社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2006−OS−101（5）
2006／2／16

CILで表現されたOSカーネルの実行方法

高橋明生[†]　加藤和彦[††]

我々は，インタープリタとコンパイラを併用することで，事前に特別なデータを予め用意する必要なしに中間言語 (CIL) によって書かれた OS カーネルを動作させる方法を研究している．この方法によると，カーネルが扱うどのようなデータも仮想マシンの内部に配置することができ，従来は最終的に仮想マシン外部に移譲してしまっていた処理も CIL カーネル内で実行することができるようになる．

An executing method of a OS kernel represented in CIL

Akio Takahashi[†] and Kazuhiko Kato[††]

This paper proposes a method to run the OS kernel written in Common Intermediate Language (CIL) without preliminary data by applying an interpreter together with a compiler. With the method, kernel data are placed inside of a virtual machine and every functions can be written directly in our CIL kernel.

1. はじめに

従来，PC 上でのアプリケーションの開発には C 言語などが使われてきたが，近年急速に C#や Java などの新しい言語を用いるようになっている．これによって，たとえば C 言語の場合に多発していたメモリ関係のバグが減少したり（あるいは無くなったり），オブジェクト指向による開発手法を取り入れることが可能になるなど，いくつもの有用な利点がある．

C#や Java のもたらす重要な利点のひとつは仮想マシンを基礎とする実行環境である．C#の場合には中間言語として CIL[5)] (Common Intermediate Language) を，仮想マシンとして CLI[5)] (Common Language Infrastructure) を定義している[☆]．

従来，C#（および CLI）はアプリケーション開発にしか用いられておらず，システムソフトウェアにはほとんど利用されていない．特に，OS カーネルのために活用するには，CLI が中間言語と仮想マシンを想定していることに由来する技術的な課題がある．

中間言語は，逐次解釈や JIT コンパイル (Just-in-time compile)，AOT コンパイル (Ahead-of-time compile) などによってコンピュータ上で実行される．このとき，仮想マシンの一部の命令は単純に処理できないので，そのような処理は実マシンに置かれたランタイムプログラム[☆☆]によって処理されることが多い．具体的には，型システムに関する機能やコンパイルに関する機能が特にランタイムプログラムへ依存してしまう．

しかし，OS カーネルに CLI や JVM に由来する技術を適用する場合，通常は実マシンに置かれるランタイムプログラムの存在は期待できないため，ランタイムへと移譲していた処理の実装が困難になる．

JX[1)] や JNode[3)]，Singularity[2)] では，配布イメージ[☆☆☆]をビルドする際に AOT コンパイルし，カーネルの機械語と初期データをセットで生成してしまうことで，中間言語から得られたプログラムがランタイムを必要とする実行ステップを事前に終えてしまう．しかし，この手法ではビルド時に生成する実行イメージとカーネルの実行時に扱うメモリイメージの間に直接の相関性がなく，カーネルに関するデータだけが特別扱いされてしまう．このような規格外のデータは一般にポインタを介してアクセスするなど，CLI 非準拠の部分では CLI の利点を生かすことができなくなってしまう．

さらに，Java および JVM を OS カーネルに適用する研究はいくつもあるものの，CLI を OS カーネルに適用する研究は非常に数が少ない.（2006 年 1 月時点で我々の他には Singularity[2)] しか見つけることができなかった.）しかし我々は特にポインタや機械語の扱いなどの点において JVM 規格よりも CLI 規格の方が OS カーネルには

† 武蔵工業大学大学院工学研究科
Musashi Institute of Technology
†† 筑波大学大学院システム情報工学研究科
Institute of Information Sciences and Electronics, University of Tsukuba

☆ CLI の場合は仮想マシンというよりは仮想環境と表現したほうが的確であるが，一般的な呼称に従い仮想マシンと呼ぶことにする．

☆☆ 実行時に必要な仮想マシンの実行を補助するためのプログラム．たとえば，インタープリタや JIT コンパイラなどがランタイムの中に含まれることになる．

☆☆☆ ここではユーザが実行可能なバイナリイメージを指している．OS の場合は，起動可能メディアの生イメージであることが多い．

適していると考えており，CLI での OS カーネル実装に意義があると考えている．

そこで我々は，AOT コンパイルや外部への特別な依存をすることなくカーネル単独で自分自身の実行環境を構築することで，カーネルを含むシステム全体が CLI という単一の系で説明できる手法を提案する．

2. 中間言語カーネルにおける課題

CIL で表現されたカーネルを実行する方法は次の 3 種類が考えられる．

(1) インタープリタが CIL のまま実行する．
(2) JIT コンパイルで実行時に機械語に変換する．
(3) AOT コンパイルで予め機械語に変換する．

方法 1 では，インタープリタ自体は機械語で書かれている必要があるため，C++などを使って記述する必要がある．しかし，この方法では C++で記述されたプログラムがランタイムプログラムとしてカーネルに残存することになり，(たとえコード量は小さいとしても）カーネル全てを CLI で記述したことにはならない．また，処理速度も機械語実行に比べて速くならないため，カーネルには不適切であると考えられる．

方法 2 でも JIT コンパイラは機械語で書かれている必要がある．方法 1 と同様に外部ランタイムプログラムとしてコンパイラが残存してしまい，カーネル全体が CLI で記述できるわけではない．

方法 3 では，AOT コンパイラはある程度☆まで処理されたデータを生成する必要があり，そのデータを OS 起動時に解釈して適切に配置する必要がある．この手順は自明ではないため，方法 1 や方法 2 のように既存手法の延長として実現することは出来ない．

このように各方法とも長所・短所があるが，方法 2 および方法 3 の機械語を得られるという特徴は，カーネルであることを考えると必須であると考えられる．たとえば，ハードウェアの処理などでは実時間で制約のある場合があり，そのような時は機械語による高速実行が求められる．しかし，カーネルは CIL で書かれているため，機械語を得るにはコンパイルする必要がある．コンパイルを事前に行うとすれば，その結果をブート時に誰が，どのように利用するかが問題になってしまう．

3. 提 案 手 法

3.1 カーネルの構造

本手法では，2 章で述べた問題をインタープリタとコンパイラを組み合わせて用いることによって解決し，事前に特別なデータを用意することなくカーネル全体を CLI 系として構築する．

☆ たとえば，起動時に必要なすべてのデータを満たすデータを用意すれば，ブート時のロード処理は簡単になるが，AOT コンパイルは複雑になる．簡素なデータのみ用意すれば，AOT コンパイルは簡単だが，ロード処理は複雑になる．

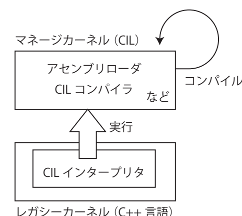

図 1 二つのカーネルの関係
Fig. 1 Relation between two kernels

まず，本手法で用いる 2 つのカーネルとその内部コンポーネントについて説明する（**図 1**）．

- レガシーカーネルと，CIL インタープリタ．
- マネージカーネルと，アセンブリローダおよび CIL コンパイラ．

レガシーカーネルは C++で記述し，OS ビルド時に機械語に翻訳される．マネージカーネルは CLS 準拠言語☆☆（大半が C#で，残りは Managed C++）で記述し，ビルド時に CIL に翻訳され，実行時に機械語に翻訳される．

本手法で用いるインタープリタとコンパイラに関して，基本的な作り方は，それぞれ単体のプログラムとしては既存のものと大きな違いはない．本手法の課題は，外部ランタイムのない状態で，インタープリタとコンパイラをどう組み合わせて動作させ，外部ランタイムプログラムに頼らないカーネル実行環境を得るかということである．

はじめに，本手法によるカーネル起動手順全体を明らかにするために 3.2 節で起動処理全体について説明する．その中で本手法に関する重要な処理として，メタデータ☆☆☆とコンパイルに関するものが重要であるため，メタデータについては 3.3 節で，コンパイラについては 3.4 節で説明する．

3.2 カーネルのブート

コンピュータが起動したあと，レガシーカーネルがメモリに配置され，通常必要とされる初期化処理を終えたとする．(PC/AT においては，ブートストラップローダがレガシーカーネルを読み込み，レガシーカーネルが外部記憶装置への I/O 準備を完了した状態である.)

次にレガシーカーネルは，インタープリタによる実行のためにマネージカーネルをロードし，そのメタデータを構築する．3.3 節で述べるものと違い，このメタデータはレガシーカーネル（のインタープリタ）が利用するものであるから，CLI とは互換性のない表現で構わない．このときの様子を**図 2** に示す．メモリ空間には C++用の領域しかなく，その中にはロードしたメタデータと，レガシーカーネルのコンポーネントである CIL インター

☆☆ Common Language Specification compliant language. CLI に対応した言語．

☆☆☆ プログラムとデータに関する情報．例えばクラスの構造（フィールドやメソッド）など．CLI ではメタデータをリフレクションを通して参照できる．

図 2　初期状態
Fig. 2　Initial state of kernel

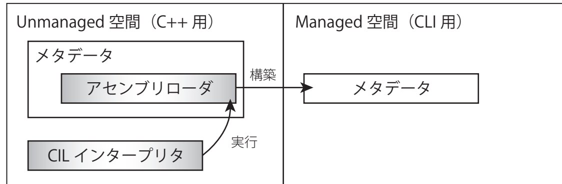

図 3　メタデータ構築手順
Fig. 3　First process: Construct Metadata

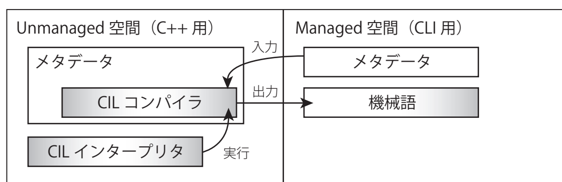

図 4　コンパイル手順
Fig. 4　Second process: Compile Kernel

プリタがある．

次にレガシーカーネルは，CLI 用のメモリ（Managed 空間）として未使用の領域を予約するなど，CIL インタープリタを動作させるための準備を行う．そして，マネージカーネル内のアセンブリローダを動作させ，Managed 空間内にもメタデータを構築する（**図 3**）．

そして最後に，構築したメタデータをコンパイルすることでマネージカーネルの機械語表現を得ることができる（**図 4**）．実行に必要なランタイムプログラムとデータも Managed 空間にあるため，生成した機械語はレガシーカーネルおよび Unmanaged 空間にあるデータへ依存しない．このとき，プログラムを全てコンパイルすることによって容易に機械語がインタープリタに依存しないようにすることが可能であるが，本研究では，必要な箇所のみ明示的にコンパイルし，残りは JIT コンパイルすることを試みている．

3.3　メタデータの構築

単純に CLI を実装することを考えたとき，プログラムの実行制御はランタイムプログラムの役割のひとつであるから，メタデータはランタイムプログラムと共に保持する手法が自然である．しかし，マネージカーネルはランタイムプログラムに依存することなく単独で動作しなければならないため，メタデータをなんらかの方法で Managed 空間内に構築する必要がある．このとき，Managed 空間が CLI 向けであることから，Managed 空間を操作する処理は CIL を使用した方が記述しやすい．そこで，マネージカーネル内にアセンブリローダとして，アセンブリをロードしメタデータを構築するプログラムを用意した．しかしここで，次のような循環する問題が起こってしまう．

- CIL プログラムの動作にはメタデータが必要である．
- メタデータは（CIL プログラムである）アセンブリローダの出力として得られる．

そこで，最初に処理するシステムアセンブリ☆について次のようなロード手順を踏むことで，上記の問題を回避した．

（1）　仮メタクラスの作成
（2）　アセンブリのロード
（3）　真のメタクラスへ切り替え

これによって，CIL で記述されたプログラムをメタデータを構築するプログラムとして利用することができる．次から各手順を説明する．

手順 1: 仮メタクラスの作成

メタクラスは CLI では System.Type クラスとして表される．（Type は抽象クラスであり，実装は派生クラスが行う．）Type の機能の一部はリフレクションと呼ばれ，そのインスタンスが表す型の名前や，フィールド・メソッドの一覧などを得ることができる．

しかし，メタデータ構築の初期段階ではアセンブリなどメタデータに関する情報が整っていないため，リフレクションの全ての機能を実現することはできない．そこで，初期段階では完全な実装を諦め，限定的な（ただし初期手順の実行には十分な）機能だけを実装した PseudoType クラスを用意した．

PseudoType を用いて，次のような手順でメタクラスを生成する．

（1）　**PseudoType を表す PseudoType のインスタンス P を確保**
「PseudoType を表す PseudoType のインスタンス」とは，リフレクションによって PseudoType に関する情報を得られる PseudoType 型のインスタンスのことである．これのための領域をメモリ上に確保しゼロで初期化する．このときコンストラクタは呼び出さない．これを P とし，P の型オブジェクト☆☆を P に設定する．
この一連の操作はレガシーカーネルにハードコーディングするため，その処理の内容は CLI の命令に限定されない．そのため，コンストラクタを呼び出さずに P を確保するといったことが可能である．

（2）　**P の親クラスを処理**
P の親クラスに関する型オブジェクトを作成する．（コンストラクタも呼び出す．）作成した型オブジェ

☆ 起動時に必要なアセンブリで，マネージカーネルと Mono クラスライブラリである．
☆☆ あるインスタンス i の型を表す Type オブジェクトを型オブジェクトと呼ぶことにする．CLI では i.GetType() で型オブジェクトを得られる．

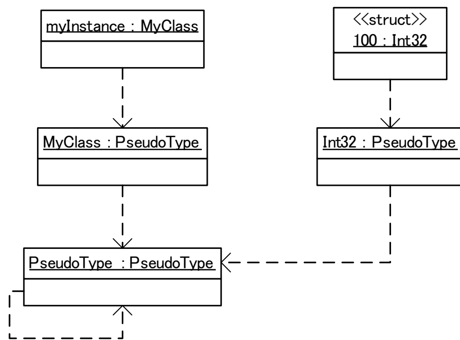

図 5 切り替え前のメタデータ構造
Fig. 5 Metadata structure before switching

クトの型オブジェクトは P になる．

(3) **P を構築**
P のコンストラクタを呼び出す．このとき，リフレクションに必要な名前や名前空間名などが渡される．

(4) **その他のメタクラスを作成**
インタープリタが参照した時点で，必要となった型のメタクラスが作成される．

以上の処理が終わった後のメタクラスに関する様子を**図 5** に示す．なお，Int32 (System.Int32) は 32 ビット符号付き整数の型である．また，MyClass はユーザ定義のクラス型であるとし，myInstance は MyClass のインスタンスであるとする．PseudoType は PseudoType 自身が定義されているアセンブリが読み込まれていない状態で使用されるため，PseudoType には本来 Type の派生クラスが持つべき機能の一部しか実装できないが，手順 3 までの実行においては未実装の機能は使用されないため問題にはならない．

手順 2: アセンブリのロード

PseudoType を整えたのち，レガシーカーネルはアセンブリファイルのイメージを CD-ROM から読み込み，マネージカーネルのアセンブリローダにイメージを解析・ロードさせる．ロードが終了すると，Int32 や System.Object などの基本的な型やマネージカーネルに含まれるクラスなどのメタデータに，CLI プログラムがアクセスできるようになる．

手順 3: 真のメタクラスへ切り替え

アセンブリのロードが終わると，PseudoType を用いて生成されたメタデータを正しい実装系へ切り替える．切り替えた後の様子を**図 6** に示す．なお，ClassType と PrimitiveType は PseudoType に代わる正しいリフレクション実装を持つクラスであり，ClassType はクラス型，PrimitiveType は Int32 などの基本型に適用されるメタクラスである．mscorlib および cscorlib はアセンブリの名前である．

ClassType など正しいリフレクション実装のためのクラスは，自分が定義されたアセンブリを知っており（図 6 の Assembly オブジェクト）メタデータを参照すること

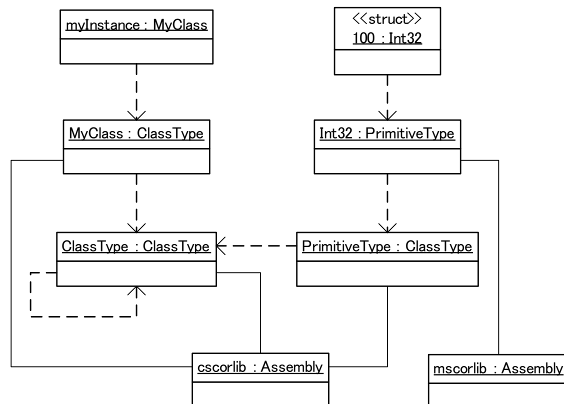

図 6 切り替え後のメタデータ構造
Fig. 6 Metadata structure after switching

ができるため，リフレクション機能を完全に実装することができる．

切り替えは，レガシーカーネルによって，メモリ中の参照を書き換えることで行われる．すでにメモリ中にロードされている（つまり使われている）型は，PseudoType を用いてロードされてしまっているため，これらについてもう一度 ClassType などで型情報をロードし直して，PseudoType を参照している箇所をロードし直した ClassType などへの参照に書き換える．これによって，以降，リフレクションを完全な形で利用可能になる．

3.4 コンパイル

メタデータがリフレクションを通して利用可能になると，その情報を基にコンパイルすることができるようになる．コンパイル作業は次のような手順で行う．このとき，単純な機械語へ置き換えることが不可能な CIL の命令があり，それらについて 3.5 節で述べる．

手順 1: 基本コレクションクラスのコンパイル

基本コレクションクラスとは特殊化された 2 つの Hashtable☆1である．（説明のために Hashtable を挙げたが，クラス階層的な関係はなく，表面的に同等という意味である．）それぞれ，キーを MethodInfo☆2と Int32 に限定している．

これらは次のようなランタイムの動作にとって必要になる情報を格納する．

- ロード済みアセンブリのテーブル．キーが数値 ID で，値がアセンブリオブジェクト．
- 生成済み機械語コードのテーブル．キーが MethodInfo で，値が CodeInfo☆3．
- ハンドル☆4に対応するオブジェクトのテーブル．キーがハンドルの値で，値がオブジェクト．

このようなデータの格納と操作は標準の Hashtable で

☆1 System.Collections.Hashtable
☆2 System.Reflection.MethodInfo
☆3 コード情報を格納するクラス．
☆4 具体的には RuntimeTypeHandle と RuntimeFieldHandle と RuntimeMethodHandle を指す．

も提供されているが，標準の実装では複雑な処理を含むため動作の予測が困難になってしまう．例えば，単にInt32をキーとした場合でも，IComparableインターフェイスを経由するため仮想関数呼び出しが関わってくるなどして好ましくない．基本コレクションクラスではキーの型を限定することで，たとえばInt32なら単に数値比較だけでキー比較をすませており，数値比較は単純な機械語に置き換えることができる．

基本コレクションクラスは仮想関数呼び出しの処理など基本的な命令の実行に必要なので，基本コレクションクラスのメソッドに対してJITコンパイルが発生してはいけない．そこで，基本コレクションクラスのメソッドを予めコンパイルすることによってJITコンパイル処理が発生することを防いでいる．これは，基本コレクションクラスの実装にあたり，仮想関数の呼び出しなど動的束縛が起こる実装を避けているために可能である．

ここで，コンパイルしたメソッドが直ちに機械語実行されるわけではない．コンパイルしたメソッドが機械語で実行されるのは，他の機械語から呼び出されたときになる．

手順2: コンパイルメソッドのコンパイル

基本コレクションクラスに続き，コンパイル処理の開始点となるメソッドをコンパイルして機械語Mを生成する．そして，コンパイル対象メソッドFを，Mによってコンパイルする．

このとき，Mが呼び出したメソッドも機械語で実行するためにJITコンパイルされる．ただし，JITコンパイル処理自体はインタープリタで実行されるようになっているため，コンパイルで実行されるメソッドのコンパイル処理はインタープリタによって行われる．これはJITコンパイル処理が循環しないようにするためである．

この手順が終わったとき，コンパイルに必要なメソッドの機械語が（JITコンパイルによって）得られていると考えられる．

手順3: 再リンク

これまでの手順で生成した機械語はJITコンパイルのためにインタープリタを呼び出してしまうため，それらの呼び出し部分を機械語のJITコンパイル手続きへの呼び出しに書き換える．(call命令およびcallvirt命令が関係する．3.5節参照.)

手順4: カーネル開始メソッドのコンパイル

最後にマネージカーネルのエントリポイント（レガシーカーネルから制御を完全に引き渡されるメソッド）をコンパイルし，そのメソッドを呼び出す．

3.5 補助メソッド

CILの命令には単純な機械語への変換が不可能なものがある．そのような命令はコンパイラが備える補助メソッドの呼び出しに置き換えられ，実行時には命令の機能を補助メソッドが代行する．**表1**に，補助メソッドを必要とする命令と，対応する補助メソッドを示す．

表1 補助メソッド一覧
Table 1 Supporting methods

命令	メソッド
newobj	AllocateObjectImpl
newarr	AllocateSzArrayImpl
ldstr	LoadStringImpl
box	BoxImpl
isinst	IsInstImpl
castclass	CastClassImpl
call	PrepareExecutableCode *
callvirt	FindActualMethod *
callvirt	FindActualMethodForInterface *

表2 Double型検証コード
Table 2 Validating code about Double

```
//static void TestCalc(double x, double y);
double[] arr = {0, 1, −1, 10, −10
    , short.MaxValue, short.MinValue
    , int.MaxValue, int.MinValue
    , double.MaxValue, double.MinValue
    , double.PositiveInfinity
    , double.NegativeInfinity
    , double.NaN };
foreach(double x in arr) {
    foreach(double y in arr) {
        TestCalc(x, y);
    }
}
```

このうち，現状では*付きのメソッドについてはインタープリタに頼って実行している．この問題については5章で触れる．

4. 実 装

提案手法を実装したOSカーネルMicrosoft Windows上でビルドし，VMware[7]上でOSとして動作させた．このとき，ホストマシンのCPUはIntel Pentium4 3.40GHzであり，VMwareには256MBのメモリ領域を割り当てた．

なお，検証のためのプログラムコードはすべてC#であり，重要な箇所のみを抜き出している．

4.1 インタープリタの数値演算

インタープリタについて，Int32とSystem.Int64とSystem.SingleとSystem.Doubleの算術演算（および整数型には論理演算）を計算させた．この4種を選んだ理由は，CILにおけるスタック上の数値が実質的にこの4種で表されるからである．**表2**にDoubleに関する試験のコードの抜粋を示す．ただし，`TestCalc`は引数の四則演算を行い，結果を表示するメソッドである． この結果，Int32とInt64については，このコードをWindows上の.NET Frameworkで計算させたときと同じ結果が得られた．SingleとDoubleについては，最大の数や最小の数の計算で差違があったものの，計算規則☆については等しい結果が得られた．

☆ たとえば，NaNに対する四則演算結果はNaNになる，など．

表 3 Double.MaxValue の値
Table 3 Value of Double.MaxValue

実行系	Double.MaxValue の値
.NET Framework	1.79769313486232E+308
Our Interpreter	1.77008066823353E+308

表 4 メタデータ検証コード
Table 4 Validating code for metadata

```
static void ShowMetatype(object o) {
    Console.WriteLine("{0}", o);
    Console.WriteLine(">␣{0}"
        , o.GetType().Name);
    Console.WriteLine(">>␣{0}"
        , o.GetType().GetType().Name);
}
static void TestMetatypes() {
    ShowMetatype(1);
    ShowMetatype("abc");
    ShowMetatype(typeof(int));
    ShowMetatype(typeof(Console));
}
```

違いが見られた Double 最大値の表示結果を**表 3** に示す．この違いは，コンソールへの表示をする際の計算誤差だと思われる．インタープリタでは計算結果を逐一 32 ビットまたは 64 ビットでスタックに格納しているのに対し，.NET Framework は最適化によって可能な限り FPU レジスタ（IA-32 の場合は最大 80 ビット）を利用しており，計算過程か文字列変換過程で誤差が発生していると考えられる．しかし，この誤差は Double.MaxValue という非常に大きな数値付近でのみ発生し，Int.MaxValue までの計算は発生しなかった．このため，インタープリタが利用される段階では浮動小数点数があまり使用されないということもあり，動作には影響しないと考えられる．

4.2 インタープリタの動作

インタープリタによって，たとえばメソッド呼び出しなどが正しく動作するかといった実験は，インタープリタが後続の試験の基礎であることを考えて，改めて行うことはしなかった．

後続の試験を行うことができていることから，少なくとも，Mono クラスライブラリ[4] や FreeType[6] について，それらの動作結果を確認できる程度には正しく実装されていると言える．

4.3 メタデータの構築

メタデータが正しく構築できていることを確認するために，**表 4** に示すコードを実行した．

実行結果（**表 5**）より，たとえば 1 という 32 ビット符号付き整数の型は Int32 であり，それの型は PrimitiveType であり，それの型は ClassType であり，最終的に ClassType の型は ClassType であるといったように，メタデータが正しく構築されている様子が分かる．

4.4 コンパイラ

インタープリタとメタデータが利用できることが確認できたので，次にそれらに基づきコンパイラが正しく動作するか確認するためのコード（**表 6**）をコンパイルし

表 5 メタデータ検証結果
Table 5 Result of medata validation

```
1
> Int32
>> PrimitiveType
abc
> String
>> StringType
Type: System.Int32
> PrimitiveType
>> ClassType
Type: System.Console
> ClassType
>> ClassType
```

表 6 コンパイル検証コード
Table 6 Validating code for compilation

```
public static int Calculate(int x) {
    if(x==0) return 0;
    return x+Calculate(x−1);
}
```

表 7 機械語実行時の性能
Table 7 Performance of native code

引数 x	逐次実行時間	機械語実行時間	速度比
1	30974	24922	1.243
10	91103	24871	3.660
100	736865	31917	23.087
1000	22781301	111316	204.654

実行した．結果を**表 7** に示す．計測は各 x について 10 回試行し，各計算でもっとも良い数値を採用した．時間の単位はクロックである．$x = 1$ でも時間が 0 に近くないのは計測のための定数コストのためであると考えられる．速度比より，明らかにインタープリタに比べて性能が向上しており，機械語実行による効果が分かる．

x が増加するにつれて性能比が飛躍的に向上しているのは，インタープリタの一部のデータ構造について，呼び出しの深さにつれて操作に掛かるコストが増加するためだと思われる．問題箇所を最適化することで性能改善は期待できるが，機械語より速くなる見込みは薄い．

4.5 JIT コンパイル

次に JIT コンパイルが動作することを確認するために**表 8** に示すようなコードを用意し，`Calculate1` のみをコンパイルしたのち実行した．

実行したときの出力を**表 9** に示す．まず Calculate1 がコンパイルされており，`0x0211926C` 番地にコードが配置されている（1〜2 行目）．同時に，Calcualte1 が呼び出す `Calculate2` を JIT コンパイルするためのコード（スタブコード）が `0x02119BF4` 番地に配置されている（3 行目）．

最初の $x = 0$ の計算では Calculate2 を呼び出さないため，計算はそのまま終了していることが分かる（4 行目）．次に $x = 1$ の実行では，スタブコードが呼び出さ

表 8　JIT コンパイル検証コード
Table 8　Validating code for JIT compilation

```
static int Calculate1(int x) {
    if(x==0) return 0;
    return x+Calculate2(x-1);
}
static int Calculate2(int x) {
    if(x==0) return 0;
    return x+Calculate1(x-1);
}
static int Test(int x) {
    Console.WriteLine("x={0,3}, y={1,3}"
        , x, Calculate1(x));
}
static int Test() {
    Test(0);
    Test(1);
    Test(10);
}
```

表 9　JIT コンパイルの応答
Table 9　Output of JIT compilation

```
1  COMPILE> sum:Calculate1
2  [0x0211926C:E] sum:Calculate1
3  [0x02119BF4:C] sum:Calculate2
4  x=  0, y=  0
5  COMPILE> sum:Calculate2
6  [0x0211D744:E] sum:Calculate2
7  x=  1, y=  1
8  x= 10, y= 55
```

れてコンパイルが行われており，Calculate2 の実行可能コードが 0x0211D744 番地に配置されている（5〜6 行目）．このとき，Calculate1 の Calculate2 呼び出し命令の呼び出し先を 0x0211D744 番地に書き換えている．

以降の実行では JIT コンパイルは呼び出されることなく計算が行われている（7〜8 行目）．

4.6　OS 動作試験

提案手法が OS カーネルにおいても有効であることを確認するために，次のようなハードウェアデバイスドライバを C#で開発した．

- VGA
- ATA/ATAPI
- キーボードコントローラ
- PS2 キーボードおよび PS2 マウス

さらに OS として動作できることを確認するために，上記デバイスドライバに FreeType[6] を加えて動作させた．そのときの様子を**図 7** に示す． この試験で，画面への出力やキーボードからの入力，CD-ROM メディアから読み込みなどを確認した．

5. おわりに

本研究では，レガシーカーネルとマネージカーネルのふたつを組み合わせて動作させ，起動のための特別なデータを用意することなくシステム全体を CLI に準拠させた状態でカーネルを実行するための手法を提案した．本報告では特に問題となるメタデータの構築手順とコンパイ

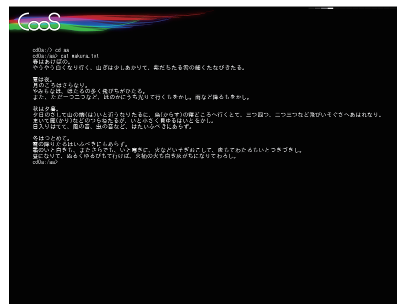

図 7　フォントレンダラで日本語を表示した画面
Fig. 7　Screenshot with Japanese characters

ル手順について解決のための手法を説明し，評価によって本提案手法が OS カーネルの実行手法として有効であることを示した．

現状では，表 1 において*付きで示したメソッドの実行を依然としてインタープリタに頼っている．そのため，インタープリタをコンピュータ上から消すまでには至っていない．これらのメソッドへの依存を取り除けない原因は，3.4 で述べた「コンパイラのコンパイル」が不十分であるためにコンパイル処理が循環するからである．この問題はコンパイル処理を行うメソッドのコールグラフを解析して必要なメソッドだけを予めコンパイルしておくことで解決できると考えられるため，現在はこの依存関係の同定作業を行っている．

謝辞　本研究は 2004 年 11 月から翌年 8 月末まで，独立行政法人情報処理推進機構 (IPA) から支援を受けた．

参 考 文 献

1) Golm, M., Felser, M., Wawersich, C. and Kleinoder, J.: The JX Operating System, *In Proceedings of the USENIX Annual Technical Conference*, pp.45–58 (2002).
2) Hunt, G. and Larus, J.: An Overview of the Singularity Project, http://research.microsoft.com/os/singularity/ (2005).
3) JNode.org: JNode Operating System, http://www.jnode.com/ (2006).
4) Mono Project: Mono, http://www.mono-project.net/ (2005).
5) Standard ECMA-335: Common Language Infrastructure (CLI), http://www.ecma-international.org/ (2005).
6) The FreeType Project: FreeType, http://freetype.sourceforge.net/ (2005).
7) VMware Inc.: VMware Workstation, http://www.vmware.com/.