情報処理学会論文誌　Vol. 52　No. 9　2602–2612 (Sep. 2011)

特徴データベースを用いない効率的な仮想マシンモニタ検出方式の提案

宮本　久仁男[†1]　田　中　英　彦[†1]

完全仮想化方式による仮想マシンモニタは，実ハードウェアを対象にしたオペレーティングシステムおよびアプリケーションを，修正を施すことなく実行させることが可能であるが，仮想マシン上でのソフトウェア動作を行わせたくないという要求もある．仮想マシン環境の検出を行う試みは，これまでも行われているが，それらは仮想マシンの機能上現れるデータや判定のための性能値を集めた特徴データベースを利用することが前提であったり，仮想マシンモニタ側の実装上の工夫で回避できたり，また判定が可能になるまでかなりの時間を要したりするなどの欠点を有する．本論文では，そのような特徴データベースを用いることなく，CPU による命令実行時に現れる性能値の変動をとらえ，アプリケーションプログラムのレベルで利用可能な，効率的な完全仮想化環境の判別方法を提案し，その評価結果について述べる．

Proposal of Effective Detection Method of VMM without Feature Database

Kunio Miyamoto[†1] and Hidehiko Tanaka[†1]

The software that is issued to be run on the real hardware can be run on the virtual machine monitor for full virtualization, and when running software on such a virtual machine monitor, any modification to the software is not needed. But there are some needs that some software must not be run on such an environment. Though many researches for detecting virtual machine environment exist, these ones use some feature databases for detection or are avoidable by some modification of virtual machine monitor or spend pretty much time to detect. In this paper, we propose a light weight and effective detection method of the underlying full-virtualized virtual machine monitor that don't use such feature databases but use the stability of CPU instruction execution performance value, and show the evaluation results.

1. はじめに

現在，仮想マシンモニタを技術的構成要素の 1 つとしたクラウドコンピューティング環境[1)] の利用が広まりつつあり，その結果，仮想マシンモニタを導入した環境でソフトウェアを開発したり稼動させたりすることが増えている．しかしその一方で，性能やセキュリティ上の理由から，このような仮想マシンモニタ上でのソフトウェア動作を行わせたくないという要求もある．特に，実ハードウェアをターゲットに性能設計やセキュリティ設計が行われたシステムにおいて，システムを構成するソフトウェアを安易に仮想マシンモニタ上の環境で稼動させることは望ましいとはいえない．また，商用ソフトウェアには，動作環境が実ハードウェア上か仮想マシンモニタ上かでライセンス体系が異なるものもあるが，その判別は，当該ソフトウェアを使用するユーザ側で行っている．

これまでも仮想マシン環境の検出を行う試みは多く行われているが，それらの試みの多くでは，仮想マシンモニタの実装が見せるリソース名や，特定の手続きを経て取得可能な文字列，未実装命令の挙動など，実装ごとに決まった手がかりを用いる．CPU が備えるハードウェアカウンタを用いた性能計測結果を評価する方法も提案されているが，この方法も事前に計測した CPU ごとの性能値を特徴データベースとして保持しておく必要がある．いずれの方法も，手がかりを格納した特徴データベースを用いるため，手がかりにあたる部分が実ハードウェアと同じである仮想マシンモニタは検出できない．また，これらの方法は，検出結果の妥当性や，検出誤りの評価，そして検出誤りの低減方法までは踏み込んでいない．その他，ネットワーク経由で仮想マシン検知を行う研究もあるが，高確率で検知を行うまでに必要となる取得データ量が多くなり，短時間での検知に向かない．

一方で，今後実装される，もしくは未知の実装である，機能面の手がかりがないような仮想マシンモニタを高確率で検出する方法は，今後仮想マシンモニタの実装が増えていく可能性があるところで重要であり，判定用の特徴データベースを持たずに済む検出方法は，仮想マシンモニタを検出するシステムの開発・運用の手間を減らす観点で重要となる．

本論文では，特徴データベースを用いることのない，ローカルマシン上での仮想マシンモニタ検出方式を提案し，その評価結果について述べる．

†1 情報セキュリティ大学院大学
INSTITUTE of INFORMATION SECURITY

2. 仮想マシンモニタ検出の必要性

本章では，代表的な仮想マシンモニタの実現方式および仮想マシンモニタを検出する必要性を述べる．

2.1 仮想マシンモニタとその実現方式

仮想マシンモニタは，仮想的なハードウェア環境をソフトウェアによって忠実に実現するためのソフトウェアレイヤであり，それによって提供される仮想的なハードウェア環境を仮想マシンと呼ぶ．仮想マシンモニタが実現する方式は，大きく完全仮想化方式と準仮想化方式の 2 つに分けられる．

完全仮想化方式とは，仮想マシンモニタが提供する仮想マシンが，実在するコンピュータハードウェアと機能的に等価である仮想化方式である．このため，仮想マシン上で動作させるオペレーティングシステムをはじめとするソフトウェアを動作させる際に，いっさいの変更を施す必要がない．本方式を採用している仮想マシンモニタには，VMware Inc. が開発・提供している製品群や VirtualBox [2]，オープンソースで開発されている QEMU [3]，Bochs [4]，PearPC [5] をはじめとして，多くのソフトウェアがある．ここであげたもののうち，PearPC 以外はすべて，Intel 製 CPU を搭載する PC 環境を仮想ハードウェアとして提供する完全仮想化方式を採用している．

準仮想化方式とは，仮想マシンモニタが提供する仮想マシンが，実在するコンピュータハードウェアと機能的に差異が生じる仮想化方式である．このため，機能的に差異が生じる部分を仮想マシンモニタ上に載せるソフトウェア側で意識し，必要に応じて改修する必要がある．改修対象は，通常はオペレーティングシステムのカーネルである．本方式を採用した仮想マシンモニタの実装としては，Xen [6]，LilyVM [7] などがあげられる．

2.2 仮想マシンモニタ検出を行う必要性

本来，仮想マシンモニタにより提供される仮想マシン環境は，実ハードウェア環境と同様のものを提供するべきものであるが，プログラムが想定している実行環境が実ハードウェアでない場合，その実行を終了させたり，実行環境に応じた処理を行わせたりすることが必要なケースがある．このような場合，プログラムを作成する側で，動作環境が仮想マシンモニタであるか否かを判定するしくみを用いることで，プログラムの利用者に判断を行わせることなく，プログラム作成者が実行環境に適した処理をさせることが可能となる．

簡単な例として，実行環境によってライセンス体系を変更したいソフトウェアでの利用や，性能上の理由により仮想マシンモニタ上での利用を前提としない，もしくは動作パラメータを動作環境に応じて適切なものに変更するソフトウェアでの利用などがあげられる．

仮想マシンモニタの実装によっては，機能上の特徴をチェックすることで，実行環境を容易に判別できることもあるが，その判別は，仮想マシンモニタの実装に依存した方法であり，「実行環境が特定の仮想マシンモニタではない」ということを知ることはできても，他の仮想マシンモニタ上で動作している可能性までは否定できない．

また，ネットワーク経由の仮想マシンモニタ検出[8] は，ローカル検知にも適用できる可能性があるが，多数のパケットを収集し，分析する必要があるために多くの時間を要する．また，ネットワーク経由での検出を行えない環境にコンピュータ類が配置されることも考えられるが，仮想マシンモニタの検出は，このような場合にも検出精度などを明らかにしたうえで検出を行える必要がある．

3. 先行研究とその課題

本章では，先行して取り組まれている，仮想マシンモニタ検出に利用可能な技術について，その概要と実利用上の課題を述べる．

3.1 機能的互換性と透過性

プログラムの実行環境が仮想化環境であることを知るための手法は，Garfinkel らによる論文[9] において手がかりを示されている．論文 9) は，「機能的互換性は，必ずしも透過性に結び付くものではない」という仮想化技術および実装の限界について，その概要を述べたものであるが，あくまで概要を示したにとどまり，具体的にどのような方法で仮想化環境の存在を知るかということを述べたものではない．

3.2 仮想マシンモニタの実装に着目した検出法

この検出法は，最も明確な機能的差異を用いた方法である．たとえばコンピュータが搭載するハードウェア資源の名称に “VMware” という文字列という文字列が含まれている場合には，仮想マシンは VMware 上で動作していると見なす[10),11)]．ほかには，マルウェアの 1 つである Phatbot[12] などにおける「VMware 検出」の手法がある．これは，具体的には特定のアドレスに格納された値によって，VMware 製品のバージョンなどを知ることができるというものである．VMware が搭載する NIC については，NIC の MAC アドレスについてベンダコードが VMware のものであることを用いて判断可能となることもある．また，RedPill [13] のように，仮想マシンモニタによって特定命令の挙動が異なることを手がかりとするものもある．

しかし，実装に着目した仮想マシンモニタ検出は，仮想マシンモニタの実装によって回避

される可能性がある．そのほか，実装によって，特徴が現れる資源が異なる可能性があるため，実装が増えるごとに検出可能な資源の特徴を，判定に用いる特徴データベースに追加する必要がある．

3.3　TLB や CPU キャッシュミスヒットに着目した検出法

複数の仮想マシンで CPU1 つの TLB を共有するケースがあるが，そのような場合に，TLB のミスヒット発生の頻度が上がることに着目した検出を行うことが考えられる．これは，Dongarra らの研究[14] により，CPU の種類ごとに TLB のミスヒットに関する性能曲線が異なるということが明らかになっているが，これを仮想マシンモニタの検出に利用することが可能である．複数の仮想マシンで CPU の TLB を共有すると，性能曲線が CPU 固有のものと異なってくるが，これを検出することで，仮想マシンモニタの存在を検出することが可能である．この際には，PAPI[15] というソフトウェアモジュールを使用することが可能である．また，Li らの報告[16] のように，CPU の種類ごとに CPU キャッシュのミスヒットに関する性能曲線が異なることを仮想マシンモニタの検出に利用することもできる．複数の仮想マシンで CPU1 つのキャッシュを共有するケースでも，TLB と同じく CPU 固有のものと異なってくるため，TLB を利用する際と同様の方法で検出が可能で，この際にも，PAPI を使用することができる．

しかし，PAPI はカーネルパッチなどを含むソフトウェアモジュールであり，Linux や Windows のような実行 OS の環境に強く依存した構造になっている．また，一部カーネル空間での実装を用いるため，検出のための環境を作成するのが難しい．

また，この方法を用いる場合，実 CPU における TLB のミスヒットを誘発するまでのメモリアクセスパターンや頻度を，検出の根拠とする必要がある．このパターンや頻度は，CPU の種類によって TLB や CPU キャッシュの容量が決まることから，CPU ごとにパターンのサンプリングを行う必要がある．このことは，新たな CPU 製品が出てくるごとに性能測定を行って得られたデータを，仮想マシン検出のためのデータとして特徴データベースに追加することを意味しており，決してメンテナンス性が高いとはいえない．

PAPI をはじめとする，カーネル空間でのコード実行をともなった性能計測を行わない場合には，CPU キャッシュの内容や TLB エントリの消費をユーザ空間から操作することになる．しかし，他のプロセスの影響を受けずに CPU キャッシュや TLB エントリの消費を行わせることは難しく，このような手法を簡便に検出ツールとして利用することは難しい．

3.4　仮想マシンと実ハードウェアの性能的な差異に着目した検出法

仮想マシンと実ハードウェアの性能差を検出するための方法として，IA-32[17] アーキテクチャや Intel64[18] アーキテクチャで定義されている rdtsc（Read Time Stamp Counter）命令を用いることにより得られるタイムスタンプカウンタ（TSC）の値を用いた命令実行時間の計測を用いることが知られている．

Ferrie の報告[19] でも，TSC のようなローカルなタイムソースを用いることが知られている旨をあげている．この検出法は，TSC がローカルなタイムソースとして正しく動作していることを前提にしているが，具体的にどのようなチェックを行うことでどのような精度の検出を行えるかまでは述べていない．また，TSC 自体の動作が正しいことを検証しないと，そこで計測された値の正しさを評価できない．

Raffetseder らの報告[20] では，IA-32 アーキテクチャを模する仮想マシンを実現する仮想マシンモニタ上での動作について，CPU 内の Machine Specific Register（MSR）へのアクセスを行い，その実行クロック数を nop 実行時の実行クロック数の比率で表すことで，実際の性能差から仮想マシンモニタの存在を検出する方法を提案しているが，この方法を用いた場合は，その判定用の比率を具体的な値として持つ必要がある．

特許[21] については，命令にかかるクロック数およびその平均値を手がかりにしており，事前に取得した判定のための特徴データベース中に保持している値と比較している．このため，新しい CPU が出るごとに特徴データベースの値を追加する必要がある．

その他，仮想マシンモニタの実装上確認されている TSC の挙動をチェックする方法もあげられている[22]．これは，特定の実装において，特定の命令実行クロック数が特定の値になることに着目した検出法である．

3.5　ネットワーク実装の挙動と仮想マシンモニタの動作に着目した検出法

嶋村らの報告[8] では，ICMP パケットに含まれる ICMP Time Stamp および IP ヘッダに含まれる Time Stamp を観測し，それぞれの Time Stamp がどのようにずれるのかをチェックしている．特定アプリケーションに依存せず，仮想マシンのディスパッチなどにのみ依存するため信頼性は高く，リモート検出の提案ではあるが，ローカル検出にも同手法を適用できる可能性があることを報告中で述べている．

一方でこの方法は，ICMP のレスポンスパケットに含まれる情報を用いることを前提としている．このため，ネットワークスタックを使えない場合には仮想マシンモニタの存在を検出できない．また，Time Stamp のずれが発生したことを検出するために，多くの ICMP Time Stamp Request に対応した Response パケットを収拾する必要がある．論文 8) の実験では，50,000,000 パケットを 12 時間かけて収集しており，1 回のパケット送出に 1 ms かけている．その結果として，90%の確度で仮想化環境の検出に必要な所要パケット数を，べ

ストケースで 150,000 パケット，ワーストケースで 2,200,000 パケットと見積もっているが，1 ms に 1 回の送出頻度であることを考えると，150〜2,200 秒程度かかる．パケット収集のために必要な時間を考えると，短時間での検出が可能とはいえない．

また，ICMP Time Stamp は RFC792[23] で規定されているが，RFC1122[24] によると，ICMP Timestamp Request と ICMP Timestamp Reply は補助的に使われる程度である．実際に Windows NT では，RFC1122 の規定に則ってタイムスタンプを通知しない[25]．このことは，使われる OS によって，検出できないことがあるということを意味する．

3.6　既存の方式における課題まとめ

本章各節において，既存の仮想マシンモニタ検出方法とその課題について述べた．以下に，課題の部分をまとめる．

- 特定の実装に依存するため，実装上の工夫で回避可能．
- CPU の種類によって決まる値や計測の結果得られる性能パターンに依存するため，CPU の種類が増えた場合には特徴データベースへの値の追加が必要．
- デバイスドライバとして実装されたり，OS カーネルに手が入ったりするため，OS の構造に強く依存するうえに，OS の安定性が損なわれる危険性がある．また，商用 OS をターゲットにした場合，OS そのもののサポートが受けられなくなる懸念がある．
- TSC を用いた方式では，具体的な仮想マシンモニタ検出プログラムの実装と，検出精度の評価結果が明確でない．
- ICMP パケットを用いた検出は，検出を行うまでに時間がかかる．
- OS によっては，ICMP パケットを用いた検出で使用する ICMP Timestamp が実装されておらず，仮想マシン上での動作を検出可能な OS が限られる．

4.　仮想マシンモニタの検出方式の提案

本章では，本論文で提案する仮想マシンモニタ検出方法について述べる．

具体的には，rdtsc 命令の実行クロック数を複数取得し，その比較結果に基づいた仮想マシンモニタ検出法について，その概要と実装を述べる．また，仮想マシンモニタの実現方式を 2.1 節で述べているが，本論文で検出対象とする仮想マシンモニタは，利用上の実績や，仮想マシンモニタとしての利用のしやすさを考慮して，完全仮想化方式によるものとする．準仮想化方式による仮想マシンモニタの検出は，今後の課題とする．

4.1　提案方式概要

提案する方式の概要は，以下のとおりである．また，この判定方式における誤検出率の議論は，5.4 節で行う．

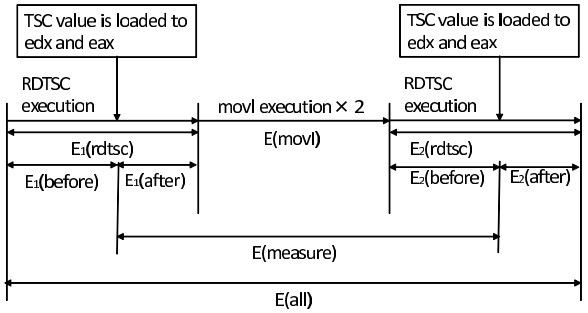

図 1　仮想マシンモニタを検出するための TSC から取得される値の評価概要
Fig. 1　Overview of evaluating TSC value for detecting virtual machine monitor.

(1)　特定の命令シーケンスを実行し，実行クロック数を測定する

このときに実行する命令シーケンスについては，命令シーケンス全体における rdtsc 命令の実行クロック数の割合ができるだけ大きくなるような命令シーケンスとする．このため，rdtsc を 2 回実行する間には，測定値を保持する為の最小処理として，レジスタ 2 つ分の退避命令を実行するのみにとどめる．

(2)　(1) で定義した処理を複数回実行し，その結果が同一か否かを確認する

具体的には以下のように判定する．

- すべて同じ実行クロック数の場合は実ハードウェア上での動作である．
- 1 つでも異なる実行クロック数が得られる場合は仮想マシンモニタ上での動作である．

この提案方式の処理の流れを，**図 2** に示す．

図 1 は，この測定状況を表している．ここで，n 回目の rdtsc 命令の実行クロック数である $En(rdtsc)$ は，TSC の値を取得するまでの命令実行クロック数を $En(before)$，TSC の値を取得した後の命令実行クロック数を $En(after)$ と分けた場合，以下のように表現できる．

$$En(rdtsc) = En(before) + En(after) \quad (1)$$

1 回目の rdtsc 命令で TSC の値を取得してから，2 回目の rdtsc 命令で TSC の値を取得するまでの命令実行クロック数 $E(measure)$ は，movl を 2 回実行するクロック数を $E(movl)$ とした場合，$E_1(after)$+$E(movl)$+$E2(before)$ となる．しかし，式 (1) に示す

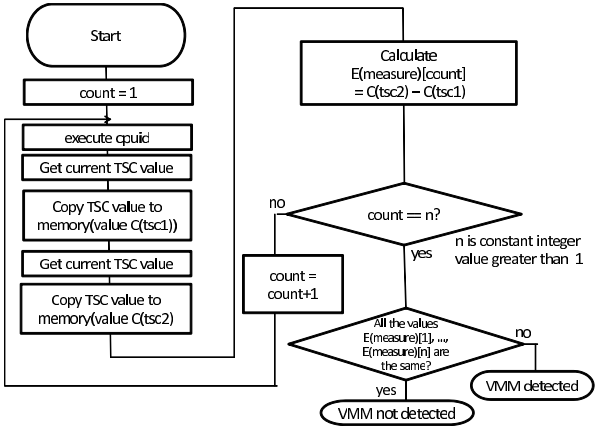

図 2　仮想マシンモニタを検出するための処理概要
Fig. 2　Overview of virtual machine monitor detecting process.

ように，$En(rdtsc) = En(after)+En(before)$ であることと，rdtsc 命令の n 回目の実行時と $n+1$ 回目の実行時が近接する場合，$En(rdtsc)$ と $En+1(rdtsc)$ をほぼ等しいと見なすと，2 回の TSC 値計測の差を $E(measure)$ とすると，$E(measure)$ は式 (2) に示すように表現できる．式中 $E(rdtsc)$ は，実行環境上の rdtsc 命令実行クロック数である．

$$E(measure) = E_1(after) + E(movl) + E_2(before) \simeq E(rdtsc) + E(movl) \quad (2)$$

実ハードウェア上では，命令実行のタイミングによらず，$En(rdtsc)$=$En+1(rdtsc)$ となるが，仮想マシン上では，仮想マシンモニタによる当該命令実行時の捕捉と処理が行われ，そうならないことがある．このため，$E(measure)$ を複数計測してその結果を比較することで，プログラムの動作環境が実ハードウェアであるか仮想マシン上であるかを判定することができる．

図 1 で実行される命令数は 4 で，rdtsc 命令 2 つ+ mov 命令 2 つである．また，1 度目の rdtsc 命令実行前には，アウトオブオーダ命令実行を行わせない目的で，cpuid 命令を実行する．2 つの rdtsc 命令ではさむ処理の命令数は，前述のように最小命令数であるが，命令数を最小限にすることで，プロセス切替えや，割込みによる処理中断が発生する確率を最小限に抑えることも目的としている．また，movl 命令によるメモリへのデータ書き込みは，CPU キャッシュに書き込まれた時点で処理を完了するため，実メモリアクセスにともなう処理の遅延は発生せず，rdtsc による書き換えが発生しないレジスタへのデータ退避を行えば，実メモリアクセスにともなう処理の遅延は考慮しなくてもよい．この一連の処理をカーネル内で実装すれば，プロセス切替えも発生しないし割込み禁止区間を設けることも可能であるが，ユーザプロセスとしての実装ではそれは困難である．一方，カーネル内で実現する場合には，どうしてもデバイスドライバもしくはカーネル内処理として当該命令を記載しなければならず，本論文で目指す利便性の向上を満たすことができない．

4.2　適 用 範 囲

本方式は，4.1 節 (1) で示す処理と，4.1 節 (2) で示す結果判定からなるが，事前計測を行ったクロック数などを収めた特徴データベースを必要としない．このことは，本方式を実システムに適用した場合に，システムの保守性を向上させるのに寄与する．

また，ここで述べたような単純な処理であれば，OS 上での他のアプリケーションプログラム実行ごとに 1 度実行するようにしても，性能劣化は微小である．処理が単純であるため，OS 起動前に動作するブートプログラムの一部として構成することも可能であろう．

一方，IA-32 [17] アーキテクチャや Intel64 [18] アーキテクチャに準拠した CPU の中には，仮想化支援機能である IntelVT [26] や AMD-V [27] が実装されているものがある．本論文で提案する方式は，仮想マシンモニタの実装上避けられない制約により，すべてをソフトウェアで処理される命令実行にかかるクロック数に変動があることを根拠にしたものである．しかし，仮想化支援機能を用いている仮想マシンモニタは，その部分をハードウェアが支援することで，仮想マシンモニタの処理を効率化し，仮想マシン上からは見かけ上そのクロック数の変動を見えないようにしていることがある．このため，本論文で提案する方法だけでは，CPU による仮想化支援機能を用いた仮想マシンモニタの検出はできない．

4.3　実 装 概 要

4.1 節で述べた方式は，IA-32 で定義された範囲のアセンブリ言語で記述可能で，特定 OS の特定の機能に依存しない．また，命令シーケンスの実行速度計測のために，CPU 外にある HPET [28] のようなタイマ機能を用いることもない．このため，4.1 節で述べた条件を満たす命令シーケンスを記述できるプログラミング言語を用いることで，ポータビリティも確保可能である．本論文では，方式の検証を行うために，インラインアセンブラを記述できる言語処理系を選択して，4.1 節で述べた命令シーケンスおよび判定ロジックを実装した．

4.4　実装上の留意点

本方式は，ユーザが実行可能な命令シーケンスを発行し，その実行クロック数の一致/不一致を判定する，というものである．これは，判別対象となるシステムの負荷がばらついた状態であっても，以下の 2 つの理由により，信頼性の高い結果を求めることが可能である．

- 負荷が高い場合であっても，アセンブリ言語にして数ステップ程度のプログラム実行中にコンテキストスイッチが発生する可能性は低い．
- アセンブリ言語にして数ステップ程度の実行であれば，その最中にシステムからの割込みによる実行遅延が発生する確率を下げられる．

前者についてはプログラム構造を工夫することで，さらにその確率を下げることができる．たとえば，Linux において，ディスク I/O を行うシステムコールの直後に，提案する方式による判定プログラムを実行する，ということが考えられる．ディスク I/O を行うシステムコールは，通常はシステムコール発行後は，I/O を行うために，CPU を有効利用するために自身の実行権を手放し，I/O 完了待ちに移行する．そして，I/O が完了した旨の通知を I/O コントローラから受け取り，実行可能状態に再度移行して，スケジューラによる実行権の割当てを待つ．実行権が割り当てられると，同時にタイムスライスも割り当てられるが，Linux において，I/O 処理実行などのためにいったん実行権を手放したプログラムは，I/O 処理が完了するなどして再度実行可能状態になった際に，通常 10 ミリ秒程度のタイムスライス値の割当てを期待できる．したがって，判定のための根拠を取得するための処理は，I/O 完了後に割り当てられる時間内で十分実行可能である．ユーザプログラム上では，たとえば read() システムコール直後に，本論文で提案する判定のための命令シーケンスを実行し，結果を評価することが考えられる．

4.5 実装詳細

図 2 に，実装した判定プログラムの処理概要を示す．この処理は，Intel の Pentium 以降の CPU での実行を想定している．TSC 値の取得は，アセンブリ言語の rdtsc 命令を発行することで実現可能であり，この命令は，Windows や Linux など普通に使われる OS ではすべての特権レベルで実行可能である．なお，CPU による実行クロック数取得は，Sun の Ultra SPARC[29),30)]，HP の PA-RISC[31)] をはじめとして，多くの CPU で可能であり，本論文で提案する方式が他の CPU においても有効に機能する可能性がある．

命令実行の遂次性を担保するためには，計測処理の間に，CPU によるアウトオブオーダ命令実行を行わないようにしたり，命令そのものが遂次実行されるように留意したりする必要があるが，cpuid 命令の実行を計測前に行うことで，cpuid 命令実行時点でのアウトオブオーダ命令実行は無効になり，さらに 2 回の rdtsc 命令とその間の movl 命令については，使用するレジスタに競合が発生するため，並列に実行はできない．したがって，この区間での命令実行の逐次性は保たれる．

5. 提案方式の評価

本章では，本論文での提案方式に基づいた仮想マシンモニタ検出を行い，その際に得られた結果をもとにその有用性と適用箇所に関する議論を行う．提案方式の核となるのは，仮想環境における rdtsc 命令の実行クロック数が，実ハードウェアでは一定であり，仮想マシン上では変動するという事実である．この事実に基づいた検出について，どの程度の精度が得られるのかを計測評価する．

5.1 評価方法

評価環境を，**表 1** に示す．

提案方式の評価については，実行クロック数の推移を確認する予備評価および，実際の検出ソフトウェア評価の 2 段階で行う．

5.2 予備評価

$E(measure)$ を取得するための評価プログラムを作成し，その実行結果を評価した．本プログラムのコンパイル時に最適化を指定した場合，2 度目の rdtsc 命令実行を行わない実行コードが生成されるため，コンパイル時にはいっさいの最適化を行わない．こうして作成した実行命令は以下に示すとおりである．

```
80483e8:       0f a2                   cpuid
80483ea:       0f 31                   rdtsc
80483ec:       89 d1                   mov    %edx,%ecx
80483ee:       89 c7                   mov    %eax,%edi
80483f0:       0f 31                   rdtsc
```

実行クロック数の評価を行うために，実ハードウェアおよび仮想マシン上で，$E(measure)$ の測定を行った．その結果を**図 3** および**図 4** に示す．

図 3 の結果より，わずかではあるが，実ハードウェアにおいて誤検出発生が出てくる可能

表 1　評価環境 1
Table 1　Evaluation environment.

CPU	PentiumM
クロック	1.7 GHz
仮想マシンモニタ	VirtualBox 1.6.6
ホスト OS	Debian GNU/Linux 5.0
ゲスト OS	Debian GNU/Linux 5.0

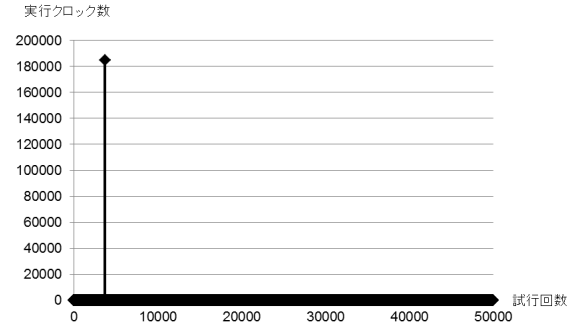

図 3　実ハードウェア上の $E(measure)$ 測定結果
Fig. 3　Result of evaluating $E(measure)$ on the real hardware.

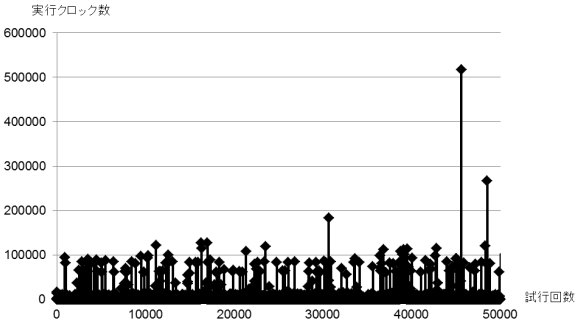

図 4　仮想マシン上の $E(measure)$ 測定結果
Fig. 4　Result of evaluating $E(measure)$ on the virtual machine.

性がある．取得された値を精査したところ，50,000 回実行時において，1 回のみほかと大きく異なる結果が得られた．

また，図 4 は，大きく値がぶれているように見えるが，これもまた取得された値を精査したところ，50,000 回実行時において，複数回連続して同じ値が取得されることがあった．50,000 回の値取得において，2 回連続が 3,148 回，3 回連続が 254 回，4 回連続が 18 回，5 回連続が 1 回となっており，それぞれ 6.296%，0.508%，0.036%，0.002%となる．誤判定率のプロット結果を，**図 5** に示す．

- 連続して 2 回同じ値が得られることを実ハードウェアであることの根拠とした場合には，大きな値をとった 1 回の測定の前後で取得値が不一致となるので，2/50000*100 = 0.004%程度の誤検出発生が，連続して 3 回同じ値が得られることを実ハードウェアであることの根拠とした場合には，3/50000*100 = 0.006%程度，連続して 4 回同じ値が得られることを実ハードウェアの根拠とした場合には，4/50000*100 = 0.008%程度，連続して 5 回同じ値が得られることを実ハードウェアの根拠とした場合には，5/50000*100 = 0.01%程度，実ハードウェアを仮想マシンというように誤検出をすることが想定される．
- 上記と同じ条件をあてはめた場合，2 回連続した場合は 6.296%，3 回連続した場合の誤検出率は 0.508%，4 回連続した場合の誤検出率は 0.036%，5 回連続した場合の誤検出率は 0.002%程度，仮想マシンを実ハードウェアであると誤検出することが想定される．

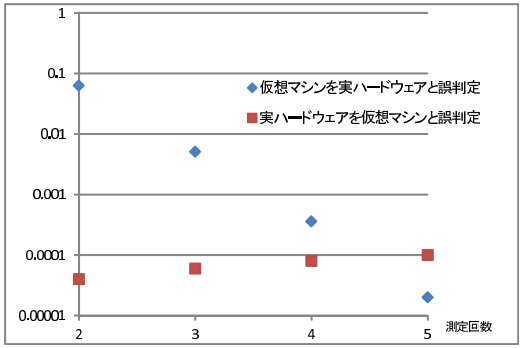

図 5　測定回数と誤判定率の相関関係
Fig. 5　Correlation of measurement count and musjudgement rate.

5.3　実評価と結果

予備評価において，実行クロック数を 2 回〜5 回取得し，その結果の吟味を行った場合の，誤検出率の推移を確認できたので，それに従って検出プログラムを作成，評価した．検出プログラムにおける実行クロック数の取得回数は，2〜6 回と推移させている．この検出プログラムもまた，5.2 節のときと同様に，コンパイル時の最適化指定を行わない．このように作成したプログラムを，2 通りの動作環境と 2 通りの負荷環境を組み合わせた 4 つのパターンで 50,000 回実行させ，判定結果を得た．誤判定数を，**表 2** に示す．通常負荷時は，Linux 環境が起動している状態を指しており，平常時の CPU 使用率は常時 1%未満である．高負荷時は，ユーザプロセスにおいて CPU のみを空費するプロセスを 200 個起動しており，CPU 使用率が常時 100%近い状態になっている．

上記の結果より，高負荷時のほうが，誤判定率が 1/3 程度になる傾向が見られる．高負

表 2　50,000 回判定時の誤判定数
Table 2　Misjudgement count of 50,000 times evaluation.

実行クロック数取得回数 n	2	3	4	5	6
実ハードウェアの通常負荷時	0	3	3	9	3
実ハードウェアの高負荷時	0	0	3	6	4
仮想マシンの通常負荷時	651	370	245	161	114
仮想マシンの高負荷時	223	126	98	56	47

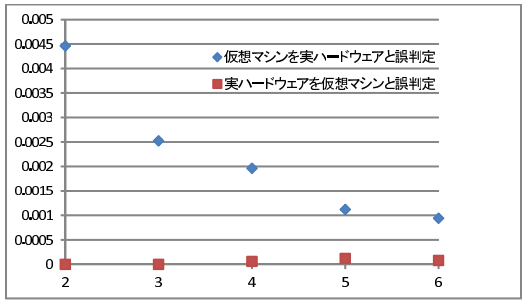

図 6　仮想マシンモニタ検出プログラムにおけるクロック数測定回数と誤判定率の相関関係
Fig. 6　Correlation of measurement count and musjudgement rate in the virtual machine monitor detection program.

荷を実現するために実行させたプログラムは，CPU のみ消費する．このため，プロセスに実行権が割り当てられた際には，タイムスライスをすべて使いきるまで走行し，実行待ちキューの最後尾につながれることが多い．通常，UNIX をはじめとするタイムシェアリング型のプロセススケジューリングを行う OS では，タイムスライスを使いきる前に実行権を手放すプログラムには，高い優先度が割り当てられるが，タイムスライスをすべて使いきったプログラムには低い優先度が割り当てられる．今回の場合は，CPU を空費する多くのプログラムが低い優先度を割り当てられる．このため，負荷が低い通常時よりも測定プログラム実行時に割り当てられるタイムスライス値が保証されやすい状態である．

CPU 使用率が高い場合であっても，検出プログラムそのものが軽量であることもあり，この傾向自体は大きく変わることはなく，通常負荷における誤判定率を低減させることで，高負荷時の誤判定率もそれにつれてさらに低減される．

表 2 に示した結果より求めた高負荷時の誤検出率の推移を，**図 6** に示す．

図 5 と比較すると，検出プログラムにおいて，仮想マシンを実マシンと誤検出する率が全般的に低減されていることが分かる．これは，予備評価の結果は単一のプロセスにおいて 1 回だけ実行されており，プロセス生成のタイミングとプロセス実行時の状態が類似のものになることが多いことと，他のプロセスがいないため，測定プロセスが走行するタイミングが直前のものと同一になりやすいためと考えられる．一方，検出プログラムのほうは，1 つのプロセス内で実行クロック数を複数回取得しているが，仮想マシン上での動作に置き換えた場合，仮想マシンモニタ内の処理に移行するタイミングが単一の処理クロック数取得の場合よりも多い．単一プロセス内における仮想マシンモニタへの処理移行のタイミングが増加するため，仮想マシンモニタが介在する場合にはその影響をより受けやすくなり，検出をしやすくなるためと考えられる．なお，誤検出率の低減の傾向は，予備調査時と変わりないため，予備調査において確認できる誤検出率が十分低ければ，本論文で述べる検出方式を実装した場合はそれよりも低くなるため，実効上は問題ないと考える．

5.4　求める誤判定率と判定方式の選択

図 5 で述べたように，最適な検出法は，どのようなタイプの誤判定を重視するかで異なる．

(1)　実ハードウェアを仮想マシンと判定する率を小さくしたい： 実行クロック数の取得回数を 2 にする．この場合，仮想マシンを実ハードウェアと判定する率は，651/50000 = 1.302%となる．

(2)　仮想マシンを実ハードウェアと判定する率を小さくしたい： 実行クロック数の取得回数を可能な限り大きくする．実行クロック数の取得回数を 6 にした場合，仮想マシンを実ハードウェアと判定する率は，114/50000 = 0.228%となり，実ハードウェアを仮想マシンと判定する率は 3/50000 = 0.006%となる．

(3)　それ以外： 実行クロック数の取得回数を 3〜4 回程度にする．実行クロック数の取得回数を 3 にした場合，仮想マシンを実ハードウェアと判定する率は，370/50000 = 0.74%となり，実ハードウェアを仮想マシンと判定する率は，3/50000 = 0.006%となる．実行クロック数の取得回数を 4 にした場合，仮想マシンを実ハードウェアと判定する率は，245/50000 = 0.49%となり，実ハードウェアを仮想マシンと判定する率は 3/50000 = 0.006%となる．

5.5　提案方式の他の実装への適用

本論文での提案方式を，他の仮想マシンモニタ実装 2 つに対して適用した．評価環境を，**表 3** および **表 4** に示す．

5.3 節で実施した評価を通常負荷時の評価環境 2 と評価環境 3 に対して行い，誤判定数を取得した．その結果を**表 5** に示す．

表 3　評価環境 2
Table 3　Evaluation environment 2.

CPU	PentiumM
クロック	1.7 GHz
仮想マシンモニタ	VMware Workstation 5.5.9
ホスト OS	Windows XP SP3
ゲスト OS	Debian GNU/Linux 5.0

表 4　評価環境 3
Table 4　Evaluation environment 3.

CPU	PentiumM
クロック	1.7 GHz
仮想マシンモニタ	QEMU 0.9.1
ホスト OS	Debian GNU/Linux 5.0
ゲスト OS	Debian GNU/Linux 5.0

表 5　評価環境 2 および評価環境 3 における 50,000 回判定時の誤判定数
Table 5　Misjudgement count of 50,000 times evaluation.

実行クロック数取得回数 n	2	3	4	5	6
評価環境 2 の通常負荷時	0	0	0	0	1
評価環境 3 の通常負荷時	56	13	9	21	18

表 5 に示すとおり，最も誤判定数が大きい場合でも 50,000 回中 56 回となっており，5.3 節での結果と比較して，誤判定率が低い傾向にあることが分かる．

6. 議　　論

本章では，本論文で提案している方式および評価結果，そしてこれまでの方式における課題と本方式での対応について，議論を行う．

6.1 提案方式の利点

本論文における提案方式は，これまで提案されてきている方式と比較して，仮想マシンモニタの機能上の特徴がなくても検出が可能である点と，事前計測結果を格納した特徴データベースを用いる必要がない点，そしてユーザプログラムと同じ権限で動作させられるという点において，他の方式より優れている．

概要と実装についてはすでに述べたとおりだが，ユーザプログラムと同じ権限で動作する以上はユーザプログラムと同じ理由でプログラム切替えが発生しうる．この部分は，提案方式を OS カーネル内で実装することで解決可能だが，そのようにした場合には，方式の簡便性が失われる．求める誤検出率とのバランスで方式を選択することになる．

本論文における提案方式の概要と動作については，単純な命令実行とその実行クロック数の評価方法に根幹があるため，CPU の実装アーキテクチャに大きな違いがなければ，誤検出率を低く抑えることが可能であり，表 2 および表 5 に示す結果で，仮想マシンモニタ上の動作を実ハードウェア上での動作と誤判定する率を低く抑えられることを確認している．

6.2 rdtsc 命令の実行クロック数変動に着目した検出方式の妥当性

仮想マシンモニタの検出のために，特定命令の性能を測定するために rdtsc 命令で取得された TSC の値を性能値の測定に用いる方法は，すでに Ferrie の報告[19] でも概要は紹介されている．しかし，この中で紹介されている方法は，TSC を信頼できる Internal Timer として用いるという方法であり，rdtsc 命令の実行結果を信頼できるという前提に立っている．しかし，TSC を信頼できる Internal Timer と見なせないケースがある．たとえば，VirtualBox のソースコードの 1 つである TMAllCpu.cpp に，以下の記述がある．

```
uint64_t u64Now = tmCpuTickGetRawVirtual(pVM, false \
/* don't check for pending timers */)
              - pVM->tm.s.u64TSCOffset;
```

`pVM->tm.s.u64TSCOffset` は，稼動中の VM が起動した時点で取得した TSC の値を保持する領域で，したがってこの部分は，仮想マシンが上位のプログラムに提供している仮想 CPU における TSC の値を算出するためのコードである．

実際には，上記の処理に至るまでに，さらに多くの判定処理があるため，実装上は TMAllCpu.cpp で記述されている以上に複雑な実行経路を通る．

仮想マシンモニタを経由した場合，このような差異に加え，仮想マシンのスケジューリングや，仮想マシンモニタ以外の要因による処理遅延に起因して rdtsc 命令の実行結果が変動することも考えられる．rdtsc 命令によって得られる TSC の値を命令実行クロック数の測定に利用した場合，得られる命令実行クロック数は，rdtsc 命令の実行クロック数も反映したものになるが，rdtsc 命令の実行クロック数が仮想マシンモニタの介在に起因した変動要素を含む場合，得られた命令実行クロック数は信頼できないものになる．このため，rdtsc 命令そのものの処理クロック数が，安定しているかそうでないかを評価することは重要であり，rdtsc 命令の実行クロック数の安定性を評価することで，本来ハードウェアが存在すると考えられる位置に仮想マシンモニタが存在する場合には，それを検出できる可能性が

ある．

このことから，rdtsc 命令そのものの実行クロック数変動に着目した検出方式は，仮想マシンモニタ検出のための手段の 1 つとして有効であるといえる．

6.3 評価結果

提案方式の評価結果は，嶋村らの報告[8] であげられているものと比較して，検出までに要する時間と精度の両者で良好な結果であるといえよう．嶋村らの報告は，あくまでリモートコンピュータの仮想マシンモニタ検出に主眼が置かれており，ローカルコンピュータにおける仮想マシンモニタ検出にも利用可能であるものの，検出までの時間がかかるうえに精度が悪い．

一方で，表 2 で示すように，実行クロック数の評価回数を増やした場合，実ハードウェアを仮想マシンモニタであると誤検出する可能性がある．仮想マシンモニタの検出システムを構成するうえでは，機能面の特徴がある仮想マシンモニタを確実に検出できる必要もあろう．機能面の特徴を持つ既存の仮想マシンモニタおよび，機能面の特徴を隠ぺいしている仮想マシンモニタの両者を検出対象とする場合には，本論文で提案する方式と既存の検出方式を併存させる方法が考えられる．このようにすることで，より高精度の仮想マシンモニタ検出が行えるようになると考える．

7. 結　　論

本論文では，特定の仮想マシンモニタ実装に関する情報を含めた特徴データベースを使わなくても，仮想マシンモニタを検出できる検出方式を提案し，その誤検出率の許容率から適切な実行クロック数の計測回数を決定して適用が可能であることを示した．

すなわち，特定の命令シーケンスを実行する際の実行クロック数が同一か否かによる仮想マシンモニタを判定する検出方法は，有効である．ただし，その結果には一定の検出誤りがある．仮想マシンを実ハードウェアであると判定するタイプの誤検出率は，1%未満に収めることが可能であり，逆の誤検出率も 1%未満に収めることが可能であることを示した．

3.6 節で述べた課題の具体的な解決状況を以下に示す．

- 事前計測の結果得られた性能値や，仮想マシンモニタの実装上作り込まれている機能上の特徴を含む特徴データベースを用いない．
- OS 上で動作するプログラムから発行可能な命令のみを用いているため，検出プログラムをデバイスドライバ化したり，OS カーネルを修正したりする必要がなく，OS の安定性を損なわない．
- CPU から得られる性能値が均一か否かを判定根拠にしているため，特定の仮想マシンモニタ実装や OS の種類に依存しない．
- 特定の組合せの命令実行を少ない回数行うため，検出に要する時間が短い．

本論文で示した方式は，仮想マシンモニタの実装をともなう特徴の偽装を行っても，検出回避は困難であり，OS より上位のユーザプログラムで，プログラム作成者の望むタイミングで，プログラムの実行権限に関係なく仮想マシンの検出を行えるという利便性を有している．また，特定の性能値や機能上の特徴を集めた特徴データベースを持たないため，本方式を実システムに適用した場合でも，特徴データベースの保守に関わる手間がない．

また，誤検出の許容ポリシ決定とあわせて，既存の検出方式の中から利用可能なものを組み合わせれば，さらに簡便で実用性に富む検出システムを構成できよう．

本論文で提案する方式は，CPU による仮想化支援機能を用いた仮想マシンモニタや，準仮想化方式による仮想マシンモニタは検出対象外としたが，今後はさらに研究を進め，このような仮想マシンモニタも，CPU アーキテクチャや仮想マシンモニタのアーキテクチャなどから想定可能な最少の手がかりをもとに，簡便に検出できる方式の確立を目指す．

参 考 文 献

1) Mell, P. and Grance, T.: The NIST Definition of Cloud Computing (2009).
2) Watson, J.: VirtualBox: Bits and bytes masquerading as machines, *Linux Journal*, Vol.2008, No.166, p.1 (2008).
3) Bellard, F.: QEMU, a Fast and Portable Dynamic Translator (2005).
4) Lawton, K.P.: Bochs: A Portable PC Emulator for Unix/X, *Linux J.*, p.7 (1996).
5) Biallas, S.: PearPC - PowerPC Architecture Emulator, available from ⟨http://pearpc.sourceforge.net/⟩.
6) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization (2003).
7) 榮樂英樹：準仮想化技術を用いた仮想計算機に関する研究，修士論文，筑波大学 (2007).
8) 嶋村　誠，河野健二：ネットワークタイムスタンプによるリモート仮想マシンモニタ検出，情報処理学会論文誌，Vol.50, No.8, pp.1870–1882 (2009).
9) Garfinkel, T., Adams, K., Warfield, A. and Franklin, J.: Compatibility is Not Transparency: VMM Detection Myths and Realities, *11th Workshop on Hot Topics in Operating Systems* (2007).
10) Liske, T.: imvirt – I'm virtualized?, available from ⟨http://micky.ibh.net/liske/imvirt.html⟩.
11) Jones, R.: virt-what – detect if we are running in a virtual machine, available from

⟨http://people.redhat.com/rjones/virt-what/⟩.
12) Phatbot, available from ⟨http://www.secureworks.com/research/threats/phatbot/⟩.
13) Rutkowska, J.: Red Pill... or how to detect VMM using (almost) one CPU instruction (2004), available from ⟨http://invisiblethings.org/papers/redpill.html⟩.
14) Dongarra, J., Moore, S., Mucci, P., Seymour, K. and You, H.: Accurate Cache and TLB Characterization Using Hardware Counters, *ICCS 2004* (2004).
15) Dongarra, J., Jagode, H., Moore, S., Mucci, P., Ralph, J., Terpstra, D. and Weaver, V.: Performance Application Programming Interface, available from ⟨http://icl.cs.utk.edu/papi/⟩.
16) Li, E. and Thomborson, C.: Data Cache Parameter Measurements, *Proc. 1998 IEEE International Conference on Computer Design* (1998).
17) Intel Corporation: Intel Architecture Software Developer's Manual.
18) Intel Corporation: Intel®64 and IA-32 Architectures Software Developer's Manuals.
19) Ferrie, P.: Attacks on Virtual Machine Emulators, available from ⟨http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf⟩.
20) Raffetseder, T., Kruegel, C. and Kirda, E.: Detecting System Emulators, *ISC 2007* (2007).
21) Rothman, M. and Zimmer, V.: DETECTING VIRTUALIZATION (2006).
22) Kang, M.G., Yin, H., Hanna, S., McCamant, S. and Song, D.: Emulating Emulation-Resistant Malware, *VMSec'09* (2009).
23) Postel, J.: INTERNET CONTROL MESSAGE PROTOCOL (1981), available from ⟨http://www.ietf.org/rfc/rfc792.txt⟩.
24) Braden, R.: Requirements for Internet Hosts – Communication Layers (1989), available from ⟨http://www.ietf.org/rfc/rfc1122.txt⟩.
25) ICMP Time Stamp Not Supported (2003), available from ⟨http://support.microsoft.com/kb/156165⟩.
26) Intel Corporation: Software Developer' s Manual Volume 3B: System Programming Guide, Part 2.
27) Advanced Micro Devices: AMD64 Architecture Programmer' s Manual Volume 2: System Programming.
28) Intel Corporation: IA-PC HPET (High Precision Event Timers) Specification.
29) Oracle Corporation: STP1030 BGA UltraSPARC-II User's Manual.
30) SPARC International: The SPARC Architecture Manual, Version 9.
31) Hewlett-Packard Development Company, L.P.: PA-RISC 1.1 Firmware Architecture Reference Specification.

(平成 22 年 11 月 30 日受付)
(平成 23 年 3 月 7 日採録)

宮本久仁男（正会員）

1991 年電気通信大学電気通信学部通信工学科卒業．同年 NTT データ通信（株）入社．OS，電子商取引実験システム等の研究開発や社内技術支援，情報セキュリティ推進管理業務等を経て，現在は情報セキュリティ関連技術の研究開発に従事．システム基盤技術や情報セキュリティ全般に興味を持っているが，特に仮想マシンモニタや OS，ネットワークをはじめとするシステム基盤技術や，それに関連したセキュリティに強い興味を持つ．2005 年情報セキュリティ大学院大学情報セキュリティ専攻科博士後期課程入学．

田中　英彦（名誉会員）

1970 年東京大学大学院工学系研究科電気工学専門課程修了，工学博士．東京大学にて計算機アーキテクチャ，並列処理，人工知能，自然言語処理，分散処理，メディア処理等の教育・研究に従事．東京大学工学部教授，同大学院情報理工学系研究科長を経て，2004 年情報セキュリティ大学院大学情報セキュリティ研究科長・教授．人工知能学会論文賞，ACM SIGGRAPH'99 Impact Paper Award，人工知能学会功績賞，東京都科学技術功労者表彰，経済産業大臣表彰等受賞．情報処理学会名誉会員，日本学術会議会員，IEEE Fellow，東京大学名誉教授．