2004－OS－96　(12)
2004／6／17

動的更新を支援するOSの運用時視覚化機構の提案

谷沢 智史　　小林 良岳　　中山 健　　前川 守

電気通信大学 大学院情報システム学研究科 情報システム設計学専攻

E-mail: {yazawa,yoshi,ken,maekawa}@maekawa.is.uec.ac.jp

概 要

本研究では，OSの動的更新を支援するための運用時視覚化機構を提案する．動的更新システムは多数提案されてきているが，これらのシステムを利用するためには，対象システムに関する知識が必要であり，十分な知識を持たない人間では運用は困難である．そこで運用時においてOSを視覚化することで理解を促すことが考えられる．しかし，視覚化に必要となるOS構成コードの実行状態を記録するためのトレース処理は，非常に大きなオーバヘッドをともなう．よって本稿では，32bit CPUを前提にOSの実行状態トレースをおこなうシステムを試作し，トレースのオーバヘッドを計測した．これに基づいて64bit CPUの特性を利用して削減する手法について検討する．

Run-time Visualization Mechanism for Operating Systems to Support Dynamic Software Updating

Satoshi YAZAWA　　Yoshitake KOBAYASHI
Ken NAKAYAMA　　Mamoru MAEKAWA

Department of Information Systems Science, Graduate School of Information Systems,
University of Electro-Communications

Abstract

This paper describes a real-time visualization mechanism for operating systems to support dynamic software updating. This technique is important to use systems supported by dynamic software updating, because many users of current systems don't understand correctly complex systems such as current operating systems. This problem can be resolved through a visualization mechanism for a running system, but a tracing mechanism to record proceedings of programs which consist of an Operating System has a large overhead. In this paper, we focus on a real-time tracing technique by softwares. We implement a prototype for the evaluation and the measurement of the overhead. Finally we discuss to reduce its overheads to take advantage of characteristics of 64bit CPUs.

1　はじめに

現在，ネットワークの進歩による情報端末の高度化により，OSを必要とする端末やOSの利用局面が多様化してきた．このような状況においては，OSの提供するサービスを柔軟に変更する機構が非常に重要である．そのため，実行中に動的更新可能なOSが提案されてきている [1]．

一方，OSは利用局面の多様化にしたがって，複雑化，巨大化してきた．これはアプリケーションソフトウェアについても同様であり，複雑，巨大なソフトウェアを設計，実装するための方法論，ツールが多数提案されている [2]．また，このようなソフトウェアを専門家ではない利用者に運用可能とするために，自律コンピューティング技術 [3] が提案されている．

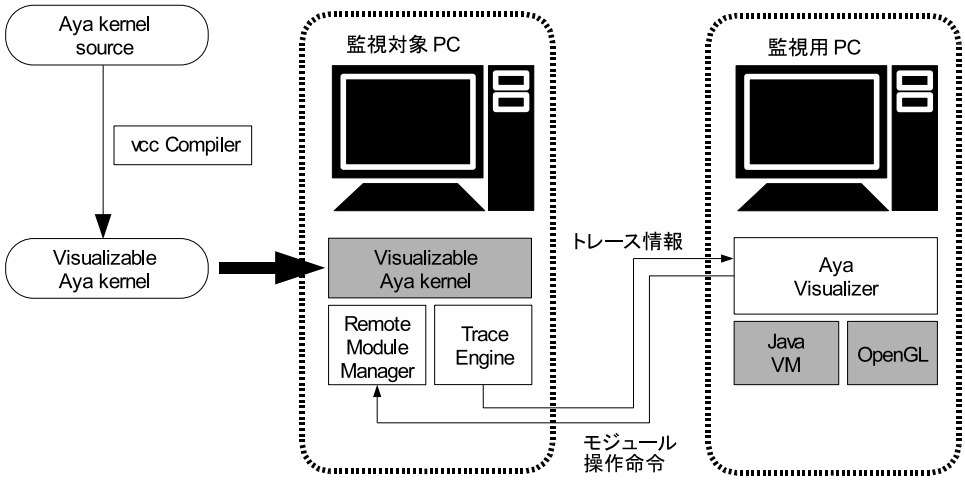

図 1: OS 視覚化システム

自律コンピューティングのアプローチは，企業においてはベンダが統一され，運用目標自体は固定的であるため，有効である．しかし，家庭においては複数のベンダ，新旧のシステムが共存する．また，ユーザの要求には，古いデバイスを使い慣れているので優先したい，新しいデバイスにすぐ移行したいなど様々なものがある．そのため，自律コンピューティングのアプローチは難しいと考えられる．

したがって，家庭利用においては，動的更新可能な OS は人間による直接操作によって運用すべきだと考える．しかし，現実的には，多くの人間がソフトウェアの構造を理解できず，理解していたとしても，その複雑さのためメンテナンス，障害対処などが非常に難しい．そこで，実際に運用している OS の中でモジュールがどのように連携し，サービスを実現しているのかを視覚的に提示し，これに基づいて人間が変更を指示する必要があると考えている．通常，ソフトウェアの実行状態の視覚化は運用中の OS には適用されず，デバッグに利用されることが多い．しかし，デバッグなどの限定された場合ではなく，運用中の OS を視覚化することには以下のような利点がある．

専門家ではないユーザが運用可能 アプリケーションがバージョンアップなどにより動作しない場合の問題追求において，専門家ではない利用者が他の環境との比較によって相違点を把握し，修復を試みることが可能となる．また，常に視覚化可能な状態にしておくことで，正常動作時の OS の振る舞いを直感的に把握することが可能となる．

開発環境によるデバッグの限界 最近の OS では複数のアプリケーションが同時実行されることが一般的である．このため，複数プロセスの特定タイミングによる問題が発生しやすくなっている．そのため，デバッグモードでは問題なく動作しても，実運用モードでは正しく動作しないことも多い．また，開発環境では正しく動作しても，実運用環境では正しく動作しないことも多い．このような問題は probe effect[5] と呼ばれ，開発者を悩ませる要素のひとつとなっている．運用中の OS を視覚化することで，このような実運用環境のみで発生する動作不良の原因を容易につきとめることが可能になる．

ソフトウェアの視覚化技術はすでに多くのものが研究されているが，現状では運用中の OS を対象にしたものはない．これは，視覚化のために実行処理を記録・処理をおこなうトレース処理が持つ，性能面での問題によるものであると考えられる．

そこで，本論文では，運用中の OS の実行状態を少ないオーバヘッドでトレースするためのアーキテクチャを提案する．オーバヘッド削減の手法としては，64 bit CPU の特性を利用し性能低下を抑制することを考えているが，今回は従来手法によるオーバヘッドを分析するため，従来の 32bit CPU で運用トレースをおこなう検証システムを試作し，

そのオーバヘッドについて調査した．

なお，本研究は，われわれが研究をおこなっている動的更新可能な OS Aya OS [1] を対象に運用中トレース処理の追加を試みる．

2 運用時視覚化機構の概要

2.1 システム構成

図 1 に Aya OS を視覚化，変更などの操作を可能にする管理システムの構成図を示す．

Aya OS はトレース用コードを埋め込む専用コンパイラ vcc によってコンパイルする．このトレースコードによって取得された情報はまず Trace Engine へと送信される．Trace Engine では，この情報を一定量バッファリングし，ネットワークを介して Aya Visualizer へと送信する．Aya Visualizer では，受信した情報をユーザに視覚情報として提示する．

この視覚情報をもとに，ユーザはモジュールの更新などを Aya Visualizer に対して指示する．この指示は，ネットワークを介して Remote Module Manager へと送信され，この指示が Aya へと反映されることになる．

以降に，各モジュールの詳細について述べる．

2.2 vcc コンパイラ

vcc コンパイラは，C 言語で記述された OS のソースコードに対して実行状態をトレースするコードをコンパイル時に埋め込む機能を持つ．本システムにおいては，Aya OS を構成する C 言語のコードに，後述する Trace Engine に以下の情報を引き渡すコードを追加する．

- 実行している関数の識別子 (16bit integer)
- 実行している関数の行番号 (16bit integer)
- 内部クロックサイクルの数値 (32bit long × 2)

実行している関数の識別子は以下の手順で決定する．まず，構文解析した内容に基づき，ローカルな関数識別子をモジュールを構成するコード内で一意に設定する．次に，最終的な識別子として，実行時の構成情報に基づいて実行環境に一意に識別子が決定される．また，実行している関数の行番号はソースコード内での行番号である．最後に，内部クロックサイクルであるが，これは `rdtsc` 命令によって取得された 64bit 値とする．

なお，一般的なデバッガで取得可能なユーザスタックの内容や特定のアドレスのメモリ内容取得は行わない．これは，オーバヘッドの大きいメモリアクセスを多く伴うためである．これらの情報を取得したい場合は，別途，デバッガを使うことにする．

2.3 Trace Engine

Trace Engine は，トレース処理が埋め込まれた Aya によって取得された情報をバッファリングし，適切なタイミングでネットワークに対して出力するモジュールである．

なお，この出力に利用するネットワーク処理も Aya の一部分である．そのため，出力処理時にはトレース処理を一時的に無効化するか，もしくは出力処理のトレース情報をメモリ上にのみ記録する，視覚化専用の小さなネットワーク処理モジュールを導入する必要がある．これは，OS の運用中トレース特有の問題である．アプリケーションのトレースの場合や，OS の部分的トレースにおいてはこのような問題は発生しない．

2.4 Aya Visualizer

Aya Visualizer は，Trace Engine によって送信されたトレース情報を受信し，視覚情報として提示する機能を持つ．視覚化には Java を利用し，プラットフォーム依存性を抑えることを重視する．また，Trace Engine と Aya Visualizer 間の通信方式についても同様にプラットフォーム非依存とする．これは，最適な視覚化表現は唯一ではなく，利用者が最も理解しやすいと感じた視覚化表現を自由に利用可能にすべきだからである．

また，視覚表現に基づいて，ユーザが Aya OS の構成変更を指示することを可能にする．構成変更には，モジュールのバージョンアップや，Aya OS で提案しているモジュールバージョン間でのインタフェース変化を吸収する機構 [4] に与えるインタフェース変換パラメータの変更なども考えられる．

2.5 Remote Module Manager

Remote Module Manager は，Aya Visualizer に対してユーザが行った操作をネットワークを介して受信し，Aya OS の構成を修正する機能を持つ．構成の修正には，Aya OS の持つ動的再構成機能を利用し，モジュールのインストールやアンイン

ストール，バージョンアップを可能にする．ユーザはこの変更によって動作がどのように変化したかを Aya Visualizer を使って即座に確認をすることが可能である．

一般的にモジュールの更新時には，モジュールに仕様として明示されていなかった振る舞いがバージョンアップによって変化し，そのモジュールを利用している機能が利用不可能になることがある．Aya Visualizer は正常動作時も常に視覚化および振る舞いの記録が可能である．これを利用して，異常動作時の視覚情報に正常動作時の視覚情報を重ねて提示するなどして，このような不具合を利用者が識別可能になることが望ましい．

3 検証システムの設計と実装

32bit CPU の命令のみを利用した場合のトレース処理のオーバヘッドを計測するため，検証システムの設計と実装を行った．検証システムの構成を図 2 に示す．

検証システムを構成する各コンポーネントの機能は以下の通りである．

vcc コンパイラ Aya OS のソースコードにトレース処理コードを付加してコンパイルを行う．

Trace Engine Aya OS のトレース処理で収集された情報を一定量バッファリングしネットワークに送出する．

Trace Receiver Trace Engine によって送出された情報を受信し，Aya OS の実行経過をテキストで出力する．

3.1 vcc コンパイラ

vcc コンパイラは OS のコードにトレース用情報を挿入するためのコンパイラである．Aya のソースコードをそのままトレース可能にするため，C 言語で記述されたコードを解析し，1 行ごとにトレースコードを追加する．動作フローを図 3 に示す．

vcc コマンドには，実行時に以下のファイル名を指定する．

- 関数 index ファイル
- ソース宣言ファイル
- トレース処理ファイル

OS を構成する C 言語のソースファイルは，まず cpp コマンドによってマクロなどが展開される．次に，コード挿入処理によって，ソース先頭にソース宣言ファイルが追加される．このソース内で，トレース処理で使用するマクロや関数プロトタイプを宣言する．さらに関数開始部が識別され，関数 index を参照し識別番号を取得する．関数 index に未登録の関数である場合は，未使用の識別番号を割り振り，関数 index に登録する．その後，変数宣言部を識別，スキップし，C 言語による処理 1 行についてトレース処理ファイルの内容を追加する．最後に，cc コマンドによってオブジェクトファイルへとコンパイルを行う．

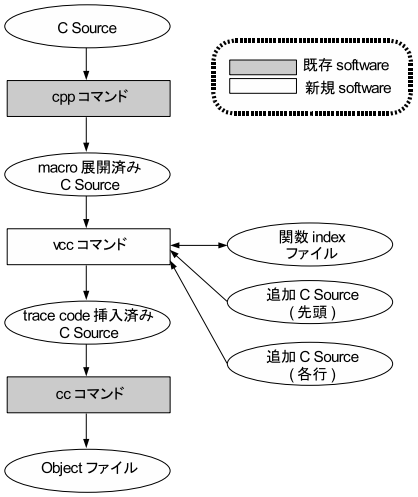

図 3: vcc 動作フロー

表 1: vcc で利用可能な組み込みマクロ

マクロ名	置換される内容
__FILE__	ファイル名
__FUNCTION__	関数名
__FUNCTION_ID__	関数識別番号
__LINE__	行番号

vcc コマンドは C 言語の字句解析器を持つが，C 言語の完全な構文解析器は持たない．関数宣言と変数宣言のみを限定的に識別できるレベルの構文解析器のみを持つ．

また，トレース処理ファイルには，表 1 で示す組み込みマクロを使用することができる．

本実装ではトレース処理ファイルに，挿入された関数の識別番号および行番号を引数に Trace Engine の関数を呼び出すコードを記述した．

3.2 Trace Engine

Trace Engine は，vcc によって挿入されたトレース処理から呼び出され，関数識別番号と行番号，アセンブラの `rdtsc` 命令で取得したクロックサイクル

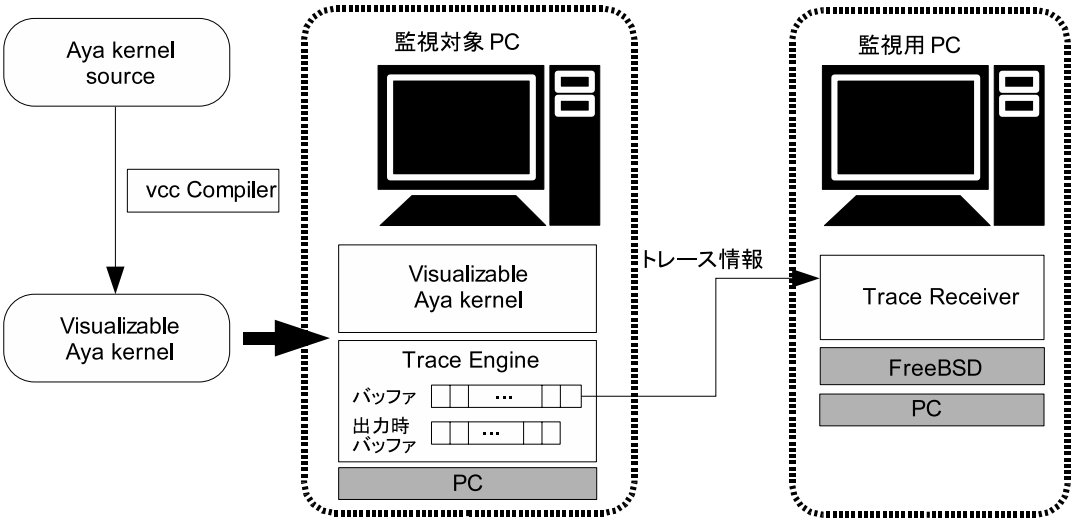

図 2: 検証システム

数をバッファに記憶する．その後，このバッファに空きがなくなるまで情報が蓄積された時点で，ネットワーク関数を利用して Trace Receiver に対してトレース情報を出力する．

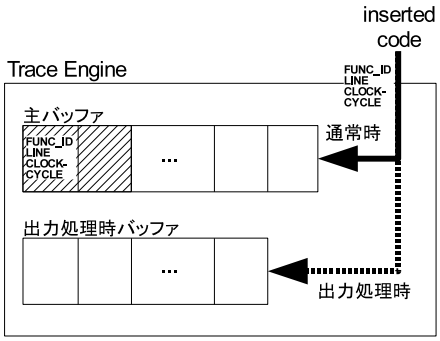

図 4: Trace Engine の構成

ここで利用されるネットワーク関数には Aya kernel が提供するネットワーク機能を利用する．よって，トレース情報を出力するための処理もトレースできることが望ましい．そのため図 4 に示すように，ネットワーク送信処理が活動している間のトレース処理は，出力時専用バッファへと蓄積する．そして，ネットワーク送信処理の終了時に，出力時専用バッファの内容を主バッファへとコピーすることで，ネットワーク送信処理をトレース対象とすることを実現している．なお，出力時専用バッファに空きがなくなった場合，送信処理が終了するまで，トレース情報は破棄することとする．

3.3 Trace Receiver

Trace Receiver は，Trace Engine からネットワークを介して送信されたトレース情報を受信し，その結果を出力するアプリケーションである．このアプリケーションは FreeBSD のコンソールアプリケーションとして実装している．

4 評価

評価用に作成したシステムについて，トレースのオーバヘッドを測定，評価した．評価システムの概要は以下の通りである．

- CPU: 2.8GHz
- OS: FreeBSD 4.8-RELEASE
- VMWare2(Network:Host-only)

なお，動作検証は VMWare によって行った．これは，Aya OS のネットワークデバイス対応が実機 PC に対応していないことによるものである．

4.1 32bit CPU におけるトレースのコスト

本実装において，トレース処理を挿入することでカーネルの操作がどの程度性能低下を起こすかについて検討する．

画面に対する printf(64 バイト× 8 回)，UDP による sendto(64 バイト× 8 回) のそれぞれの操作について，トレースなし，トレースあり/出力処理なし，トレースあり/出力処理あり (File, UDP) それぞれの場合について必要な処理時間を計測した．このグラフを図 5 に示す．

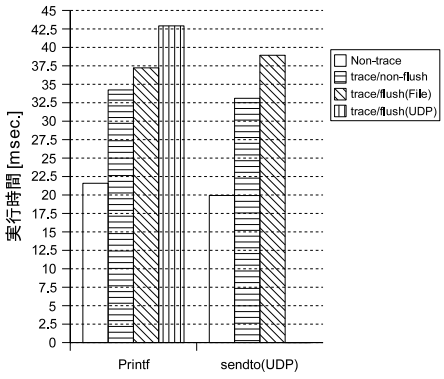

図 5: トレースのコスト

トレースあり/出力処理 (UDP) ありについて，UDP 送信処理内で UDP パケットを送信しようとした場合に内部のデータに不具合が発生するため，sendto(UDP) の値はゼロとした．

printf の処理時間が UDP の処理時間よりも長いのは，画面描画処理をともなうことに起因している．VMWare での画面描画エミュレーションは通常のコード実行に比べオーバヘッドが非常に大きいため，UDP よりも処理時間が長い．

次に，表 2 に，トレースを行わない処理を 1 とした場合のトレース有効時の出力処理がない場合とある場合の比を示す．

表 2: トレースコストの比

処理名	no trace	trace		
		no flush	flush	
			File	UDP
printf	1	1.59	1.72	1.99
sendto(UDP)	1	1.66	1.95	-

UDP による sendto システムコールのトレース時の処理時間が非トレース時の 1.66 倍となっており，対して，printf による処理時間が非トレース時の 1.59 倍となっている．この差は，printf による処理時間の多くは画面出力に費やされるのに対し，sendto によ処理時間の多くはパケットの加工に費やされており，ハードウェアへの出力処理にかかる時間はわずかであることに起因していると考えられる．

4.2 送信処理のトレースによるコスト増大

本実装においては，Trace Engine のネットワーク出力機能もトレース対象としている．そのため，トレースすべき情報量が増大し，性能を著しく低下させる要因となっている．ここでは，その性能低下について検討する．

Trace Engine は図 4 に示したように主バッファ，出力時専用バッファの二重構造によってトレース情報出力処理をトレースする．このため，出力処理のトレース内容をさらに出力する必要を招く．ここで，出力時専用バッファのサイズが主バッファのサイズ以上であると，出力処理の実行が，その出力処理のトレース情報の出力処理を発生させるという循環に陥り，1 行ごとに大量のトレース情報を Trace Receiver に送信するという事態を招く．

図 6 に，主バッファのサイズを固定した場合の出力時専用バッファのサイズの変化とトレース情報出力回数の関係を示す．この実験では，バッファリングせずに null デバイスへ 1000 文字を書き出す処理において，主バッファサイズは 1024 個に固定とし，出力専用バッファサイズを 0 個から 1000 個まで変化させ，出力回数の変化を調べた．なお出力処理には，ネットワーク出力処理よりも実行すべき行数が少ないファイル出力処理を設定した．

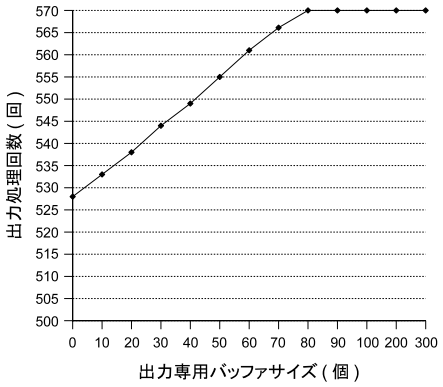

図 6: 出力専用バッファサイズと出力回数

図 6 より，出力時専用バッファが 100 個以下では出力時専用バッファの追加にともない出力回数が増加するが，それ以上では出力回数は増加しないことがわかる．これは，ファイル出力処理で実行すべき行数が 100 行程度であるため，100 個以上に出力時専用バッファを設定してもすべてにデータが記録されるわけではないためである．したがって，出力時専用バッファはトレース処理そのものの振る舞いの把握に必要であるが，出力処理が軽量な実装でなければパフォーマンスを著しく阻害する要因となる．

5 64bit 化への考察

試作システムで得られた評価をもとに，トレー

スシステムを 64bit CPU 専用にするための手法について考察する．

現在の高速 CPU においては，メモリアクセス時間のオーバヘッドが性能上のボトルネックとなってきている．トレース処理に要するコストの検証で得られたとおり，出力処理はせず実行経過を記録する処理のみの場合でも主処理に対して非常に大きなオーバヘッドをともなっている．これを削減するためには，トレース処理の多くを占めているメモリアクセスのオーバヘッドを削減する必要がある．

そこで，64bit CPU に採用されている技術のうち，メモリアクセスのオーバヘッド改善と関連のあるものを挙げる．

5.1 メモリインタリーブ

メモリインタリーブ技術とは，メモリを複数のメモリバンクという単位に分割し，異なるメモリバンクへの同時アクセスを高速化する技術である．複数のメモリバンクへアクセスする際は，片方のバンクへのアクセス待ち時間にもう片方のバンクへとアクセス要求を出すなどにより，待ち時間の有効利用を実現している．これを 32bit CPU においても標準的な技術となっている命令の並列実行機能と組み合わせることによって，トレース処理によって発生するメモリアクセスのコストを削減することができると考えられる．

具体的には，関数識別子や行番号，クロックサイクル数の代入をレジスタに対して直接実行し，そのレジスタからメモリへの書き出し処理のみをトレース対象の処理と同時実行してしまうことが考えられる．なお，この場合，レジスタの値がトレース対象処理によって上書きされてしまうことのないように，cc コマンドによってコンパイルされた機械語コードを解析し，未使用レジスタを割り付けるといった工夫が必要であると考えている．

5.2 レジスタ長，データバスの拡張

64bit CPU においては，レジスタ長とデータバス長も 64bit に拡張されている．これは，試作プログラムでは複数回に分割して出力しているメモリ書き出し処理を 1 つ，あるいは 2 つへと削減できる可能性を示唆している．コンパイル時に演算可能な処理はあらかじめ処理しておく．たとえば関数識別子と行番号はひとつの 64bit 値としてプログラムコードに直接記述しておく．このような対処によって必要なメモリアクセス処理の回数を削減することで，メモリアクセスのオーバヘッドを削減できると考えられる．

6 関連研究

ソフトウェアの実行状態を視覚化するためのトレース手法には大きく分けてハードウェアによるトレース手法とソフトウェアによるトレース手法がある．それぞれの利点と欠点を挙げる．次に，OS 内部をトレースする技術としてカーネルデバッガについて検討する．

6.1 ハードウェアによるトレース手法

プログラムの実行速度を低下させずに実行状態をトレースする手法として，ハードウェアによるトレースがある．

これは，CPU にトレース専用外部ハードウェアを接続し，実行状態を取得する手法である．そのため，CPU の実行速度低下は伴わない．このようなハードウェアを実時間システムに適用した研究がある [6]．ただし，CPU の動作クロックが高速になった場合，外部ハードウェアを同期することが困難となるため，トレースは不可能となってしまう．本研究で対象とするシステムは現在の PC も含むため，ハードウェアによるトレース手法の適用は困難であると考えられる．

また，外部ハードウェアが CPU の構造に強く依存する，外部ハードウェアで取得可能な情報が固定的で柔軟性に欠けるなどの問題もある．

6.2 ソフトウェアによるトレース手法

ソフトウェアによるトレース手法は，プログラムにトレース用のコードを挿入し，ソフトウェアの実行状態を追跡する手法である．

この方法は簡易に実現可能であるが，実行すべきコード量が増加するため，必ず性能低下を伴う．特にこれらのコードはメモリアクセスを伴うため，現在のアーキテクチャにおいては大きなオーバヘッドとなる．そのため，ソフトウェアによるトレースを適用する場合，ソフトウェアの中で関心のある部分のみを視覚化されている．この手法は，デバッグや教育用途として研究がなされている．

プログラムコードにコメントとして視覚化指示を記述することでソフトウェアを部分的に視覚化

する研究がある [7]．この研究では，トレース処理を高速化するのではなく，トレース範囲を限定することで性能低下を抑えているため，本研究とは異なる．また，運用中トレースを対象としたものには，視覚化対象を実行プロセスの実行状態変化に絞り，分散システムを視覚化する研究がある [8]．この研究では，プロセスがどのようなタイミングで走行・休眠するかを把握可能であるが，資源競合による障害が発生する場合などは，本研究のようなプログラムレベルの実行状態把握が必要である．

6.3 カーネルデバッガ

OS をデバッグする技術として，カーネルデバッガがある．

実際に利用されているカーネルデバッガとして，Linux カーネルをデバッグするためのツール KDB[9] がある．KDB では，あらかじめ Linux カーネルに通知コードを含んだパッチを適用しておいた後，実行時にブレークポイントの設定することで，開発者が確認したいプログラムコード中の特定の行について逐一実行を確認することなどが可能となる．本研究は実際の運用中の実行経過を取得することが目的であり，ブレークポイントによる実現とは目的が異なる．

高速なトレースの実現を試みる研究として FKT[10] がある．この研究におけるトレース対象には関数の呼び出しと呼び出した関数からのリターンが想定されている．カーネルをデバッグする開発者は，FKT の提供するマクロをトレース対象のコードに埋め込む必要がある．これに対し，本研究では条件分岐，ループなどの制御レベルでの差異を明確にするためにさらに細粒度のトレースを目指す．また，FKT ではデバッグが目的であるため，トレース処理は開発者が手作業で挿入するという点でも異なる．

7 結論と今後の課題

本稿では，動的再構成可能な OS の運用を支援するための運用時視覚化機構について提案し，運用時視覚化を実現するための方法について検討した．処理トレースのオーバヘッドについてはシステムを試作し，現状の 32bit CPU 前提のトレースによるオーバヘッドを計測した．また，このオーバヘッドを削減するために，64bit CPU を用いる手法について検討した．

今後は，検討した 64bit CPU 対応への考察をもとに，64bit CPU 専用トレースシステムを実装し，そのオーバヘッドの評価をおこなう予定である．その後，多くのモジュールの相互作用を視覚化する場合の提示手法について検討し，運用時視覚化システムを実現したいと考えている．

参考文献

[1] 小林 良岳, 佐藤 友隆, 唐野 雅樹, 結城 理憲, 前川 守: 彩：コンパイル時に自動生成される Portal をもとに動的再構成可能なオペレーティングシステム， 電子情報通信学会論文誌 VOL.J84-D-I No.6, pp.605-616, Jun 2001.

[2] Object Management Group, Inc.: UML: Unified Modeling Language, http://www.uml.org

[3] IBM Corporation: Project eLiza, http://www-1.ibm.com/servers/autonomic/

[4] 小林良岳, 唐野雅樹, 結城理憲, 紅谷順, 姜亨明, 中山健, 前川守: モジュール差し替え時のプログラム構造変化に対応する動的リンカ， コンピュータシンポジウム, pp.65-72, Nov 2001.

[5] J. Gait: A probe effect in concurrent programs, Software - Practice and Experience, 16(3):225-233, March 1986.

[6] Mohammed El Shobaki: Non-Intrusive Hardware/Software Monitoring for Single- and Multiprocessor Real-Time Systems, 2001

[7] Ohki Mikio,Hosaka Yasuo: A Program Visualization Tool for Program Comprehension, Proc.of 2003 IEEE Symposium on Human Centric Computing Languages and Environments,pp.263-265, 2003

[8] Hideyuki Tokuda, Makoto Kotera, Clifford E. Mercer: A real-time monitor for a distributed real-time operating system, Proc.of 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging, pp.68-77, 1988

[9] Silicon Graphics, Inc.: KDB http://oss.sgi.com/projects/kdb/

[10] Robert D. Russell: FKT: Fast Kernel Tracing, Technical Report TR 00-02, Computer Science Department, University of New Hampshire, Durham, New Hampshire, March 2000.