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

2008－SLDM－134／2008－EMB－8（3）
2008 / 3 / 27

マルチプロセッサRTOS対応シミュレーション環境の機能拡張と効率化

相庭 裕史† 柴田 誠也†† 古川 貴士††
本田 晋也†† 冨山 宏之†† 高田 広章††

† 名古屋大学 工学部電気電子・情報工学科情報工学コース
†† 名古屋大学 大学院情報科学研究科情報システム学専攻
E-mail: †{h-aiba,shibata,furukawa,honda,tomiyama,hiro}@ertl.jp

あらまし 我々は，マルチプロセッサシミュレーション環境の精度の向上と高速化を行った．本研究で対象としたマルチプロセッサシミュレーション環境は，シングルプロセッサ用の命令セットシミュレータ（ISS）を連携させてマルチプロセッサのシミュレーションを実現している．各ISSは異なるアプリケーションであるため，同期していない．本研究では，任意のタイミングで各ISSを同期させることにより，シミュレーション精度の向上を実現した．また，ISS上で実行するRTOSのアイドル処理を活用したシミュレーション時間の短縮手法を提案した．MPEG4デコーダのシミュレーションを1000サイクルの同期で実行した結果，40%のシミュレーション時間削減を実現した．
キーワード シミュレーション, マルチプロセッサ, RTOS

Function and Efficiency Enhancement of A Simulation Environment with Multiprocessor RTOS

Hiroshi AIBA†, Seiya SHIBATA††, Takashi FURUKAWA††, Shinya HONDA††,
Hiroyuki TOMIYAMA††, and Hiroaki TAKADA††

† Dept. of Information Engineering, Nagoya Univ.
†† Graduate School of Information Science, Nagoya Univ.
E-mail: †{h-aiba,shibata,furukawa,honda,tomiyama,hiro}@ertl.jp

Abstract This paper presents two improvement techniques which we have applied for our multiprocessor simulation environment with multiple ISSs. In this simulation environment, Multiple ISSs are executed without synchronization. In order to validate the timing correctness of communications performed between ISSs, first improvement technique provides cycle level synchronization mechanisms to ISSs. Another one reduces simulation time by using host CPU efficiently, considering RTOS's idle state. When synchronizing ISSs every 1000 cycles, this improvement reduced simulation time of MPEG4 decoder by 40%.
Key words simulation, multiprocessor, RTOS

1. はじめに

近年，大規模・複雑化する組込みシステム開発において，発生頻度の低い現象の再現が難しいなどの理由から，実機検証が困難となっている．このような問題を解決する方法の1つが，シミュレータを使った検証である．また，組込みシステムでは，性能向上・消費電力削減等の要求から，マルチプロセッサ化が進んでおり，マルチプロセッサ用のリアルタイムOS（以下，RTOS）の使用が広がりつつある．したがって，マルチプロセッサRTOSに対応したシミュレータが必要となっている．我々はこれまでに複数の命令セットシミュレータ（以下，ISS）と，シミュレータ間の通信を仲介するデバイスマネージャというツールにより，マルチプロセッサRTOS対応シミュレーション環境を開発した[1]．図1は，このシミュレーション環境の概要図である．デバイスマネージャには，様々なハードウェアシミュレータを接続することができ，ハードウェア/ソフトウェア・コシミュレーション環境を構築することも可能である．

本シミュレーション環境では，各ISSは独立したアプリケーションとして動作しているため，ISSが同期していないという問題があった．そこで，本研究では各ISSが保持する実行サ

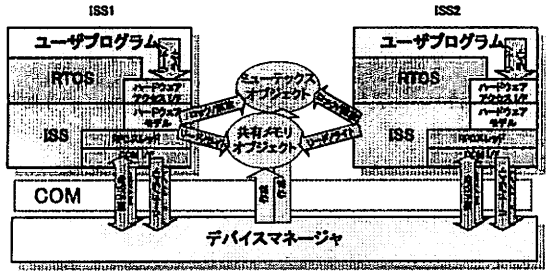


図 1 マルチプロセッサ RTOS 対応シミュレーション環境

イクル数により ISS を同期させ，タイミング精度の向上を図る．ISS をサイクル数で同期させることにより，より実機に近いシミュレーションが実現できる．加えて，実行ログからの協調動作解析や，デバッグが容易になり，設計生産性の向上につながる．

本研究ではさらに，ISS 上で動作する RTOS のアイドル処理を活用したシミュレーション時間の短縮手法を提案する．ISS 自体の高速化手法はこれまでにいくつか提案されているが [2]，本研究では RTOS の利用を前提としたシミュレーション時間短縮手法を提案する．本シミュレーション環境では，ISS 上で，ターゲットプロセッサ用にコンパイルされたマルチプロセッサ RTOS を実行する．RTOS は，実行すべきタスクが無い場合，割込みが発生するまで繰り返し処理により待つ．このときに実行される命令は，ユーザプログラムにとって無意味であるため，機能レベルの検証では省くことが可能である．そこで本研究では，RTOS アイドル時のシミュレーション実行を省略し，その間，他の ISS にホスト CPU を譲ることでホスト CPU を効率的に利用する方法を提案する．

本論文の構成は以下のとおりである．2 章では，我々がこれまでに開発した，マルチプロセッサ RTOS 対応シミュレーション環境について説明する．3 章では本研究で行った ISS 間の同期について述べ，4 章でホスト CPU の効率的利用方法について述べる．5 章では 3 章，4 章の改良の影響と効果を調べるために行った実験について説明する．6 章で本論文のまとめを行う．

2. マルチプロセッサ RTOS 対応シミュレーション環境

本章では研究対象としたシミュレーション環境に関して，用いた ISS の詳細と，マルチプロセッサ RTOS への対応に必要な機能について説明する．

本シミュレーション環境は複数の ISS とシミュレータ間の通信を仲介するデバイスマネージャから構成される．デバイスマネージャには，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の接続も可能である．ISS 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．

2.1 ISS

本シミュレーション環境では ISS として，シングルプロセッサ用 ISS である SkyEye [4] を拡張して利用している．SkyEye は GDB/ARMulator をベースとしたオープンソースのシミュレータであり，ARM アーキテクチャ用のオブジェクトファイルを 1 命令毎に読み込み，デコードしながらシミュレーションを行う．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 によるデバッグを行うことができる．実行命令やメモリアクセスの履歴をとることも可能である．メモリ構成など，シミュレーションターゲットの構成は，設定ファイルにより柔軟に変更することができる．ISS は命令のデコード・実行を行うメインスレッドとデバイスマネージャとの通信を実現するための，RPC スレッドにより構成されている．

2.2 共有メモリ

マルチプロセッサ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プロセッサからアクセス可能な非共有メモリに大別される．本環境では非共有メモリを ISS のローカルメモリに，共有メモリをホスト OS の共有メモリにより実現している．ISS は起動時にデバイスマネージャに共有メモリ作成要求を出し，デバイスマネージャが作成して，そのハンドルを ISS にわたして ISS がオープンする．

2.3 排他制御

マルチプロセッサ RTOS の必要とするロック機構は，ホスト OS の提供するミューテックス機能により実現されている．ミューテックスは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 にわたして ISS がオープンする．ISS がミューテックスの取得に失敗した場合，開放待ちにならずにすぐに RTOS に制御を返す．RTOS はポーリングによりロックが完了するまで待つなどの処理を行う．

2.4 プロセッサ間割込み

プロセッサ間割込みはデバイスマネージャを介した RPC 通信により実現されている．RTOS は割込み出力レジスタにライトすることで 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に書き込みがあると，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 を呼び出すことで，その ISS に割込みを発生させる．

2.5 プロセッサ間の同期

本環境では機能検証を目的としている．プロセッサ間でサイクルレベルの同期はとっておらず，複数ある SkyEye 間で時間の進み方が異なる．

3. ISS 間の同期

本章では，本研究で行った ISS 間のサイクルレベルの同期について説明する．

本シミュレーション環境では単一プロセスではなく，複数の ISS を用いることでマルチプロセッサのシミュレーションを実現している．そのため，ホスト計算機がマルチプロセッサの場

合には負荷分散により高速なシミュレーションが可能である．しかし，各 ISS が独立したアプリケーションとして動作するため，ISS が同期していないという問題があった．そこで，本研究では ISS をサイクル数により同期させ，シミュレーション精度の向上を図る．

3.1 同期の利点

ISS 間で同期をとることには，3 つの利点がある．第一に，実行サイクル数をある程度の精度で計測できるようになる．ISS は内部で実行サイクル数を保持しており，その値をシステムの評価指標として用いることができる．このとき，ISS 間で実行サイクル数の差が小さければ，より正確な値を得ることができる．第二に，デバッグが容易になるという利点がある．今回 ISS として用いた SkyEye には GDB でのデバッグが可能である．GDB によるデバッグ時に，1 つの ISS 上のプログラムを停止させた場合，全ての ISS の動作を停止させることが可能になる．第三に，実行ログによる ISS 間の協調動作解析が容易になる．これまでは複数の ISS の実行ログを解析するとき，ある時点において，それぞれどのコードを処理していたのか調べることが困難であった．しかし，ISS を同期させることで，サイクル数をもとに同時刻における ISS の実行コードを推測することが可能となる．

3.2 実現方針

本研究では，各 ISS の保持する実行サイクル数を任意の間隔で一致させることで ISS 間の同期を実現した．シミュレーション精度の観点からは，1 サイクル毎に同期することが望ましい．しかし，オーバヘッドが膨大となることが予想されるため，任意サイクル数で全ての ISS が待ち合わせる，バリア同期手法を採用した．

同期機構にはまず，COM を用いた方法を提案した．COM とは，マイクロソフトが提唱するソフトウェアの再利用を目的とした技術である．COM を用いることで言語に関係なく，互いに通信が可能である．そのため，この方法では，様々な言語で記述されたシミュレータとの同期が実現できる．しかし通信に大きなオーバヘッドがかかる．そこで，オーバヘッドの小さいホスト OS の共有オブジェクトを用いる方法を提案した．

3.3 COM を用いた同期方法

各 ISS は起動時に，同期に参加する ISS の数と，同期サイクル数をデバイスマネージャに通知する．ISS 上のプログラムが動作を開始し，設定したサイクル数分のコードを実行したときに，デバイスマネージャにそのことを通知し，待ち状態に入る．デバイスマネージャは通知してきた ISS の数と始めに設定した ISS の数を比較し，同じであれば全ての ISS の待ち状態を解除する．そして再び次の同期地点までシミュレーションを開始する．デバイスマネージャと ISS 間の通信には COM 通信を用いる．図 2 は 2 つの ISS の同期の流れを示している．ISS1 は先に同期地点に到着し，待ち状態になる．ISS2 が同期地点に到着し，デバイスマネージャにそのことを通知すると，デバイスマネージャが内部でもつカウンタの値から，全ての ISS が同期地点に到着したと判断し，待ち状態を解除している．

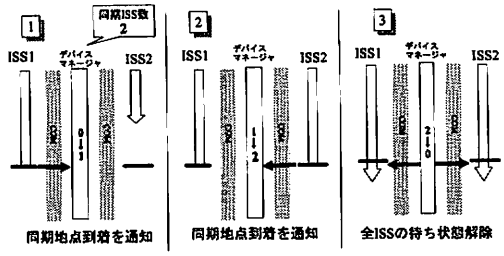


図 2 同期の流れ

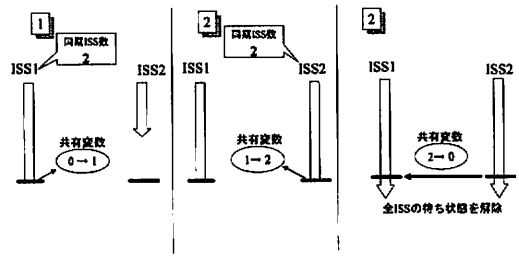


図 3 同期の流れ

3.4 ホスト OS の共有オブジェクトを用いた同期方法

COM を用いた同期方法とは別に，ホスト OS の共有オブジェクトを用いる同期方法を提案する．この方法では，デバイスマネージャを介さずに ISS だけで同期を行う．ISS は起動時に，待ち状態の ISS の数を保持する共有変数を生成する．また，全ての ISS からアクセス可能な共有セマフォを，ISS の数と同数生成する．セマフォと ISS は 1 対 1 に対応し，同一のセマフォの取得要求は常に同一の ISS から出される．

各 ISS は同期地点到着時に共有変数の値を更新し，共有セマフォの取得待ち状態になる．このとき，ISS は共有変数の値から，全 ISS が同期地点に到着したかどうかを判断する．全 ISS が到着したと判断した場合には，全てのセマフォを開放することで待ち状態を解除し，再び次の同期地点までシミュレーションを開始する．

図 3 は 2 つの ISS の同期の流れを示したものである．同期地点に先に到着した ISS1 は待ち状態になる．その後同期地点に到着した ISS2 は共有変数の値から，全 ISS が同期地点に到着したことを確認し，他の ISS の待ち状態を解除している．

4. RTOS のアイドル処理を活用したホスト CPU の効率的利用方法

本章では RTOS のアイドル処理を活用したホスト CPU の効率的利用方法について述べる．

RTOS 上で実行すべきタスクが無い状態をアイドル状態という．ホスト計算機の CPU コア数が実行中のシミュレータ数よりも少ない場合には，アイドル状態中は図 4 のように，他の ISS のシミュレーション実行を行うことで，シミュレーション時間の短縮が可能である．

4.1 ISS 非同期時の効率的利用方法

ISS 非同期時は RTOS がアイドル状態に遷移するたびにホス

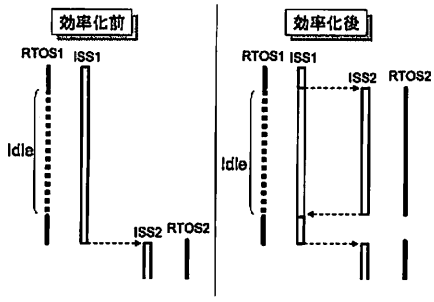


図 4　効率化の概要

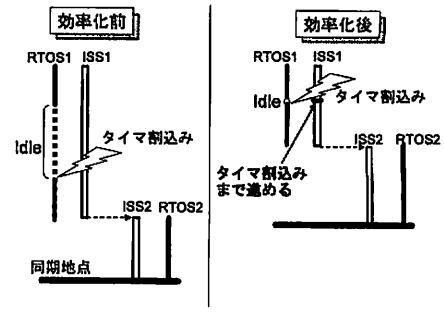


図 5　タイマ割込みへの対応

トOSの時間待ちAPIを用いて一定時間待ち状態にし，起床時に割込みの有無を調べる．ただし，アイドル状態からの復帰要因となるのは，主に他プロセッサからの割込みであると予想し，プロセッサ間割込みが発生した場合にはすぐに起床させる．また，本研究で用いたISSは一定数の命令を処理するごとに，タイマ割込みを発生させる仕様となっているため，ISSが待ち状態の間は命令が処理されずタイマ割込みが発生しなくなってしまう．そこで今回は，ISS起床時に，タイマカウンタの値にかかわらずタイマ割込みを発生させることとした．

4.2　ISS同期時の効率的利用方法

3章の同期方法を適用している場合は，非同期時と同様の方法をとると効率が悪くなる可能性がある．同期地点に最後に到達するISSが時間待ちAPIを発行すると，全てのISSの起床が遅れてしまうためである．そこで，ISS同期時にはサイクル数を進めることでアイドル状態中のシミュレーション実行を省く．ここで，非同期時と同様に，タイマ割込みに対する対応が問題となる．アイドル状態遷移時に次同期地点までサイクル数を進めてしまうと，その同期サイクル内で実行されるはずだったタイマ割込み処理の実行タイミングが変わってしまう．そこで，アイドル状態遷移時にタイマ割込み発生までのサイクル数を計算し，同期地点までの残りサイクル数よりも少ない，すなわち，同期地点の到達よりもタイマ割込みの発生のほうが早い場合には，タイマ割込みまでサイクル数を進める．そうでない場合には次同期地点までサイクル数を進め，そのサイクル数に相当する分だけタイマカウンタの値を更新する．図5の左側はRTOS1のアイドル時にタイマ割込みが発生し，その後同期サイクル数まで実行して，待ち状態になり，ホストOSによりISS2にディスパッチされる様子を表している．図5の右側では，ISSがアイドル状態に遷移したときに，次同期サイクル数まで進めるのではなく，タイマ割込みの発生まで進めている．そのため，本来その同期サイクル内で行われたはずの，タイマ割込み処理を，効率化後もその範囲内で行うことができる．

5.　実　　験

3章，4章に示した2つの改良に対して，その効果と影響を調べるため，以下の環境で実験を行った．

- ホストOS：Windows XP professional
- ホストCPU：Intel Core 2 Quad 2.66GHz 1.0GBRAM

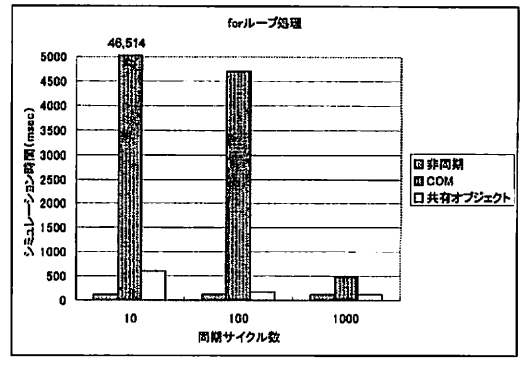


図 6　2つの同期手法のシミュレーション時間の比較
ホストコア数 2，ISS 数 2 とした場合

5.1　同期手法の比較

COMを用いた方法と共有オブジェクトを用いた方法の2つの実装方法の処理時間を比較するため，100,000回の空のforループ処理にかかるシミュレーション時間を計測した．

図6はホストコア数を2とし，2つのISSを同期させた場合の結果である．forループに要したサイクル数は1,302,228サイクルであった．図6から，COMを用いた同期方法（方法1）では，共有オブジェクトを用いた同期方法（方法2）と比べて，非常に長いシミュレーション時間がかかることがわかる．同期サイクル数100の条件で比較すると，方法1は非同期時の約40倍の時間を要したのに対し，方法2では約1.5倍の時間に抑えることができた．ISS間のみで同期を行う場合は方法2を適用すべきである．しかしCOMには柔軟で汎用性が高く，記述言語に依存しないという利点があり，ハードウェアシミュレータと同期させる場合には方法1しか適用できない可能性もある．

5.2　ISS数とホストCPU数の影響

ISS数とホスト計算機のCPUコア数の影響を調べるため，共有オブジェクトを用いた同期方法で，ISS数，ホスト計算機のCPUコア数を変え，5.1節と同様の処理にかかるシミュレーション時間を計測した．ホスト計算機は4つのCPUコアを持つが，ISSに静的に割り当てることで任意のコア数を再現した．本論文では以降，ホスト計算機のCPUコアをホストコアと記述する．

図7はそれぞれ，ISSを2〜4個同期させた場合のシミュレー

ション時間を表す．同期サイクル数がある程度の大きさならば，ホストコア数が増えるほど，各 ISS が並列に処理されシミュレーション時間が短くなる．しかし，同期サイクル数が小さい場合，ホストコア数を増やすとシミュレーション時間が長くなってしまう場合があった．この原因はホスト計算機の CPU のキャッシュ構成によるものと考えられる．実験に用いたホスト計算機の CPU は Intel Core2 Quad である．これは L2 キャッシュを持つ 2 つのダイにより構成されており，1 つのダイに，さらに高速な L1 キャッシュを備えた，2 つのコアが搭載されている [5]．したがって，3 つ以上の ISS を起動した場合にはメインメモリまでアクセスする必要があり，共有変数へのアクセス時間が長くなる．ISS の処理時間がメモリアクセス時間と比較して小さくなると，メインメモリまでのアクセス時間と，キャッシュメモリまでのアクセス時間の差の影響が大きくなると考えられる．

5.3 同期の有用性の確認

TOPPERS/FDMP カーネルに付属するログトレース機能を用いて非同期時と同期時のトレース結果を比較し，同期の有用性を確かめた．TOPPERS/FDMP カーネルは，ITRON 仕様 [3] を機能分散マルチプロセッサ向けに拡張した機能分散マルチプロセッサ向けリアルタイムカーネル仕様に準拠した RTOS である．

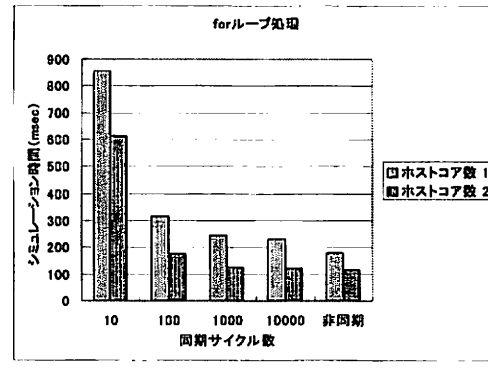
図 8 は，ログトレース結果を，波形表示したものである．TOPPERS/FDMP カーネルのトレース機能にはフェイズという概念があり，トレースを行ったコード上の位置を示す．トレース結果のそれぞれのラインがフェイズに対応している．2 つの実線で囲れているのはそれぞれ別の ISS 上のタスクのフェイズを表している．波形の立ち上がりはタスクの起動を意味し，横軸はサイクル数をもとにしている．使用したテストプログラムでは，点線で囲まれたフェイズ 3 のときに MAIN_TASK が，別 ISS 上の TASK2 の起動要求を発行している．図 8(a) は非同期時のトレース結果であるが，起動要求発行のタイミングと起動のタイミングがずれて表示されている．これは，それぞれの ISS が保持するサイクル数が異なるためである．それに対し，同期を行った場合の図 8(b) では，MAIN_TASK のフェイズ 3 の間に TASK2 が起動されていることが確認できる．

システム開発時にもこのように，実行ログから検証を行うことが考えられる．その場合，ISS 間のタイミングのずれを解消することで，ISS 間の協調動作の流れがわかりやすくなり，検証が行いやすくなると思われる．

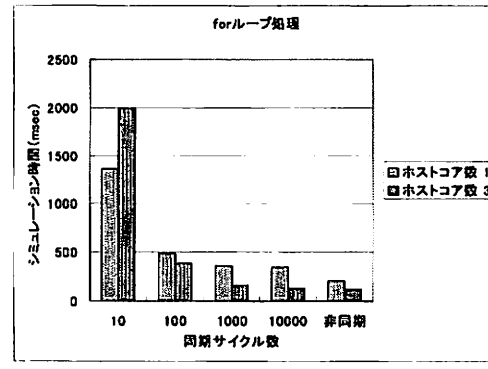
5.4 ISS 非同期時の効率化

ISS 非同期時のホスト CPU の効率的利用方法の効果を調べるため，デュアルプロセッサ用 MPEG4 デコードシステムのシミュレーション時間を計測した．

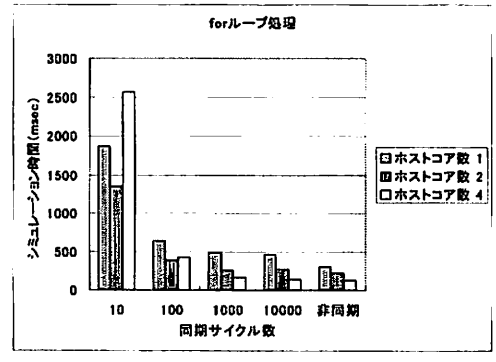
表 1 は非同期時の効率化手法適用前後のシミュレーション時間を比較したものである．アイドル時の ISS 起床周期は 1msec としている．ホストコア数が 1 の場合，効率化手法の適用により，シミュレーション時間が従来の約 90 % 減少した．ホストコア数が 1 の場合，提案する手法により，大幅にシミュレーション時間を短縮することができた．しかし，ホストコア数が 2 の



(a) ISS 数 2 の場合



(b) ISS 数 3 の場合

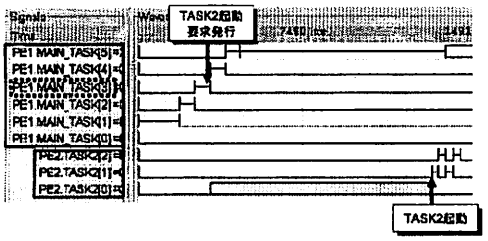


(c) ISS 数 4 の場合

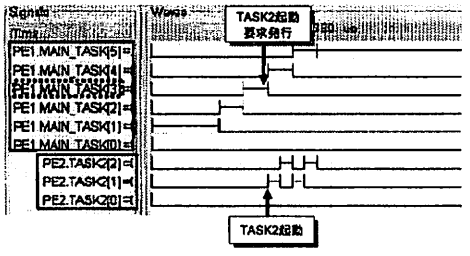
図 7 シミュレーション時間

場合，シミュレーション時間が延びてしまった．ホストコア数が ISS 数と同じであれば，CPU を譲る対象が無いため，シミュレーション時間が短縮されることは無く，時間待ち API によるオーバヘッドがかかったためと考えられる．

ここで，効率化後のシミュレーション精度は，効率化前より



(a) 非同期時



(b) 同期時

図 8 トレース結果

表 1 効率化前後のシミュレーション時間の比較

ホストコア数	効率化前	効率化後
1	1307sec	131sec
2	65sec	116sec

も低下している．アイドル時に入った割込みの処理タイミングや，タイマ割込みの発生頻度が，効率化を行うことで変わってしまうと考えられる．本手法では，高精度のシミュレーションを行うことはできない．しかし，機能レベルの検証を非常に高速に行うことが可能である．

5.5 ISS 同期時の効率化

ISS 同期時の効率化手法の効果を調べるため，デュアルプロセッサ用 MPEG4 デコードシステムの 1 フレーム処理にかかるシミュレーション時間を計測した．

図 9 は効率化前後のシミュレーション時間を比較したものである．ホストコア数は 1 とした．

同期サイクル数が大きくなるほどシミュレーション時間の削減量が多くなっている．これは，同期サイクル数が大きいほど，短縮できる命令数が多いためと考えられる．非同期時と比較してシミュレーション時間の削減量は少ないが，シミュレーション精度の低下を抑えることができる．非同期時はタイマ割込みに関して全く保障していなかったが，同期時では，そのタイミングのずれは同期サイクル内にとどまる．

6. おわりに

本研究ではまず，ISS 間の任意サイクル数での同期を実現し，シミュレーション精度を向上させた．実機に近いシミュレーションが可能となっただけでなく，実行ログ解析やデバッグを容易に行うことができるようになった．

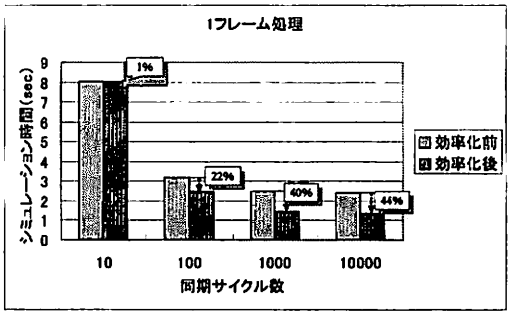


図 9 効率化前後のシミュレーション時間の比較

次に RTOS のアイドル状態を考慮したホスト CPU の効率的利用方法を提案し，実験により，シミュレーション時間が大幅に削減されることを確かめた．ただし，この手法の効果は，シミュレーション対象とシミュレーション環境に大きく依存する．シミュレーション対象にアイドル状態が発生し，なおかつ，ホストコア数がシミュレータ数よりも少ない場合しか効果が得られない．しかし，組込みシステムのマルチプロセッサコシミュレーションでは，多くの場合上記の条件を満たすと思われる．組込みシステムでは主に，各プロセッサにタスクが固定された，機能分散型のマルチプロセッサ RTOS が用いられる．そのため，あるプロセッサ上の処理が終了するまで，他のプロセッサ上の RTOS は，アイドル状態となって待たされる可能性がある．また，多くのシミュレータを起動する，マルチプロセッサシステムのコシミュレーションにおいては，1 つの ISS が 1 つのホスト CPU を占有できない場合が予想されるからである．

本研究で行った ISS 間の同期とホスト CPU の効率的利用方法を，要求される精度や速度に応じて適用することで，柔軟なシミュレーションが可能となった．

今後の課題としては，ISS 間だけでなくハードウェアシミュレータとサイクルレベルで同期したコシミュレーションの実現や，効率化手法の改良によりシミュレーション精度を高めることなどが挙げられる．

文 献

[1] 古川貴士，柴田誠也，本田晋也，冨山宏之，高田広章，“マルチプロセッサ RTOS 対応コシミュレータ，” 情報処理学会研究報告組込みシステム，Vol.2008,No.7, pp. 17-22, 神奈川，Jan 2008.
[2] Chen Yu, Ren Jie, Zhu Hui, Shi Yuan Chun, “Dynamic Binary Translation and Optimization in a Whole-System Emulator - SkyEye,” icppw, pp. 327-336, 2006 International Conference on Parallel Processing Workshops (ICPPW'06), 2006
[3] ITRON プロジェクト. http://ertl.jp/ITRON/.
[4] SkyEye. http://www.skyeye.org/
[5] Intel. http://www.intel.co.jp/