システムソフトウェアと
オペレーティング・システム 90−17
（２００２．６．２８）

仮想マシンを用いたOS構成法とその通信機構

高野了成[†] 石口栄一[††]
早川栄一[†††] 高橋延匡[†††]

ウェアラブルコンピューティングや組込みシステムなど，ターゲットとするハードウェアの性能が幅広く，環境の変化に対する動的な対応が必要なシステムでは，アプリケーションやその用途に合わせてシステムを適合できることが重要である．本稿では，資源管理ポリシを仮想マシンとして提供し，仮想マシン上で動作するモジュールの組み合わせにより柔軟なシステム構成を可能とするOS構成法とその通信機構について報告する．本システムはカーネル層，仮想マシン層，アプリケーション層の三階層から構成され，カーネル層ではハードウェアアクセスの基本的なプリミティブを提供し，仮想マシン層は資源管理ポリシを提供する．また，仮想マシン間を横断して処理を実行するアプリケーションに対して継続を用いた軽量なフロー制御を提供している．本システム上にてネイティブ環境とWabaVMの双方の環境でFATファイルシステムを実装したところ，ファイル読込みの性能は約25倍という結果が得られた．

Virtual Machine Based OS Structure and Its Communication Mechanism

TAKANO Ryousei ,[†] Ishiguchi Eiichi ,[††] HAYAKAWA Eiichi [†††]
and TAKAHASHI Nobumasa [†††]

In ubiquitous computing and embedded software, systems requires dynamic configurations and adaptations for applications. Because target hardware performance is various, and it is sensitive to an environmental change. This paper describes an extensible operating system construction model that based on a virtual machine concept and its communication mechanism. We supposed a three-tiered system model: application layer, virtual machine layer, kernel layer. A kernel layer provides a hardware abstruction and primitive operations for hardware access. A virtual machine layer provides resource management policies. Further we supposed lightweight flow control using continuation for inter virtual machine communications. We have developed a FAT file system in C and WabaVM each other, and measured virtualization overheads such as file read. As a result, we found WabaVM's file read execution time is about 25 times than C.

1. はじめに

ウェアラブルコンピューティングや組込みシステムなど，ターゲットとするハードウェアの性能が幅広く，環境の変化に対する動的な対応が必要なシステムでは，アプリケーションやその用途に合わせてシステムを適合できることが重要である．そこでアプリケーション層において資源管理を行なうマイクロカーネルやシステムコンポーネントを仮想マシン上で実行するバイトコードとして提供するJava，Squeak，Inferno，.NET CRLなどのシステムが開発されている[4]．

上記のような仮想マシンは幅広いユーザのニーズに答えるため高い抽象度の命令セットを持つ高級言語マシンになっている．その特徴を次に挙げる．

- ネットワークアクセス，スレッド，ガーベジコレクション，デバイス操作など，オペレーティングシステム(以下，OS)の機能全体をカバーする抽象化層
- バイトコードを実行時にネイティブコード化するJust-In-Time(JIT)コンパイラなどによる高速化，最適化

† 東京農工大学工学部
Faculty of Engineering, Tokyo University of Agriculture and Technology
†† アルゴ21
ARGO21
††† 拓殖大学工学部
Faculty of Engineering, Takushoku University

- オブジェクト指向言語のサポート

仮想マシンを用いたアプローチはシステムに柔軟性，拡張性，安全性をもたらすが，一方で仮想化にともなうオーバヘッドが問題となる．したがって，資源管理ポリシ，言語の記述性，性能などのトレードオフを判断して，ユーザの要求に応じた実行環境が選択できることが求められる．また，複数の実行環境が並列に動作する場合は，その環境間の通信機構，資源割当て，保護機構を提供するフレームワークが必要になる．

本稿では，資源管理ポリシを仮想マシンとして提供し，仮想マシン上で動作するモジュールの組み合わせにより柔軟なシステム構成を可能とする OS 構成法とその通信機構について報告する．本システムでは，OS の各機能を仮想マシン上で動作するモジュールとして提供する．各モジュールはインタフェース記述言語によって定義し，実装言語に依存しないモジュールバインディングと名前サービスを提供する．また，仮想マシン間の軽量な通信機構を実現するために，継続をベースにしたフロー制御を提供する．

以下，本稿では，2 章でシステムの設計方針について述べ，3 章では仮想マシンを用いた OS 構成法について述べる．4 章では複数の仮想マシン間での通信機構の考察を行ない，継続を用いた設計について述べる．5 章では SH3 組込みボード上での実装，そして評価実験としてネイティブ環境，Waba 仮想マシン環境における FAT ファイルシステムの性能比較について述べる．6 章では関連研究について述べる．

2. 設 計 方 針

資源管理ポリシを複数のモジュールを組み合わせることで柔軟にシステムを構築するために，次に示す設計方針を立てた．

(1) 複数の仮想マシンを用いた OS 構成
アプリケーションに対する実行環境を仮想マシンとして提供し，資源管理ポリシごとに複数の仮想マシンを提供する．
アプリケーションは仮想マシン経由で資源にアクセスする．

(2) 軽量な仮想マシン間通信機構
単一のアプリケーションでも処理内容に応じて複数の言語を利用して実装したり，異なる資源管理ポリシで動作させるには，仮想マシン間を横断するための通信機構が必要である．ターゲットとする環境はハードウェア性能の幅が大きいので，汎用的で省資源で動作する軽量な通信機構を提供する．

(3) 言語中立なインタフェース情報の管理
アプリケーションの実装言語や実行される仮想マシンに依存せず，モジュール間のバインディングを可能にするためにシステムで一意な名前と型情報管理を提供する．

3. 仮想マシンを用いた OS 構成法

本章では，まずシステム全体の概要について述べ，各構成要素について述べる．

3.1 システムの概要

本システムは **図 1** で示すようにカーネル層，仮想マシン層，アプリケーション層の三層から構成される．カーネル層はハードウェアの抽象化を行ない，仮想マシン層はアプリケーションに対して実行環境，資源管理のポリシを提供する [1]．各仮想マシンは完全なサンドボックスとして分離されているわけではなく，システムに登録された公開インタフェースを介して相互に呼び出すことが可能である．

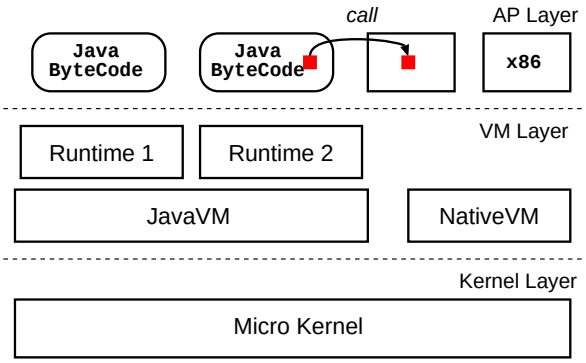

図 1 システムの概要
Fig. 1 A System Overview

3.2 カーネル層

本システムでは複数の仮想マシン上で動作するモジュールを提供するために，カーネル層では次の二つの機能を提供する必要がある．

- モジュールの公開インタフェースを管理するシステムで一意の名前空間機構の提供
- モジュールの各実装とその実行環境である仮想マシンの対応付け

3.3 仮想マシン層

仮想マシンはバイトコードインタプリタなど実行言語ごとに共有される部分と，資源管理ポリシを提供するランタイムから構成される．前者はネイティブ言語，後者は Java で実装される．

また，ランタイムを積み重ねることで新しいランタイムを作ることが可能である．この場合，処理がオーバライドされた場合はその処理を実行し，新たな定義がない場合は，以前の処理をそのまま実行することになる．

仮想マシンは次に示すようなインタフェースを提供する必要がある．

- カーネル層とのインタフェース
  - 仮想マシンの生成，削除
  - 資源管理ポリシの設定
- アプリケーション層とのインタフェース
  - モジュールバインディング

– 手続き呼出し

3.4 アプリケーション層

アプリケーションは仮想マシンの資源管理ポリシにしたがって資源にアクセスする．

4. 仮想マシン間の通信機構

本システムでは複数の仮想マシンが並列して存在する．本章ではアプリケーションがこのような仮想マシン間を横断して処理を実行する機構について述べる．

4.1 名前空間とインタフェース情報

本システムにおけるモジュールの定義から実行までの流れを**図 2** に示す．モジュールのインタフェースはIDL(Interface Difinition Language) によって記述され，そのインタフェース情報はシステムが管理する名前空間に登録される．インタフェース情報はメソッドの識別子名と引数，戻り値の型情報から構成される．型情報は各言語により異なるが，共通の型表現を定義し，利用する必要がある．また，そのインタフェースを実装したモジュールは各仮想マシン上で実行される．図 2 で定義しているTime モジュールは，C と Java など複数言語によって実装されたり，同じ言語での実装でも仮想マシンにおけるポリシの違いから異なる仮想時間を提供する実装になることが考えられる．

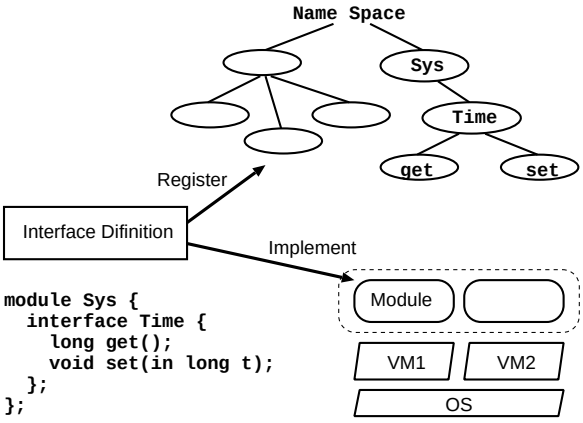

図 2 モジュールと名前空間
Fig. 2 Modules and Name Space

本システムにおけるプログラミングの流れを次に示す．

(1) インタフェースの定義
IDL によってモジュールのインタフェースを記述する．

(2) インタフェースの実装
IDL プログラムを実装プログラミング言語用にコンパイルすることで，各言語用のスケルトンコードが出力される．プログラマはスケルトンの中身を記述する．

(3) インタフェースの登録
インタフェース定義に対する実装であるオブジェクトモジュールを使うには，まず名前空間に登録する必要がある．

4.2 フロー制御方式

仮想マシン間の動作フローを制御する方式として次の三つの方式が考えられる．

- フォールト
  ソフトウェア割込みなどのフォールトを発生させ，フォールトハンドラによって通信相手の仮想マシンに制御を渡す．
- スレッド
  仮想マシンごとにプロセッサコンテキストを持ち，メッセージパッシングなどのスレッド間通信を利用する．並列なフローを容易に記述することができる．ただし同期が必要になる．
- 継続
  単一スレッドなのでカーネルによるスケジューリングが不用であり，スケジューリングポイントをユーザ側で制御できる．
  継続は大域脱出やコルーチン，バックトラックの実装に利用されており，Scheme の call/cc，C の setjmp/longjmp，Java の try/catch などに相当する．

OS/omicronV4 では，フォールトをトリガにした動的リンクを利用してモジュールバインディングを含むフロー制御を実現している [2]．フォールトは割込みハンドラの処理やパイプラインの乱れによるオーバヘッドが問題になる．また実装には外部ページャを用いており，柔軟ではあるが，汎用的な実装とは言えない．

4.3 継　　続

仮想マシン間通信に継続を用いることは，フォールトやスレッドを利用するより高速で，メモリフットプリントを押えることができる．一方でコンテキストが異なる仮想マシン間では，仮想マシン依存の情報を保持し，解釈する必要がある．

本システムでは，通常の継続情報に加えて次に示すような仮想マシン依存の情報を保持することで，仮想マシンに依存しない仮想マシン中立な継続を実現する．継続はこの情報を保持したまま動作するので，その環境に応じて処理が実行される．例えば，名前空間が異なれば，同じアプリケーションでもメソッドのバインドが異なることになる．

- 仮想マシンの識別子
- エントリ関数
- 名前空間の指定

言語 C で記述されたアプリケーションが Java のメソッドを呼び出すときのフロー制御を **図 3** に示す．継続に関する処理は **表 1** に示すように言語 C のレベルで提供しており，Java アプリケーションは JNI 経由で継続処理を呼び出すことになる．

4.4 仮想マシンの記述

仮想マシンインタフェースに関する API を**表 2** に示す．

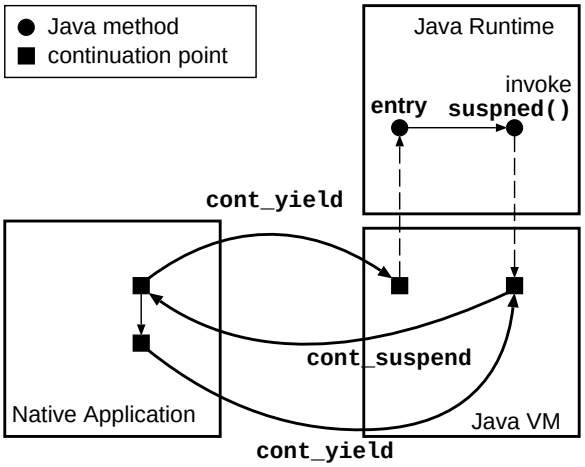

図 3 継続を利用したフロー制御
Fig. 3 A flow control using continuation

表 1 継続関数
Table 1 Continuation Functions

cont_create	新規に継続を作成する
cont_destroy	継続を破棄する
cont_yield	最後に suspend した直後から実行を継続する．はじめて実行する場合は，スタートアップ関数を実行する
cont_suspend	現在の実行状態を保存し，yield から実行を継続する
cont_test_terminated	継続が終了しているか調べる
cont_reset	継続の内容を再設定する
cont_attr_init	属性を初期化する

表 2 仮想マシンインタフェース
Table 2 Virtual Machine Interface

vm_init	仮想マシンを初期化する
vm_exit	仮想マシンを終了する
vm_entry	仮想マシンのスタートアップ関数

仮想マシンの初期化処理のコードを次に示す．cont_t が継続情報であり，仮想マシン間で環境を引き継いで実行するために vm_init で得られた仮想マシンごとの環境情報を cont に保持した状態で，継続処理が実行される．

```
cont_t cont;  /* スタートアップ用の継続 */
vmenv_t vmenv; /* 仮想マシンの環境情報 */

/* 継続の作成 */
cont_create(&cont, attr, vm_entry, arg);
/* 初期化処理 */
vmenv = vm_init();
/* 仮想マシンの環境情報を cont に設定する */
cont->vmenv = vmenv;
/* 実行開始 */
cont_yield(cont);
  :
/* 終了処理 */
vm_exit(cont);
```

継続処理用に contos.system.cont ライブラリを提供しており，対応するネイティブ関数が実行される．cont_suspend に対応する処理を次に示す．

```
/* contos.system.cont.suspend() に対する
   ネイティブ関数 */
Var contos_system_suspend(Var stack[])
{
  cont_t cont = stack[0].obj;
  cont_suspend(cont);
}
```

5. 実現と評価実験

本システムを日立 SH3 を搭載した組込みボード CAT68701(**図 4**) 上にて実装した．CAT68701 の主なスペックを**表 3** に示す．なお，クロス開発環境として RedHat Linux 7.2 と gcc 2.95.3 を用いた．

表 3 CAT68701 のスペック
Table 3 CAT68701 Specification

CPU	SH7708R 98MHz
メモリ	SDRAM 32MB
ストレージ	Compact Flash I/F (SanDisk 64MB CF)
NIC	10BaseT Ethrenet (CS8900A)

図 4 CAT68701
Fig. 4 CAT68701

次にカーネル層が提供している機能を挙げる．

- 仮想プロセッサ管理
  優先度ベースラウンドロビンスケジューリング
- メモリ管理
  可変長メモリブロック
- 同期排他制御
  Mutex，同期変数
- 割込み管理
  割込みハンドラ，多重割込み
- 名前空間管理
  名前空間の生成，エントリの登録

現在，仮想マシンとして Java のサブセットである Waba の移植を完了し，その他のシステムコンポーネントとして FAT16 ファイルシステム，UDP/IP プロトコルスタックを実装中である．また，アプリケーションとしては TriState の PIC マイコン搭載 NIC である PICNIC と UDP 通信を行ない，温度センサのデータを取得するものなどを実現した．

コードサイズはカーネル層が言語 C で約 5,500 行，アセンブラで 700 行であり，Waba 仮想マシン (以下，WabaVM) が言語 C で約 10,000 行である．仮想マシン間通信機構のベースには軽量継続ライブラリ cont[3] を SH3 に移植し，利用している．

表 4 にカーネル層だけの場合と，WabaVM を動作させた場合のメモリフットプリントを示す．

表 4 メモリフットプリント
Table 4 A Memory Footprint

システム構成	フットプリント
カーネル	22 KB
カーネル + WabaVM	60 KB

5.1 仮想化によるオーバヘッド

ネイティブ環境と WabaVM 上で同一の処理を行なった場合の性能比を **表 5** に示す．なお，CF read はコンパクトフラッシュを 1 セクタ (512 バイト) 読み込むのにかかった時間であり，file read は FAT のクラスタサイズ (2048 バイト) のファイルを読み込むのにかかった時間である．

JNI☆のオーバヘッドは約 60 μs であり，CF read の結果からさらに 100 μs がバイト配列の境界チェックなどに費やされている．この場合はハードウェアアクセスが遅いので，ほとんど性能差がない．一方，再帰関数呼び出しを多用するが，ハードウェアアクセスをともなわない竹内関数の性能比は約 430 倍であった．

☆ Waba は Java の規格としての JNI(Java Native Interface) をサポートしていないが，ネイティブメソッドを実行することはできる．便宜上，本稿では JNI と呼ぶことにする．

表 5 仮想化によるオーバヘッド
Table 5 An Overhead of Virtualization

	CF read	file open	file read	竹内関数
WabaVM	1.805 ms	11.72 ms	396.09 ms	1.244 μs
C VM	1.703 ms	1.72 ms	15.72 ms	2.914 ms
比率	1.06	6.81	25.20	426.9

5.2 ファイルシステムの実装

表 5 ではファイルの open と read にかかった時間を測定したが，さらに (1) 完全にネイティブなファイルシステムの実装，(2)JNI を利用しファイルシステムを呼び出す Waba アプリケーション，(3) 最終的なディスクアクセス以外はすべて Waba で実装されたファイルシステムの三つの仮想マシン環境で読込みサイズを変えて，その性能を比較した．結果を **図 5** に示す．

(1) と (2) の差は数 10 ms であり相対的にほとんど差がない．一方，(1) と (3) の性能比は約 25 倍であった．これは単純にバイトコードインタプリタのオーバヘッドであると考えられるので，JIT コンパイラを使った高速化が有効であると考えられる．

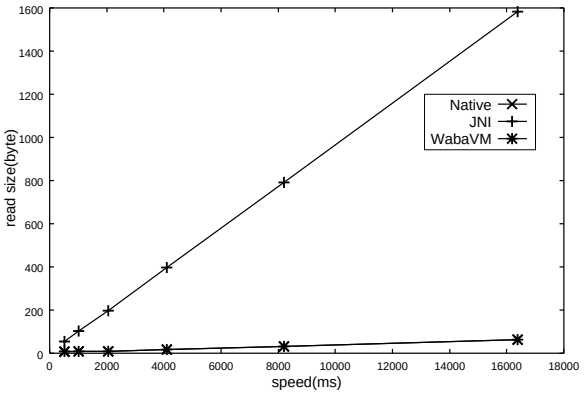

図 5 ファイルシステムの性能比較
Fig. 5 A Performance Comparison of File Systems

6. 関 連 研 究

6.1 仮想マシン

仮想マシンは一種の命令セットアーキテクチャを提供するものであり，実プロセッサの命令セットをそのまま提供する狭義の仮想マシンと，仮想的なプロセッサの命令セットを提供する広義の仮想マシンとに区別することができる．前者は VM/370 や VMWare のように特権命令と I/O 命令だけをエミュレートすることで実マシンを多重化するような仮想マシンであり，後者は Mach における Unix エミュレーションや，Java，Tcl スクリプトなどのインタプリタなどが該当する．UCSD Pascal のように中間言語である P-Code とランタイムであるスタックマシンを使うアプローチはプログラミング言語の分野を中心に利用されてきたが，最近は Java，Squeak，Inferno，.NET CRL などシステム抽象層としての利用が注目されている．

これらのシステムと同様に本システムにおける仮想マシンの定義も広義の仮想マシンである．

JavaVM の実行環境はベリファイア，セキュリティマネージャ，JIT コンパイラ，プロファイラ，インタプリタなどのサービスコンポーネントから構成される．上記の仮想マシンの実行環境はどれも単一の計算機上で実行されるモノリシックな構造であるが，Kimera[5] は仮想マシンの各コンポーネントをネットワーク上に分散することで，管理可能性，性能，安全性，スケーラビリティを改善している．このような分散仮想マシンはユビキタスコンピューティングなど，ユーザが性能，用途の異なる複数の計算機を利用し，それぞれがネットワークで接続される場合にも有効であると考えられる．

また，立野 [6] らは高い移植性，安全性，設定可能性を持つシステムモジュールの実現の一環として，Java で TCP/IP プロトコルスタックを実装している．Java でシステムモジュールを記述する場合は，メモリアドレスが不可視であり，アドレスポインタが使えないというシステム記述力，ハードウェアアクセスの手段が問題になるが，バイトストリームベースの高速なバッファ管理，スタックのマルチスレッド化によって実用的な性能を出せると主張している．これらの成果は我々の実装にも活かせると思われる．

6.2 OS 構成法と資源管理

Fluke[7] は OS に対する拡張を仮想マシンとして提供し，仮想マシンを再帰的に定義することでアプリケーションに適したシステムに拡張することができる．このような再帰的仮想マシンは，ハードウェアトラップをトランポリンするため，入れ子が深くなるとオーバヘッドが大きくなるが，Fluke ではトラップを使わずソフトウェアとして実現している．一方，本システムでは継続を利用することで実現している．

また，Mach 3.0 では割込みハンドラなどにおけるカーネルスタック利用のオーバヘッドを軽減するために継続を利用してる [8]．継続を利用することでカーネルスレッドとプロセス間通信の性能が向上したという結果が示されている．

RT-Mach における Resource Kernel[9] はリアルタイム OS における QoS 保証を実現するための資源予約機構である．Resource Kernel は Reserve と Resource Set という二つの抽象クラスを提供しており，前者は CPU サイクル，ディスク，メモリ，ネットワーク帯域など個々の資源予約を抽象化したものであり，これらの集合が Resource Set になる．ユーザプロセスはいずれかの Resource Set に属し，その資源管理ポリシに従ってスケジューリングされる．

7. おわりに

本稿では仮想マシンを用いた OS 構成法とその通信機構について述べた．本システムはカーネル層，仮想マシン層，アプリケーション層の三階層から構成され，カーネル層ではハードウェアアクセスの基本的なプリミティブを提供し，仮想マシン層は資源管理ポリシを提供する．システムコンポーネントは仮想マシン上で動作し，アプリケーションを任意の仮想マシン上で実行することができる．また，仮想マシン間を横断して処理を実行するアプリケーションに対して継続を用いた軽量なフロー制御を提供している．さらに，ネイティブ環境と WabaVM の双方の環境で FAT ファイルシステムを実装したところ，ファイル読込みの性能は約 25 倍になることがわかった．

今後の課題としてファイルシステムだけではなく，スケジューラ，ネットワークプロトコルなど OS の各コンポーネントを VM 層にて実装し，アプリケーションレベルも含めた性能，有用性についての評価を行なうことが挙げられる．また，本研究で利用した WabaVM は処理性能の追求ではなく，単純で移植性の高いシンプルな実装を目的としている．そこで，JIT コンパイルをサポートする仮想マシンを使用した実装も検討している．

参 考 文 献

1) 石口, 高野, 早川, 高橋, 仮想マシン機構を用いたオペレーティングシステムの実現, 情報処理学会第 64 回全国大会論文集, 4U-01, (2002)
2) 高野, 石口, 早川, 高橋, 仮想マシン機構を用いた拡張可能 OS におけるモジュール定義方式, 情報処理学会第 64 回全国大会論文集, 4U-02, (2002)
3) cont: lightweight continuation library, `http://todo.org/download/cont/`
4) K John Gough, Stacking them up: A Comparison of Virtual Machines, In Proceedings of ACSAC 2001, (2001)
5) Emin Gun Sirer, Robert Grimm, Arthur J. Gregory, Brian N. Bershad, Design and Implementation of a Distributed Virtual Machine for Networked Computers, In Proceedings of the 17th ACM SOSP, pp.202-216, (1999)
6) 立野広樹，萩野達也, Java による TCP/IP プロトコルスタックの設計と実装, 情報処理学会 OS 研究会報告，2000-OS-84, pp.99-106, (2000)
7) Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, Stephen Clawson, Microkernels Meet Recursive Virtual Machines, In Proceedings of the 2nd OSDI, pp.137-151, (1996)
8) Richard P. Draves, Brian N. Bershad, Richard F. Rashid, Randall W. Dean, Using Continuations to Implement Thread Management and Communication in Operating Systems, In Proceedings of the 13th ACM SOSP, pp.122-136, (1991)
9) Shuichi Oikawa, Ragunathan Rajkumar, Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior, In Proceedings of the IEEE Real-Time Technology and Applications Symposium, (1999)