*i*Reserve アーキテクチャ：統合的資源予約機構

西尾 信彦[†] 徳田 英幸[†]

より予測可能性の高い計算機環境を構築するために，計算機資源の予約機構は必須であるが，それを実装するシステムレベルの指定方式は，アプリケーション開発のレベルからは厳密過ぎて扱うのが困難である．そこで我々は統合的な資源管理機構として *i*Reserve 機構を設計実装し，連続メディア処理を扱いやすい表現で資源予約管理を抽象化したミドルウェアを開発した．本論文ではその *i*Reserve 機構が，(i) 複数種類の資源予約，(ii) 協調する複数スレッドによる予約の共有とハンドオフ，(iii) 複数階層の QOS 指定表現，(iv) 分散資源管理，のそれぞれをいかに統合化しているかを説明し，それが実現した QOS プロファイリング機能と動的 QOS 翻訳機能，ネットワーク性能保証に関する実験結果について報告する．

*i*Reserve Architecture: An Integrated Resource Reservation Mechanism

NOBUHIKO NISHIO[†] and HIDEYUKI TOKUDA[†]

To construct a more predictable computing environment, especially in end-systems, resource reservation mechanisms are indispensable in the modern multimedia systems. We build an integrated resource reservation architecture named *i*Reserve, and have developed a more comprehensive and powerful environment for continuous media processing middleware. In *i*Reserve architecture, we manage to resolve these open issues on reservation mechamisms: (i) QOS specifications in the system layer are so primitive and platform-dependent that it is useless for application programmers. (ii) multiple resource types handling is not consistent or not unified. (iii) resource reservation sharing/hand-off facility is not fully investigated. (iv) distributed resource reservation has not established yet. This paper describes these integrated reservation mechanisms and reports on the QOS profiling, dynamic QOS translation and network-access performance guarantee experiment within it.

1. はじめに

近年の連続メディア処理に代表されるソフトリアルタイムシステムの開発を支援するには，より予測可能性の高い計算機環境を構築する必要がある．そのために，計算機資源の予約機構は必須である．

計算機資源にはプロセッサ，メモリをはじめ，ストレージやネットワークとのバンド幅などが考えられ，それぞれについての予約機構の研究が種々行われてきた．

このようなシステム開発の一方，連続メディア処理のアプリケーション開発の支援のためにもミドルウェアの開発といった研究がなされている．しかし，両者の研究の間には溝があり，資源予約機構を利用した予測可能性の高いアプリケーションの開発は容易であるとはいいがたい．我々は，以下に述べる様々な問題点を解決するためにより統合化をすすめた資源予約機構 *i*Reserve 機構を導入し，主にエンドシステムにおいてその実装開発を行っている．

近年では，QOS 制御というとネットワークシステム，特にルータ内/間制御での話題がさかんであるが，それを利用するエンドシステムにおいても予測可能性の高い処理は同様に要求される．ユーザの要求を直接的に知ることができるのはまさにエンドシステムであり，予約したネットワークの資源を忠実に使うことが要求されている．また，エンドシステムはクライアントホストにとどまらず，ビデオオンデマンドシステムなどにおけるサーバのアーキテクチャとしても今後も重要な機構である．

我々のシステムが統合化をすすめるうえで扱う問題点としては，主にシステムレベルとアプリケーションレベルでの QOS 表現の翻訳機能に基づくものとして，

(1) システムレベルで管理する計算機資源の使用量などに関する表現と，アプリケーションレベル

† 慶應義塾大学環境情報学部
Faculty of Environmental Information, Keio University

での QOS（Quality Of Service）の表現との間にギャップがあること，

(2) 動的にシステムレベルでの資源の使用量を知る手法が確立していない，

(3) 同じ QOS を実現する場合でもプラットフォームが異なればそれに応じてシステムレベルで要求される資源の使用量は異なってしまうこと，

があげられる．

システムレベルの QOS は通常，資源の使用量についての厳密でかつ定量的な表現，たとえばデバイスとのバンド幅やバッファのサイズといったものが用いられるのに対し，アプリケーションレベルではそれより抽象度が高くビデオフレームレートやフレームサイズといった表現が使われている．この両者間での翻訳は容易ではなく，システムレベルでどれだけの資源が必要になるのかは一般に不明であり，またそのプログラムが走行するプラットフォームの性能にも依存し一意に決定することができない．

さらに，資源予約機能の統合性の欠如からくる事項としては，

(1) 複数種類の計算機資源への対応，

(2) 協調動作する複数のスレッドでの資源予約の共有とハンドオフ，

(3) 分散アプリケーションのための分散資源管理への対応

といった事項がいまだに十分には検討されていない．

複数種類の資源タイプを同時に扱うには，それらの間に潜在的に存在する依存関係を明らかにしなければならない．すなわち，より広いネットワークのバンド幅を使用するには，より多くのプロセッサパワーが必要となり，プロセッサ資源とネットワーク資源とは独立には扱うことができない．また，資源予約機構は通常スレッド単位に割り付けられるが，協調動作する複数のスレッドからなるセッション単位に割り付けられた方が利用には自然である．セッションとして確保した資源を，そこに属するスレッドどうしで共有したりハンドオフしたりするのを支援すべきである．分散アプリケーションについては，クライアントとサーバのホストは各々の QOS 情報をヘテロジニアスな環境にあっても交換できなければならない．これにはプラットフォーム独立な QOS 指定表現もしくは QOS の翻訳機構が備えられることが重要となるであろう．

我々はこれまでに，実時間マイクロカーネルである Real-Time Mach[1] に，プロセッサとメモリに関しての資源予約機構を加えた[3]．このシステムレベルの機能に適切な抽象化を与えてユーザに提供するために，QOS チケットモデルに基づいて実時間周期スレッドを抽象化した Q-Thread ライブラリを開発してきた[2]．また連続メディア処理用のミドルウェアを開発し，それに 3 階層の QOS 表現間での QOS 翻訳機構を組み込んだ[4),5]．本論文では，上で述べた事項を統合的に扱うことのできる資源管理機構としてミドルウェアを再構成するために，*i*Reserve（*i*ntegrated resource Reservation）機構を導入し，その設計と実装およびその動的な適応性能についての実験を報告する．

本論文では，次章で *i*Reserve 機構に基づく統合的資源管理機能について説明する．3 章では，*i*Reserve 機構の Real-Time Mach 上の Conductor/Performer ミドルウェア[6] への実装について説明する．4 章では，*i*Reserve 機構の QOS プロファイリング機能を用いた動的 QOS 翻訳機構および分散資源管理のためのネットワーク性能保証の実験について報告する．5 章では，関連研究との比較をし，最後に結論と今後の課題を述べる．

2. 統合的資源管理機構

我々が提案する *i*Reserve 機構は，OS レベルの資源管理機構とアプリケーション開発との間の溝を埋める統合的な計算機資源管理機構を実現しようとしている．本章では従来の計算機資源管理機構に不足していた事項について検討し，次章で *i*Reserve 機構におけるそれらの対応について述べる．

2.1 複数種類の資源の統合

たとえば動画転送処理といった 1 つのセッション☆ではプロセッサやメモリ，ネットワークなど，複数の資源が必要とされる．新たに資源を 1 つ管理するためにはシステムレベルではその管理機構を個別に構築するが，アプリケーションレベルからは，それらが統合されて抽象化して見せておく必要がある．

QOS 制御に関しては，主にネットワークのバンド幅制御という形での研究がさかんであるが，たとえばプロセッササイクルとメモリ，ネットワークといったような複数の種類の資源に関しての統合的な管理についてこれまで十分には研究がなされていない[7]．我々はすでに Conductor/Performer アーキテクチャにおいて，プロセッササイクルとディスクバンド幅という複数の資源を扱うシステムを構築し，その実装を示している[5]．しかし，このときのディスクバンド幅は理想化されたシミュレーション環境によってなされてお

☆ セッションとは，ある処理を行うために協調動作するスレッドおよびプロセスの集合の意．

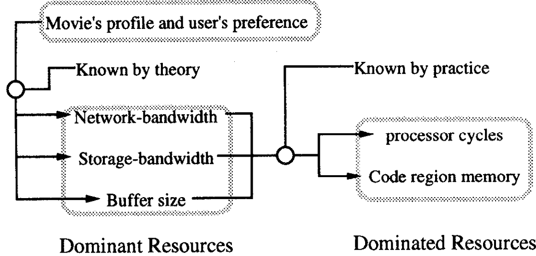

図 1 支配的/被支配的資源
Fig. 1 Dominant and dominated resources.

り，両者の間に存在する依存性を解決した複数の資源の統合はできていなかった．

複数の種類の資源の扱いを困難にしているのは，それら複数の資源の間に依存性があるためである．つまり，ネットワークに大量のデータを流そうとすればそれだけプロセッササイクルも依存して必要になる．システムレベルでの資源量の表現では，各資源の間は独立に扱うことが不可能なので，我々はそれらを「支配的な（dominant）」ものと，「被支配的な（dominated）」ものに分けて扱うことにした．

QOS の指定は本来ユーザから指定されるものであり，その指定から資源量が自動的に算出できるものを支配的と呼ぶ．それらはシステムレベルで表しても一般にプラットフォームに独立な指定表現である．たとえば一定ビットレートの動画転送の場合には，予約すべきネットワークやストレージデバイスとのバンド幅は計算によって得られ，それはプラットフォーム独立である．一方，その支配的な QOS ファクタを実現するためにプロセッササイクルの必要な量は単純には算出できず，プラットフォーム依存の値となる．これらを被支配的な QOS 表現であるという．メモリ資源については，算出可能な側面と困難な側面の両方が存在する．たとえば要求された性能を出すためのバッファとして用いる量などは計算可能で，wire-down すべきテキスト領域については困難である．言い換えれば，理論的にユーザの要求から使用量を導出できない資源を被支配的と呼ぶ．**図 1** に両者の関係を示した．これは動画ファイルのプロフィールとそれを再生したいユーザの希望によって，必要とされるネットワークやストレージとのバンド幅やバッファのサイズは計算すれば分かるが，それを実現するのに必要なプロセッサやテキストエリアに必要とするメモリの量はプラットフォームに依存しており，実際に動かしてみないと分からないことを示している．

複数の資源タイプを統合的に扱えるようになれば，QOS 翻訳についてもその益がある．ユーザレベルでの QOS の表現にはシステムレベルの表現では複数の資源を必要とする場合が多い．ユーザレベルでの要求をシステムレベルでの QOS 表現に翻訳するために，資源間での支配–被支配関係を明確にするべきであろう．

また複数種の資源を扱うときにいままで問題とされてきたのは，それらを要求するセッションが複数あるときに，その間を調停するアルゴリズムがそれ自体で NP-完全問題ともなりうることである．これを解決するには主に 2 つの手法があり，1 つには資源利用についての評価関数を設定し，その合計を最大化することで最適な資源分配を近似するものである[8]．もう 1 つは，セッションが資源を要求する仕方に制約を与え，最適解を容易に得られるようにするものである．我々の手法[4] は後者であり，セッションの QOS 指定には若干の制約が設けられている．

2.2 多階層の QOS 表現の統合（プラットフォームからの独立性）

プロセッサ資源予約などを実装する場合に最も困難な問題の 1 つがプラットフォームへの依存性である．同じ QOS の処理でも，プラットフォームが異なればその性能が異なるため，その処理に必要なプロセッサ資源予約のパーセンテージは一意に決めることはできない．しかしこれを理由として，回復周期分の計算時間（C/T）といった単位でプロセッサ資源予約を発行するというスキーマ自体を否定するのは適切ではない．むしろプラットフォーム独立な QOS 表現とプラットフォーム依存の QOS 表現を使い分け，システムレベルでのプロセッサ予約ではプラットフォーム依存の従来の表現を用いて，その両表現間での翻訳機構を構築するのが現実的である．プラットフォームからの独立性の確保は複数階層にわたる QOS 表現の翻訳機能の確保だと言い換えることができる．

しかしこの翻訳機構は容易には実現できない．前節で述べたように支配的資源の使用量は理論的な解析によって算出可能であるが，プロセッササイクルのような被支配的資源の使用量は一般に算出が困難であり，マシンの性能によっても変化する．そこで，これを解決するためには我々は何らかのプロファイリングもしくは較正機構をシステムに取り入れることとした．

さらに近年ではモーバイルコンピューティングの発展により，ノート型計算機のバッテリ駆動時間を伸ばすために「プロセッササイクリング」と呼ばれる技術が採用されており，同一のハードウェアを用いていても動的に性能が変化してしまう状況が起こりうる．このようなケースで同じ品質での処理を要求するには，QOS 表現の翻訳機構に動的な適応性を持たせなけれ

ば対応できない．以前我々はQOS翻訳機構で静的に変換テーブルを作成していたが，QOSプロファイリング機構を付加することにより動的なQOS翻訳を実現する．

2.3 複数のスレッドの統合

通常1つの連続メディア処理のセッションは複数のスレッドの協調動作により構成される．動画をキャプチャするサーバの中のスレッドとそのクライアントの中のスレッドや，それを描画するX Windowのサーバ内のワーカスレッドは協調して1つのセッションを構成している．このセッションの動的QOS制御を行う場合には，これらのスレッドが使用するすべての資源についてまとめて制御をすべきであるので，これらのスレッドが走行するのに必要な計算機資源の予約はまとめて行われるべきである．

そのために，資源の予約は協調動作するスレッド間で共有，もしくはハンドオフ☆ができるようにするべきである．資源予約の共有とハンドオフでは以下の2つのタイミングで行われる：

- スレッドが新しいスレッドを生成し，それと協調動作するとき，そして，
- スレッドが別のスレッドと通信して，それに自分の仕事を手渡すとき

である．

新しいスレッドを生成するケースで，子スレッドが親スレッドと共同作業をする場合には親スレッドが予約している資源を共有させる．また，もちろん，子スレッドは個別に資源予約をして独立した仕事をするかもしれない．通信するケースではサーバスレッドはクライアントスレッドから要求された仕事をするので，一時的にクライアントスレッドが予約している資源を使えるようにする．この場合は資源予約の「ハンドオフ」もしくは「継承」と呼ばれる．どちらも，資源予約を共有する機能とスレッドと資源予約を束縛する/束縛を解く機能を用意すれば実装できる．ただし，各資源ごとにその機能が実装されていなければならない．また，既存のソフトウェアを極力書き換えることなく利用するためには，この機能はカーネル内で実装されている必要がある．

2.4 分散環境での資源の統合

連続メディアシステムはネットワークを隔てたホスト間で構築することもあるので，分散環境での資源管理も重要であるが，これもネットワークのバンド幅や遅延，エラーレートといった観点からの研究を除いては，エンドシステムどうしでの資源管理を統合する試みは単にフレームワークの提示[9]を除いてはほとんどなされていない．

この分散環境での資源管理には前項のプラットフォーム独立性が確保されていることが必須である．それが確保されたうえでプラットフォーム独立でのQOS表現のみをホスト間でやりとりすることによって実現できる．

また，前項の複数のスレッドによる資源予約の共有もしくはハンドオフ機能も重要となる場合もある．もし複数のホストにわたって共通の資源予約管理を行う場合には，遠隔のサーバにクライアントからの要求が受け付けられるときにプラットフォーム独立のQOS表現による資源予約がハンドオフされ，かつエンドシステムにおいてそれがプラットフォーム依存形式に翻訳されたうえで資源予約管理が行われなければならない．また，ネットワークプロコトルのように比較的重い処理がカーネル内で実装されていると，セッションごとのスケジューリングは困難になるため，このハンドオフや共有機能は，ネットワークプロトコルがサーバ形式などのユーザレベルで実装されている場合にも重要となる．さらに，支配的資源であるネットワークアクセス性能を予約する場合には，プロセッサやメモリといった被支配的資源への翻訳が必要となる．このためにもユーザレベルでのネットワークプロトコルの実装は重要である．

3. *i*Reserve機構とその実装

前章であげた事項に対応するための統合的資源管理機構として，我々は*i*Reserve機構（*i*ntegrated resource Reservation mechanism）を実装している．本機構は，Real-Time Machマイクロカーネルと4.4BSD Liteサーバによって構成される環境をベースとした連続メディア処理ミドルウェアであるConductor/Performerアーキテクチャに組み込まれ，EISS（環境情報サーバスイート）[10]との協調動作を行う☆☆．図2に我々が実現している*i*Reserveの予約オブジェクトの概念図を示した．ここでは*i*Reserveが，プロセッサやメモリやデバイスとのバンド幅の予約を束ねる，より抽象的な予約オブジェクトを実現することを表し，そこでQOSの翻訳とプロファイリング，複数のスレッドでの共有の機能があることを図示した．

☆ ハンドオフとは一方のスレッドがもう一方のスレッドに自分が予約した資源を一時的に共有することを許すことを意味する．

☆☆ EISSはモーバイル計算機環境などにおける計算機資源の動的な変化に対応するため，カーネル内で起きたイベントをユーザレベルに配送する役割を担う．

表 1 3-層 QOS 表現
Table 1 3-layer QOS specifications.

階層	表現	使用主体	表現例
ULQ	抽象的，一意的ではない	エンドユーザ	good/fair/poor
MLQ	H/W 独立，一意的	ミドルウェア	24 fps，22 KHz
SLQ	定量的，H/W 依存，一意的	カーネル etc.	18%，20 MB/s

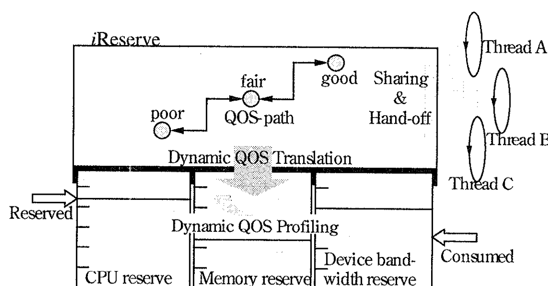

図 2 *i*Reserve オブジェクト
Fig. 2 *i*Reserve object.

以下では，*i*Reserve が実装する各機能について説明する．

3.1 QOS パス

我々は Conductor/Performer ミドルウェアにおいて「QOS パス」と呼ばれる QOS の表現を用いている．これはユーザがセッションに対して指定するもので，そのセッションがとるべき QOS 指定の候補を QOS（資源要求量）の高いものから低いものへと順序関係をつけて並べたものである．このとき，各候補点の要求する資源量はその前後の候補点が必要とする各資源の要求量の間になければならない．この単調増加性の制約を受け入れることによって，複数セッション間の資源の割り振り調整に必要な計算を単純化している．

QOS パスの各 QOS 指定の表現はユーザから渡された後，システムが扱いやすいように翻訳が行われる．次節でその翻訳機構について説明する．

3.2 QOS 表現の翻訳機構

我々は Conductor/Performer ミドルウェアにおいて，QOS パスの各 QOS 指定を表現するのに，3 階層の QOS 表現を導入している．この階層は，ユーザレベル（ULQ; User-Level QOS）とミドルウェアレベル（MLQ; Middleware-Level QOS）と，システムレベル（SLQ; System-Level QOS）である．

ULQ は最もエンドユーザレベル寄りでの優良可といった抽象的な表現である．このような表現では要求している処理の一意性はない．この抽象的な表現から，30 fps とか 24-bit カラーといったような一意性を持った QOS 表現に直したものが MLQ である．MLQ では要求される QOS の一意性があり，この表現では処理するハードウェアプラットフォームに依存する表現を含まない．これをまた，システムレベルでの資源予約機構に則した，プロセッササイクル 20%とか，2.5 MB の物理メモリ，といったプラットフォーム依存の表現にしたものが SLQ である☆．**表 1** にこの表現の階層についてまとめた．

ULQ から MLQ への翻訳は，スライダやボタンなどの GUI のためのオブジェクトを管理するライブラリによって行われる．この翻訳に関しては本質的な問題はなく，ただ両者の対応をアプリケーションやそのライブラリのレベルで定めればよい．しかし，MLQ から SLQ への変換に関しては，本質的な問題がある．すなわち，プロセッサ使用率などの被支配的資源については単純には算出できないためである．従来の Conductor/Performer ミドルウェアではこの翻訳のため，事前に個々の処理レベルに応じてどれだけのシステム資源を必要としたかを計測し，それを用いた変換テーブルを静的に生成しておき，実行時に参照するという形で対処していた[5]．しかし，このようなテーブルを作成するのは容易ではないし，各プラットフォーム（ホスト）ごとに作成する必要があった．

*i*Reserve モデルでは，この翻訳テーブルの作成を動的にも行うために QOS プロファイリング機能を実装し，被支配的資源についてのみそれを適用して，実際に消費された資源量から動的に翻訳テーブルを生成する．次節でこの動的生成機能について説明する．

*i*Reserve では，プロセッササイクルと（テキスト領域に割り当てる）メモリ資源を被支配的資源として扱い，それ以外のストレージバンド幅やネットワークバンド幅を支配的資源として扱う．

3.3 QOS プロファイリング機能

現在の Real-Time Mach マイクロカーネル[11]では楽観的なプロセッサ資源予約機構[14]を採用している．この機構では，従来の資源予約機能が強制していたポリーシング機構をオプションの 1 つとしており，資源使用量をモニタリングする機構と予約量を超えた場合

☆ SLQ はすべてがプラットフォーム依存の表現ではない．指定されたフレームレートを実現するネットワークバンド幅などは非依存の表現である．しかし，それを実現するために必要とされるプロセッサ資源は被支配的な資源である．

(オーバラン) の通知機構を有している. つまり, オーバラン時の対応としては以下の3つのオプションから選択できる:

- 次の回復周期までサスペンドする.
- バックグラウンドの優先度で走り続ける.
- 同じ優先度で走り続ける.

それぞれがプロセッサ予約オブジェクト (プロセッサ予約) ごとに選択できる. 最初のオプションが従来の手法である. 残りの2つが楽観的なオプションである. 通常, プロセッサ予約機構では, スレッドが自分の予約した使用量以内である限り, プロセッサ予約に指定された高優先度に格上げして実行される. ここでバックグラウンドの優先度と呼ぶのは, 格上げされる前のスレッド自身につけられた優先度のことである. 2つ目のオプションを選択すればスレッドは予約した範囲以上にプロセッサを使用することもあるが, 適切に優先度をつけることによって他の予約をしたスレッドに悪い影響を与えることは防げる. さらに3つ目のオプションではオーバランした場合でも同じ格上げされた優先度で走り続けるものである. この場合は以下に述べるオーバラン通知機能と併用することが勧められる.

モニタリング機能では, 実際にどのくらいのプロセッササイクルを消費しているかを, そのときのプロセッサ使用のパーセンテージと占有時間の累計といった統計量で知ることができる. また, プロセッサ予約ごとに, オーバラン発生時にメッセージを送信するポートを登録することが可能である.

*i*Reserve 機構ではこのモニタリング機能を用いて, QOS のプロファイリングを行い, 与えられた QOS 表現を実現するために必要なプロセッササイクルを動的に決定する.

また Real-Time Mach のメモリ資源予約機能では, ページの wire-down 以外に,

- wire-down するページ数の予約機能
- 予約を超えて wire-down しようとした場合の通知機能
- アクセスしたページを wire-down する機能
- wire-down するかどうかのポリシーを設定する機能

を持っている.

*i*Reserve 機構では, 動的に割り付けるバッファメモリに関しては支配的 QOS ファクタとして扱われ, テキスト領域に関しては被支配的 QOS ファクタであるとして扱われる. 被支配的資源については wire-down するページの制限を予約し, 実時間スケジューリングポリシーを選択しているときにアクセスしたページを wire-down するにように設定する.

3.4 *i*Reserve 遅延予約機能

*i*Reserve では, まずプロセッサ予約機能を用いて処理に必要なプロセッササイクルの QOS プロファイリングする. このためには, 予約は最小限の1%として, それ以上に必要とするプロセッササイクルはバックグラウンドの優先度で走らせることによってプロファイリングする. これは, プロファイリングにより, 他に *i*Reserve を用いて稼働しているセッションに影響を与えないためである.

プロファイリングのためには, オーバラン時のメッセージ受信を契機とする. このメッセージは1%以上のプロセッササイクルを消費すれば受信するので, 何回かこのメッセージを受信することを繰り返して, 実際にこの処理に必要なプロセッササイクルを算出する.

このプロファイリングにより得られたプロセッササイクルのパーセンテージを再度予約し直すことにより, 予約処理が完了する. このように実際の資源予約はセッションの開始から遅延するために, この機能を遅延予約と呼ぶ.

このとき,

- 何をトリガーに予約をする (どれだけ遅延する) か,
- QOS パスで指定された QOS のどの候補点で始めるか,
- プロファイリングされたプロセッササイクルから予約量をどう計るか,

は, そのときに, この *i*Reserve を各セッションに割り振る QOS マネージャ (我々の場合には Conductor) がどのようなセッション調停のためのポリシー☆を選択するによって決定される.

メモリ資源予約に関しても遅延予約方式をとる. メモリ資源に関しても, 予約した wire-down ページ数をオーバするとイベントを受けることができる. バッファメモリなどの支配的 QOS ファクタに関しては明示的なページの wire-down を行うが, テキスト領域に関してはこのイベントを参考に遅延予約を行う.

ノート型計算機などに見られるプロセッササイクリングへの対応は, EISS からの AC 電源の有無に関する環境変化のイベント受信を契機として, この QOS プロファイリングによってオーバランイベントの受信によって行う. オーバランイベントのみを契機としないのは, 動的な性能の変化の原因を見い出すことができず, それが一過性のものなのか恒常的なものなのか

☆ どのようなポリシーがあるかは文献 5) を参照されたい.

の判断がつかないという点と，AC 電源が接続されたことによる性能の動的向上にはオーバランイベントだけでは対応できないためである☆．

3.5 *i*Reserve でのスレッド間共有とハンドオフ機能

*i*Reserve 機構では協調する複数のスレッドに対応して予約の共有とハンドオフをユーザレベルでサポートするための機能を持つ．これは，*i*Reserve を利用するプログラマは *i*Reserve とスレッドとの束縛関係を記述できることを意味する．これは，クライアントスレッドとサーバ内で要求に応えるスレッドの対応が動的にしか分からない場合には束縛関係を持つためのオーバヘッドがある．

我々が *i*Reserve 機構を組み込む Conductor/Performer ミドルウェアでは，アプリケーションが Conductor のクライアント，Conductor が Performer のクライアントというサーバ–クライアント方式であるので，複数のスレッドでの *i*Reserve 共有の機能は必須である．Conductor/Performer ミドルウェアでは 4.4BSD Lites☆☆の機能は使っておらず，セッションを確立するときに，サーバ内のどのスレッドがクライアントの要求を処理するかが事前に決まるので，*i*Reserve のインカーネルレベルでのハンドオフ機能は必要としない．単に *i*Reserve を共有できればよいので，動的に予約との結合を解消して担当スレッドにバインドするオーバヘッドがかからない．

ただし，動的にスレッドが生成されそれが *i*Reserve を共有しなければならない場合には，インカーネルでのハンドオフ機能が必要となる．また既存のソフトウェアを改変せずに動作保証するためにもインカーネルでの実装は今後重要であろう．

協調する複数のスレッドが消費するメモリ資源の管理については，Real-Time マイクロカーネル内には共有のための機能がないので *i*Reserve のレベルで対応する．これは複数のスレッドが予約したメモリ予約からのページ wire-down の超過使用イベントを共通の 1 つのポートで受けることにより実装する．

3.6 *i*Reserve 機構での分散資源管理

分散資源に関する *i*Reserve の役割は，上位で階層での複数の分散した環境での性能の差の隠蔽と，下位でのネットワークバンド幅などの転送品質の保証とに大きく分かれる．ネットワークにより接続された分散環境はホスト自体の性能だけではなく，ホストに接続された NIC の性能，接続形態，ルータやスイッチの性能によって左右される．本論文では，NIC およびルータやスイッチの性能といった伝送路での資源予約については触れない．これは RSVP や WFQ をはじめとして広く研究が行われており，ここでは，本論文が対象としているエンドシステム内のシステムソフトウェアによって制御可能な資源予約管理に集中して考察する．

この上位でのプラットフォーム非依存性の実現に関してはすでに述べてきたので，以下では下位のネットワーク性能の保証に関して *i*Reserve での実装方針に関して説明する．

3.6.1 *i*Reserve 機構でのネットワーク性能保証

ネットワークアクセス性能の保証はアクセス事前のアドミッション制御とアクセス時の処理保証とから構成される．アドミッション制御はプロセッサ資源の予約などと同様にクライアントからの予約量の事前申告により有限の資源量の限界使用への調整とクライアントにつけられた優先度に応じたネットワーク資源の割当てをチェックする．これは主にユーザレベルに設けられたマネージャによって実現できる．

次に実際にネットワークへのアクセス性能を予約する手法について検討する．これはネットワークアクセスの機構がシステムにどのように実装されているかによってその実装を適合させなければならない．その機構は**インカーネルプロトコルスタック方式**と**ユーザレベルプロトコルスタック方式**の 2 つに大別できる．前者は，ネットワークアクセスの通常の実装方式であり，ネットワークへのアクセスはアプリケーションプログラムから socket 関連のシステムコールを発行することによって行われる．すなわち socket レイヤより下位のプロトコルスタックに含まれる処理はカーネル内で行われる．また，マイクロカーネル方式のシステムでは後者の方式がよく採用される．ネットワークアクセスのための socket 関連のシステムコールはユーザレベルプログラムである OS パーソナリティサーバやネットワークプロトコルサーバとして実装される．アプリケーションプログラムはこのサーバ経由でネットワークアクセスを行う．このとき，サーバ内でプロトコル処理が行われ，マイクロカーネル内ではデバイスドライバ処理などの最下層の処理のみが行われる．

インカーネルプロトコル方式でネットワーク性能を保証するためには，アプリケーションプログラムの処理の予約とともにカーネル内でのプロトコル処理にも予約が必要である．ただ，カーネル内の処理は基本

☆ ただし，現在の APM 機構では，このようなイベントは割り込みはかからずポーリングしなければならない．我々の実装ではこのポーリングの周期は 1 秒である．

☆☆ RT-Mach マイクロカーネル上で 4.4BSD Lite 機能を実現する OS パーソナリティサーバのこと．

的にスケジューリングすることは困難であり，socket システムコール以下を大幅に変更拡張しなければ複数のセッション間の予約保証は実現できない．すなわちカーネル内部での処理はできる限り小さくし，もしネットワーク機器自身が性能保証機能を持つのであれば，それを利用するだけの小さなデバイスドライバのみをカーネル内に残すのが都合がよい．その意味では第2の方式であるユーザレベルプロトコルスタックの方が資源予約保証をするにはカーネルへの変更を最小限に抑えられるので適用しやすい．

ユーザレベルプロトコルスタック方式では，ネットワークアクセスはアプリケーションプログラムとプロトコル処理を行うサーバ，そして最小限のデバイスドライバを持つマイクロカーネルの3者によって実現される．プロトコルサーバはユーザレベルで処理されるので，性能の保証に関してはアプリケーションプログラムと基本的には同様であるが，サーバには複数のセッションからのアクセス要求が集中するので，セッションごとに性能を保証しなければならない．すなわち，すべてのアクセス要求に対してアプリケーションの予約をハンドオフもしくは一時的に共有することにより，セッション単位での性能を保証する必要がある．この方式に関しては，Real-Time Mach マイクロカーネル上での NPS（Network Protocol Server）の実装において実現されている[12]．そこではアプリケーションプログラムからの IP データグラムに優先度をつけて NPS に渡されると，その優先度別のキューとスレッドによりプロトコル処理がなされることによって性能保証が実装されている．

ユーザレベルプロトコルスタック方式では性能保証機構を統合しやすい反面，IPC が余分に増えるためにインカーネル方式ほどの性能を期待できない．この性能の低下はプロトコルスタックをライブラリ化してアプリケーションにリンクすることで相当に向上できる[13]．この方式はユーザレベルプロトコルスタック方式に分類できる．ネットワークアクセスの処理のすべてをプロトコルスタックのあるサーバを経由するのではなく，セキュリティ保持のためにセッションの確立までをサーバで処理する．データの転送処理はアプリケーションプログラムにもプロトコル処理を行うライブラリをリンクしておき，プロトコルサーバとの通信なしに，プロトコルライブラリから直接デバイスドライバへの要求を発行するものである．

本論文ではこのライブラリ化したプロトコルスタックの方式を採用したネットワーク性能について実装評価している．アプリケーションプログラムにこのプロトコルライブラリをリンクし，アプリケーションプログラムの処理のみ☆を予約することによって，ネットワーク性能を保証している．

3.6.2 *i*Reserve ネットワーク性能保証機能の実装

本機能の実現には Real-Time Mach マイクロカーネル上で 4.4BSD lite サーバを稼働させた環境で，TCP/IP プロトコルスタックをライブラリ化したものを用いる．このライブラリでは，バークレーソケットのうち socket，bind，listen，accept，close は lites サーバで行い，それ以外の送受信機能をライブラリとして実行できるようにしたものである．ライブラリの中から Mach マイクロカーネルへ直接 `device_write`，`device_read` を発行している．

パケットの受信のためには，ライブラリから受信用のスレッドを生成し，Mach マイクロカーネル内のパケットフィルタリング機能を用いて，入力されたパケットのポート番号を先読みすることによってこの受信スレッドにパケットに振り分けている．

このネットワークプロトコルライブラリをリンクするアプリケーションプログラムに対して，プロセッサ資源予約機構を用いて回復周期ごとに資源使用量を計測できるようにする．また，その周期に合わせてネットワークアクセスの量も同時に計測することによって，ネットワークアクセス性能とプロセッサ資源使用量との相関をプロファイリングする．

ネットワークアクセスを要求するアプリケーションプログラム側が周期スレッドを用いるなどの手法で流量制御をする場合には，オーバランポリシーにバックグラウンド優先度での走行継続指定をしたうえで1%予約して，プロファイリングを行ってそれに基づいて遅延予約を行う．しかし，アプリケーション側で流量制御をしていない場合には，プロセッサ予約のオーバランポリシーとしてサスペンド指定をすることによって，適切な流量に安定させなければならない．この場合には，予約を生成する時点で必要とするネットワークアクセス性能を指定することによって，それに必要な資源を遅延予約して実現する．

4. *i*Reserve 機構の評価実験

4.1 *i*Reserve の QOS プロファイリング機能の実験

*i*Reserve 機構が持つ QOS プロファイリング機能を実証するために，AC 電源の抜き差しによって，ある

☆ ただし，スレッドの数は4つあり，それで1つの *i*Reserve を共有している．

表 2 3 つのスレッドの諸元
Table 2 Information on three threads.

Thread	Thread priority	*i*Reserve priority	Period
A	13	5	30 msec
B	12	N/A	30 msec
C	12	N/A	forever loop

処理を行っているスレッドがプロセッサ予約の何パーセントを消費しているかをプロファイリングし，それに合わせて遅延予約をする実験を行った．実験環境は以下のとおりである．実験マシンはプロセッサが MMX-Pentium 233 MHz，512 KB セカンドキャッシュ，96 MB メインメモリ，3.2 GB ハードディスクの IBM-PC/AT 互換ノート PC で，Real-Time Mach/Lites（MKG008，1998-05-12 snapshot）を稼働させた．実験マシンでは，AC 電源供給時には，プロセッササイクリングは起こらず 100%の稼働をするが，バッテリ駆動時には，50%でのプロセッササイクリングを行うように設定した．

最初，マシンは AC 電源を接続しておき，その後それを切断し，最後に AC 電源に再接続する．この実験のために準備した 3 つのスレッド（A，B，C）は Fibonacci 数列の計算を行う．このうちの 2 つ（A と B）は 30 msec の周期スレッドである．スレッド A は *i*Reserve つきであり，スレッド B はなし，スレッド C もなしだが，スレッド C は周期スレッドではなく終了しない（非常に時間のかかる）数列計算を無限に行う．周期スレッドの方の計算は周期内に完了できる計算量である．この 3 つのスレッドの諸元を表 2 に示した．優先度は値が少ないほど高く，スレッド A は自分に与えられた資源予約量をオーバランしなければ優先度 5 にアップグレードされるが，オーバランしたときには，優先度は 13 となり，3 者のうち最低となる．これは *i*Reserve のオーバランポリシーとして RESERVE_POLICY_BACKGROUND が選ばれているためである．スレッド B と C とは標準プロセスの優先度である 12 がともに与えられている．スレッドのスケジューリングポリシーは固定優先度方式であり，同一優先度スレッド間では 10 msec でのラウンドロビンを行っている．

QOS プロファイリングと遅延予約は，オーバランメッセージと EISS の APM 関連イベントメッセージによってトリガーされる．プロセッサ予約機構の時間分解能は 1 msec であり，プロセッサ予約の回復周期は予約機構の同期機能によってスレッド A の周期 30 msec に同期している．スレッド C はスレッド A と B の処理をたえず妨害し，C スレッドの処理は実験中には終了しないのでつねにプロセッサの使用率は 100%である．

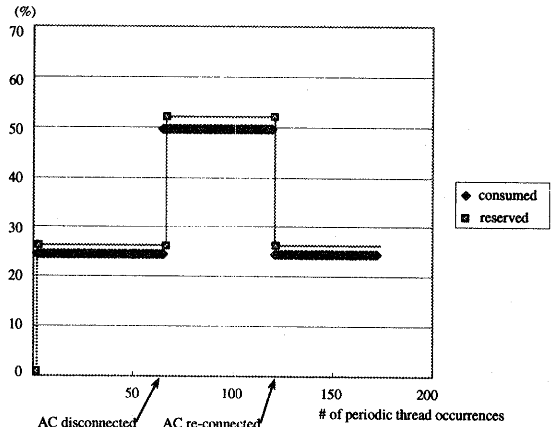

図 3 QOS プロファイリング実験（1）
Fig. 3 QOS profiling experiment (1).

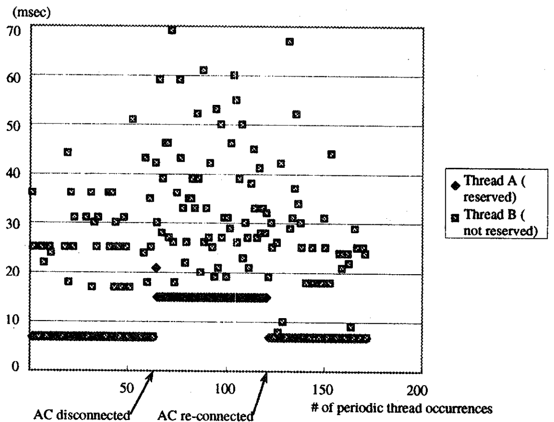

図 4 QOS プロファイリング実験（2）
Fig. 4 QOS profiling experiment (2).

図 3 では *i*Reserve におけるプロセッサ資源予約の実際の使用量とその予約量の推移を示している．ここではスレッド A がバインドしている予約をしている *i*Reserve の予約量と消費量を表している．縦軸はプロセッサ資源の使用量のパーセンテージであり，横軸は周期スレッドの回数単位として経過時間を表している（1 単位 30 msec）．図 4 では，横軸は同じだが，ス

レッド A と B が各周期の処理にどれだけの時間を必要としたかを縦軸としている☆.

以上により，*i*Reserve つきのスレッド A は，

- すべての周期で処理が時間制約を守れている，
- AC 電源の有無によりマシンの性能が動的に変化した際にもプロセッサ予約が動的に変化し乱れがなく，浪費的な予約もない，

ことが観測された.

4.2 *i*Reserve の QOS プロファイリング実験の考察

オーバランの検出と QOS プロファイリングは前節の実験では単純で頻度も低いが，プロファイリングにより遅延予約が有効に働き一定の性能を持続することが実現できたことを示した．しかし，さらに処理内容の動的な変化による資源要求量の増減にいかに適切に対応してプロファイリングと遅延予約や再予約を行うかは重要であり，その適切な較正は単純ではない．以下では周期スレッドを *i*Reserve とバインドしたときの適応について考察する．

オーバランは予約の 1 周期中に周期 * 予約使用率よりもプロセッサを消費したときに検出されメッセージが飛ぶ．予約の周期は回復周期のことで，基本的には予約生成時の `reserve_attribute_init()` での引数で与えたものだが，バインドしたスレッドが周期的だとその周期と同期するようになる．

この場合，最悪でも処理状況に変化があってからオーバランを検出するまでに 1 周期 * (1+予約使用率) だけの遅延がある．これは前回の周期の終了直後に変化が起き，それまでの消費が限りなく 0%だったときである．ただし，今回の周期が他のスレッドのために遅れる場合もあるので，そのときにはデッドラインミスの方が先に検出されるであろう．そのときには状況変化から検出まで最悪 2 周期の遅延である．ただし，このときには自分の処理状況の変化が起きているかどうかは判断できない．

処理自体が周期スレッドであると，デッドラインミスも検出できるが，状況の変化の検出は本質的にはデッドラインミスによってではなく，オーバランによって検出される．デッドラインミスによって検出されるのはスケジューリングの不適合である．オーバラン検出よりも先にデッドラインミスが起きるのは本来，スケジューリングの適応をすべきか，アドミッション制御に失敗しているかである．

次に，オーバランが検出された後の処理について検討する．まず，処理状況がどのように変化しているのかを知るためにプロファイリングの必要がある．オーバランの検出はこのプロファイリング処理のトリガーにすぎない．プロファイリングに必要な遅延はどうなるか．オーバランを検出したということは，すでにその予約オブジェクトにどのくらいオーバランしたかの情報があるので，最短で，超過した処理の終了時にプロファイリングが可能になる．*i*Reserve のプロセッサの予約では，前回の周期の予約使用率と，現在の周期での使用量が観測できる．

次回の周期の開始までに予約を再設定するためには，オーバラン検出時にはプロファイリング開始の必要ありのフラグを設定するだけでもよい．実際，プロファイリングが 1 周期分だけでよいかどうかはアルゴリズムにもよる．

4.3 ネットワーク性能保証機能の評価実験

*i*Reserve 機構におけるネットワーク性能制御/保証機能の評価は以下の 2 種類の実験によって行った．まず，UDP プロトコルを用いてデータの転送を行うプログラムに与える予約量を変化させることによって転送性能の変化を計測した．この実験では，処理性能の異ったプラットフォームで同様のソフトウェアによる転送実験を行って，ネットワーク性能保証がプラットフォームに依存せず行えるかどうかを評価した．

続いて，上記の実験に対して外乱を起すスレッドを起動して転送性能の変化を計測することによって予約保証が実現できているかを調べた．

4.3.1 評価実験環境

データ転送は最新の Real-Time Mach マイクロカーネルと 4.4BSD lite サーバで稼働するホスト（以下 Lites）と FreeBSD-2.2.5R で稼働するホスト（以下 FreeBSD）の間で行った．2 台のホストは性能の違いを出すために Pentium/166 MHz，メモリ 32 MB の PC/AT 互換機（以下 Pentium）と，Celeron/333 MHz，メモリ 128 MB のやはり PC/AT 互換機（以下 Celeron）を用いた．この 2 台の性能の異なるホストをスイッチングハブ経由で 100 Base/TX で接続した．NIC にはどちらのホストにも DEC の 21140 チップを用いた同じもの（de0 デバイス）を用いている．この 2 台以外のパケットは発生させない☆☆.

UDP パケットを送信するアプリケーションには ttcp というネットワーク性能評価のためのフリーウェ

☆ 処理完了に要する時間はプロセッサ資源の消費時間とは異なることに注意されたい．これにはスレッドが横取りされている時間も含まれている．

☆☆ これは輻輳による性能への影響の回避は本論文では扱わない立場とするためである．ここではデバイスドライバ以下の層，NIC のファームウェアなどによっての解決を想定している．

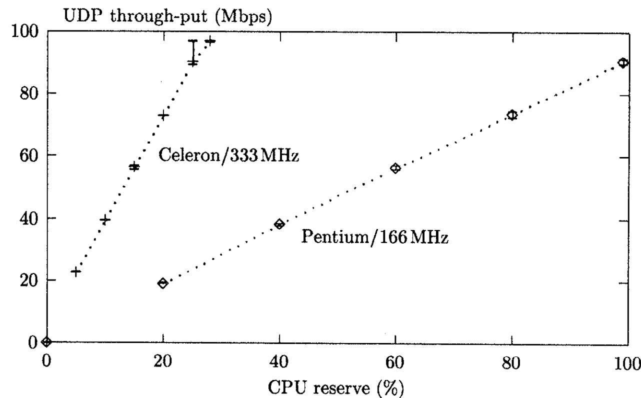

図 5 転送性能制御評価実験
Fig. 5 Network-access performance control experiment.

ア☆を用いた．ttcp を TCP/IP プロトコルライブラリである libsockets.a をリンクし，*i*Reserve 機構を利用するためのモジュールを追加してビルドした．このとき ttcp コマンドの起動オプションに *i*Reserve への性能保証指定（予約資源量）を追加した．

プロセッサ予約機構の設定は，予約範囲内の使用時には高優先度を与えて，オーバランしたときにはサスペンドするようにした．オーバラン時にバックグラウンドの優先度で動作させるようにすると予約以上のネットワーク帯域を使用してしまうためである．スケジューリングポリシーとしては固定優先度+FIFO 方式を採用している．ホストのクロック分解能は 1 msec に指定し，プロセッサ予約の回復周期は 100 msec とした．

4.3.2 転送性能制御評価実験

送信側のホストで Lites を稼働し，受信側のホストで FreeBSD を稼働し，Lites ホストの *i*Reserve 版 ttcp から FreeBSD ホストに UDP パケットを送信した．送信側ホストが Pentium ホストの場合と Celeron ホストの場合での性能を計測した．本実験ではほかに走る干渉スレッドがいないのでオーバラン時のポリシーをサスペンド指定として流量制御は *i*Reserve によって行った．

図 5 に結果を示した．横軸はプロセッサの予約率（単位は%）であり，縦軸はスループット（単位は Mbit/sec）である．1472 バイトのパケットを 21000 回送信（約 30 MB）する処理ごとのスループットの 10 回計測し，その最大と最小と平均をプロットした☆☆．Pentium ホストからの送信の場合も，Celeron ホストからの場合もプロセッサ資源の使用量とほぼ比例してスループットが変化することが分かる．また，各計測点における偏差もほとんどないことが分かる．

4.3.3 転送性能保証評価実験

前項で述べた実験環境と同様に，Pentium ホストで Lites を稼働し，Celeron ホストで FreeBSD を稼働し，Lites ホストの *i*Reserve 版 ttcp から FreeBSD ホストの ttcp に対して UDP パケットを送信した．ただし，ここでは干渉するプログラムを Lites ホストで同時に動かしている．

干渉プログラムとしては，プロセッサ資源についてのみ干渉するプログラムと，主にネットワーク資源について干渉するプログラムとの 2 種類ついて計測した．前者は Fibonacci 数を計算するプログラムで転送実験の間中つねに動作させていた．このプログラムしか動作していない環境ではプロセッサ資源を 100%独占してしまう．後者のネットワーク資源を利用する干渉プログラムは *i*Reserve 機構を利用しない通常の ttcp を用いた．この干渉用の ttcp プログラムは，Pentium ホストから Celeron ホストの別のポートに対して UDP パケット転送を流量制御なしで行う．

どちらの場合も *i*Reserve 版の ttcp は予約が有効な間は干渉プログラムよりも高い優先度が与えられ，オーバランしたときにはサスペンドするように指定した．

図 6 にプロセッサ資源への干渉の実験結果を，**図 7** にはネットワーク資源への干渉の実験結果を示した．横軸ともにはプロセッサの予約率（単位は%）であり，縦軸はスループット（単位は Mbit/sec）である．1472 バイトのパケットを 21000 回送信（約 30 MB）する処理ごとのスループットの 10 回計測し，その最大と最小と平均をプロットした．また，参考のためにいず

☆ ftp.sgi.com にて入手．
☆☆ 1 パケットのサイズを 1472 バイトとしたのは，libsockets が IP フラグメントの処理の実装を省略しているためである．

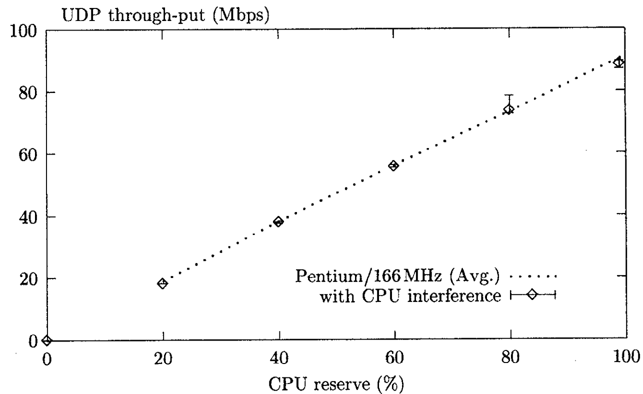

図 6 転送性能保証評価実験（CPU）
Fig. 6 Network-access performance guarantee experiment (CPU).

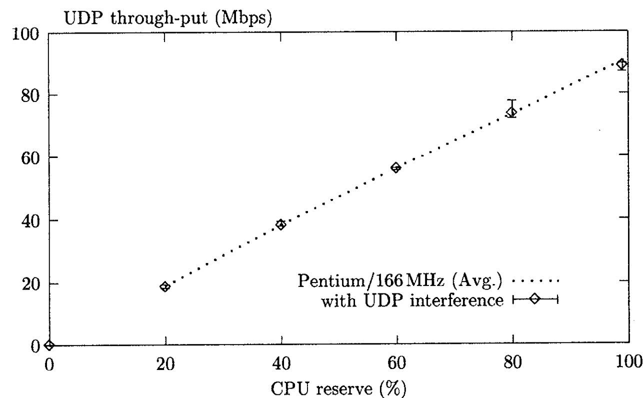

図 7 転送性能保証評価実験（UDP）
Fig. 7 Network-access performance guarantee experiment (UDP).

れの場合にも，干渉プログラムがいないときの性能の平均値のみを点線でプロットしてある．

いずれの場合にも各計測点がほぼ無干渉時の点線上に乗っており，干渉スレッドによるスループットへの影響は見られないことから，*i*Reserve の予約保証機能が有効に働いていることが分かる．また，各計測点における偏差もほぼない安定した性能を実現している．

4.4 ネットワーク性能保証機能の評価実験の考察

ネットワークプロトコルのライブラリ化により，UDP 送信に関しては性能差のあるプラットフォームでも，干渉負荷を与えても偏差の少ない安定した性能を実現した．このことにより，ネットワーク性能保証に関してプラットフォームに依存しない性能保証が期待できることが分かった．

実験では UDP を選択したが，これは *i*Reserve 機構の主な適用範囲として連続メディア処理を考慮したためである．実際に実時間性のあるデータの送信に利用される RTP も UDP 層の上に実現されている．

TCP のような信頼性のあるプロトコルは，以下の 2 点について連続メディア処理には適していない．まず，信頼性のために行われるパケットの再送は連続メディア処理では無駄になることが多く，かえって実時間性を阻害してしまう．次に，end-to-end の輻輳制御のため必要以上に自律的に帯域を狭めてしまう点である．

UDP はこれらの点に関しては何もしないので弊害は起こさないが，その反面パケットの送出間隔の制御がユーザに任されているので，この制御が適切に行われることが *i*Reserve 機構の目指すプラットフォーム非依存性についても重要である．送出間隔制御は，*i*Reserve のプロファイリング機能とも関係する．すなわち処理が平滑化されていないほどプロファイリングした数値にばらつきがでてしまうからである．

今回の実験では ttcp が送信バッファのオーバフローを ENOBUF の返り値で認知しスリープ（18 msec）している．これはアプリケーション層での制御だが，*i*Reserve 機構のいまの実装では回復周期単位での制

御をしているが，割り込み処理頻度を考慮してミリ秒オーダであり，UDP のパケットはマイクロ秒オーダで送出されていて，今後のギガビット時代にはさらに追いつけなくなる．特にネットワークに関しては，多少の遅延時間が増えるが送受信バッファを大きめにとるか，ハードウェアでの実現を期待することになろう．その意味では送受信バッファは被支配的な資源となる．

我々はネットワークアクセス性能保証機構にライブラリで実現したプロトコルスタック方式を採用し，プロセッサ予約については周期的な回復処理方式を採用したが，これであらゆる場合に適切であるとはいえない．回復周期による予約はパケット受信については最適とはいえないし，予約のハンドオフ機能をインカーネルで実現してユーザレベルサーバ方式のサポートも有効であろう．

実装をさらに適切なものに拡張していくことは *i*Reserve の上位層に影響を与えない限りは可能である．今後は，たとえばパケット受信に関しては（悪意のアタックのない環境では）プロセッサ予約をしないで受信バッファを増して，実時間スレッドの自己安定化（self-stabilization）手法を用いることも考えられるであろう．

5. 関連研究

*i*Reserve が目的とする統合的資源管理機構はケンブリッジ大学の Nemesis[15] が目指すものや，QOS チケットモデル[2] と本質的には同じである．QOS チケットモデルでは，QOS ファクタが我々の QOS パスに対応し，QOS チケットが我々の *i*Reserve に対応する．違いは QOS ファクタは我々の単調増加性の制約はないので，自由な QOS 設定が指定できるが，セッション/資源調停の計算量が増加する．

QOS チケットは *i*Reserve とモデル的には変わらないが，実装されているものが Q-Thread ライブラリと QOS マネージャ[16] である．これはスレッドによるプロセッサ予約以外の管理が実装されていない点が，複数種類の資源予約を考慮した *i*Reserve と異なる．また，協調する複数スレッドでの予約の共有は Q-Thread のようにスレッドを抽象化するのではなく，*i*Reserve のように予約オブジェクトを用いるのが自然である．また，Q-Thread ライブラリでの QOS 翻訳はプロセッサ時間を全体のパーセンテージに変換する自明なものしか扱っていない．

Nemesis ではマイクロカーネルをベースとして共有ライブラリによって構成され，プロセッサ資源管理を持っているが，メモリ資源予約管理機能や QOS 翻訳機能，複数の資源を扱う統合的な機構は持っていない．

CMU の QRAM モデル[8] では QOS ベースの資源割当て手法を導入している．我々の *i*Reserve では資源割当てのポリシーは予約機構の中ではなくそれを調停する QOS マネージャである Conductor にアドオンできるよう分離しており，今後，資源割当て問題についての最適解を得られるアルゴリズムを取り込んでいくことが可能である．

6. おわりに

本論文では，より予測可能性の高い計算機環境を構築するために，統合的な計算機資源管理機構である *i*Reserve 機構とその実装について説明した．*i*Reserve 機構は QOS プロファイリング機能による動的な QOS 翻訳機能によってプラットフォーム独立性を獲得し，統合化のための複数資源の抽象化，複数スレッドによる予約の共有化などの機能を持っている．この *i*Reserve 機構を，我々の連続メディア処理のミドルウェアである Conductor/Performer アーキテクチャに組み込んで，環境情報サーバスイートと協調させる実装を行った．またネットワークアクセス性能の保証の実装および実験も行った．

今後は，*i*Reserve 機構が持つ QOS 翻訳機能と統合化された資源予約オブジェクトによって，分散連続メディアアプリケーションでのより精密な実験を行うために，特に UDP での受信のような非同期的処理に適した予約機構の実装についても行っていく．またそれ以外に IPC やスレッド生成時の資源予約のハンドオフ機構をマイクロカーネルの中に実装する予定である．資源のハンドオフは現在システムコールを通してユーザレベルから行うことは可能であるが，X サーバや Lites サーバのように既成のバイナリのサーバと協調動作するにはカーネル内実装が有効となるであろう．

謝辞 筆者らは慶應義塾大学 MKG プロジェクトのメンバーに，その協力と助言について深く感謝いたします．

参考文献

1) Tokuda, H., Nakajima, T. and Rao, T.: Real-Time Mach: Towards a Predictable Real-Time System, *Proc. USENIX Mach Workshop* (Oct. 1990).

2) Kawachiya, K. and Tokuda, H.: Dynamic QOS Control Based on the QOS-Ticket Model, *Proc. IEEE ICMCS'96*, pp.78–85 (1996).

3) Mercer, C.W., et al.: Processor Capacity Reserves: Operating System Support for Multi-

media Applications, *Proc. 1st Intl. Conf. on Multimedia Computing and Systems*, pp.90–99 (1994).

4) Nishio, N. and Tokuda, H.: Simplified Method for Session Coordination Using Multi-Level QOS Specification and Translation, *5th International Workshop on Quality of Service* (1997).
5) 西尾信彦，徳田英幸：QOSの3-階層指定とその翻訳を用いたセッションの単純化調停方式，情報処理学会論文誌，Vol.39, No.2, pp.328–336 (1998).
6) Nishio, N., et al.: Conductor-Performer: A Middle Ware Architecture for Continuous Media Applications, *Proc. 1st Intl. Workshop on Real-Time Computing Systems and Applications*, pp.122–131 (1994).
7) Lakshman, K., Yavatkar, R. and Finkel, R.: Integrated CPU and Network-I/O QoS Management in an End-system, *5th International Workshop on Quality of Service* (1997).
8) Rajkmar, R., Lee, C., Lehoczky, J. and Siewiorek, D.: A Resource Allocation Model for QoS Management, *IEEE Real-Time Systems Symposium* (Dec. 1997).
9) Jones, M.B., et al.: Support for User-Centric Modular Real-Time Resource Management in the Rialto Operating System, *Proc. 5th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp.55–65 (1995).
10) Nishio, N. and Tokuda, H.: EISS: Environmental Information Server-Suite for System Adaptation, (in Japanese), *Proc. 9th Computer System Symposium* (1997).
11) Keio University, Microkernel the Next Generation Project, WWW Home Page, URL: <http://www.mkg.sfc.keio.ac.jp>
12) Nakajima, T., Kitayama, T. and Tokuda, H.: Experiments with Real-Time Servers in Real-Time Mach, *Proc. 3rd USENIX Mach Symposium* (1993).
13) Maeda, C. and Bershad, B.N.: Protocol Service Decomposition for High-Speed Networking, *Proc. ACM SOSP'93*, pp.244–255 (1993).
14) Nakajima, T.: A Dynamic QOS Control Based on Optimistic Processor Reservation, *Proc. IEEE ICMCS'96*, pp.95–103 (1996).
15) Oparah, D.: Adaptive Resource Management in a Multimedia Operating System, *Proc. 8th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp.91–94 (1998).
16) Kawachiya, K. and Tokuda, H.: Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing, *6th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video* (1996).

(平成10年12月1日受付)
(平成11年 4 月1日採録)

西尾 信彦（正会員）

昭和37年生．平成4年東京大学大学院理学系研究科情報科学専攻博士課程単位取得後退学．平成5年より慶應義塾大学環境情報研究所に勤務．連続メディア処理システムの研究開発に従事．平成8年より慶應義塾大学環境情報学部助手．分散リアルタイムシステム，動的QOS制御に関する研究に従事．平成6年山下記念研究賞受賞．

徳田 英幸（正会員）

昭和27年生．慶應義塾大学より工学修士．カナダ，ウォータルー大学よりPh.D.（Computer Science)．現在，慶應義塾大学常任理事，同環境情報学部教授．SFC研究所長．分散リアルタイムシステム，マルチメディアシステム，通信プロトコル，超並列・超分散システム，モバイルシステム，家電情報システム等の研究に従事．IEEE, ACM，日本ソフトウェア科学学会各会員．