修 士 論 文

システムの実行状態の視覚化に関する研究

指導教官　中島達夫 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

中野真紀子

1997 年 2 月 14 日

要旨

近年，コンピュータグラフィックスの技術が向上し様々な分野で情報との対話手段としてグラフィックスが利用されるようになってきた．科学技術データの表示はもちろん，情報検索，ソフトウエア開発などである．このような情報視覚化（Information Visualization）の分野のなかにおいてシステムの実行状態の視覚化のことをパフォーマンス・ビジュアライゼーション（Performance Visualization ）と呼ぶ．

これまでのパフォーマンス・ビジュアライゼーションは主に1計算機，あるいは並列計算機における多重プロセスの解析に利用されてきたが，1計算機のシステム全体の様子を視覚化したツールは少なく，また現存するツールでは機能が限られている．

本研究では，1計算機のシステムの実行状態に関する視覚化を実時間で行なう．1計算機の実行状況の視覚化を実時間で行なうことで，アプリケーションを最適化する上での指標を示す．

目 次

1 はじめに **1**

2 背景 **4**

2.1 情報視覚化 4

2.2 システムの実行状態の視覚化 5

2.2.1 関連研究 6

2.3 Mach の基本概念 11

2.4 まとめ 13

3 設計方針 **15**

3.1 全体構造 15

3.2 インタフェース 18

4 実装 **19**

4.1 全体構成 19

4.2 Event Tap, Reporter 21

4.2.1 データ構造 22

4.3 CounterControlServer 22

4.4 PerformanceTool 25

4.4.1 データ構造 25

4.4.2 データの解析 29

4.5 ユーザーインタフェース 35

4.5.1 メイン ウィンドウ 35

4.5.2 カウンタ コントロール ウィンドウ 36

4.5.3　モニタ ウィンドウ 39
454　タスク ウィンドウ 40
455　スレッド ウィンドウ 41

5　評価 **42**

6　結論と今後の課題 **45**

第 1 章

はじめに

従来，視覚化（visualization ）と言うと，スーパーコンピュータで処理された膨大な量のデータをコンピュータグラフィックスを用いて表示するサイエンティフィック・ビジュアライゼーション（scientific visualization ）が代表的な例であった．そしてこのサイエンティフィック・ビジュアライゼーションは主に流体工学，高分子化学といった分野において，複雑な数値データをわかりやすく表示する事や，見えないものや見づらいものを可視化する事などが目的であった．[15]

これに対し，最近では低価格で高性能なマシンが個人に普及したこともあり，単なるデータの表示に留まらず，X ウィンドウや Windows などに代表されるウィンドウシステムや，Netscape や Mosaic などの WWW のブラウザ，情報検索，ソフトウエア開発といった様々な分野における情報との対話手段としてグラフィックスが利用されるようになってきた．

昨今のコンピュータのアプリケーションを見ると，すべてにといってよいほど，グラフィカルユーザインタフェース（GUI）が提供されている．このようなグラフィックスの利用を scientific visualization に対して information visualization （情報視覚化）と呼ぶ．[13]

この情報視覚化の目的は，情報を抽象化を行なったり，図の特徴を利用することによって情報に対する理解をより早く深くすることである．事実，このようなビジュアル（またはグラフィカル）インタフェースの誕生をきっかけに，これまでコンピュータとは縁のなかった一般の人々にとってもコンピュータが身近な存在になった．タイプライターなどが

従来から普及してた欧米諸国にくらべ，キーボードに馴染みの薄い日本人にとっては，ビジュアルインタフェースの存在がコンピュータの普及に大きく貢献したといっても過言ではないだろう．

近年のコンピュータ技術の急激な発達と共に膨大かつ多種のデータを得られるようになったきたが，コンピュータは人間の知的活動を支援するための道具であり，それを使う人間とのかかわりの上で評価されるべきである．したがっていくら高性能，高機能なコンピュータがあり，そこから正確なデータが与えられたとしてもユーザが理解できなければ意味を成さないといえる．よって．人間とコンピュータとの間に優れたインタフェースを構築することはコンピュータを利用する上できわめて重要だと言える．

情報の視覚化には様々な分野があるが，本稿ではシステムの性能を解析するために実行状態に関する情報を視覚化するパフォーマンス・ビジュアライゼーション（Performance Visualization）についての考察を行なう．

解析の対象とするのは，本研究室で研究が行なわれているリアルタイムマイクロカーネル Real-Time Mach [2] [11] [1] [9] 上におけるアプリケーションの実行状態である．
リアルタイム処理を扱うシステムでは，システムの実行状態を把握することは非常に困難であるが，システムの実行状態を視覚化することで，状態を把握することができる．ARM（Advanced Real-Time Monitor）は，Real-Time Mach 上のシステムの実行状態を可視化するための X11 を用いたツールである．[3] [4]
ARM ではタスクの実行状況の表示方法として，縦軸にタスク，横軸に時間をとっているが，このような表示方法には問題がある．まず第一に ARM では全タスクを表示しているため，タスク数が多数になるとウィンドウに表示できなくなってしまうこと，第二にすべての情報がリアルタイムで表示されてしまうため，データが提示されてもユーザに伝わらない場合がほとんどであること，第三に時間の刻み幅が大きいため，複数のイベントが短時間に発生した場合には各々のイベントの状態が把握できなくなってしまうことである．また，Mach における処理の実行環境はスレッドであるがタスクの実行状態を視覚化した例はあっても，タスクの中に含まれているスレッドに関する詳細を把握できるツールは存在しない．

このようなことをふまえた上で,

- 大量のデータから必要とするデータを抽出
- ユーザが必要とする時に情報を提示
- ユーザが理解しやすいユーザインタフェースの提供
- タスク・スレッド各々の状態の把握

を考慮した新たなパフォーマンス・モニタリング・ツールを提案する.

本研究では,上記考えに基づきシステムの実行状態を視覚化するツールの設計・構築を行ない,システムの実行中の様子を視覚化し,アプリケーションを最適化する場合の指標を示したい.

本稿で述べる内容は次のとおりである.2 章では今回の研究の背景として従来の情報視覚化の特徴とその問題点について述べ,関連研究についてふれる.3 章では,今回提案するパフォーマンス・モニタリング・ツールの設計について述べる.4 章ではその実装について述べ,5 章で実装の評価について述べる.最後に 6 章にて結論と今後の課題について述べる.

第 2 章

背景

本章では，情報視覚化（Information Visualization）について述べる．その後に，本研究で行なうシステムの実行状態の視覚化（Performance Visualization ）について述べ，従来のパフォーマンス・ビジュアライゼーションの手法の特徴とその問題点に関して考察する．

2.1 情報視覚化

近年，コンピュータの発達により大量かつ多種のデータが高速で処理されるようになってきた．人間がこのようなデータを的確に理解することは困難になってきている．このようにコンピュータで処理できるデータと人間の認知・洞察力能力とがバランスを欠いている状態にある．[15] このため，コンピュータを単なる計算道具としてではなく，計算結果のわかりやすいプレゼンテーションや対象とする現象のシミュレーションなどに利用することが重要である．情報をわかりやすく提示するには文字情報のみではなく，図式情報と組み合わせるほうが理解しやすいことが加藤・Jemes P.Chunningham らの研究で報告されている．[10]

情報の視覚化の目的は，情報を抽象化したり，データをそのままの形で表現するよりも，より人間にとって親しみやすくしたり，情報に対する理解度をより早くより深めることである．しかし，情報というものは色も形ももたないものであるため，それを視覚化する表現形態は柔軟性があまりに高く，表示の仕方によっては人間の理解を深めるという目的とは反して，情報の理解度を下げてしまうことにもなりえる．そのため，見るべき情報

の種類によってどのような手法で視覚化すべきかをよく考察しなければならない.

従来，大型のコンピュータで処理した大量の科学技術データの処理のために可視化の技術が用いられてきた．その代表的な例としてあげられるのが，気象・流体工学などの物理現象を可視化するサイエンティフィック・ビジュアライゼーション（scientific visualization ）の分野である．この場合には，数値データなどをわかりやすく表示することとともに見えないもの見づらいものの可視化が目的である，サイエンティフィック・ビジュアライゼーションの可視化の技術においては人間に対して情報を提示すること，つまり人間が情報を受けることが主になり，情報とのインタラクションはあまり重要視されない．またCTやMRIで得た医療画像のデータを可視化するメディカル・ビジュアライゼーションの分野においては，人体内部の忠実な再現が可視化の目的となる．

近年では高度なグラフィックス機能をもつ低価格，高性能なコンピュータが個人に対して普及した．その結果，視覚化の技術の利用は従来のような単なる科学的データの表現にとどまらず，計算機のユーザインタフェース，情報検索，ソフトウエア開発といった様々な分野でグラフィックスが利用されるようになってきた．近年の情報視覚化は，従来の視覚化とは異なり，情報を提示するだけでなく，情報とのインタラクションが重要である．また，情報量が膨大になって来たため，必要な情報をいかにして抽出し表現するかということも重要である．

2.2　システムの実行状態の視覚化

本研究で目的とする視覚化はシステムの実行状態の視覚化（Performance Vizualization）である．
これまで，計算機の実行状態の視覚化というと，並列処理において並列で実行されている処理の実行状況などが多くを占めているが，本研究で行なうシステムの実行状態の視覚化とは1計算機における実行状態の視覚化である．以下にPerformance Visualizationに関連研究を示す．

2.2.1 関連研究

PerVis

PerVis [6] は，Multilisp のプログラムの実行状態を視覚化するツールである．横軸に時間，縦軸にタスクを取った帯グラフ状のガントチャートによってタスクの実行状態を把握することができる．以下の図 2.1は Par Vi sのディスプレイである．タスクの 3 つの状態を色別にして表している．

- プロセッサが実行しているタスク（Runni ng)：dark gray
- wor ki ngqueue の中にあるタスク（Queued)：medi um gray
- プロセッサの空き待ちをしているタスク（Wai ti ng)：l i gh dray

タスク間の親子関係，データの依存関係が矢印で示されている．タスクの状態を見るためのもので，Mil ti Lisp以外には対応していない．

JED

各イベントの状態の詳細を示した例がある [7]．Just An Event Displa y (JED) はユーザが見たいイベントを選びその情報を見ることができるツールである．
これは， Cedar multiprocessor system の parallel program のイベントの様子を視覚化するためのものである．このツールでは， event id, processor id, cluster state, event data size, timestamp, dsize bytes of event data などの情報が得られる．また，イベントに名前とアイコンを割り当て，イベントが発生した時に対応するアイコンを表示することでイベントの実行状態を示す．図 2. 2参照.

Vi s uaLi nda

Vi s ua Li nda[14]は，並列プログラミング言語 Li nda の実行状態を視覚化するためのシステムである．これはプロセスの状態を 3 次元空間に柱状に示したものである．柱がプロセスで，柱と柱を結ぶ点線でプロセス間通信を示している．マウスの操作によって各タスクの状態を表すことが可能である．この手法は 2 次元で表現するよりも多数のデータを表示できることが特徴であり，並列プログラムにおける通信のバグの発見などに用いら

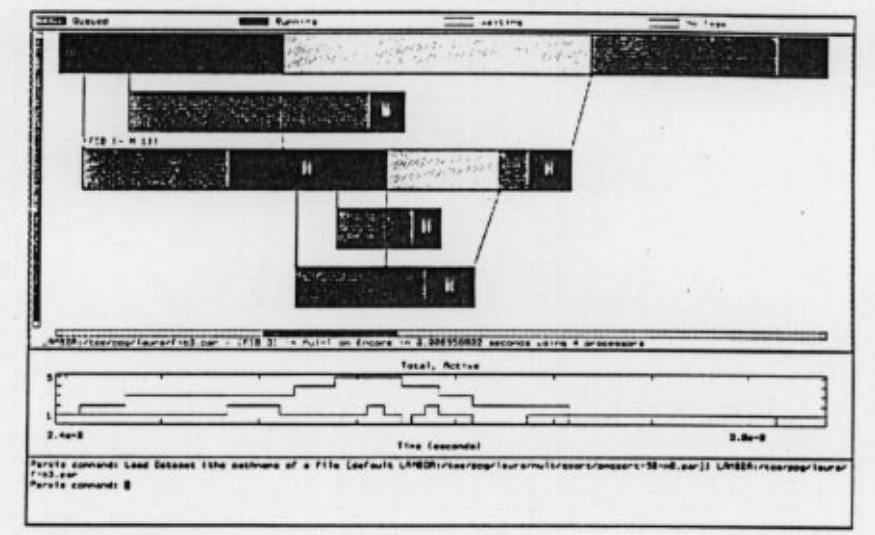


図 2.1: PerVis による視覚化：Multilisp code で書かれたフィボナッチ数列の計算の様子の例

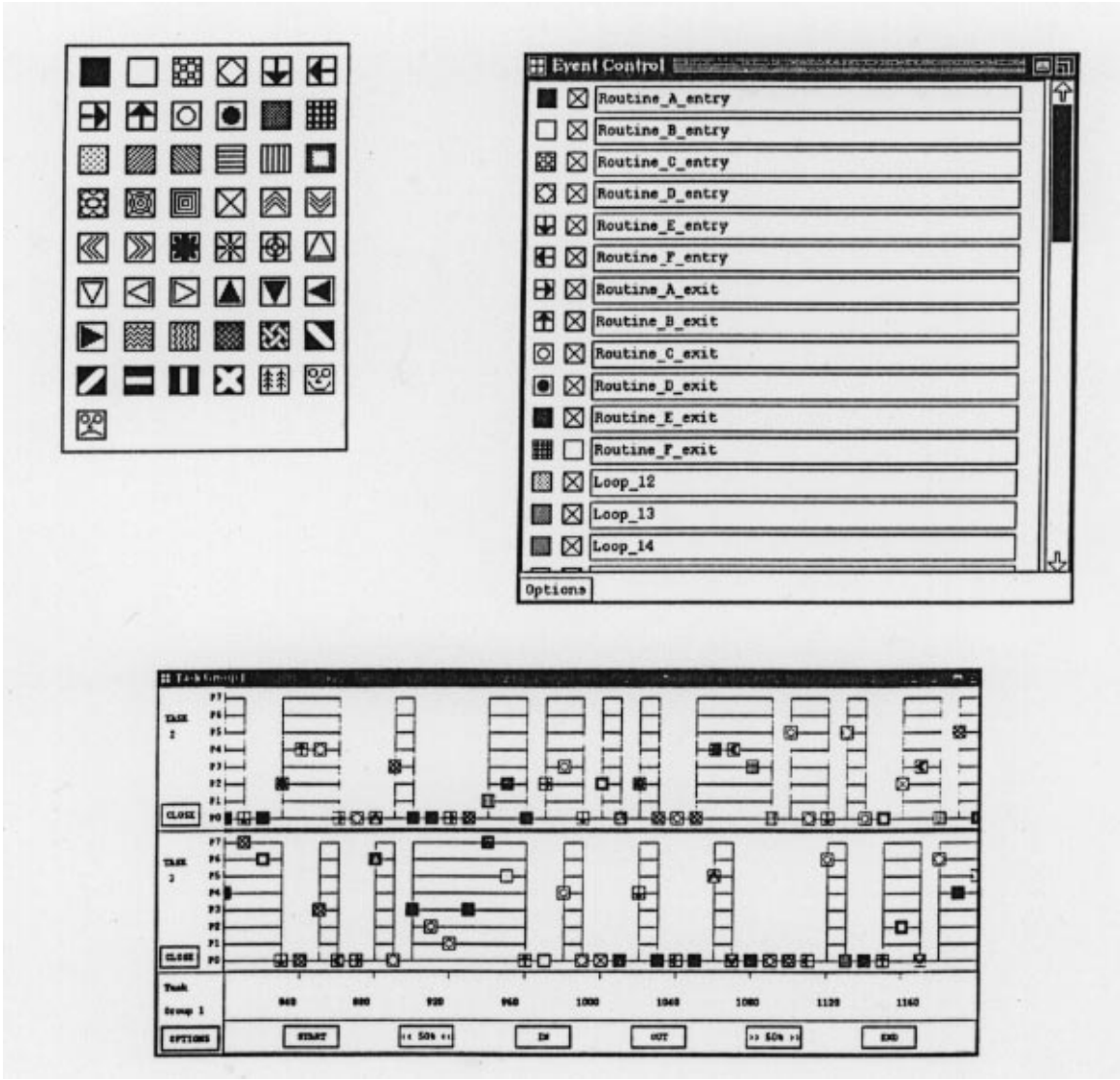


図 2.2: JED による視覚化の例, 左上：Event image map, 右上：Event control, 下： Task display a two tasks

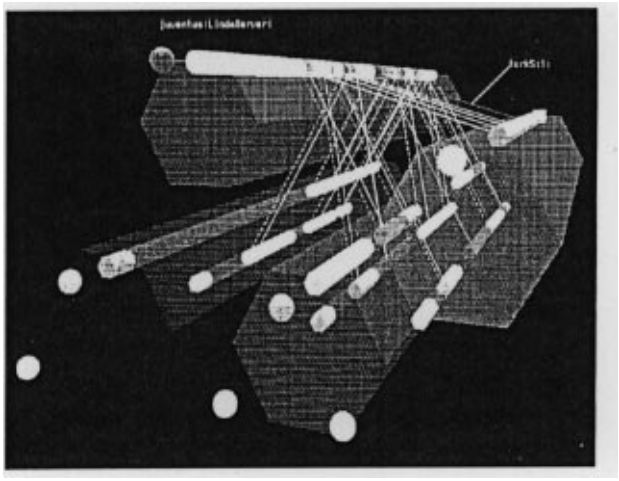


図 2.3: VisuaLinda による視覚化の例

れている．図 2.3参照.

PIE

PIE [12] は Carnegie Mellon University の T. Lehr らが作成したツールで， Mach OS 上で稼働する多重プロセスを視覚化するシステムである．これをカーネルのコンフィグレーションの際に利用した．図 2.4参照.

ARM

ARM（Advanced Real-Time Monitor）[13] は，分散リアルタイム OS Real-Time Mach 上で実行されたアプリケーションのスケジューリングに関するイベントを可視化するためのツールである．図 2.5参照.

一般にアプリケーションが正しく実行されたかどうかを検証することは非常に困難なことであるが，ARM で各スレッドが実行した状態を見ることで確認することができる．システムの状態をモニタするために，測定対象のマシンので reporter と呼ばれるプログラムが実行される．reporter はリモートホストのイベントの情報をローカルホストへ定期的に送る．図 2.6参照．Event Tap がカーネル内のイベントを Buffer に入れ， Reporter が

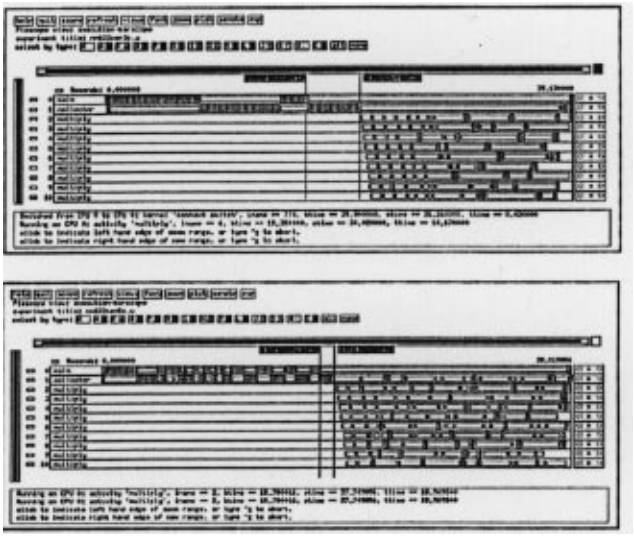


図 2.4: PIE による視覚化の例

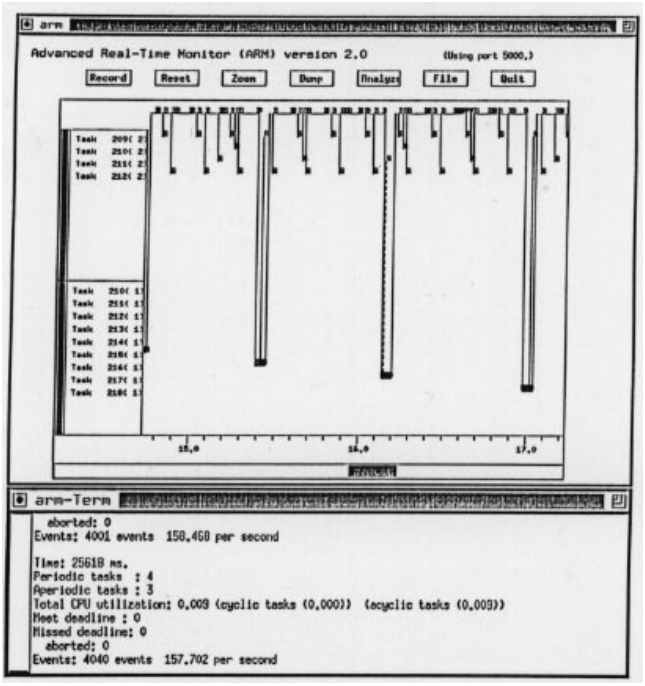


図 2. 5: ARM による視覚化の例

Buffer の中の情報をローカルホストへ送る．本研究では，このカーネル内のイベントを取得する部分と得られたデータを送信する部分とを利用する．

2.3 Mach の基本概念

本研究は Real-Time Mach 上のシステムの実行状態の視覚化を行なう．この Mach では様々な機能を提供しているが，以下に Mach が提供しているプリミティブを簡単に述べる．

- タスク
  従来の UNIX プロセスは Mach では２つの独立した構成要素に分けられる．その１つがタスクであり，もう１つがスレッドである．タスクは１つあるいは複数のスレッドが実行できる環境である．タスクはページングされる仮想記憶領域やシステムのリソースへのアクセスなどを含んでいる．タスクは受動的なリソースの集まりでありプロセッサ上で実行するものではない．

- スレッド
  スレッドは能動的な実行環境である．１つのタスク内にある複数のスレッドはそのタスクのすべての資源を共有している．
  Real-Time Mach ではカーネルレベルスレッドに対して時間属性を持たせることによってリアルタイムスレッドを実現している．このリアルタイムスレッドは，周期的にアプリケーションの関数を実行する周期スレッドと，非周期的な実行を行なう非周期スレッドがある．ビデオや音声のような連続メディアアプリケーションは，時間的制約があるため，決められた時間内に正しい処理を行なう必要性がある．このため連続メディアを扱うために周期スレッドが用いられ，時間的制約のないものには非周期スレッドが用いられることが多い．

- ポート
  スレッドが相互に相互に通信するための通信チャネルがポートである．ポートはリソースでありタスクが所有している．

- メッセージ
  異なるタスク中のスレッドはメッセージを交換することで相互に通信を行なう．ス

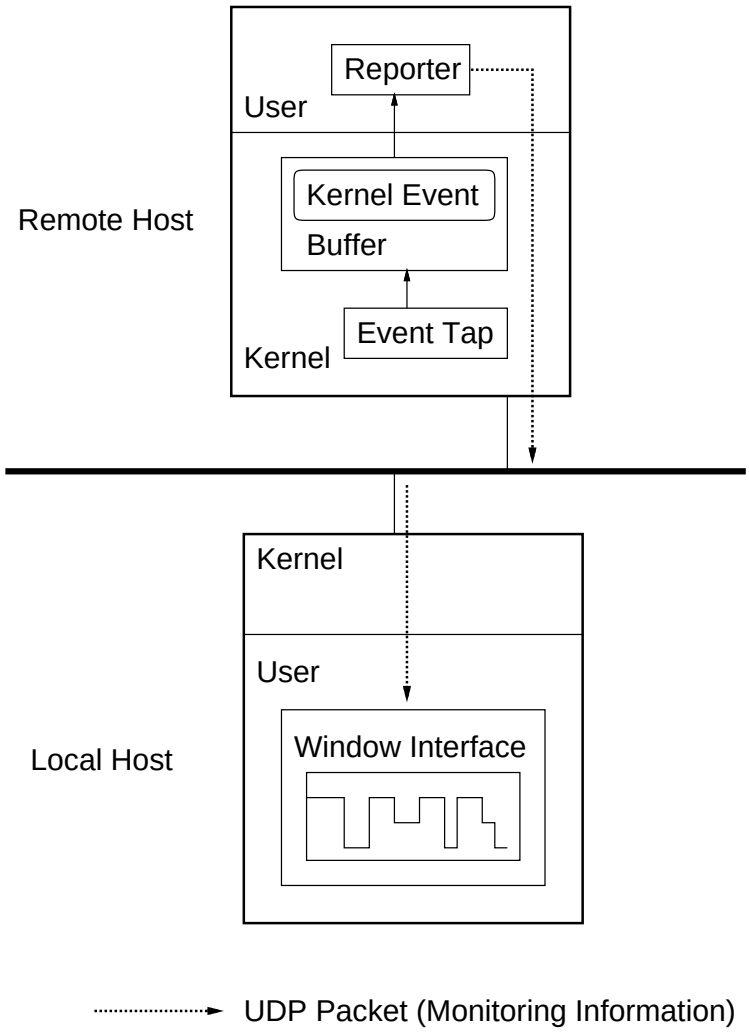


図 2.6: ARM の全体構造

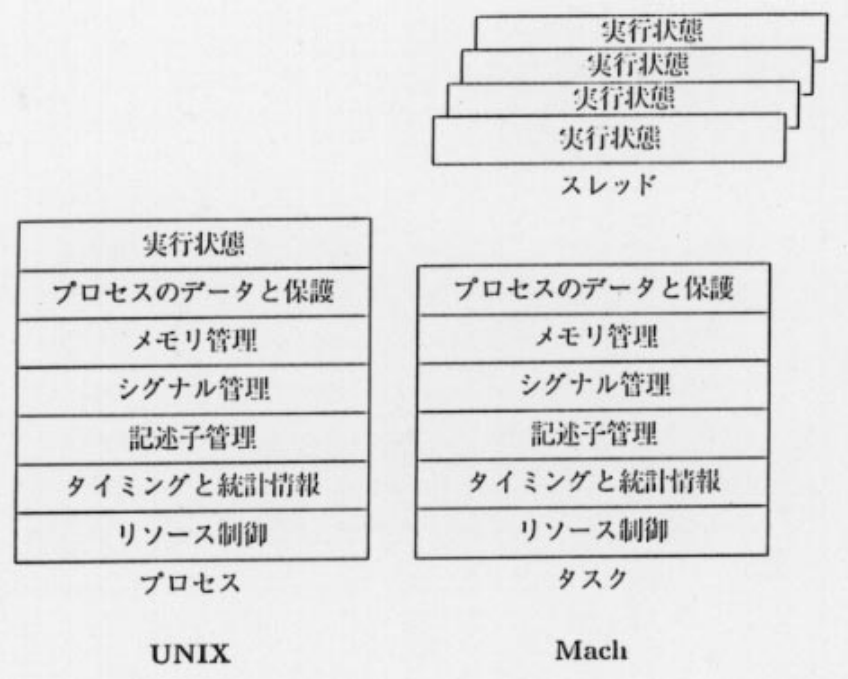


図 2.7: UNIX のプロセスと， Mach のタスクとスレッド

レッド間の通信に使われるデータオブジェクトの集まり．

- メモリオブジェクト
  Mach はユーザレベルのプログラム中で仮想メモリのページングポリシーをサポートするが，この機能をサポートする概念がメモリオブジェクトである．

図 2.7は UNIXと Mach との実行環境の違いをしめしたものである．図からもわかるとおり，Mach においては各スレッドで処理を実行する．よって，Mach 上においてシステムの実行状態を把握するためにはタスクの実行状態を知ることはもちろんであるが，それと同時にタスク内にあるスレッドの状態を把握することは不可欠である．

2.4 まとめ

縦軸にプロセスあるいは負荷，横軸に時間をとった 2 次元でのガントチャート形式のものは汎用性が高く，利用しやすい半面，縦軸に並べられる項目数には限りがある．3 次元で表現したものは，2 次元での表現に比べ多くのデータを表示することができるがデータ

数が多過ぎると視認性が下がってしまうため，表示すべきデータ数と視認性の良さとはトレードオフの関係にあるといえる．

得られたデータの全てを，正確に表示できたとしても，それら全てを使用者が必要としている情報であるのかどうかの吟味と，得られたデータから必要とするデータを抽出する場合，どのデータを視覚化し，どのデータを数値データとして表示すべきかの選択が重要だと言える．

本研究では Real-Time Mach 上におけるシステムの実行状態の視覚化を行なう．Real-Time Mach 上のシステムの実行状態を視覚化したものに ARM があるが，ARM では文字情報をリアルタイムでスクロールさせて表示させているため，情報が得られていても，使用者には伝わらない場合が多い．また，ARM では全タスクの実行状態を表示させているが，表示しているタスクが多過ぎるため必要とするタスクの実行状態が理解できなくなってしまっている．このため，ユーザにとって最適なインタフェースとは言い難い．

本稿ではこのようなを考慮し，タスクとスレッドの実行状態に着目し，多種のデータの中から，ユーザが必要とした情報を適切に提供するようなツールの構築を目指す．

第 3 章

設計方針

本研究では，タスクとスレッドの実行状態を把握すると共に，マシンとユーザとのインタラクションを重視している．得られたデータからユーザが指定したものを抽出しそれを表示できることが重要である．また，マシンの実行状態の測定にあたり測定対象のマシンに必要以上の負荷をかけないことが大事である．このようなことを考慮した上で，全体の構造とユーザインタフェースの設計を行なった．

3.1 全体構造

図 3.1は，本研究で構築するツールの簡略化した図である．ここでは，測定対象のマシンをリモートホスト，測定対象のマシンからデータを受信し，モニタリングするマシンをローカルホストとする．
構築するツールは以下のものから構成する．

- Event Tap：（ARM の物を拡張）測定対象のリモートホストのカーネル内に位置し，データを集め，バッファに貯める．
- Re port e:r（ARM の物を拡張）バッファの中のデータをローカルホストの Pe r fo r ma n ce Too l へ送る．
- Pe nt iumCo u nt e rCo nt r oSe r ever:（オリジナル）ローカルホストからカウンタで測定すべきイベント名などのカウンタ制御用の命令をうけとり，カウンタへその命令を送る．

- Monitor Tool：（オリジナル）
  - Data Analyzer：リモートホストから送られたデータを受けとり，各タスク・スレッドごとのデータとして蓄積する．そのデータを利用して CPU 使用率などの計算も行なう．
  - Visualizer ：ユーザインタフェースを提供する部分である．Data Analyzer で処理されたデータの表示を行なう．メインのウィンドウにはユーザが選択したタスクの実行状況を表示し，サブウィンドウでは選択したタスク・スレッド各々の CPU 使用率などを表示する．また，カウンタで測定した値なども表示される．

測定対象であるリモートホストにある reporter が，モニタリングを行なっているローカルホストに対して，モニタリングデータを送信する．ここで reporter の送信部は ARM で用いられていたものを利用する．モニタリングデータを送る場合には UDP を用いる．測定しているマシンで，実行状態の表示を行なわず他のマシンで表示を行なうのは，GUI の部分は負荷が高いため，Performance Tool の負荷を観測対象のマシンにかけないようにするためである．また，リモートホストにおいてペンティアムのカウンタが使えるのであればユーザは，モニタリングツールを用いて，カウンタでどのような項目を測定すべきかの指示を送ることができるようにする．カウンタで測定したデータも得ることで，よりシステムの詳細が把握できる．[5] このカウンタへの制御命令を送信する時は TCP を用いることにする．この理由は，reporter は短い時間間隔で定期的にリモートホストからローカルホストへ測定データを送信するのため多少パケットが消失してデータが失われたとしてもリアルタイム性に重大な影響は出ないが，カウンタの制御命令はユーザが選択した時に確実に届く必要があるためである．

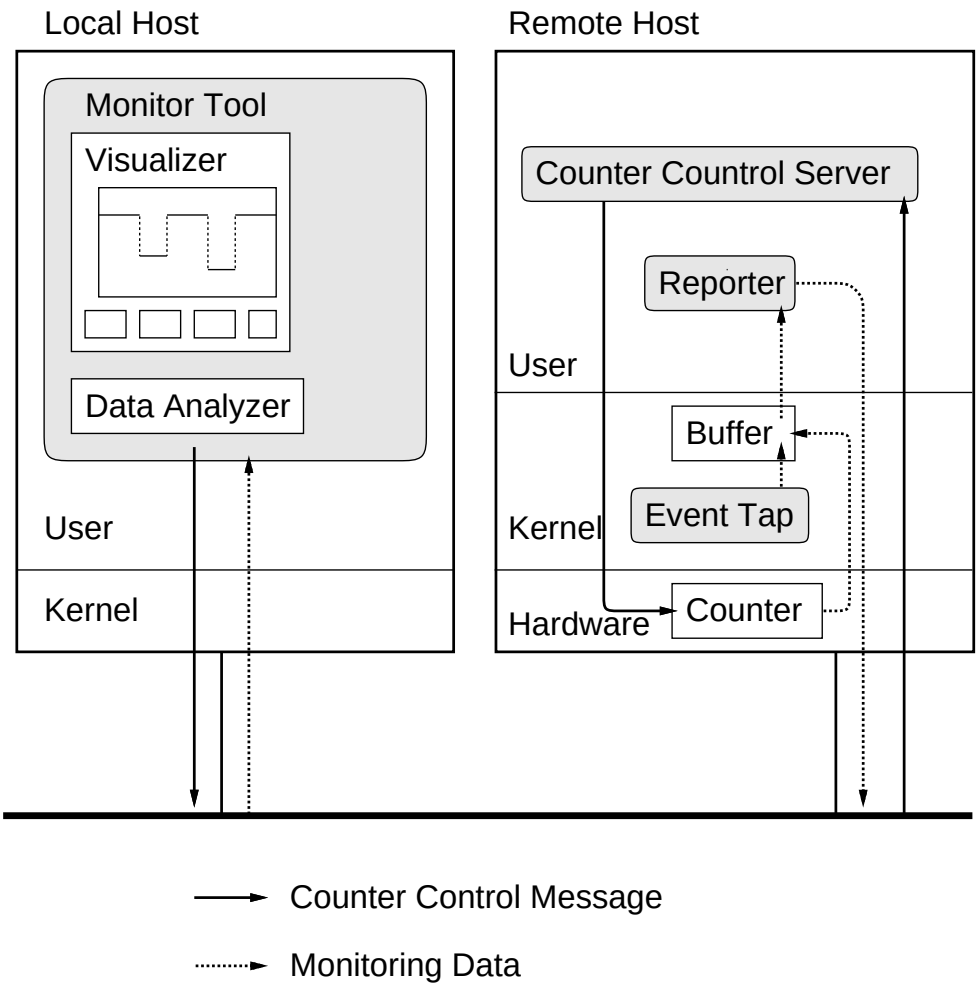


図 3.1: Performacnce Tool の全体構造の概略

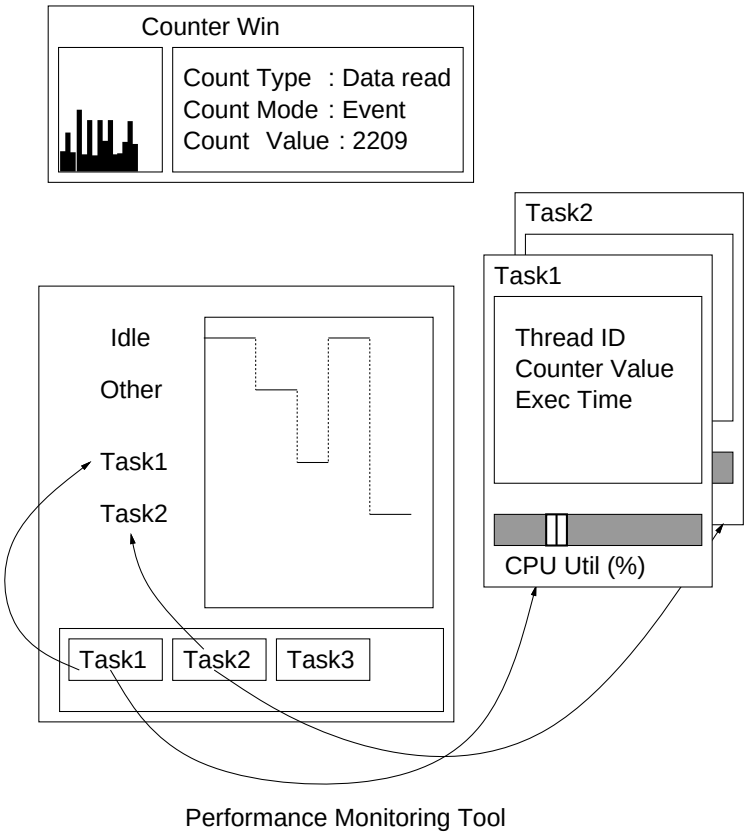


図 3.2: Performacnce Tool のユーザインタフェース部の概略

3.2 インタフェース

ARM ではタスク番号，CPU 使用率などすべての情報を，ウィンドウに文字を縦スクロールする形で表示していた．このスクロールのスピードが早いため情報が表示されていても，ユーザは読みとれなかったが，本研究では生成されたタスク ID を表示したボタンをメイン画面にボタン表示し，そのボタンをクリックすることでその選択されたタスク情報が提示されたウインドウが開くようにする．そして，CPU 使用率などはスライダーを利用して表示する．図 3. 2

文字情報（数値データ）と図情報（グラフ，スライダ，ボタン表示）とを組み合わせることで，使用者の理解度を深めることができる．[10] また，すべての情報を一度には表示せず，ユーザの指示を受けてから表示することにより，無数のデータが散乱することなく，ユーザに必要な情報だけを的確に抽出することができる．

第 4 章

実装

この章では，前の章で述べた設計に基づいて，実際のリアルタイム・パフォーマンス・モニタリング・ツールの実装方法について詳しく述べる．実装は，我々の研究室で研究・開発を行なっている Real-Time Mach におけるマイクロカーネルと，ARM のデータ転送部の一部を拡張し，使用した．カーネル側はカーネル内のデータを取得する部分に対し拡張という形で実装を行なった．また，ローカルホストにてリモートホストから受信したデータを表示するユーザインタフェース部分を新たに作成した．

4.1 全体構成

本章では，本研究でモニタするデータの流れとデータ構造とを解説する．まず全体構造であるが図 4.1参照されたい．

- Event Tap：測定対象であるリモートホスト上にあり，カーネル内のイベントを バッファへ貯める．（ARM の Event Tap のデータ取得機能を利用）
- Reporter：バッファの中のデータをローカルホストへ送る．
  （ARM の Reporter のデータ送信機能を利用）
- MonitTool：DataAnalyzer で処理されたデータを表示する部分．ユーザの選択に応じて，DataAnalyzer からのデータを示す．（オリジナル）
  - DataReceiver：Reporter から送信されたデータを受けとる．

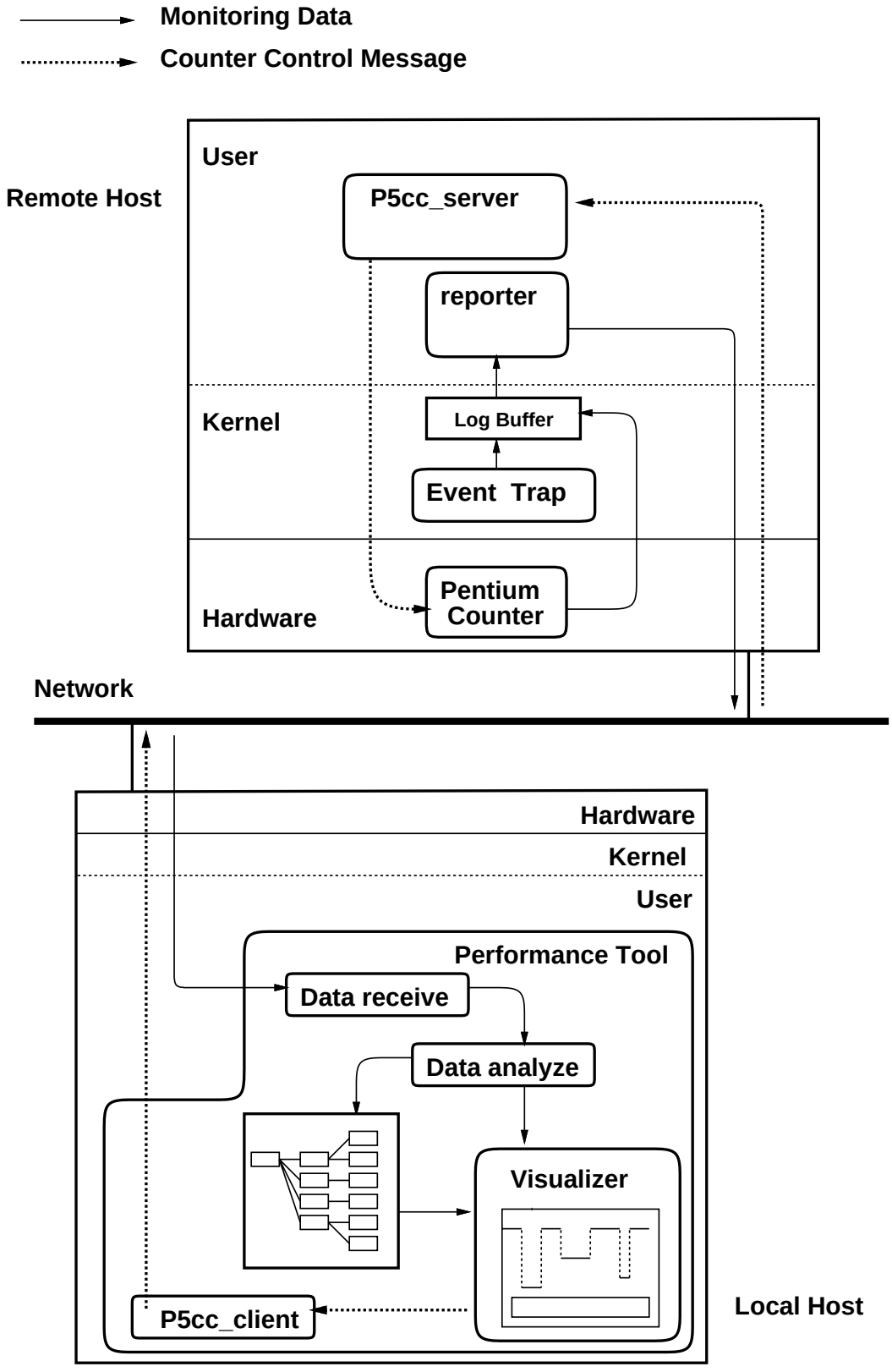


図 4.1: Performacnce Tool の全体構造とデータの流れ

- Data Analyzer：データの解析とデータの蓄積を行なう．この蓄積されたデータはユーザが必要となった場合に応じて使用される.（オリジナル）
- P5cc_client：カウンタで測定する項目をユーザが指定したら，リモートホスト上のp5cc_server へ送る．

- p5cc_server：ユーザが指定した，カウンタで測定すべき項目をカウンタへセットする．

4.2 Event Tap，Reporter

リモートホストにおいて，Event Tap がカーネル内のデータを取得，Reporter がローカルホストへ送信する．このモニタリングデータはリモートホストからローカルホストへの一方通行である．モニタリングデータの送信には UDP を用いている．このデータ取得部とデータ送信部は ARM の機能の一部を拡張して利用している．

Event Tap が kernel内で行なっている処理は以下のとおり．

- arm_sched_log()： task ID，theread ID，イベントの種類などをバッファに保存する．

- arm_sched_report()： reporter 側で，rt_log_report() システムコールが発行されると，この関数が最終的に呼ばれる．渡されてきたポインタの示すメモリ領域に，バッファ内のデータをコピーし返す．

reporter 内で行なっている処理は以下のとおり

- events_report()：reporter の中核
  Mach の周期スレッドとして定期的に（1秒毎に）実行される rt_log_report() を呼び kernelからデータを取得し，そのデータを UDP にて送信する．

- rt_log_report()：RT-Mach のシステムコール
  Event Tapが集め、buffer に貯めたデータを取得する．あらかじめ、メモリを allocate しておき，そのメモリのポインタを システムコールの引数として渡すと，そこにデータ (構造体の配列) が入って戻ってくる．

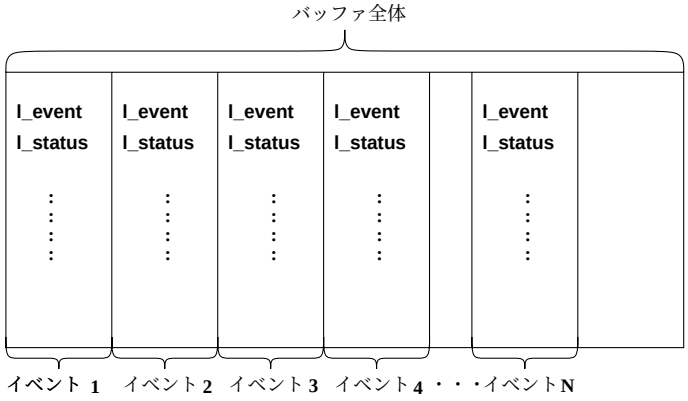


図 4.2: バッファの構造

4.2.1 データ構造

リモートホスト上の Event Tap がバッファに蓄積したカーネル内のデータは Reporter によってローカルホストへ送られる．この送られてきたデータを受けとり，処理する部分が Data Analyzer である．カーネル内のバッファ（図 4.2参照）の中には図 4.3のようなデータが入る．カーネル内でイベントが起こるごとにバッファにデータが蓄積されていく．

4.3 Counter Control Server

Intelが提供している Pentium では，カーネル内のイベントをカウントできるカウンタを提供している．このカウンタを用いることで，より詳しくシステムの実行状態を把握することができる．[5] [8] cpu 内のカウンタの設定 flag に、カウントしたいことをセットすると，その項目のカウントをする．たとえば、カウントしたいイベントとして、「Data read」という項目をセットすると，CPU が、data を メモリから read したという、event の回数 (cycle の数) をカウントする．どのようなイベントをカウントできるかは，後のセクションで示す．

リモートホストからローカルホストへ測定データを送信する場合は UDP を使用しているが，ローカルホストからリモートホストへカウンタの制御では TCP を使っている．その理由は，データを受けて表示する部分は多少データが失われても短い時間間隔で次のデータが届くため問題は起こらないが，取得すべきデータを指定する制御の命令は１つ１つ確

```
struct log_record {
    long                l_event;        /* スレッドの状態 */
    short               l_status;       /* 周期・非周期スレッドの判別 */
    short               l_monitor;      /* モニタしているか否かの判別 */
    long                l_thread;       /* スレッド ID */
    long                l_task;         /* タスク ID   */
    time_value_t        l_tstmp;        /* タイムスタンプ */
    unsigned int        l_cflag0;       /* カウンタ０にセットしている値 */
    unsigned int        l_cflag1;       /* カウンタ１にセットしている値*/
    long long           l_count0;       /* カウンタ０で測定した数値 */
    long long           l_count1;       /* カウンタ１で測定した数値 */
};
typedef struct log_record   log_record_data_t;
```

図 4.3: バッファ内のデータ構造

実に届く必要があるためである.

以下は Monitor Tool のp5cc_client にて行なう処理である.

- connect_p5cc()：ホスト名を指定し，データを送信するポートの確立を行なう.
- set_p5cc()：ユーザが指定したペンティアムのカウンタで測定する項目をリモートホストへ送信する.

```
connect_p5cc(char *hostname, int port)
{
        struct hostent          *hp;
        struct sockaddr_in      serv_addr;

        if ((hp = gethostbyname(hostname)) == NULL) {
                fprintf(stderr, "%s: unknown host.\n", hostname);
```

```
                exit(1);
        }

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(port);
        bcopy(hp->h_addr, &serv_addr.sin_addr, hp->h_length);

        if ((sockfd_p5cc = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                perror("Can't open stream socket to p5cc_server:");

        if (connect(sockfd_p5cc, (struct sockaddr *)&serv_addr,
                                    sizeof(serv_addr)) < 0) {
                perror("client: connect");
                exit(1);
        }
}

set_p5cc(char *cnt_str, char *sel_str, char *ev_str)
{
        char buf[64];

        printf("%s %s %s\n", cnt_str, sel_str, ev_str);
        sprintf(buf, "%s %s %s\n", cnt_str, sel_str, ev_str);
        send(sockfd_p5cc, buf, strlen(buf), 0);
}
```

4.4 Performance Tool

4.4.1 データ構造

リモートホスト上にある reporter から，ローカルホストへ送られてきたデータはローカルホストの Performance Tool 内で（図 4.1）以下に示すような構造でプールされる．マシン全体の情報，タスクごとの情報，スレッドごとの情報を配列に格納しポインタで次のパケットで届いたつぎのデータを指し示す．多種のデータが取得可能であるが，本ツールではリアルタイム性を重視したためすべてのデータを一度に表示することは行わず，使用するデータを，タスク ID，スレッド ID，タスク・スレッドの実行時間，などに絞っている．以下にデータ構造を示す．図 4. 4は以下のデータ構造の図解である．

```
struct thread_entry {
  struct thread_entry *next;

  int type;

  int last;

  long int thread_id;
  long int exec_time;
  long int runnable_time;
  long int chosen_time;

  int invocations; /* Number of times started. */
  int exits; /* Number of times exited. */
  int completions; /* Number of times task completed. */
  int cancelations; /* Number of times task canceled. */
  int abortions; /* Number of times task aborted. */

  int latency_total;
  int latency_ave;
```

```
  int latency_max;
  int latency_min;
  long long sum_counter_0;
  long long sum_counter_1;
  long long chosen_c_0;
  long long chosen_c_1;
  unsigned int cflag_0;
  unsigned int cflag_1;
};

typedef struct thread_entry Thread_entry;
typedef struct thread_entry *thread_HL; /* thread Hash List */

struct task_entry {
  struct task_entry *next;

  int type;

  int last;
  long int task_id;
  long int exec_time;
  long int runnable_time;
  long int chosen_time;

  int invocations; /* Number of times started. */
  int exits; /* Number of times exited. */
  int completions; /* Number of times task completed. */
  int cancelations; /* Number of times task canceled. */
  int abortions; /* Number of times task aborted. */
```

```
  int latency_total;
  int latency_ave;
  int latency_max;
  int latency_min;
  long long sum_counter_0;
  long long sum_counter_1;

  thread_HL HT_thread[HASH_SIZE];
};

typedef struct task_entry Task_entry;
typedef struct task_entry *task_HL; /* task Hash List */

typedef struct {
  long int start_time;
  long int end_time;
  long int time_periodic;
  long int time_aperiodic;
  int num_periodic;
  int num_aperiodic;
  int invocations; /* Number of tasks started. */
  int exits; /* Number of tasks exited. */
  int completions; /* Number of tasks task completed. */
  int cancelations; /* Number of tasks task canceled. */
  int abortions; /* Number of tasks task aborted. */
  int num_events;

  task_HL HT_task[HASH_SIZE];
} util_set;
```

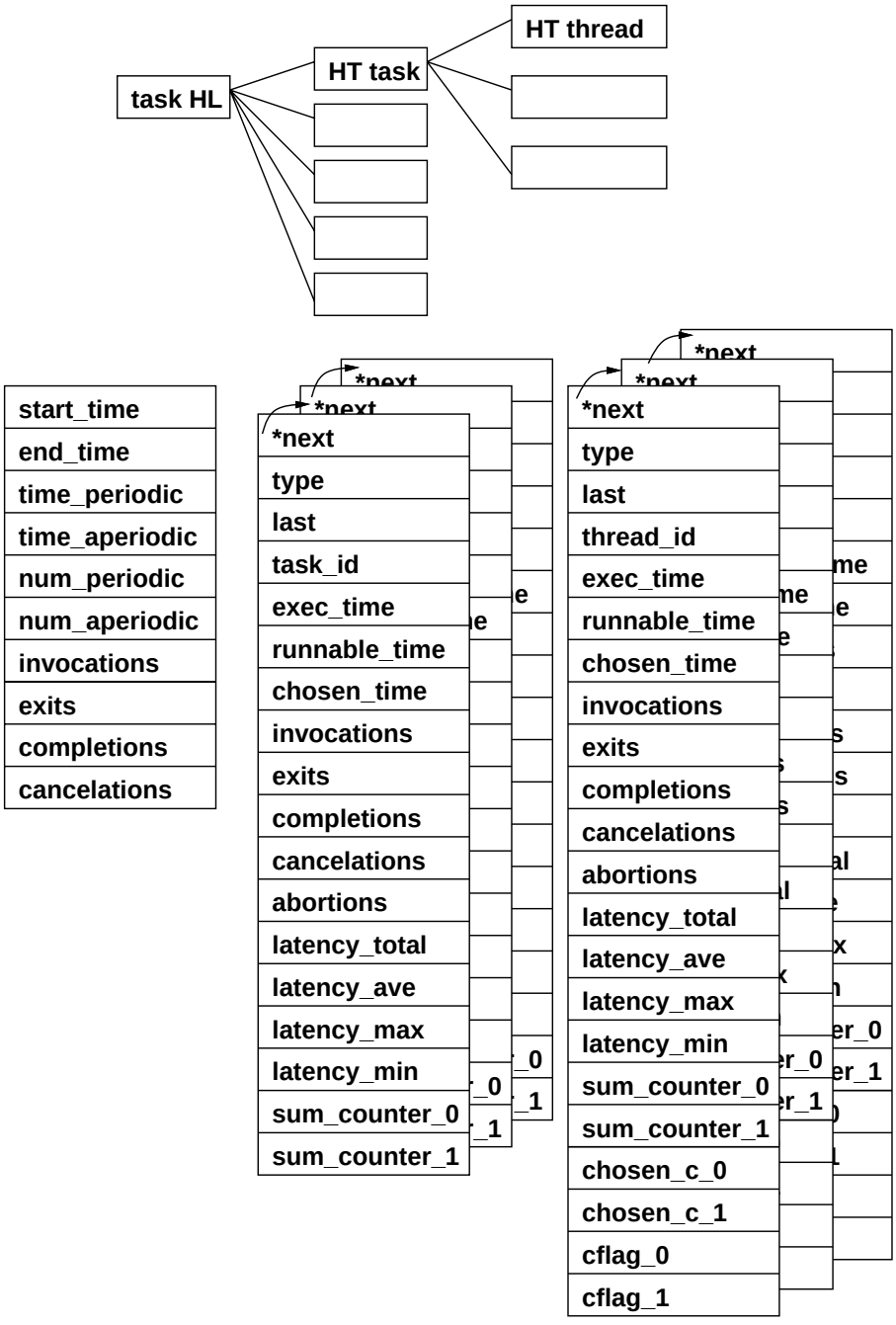
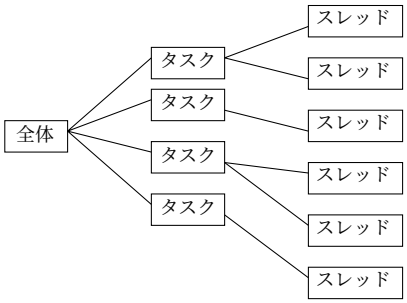
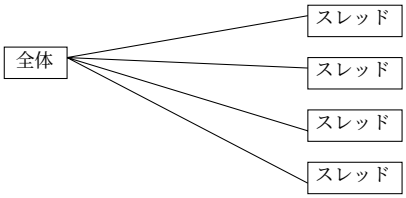

task HL
HT task
HT thread
start_time
end_time
time_periodic
time_aperiodic
num_periodic
num_aperiodic
invocations
exits
completions
cancelations
*next
type
last
task_id
exec_time
runnable_time
chosen_time
invocations
exits
completions
cancelations
abortions
latency_total
latency_ave
latency_max
latency_min
sum_counter_0
sum_counter_1
*next
type
last
thread_id
exec_time
runnable_time
chosen_time
invocations
exits
completions
cancelations
abortions
latency_total
latency_ave
latency_max
latency_min
sum_counter_0
sum_counter_1
chosen_c_0
chosen_c_1
cflag_0
cflag_1

図 4.4: データ構造



本研究で用いるデータ構造



ARM で用いるデータ構造

図 4.5: データ構造の比較

ここで ARM で用いられているデータ構造と，本研究で作成したツールで用いているデータ構造の違いについて簡単にのべる．図 4.5はそれぞれのデータ構造の概略を表したものである．ARM では全体情報とスレッド情報の 2 段構造になっているため，もしも，ARM で用いられているデータ構造で各タスク情報を表示するならば，全スレッドの情報の中から目的のタスク ID のものを探してきて合計することになり，検索のために多くのコストがかかるが，本研究で用いているデータ構造ならば，タスク・スレッド各々の情報を簡単に読み出すことができる．

4.4.2 データの解析

`stat_update()` は，reporter から受けとったデータを処理し Visualizer へ送る部分である．(図 4.1の PerformanceTool 内の Data analyze の部分である．)
ここで行なう主な処理は以下のとおりである．

- 各タスクの CPU 使用時間の合計を出し，全使用時間に対する割合を計算し（図 4.6 参照） へ送る．

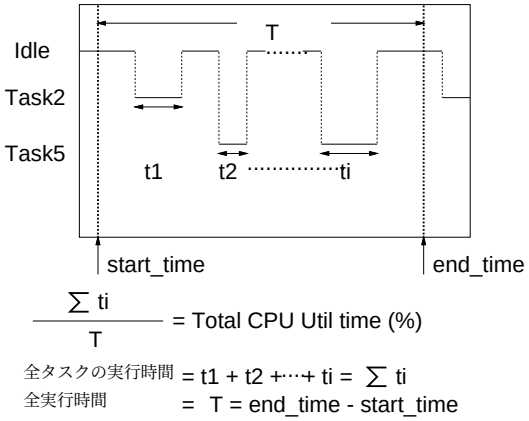


図 4.6: CPU 使用率の算出

- ユーザから指定されたタスクを検索し，タスク情報を PerformanceTool へ送る
- ユーザから指定されたスレッドを検索し，スレッド情報を PerformanceTool へ送る
- `task_HL find_and_create_task(long l_task)`
  reporter からタスクが生成したという情報が届いたら，PerformanceTool にタスクボタンを生成する．
- `delete_thread(long l_task, long l_thread)`
  reporter からタスクが消滅したという情報が届いたら，PerformanceTool に表示されているタスクボタンから，該当するタスク ID のボタンを探しだし，消去する．

ここで，`stat_update()` の処理を説明する．

```
stat_update(LOGRECORD *evnt)
{
        static  long  old_task_id = -1;
        static  long  old_thread_id = -1;
        task_HL c_task, o_task;
        thread_HL c_thread, o_thread;
        long int        time, dt;
        long long       diff_count;
```

↑使う変数の定義.

```
        us->counter_0 = evnt->l_count0;
        us->counter_1 = evnt->l_count1;
        us->cflag_0 = evnt->l_cflag0;
        us->cflag_1 = evnt->l_cflag1;
```

↑全体の値をセット
us は，全体のデータを保持する構造体で，
us->なにがしは，それぞれの値にアクセス.

```
        notify_get_event(evnt->l_task, evnt->l_thread, evnt);
```

↑「イベント発生を受け取った」ことを知らせる関数.

```
/* from RTM_analysis.c */
        if (evnt->l_event == TDS_STARTUP) {
                time = 0;
        } else {
                time = cnvt_time(&evnt->l_timestmp);
        }
```

↑送られてきた情報には，タイムスタンプがついている.
それをここでミリ秒に直している.
(一番最初にパケットを受け取った時の，そのタイムスタンプの
時刻 (base_time) を基準とし，そこからの時間経過が
帰ってくる.)

```
        if (us->start_time == 0) {
                        us->start_time = time;
```

```
}
```

↑ cpu 使用率を計算するために使用する.
　使用率の計算に使う時間の，始まりの時刻を保存.

```
us->end_time = time;
```

↑ 使用率の計算に使う時間の，終わりとなる時刻を保存.
　（最新のイベントの時刻でもある）

```
c_task = find_and_create_task(evnt->l_task);
```

↑ evnt->l_task の番号の task のデータを取り出してくる.
　c_task は，構造体.（c_task は，current task の意）
　まだ，その task ID のデータがツリーにない場合は，
　自動的に追加される.

```
c_thread = find_and_create_thread(c_task, evnt->l_thread, evnt);
```

↑ evnt->l_thread の番号の thread のデータを取り出してくる.
　親となる c_task の情報が必要.
　task 同様に，なければ追加される.

```
++us->num_events;
c_task->num_events++;
c_thread->num_events++;
```

↑それぞれのイベント発生の数を +1

```
plot_thread_status(evnt->l_task, evnt->l_thread,
                   evnt->l_status, time, c_thread->display_pos);
```

↑ main window にて縦の破線（スレッドの切替え状況を示す）を、
　描画する関数を呼出し．これは，イベントがひとつ届く度に描画.

（他は、TimeOut で描画）

```
switch (evnt->l_event) {
case TDS_RUN :
        c_task->runnable_time = time;
        c_thread->runnable_time = time;
        break;
        ↑ thread が実行可能 (runnable) になった場合の
          イベントの処理.
           runnable になった時間を保存.
case TDS_CHOOSE :
(ソースは省略)
        ここでは, thread が選択されたという処理を行なう.
        選択されたということは、実行が始まったということを意味する.
        つまり, context switch が発生したということである.
        そのため, 前に動いていた thread はそこで実行が中断される.
        前に動いていた task やスレッドのの情報は o_task, o_thread に
        (o_task=old task, o_thread=old thread の意) 残されている.

        よって, 古いほうに実行された時間や, p5 counter の増加分など
        の情報を足しあわせる. p5 counter の部分は, カウントする項目
        (flag) が変わった場合に, 加算する変数を 0 クリアする.

case TDS_COMPLETE :
        stat_proc_exit(c_task, c_thread, time);

        ここでは、thread が complete した時の処理を行なう.
        stat_proc_exit() は, thread が死んだ時の共通の処理を
        行なう関数である. 死んだスレッドが, 死ぬ直前まで使った
```

```
                cpu 時間などを加算したりする.

                ++us->completions;
                delete_thread(evnt->l_task, evnt->l_thread);
                ↑実際に，データ保存のツリーから，そのスレッドを
                  取り除く.（その際に thread がひとつもない場合には
                  task も消去される）

        case TDS_ABORT :
                thread が abort した時の処理.
                complete した場合とほぼ同じ処理を行なう.

        case TDS_CANCEL :
                thread が cancel された時の処理.
                complete した場合とほぼ同じ処理を行なう.
  }
}
```

4.5 ユーザーインタフェース

ここで，インタフェースについて説明する．まず関数の説明の後で，各ウィンドウの詳細の説明を行なう．

- `shell2_update_display_info()`
  この関数は Time Out ルーチン により定期的に呼ばれ，パフォーマンス・モニタリング・ツールのウインドウに表示する情報を表示・更新させている.（カウンタの値，全 CPU 使用率など）

- `wakeup_task_info_button()`
  パフォーマンス・モニタリング・ツールのメインウィンドウに生成したタスク ID を示したボタンを表示させる．

- `update_task_info()`
  この関数は Ti meOu t ルーチンにより定期的に呼ばれ，パフォーマンス・モニタリング・ツールのタスクウィンドウに，そのタスクに所属するスレッド数，そのタスクの CPU 使用率などの情報を表示・更新させている．

- `wakeup_thread_info_button()`
  パフォーマンス・モニタリング・ツールのタスクウィンドウに生成したスレッド ID を示したボタンを表示させる．

- `update_thread_info()`
  この関数は，Ti meOu t ルーチンにより定期的に呼ばれ，パフォーマンス・モニタリング・ツールのスレッドウィンドウにタスク ID，スレッド ID，そのスレッドの CPU 使用率などの情報を表示・更新させる．

4.5.1 メイン ウィンドウ

メインウインドウの様子は，図 4.7を参照されたい．図の中のウィンドウに番号がふってある，それぞれのウィンドウで表示しているものは以下のとおりである．

① 生成したタスク ID をボタンで表示，ボタンを選択すると（ボタンをクリックして選択）該当するタスク情報が表示されたウィンドウが開く（タスク ウィンドウ参照）

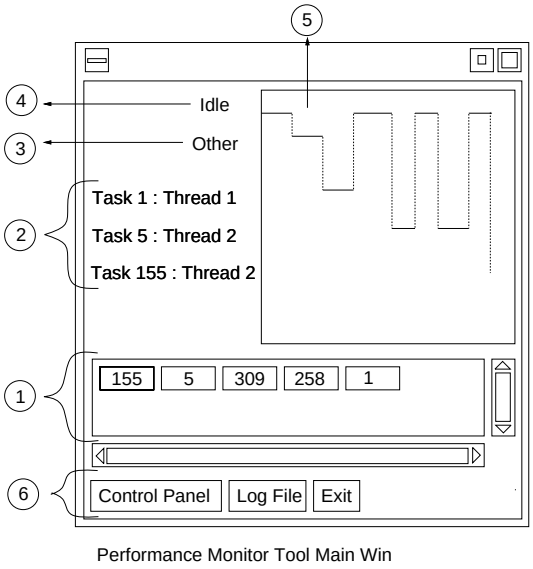


図 4.7: Performance Monitoring Tool メイン ウィンドウ

② ①で選択したタスクが表示されている，そのタスクと親子関係にあるスレッドも同時に表示される．

③ 選択されたタスク以外のタスクはすべて Othre として表す．

④ Idle 状態を表す．

⑤ 各タスクの実行状態を表す．横軸に時間，縦軸にタスクをとっている．実線の部分は実行時間，縦の破線はスレッドの切替えを表している．

⑥ Control ボタン：このボタンをクリックすると，ペンティアムのカウンタで測定する項目の指定をするウィンドウが開く．

4.5.2 カウンタ コントロール ウィンドウ

測定対象となるホストでペンティアムのカウンタが使用可能ならば，ユーザは，ペンティアムのカウンタでどのような項目を測定すべきかを選択することができる．以下の図 4. 8が，カウンタで測定すべきデータを選択するためのウィンドウである．

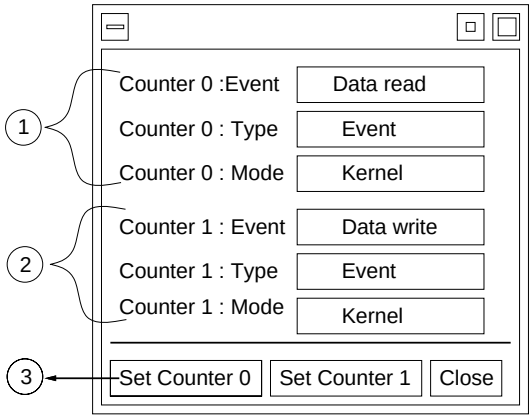


図 4.8: Performance Monitoring Tool カウンタ制御用 ウィンドウ

① カウンタ０（ペンティアムには２つのカウンタがある，カウンタ０とカウンタ１である．各々で別々な項目の測定が可能である）で測定すべき項目を選択する．右にならんでいるボタンをクリックすると測定できる項目の一覧が表示され選択するものをクリックすると選択した項目がボタンの上に表示される．

- Event：測定するイベント名（測定できるイベントの一覧は表 4.1に示す）
- Type ：カウントするタイプを選択,（event 数をカウントするならば，event を選択，cycle 数をカウントするならば， cycle を選択する）
- Mode：カウントするモードの選択（カーネルモード内で起きたイベントをカウントするならば Kernel を選択，ユーザモードで起きたイベントをカウントするならば User を選択，全て（カーネルとユーザ）をカウントするならば Kernel & User を選択）

② カウンタ１で測定すべき項目を選択.

③ １，２ で選択した項目をセットする．このボタンをクリックしたときに選択したカウンタで測定すべき項目をリモートホストへ送信する．

Pentium counter event types	
Data read	Pipeline flushes
Data write	Instructions executed
Data TLB miss	Instructions executed in the V-pipe
Data read miss	Bus utilization (clocks)
Data write miss	Pipeline stalled by write backup
Write (hit) to M or E state lines	Pipeline stalled by data memory read
Data cache lines written back	Pipeline stalled by write to E or M line
Data cache snoops	Locked bus cycle
Data cache snoop hits	I/O read or write cycle
Memory accesses in both pipes	Noncacheable memory references
Bank conflicts	AGI (Address Generation Interlock)
Misaligned data memory references	Floating-point operations
Code read	Breakpoint 0 match
Code TLB miss	Breakpoint 1 match
Code cache miss	Breakpoint 2 match
Any segment register load	Breakpoint 3 match
Branches	Hardware interupts
BTB hits	Data read or data write
Taken branch or BTB hit	Data read miss or data write miss

表 4.1: カウンタで測定できるイベントの一覧

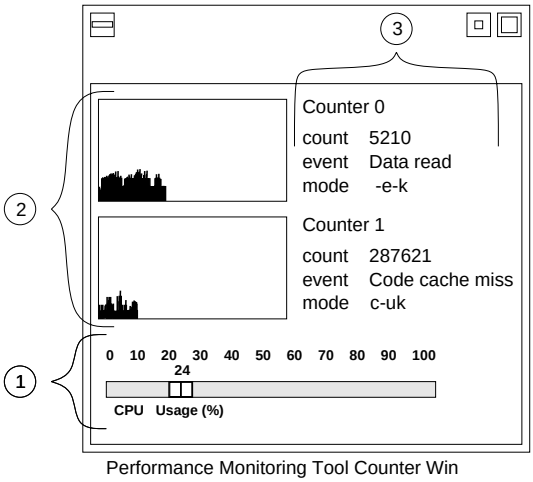


図 4.9: Performance Monitoring Tool モニタ ウィンドウ

4.5.3 モニタ ウィンドウ

ペンティアムのカウンタで測定した値とシステム全体の CPU 使用率を表示する場所がモニタ・ウィンドウである．図 4.9を参照されたい．表示している内容については以下のとおりである．

① アプリケーション全体が使用している CPU（%）

② 横軸：時間，縦軸：単位時間におけるカウンタの値

③ それぞれのカウンタで計測している内容

count カウンタの値，

event 測定しているイベント名（例：read miss，cache miss，など）

mode event 数をカウントする場合は e を表示，cycle 数をカウントする場合 c を表示，カーネル内で起きたイベントをカウントする場合 k を表示，ユーザモードで起きたイベントをカウントする場合 u を表示，全てのイベント（ユーザ・カーネル両方）をカウントする場合 uk を表示

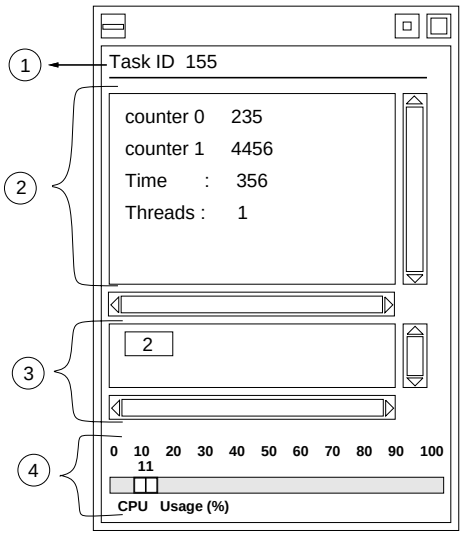


図 4.10: Performance Monitoring Tool タスク ウィンドウ

4.5.4 タスク ウィンドウ

メインウィンドウの中で表示されているタスク ID のボタンをクリックすると表示されるのが，タスクウィンドウである．このウィンドウは，タスクボタンに表示されているタスク ID に該当するタスクの情報を表示するウィンドウである．図 4. 10を参照.

① タスク ID を表示

② タスクに関する情報を表示

counter0, counter1 そのタスクの実行時間中のカウンタの値

Time タスクの実行時間

Threads タスクと親子関係にあるスレッドの数

③ タスクと親子関係にあるスレッドのスレッド ID をボタン表示する，ボタンをクリックすると該当するスレッド ID のスレッド情報を表示する.

④ そのタスクの CPU 使用率を表示している.

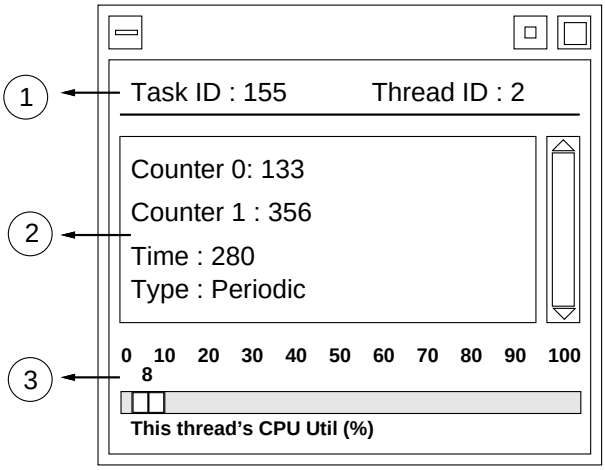


図 4.11: Performance MonitoringTool スレッド ウィンドウ

4.5.5 スレッド ウィンドウ

タスクウィンドウの中で表示されているスレッド ID のボタンをクリックするとスレッドウィンドウが表示される．図 4.11を参照．このウィンドウでは，選択したスレッド ID に対応するスレッドに関する情報を表示している．

① タスク ID とスレッド ID を表示

② そのスレッドの実行中にカウンタで計測された値，そのスレッドの実行時間，そのスレッドのタイプ（周期スレッドならば Periodic，非周期スレッドならば Aperiodic を表示）が表示される．

③ そのスレッドの CPU 使用率を表示

第 5 章

評価

ここで作成したツールを使用し，リモートホストの状態を観察してみる．実行例が図 5.1 である．

このツールを実行するとまず 2 つのウィンドウが開く，1 つはモニタ ウィンドウで全体での CPU 使用率がスライダーで表示され，単位時間あたりのカウンタの値が棒グラフとカウンタなどで表示される．もう 1 つは，メインウィンドウでタスク ID が表示されたボタンを選択すると，ユーザが選択したタスクの実行状態が折れ線グラフで表示されタスクウィンドウが表示される．選択したもの以外は Other として表示されている．周期スレッドは黒，非周期スレッドは赤で表示している．横方向の実線が実行状態で，縦方向の破線はコンテキストスイッチである．このグラフを見ることで，各アプリケーションの実行状況が理解できる．たとえば，reporter のように一定時間ごとに実行されているタスクがあるとすると，メインウインドウのそのタスクの位置に同じ時間間隔で実行を示す実線が表示され，処理が終了すれば線の描画は行なわれない．

各タスクの情報は，メインウィンドウのタスク ID のボタンをクリックした時に開くタスクウィンドウで表示される．このタスクウィンドウではそのタスクと親子関係にあるスレッド ID のボタンを表示[1] している．スレッドの情報はタスクウィンドウの中で表示される．このように選択した情報を表示する方法をとったことによって，よりユーザ本位なインタフェースを提供したといえる．

[1] このようにボタン表示を行なった理由は，ユーザがタスクのリストの中から自分の選択したい項目を探し出すよりも，表示されているボタンから選択するほうがわかりやすく，かつ処理がスムーズであったからである．

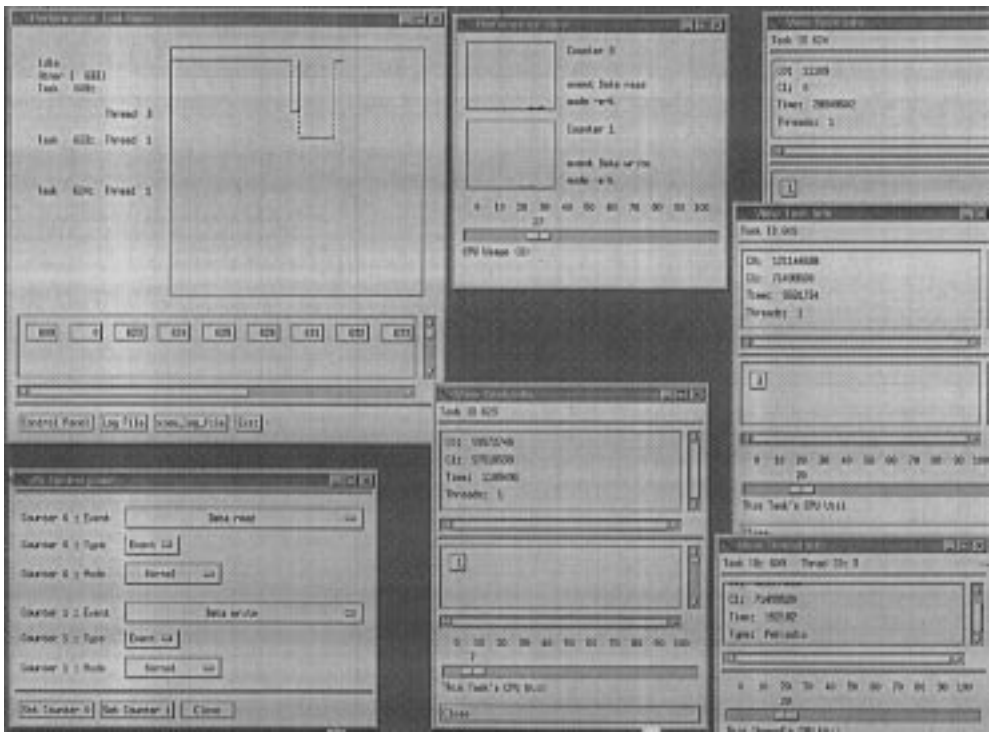


図 5.1: Performance Monitoring Tool の実行例

欠点として上げられるのが，レスポンスの時間である．ユーザが多くの情報を 1 度に見ようとして多くのウィンドウを開いた場合，何か選択しようとしてクリックしても反応が鈍くなる．また．ウィンドウにリアルタイムで描かれるグラフの描画が遅くなり不自然な状態になってしまう．

また，測定しているデータはリモートホスト上のカーネル内のバッファの中に貯めてからネットワークを通じて届くため，リアルタイムでモニタしてはいるがある程度時間差が出てしまうことが避けられない．

第 6 章

結論と今後の課題

従来の RT-Mach 上におけるアプリケーションの状況を把握するツールではタスク・スレッド各々の詳しい情報が得られなかった，また，得られたタスク情報のすべてを表示していたため，情報が表示されていてもユーザはそれを有効に利用することができなかったが，本研究では，ユーザが必要とするタスクを選択することによって，そのタスクの実行状況，そのタスクに含まれているスレッド数，タスク・スレッド各々の CPU 使用率，など情報を表示するようにした.

ユーザーに各タスクごとのリソースの使用状況を示すことは重要なことである．その理由は，RT- Mach では各アプリケーションが使えるリソースを動的に変える リアルタイム QOS を提供している．それぞれのアプリケーションがどのくらいリソースを使用しているかは，リアルタイムで変化するためその状況を把握することは難しいことであるが，実行状態を視覚化するツールを用いることで容易になる． 本研究では，ユーザがシステムの状態を理解するのを扶助するためにシステムの実行状態を視覚化するツールを提供した.

最後に今後の課題について述べる

・表示する情報

現在表示できるリソースの種類は限られているが，今後拡張することも可能である．ただし，リアルタイムで変動する QOS の状況をリアルタイムで観測するため，表示する情報数を増やすとツールのレスポンスが悪くなり，リアルタイム性が薄れてしまう．このため，リアルタイムでのモニタリングにおいては，多数の情報を表示できることが必ずしも良いこととは限らない.

・パケットを受けとり損ねた時の処理

本研究では，タスク情報などを測定対象のホストから UDP で送るためデータが欠落することもありうる．欠落したデータの中にタスクの消滅などの重要なデータが入っていた場合，モニタツールは消滅したタスクのことを探知できずに，画面に表示し続けてしまうため，ユーザはタスクボタンを見ただけでは，タスクが消滅したかどうかが判別できない．

解決策はいくつか上げられるが，１つはパケットを落さないようにするために，UDP よりも信頼性の高い TCP に変えること，もう１つは，一定時間ごとにタスクボタンを表示させているエリアに表示されているボタンを全てクリアして，その時点で実行されているタスクを検索し直し，表示を行なうことである．どちらの方法にしても，マシンにかかる負荷が高くなってしまうことは避けられない．情報の信頼性とリアルタイム性とはトレードオフの関係にあるため，リアルタイム性を重視する場合，ある程度の信頼性の低下はやむを得ないといえる．

・ユーザインタフェース

本研究ではタスク ID を選択することで，その選択されたタスク情報の表示を行なったが，これとはインタフェースを変えた方がより使い易いツールとなるだろう．たとえば，実行しているプログラム名から，その処理を行なっているタスクとスレッドの状態を出せるようにしたり，CPU 使用率の高い順にタスク ID を表示するなど，各タスクが使用したリソース量に応じて表示するなど使用者が選択できた方がより使いやすいツールとなるだろう．

謝辞

本研究を進めるにあたり御指導を頂いた中島達夫助教授に心より御礼を申し上げます. そして会津君をはじめとする中島研究室の皆様にはさまざまな面から御助力をいただき心から感謝いたします.

参考文献

[1] T. Nakajima H. Tezuka. Design and implementation of a continuous media storage system on real-time mach. *JAIST Research Report*, No. IS-RR-94-15S, 1994.

[2] T. Nakajima H. Tokuda and P. Rao. Real-time mach : Towards a predictable real-time system. *In Proceeding of the USENIX 1st Mach Symposium*, 1990.

[3] Makoto Kotera Hideki Tokuda. A real-time set for the art s kernel. *PROCEEDINGS Real-Time Systems Symposium*, p. 289, 1988.

[4] Makoto Kotera Hideki Tokuda and Clifford W. Mercer. A real-time monitor for a distributed real-time operating system. *Proceedings of ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging*, 1989.

[5] Kee Chan David Mazieres Antonio Dias Margo Seltzer J. Bradley Chen, Yasuo Endo and Michael D. Smith. The measured performance of personal computer operating systems. *The Proceedings of the 15th ACM Symposium on Operating System Principles*, 1994.

[6] Laura Bagnall Linden. *Palallel Program Visualization Using ParVis*. acm PRESS Performance Instrumentation and Visualization, 1990.

[7] Allen D. Malony. *Just An Event Display*. acm PRESS Performance Instrumentation and Visualization, 1990.

[8] THRJE MATHISEN Pentium secrets- cpus core technologies-. *BYTE*, p. 191, 1994.

[9] Tatsuo Nakajima and Hiroshi Tezuka. Experience with real-time mach for writing continuous media applications and servers. *JAIST Research Report*, No. IS-RR-94-19S, 1994.

[10] James P. Cunningham Joseph P. Rotella Shuichi Kato, Katsuhiko Ogawa. Design guidelines in the type of screen symbol commonly used for telecommunications systems in u.s. and japan. ヒューマンインタフェース, p. 9, 1992.

[11] Hideyuki Tokuda Tatsuo Nakajima Design and implementation of a user-level real-time network engine. *JAIST Research Report*, No. IS-RR-94-14S, 1994.

[12] Dalibor F. Vrsalovic Eddie Caplan Alan L. Chung Ted Lehr, Zary Segall and Charles Eftneman. Visualizing performance debugging. *IEEE Computer*, p. 38, 1989.

[13] 小池英樹. ビジュアライゼーション. ビジュアルインタフェース, p. 24, 1996.

[14] 高田哲司, 小池英樹. 並列言語 Linda のプログラムの実行状態の 3 次元視覚化. 1994.

[15] 中嶋正之・川合さとる. グラフィックスとマンマシンシステム. 1995. ISBN 4-00-010350-4.