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

2008－ARC－176 (4)
2008－EMB－ 7 (4)
2008／1／15

マルチプロセッサRTOS対応コシミュレータ

古川貴士 柴田誠也 本田晋也 冨山宏之 高田広章
名古屋大学 大学院情報科学研究科
Email: { furukawa, shibata, honda, tomiyama, hiro } @ertl.jp

概要 本研究では，シングルプロセッサ用ISSを拡張し，マルチプロセッサRTOSに対応したコシミュレータを実現した．本コシミュレータでは，複数のISSとハードウェアシミュレータを同時に起動・接続し，各ISSが一つのプロセッサをシミュレーションすることで，マルチプロセッサシステムのコシミュレーションを行う．本コシミュレータを用いて，MPEGデコーダのコシミュレーション実験と，ISSの拡張によるオーバヘッドの測定を行った．

A Cosimulator with Multiprocessor RTOS

Takashi Furukawa Seiya Shibata Shinya Honda Hiroyuki Tomiyama Hiroaki Takada
Dept. of Information Engineering, Graduate School of Information Science, Nagoya Univ.
Email: { furukawa, shibata, honda, tomiyama, hiro } @ertl.jp

Abstract This paper presents a cosimulator with multiple ISSs where a multiprocessor embedded RTOS can be executed. The ISS was extended from an ISS for single processor embedded systems. In our cosimulator, multiple ISSs and hardware simulators can be executed concurrently with communication. Each of the ISSs simulates one processor. Using the cosimulator, we performed cosimulation of MPEG decoding system, and evaluated the overhead due to extension of the ISS.

1 はじめに

近年大規模化・複雑化が進む組込みシステムにおいて，RTOSの利用が重要である．近年，高性能・低消費電力を実現するために，組込みシステムにおいてもマルチプロセッサの利用が広がっている．現在組込みマルチプロセッサシステムのソフトウェア開発では，シングルプロセッサ用のRTOSを利用している場合が多いが，より効率的な開発のためにはマルチプロセッサ用RTOSの利用が必要である．また，マルチプロセッサシステムは実機を使ったデバッグ環境が貧弱である．そのため，マルチプロセッサRTOSに対応したコシミュレータによるデバッグ環境が求められている．

我々はこれまでに，RTOSシミュレータを使ったコシミュレータ [1] [2] を開発した．このコシミュレータでは，シングルプロセッサ用RTOSを利用したマルチプロセッサシステムのコシミュレーションが可能である．RTOSシミュレータは，命令セットシミュレータ(ISS)と違ってネイティブで動作するため非常に高速なシミュレータであるが，シングルプロセッサ用RTOSのシミュレーションモデルを用いているため，マルチプロセッサRTOSはシミュレーションできない．

本研究では，複数のISSとハードウェアシミュレータを接続し，マルチプロセッサRTOSを適用することで，マルチプロセッサRTOS対応コシミュレータを実現した．適用したRTOSは，我々がこれまでに開発した *TOPPERS/FDMP* カーネルと呼ばれるRTOSである．ISSはシングルプロセッサ用ISSを拡張して利用した．本コシミュレータでは，シミュレータ間接続のためにWindowsの標準RPC（Remote Procedure Call）を利用している．これにより，ハードウェア記述言語（HDL）用のシミュレータや，最近普及が進んでいるSystemC言語用のシミュレータなど，様々なシミュレータが接続可能である．また，複数のISSでマルチプロセッサをシミュレーションするため，ホスト計算機もマルチプロセッサである場合は，シミュレーション負荷の分散によって高速なシミュレーションが可能である．

本論文の構成は以下の通りである．まず2章で過去に開発したシングルプロセッサ用RTOSのコシミュレータを紹介する．3章ではマルチプロセッサRTOSについて述べる．4章でマルチプロセッサRTOSシミュレータの設計手法を比較し，5章で開発したマルチプロセッサRTOS対応コシミュレータについて説明する．6章では本コシミュレータを用いた実験を述べる．最後に7章でまとめと今後の課題を述べる．

2 シングルプロセッサRTOSシミュレータ

過去に開発したコシミュレータの構成を図1に示す．

本コシミュレータは論理回路やセンサ，ディスプレイ等の物理デバイスをシミュレーションするハードウェアシミュレータ，ソフトウェアとRTOSをシミュレーションするRTOSシミュレータ，及び，ハードウェアシミュレータ/RTOSシミュレータ間を接続し，通信を実現するデバイスマネージャから構成される．

2.1 デバイスマネージャ

デバイスマネージャは，シミュレータ間のRPC通信を仲介し，リード/ライト要求，及び，割り込み要

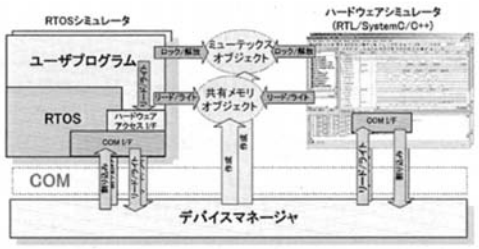

図 1　シングルプロセッサ用 RTOS を利用したマルチプロセッサシステムシミュレータ

求を実現する．各シミュレータは起動後にデバイスマネージャに接続する．本コシミュレータではメモリマップト I/O を前提としていて，各ハードウェアシミュレータには固有のアドレス空間がマッピングされる．RTOS シミュレータは，アドレスを指定することでハードウェアシミュレータにアクセスする．同様に各 RTOS シミュレータには固有のプロセッサ ID がマッピングされ，各シミュレータは，プロセッサ ID を指定することで RTOS シミュレータに割込み要求を送信する．アドレス空間，及びプロセッサ ID のマッピングはデバイスマネージャが管理し，シミュレータからの RPC 呼出しがあった時に，操作すべき適切なシミュレータの RPC を呼び出す．各シミュレータはデバイスマネージャへの接続後に，自分のアドレス空間またはプロセッサ ID をデバイスマネージャに知らせる．

デバイスマネージャは別の機能として，共有メモリもサポートしている．各シミュレータは，ホスト OS の共有メモリ機能を利用することでターゲットシステム上の共有メモリをシミュレーションする．ホスト OS 上の共有メモリは，シミュレータの要求に応じてデバイスマネージャが作成する．

さらに，デバイスマネージャは RTOS シミュレータ間の排他制御のためのロック機構もサポートしている．RTOS シミュレータは，ホスト OS のミューテックス機能を利用することで排他制御を実現する．ホスト OS 上のミューテックスは，RTOS シミュレータからの要求に応じてデバイスマネージャが作成する．

シミュレータ/デバイスマネージャ間の通信を実現するために，*COM* (*Component Object Model*) と呼ばれる汎用的な RPC 機構を使用する．これにより，ハードウェア記述言語 (HDL) 用のシミュレータや，SystemC 言語用のシミュレータなど，様々なシミュレータが接続可能である．

2.2　RTOS シミュレータ

RTOS シミュレータはユーザプログラムと共にホスト計算機上のネイティブコードにコンパイルされ，高速に動作する．本 RTOS シミュレータは，*μITRON*4.0 仕様 [3] [4] に準拠している．また，本 RTOS シミュレータは文献 [5] で定められたデバイスアクセス用

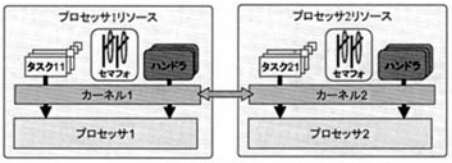

図 2　オブジェクト管理の概念図

API をサポートしている．この API を用いてアドレスを指定することにより，ソフトウェアはデバイスマネージャを介してハードウェアシミュレータにアクセスする．本 RTOS シミュレータではこれらの API をサポートしているため，シミュレーション用にユーザプログラムを変更する必要がない．ただし，本 RTOS シミュレータがサポートしている RTOS はシングルプロセッサ用 RTOS である．

3　TOPPERS/FDMP カーネル

3.1　概要

組込みシステムのソフトウェア開発においては，リアルタイム OS が用いられることが多い．そのため，マルチプロセッサを用いて組込みシステムを構築する際には，マルチプロセッサ上でのシステム開発をサポートするリアルタイム OS が必要になる．

そこで我々は，ITRON 仕様をマルチプロセッサ向けに拡張し (以下，拡張仕様と呼ぶ)，TOPPERS/FDMP カーネル (以下，FDMP カーネルと呼ぶ) として実装した．

拡張仕様は，μITRON4.0 仕様のスタンダードプロファイルをベースにしており，プロセッサを跨いで通常の μITRON4.0 仕様のシステムコールを実行することが可能である．

タスクやセマフォなどのカーネルオブジェクトはいずれかのプロセッサに属する．タスク等の処理単位となるオブジェクトはそれが属するプロセッサでのみ実行される．すなわち，あるタスクがどのプロセッサで実行されるかは静的にのみ決定され，システム動作中にタスクが実行されるプロセッサが変わることはない．オブジェクト管理の概念図を図 2 に示す．

3.2　システムコールの実現方法

プロセッサを跨ぐシステムコールの実現方法を図 3 に示す．FDMP カーネルでは，プロセッサ毎にそのプロセッサに所属するカーネルオブジェクトの管理ブロックを持つ．そして，それらの管理ブロックを配置するメモリを共有していて，お互いのプロセッサの管理ブロックを直接操作する (直接操作法)．

例えば，図 3 で，プロセッサ 1 のタスク A がプロセッサ 2 のタスク B を起動するシステムコールを発行すると，図中の ①に示したように，プロセッサ 1 は，タスク B の状態を管理するデータ構造をから直接アク

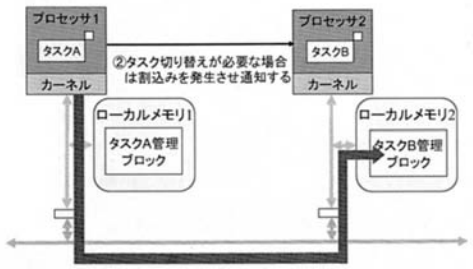

図 3　システムコールの実行例

セスして起動状態とする．タスク B を起動した結果，プロセッサ 2 でのタスク切替えが必要になった場合には，プロセッサ 1 からプロセッサ 2 に割込みをかけ，タスク切替えを依頼する（図中の ②）．

3.3　ハードウェア要件

FDMP カーネルでは，直接操作法でシステムコールを実現しているため，FDMP カーネルを動作させるには，以下のアーキテクチャのマルチプロセッサシステムが必要となる．

(1) 各プロセッサのデータを置くメモリが全てのプロセッサから同一のアドレスでアクセス可能である
(2) 任意のプロセッサに割込み（プロセッサ間割込み）を発生可能である
(3) プロセッサ間での排他制御のための機構を持つ
(4) プロセッサの識別機構を持つ（プロセッサ ID）

4　マルチプロセッサ RTOS シミュレータの設計方針

本研究の目的は FDMP カーネルを使ったマルチプロセッサシステムのシミュレータを実現することである．実現方法として，以下の 3 つの手法が考えられる．

(1) RTOS シミュレータを作成する手法
(2) マルチプロセッサ対応 ISS を利用する手法
(3) 複数の ISS を接続する手法

(1) の RTOS シミュレータを用意する手法は，ホスト計算機上で直接実行可能なため非常に高速に動作するという利点がある．

まず，複数プロセスでシミュレーションすることを考える．FDMP カーネルでは，直接操作法を採用していることからメモリ上のタスク情報等を共有する必要がある．このタスク情報は静的にリンクされる．タスク情報をホスト OS の共有メモリ上に配置することで共有することが可能であるが，共有メモリの仮想アドレスはホスト OS が動的に設定する．そのため，静的なリンクを行うことができない．

一方，単一プロセスでシミュレーションする場合を考える．本来の FDMP カーネルはプロセッサ毎に独立にコンパイル・リンクするため，異なるプロセッサ上のユーザプログラムで同じ名前の変数や関数を別々に扱う．そのため，一つのアプリケーションとして RTOS シミュレータを生成することができない．

(2)，(3) の ISS を利用する手法は RTOS シミュレータでの実現に比べて，ターゲットプロセッサのバイナリファイルを使用するため，命令単位のトレースを取ることが可能である．

(2) のマルチプロセッサに対応した単一の ISS でシミュレーションする場合，プロセッサ間でのタイミングの同期を取りやすいが，プロセッサ数にほぼ比例してシミュレーション時間が増大するという特徴がある．

一方，(3) の複数の ISS でマルチプロセッサをシミュレーションする場合，ホスト計算機がマルチプロセッサであればシミュレーション負荷の分散によって比較的高速なシミュレーションが可能である．その反面，ISS 間でのタイミングがずれるため，正確なシミュレーションは不可能である．

これらの手法を比較した結果，複数の ISS でマルチプロセッサをシミュレーションする手法を採用した．

5　マルチプロセッサ RTOS シミュレータ

5.1　概要

我々は，マルチプロセッサ RTOS に対応したコシミュレータを実現するために，ISS を用いた．開発したコシミュレータの構成を図 4 に示す．

本コシミュレータは，Instruction Set Simulator (ISS)，ハードウェアシミュレータ，及び，デバイスマネージャから構成される．マルチプロセッサのシミュレーションのために複数の ISS を起動し，各 ISS が一つのプロセッサをシミュレーションする．ISS 間，ISS/ハードウェアシミュレータ間の通信は，デバイスマネージャを介した RPC 通信，及び，ホスト OS の提供する共有メモリ，ミューテックス機能により実現する．シミュレータ間の接続や，共有メモリ，ミューテックスは，デバイスマネージャにより管理されている．

5.2　SkyEye

本コシミュレータでは ISS として，シングルプロセッサ用 ISS である *SkyEye*[6] [7] を拡張して利用した．SkyEye は GDB/ARMulator から派生した ARM アーキテクチャの ISS である．SkyEye は，GDB デバッグインタフェースをサポートしており，アプリケーションの GDB によるデバッグが可能である．また，SkyEye は高速なシミュレーションのために，バイナリトランスレーションをサポートしている．

SkyEye ではシミュレータターゲットの構成を柔軟に変更することが可能である．具体的には，メモリの構成，プロセッサ ID，ペリフェラル構成等を変更可能である．構成はコンフィギュレーションファイルに記述する．コンフィギュレーションファイルの例を図 5 に示す．設定可能な項目の詳細については，後述する．

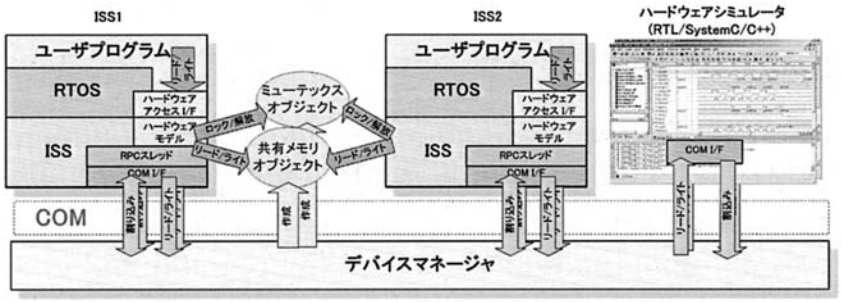

図 4　マルチプロセッサ RTOS に対応したマルチプロセッサシステムシミュレータ

5.3　デバイスマネージャとの接続

デバイスマネージャとの接続のためにメインスレッドの他に RPC スレッドを追加した．メインスレッドから必要に応じて RPC スレッドに要求を渡し，RPC スレッドがデバイスマネージャの RPC を呼び出すことで，他のシミュレータとの通信を実現している．

メインスレッドは，グローバル変数とホスト OS の提供するミューテックスを利用することで，RPC スレッドに要求を渡す．具体的には，RPC スレッドは基本的にミューテックスか RPC 呼出し待ち状態になっていて，メインスレッドがグローバル変数に要求メッセージを書き込み，ミューテックスを解放する．これによって RPC スレッドが待ち状態を抜け，要求メッセージを読んで RPC 通信を行う．RPC 通信を終えた RPC スレッドは再び待ち状態に遷移する．

5.4　プロセッサ ID

各 SkyEye にはプロセッサ ID が割り当てられている．プロセッサ ID は，コンフィギュレーションファイル内で指定する．

```
 1 mem_bank: map=M, type=RW, addr=0x00000000,
    size=0x00100000
 2 mem_bank: map=S, type=RW, addr=0x80001000,
    size=0x00001000, com_addr=0x90000000
 3 mem_bank: map=I, type=RW, addr=0xf0000000,
    size=0x10000000
 4 com_mutex_cont: base=0xffffff00
 5 com_mutex_obj: id=0
 6 com_mutex_obj: id=1
 7 com_int_id: 0x01
 8 com_int_out: base=0xffffff40
 9 com_int_in: subid=0x01, IRQ=0
10 com_int_in: subid=0x00, IRQ=1
11 prc_id: base=0xffffffa0, id=1
12 prc_affinity: id=0
13 perf_time: base=0xffffff80
14 cycle_counter: base=0xffffffc0
```

図 5　コンフィギュレーションファイルの例

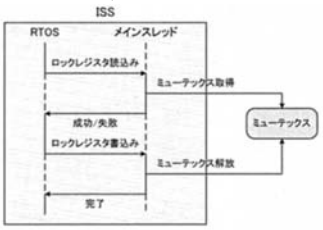

図 6　ロックシーケンス

ハードウェア要件 (4) に示すように，ISS 上で動作する RTOS では，自分がどのプロセッサ（ISS）で動作しているか判断する必要がある．そのため，プロセッサ ID を格納する専用レジスタ（プロセッサ ID レジスタ）を ISS に追加することで，RTOS から自分が動作している ISS のプロセッサ ID の取得を可能とした．

プロセッサ ID とプロセッサ ID レジスタのアドレスはコンフィギュレーションファイルに記述する．図 5 の例では 11 行目でレジスタアドレスを 0xffffffa0 に，プロセッサ ID を 1 に設定している．RTOS は，該当アドレスをリードすることでプロセッサ ID を取得可能である．

5.5　ロック機構

ハードウェア要件 (3) で述べたロック機構をサポートするため，ISS 間でのロック機能をサポートして RTOS 側に提供する．ISS 間のロック機能は，ホスト OS の提供するミューテックス機能により実現する．

ロックのシーケンス図を図 6 に示す．シミュレーション開始時に，ISS からデバイスマネージャにミューテックスオブジェクト作成要求を出し，デバイスマネージャが作成して，そのハンドルを ISS に渡して ISS がオープンする．シミュレーション中は，コンフィギュレーションファイルで指定したロック用のアドレスを RTOS がリード/ライトすることで，ISS がロック/アンロックする．ロックに失敗した時は，解放待ちにならずに直ちに結果を RTOS に返し，RTOS 側でポーリングによってロックが完了するまで待つか，他のタスクに切り替える等の処理を行う．

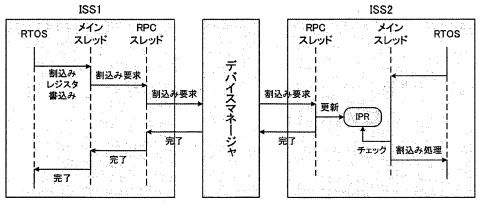

図 7　プロセッサ間割込みシーケンス

図 5 のコンフィギュレーションファイル記述例では，4 行目でロック用のアドレスを 0xffffff00 に設定し，5，6 行目で ID 0 と 1 のミューテックスを使用することを指定している．

5.6　プロセッサ間割込み

ハードウェア要件 (2) のプロセッサ間割込みは，デバイスマネージャを介した RPC 通信により実現する．プロセッサ間割込みのシーケンスを図 7 に示す．

プロセッサ間割込みを RTOS から使用するため，割込みレジスタを ISS に追加した．RTOS は割込み出力レジスタの上位 16 ビットにプロセッサ ID を，下位 16 ビットに割込み番号を書き込むことで，ISS に対して割込み要求を出す．

ISS のメインスレッドは，割込み出力レジスタへ書き込みがあると，RPC スレッドに割込み要求を送り，RPC スレッドはデバイスマネージャに対して，割込み要求用 RPC を呼び出す．

デバイスマネージャは，指定されたプロセッサ ID に対応する ISS の割込み要求用 RPC を呼び出す．割込み要求用 RPC を呼び出された RPC スレッドは，割込み番号に従って ISS 上の割込みペンディングレジスタの該当ビットをセットする．割込みペンディングレジスタはメインスレッドからも変更されるため，変更時に排他制御を行う．

割込み要求を受け取った ISS のメインスレッドは，現在実行中の命令を完了した後に割込みペンディングレジスタをチェックして，割込みが許可されていれば割込み処理を行う．

図 5 の例では，7 行目でプロセッサ ID を 1 に，8 行目で割込み出力レジスタのアドレスを 0xffffff40 に，9，10 行目で割込み番号 1，0 を外部割込み 0，1 にそれぞれ設定している．

5.7　メモリ構成

マルチプロセッサシステムのメモリは，全てのプロセッサからアクセス可能な共有メモリと，特定のプロセッサからアクセス可能な非共有メモリに分類することができる．さらに，ハードウェア要件 (1) より，共有メモリは，どのプロセッサからも同じアドレスとして参照できる必要がある．

非共有メモリを ISS のローカルメモリに，共有メモリをホスト OS の共有メモリより実現することで，前述のメモリ構成に対応したシミュレーションを実現する．

これらのメモリ構成は，図 5 の 1，2 行目のようにコンフィギュレーションファイルに記述する．map=M はローカルメモリを表し，map=S は共有メモリを表す．この例ではアドレス 0x00000000 から 0x00100000 バイトをローカルメモリに，アドレス 0x80001000 から 0x00001000 バイトを共有メモリにマッピングしている．com_addr はデバイスマネージャに共有メモリ作成要求を出す時に指定するアドレスであり，このアドレスを統一することで ISS 間でリニアなメモリアドレスを管理することができる．

5.8　性能評価機能

性能評価機能用に，シミュレータ上で計算するターゲットのサイクル数を保持するレジスタ（サイクル数レジスタ）を ISS に追加した．ARM マニュアルに記載されている命令毎にかかるメモリサイクル数の計算式を用いてサイクル数を計算する．ソフトウェアはコンフィギュレーションファイルで指定したサイクル数レジスタをリードすることで，サイクル数を取得可能である．

図 5 の例では，14 行目でサイクル数レジスタのアドレスを 0xffffffc0 に設定している．

5.9　実時間計測機能

性能評価機能用に，ホスト計算機を起動してからの経過時間を保持するレジスタを ISS に追加した．経過時間はクロック周波数とホスト計算機が起動してからのクロック数から計算し，クロック周波数は ISS 起動時に 1 秒間のクロック数を計測して求める．ただし，ホスト計算機が起動してからのクロック数がプロセッサ毎に異なる可能性があるため，実行するホストプロセッサを ISS 毎に固定する．プロセッサ ID 同様，コンフィギュレーションファイルに指定したアドレスをリードすることで実時間を取得可能である．図 5 の例では，12 行目でホスト計算機のプロセッサ 0 で実行することを，経過時間レジスタのアドレスを 0xffffff80 に，それぞれ設定している．

5.10　直接操作法の実現

3.2 節で述べたように，FDMP カーネルではプロセッサを跨ぐシステムコールは管理ブロックを直接操作して行う．管理ブロックは排他的に操作する必要があるため，ロック機構を利用する．RTOS はロックレジスタをリードし，ISS はミューテックスを取得する．その後，共有メモリ上の管理ブロックを操作する．管理ブロックの操作が完了した後は，ミューテックスを解放する．さらに，タスク切替えが必要な場合は，RPC 通信による割込みを発生させて割込みペンディングレジスタを更新する．

プロセッサ間割込みは RPC 通信により実現するため，オーバヘッドが大きい．それに比べて，ロックや共有メモリはホスト OS のサポートするオブジェクトに直接操作するため，オーバヘッドが小さい．しかし，プロセッサ間割込みは起こる頻度が少ないため，RPC 通信によるオーバヘッドはあまり問題にならない．

表 1　拡張前後の ISS の実行時間比較

シミュレータ	画像 1 の処理	画像 2 の処理
拡張前の ISS	14.93（秒）	17.17（秒）
拡張後の ISS	14.96（秒）	17.20（秒）

6　評価

開発したコシミュレータを使って，MPEG デコーダのコシミュレーション実験と，元の ISS と拡張した ISS での性能比較実験を行った．実験は，Intel2.66GHz プロセッサコアを 4 個搭載した計算機で行った．使用 OS は WindowsXP である．ただし，Intel プロセッサの動的なクロック周波数切替え機能は無効にした．

6.1　性能比較

ISS 拡張前と拡張後のシミュレーション速度の比較を行った．ただし，シミュレーション時間の計測のために，オリジナルの ISS に時間計測用のルーチンを追加したものを拡張前の ISS とする．共有メモリやロック機能を使用しない同一のソフトウェアを両シミュレータに適用し，コンフィギュレーションファイルも同一のものを用いた．そのため今回の拡張によるオーバヘッドは，デバイスマネージャとの接続（デバイスマネージャのバックグラウンド実行）と RPC 用スレッドの実行，及び，割込みペンディングレジスタ書込み時の排他制御処理のみである．そのため，シミュレーション速度はあまり変わらないことが予想される．

拡張前後の ISS を用いて，JPEG デコーダのシミュレーションにかかる時間を測定した．対象システムはプロセッサ数 1 で専用回路はないものとした．2 つの画像に対する実験結果を表 1 に示す．拡張によるオーバヘッドは約 0.2 %であった．

6.2　MPEG デコーダによる評価

MPEG デコーダについて，ターゲットプロセッサ数が 1 個の場合と 2 個の場合，さらにそれぞれについて IDCT 専用ハードウェアがある場合とない場合の，合計 4 種類のシステム構成でシミュレーションを行った．ハードウェアは HDL で記述し，HDL シミュレータを利用した．実行結果を表 2 に示す．実行結果から，プロセッサ数を 2 個にした方がプロセッサ数 1 個の場合よりシミュレーション時間が短い事が分かった．プロセッサ数を増やすと，通常の ISS ではプロセッサ数にほぼ比例してシミュレーション時間が長くなるが，本コシミュレータでは ISS 毎に処理を分散させる事が可能であるため，一つの ISS での処理が少なくなる分シミュレーション時間が短くなったと考えられる．

また，ターゲットプロセッサ数 1 で専用回路がないシステムに関して，ISS でバイナリトランスレーション (BT) を行った場合と行わなかった場合，さらに RTOS シミュレータを用いた場合とでシミュレーションを行い，実行時間の比較を行った．実行結果を表 3 に示す．バイナリトランスレーションを使うことによって，シミュレーション時間が 16.7 %短縮した．

表 2　MPEG デコーダのシミュレーション実行時間

システム構成	実行時間
1 コア全 SW	288.7（秒）
1 コア +HW（IDCT）	586.6（秒）
2 コア全 SW	256.5（秒）
2 コア +HW（IDCT）	562.6（秒）

表 3　各シミュレータのシミュレーション実行時間

シミュレータ	実行時間
ISS（BT なし）	288.7（秒）
ISS（BT あり）	240.5（秒）
RTOS シミュレータ	4.1（秒）

7　おわりに

本研究ではマルチプロセッサ RTOS に対応したコシミュレータを実現した．本コシミュレータでは，複数の ISS を使うことによりホスト計算機上での負荷分散を可能にした．評価実験から本コシミュレータの有用性（と負荷分散の効果）を示した．また，ISS を拡張したことによるオーバヘッドは約 0.2 %であった．

今後の課題としては，ISS 間での時間同期を取ることが挙げられる．

参考文献

[1] S. Honda, T. Wakabayashi, H. Tomiyama, and H. Takada. “RTOS-Centric Cosimulator for Embedded System Design”. *IEICE Trans. Fundamentals*, vol. E87-A, no. 12:pages 3030–3035, Dec. 2004.

[2] T. Furukawa, S. Honda, H. Tomiyama, and H. Takada. “A Hardware/Software Cosimulator with RTOS Supports for Multiprocessor Embedded Systems”. In *Proc. of ICESS*, pages 283–294, May. 2007.

[3] ITRON プロジェクト. `http://ertl.jp/ITRON/`.

[4] H. Takada and K. Sakamura. “μITRON for Small-Scale Embedded Systems”. *IEEE Micro*, vol. 15, no. 6:pages 46–54, Dec. 1995.

[5] μITRON4.0 仕様研究会 デバイスドライバ設計ガイドライン. `http://www.ertl.jp/ITRON/GUIDE/device-j.html`.

[6] SkyEye. `http://www.skyeye.org/`.

[7] S. Kang, H. Wang, Y. Chen, X. Wang, and Y. Dai. “Skyeye: An Instruction Simulator with Energy Awareness”. In *Proc. of ICESS*, pages 456–461, Dec. 2004.