてい

る．結果として，改変に伴う OS へのバグ混入の可能性が低くなる．

本論文で提案した軽量仮想計算機モニタは，従来の仮想計算機モニタと異なり，一つの実ハードウェア上で複数 OS を並行実行させることを目的としていない．軽量仮想計算機モニタ内に含まれるリモートデバッグ機能を OS の異常動作から守り，デバッグ機能を安定稼働させることがその目的である．仮想計算機モニタに従来とは異なる利用用途を提唱する研究もいくつか存在する．

Disco[54] は，高多重 SMP 向けのエンハンスがなされていない商用 OS を ccNUMA 型高多重 SMP マシンで高性能に動作させるために，仮想計算機モニタを用いる．高多重 SMP マシンを複数の仮想計算機に見せ，かつ，各仮想計算機上で商用 OS を動作させれば，仮想計算機モニタ内に実装されている高性能 ccNUMA 向けメモリ管理機構やメモリ共有機能が，商用 OS に特別なエンハンスを加えなくとも利用されるようになり，システム全体として高いパフォーマンスが達成できる．仮想計算機モニタを従来と異なる用途で利用している点で軽量仮想計算機モニタと類似しているものの，本論文で課題としている多様な I/O デバイスサポート等については Disco は考慮しておらず，特に類似点はない．また，ターゲットハードウェアも PC/AT 互換機ではないため，CPU のハードウェア資源の仮想化方法も大きく異なる．

Bressoud ら [55] は，仮想計算機モニタ内に Fault-tolerant Computing を実現するためのプロトコルを実装している．一つのハードウェア上に一つの OS のみを動かすことを仮定し，仮想化の対象を必要最小限に限っている点で本研究と類似している．しかし，本論文における軽量メモリ領域保護機能のように，仮想計算機モニタと OS 間のメモリ保護については考慮されていない．また，Disco 同様，ターゲットハードウェアも PC/AT 互換機ではないため，CPU のハードウェア資源の仮想化方法も大きく異なる．

また，本 OS デバッガに組み込んだロギング&リプレイ機能の関連研究も多数存在する．

Flight Data Recorder（FDR）[56] は，メモリの更新履歴，割り込み発生履歴，I/O ポートアクセス履歴，DMA 転送履歴等を保存できるモジュールを組み込んだ専用ハードウェアを実装し，OS 動作の完全な再生（ロギング&リプレイ機能）を可能にしている．しかし，本研究でターゲットとしている汎用 PC サーバは，チップセット，バスアーキテクチャ，周辺 I/O デバイスが頻繁に更新

され，ハードウェアにロギング&リプレイ機能を組み込む方式は開発工数がかかる．軽量仮想計算機モニタによるロギング&リプレイ機能は，汎用 PC サーバが提供するインタフェースが変わらない（PC/AT 仕様に準拠し続ける）限り，VM 層デバイスドライバの新規開発のみで上記更新にも対応できる．

ReVirt[27]，ExecRecorder[57] は，通常の仮想計算機モニタ上でロギング&リプレイ機能を実現し，OS 動作の完全再生を可能にしている．仮想計算機モニタにて I/O デバイスの完全なエミュレーションを行っているため，デバイスレジスタアクセス履歴や DMA 転送履歴の保存・再生は容易に実現できるものの，仮想計算機モニタにてエミュレートできる I/O デバイスに限りがあり，新規 I/O デバイスのドライバ開発への適用が難しい．軽量仮想計算機モニタは，デバイスレジスタアクセス等の捕捉は行うがエミュレーションは行わないことで，新規 I/O デバイスのドライバ開発への適用を可能にしている．

BugNet[58]，FlashBack[59] は，専用ハードウェアもしくは OS の機能拡張により，ユーザレベルのコードのロギング& リプレイを可能にしている．デバッグ対象をユーザレベルのコードに限定しているため，割り込み発生履歴や DMA 転送履歴の保存・再生は行わない．軽量仮想計算機モニタのロギング&リプレイ機能は，これらの履歴の保存・再生もサポートし，OS 動作の完全再生を実現している．

第6章　まとめ

本研究では，ストリーム配信サーバのコストパフォーマンスの改善を目的に，以下を行った．

- 汎用 OS 搭載のストリーム配信サーバと専用 OS 搭載のストリーム配信サーバの利点を兼ね備えた「外付け I/O エンジンアーキテクチャ」の提案，及び当該アーキテクチャに基づくストリーム配信サーバの試作と，その性能向上効果や開発コード量削減効果などの定量的な評価．
- 上記アーキテクチャのストリーム配信サーバの開発において必須となる，専用 OS，及び専用 OS 上のアプリケーションの開発の開発効率を改善することを目的とした「軽量仮想計算機モニタを利用した OS デバッガ」の提案と試作．さらに，当該 OS デバッガが，現実的な CPU 負荷で開発効率改善を達成できることの実機での性能評価を通じた確認．
- 「外付け I/O エンジンアーキテクチャ」「軽量仮想計算機モニタを用いた OS デバッガ」を用いることにより達成できるコストパフォーマンス改善の評価．

外付け I/O エンジン方式は，既存の市販ストリーム配信サーバの機能互換性を維持しつつ，当該サーバの配信性能の向上，配信品質保証機能追加を少ない開発工数で実現するストリーム配信サーバの構成方式である．本方式は，汎用 OS 上で動作する既存の市販ストリームサーバをベースとして使用するが，配信性能の向上，配信品質保証のため，そのサーバの配信機能のみを専用 OS（筆者らが開発した HiTactix など）搭載サーバに移植，汎用 OS 搭載サーバと専用 OS 搭載サーバを連動させて，一つのストリーム配信サーバを構築する．本方式の最適な実現方式を設計するため, まず, 既存のストリーム配信サーバのモジュール構成を調査した．その結果，市販ストリームサーバは，クライアントからの配信開始・停止要求を受け付ける配信要求処理モジュール，当該要

求到達を契機に管理処理を行う管理処理モジュール，当該要求到達を契機にストリームデータのI/O スケジューリングを行う配信制御モジュール，当該スケジュールに応じて実際のI/O 処理を実行する配信ドライバモジュールから通常なっていることがわかった．さらに，上記4モジュールのうちどのモジュールを専用OS搭載サーバに移植すれば移植量，配信性能，連動オーバーヘッド等を最適化できるかを検討した結果，配信制御モジュールと配信モジュールのみを専用OSに移植し，汎用OS搭載サーバと専用OS搭載サーバを，配信スケジュール開始・停止を行うインタフェースで接続するのが最善であることがわかった．Darwin ストリームサーバ（Apple社が提供する市販ストリーム配信サーバ）とHiTactix搭載の外付けI/O エンジンを本方式に基づき連動させたところ，上記連動は9.1K行程度のコード追加で実現できること，連動オーバヘッドによるストリーム配信性能の劣化は1%以下に抑えられること，配信性能は既存のDarwin ストリーム配信サーバと比べて5倍程度向上できること等を確認できた．

「軽量仮想計算機モニタを用いたOSデバッガ」は，専用OSの特にI/O デバイスドライバや，専用OS上のアプリケーションの開発効率向上の実現を目的としている．特に，デバッグ環境の安定稼働保証，多様なOSやI/O デバイスドライバ開発への適用，デバッグ時に高いI/O 性能達成，の3条件の同時充足ができないのが従来デバッグ環境の課題であり，これらの同時充足を目指した．さらに, 提案した軽量仮想計算機モニタにロギング&リプレイ機能を追加するための実装方式を提案し，タイミングクリティカルなバグのデバッグ効率の向上を実現した.

提案したデバッグ方式は，従来のソフトウェアリモートデバッグ方式の改良である．しかし，従来方式と異なり，ターゲットマシン上でリモートデバッグ機能も内部に持つ軽量仮想計算機モニタが動作する．軽量仮想計算機モニタは，リモートデバッグ機能が使用するハードウェア資源のみを仮想化する部分ハードウェアエミュレーション機能を持つ．リモートデバッグ機能が使用するハードウェア資源はデバッグ対象OSが直接アクセスしないため，OSの異常動作時にも，デバッグ環境の安定稼働が保証できる．また，軽量仮想計算機モニタがOS に提供するインタフェースは実ハードウェアインタフェースと互換性があるため，様々なOSのデバッグに使用できる．さらに，NIC やHBA のような高速I/O デバイスに関しては，軽量仮想計算機モニタでエミュレーション

処理を行わず，デバッグ対象OSが直接アクセスを実行するので，多様なI/Oデバイスドライバ開発への適用が可能になる上，デバッグ時の高いI/O性能達成も可能になる．実装した軽量仮想計算機モニタの実行性能の評価を行った結果，従来のHosted Virtual Machine Monitorと比べて5.4倍のI/O性能は達成し，現実的なCPU負荷で上記3条件を充足するデバッグ環境が実現できることがわかった．

さらに，軽量仮想計算機モニタを用いたOSデバッガに，ログ&リプレイ機能を追加するため，その実装方式の提案をおこなった．軽量仮想計算機モニタを用いたOSデバッガでは，デバッグ対象のOSがNICやHBAなどのハードウェアに直接アクセスを行い，デバイスレジスタアクセスやDMA転送時に仮想計算機モニタを介さない．ロギング&リプレイ機能の追加を行うためには，これらの履歴の保存・再生が必要になるため，軽量仮想計算機モニタの起動契機付与方式，及びデバッグ対象OSによる直接アクセス動作の追跡方式を新規に設計・実装する必要がある．起動契機付与方式，及び直接アクセス動作の追跡方式の設計はDMA転送履歴の保存・再生の際に特に問題となる．本研究では，NIC受信時の動作を中心に設計・実装を行った．NICドライバ動作をモデル化して上記方式検討を行った結果,（1）デバッグ対象OS（NICドライバ）割り込み要因レジスタ参照時，受信インデックスレジスタ更新時に軽量仮想計算機モニタの起動契機を与える,（2）受信ディスクリプタ，受信済みインデックスレジスタの更新履歴の追跡により，1回の割り込みにおいて受信処理されたDMA転送アドレス，サイズ，データ群の取得を行う方式にて履歴の保存・再生が可能であることがわかった．さらに，実装した軽量仮想計算機モニタによるロギング&リプレイ機能の実用性を評価するため，ネットワーク受信処理の履歴保存に要するCPU負荷の測定を行った．評価の結果，ロギング&リプレイ処理を行うと，VMWareによるネットワーク受信処理と比較しても約38%程度のCPU負荷の上昇に抑えられ，現実的なCPU負荷でログ&リプレイ機能の実装が動作することがわかった．

最後に，外付けI/Oエンジン方式及び軽量仮想計算機モニタを用いたOSデバッガによる，ストリーム配信サーバのコストパフォーマンス改善効果の見積りを行った．コストパフォーマンスの見積りは，想定している規模のストリーム配信サービスを提供するために，サービス事業者が

負担すべきコストの総計の見積り値を比較することにより行った．見積りの結果，上記 2 方式を使用することにより，従来のストリーム配信サーバと比べて，34%～59%のコストパフォーマンスの改善が達成できる見込みがあること，特に，性能向上によるハードウェア購入・保守・運用維持コストの低減量が大きく，専用 OS 使用によるソフトウェア購入・保守コスト（ストリーム配信サーバベンダのソフトウェア開発費用）の増大量を上回ることを十分期待できることがわかった．

関連図書

[1] 榊原 康. 急成長するネット放送「gyao」 システム刷新で 1000 万人に対応. **日経コミュニケーションズ**, pages 130–133, Dec 2005.

[2] 生まれ変わるか iptv 市場立ち上げを阻む「三つの壁」. **日経コミュニケーションズ**, pages 46–50, Nov 2007.

[3] Microsoft. Windows media. *http://www.microsoft.com/ windows/ windowsmedia/ default.mspx*.

[4] Real Networks. Real networks media servers. *http://www.realnetworks.com/ products/ media_delivery.html*.

[5] Apple. Quicktime streaming server. *http://www.apple.com/ quicktime/ sreamingserver*.

[6] Masaaki Iwasaki, Tadashi Takeuchi, Takahiro Nakano, and Masahiko Nakahara. Isochronous scheduling and its application to traffic control. *19th IEEE Real-Time System Symposium*, pages 14–25, Dec 1998.

[7] 竹内 理ほか. Hitactix-bsd 連動システムを応用した大規模双方向ストリームサーバの設計と実装. **情報処理学会論文誌**, 43(1):137–145, Jan 2002.

[8] Kasenna. Cost-effective video delivery for large-scale vod solutions (white paper). http: //www.kasenna.com /downloads /white_papers /LargeScaleVODDeliveryWhitepaper.1.8.03.pdf.

[9] Concurrent. Mediahawk content delivery system. *http://www.ccur.com/ products_od_main.aspx*.

[10] Streamonix. Streamonix and mikrom announce alliance to provide video over ip streaming solutions at nab2005. http: //www.streamonix.com/ press/ Streamonix _and _MikroM _Announce _Alliance.pdf.

[11] nCube. n4 streaming media system. *nCube White Paper*, (PN109786), Nov 2000.

[12] Nalini Venkatasubramainian and Klara Nahrstedt. An integrated metric for video qos. *Proceedings of the 5th ACM international conference on Multimedia*, (208):371–380, Nov 1997.

[13] 竹内 理ほか. 外付け i/o エンジン方式を用いたストリームサーバの実現. **情報処理学会論文誌** , 44(7):1680–1694, Jul 2003.

[14] 竹内 理ほか. 軽量仮想計算機モニタを用いた os デバッグ方式の提案. **情報処理学会論文誌**, 46(7):1735–1751, Jul 2005.

[15] Tadashi Takeuchi. Os debugging method using a lightweight virtual machine monitor. *DATE2005*, pages 1058–1059, Mar 2005.

[16] 竹内 理ほか. 軽量仮想計算機モニタを利用した os デバッガのロギング&リプレイ機能の提案. **情報処理学会論文誌**, 50(1):(to be appeared), Jan 2009.

[17] 岩嵜 正明ほか. 連続メディア処理向きマイクロカーネルの開発（1）～（5）. **第 *53* 回全国大会講演論文集（*1*）**, pages 141–150, Sep 1996.

[18] 竹内 理ほか. 連続メディア処理向き os の周期駆動保証機構の設計と実装. **情報処理学会論文誌**, 40(Mar):1204–1215, 1998.

[19] 竹内 理ほか. Os 接続モジュール symbiose を用いた bsd-hitactix 連動システムの設計と実装. **情報処理学会研究報告**, 2000-OS-83:31–36, Feb 2000.

[20] H. Schulzrinne. Real time streaming protocol (rtsp). *RFC-2326*, Apr 1998.

[21] Intel. Accelerating high-speed networking with intel i/o acceleration technology (white paper). http://download.intel.com/ technology/ comms/ perfnet/ download/ 98856.pdf.

[22] Lars Albertsson and Peter S Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. *IEEE 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecomunication Systems*, pages 191–198, Aug 2000.

[23] 安田 絹子ほか. オペレーティングシステム開発環境の設計と実装（言語処理／ os 支援アーキテクチャ、および一般）. **情報処理学会研究報告「システムソフトウェアとオペレーティング・システム」**, 072-011:872–879, May 1996.

[24] 清水 正明ほか. Os デバッグ環境の設計と実装. **情報処理学会研究報告「オペレーティング・システム」**, 057-001:1–8, Feb 1992.

[25] Amit S. Kale. kgdb: linux kernel source level debugger. http://kgdb.sourceforge.net.

[26] SGI 社. Kdb(built-in kernel debugger). http://oss.sgi.com/ projects/ kdb.

[27] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 211–224, Dec 2002.

[28] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent*, (US 6,397,242 B1), Oct 1998.

[29] Intel 社. Ia-32 intel architecture software developer's manual volumes i, ii and iii. 2001.

[30] TOPPERS プロジェクト. Toppers プロジェクト. http://www.toppers.jp.

[31] マクニカネットワークス（株）. Bsdi internet server edition. http://www.networks. macnica.co.jp/ windriver/ index.html.

[32] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, and Tim Harris. Xen and the art of virtualization. *ACM Symposium on Operating Systems Principles*, pages 164–177, Oct 2003.

[33] Jeremy Sugerman, Ganesh Venkiachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. *USENIX Annual Technical Conference*, pages 1–14, Jun 2001.

[34] Carl A. Waldspurger. Memory resource management in vmware esx server. *USENIX 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, Dec 2002.

[35] Intel. Intel 64 and ia-32 intel architecture software developer's manual volumes 3b. 2006.

[36] 標準技法には落とし穴がある 穴をふさぐ現場ルールを作ろう. **日経システムズ**, pages 20–23, Jul 2007.

[37] Domain キーパー社. ハウジングサービス料金表. *http://www.domain-keeper.net/ price_list/ housing/ index.html.*

[38] Gdb: The gnu project debugger. *http://www.gnu.org/ software/ gdb.*

[39] 沖電気. 映像配信サーバ「oki mediaserver」の ipv6 対応版を販売開始. *http://www.oki.com/ jp/ Home/ JIS/ New/ OKI-News/ 2003/ 11/ z03078.html.*

[40] H. K. Jerry Chu. Zero-copy tcp in solaris. *Proceedings of the USENIX Annual Techinical Conference*, pages 253–264, Jan 1996.

[41] Vivek S. Pai, Peter Druschel, and Willy Zwanepoel. I/o-lite: A unified i/o buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, Feb 2000.

[42] Kevin Fall and Joseph Pasquale. Improving continuous media playback performance with in-kernel data paths. *International Conference on Multimedia Computing and Systems*, pages 100–109, May 1994.

[43] Frank W. Miller and Satish K. Tripathi. An integrated input/output system for kernel data streaming. *SPIE/ACM Multimedia Computing and Networking*, pages 57–68, Jan 1998.

[44] Frank W. Miller, Pete Keleher, and Satish K. Tripathi. General data streaming. *19th IEEE Real-Time System Symposium*, pages 232–241, Dec 1998.

[45] P. R. Barham. A fresh approach to file system quality of service. *7th International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 119–128, May 1997.

[46] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the tiger video fileserver. *16th ACM Symposium on Operating Systems Principles*, pages 212–223, Oct 1997.

[47] H. Schulzrinne. Rtp: A transport protocol for real-time applications. *RFC-1889*, Jan 1996.

[48] Robert P. Goldberg. Survey of virtual machine reserach. *IEEE Computer*, pages 34–45, Jun 1974.

[49] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Reserach and Development*, 25(5):483–490, Sep 1981.

[50] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Reserach and Development*, 27(6):530–544, Sep 1983.

[51] 梅野 英典ほか. 仮想計算機システムの高性能化方式. **情報処理**, 31(12):1665–1680, Dec 1990.

[52] Gerald J. Popek and Robert P. Goldberk. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, Jun 1974.

[53] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and network applications. *University of Washington Technical Report 02-02-01*, 2002.

[54] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Symposium on Operating Systems Principles*, pages 143–156, Oct 1997.

[55] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault-tolerance. *ACM Symposium on Operating Systems Principles*, pages 1–11, Dec 1995.

[56] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flightdata recorder" for enabling full-system multiprocessor deterministic replay. *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, Jun 2003.

[57] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. Execrecorder: Vm-based full-system replay for attach analysis and system recovery. *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 66–71, Oct 2006.

[58] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, Jun 2005.

[59] Sudarsha M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay debugging. *Proceedings of the USENIX Annual Technical Conference 2004*, pages 29–44, Jun 2004.

[60] 中野 隆裕ほか. Ethernet 上で qos を保証する通信方法の設計と実装. **情報処理学会論文誌**, 40(Feb):322–332, 2000.

[61] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. *14th ACM Symposium on Operating System Principles*, pages 189–202, Dec 1993.

[62] Microsoft. Wdm kernel streaming architecture. *http://www.microsoft.com/ hwdev/ desinit/ csa1.htm*, Dec 1998.

[63] Peter Bosch. Real-time disk scheduling in a mixed-media file system. *6th IEEE Real Time Technology and Applications Symposium*, pages 23–32, May 2000.

[64] 竹内 理ほか. アイソクロナススケジューラを応用した qos 保証型通信の設計と 実装. **情報処理学会論文誌**, 40(10):3737–3751, Oct 1999.

[65] 岩嵜 正明ほか. Hitactix/symbiose の開発（1）〜（7）. **第 *59* 回全国大会講演論文集（*1*）**, pages 125–138, Sep 1999.

[66] Ian Leslie, Derek McAuley, Richard Black, and Timothy Roscoe. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Area s in Communications*, 14(Jul):1280–1297, 1996.

[67] Johannes Helander. Unix under mach – the lites server. Master's thesis, Helsinki University of Technology, 1994.

[68] 宿口 雅弘. 組み込みシステムのデバッグ手法. **情報処理**, 38(10):886–891, Oct 1997.

[69] 山之内 暢彦ほか. Ieee1394 を利用したオペレーティングシステムの振舞いの測定 方法. **情報処理学会論文誌**, 44(1):29–38, Jan 2003.

[70] Dawson R. Engler, M. Frans aashoek, and James O Toole Jr. Exokernel: An operating system architecture for applicatoin-level resource management. *ACM Symposium on Operating Systems Principles*, pages 1–17, Mar 1995.

[71] 若林 隆行ほか. Itorn デバッギングインタフェース仕様における標準化アプローチとその適応性に関する評価. **情報処理学会論文誌**, 42(06):1503–1513, Jun 2001.

[72] kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: Resource-management using virtual clusters on shared-memory multiprocessors. *ACM Symposium on Operating Systems Principles*, pages 154–169, Dec 1999.

[73] 日立製作所. 高性能映像配信サーバシステム「videonet.tv」を販売開始. http://www.hitachi.co.jp/ News/ cnews/ month/ 2007/ 06/ 0611.html.