QOSプロファイルハンドラつき資源予約機構の設計と実装

西尾信彦[†] 徳田英幸[††]

より predictable な計算機環境を構築するために，計算機資源の予約は必須であるが，アプリケーションプログラマからは予約すべき資源量の算定が容易ではないうえに，それが通常は動的に変動してしまうために必ずしも利用は容易ではない．本稿では，従来の資源予約オブジェクトごとに QOS プロファイルハンドラを生成することにより，適切な資源量の見積りを動的に実施し，さらに動的な処理内容やシステム状況の変化への適応性を高めることを目標とした統合的資源管理機構の設計と実装について報告する．本実装は，OS 内の低レベル資源予約管理と資源予約調停および翻訳のための資源予約管理サーバ，QOS プロファイルハンドラと動的適応するための非同期イベントメッセージを受信するポートを備えたライブラリにより構成し，GUI インタフェースをサーバに与えることにより，よりユーザに明解で抽象度の高いプログラミング環境を提供することができた．

Design and an Implementation of Resource Reservation with QOS Profiling Handler

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. This article describes the design and implementation of our integrated reservation mechanisms, which consists of in-kernel low-layer resource reservation, resource coordination server and library with a QOS profiling handler and communication port for dynamic QOS control. Besides, GUI interface of the server is developed in order to bring application programmers comprehensive and more abstract programming environment for predictable software.

1. はじめに

近年 CMU の Resource Kernel[1] や Lucent 社の Eclipse/BSD[2]，Cambridge 大学の Nemesis[3] などのように資源予約機構を用いて実行時の厳密な資源利用についての管理を行うことは連続メディア処理をはじめとするリアルタイム処理のためには欠くべからざる要件となってきつつある．しかし，これらの資源予約機構を用いて実用となるアプリケーションを開発するのはまだまだ容易なことではない．たとえばある処理にどれだけの資源が必要なのかは前もって知っていなければ効率良く資源を予約することはできないが，それは処理の内容によってもハードウェアやソフトウェアのプラットフォームによっても異なり，容易には予測できない．たとえ予測が可能であったとしても，一定の負荷を生じ続けるような処理ばかりではなく，行っている処理自体も時々刻々変化するので，それらに動的に適応する機構も必要であろう．そもそも，ある処理に必要な資源量をその絶対的な数量でしか記述できない現行の資源予約方式は理想からは程遠い．

我々はこのように刻々と必要な資源量が変化するプログラムをアプリケーションプログラマが意識する個々の処理の単位に分けて，その処理単位ごとに必要とする資源量を，その処理の QOS に対応するような抽象的な名前で呼べるようにした．たとえば，電話程度の品質で音声を送受信する処理に必要な資源量を “Telephone-level”，CD 品質での場合には “CD-level” というような名前をつけて管理できるようにする．実際に CD 品質の音声通信の処理時には，プロセッサ資源を 24%とネットワークバンド幅を 150 Kbps といった低レベルな表現ではなく，“CD-level” という抽象的な名前での資源予約要求を可能にすることによって，プラットフォーム非依存な資源管理を実現することを

† 慶應義塾大学政策・メディア研究科
Graduate School of Media and Governance, Keio University
†† 慶應義塾大学環境情報学部
Faculty of Environmental Information, Keio University

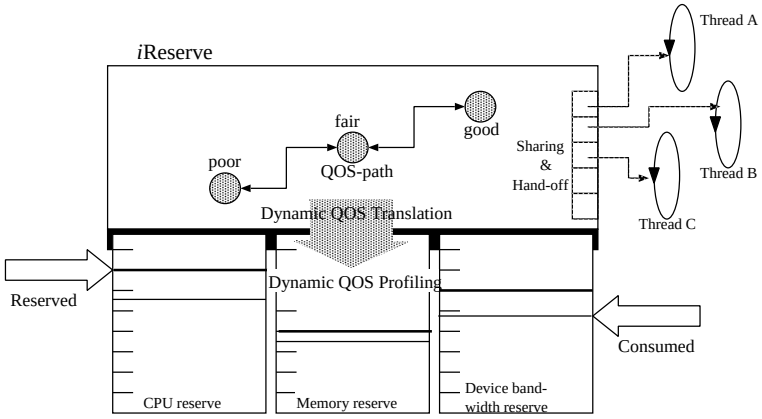

図 1 *i*Reserve オブジェクト
Fig. 1 *i*Reserve object.

表 1 3–層 QOS 表現
Table 1 3-Layer QOS specifications.

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

目指している．

このため，複数種類の資源を一貫して扱えたり，静的に必要な資源量を与えなくとも動的にそれを調べる機構や，このように抽象性を向上することによってプラットフォーム非依存で，そのおかげで資源予約の分散化に適した統合的な資源予約機構として *i*Reserve アーキテクチャを提案し，個々の機能の有効性を検証してきた[4]．この統合的資源予約機構では，ユーザから QOS パスと呼ばれる抽象的な資源予約要求によって**図 1** に示すような，(a) 資源使用に関する動的なプロファイル機構を持ち，(b) 複数種類の資源予約を束ねて，(c) 資源使用をチャージするスレッドを動的に選択できるような資源予約オブジェクトを生成し，予測可能なソフトウェアの開発をより高い抽象度でプログラミングできることを目的としている．これは以前から我々の提案している多階層 QOS 表現[5),6)] を利用した資源予約機構を実現するフェーズにあたる．

この多階層 QOS 表現は “CD-level” のようなユーザよりの抽象的な資源表現と，プロセッサ資源 24%のような資源管理を実現するシステムよりの定量的な資源表現を両立させるために導入されており，我々は**表 1** に示すような多階層表現を提案している．多階層表現では各層での表現の変換が要求され，以前は変換に必要な情報を処理ごとでプラットフォームごとに静的に与えていたが，*i*Reserve は QOS プロファイルにより変換を動的に扱うことを目指している．

本稿では，予約オブジェクトごとに QOS プロファイルハンドラと呼ばれるスレッドを生成し，このハンドラを用いて上記の資源予約管理における動的な資源量の変換や，動的な処理内容の変化への適応機能の設計実装と，この機構をアプリケーションプログラマが利用するためのプログラミングインタフェースの設計と実装について報告する．ここでは，資源予約をする各クライアント間の調停アルゴリズムやプロファイル機構，資源使用のチャージの仕方を汎用的に取り換え可能にし，入出力一体の GUI を持った資源モニタを設計実装している．

本稿の関連研究としては，前述の Eclipse/BSD，Resource Kernel，Nemesis などがあるが，これらでは多階層での QOS の表現についての言及はあるものの，それらの層の間での具体的な変換方式や実装をともなっていない．また，本稿では，主にエンドシステムにおける資源管理に注目しているが，ネットワークシステムにおける資源予約管理や資源要求の多階層化はコロンビア大学の Comet プロジェクト[7)] やイリノイ大学の QOS Broker モデル[8)] をはじめとして多くある．しかし，ネットワーク資源は比較的資源量の算出が容易であり，エンドシステムにおけるプロセッサ資源の判定に直接それを適用することはできない．

以下では，次章で QOS プロファイルハンドラつきの資源予約機構について，その動作機構とプログラミングインタフェースについて述べる．続いて 3 章では

Real-Time Mach 上での実装と性能評価実験について述べ，最後に結論と今後の課題についてまとめる．

2. 統合的資源予約機構

我々は *i*Reserve と呼ばれる統合的な資源予約のアーキテクチャを設計し，その機構を開発している．ここでは，資源予約オブジェクトを生成するときにそのメンテナンスをするためのスレッドを QOS プロファイルハンドラとして生成して，そのハンドラにより資源予約管理における各種の動的な適応を行う．本章ではまずこの統合的資源予約機能の概要について説明し，QOS プロファイル機構および動的適応機構について述べ，その後プログラミングインタフェースの詳細について述べる．

2.1 本機構の前提条件

本機構の設計において仮定しているのは，マルチメディア処理において一定レートの処理を行うときに見られるような秒オーダ以上の比較的長いスパンでほぼ一定の資源量要求が発生するような場合をサポートすることである．たとえば，一定レートでのディスクなど周辺機器へのアクセスやネットワーク転送にともなうプロトコルスタック処理，マルチメディアデータの圧縮伸張などの codec 処理などが考えられる．このような処理に対してプロファイル処理により適切な資源量を予約することを QOS プロファイル機構によって実現し，処理内容が動的に変化した場合や自分以外の処理主体が優先して予約していた資源が減少した場合なども動的適応機構によって可能な限り適応することをシステムの設計の目的とした．本稿では以上のような仮定のもとで，プロセッサ資源とネットワーク転送性能に要する資源予約に関しての統合的管理機構の設計実装について報告する．

2.2 本機構の利用手順

従来の資源予約機構では，ユーザは自分の行う処理に必要な資源の量をあらかじめ見積もり，それを資源要求の中でじかに記述する必要があった．また，処理の内容が変更された場合には，いままで予約している資源では過不足が生じる場合があるので，処理内容の変更が生じるごとに予約した資源の解放や追加予約を陽に記述しなければならない．

我々の統合的資源予約機構では，統合的資源予約を要求するクライアントプログラムがリンクするライブラリと資源予約管理サーバによってシステムを構成している．クライアントの要求により生成された統合的資源予約オブジェクト（以後 *i*Reserve オブジェクトと呼ぶ）は生成時には処理主体とは独立であり，生成後に処理主体の単位であるスレッドとの結び付きを動的に発生させる☆．

ユーザの記述するクライアントプログラムでは，本ライブラリを用いてこの *i*Reserve オブジェクトを生成して資源予約要求するときに資源使用量のプロファイルを行うハンドラが生成される．このハンドラがユーザからのプロファイル開始の要求をトリガに，それ以降の処理による資源消費量をプロファイルする．そのプロファイルの結果，必要量の資源予約要求を生成し，あらためて予約要求を提出する．つまりユーザからの資源要求が発行されて，プロファイル処理を完了した後で実際の資源予約要求がなされる．

ユーザは資源予約要求に低レベルな資源量を記述しない代わりに，その処理が必要とする資源量に対して抽象的な名前をつけ，今後同じ処理が行われる場合にはそのときにつけた名前で資源予約要求を行う．このとき用いる名前の例としては前述の “CD- level” とか “Telephone-level” などが考えられる．本稿ではこのような名前を「処理レベル名」と呼ぶ．処理レベル名には，複数種類の資源が必要となる場合もある．たとえば，ネットワークアクセスやストレージアクセスでは，I/O 機器とのバンド幅という資源以外にそのバンド幅を消費できるだけのプロセッサ資源も必要とするし，その性能のためのバッファリングとしてのメモリ資源を必要とする場合もある．処理内容が異なる場合には，異なる処理レベル名をつけるのはユーザの責任であり，どの処理レベル名にどれだけの資源が必要かは機構側で管理される．これにより，ユーザは本来ハードウェアプラットフォーム依存であるはずの資源量の記述から解放される．

2.3 QOS プロファイル処理

QOS プロファイル処理は，それにかかる時間が無視できないことと，ホスト内ではつねに複数の処理が並行して資源予約保証を期待して実行されていることから 1 通りの手法ですべての場合には適切に対処できない．そこで我々はプロファイルの実行ポリシーを協調的ポリシーと優先的ポリシーの 2 種類に分けて取り扱うこととした．協調的ポリシーのプロファイルは，あらたに投入されプロファイルされる処理主体がすでに稼働している他の処理主体にできるだけ影響を与えないもので，優先的ポリシーではあらたに投入する処理主体をプロファイル時から極力優先的に扱うものである．

☆ 複数のスレッドを単一の予約オブジェクトと結び付けることは許されるが，逆は不可である．

協調的プロファイル 協調的プロファイルでは，プロファイルしている間は資源予約をせず走行させその間の資源消費を測るもので，実際の資源予約はプロファイル後に本機構によって発行される資源予約要求が成功したときに初めて保証される．このため，実時間性を求められデッドラインなどが設定されているものに関してはプロファイル時の実行でデッドラインミスを起こす可能性がある．ここで，資源利用状況のプロファイル時に，資源予約をしないのは，プロファイル処理時の動作のために他の動作をすでに保証されているプログラムの実行に極力影響を与えないためである．この予約しないプロファイル期間に時間制約が守れない場合にも，資源の消費量を測ることは可能なのでプロファイル自体への影響は少ないと考えられる．

優先的プロファイル 優先的プロファイルでは，協調的プロファイルとは対照的にプロファイル時からその処理が必要とする以上の資源を予約して走行させ，プロファイルによってその過剰部分を適切な予約量に減らしていくものである．この場合，実際に必要とする資源量はそもそも不明なのでプロファイルをするために，それ以上の予約を行うことは単純にはできない．可能な手法としてはプロファイル時に資源管理機構に強制的に要求してそのときに利用可能な資源をすべて予約してしまうことが考えられる．しかし，これでも資源が足りない場合はありうるので，その対策としてはあらたに投入される処理主体がどの程度優先的なのかの優先度などの指標が必要となる．すなわち，その優先度以下の処理主体が予約している資源を解放させて割り当てる手法である．

本稿で想定するマルチメディア処理のようなソフトリアルタイムシステムにおいては，基本的に協調的ポリシーのプロファイルが一般的であり，どうしても動作保証をしたいタスクの投入時のみ優先的ポリシーを適用するのが適当であろう．以後，本稿では基本的に協調的プロファイルを行うものとする．

プロファイル処理のアルゴリズムに関しては，各種資源ごとに採用するものであり，また管理されるプログラムの性格によっても変えた方がよいことがあるので固定化はしない．プロセッサ資源とネットワーク転送性能については，次章で詳述するが，一般的には，以下の 2 種類に大別される．

まず，プロセッサ資源の場合のような「被支配的資源」[4] の場合☆は実際に計測した結果何らかの安定状態を定義することによって判断できるであろう．

ネットワーク資源のバンド幅などのような「支配的資源」は処理内容から実行前に動的なプロファイルを必要とせず，静的に算出することも可能である．ただし，静的に算出されたバンド幅などのネットワーク転送性能を実現するためには，被支配的資源であるプロセッサ資源の必要量の動的なプロファイルが必要で，そのプロファイル結果からネットワーク転送性能との関係を最小自乗法などを用いて推定し，その結果から予測値を算出することになろう．

一方，資源管理されるプログラムの性格に着目すると，いわゆるサーバ・クライアント方式におけるサーバとクライアントで異なってくる．サーバは基本的にクライアントの仕事を肩代りして遂行するので，資源を消費しているのはサーバというよりもむしろクライアントの方である．よってこの場合の資源消費のチャージはクライアントに課すことになるが，そのときの資源消費のプロファイルは，やはりそのサーバの何らかの性能からの関係式を用いるべきであろう．たとえばネットワークへの転送を担ったプロトコルスタックサーバ[9] などの場合には，このサーバ全体の消費量をクライアントごとにパケット荷重分配してチャージを分けることなどが考えられる．

本機構の設計では，このようにプロファイルを実際に行う部分が管理されるプログラムにリンクしていた方が都合がよいので，QOS プロファイルハンドラはライブラリとして，資源予約管理サーバとは分離したものとする．これにより様々な種類のプロファイルアルゴリズムや資源使用のチャージ方式を使い分けることが可能になる．

2.4 動的適応機構

前述のように，実質的な資源予約要求はプロファイル完了後に資源管理機構によって承認されて初めて資源予約が成功する．よって，プロファイル後に資源予約要求が承認されない場合もあり，クライアントプログラムにはそのための非同期な処理が必要になる．このように，

- 資源予約要求が失敗する場合以外にも，
- 動的に処理の内容が変化しているにもかかわらずユーザがそれに気づかず資源予約要求を適切に出していない場合や，
- 自分以上に優先的な処理主体が投入されそちらに

☆ ネットワーク転送性能などの他の達成目標に依存して，それから間接的に要求される資源のこと．

資源が割り当てられたために自分の資源が減ってしまう場合や，
- その逆に資源が解放され自分の処理レベルを向上できるような機会，

といった 4 種類のケースにも動的に対応できるとよい．

以上のような非同期処理は本機構での動的適応機構として，ユーザがそのような状況に適応するためのハンドラを記述するか，そのようなハンドラが提供されない場合にはデフォルト動作として QOS プロファイルハンドラが対処する．ただし，QOS プロファイルハンドラができることはすでに登録された別の処理レベルへの予約変更要求を出すことだけで，このような場合にはアプリケーションの作成者が非同期処理のためのハンドラを記述をするのが最適である．このように，システム全体の調停と資源の再配分は機構側が担当し，実際に割り当てられた資源量に合わせた処理内容への移行はアプリケーションが担当するわけである．

2.5 プログラミングインタフェース

本節では，設計した QOS プロファイルハンドラつき資源予約のためのプログラミングインタフェースについて述べる．前節でも述べたように本機構はクライアントプログラムにリンクするライブラリ部分と，資源予約管理サーバ部分から構成される．

2.5.1 ライブラリインタフェース

直接ユーザが利用するライブラリインタフェースの概要は以下のとおりである：

`irsv_init` 初期化ルーチン．

`irsv_create` *i*Reserve オブジェクトの生成．

`irsv_reserve_thread` プロセッサ資源の予約．

`irsv_reserve_network` ネットワーク転送性能の予約．

`irsv_sendto` 本機構での UDP パケット送信

具体的には，以下の擬似コードで示したプログラム例のように利用する．

```
qos_level_t qos;
main()
{
    mach_port_t async_port;
    irsv_init();
    irsv_create(..., &async_port);
    create_thread(async_handler);
    ...
    qos = GOOD_QOS;
    irsv_reserve_thread(..., GOOD_QOS, TRUE);
    while (...) {
        process_in(qos);
    }
}
async_handler()
{
  message_t msg;
  for (;1;) {
      receive_message_from(async_port, &msg, ...);
      switch(msg) {
      case QOS_UPGRADE:
        qos = BETTER_QOS;
        irsv_reserve_thread(..., BETTER_QOS, TRUE);
        break;
      case QOS_DOWNGRADE:
        qos = POOR_QOS;
        irsv_reserve_thread(..., POOR_QOS, TRUE);
        break;
      case OVER_RUN:
        qos = NEW_QOS;
        irsv_reserve_thread(..., NEW_QOS, TRUE);
        break;
      ...
        }
    }
}
```

この例では処理レベル名 “GOOD_QOS” と，それよりも負荷の低い処理レベルである “POOR_QOS”，負荷の高い処理レベル “BETTER_QOS” の 3 つのレベルを用意したプログラムをプロセッサ資源を予約して動作させるものである．プログラムは main 本体と非同期ハンドラとに分かれて記述されており，メインの処理は `process_in()` で繰り返し実行される．初期化後，“GOOD_QOS” で資源予約要求するとプロファイルが `process_in(GOOD_QOS)` の繰返し処理に対して行われ，プロファイル後，必要な資源が予約され，その資源量情報と処理レベル名 “GOOD_QOS” とのペアが機構側に保持される．

その後，前述のような要因で動的な状況変化があるとサーバから非同期メッセージが登録した `async_port` に送られる．非同期ハンドラは到着したイベントに合わせて適切な資源予約要求を発行する．`OVER_RUN` メッセージはクライアント側から処理レベルの変更が知らされぬまま，処理の負荷が定常的に変化したことを機構側が認識した場合に送られるイベントである．

また，資源予約要求の最後の引数の `TRUE` はプロファイル開始を意味し，複数のスレッドを *i*Reserve オブジェクトにバインドするときは `FALSE` をつけ，最後のプロファイル開始直前の要求にのみ非零の値を与える．このときの非零の値により，プロファイル処理のアルゴリズムを選択することもできる．資源予約要求のライブラリコールは，引数の処理レベル名によって，プロファイル処理要求なのか，すでに登録された処理レベルへの遷移要求なのかを知的に判断している．

2.5.2 サーバインタフェース

続いてサーバインタフェースについて説明する．概要は以下のとおりである：

`irsvmgr_register` *i*Reserve オブジェクトの生成.
`irsvmgr_delete` *i*Reserve オブジェクトの廃棄.
`irsvmgr_map_info` 処理レベル名と必要資源量のペアの情報を登録.
`irsvmgr_QOS_level` 処理レベル名に遷移要求.
`irsvmgr_reserve_thread` 処理主体とプロセッサ資源予約の対応を登録.
`irsvmgr_reserve_network` 処理主体とネットワーク性能予約の対応を登録.

本サーバインタフェースはプロファイル処理を行う部分をライブラリとして分離した残りの部分であり，ライブラリ以外からは直接呼び出されることはない．`irsvmgr_map_info` は翻訳情報の登録だけで，実際のシステムレベルでの資源予約要求は `irsvmgr_QOS_level` が発行されたときに発生する．これによって遷移先となる処理レベルが必要とする物理資源の予約要求が生成され，このとき資源分配のために調停アルゴリズムが起動される．調停アルゴリズムの選択はこのときに呼ばれるルーチンを動的に変更することによって実現される．

3. *i*Reserve 機能の実装と性能評価実験

3.1 実装環境

本統合的資源予約機構 *i*Reserve は Real-Time Mach [14] 上で実装し，プロセッサ資源と UDP によるネットワーク送信性能を予約可能にしている．ハードウェアプラットフォームとしては，Celeron 300A MHz プロセッサ，128 MB のメモリ，Intel 100/10 FastExpress Ethernet の 100 Base/TX ネットワークを備えた組立て PC 互換機 2 台をスイッチで接続している．実装は Real-Time Mach マイクロカーネル内の低レベル資源予約管理機構[15),16)] を土台として，前述のとおり資源予約の翻訳およびセッション間の調停を行うサーバ，さらにアプリケーションにリンクして QOS プロファイルハンドラを走行させるライブラリから構成し実装した．低レベルの資源予約管理機構ではプロセッサ資源のみを管理しパケットスケジューリングなどはしていない．ネットワーク送信性能は TCP/IP プロトコルスタックをユーザレベルのライブラリにより実装して[13)] アプリケーションにリンクしている．ネットワーク送信性能を出すにはネットワークのバンド幅を予約するとともに，TCP/IP プロトコル処理に必要なプロセッサ資源を予約しなければならない．本実装ではプロセッサ資源量を制御することによりネットワーク性能を制御し予約保証することを可能にしている．

3.2 QOS プロファイル処理の実装

アプリケーションプログラマは，これから行う処理を担うスレッドとその処理レベル名を与えるだけでよく，QOS プラファイルハンドラがそのスレッドの挙動をプロファイルする．本実装では，協調的プロファイル方式を用いており，プロセッサ資源消費のプロファイルのアルゴリズムとしては，周期スレッドの場合はその周期ごとに，非周期スレッドの場合にはバインドするプロセッサ資源予約オブジェクトの回復周期ごとに消費量をサンプルし，それからプロセッサ資源の予約量を算出する方式を採用している．具体的にはサンプルしたプロセッサ資源の消費量の平均 m と標準偏差 s を求め，$2s$ を変動時の余裕として $m+2s$ を予約要求している．これは，サンプルした値が正規分布の場合には，m と s の値を全サンプルから算出した場合にほぼ 96%以上をカバーできることが期待できるためである．

ネットワーク送信性能の予約機構の実装では，UDP プロトコルを利用し，前述のとおりネットワークバンド幅という資源とは別にプロトコル処理のために必要となるプロセッサ資源量をプロファイルにより算出して予約要求している．このネットワークプロトコル処理を担う資源予約のオーバユース時のポリシーはサスペンドとすることによって要求性能の保証と遵守を維持するようにした．

本実装でのネットワーク性能の QOS プロファイル処理のアルゴリズムを以下に説明する．まずプロセッサ資源の初期値は同様に 1%予約として 1 回復周期送信を行い，その性能サンプルから目標性能への予測を行う．プロセッサ資源の消費量に合わせて送信性能は単調に増加するので，毎回復周期ごとに資源予約量を倍にして目標値を超えた時点で 2 分探索法で収束させるといった理論的な手法も一般には考えられる．しかし我々はいままでにプロセッサ消費量とネットワーク送信性能がほぼ一次近似できることを示している[4)] ので，ほぼ固定サンプル数のプロファイルを最初に行うだけで今後の資源消費量予測に適用可能な関係式を導けることが分かる．我々は目標値に対して低性能地点（プロセッサ資源予約 1%）と高性能地点（目標値の倍の性能地点）でサンプルしてそこから関係式を導出した．高性能地点でのサンプルは，必要なプロセッサ資源量を低性能地点でのサンプルから推測してそれだけの資源予約を実際に行うことにより実現する．目標値の倍にしたのはプロファイル期間も平均すれば目標性

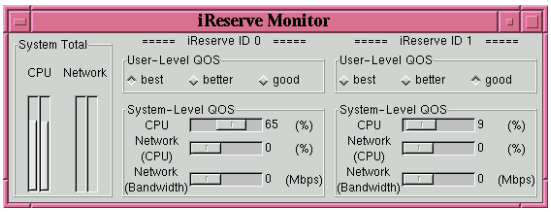

図 2 *i*Reserve GUI パネル
Fig. 2 *i*Reserve GUI panel.

能に近づくようにしたためである☆．この 2 地点での予約量と送信性能より最小自乗法などにより関係式を導出して，最後に目標地点に移す．今後は，この関係式により最初から目標地点に性能を移動することが可能になる．

以上のようなプロファイルの結果，指定されたスレッドの処理レベルに必要なシステムレベルでの資源量と処理レベル名のペアが資源予約管理サーバに登録され，以後その処理レベルについてのシステムレベルでの知識として活用される．サーバはアプリケーションからの処理レベル名による予約要求をそのプラットフォームに較正された厳密な資源量に翻訳する役目と，各資源予約オブジェクト間での資源分配の調停の役目を担っている．このセッション間での資源分配の調停に関しては，その調停ポリシーを分離して実装，選択可能としており，それにより様々な利用シチュエーションに対応する．本実装における資源分配のポリシーとしては，予約資源の合計が全体の 90%を超えない限り，要求された資源を分配し，資源が足りなくなった場合には最初に割り当てたクライアントの処理レベルを下げるなどして資源を解放するという単純な方式を実装している．

3.3 GUI インタフェースの実装

各種資源の利用/予約状況と多階層での QOS 表現を可視化するために本実装では，資源予約管理サーバ部分に GUI インタフェースを追加した．**図 2** にそのスナップショットを示す．この GUI は，左側にシステム全体の状況，右側に資源予約オブジェクトごとの利用/予約状況を描画している．各 2 本のゲージは現在予約されている量と消費されている量を示している．資源予約オブジェクトごとの表示では，上部と下部に表示が分かれており，上部がアプリケーションプログラマが動的に与えた処理レベル名による表示（ユーザレベル QOS 指定表現）になっており，下部がシステムレベルでの資源（システムレベル QOS 指定表現）を表示している．ここでは，プロセッサ資源とネットワーク送信性能の利用/予約状況を描画している．上部のユーザレベル QOS 指定表現である処理レベル名は，それぞれが GTK+ [17] により実装されたラジオボタンになっており，そのときに選択されている処理レベルが押されている．これらラジオボタンはユーザが処理レベル名をつけてプロファイルが完了したときに動的に生成され出現する（GUI パネル上にビューが与えられる）．これらラジオボタンは入出力一体型のインタフェースを持っており，ユーザは本 GUI インタフェースを通じることによっても処理レベルを動的に変更することが可能であり，かつシステム側は各クライアントの動的な資源の再分配を行った場合にはそれが押されているラジオボタンを変化させることにより表現する．

☆ ただし，低性能地点でのサンプルによる予測から高性能地点がプロセッサ資源の 50%を超えるときは高性能地点ではなく目標地点でのサンプルを行う．

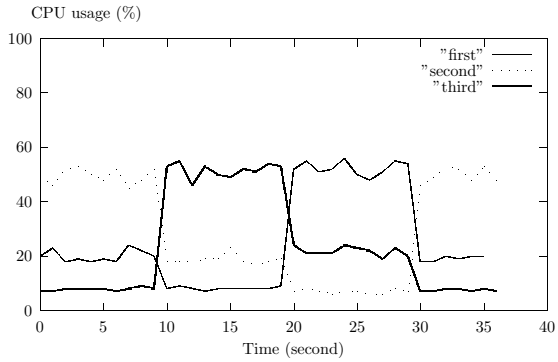

図 3 マルチセッション資源予約制御実験（CPU 酷使タスク）
Fig. 3 Multi-session reservation control experiment (CPU intensive task).

3.4 性能評価実験

実装した *i*Reserve 機構を用いてまず複数のスレッドに複数種類の処理レベルを与えて，動的に制御した実験を**図 3** のグラフに示した．ここでは，Fibonacci 数の計算というプロセッサを酷使する処理を何度も繰り返すスレッドを 3 つ同時に動かした．各スレッドとも処理内容は同じであるが，初期位相をずらしてある．“first” スレッドはまず中程度の負荷の処理（Fibonacci(25) を周期 100 msec で繰り返す，処理レベル名は “better”）を 10 秒間繰り返してから最も軽い負荷の処理（同じく Fibonacci(23) で “good”）を 10 秒間繰り返す．その後，最も重い処理（同じく Fibonacci(27) で “best”）を 10 秒間繰り返してから，また中程度の負荷の処理に戻るといった 3 段階の負荷を 10 秒間隔で行き来するという 30 秒周期の処理を行っている．“second” スレッドは “first” スレッドと

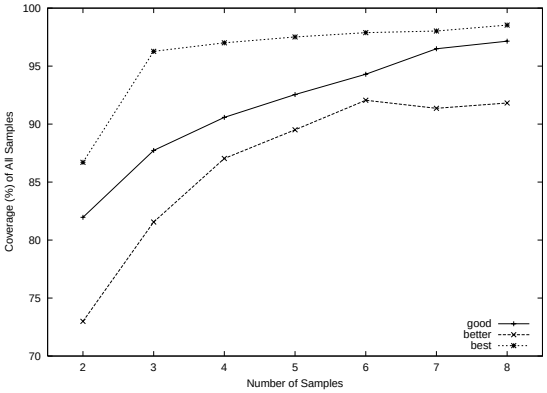

図 4 QOS プロファイル性能実験（プロファイル期間）
Fig. 4 QOS profile performance experiment (duration).

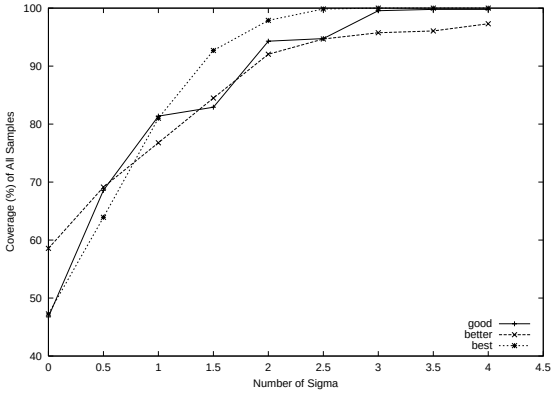

図 5 QOS プロファイル性能実験（予約量の余裕）
Fig. 5 QOS profile performance experiment (delta).

同じ処理を重い負荷の処理から始め，“third” スレッドは軽い負荷の処理から始めるといったように，初期位相のみをずらしてある．

図 3 には，それぞれで実際に消費されたプロセッサ資源（システムレベル QOS 指定表現）がグラフの縦軸として記されている．それぞれの処理レベルの最初の周期の頭でプロファイルが実行されている．これにより 3 種類の処理レベルがプロファイルされて，それぞれに必要なシステム資源が動的に確保され処理が遂行されているのが示されている．

本機構における QOS プロファイル処理は，そのプロファイルの期間が長いほど，つまりサンプル数 N が大きいほど，正確なプロファイル結果に近づく傾向にあるが，それだけ実際の資源予約要求が遅延してしまう．そこで，どれだけ長くプロファイル処理をすればどの程度正確なプロファイル結果を出せるかについて以下のように評価した．上記の CPU 酷使タスクにおいて，プロセッサ資源のプロファイルはカーネル内のプロセッサ予約機構における回復周期の分解能で採取できる．そこで何回復周期のプロファイルを連続してサンプルすることによりどれだけ正確なプロファイル結果となるかを測定した．**図 4** は，横軸がプロファイルを何サンプルから行うかを，縦軸をそのプロファイル結果で予約した範囲に全サンプルの何パーセントがおさまったかを示した．ここで回復周期はスレッドの周期である 100 msec なので，実際のプロファイル期間は 100 msec のサンプル数倍である．結果は，3 種類の負荷である “best,” “better,” “best” のそれぞれについて示した．

本実験の結果によると 6 サンプルつまり 600 msec のプロファイル期間で，3 種類のどの負荷を与える処理についてもその 90%以上をカバーする予約量を算出できていることが分かる．また，この評価では予約量に平均値から $2s$ の余裕を与えたが，その増減により全サンプルのカバーできる範囲についての評価を行った結果が**図 5** である．ここでは，特に 6 サンプル（600 msec）にプロファイル期間を固定して，標準偏差 s の倍数を 0 から 0.5 刻みで 4 まで変化させ，横軸を標準偏差 s の倍数に，縦軸に全サンプルのカバー率を表した．ここでも標準偏差の 2 倍の余裕を平均値に足せば 3 種類の負荷の処理とも 90%以上がカバーされることが分かる．

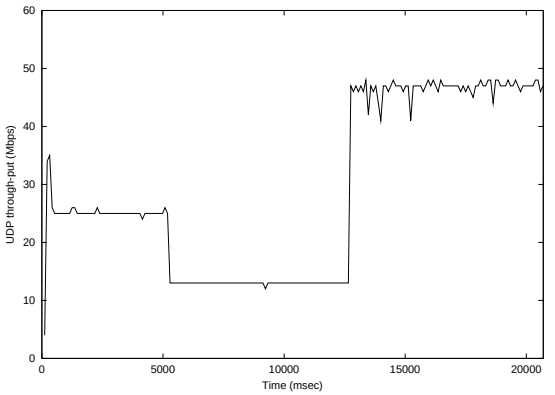

図 6 ネットワーク転送性能制御実験（UDP）
Fig. 6 Dynamic network-access performance control experiment (UDP).

続いて，本実装のネットワーク送信性能保証の性能評価実験を行った．ここでは，目標値となる UDP パケット送信量を動的に 24 Mbps，12 Mbps，48 Mbps の順に変化させ，それに追随した性能を示せるかを調べた．**図 6** に，横軸を時間で縦軸を達成したスループットとして結果を示した．最初の性能である 24 Mbps を達成するまでの間がちょうど前述の最小自乗法を用い

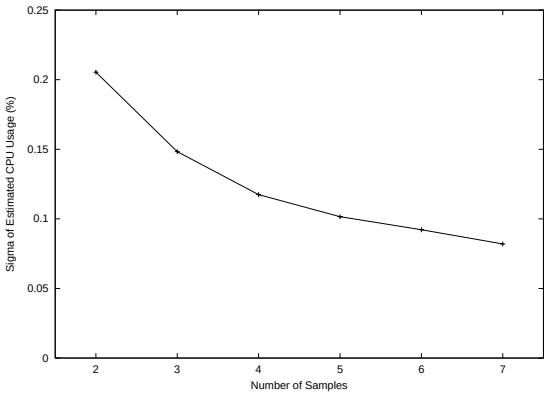

図 7 ネットワーク性能のプロファイル処理の精度
Fig. 7 Dynamic profiling performance for network access.

た較正フェーズである．本実装がベースとする低レベルでのプロセッサ資源予約の分解能が 1%単位であり，本プラットフォームでは 1%のプロセッサ資源予約の差で実際には数 Mbps の転送性能の差がついてしまう．そこで，実装としては要求性能以上を出せる資源量を切り上げで予約し，クライアントアプリケーションでのデータ転送処理でタイマや実時間スレッドなどを利用して調整する方式とした☆．

このときに獲得した関係式により，それ以降の性能変化には迅速に対応できているのが分かる．ただ 24 Mbps あたりまではほぼ安定した性能を出せているが 48 Mbps あたりでは，これは以前から観測されている傾向[4] だが，やや安定性を若干欠いている．精度の高い関係式を導出するには 40 Mbps を超える高性能地点ではサンプル数をある程度増さなければならないかもしれない．そこで 48 Mbps 地点で N サンプルとったときに，24 Mbps を達成するのに必要なプロセッサ資源量を予測した場合の予測値の標準偏差を調べてみたのが**図 7** である．横軸がサンプル数で，縦軸が予測した予約資源量の標準偏差である．ここで分かるように，2 サンプルでも標準偏差は 0.2%程度におさまっているので，低性能地点，高性能地点，目標性能地点の 3 地点で各 2 サンプルしたとしても 600 msec 程度で予測できることが分かる．

4. おわりに

我々は予測可能性の高いソフトウェアをより容易に開発するために，資源予約機構を統合化し，可能な限り動的な適応性を持たせることを目標とし *i*Reserve 機構の設計と実装を行った．本来，指定された処理がどれだけの資源量を必要とするかは確実に予測することは困難であるうえに，その処理の負荷自体も動的に変化してしまう．これらに適応するために，専用の QOS プロファイルハンドラを設け，それを通じて適切な量のシステム資源を要求したり，適切な処理への移行をアプリケーションへ通知したりすることが可能になった．また，アプリケーションプログラマは，より抽象的なレベルでの予測可能なソフトウェアの開発を促進することができるようになった．さらに，GUI インタフェースをサーバに持たせることによりシステム全体や予約オブジェクト単位での状況把握を明確することができ，ユーザレベルでの QOS 表現を用いた制御を可能にした．

今後は，本稿ではクライアントのように自分に課すべき処理のみのタイプのアプリケーションへの対応を想定しているが，今後は複数のクライアントの処理を並列処理するサーバタイプのソフトウェアへの適用も，資源利用のプロファイリングの方式を選択可能にすることにより視野に入れていきたい．また実装面では，現在のプロセッサ資源予約の分解能が 1%であり，今後プロセッサの性能の向上を考えてこの分解能をさらに向上する必要がある．また現在の QOS プロファイルによる資源消費量の測定はシステムコール経由であるがそれを共有メモリを用いた実装とすることで，より効率的なプロファイルを可能としていきたい．

謝辞 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．なおこの研究は，情報処理振興事業協会（IPA）が実施している次世代アプリケーション開発事業のもとに行われた．

☆ 本実験ではクライアントは特に調整しないで best effort で転送しているので，予約した資源一杯を利用しているために要求性能をやや超えた箇所がある．

参考文献

1) Rajkumar, R., et al.: Resource Kernels: A Resource-Centric Approach to Real-time and Multimedia Operating Systems, *Proc. SPIE Conf. on Multimedia Computing and Networking* (Jan. 1998).
2) Bruno, J., et al.: Retrofitting Quality of Service into a Time Sharing Operating System, *USENIX Annual Technical Conferance* (June 1999).
3) Reed, D. and Fairbairns, R. (Eds): The Nemesis project, Nemesis, the kernel: Overview (May 1997).
4) 西尾信彦，徳田英幸：*i*Reserve アーキテクチャ：統合的資源予約機構，情報処理学会論文誌，Vol.40, No.6, pp.2645–2658 (1999).

5) 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).
6) 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).
7) Comet Project WWW page. `www.comet.columbia.edu`.
8) Nahrstedt, K. and Smith, J.M.: The QoS Broker, *IEEE Multimedia*, Spring 1995, Vol.2, No.1, pp.53–67 (1995).
9) Nakajima, T., Kitayama, T. and Tokuda, H.: Experiments with Real-Time Servers in Real-Time Mach, *Proc. 3rd USENIX Mach Symposium* (1993).
10) Kawachiya, K., et al.: Evaluation of QOS-Control Servers on Real-Time Mach, *Proc. 5th Int. Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp.123–126 (1995).
11) Rajkmar, R., Lee, C., Lehoczky, J. and Siewiorek, D.: A Resource Allocation Model for QoS Management, *IEEE Real-Time Systems Symposium* (Dec. 1997).
12) 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).
13) Maeda, C. and Bershad, B.N.: Protocol Service Decomposition for High-Speed Networking, *Proc. ACM SOSP '93*, pp.244–255 (1993).
14) Tokuda, H., Nakajima, T. and Rao, T.: Real-Time Mach: Towards a Predictable Real-Time System, *Proc. USENIX Mach Workshop* (Oct. 1990).
15) Mercer, C.W., et al.: Processor Capacity Reserves: Operating System Support for Multimedia Applications, *Proc. 1st Intl. Conf. on Multimedia Computing and Systems*, pp.90–99 (1994).
16) Nakajima, T.: A Dynamic QOS Control Based on Optimistic Processor Reservation, *Proc. IEEE ICMCS '96*, pp.95–103 (1996).
17) GTK+ web page: `http://www.gtk.org/`.

(平成 12 年 12 月 18 日受付)
(平成 13 年 4 月 6 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所定単位取得後退学．平成 5 年より慶應義塾大学に勤務．同大学より博士（政策・メディア），現在，政策・メディア研究科助教授．連続メディア処理システムの研究開発，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．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．