Vol. 48 No. SIG 13(ACS 19) 情報処理学会論文誌：コンピューティングシステム Aug. 2007

プラットフォーム独立の追加ハードウェアIPによるSoC仮想化方式

矢 尾 浩[†] 吉井 謙一郎[†] 金 井 達 徳[†]

デジタルメディア機器に組み込まれる System-on-Chip（SoC）は，プロセッサコアだけでなくハードウェアエンジンや DSP 等様々な機能モジュールを混載し，ハードウェアとソフトウェアが高機能化そして複雑化している．このような SoC では，各機能モジュールにおける処理間の不要な相互干渉等を防ぐために，SoC 全体を対象とした安全で信頼性の高い実行環境が望まれている．そこで本稿では，SoC 全体を仮想化することによりそのような実行環境を実現する方式を提案する．本方式ではプラットフォームに依存しないハードウェア IP（Intellectual Property）を SoC に追加する．このハードウェア IP とハイパーバイザが連携して，メモリとデバイスへのアクセス制御，および割込みの適切な配送を行うことにより，SoC 全体の仮想化を実現する．さらに FPGA を用いて本方式の試作と性能測定を行い，本方式の実現可能性を検証した．

SoC Virtualization by Platform-independent Hardware IP Components

Hiroshi Yao,[†] Ken-ichiro Yoshii[†] and Tatsunori Kanai[†]

System-on-Chips (SoC) embedded in digital media systems become more complicated and sophisticated because various functional modules are integrated into them. Safe, secure, and dependable runtime environment that covers whole SoC is expected to avoid interference between processes of functional modules. Therefore, we propose an architecture of SoC virtualization that realizes such runtime environment for software and hardware. The architecture adds platform-independent hardware IPs to SoC. The hardware IPs and hypervisor realize SoC virtualization by access control for memories and devices, and delivery of interrupts from devices to an appropriate destination. Feasibility is studied by prototyping with FPGA and performance measurement.

1. はじめに

組み込み機器では，1つの機器内で複数の機能を同時に実行したり，新しい機能をダウンロードして機能を追加したりする等，機能の複雑化が進んでいる．特に，組み込み機器に用いられる System-on-Chip（SoC）には多様な機能モジュールが混載され，プロセッサ単体でのアクセス制御だけでなく，デジタル信号処理装置（DSP）やハードウェアエンジン等を含む SoC 全体でのアクセス制御を考慮しなければならない．このようにアーキテクチャ上からも高信頼化とセキュリティの確保がより困難となっている．

そこで我々は，SoC 全体を仮想化することにより各機能モジュールにおける処理間の不要な相互干渉等を防ぎ，SoC 全体を対象とした安全で信頼性の高い実行環境を実現する方式を提案する．本システムは，小規模な追加ハードウェア IP と，ファームウェアとして実現されるハイパーバイザの組合せにより仮想 SoC を提供する．ハイパーバイザを実行するための領域は，専用のハードウェアによって保護する．

本稿ではまず 2 章で我々が目指す SoC 仮想化方式を示す．次に 3 章で SoC 仮想化を実現するアーキテクチャの概要と追加ハードウェア IP，4 章でハイパーバイザについて説明する．5 章で FPGA を用いた試作による性能測定について報告し，その結果をふまえて 6 章で性能への影響や適用範囲について考察する．

2. SoC の仮想化

高機能な SoC において，ハードウェアおよびソフトウェアに対して安全で信頼性の高い実行環境を提供するには，プロセッサだけでなく，混載された機能モジュールの動作も考慮する必要がある．たとえばメディアプレーヤに使用される SoC では，ハードウェ

† 株式会社東芝研究開発センター
Corporate Research & Development Center, Toshiba Corporation

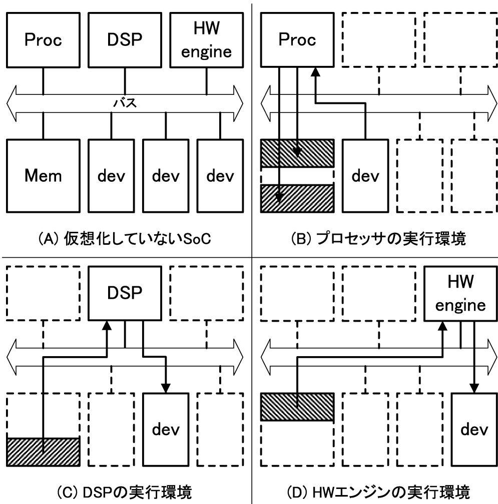

図 1 SoC 仮想化のイメージ
Fig. 1 Concept of SoC virtualization.

アエンジンや DSP による映像や音声信号の復号化や加工そしてデバイスへの出力処理，さらにはプロセッサ上で動作している再生ソフトウェアによる全体処理の制御が並行して行われている．

このとき，信頼性の観点から，再生ソフトウェア等一部の機能の不具合でシステム全体が停止しないように，各処理に必要なデータやソフトウェア本体を他の処理から保護したり，機能モジュールが特定の処理に不当に占有されないように制御したりする必要がある．

またセキュリティの観点から，悪意のあるソフトウェアやそれに操作された機能モジュールによるデータの横取りや，本来出力が許されないデバイスへの出力による漏曳を防ぐ必要がある．

そこで我々は，SoC 全体を仮想化することにより，安全で信頼性の高いハードウェアとソフトウェアの実行環境を実現することを考えた．**図 1** を使用してこの仮想化のイメージを説明する．

図 1 (A) は，プロセッサ，DSP，ハードウェアエンジン，メモリ，そして入出力デバイスを備えた SoC のブロック図である．本仮想化方式は，図 1 (A) のプロセッサ，DSP，ハードウェアエンジンに対してそれぞれ図 1 (B)〜(D) のような実行環境を提供する．図中の実線はアクセスできる機能モジュールを，そして破線はアクセスできない機能モジュールを示す．

たとえば，図 1 (D) はメモリからデータを読んで処理を行い結果をデバイスに出力するハードウェアエンジンの実行環境である．図のとおり，ハードウェアエンジンは処理に必要なデータが格納されたメモリ領域と出力デバイスしかアクセスできない．また図 1 (B) と (C) のとおり，プロセッサと DSP はハードウェアエンジンが処理するデータやデバイスへのアクセスを制限される．したがって，このハードウェアエンジンの処理は，他の機能モジュールの処理に不具合があってもその影響を受けず，また処理中のデータも保護される．

このように，本技術が目指す SoC の仮想化は，従来の仮想計算機技術[1),2)] が実現しているプロセッサ上での複数 OS の動作を含め，図 1 で見たようにプロセッサ以外のハードウェアエンジンや DSP といった機能モジュールに対して仮想的な実行環境を提供するものである．また，SoC に組み込まれるプロセッサは仮想計算機の構築を支援するようなハードウェアを搭載していないものが多い．以上から，SoC を仮想化するために従来の仮想計算機技術をそのまま適用することはできず，SoC ならではの工夫が必要となる．

3. 仮想化アーキテクチャ

本章では，SoC の仮想化を実現するために我々が提案するアーキテクチャの基本的な設計方針と特徴，およびハードウェア IP の構成について説明する．

3.1 アーキテクチャ概要

本方式では，前章で述べた SoC の仮想化機能のすべてをハードウェアで実装するのではなく，既存のプラットフォームに付加するハードウェア IP とそれを制御するファームウェアの適切な組合せで実現する．

本方式のアーキテクチャにおける大きな特徴は 2 つである．1 つは，プロセッサのメモリ管理機構に頼らず，メモリやデバイスをバスに接続するコントローラにアクセス制御機構を集約することにより，複数の機能モジュールからのアクセスを一括制御することである．もう 1 つは，組み込み機器に使用されるような安価で仮想化支援機構を備えないプロセッサを利用する場合でも，ハードウェア IP とファームウェアの連携によって複数ゲスト OS が動作する環境を提供することである．

本方式のハードウェア IP を含む SoC は**図 2** のようになる．図 2 中の網掛けの回路がハードウェア IP であり，それぞれ**動作モード管理回路**，**メモリアクセス制御回路**，**デバイスアクセス制御回路**，そして**割込み処理監視回路**である．

ファームウェアとして実装される**ハイパーバイザ**はプロセッサ上で動作し，これらのハードウェア IP の設定を行うとともに，抽象度の高い機能をゲスト OS に提供する．

このような構成により，SoC に搭載される様々な組

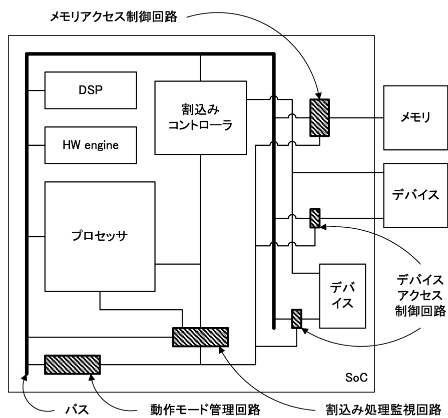

図 2 仮想化ハードウェア IP を付加した SoC
Fig. 2 SoC with hardware IPs for virtualization.

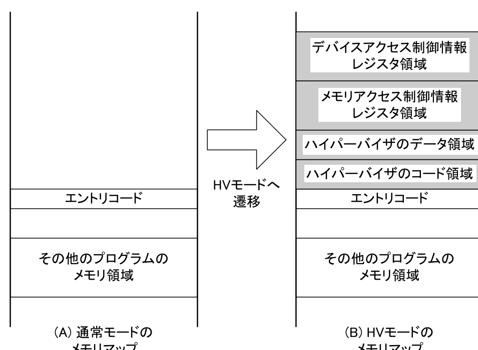

図 3 動作モードによる保護領域の見え方の違い
Fig. 3 Protected area in HV mode.

み込みプロセッサと組み合わせて使用することが可能である．また，複雑な処理の記述をファームウェアにまかせ，ハードウェア IP を軽量にできる．

3.2 保護領域の確保

仮想化支援機構を備えないプロセッサを利用して図 1 (B)〜(D) のような仮想実行環境を実現する際の大きな課題は，メモリおよびデバイスへのアクセス制御の設定情報等，仮想化のために重要な情報を，ゲスト OS 等のソフトウェアやハードウェアエンジンのような機能モジュールから保護することである．この保護が実現できないと，アクセス制御の設定情報を自由に操作され，他のゲスト OS やソフトウェアそして機能モジュールに割り当てられたメモリやデバイスが不正にアクセスされてしまう．

この保護を実現するにはハイパーバイザとゲスト OS の動作を区別する必要があるため，本方式ではプロセッサから独立した新しい動作モードを導入する．この動作モードは 2 個のモードからなり，ハイパーバイザが動作している状態を **HV モード**，それ以外を通常モードと呼ぶことにする．この動作モードは動作モード管理回路の内部状態により保持する．ハイパーバイザはプロセッサの動作モードとしてはゲスト OS と同様に最上位の特権モードで動作するが，動作モード管理回路の動作モードとしては HV モードで動作する．

本方式では，SoC 上にハイパーバイザのみがアクセスできる保護領域を作成する．保護領域にはハイパーバイザのコード領域とデータ領域が含まれる．またアクセス制御回路の制御レジスタはメモリマップドレジスタとして構成され，保護領域にはそのレジスタ領域が含まれる．

動作モードによるこの保護領域の見え方の違いは

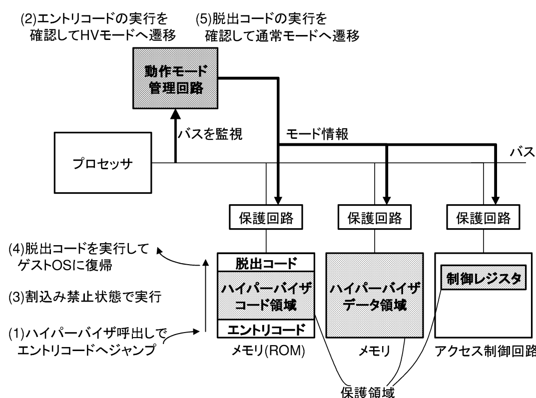

図 4 動作モード管理回路と保護領域の構成
Fig. 4 Mode manager and protected area.

図 3 のようになる．プロセッサ上でゲスト OS が動作しているときは動作モードは通常モードであり，このときは図 3 (A) のように保護領域は存在しないように見える．それに対してハイパーバイザが動作しているときは HV モードであり，図 3 (B) のように，ハイパーバイザのコードやデータ領域をはじめ，メモリアクセス制御回路およびデバイスアクセス制御回路が備える設定情報にアクセスできるようになる．

次に，**図 4** に動作モード管理回路と保護領域の接続関係を示し，ハイパーバイザ呼び出し時の動作の流れを説明する．

本方式では，ハイパーバイザの入り口となる**エントリコード**を 1 カ所に限定する．エントリコードはプロセッサが割込み禁止であることを確認する命令列で構成され，このエントリコードをハイパーバイザの先頭に配置する．この配置アドレスはシステムで固定とする．同様にハイパーバイザの出口となる脱出コードも 1 カ所に限定し，配置アドレスをシステム固定とする．

ゲスト OS がハイパーバイザの機能を利用すると

きやリセット時にはエントリコードにジャンプする（図 4 (1)）．動作モード管理回路はつねにバス上の信号を監視しており，エントリコードが正しく呼び出されたことを確認すると，通常モードから HV モードに遷移させる（図 4 (2)）．エントリコードによって割込み禁止状態であることが保証され，以降は HV モードでハイパーバイザ本体が実行される（図 4 (3)）．ハイパーバイザの処理が終了すると，脱出コードを実行しゲスト OS に復帰する（図 4 (4)）．動作モード管理回路は脱出コードの実行を確認すると，通常モードに遷移させる（図 4 (5)）．

バスと保護領域の接続部には保護回路が挿入されており，動作モード管理回路から動作モードの識別信号が保護回路に入力される．モード信号が通常モードを表す場合，保護回路は保護領域（図 4 の網掛け部分）へのアクセスをブロックし，HV モードの場合にはすべてのアクセスを通す．このように動作モードの識別信号を利用することで，仮想化のために重要な情報が格納されるメモリ領域の保護を実現している．

3.3 動作モード管理回路

保護領域に格納される情報は本方式でのアクセス制御の根本となっており，これらの情報を任意のユーザコードに不正にアクセスさせないことが必要不可欠である．すなわち，動作モード管理回路が HV モードに遷移する際には，HV モード中にハイパーバイザ以外のコードが実行されないことを保証する必要がある．

上記を保証するには，まず，ゲスト OS が自由に通常モードと HV モードを遷移させることが可能であってはならない．これを防ぐために，プロセッサが動作モード管理回路の制御レジスタに動作モードを書き込んで設定するのではなく，動作モード管理回路がバス上の信号を監視することにより自律的に動作モードを遷移させる．

さらに，ハイパーバイザが割込み禁止状態で実行されることが必要である．もしハイパーバイザの実行中に割込みが発生すると，HV モードのまま割込み処理ルーチンへジャンプしてしまう．このとき，割込みベクタテーブルや割込み処理ルーチンが保護可能であってハイパーバイザの一部であれば問題はない．しかし，仮想化支援機構を備えないプロセッサではこれを実現できないため，割込みを悪用されると任意のコードを HV モードで実行可能であり，結果として保護しようとしていた情報を操作されてしまう．

そこで本方式では，ハイパーバイザへ突入するためのエントリコードを 1 カ所に限定してシステム固定のアドレスに配置し，そのアドレスへの命令フェッチを検出する．さらにエントリコードの実行によって割込み禁止であることを確認する．これはプロセッサの状態レジスタの値を読み出して，動作モード管理回路に割り当てられたチェック用アドレスに書き込むことにより行われる．この命令フェッチの検出と割込み禁止の確認が不可分に行われた場合にのみ動作モードを HV モードに遷移する．なお，エントリコードはキャッシュされないメモリ領域に格納し，プロセッサからの命令フェッチがバスに出るようにする．

ただし，プロセッサの命令セットによっては上記のエントリコードが複数の命令で構成される．この場合，エントリコードの実行中に割込みを発生されて偽装コードを実行され，チェック用アドレスに書き込まれる状態レジスタの値が偽装される可能性がある．これを防ぐために，以下のチェックを行う．

- エントリコードの命令フェッチを順に連続して検出．
- 最初の命令フェッチからチェック用アドレスへの書き込みまでの時間が所定の制限時間を超えない．
- チェック用アドレスに書き込まれた値が割込み禁止を表す．

これらがすべて満たされた場合にのみ HV モードに遷移する．

上記の仕組みを用意することで，ゲスト OS が自由に動作モードを遷移させることはできず，エントリコードの実行によってのみ動作モードを HV モードに遷移する．その後は割込み禁止状態でハイパーバイザが動作する．

ハイパーバイザからゲスト OS に復帰する際にも，エントリコードと同様に脱出コードをシステム固定のアドレスに配置し，脱出コードの命令フェッチを検出することにより HV モードから通常モードへ遷移する．

エントリコードの命令列と命令数はプロセッサに依存し，動作モード管理回路の詳細な仕様はエントリコードの構成と配置に依存する．たとえば，5 章で後述する試作に使用した PowerPC405 プロセッサでは，エントリコードは以下の 2 命令で構成できる．

命令 1：`mfmsr r0`

命令 2：`stw r0, CHECKADR`

命令 1 は状態レジスタ（MSR）の値を汎用レジスタ r0 に読み出す．命令 2 は r0 をチェック用アドレス CHECKADR に書き込む．動作モード管理部では，このエントリコードの命令フェッチ要求を検出して，さらに命令 2 で書き込まれた値が割込み禁止状態を表すかをチェックする．つまりこの場合は，動作モード管理回路はバスを監視することで以下のように状態遷移

が起こることをチェックして，すべての条件が満たされれば HV モードへ遷移する．

状態 1　フェッチアドレス=命令 1 なら状態 2 へ．

状態 2　フェッチアドレス=命令 2 なら状態 3 へ．

状態 3　書き込みアドレス=CHECKADR なら状態 4 へ．

状態 4　書き込みデータ=割込み禁止状態なら状態 5 へ．

状態 5　HV モードへ遷移．

さらに，命令 1 と命令 2 の間で割り込まれて r0 の値を偽装されないように，状態 1 から状態 5 までの遷移の時間が所定の制限時間以内かどうかをチェックする．あらかじめ，命令 1 と命令 2 が順に実行された場合において，命令 1 のフェッチ検出から命令 2 での書き込み検出までのクロック数を実測またはシミュレーションにより測定しておき，その値を制限時間とする．動作時には，状態 1 での命令 1 のフェッチ検出から時間計測を開始し，各状態において制限時間を超えた場合は状態 1 へ遷移する．

3.4　アクセス制御回路

本方式ではアクセス制御回路を図 2 に示すようにバスとメモリやデバイスの接続部に配置することにより，プロセッサ上で動作するゲスト OS 等のソフトウェアだけでなく，DSP やハードウェアエンジンといった機能モジュールからのアクセスを漏らさず制御する．

アクセス元の判別にはバスマスタ ID を使用する．アクセス制御情報は各バスマスタ ID に対応するアクセス先の情報からなる．アクセス制御情報を設定するためのレジスタは図 3，図 4 の保護領域内に存在し，ハイパーバイザによって設定される．

メモリアクセス制御回路は，ゲスト OS や SoC が搭載する機能モジュールに対して実メモリを仮想化したゲストアドレス空間を提供する．アクセス制御情報は各バスマスタ ID に対するゲストアドレス空間と実メモリ空間のマッピング情報およびその他属性情報からなる．メモリアクセス時にはバスマスタ ID に対応する設定情報に従いアクセスが許可されているかどうかをチェックし，アドレス変換を行い，実メモリにアクセスする．

図 5 にメモリアクセス制御回路の構成を示す．アクセス制御情報のエントリは，バスマスタ ID（bid）に対してアクセスを許可するゲストアドレス空間の開始アドレス（saddr）および終了アドレス（eaddr），実メモリ空間のアドレスへのオフセット（offset）からなる．メモリアクセス制御回路はこのエントリを複数保持する．このエントリごとに，アクセスに対して有効

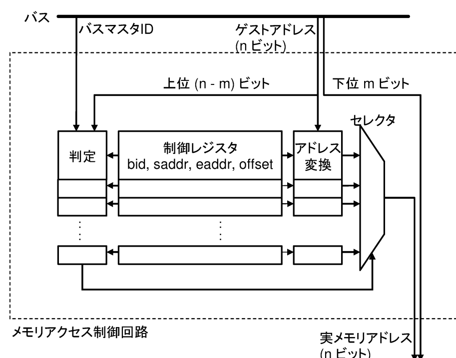

図 5　メモリアクセス制御回路の構成
Fig. 5　Memory access controller.

かどうかの判定回路とアドレス変換回路が用意される．プロセッサからバス上に出る要求のアドレスはゲストアドレス空間上のアドレス（n ビット）であり，その下位 m ビットはそのまま実アドレスの下位ビットとして使用される．アドレス変換回路では上位（$n-m$）ビットに対して offset を加えることでアドレス変換を行い，結果をセレクタに出力する．判定回路では，バス上のバスマスタ ID と bid が等しく，アドレスの上位（$n-m$）ビットが saddr 以上かつ eaddr 未満であるという条件が成立する場合にエントリが有効と判定し，その結果をセレクタに出力する．セレクタは判定回路の出力をキーとして実アドレスの上位（$n-m$）ビットを出力し，下位 m ビットと合わせてメモリコントローラへ出力する．

有効なエントリが存在しない場合にはアクセスをブロックする．アクセスをブロックするには様々な方法が考えられるが，たとえば接続されるメモリのアドレス範囲外の無効なアドレスを出力すればよい．この方法は様々なバスプロトコルに適用が容易である．

デバイスアクセス制御回路は，バスマスタ ID ごとに設定された各デバイスへのアクセス権の有無に基づいてアクセス制御を実現している．

図 6 にデバイスアクセス制御回路の構成を示す．アクセス制御情報のエントリはバスマスタ ID の数だけ用意される．各エントリは，対応するバスマスタから各デバイスへのアクセス権を表すビットマスクからなる．各エントリのビットマスクはセレクタに入力され，バス上のバスマスタ ID をキーとして選択される．バスとデバイスの接続部には保護回路が挿入されており，選択されたビットマスクからデバイスに対応するビットがアクセス権として保護回路に入力される．各保護回路は入力されたアクセス権が 0 の場合，そのデバイスへのアクセスをブロックする．

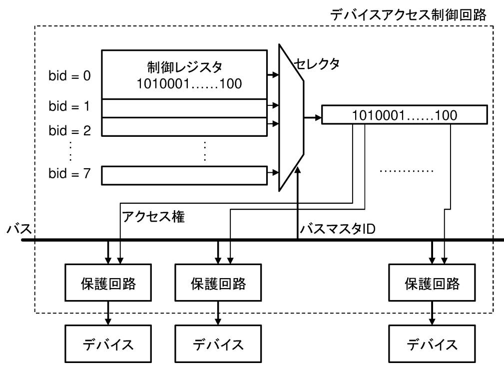

図 6　デバイスアクセス制御回路の構成
Fig. 6　Device access controller.

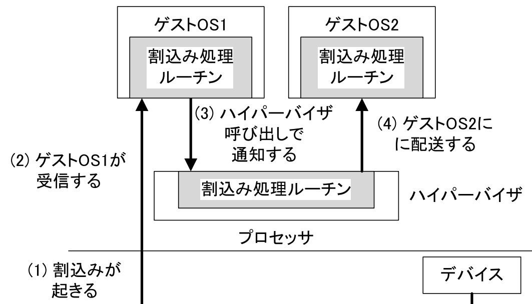

図 7　デバイス割込みの配送
Fig. 7　Device interrupt processing.

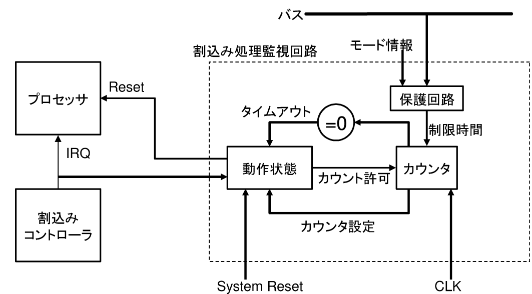

図 8　割込み処理監視回路の構成
Fig. 8　Device interrupt monitor.

この保護回路でのブロック方法もメモリアクセス制御回路と同様に，接続されるデバイスのアドレス範囲外の無効なアドレスを出力する．なお，アドレス信号のすべてのビットを変更する必要はない．たとえばデバイスの制御レジスタのアドレスが 0x80000000–0xFFFFFFFF の範囲に収まる場合，最上位ビットのみをマスクすればよい．この方式にはハードウェア規模を小さくできるメリットがある．

デバイス自体がメモリや他のデバイスに直接アクセス可能な場合には，そのデバイスは機能モジュールと同様に扱われる．このようなデバイスには固有のバスマスタ ID が割り当てられ，そのデバイスからのアクセスは DSP やハードウェアエンジンからのアクセスと同様にアクセス制御回路の制御の対象となる．

なお，アクセス制御回路では，バスマスタ ID 単位でアクセス元を判断するため，プロセッサ上で動作するゲスト OS の切替え時には，ハイパーバイザがプロセッサのバスマスタ ID に対応するすべてのエントリを更新して，次のゲスト OS のアクセス制御情報を設定する．

ハイパーバイザが制御情報をいったん設定した後は，アクセス制御回路はその設定に従って自律的に動作する．バス上にアクセス要求が発生するたびにプロセッサ上でハイパーバイザが呼び出されてアクセス権をチェックしたり，エントリを入れ替えたりすることはなく，プロセッサのスループットには影響を与えない．

3.5　割込み処理監視回路

一般にプロセッサは，割込み要因に対応する処理を記述した割込みベクタテーブルを参照して割込み処理を行う．従来の仮想計算機技術では，仮想化層がデバイス割込みをいったんすべて受け取ってそれを適切なソフトウェアに配送している．しかし，本方式はプロセッサの仮想化支援機構を前提にできないため，割込みベクタテーブルが配置されるメモリ領域を保護できず，デバイス割込みの適切な配送を保証できない．そこで，割込みベクタテーブルを保護する代わりに，すべてのゲスト OS は割込みを受け取るとまずハイパーバイザを呼び出すようにする．

本方式における割込み処理の様子を**図 7** に示す．デバイスが割込みを起こすと，その時点でプロセッサ上で動作しているゲスト OS1 に割込みが通知され，ゲスト OS1 の割込み処理ルーチンが実行される．すると，ゲスト OS1 はハイパーバイザに割込みを受信したことを通知する．通知を受けたハイパーバイザは，その割込みの内容を調べて適切なゲスト OS2 に配送する．

ただし，このままでは不具合があるゲスト OS や悪意のゲスト OS によってデバイス割込みを無視される（ゲスト OS1 が (3) の処理をしない）おそれがある．

割込み処理監視回路はこのような状況に対応するために追加される回路である．**図 8** に示すように，本方式では，割込み処理監視回路がプロセッサに入る割込み信号（IRQ）を監視している．

ハイパーバイザが制限時間を設定するとその値はカウンタに設定される．割込み信号が active になったことを検出すると，カウントダウンを開始する．その後ゲスト OS が割込みを通知して，ハイパーバイザがカ

ウントの停止を指示する前にカウンタの値が 0 になりタイムアウトすると，本回路はプロセッサに対してリセットをかける．このように本回路はウォッチドッグタイマとして動作し，ゲスト OS により割込みが無視され続けた場合に，いったんハイパーバイザに制御を戻す契機を作る．

この仕組みによって，割込みベクタテーブルが保護できなくても，ハイパーバイザがデバイスの起こした割込みをすべて把握することができ，適切なゲスト OS への配送が可能となる．

4. ハイパーバイザ

ファームウェアとして実装されるハイパーバイザはハードウェア IP の設定を行うとともに，抽象度の高い機能を API（Application Programming Interface）としてゲスト OS に提供する．API は**表 1** に示すとおりであり，大きく分けて以下の機能に分類される．

- パーティション管理
- 割込みコントローラ制御
- タイマ制御

この章ではこれらの機能とその動作，およびゲスト OS の切替えに関する内部動作について説明する．

4.1 パーティショニング

パーティションはリソースを分割してアクセス制御を行う単位である．本方式のパーティションは，アクセス元となる機能モジュールに対して，アクセス可能なメモリ領域およびデバイスを割り当てたものである．

プロセッサ上で動作するゲスト OS が複数存在する場合，各ゲスト OS に対して個別のパーティションを割り当てることで，ゲスト OS 間のアクセス制御を行う．プロセッサ以外の機能モジュールに対しても，個別にパーティションを割り当てることで，機能モジュール間でのリソースへのアクセス制御が可能になる．

本方式でのパーティションは階層構造を持つ．パーティションは新しい子パーティションを生成し，自身が持つメモリ領域やデバイスといったリソースの一部のみを子パーティションに譲渡することにより，この階層構造を構築する．

パーティションには `createPartition` での生成時にパーティション識別子（parID）が割り当てられる．以降のパーティションの操作やリソースの譲渡等は parID を用いて対象のパーティションを指定する．ハイパーバイザは譲渡されるリソースが呼び出し元の持つリソースであることと，指定された parID が呼び出し元の子パーティションであることをチェックする．

これにより，子に対してより制限された実行環境を構築し，子に不具合が生じても親兄弟には悪影響を与えないようなサンドボックスとして扱うことができる．

逆に，親に不具合がある場合には子のパーティション設定も信頼できず，動作も保証できない．すなわち，この階層はパーティションの信頼性の階層を表しており，上位のパーティションほど高い信頼性が要求される．特にハイパーバイザはこの階層の頂点にあるため，絶対の信頼性が必要である．

図 9 はパーティションの階層構造を示した例である．システム起動時にはマスターパーティションと呼ぶすべてのリソースにアクセス可能な特別なパーティションが作成され，その上でハイパーバイザが動作する．ハイパーバイザはシステムの設定に従ってルート

表 1 ハイパーバイザ API
Table 1 Hypervisor API.

機能	API
パーティション管理	
パーティション生成	`createPartition(busmasterID)`
パーティション削除	`deletePartition(parID)`
パーティション実行開始	`activatePartition(parID, saddr)`
パーティション実行停止	`deactivatePartition(parID)`
メモリ領域の生成	`createSection(saddr, size)`
メモリ領域の譲渡	`attachSection( parID, secID, offset)`
デバイスの譲渡	`leaseDevice(parID, devID)`
割込みコントローラ制御	
初期化	`initPIC()`
割込み番号の取得	`getIRQ()`
割込みの許可	`enableIRQ(irqNo)`
割込みの禁止	`disableIRQ(irqNo)`
割込みのクリア	`ackIRQ(irqNo)`
タイマ制御	
RTC の読み出し	`readRTC()`
タイマの設定	`setTimer(time)`
タイマのキャンセル	`cancelTimer()`

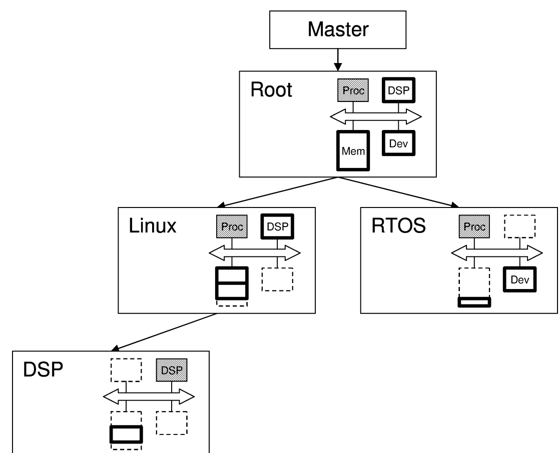

図 9 パーティションの階層構造
Fig. 9 Hierarchy structure of partitions.

パーティションを生成し，リソースを譲渡したうえで，最初のゲスト OS としてモニタを起動する．モニタは自身の設定に従って 2 つの子パーティションを生成し，自身のリソースの一部を子パーティションに渡して，それぞれ Linux とリアルタイム OS（RTOS）といったゲスト OS を起動する．さらに Linux では，DSP 用にリソースを制限した子パーティションを生成するといったことができる．この場合，RTOS は Linux のパーティションとその子パーティションが持つリソースにはアクセスできないし，逆も不可能である．

本方式ではメモリ割当てをゲスト OS の責任で行うため，ハイパーバイザ自体はメモリ割当て機構を持たない．また，システム起動時にはハイパーバイザはルートパーティションの初期設定のみを行えばよく，その他のパーティション階層の構築はゲスト OS によって順次行われる．このように，ハイパーバイザ自体は最小限の機能を提供するにとどめ，その他の機能はゲスト OS で実装する．これによりハイパーバイザの規模や複雑さを削減し，ハイパーバイザ自体に不具合が入り込むことによるシステム全体の信頼性やセキュリティの低下を避けている．

4.2　メモリ領域管理

メモリ領域を子パーティションに譲渡する場合，まず `createSection` を用いて親パーティションが持つメモリ領域の一部から新しいメモリ領域を生成し，セクション識別子（secID）を割り当てる．次に `attachSection` を用いて新規メモリ領域を子パーティションに追加する．このとき，子パーティションの parID，新規メモリ領域の secID，および子パーティションのゲストアドレス空間上での開始アドレスを指定する．なお，このメモリ領域がメモリアクセス制御回路のエントリ 1 個に対応する．

さらに，この新規メモリ領域を別の子や親自身に追加することにより，メモリ領域を共有できる．共有メモリ領域はパーティション間で情報を交換するために使用される．

4.3　デバイス管理

各デバイスにはシステム内でユニークな識別子（devID）を定義する．`leaseDevice` を用いて子パーティションにデバイスの譲渡を行うには devID を指定する．デバイスの共有は許可しない．子パーティションにデバイスを譲渡している間は，親パーティションからそのデバイスへのアクセスは禁止される．

デバイスを譲渡されたパーティションはそのデバイスを占有して使用できる．ゲスト OS は排他制御を考慮せずに単一 OS の場合と同じようにデバイスを直接操作できるので，操作に余分なオーバヘッドが加わらない．

複数のゲスト OS からデバイスを操作したい場合には，仮想デバイスとしてそのデバイスの処理を受け持つゲスト OS を用意し，共有メモリ領域経由で他からの要求を受けるようにパーティションを構成するとよい．

すべてのゲスト OS が操作する基本的なデバイスである割込みコントローラとタイマについては，後述するようにハイパーバイザが仮想デバイスとその操作 API を提供する．これらはゲスト OS の切替えにも深く関わっており，ハイパーバイザとは不可分である．

4.4　仮想割込みコントローラ

ハイパーバイザは仮想的な割込みコントローラをゲスト OS に提供する．この仮想化により，割込みコントローラの排他制御，および割込み要因単位でのアクセス制御を実現する．

ハイパーバイザが提供する API は割込み状態の取得（`getIRQ`），割込みの許可（`enableIRQ`），禁止（`disableIRQ`），クリア（`ackIRQ`）の 4 個である．一般にこれらの操作は，複数の割込み要因に対応するビットマスクを操作用のレジスタに対して読み書きすることで実行される．しかし，単純にゲスト OS から指定された値を設定すると，他のゲスト OS に属するデバイスの割込み要因も操作してしまう．

`enableIRQ`，`disableIRQ`，`ackIRQ` では上記の問題に対応するため，引数に指定されたビットマスクのうち，呼び出し元に属するデバイスの割込み要因のみが操作されるようにビットマスクを修正して，実割込みコントローラに設定する．`getIRQ` では呼び出し元に属するデバイスの割込み要因に関してのみ状態を返す．

4.5　仮想タイマ

ハイパーバイザは，1 個の実タイマを仮想化することにより，各パーティションに対して仮想タイマを 1 個提供する．この仮想タイマは操作 API（`setTimer`）で設定された時刻になると割込みが発生し，その割込みは時刻を設定したゲスト OS に配送される．

SoC 内に用意される実タイマは 1 個である．そのため，各パーティションの仮想タイマを設定時刻の順にソートし，最も早い仮想タイマの設定時刻を実タイマに設定する．

また，仮想タイマは，ゲスト OS 間で時分割処理を行うための割込みを発生させるために，ハイパーバイザからも直接使用される．

4.6　複数ゲスト OS の動作

本方式ではプロセッサ上で動作する各ゲスト OS に個別のパーティションを割り当てる．ハイパーバイザはこれらのゲスト OS に対する割込みの配送と，それに

ともなうゲスト OS とパーティションの切替えを行う．

基本的な割込み配送シーケンスは以下のとおりである．まず，動作中のゲスト OS でいったん割込みを受けて，ハイパーバイザに通知する．次にハイパーバイザは割込み処理監視回路の時間計測を停止する．その後，その割込みを処理すべきゲスト OS を選択し復帰する．

ゲスト OS の切替えが発生する場合には，ハイパーバイザは，プロセッサのバスマスタ ID に対応するアクセス制御情報をメモリおよびデバイスアクセス制御回路に再設定することにより，パーティションの切替えを行う．

この際，ハイパーバイザはゲスト OS の状態保存・復帰を行う．ゲスト OS の状態には，各種レジスタの値に加え，Translation Look-aside Buffer（TLB）等のメモリ管理機構の状態が含まれる．

さらに，キャッシュのフラッシュと無効化も必要である．もしキャッシュ上にエントリが残っていた場合，次のゲスト OS にその情報が漏洩してしまう．また，ゲストアドレス空間から実メモリ空間へのマッピングが変更されるため，パーティション切替え後にキャッシュ上のダーティなエントリがフラッシュされると，誤った実メモリへ書き込みが行われてしまう．

割込み発生後一定時間以内にゲスト OS がハイパーバイザに通知を行わない場合，割込み処理監視回路がタイムアウトを起こしてプロセッサにリセットがかかる．リセット後に起動されたハイパーバイザは，実行中であったゲスト OS を異常と見なし，該当するパーティションを無効化する．具体的には，このパーティションに属するデバイスの割込みを禁止およびクリアし，以降はこのゲスト OS への切替えを行わない．それ以外のデバイスの割込みに関する情報は割込みコントローラとデバイスに残っているので，それを参照して本来処理すべきゲスト OS を選択し復帰する．

5. 性能評価

これまで述べた SoC 仮想化方式の基本性能を確認するため，FPGA を用いて試作を行い，ハイパーバイザが提供する各種 API の処理時間を測定した．

5.1 測定条件

試作に用いた FPGA は Xilinx 社製の Virtex-II Pro である．この FPGA はプロセッサコアとして PowerPC405 プロセッサ（PPC405）を内蔵している．PPC405 の命令およびデータキャッシュのサイズはそれぞれ 16 KB である．今回の試作ではプロセッサの動作クロックを 300 MHz に設定した．ゲスト OS が格納される外部メモリには動作クロック 100 MHz の SDRAM を使用した．保護領域は PPC405 のオンチップバスに接続し，ハイパーバイザのコードとデータはオンチップメモリに格納した．オンチップメモリはキャッシュ不可能な領域であるが，外部メモリより高速である．

ゲスト OS には API を呼び出すライブラリを提供し，各 API の処理時間としてゲスト OS 側でのライブラリ呼び出し時間を測定した．ゲスト OS としては測定用に作成した独自カーネルを用いた．このカーネルには割込み処理を行う最小限の機能のみを実装した．また，2 個のパーティションを生成し，ゲスト OS の切替えの有無およびキャッシュのフラッシュ量の最小・最大の状態を意図的に作り出して，各条件での時間測定を行った．別途，時間測定自体のコストを測定し，最終的な補正を行った．時間測定には周波数 100 MHz のリアルタイムクロック（RTC）を使用した．したがって測定結果の精度は 10 ns である．

5.2 測定結果

表 2 上段は各 API の処理時間の測定結果である．`getIRQ` は割込み状態を取得する API であり，`getIRQ (w/o cs)` がゲスト OS の切替えなしの場合，`getIRQ (Min w cs)`，`getIRQ (Max w cs)` がそれぞれゲスト OS 切替えありで最小の場合と最大の場合の測定結果を示す．また，`enableIRQ`，`disableIRQ`，`ackIRQ` は仮想割込みコントローラの制御，`readRTC`，`setTimer` はそれぞれ RTC の値の読み出しと仮想タイマの設定を行う API である．

測定結果を簡単にまとめると，`getIRQ` 以外では API 呼び出し 1 回あたり 10 μs のオーダであり，最大でも `setTimer` の 20 μs 程度で済む．しかし `getIRQ` では，ゲスト OS の切替えが生じない場合には 30 μs で済む

表 2　ハイパーバイザ API の処理時間
Table 2　Execution time of hypervisor API.

API 呼び出し	処理時間（ns）
getIRQ (w/o cs)	28,040
getIRQ (Min w cs)	93,360
getIRQ (Max w cs)	144,740
enableIRQ	11,080
disableIRQ	11,080
ackIRQ	10,200
readRTC	8,780
setTimer	19,740
ゲスト OS の切替え処理	**処理時間（ns）**
Register Save	3,160
Register Restore	3,540
TLB Save	7,860
TLB Restore	9,240
Cache Flush（Min）	21,280
Cache Flush（Max）	73,260

が，ゲスト OS の切替えが生じる場合に最小で 90 μs，最大で 150 μs に処理時間が増加する．

ゲスト OS の切替え処理の内訳については，レジスタの保存/復帰（Register Save/Restore），TLB の保存/復帰（TLB Save/Restore），キャッシュのフラッシュ（Cache Flush Min/Max）に処理を分けて測定した．表 2 下段にその結果を順に示す．

今回保存の対象としたレジスタ数は 51 個，TLB のエントリ数は 64 個である．PPC405 は組み込み用プロセッサとしては高機能であり，これらの数はけっして小さいものではないが，それぞれの保存/復帰は 10 μs 以下で済む．しかし，これらと比較してもキャッシュのフラッシュにかかる処理時間は 1 桁大きく，最大で 73 μs であった．

6. 考 察

6.1 スループットへの影響

本方式で処理時間のオーバヘッドとなるのはハイパーバイザの API 呼び出し時間である．これら API のうち，パーティション管理に関わる API の呼び出しは OS やアプリケーションの起動時のみであり，全体のスループットに対する影響はほとんどないと考えられる．それに対し，割込みコントローラの操作 API は外部割込みが発生するたびに必ず呼び出す必要があり，全体のスループットへの影響は大きい．タイマの操作 API も，短い周期でタイマ割込みを発生させる処理が存在する場合には，同様にスループットへの影響がある．

なお，プロセッサが備えるメモリ管理機構の操作やページテーブルの更新等，割込み処理関係以外の特権命令の処理とそれにともなう例外処理にハイパーバイザが関与しないので，ゲスト OS によるそれらの特権命令の実行にともなうオーバヘッドは生じない．

したがって，本方式におけるオーバヘッドは割込み処理に関わる API 呼び出しを考慮すればよい．ゲスト OS における一般的な割込み処理の流れは，デバイスの操作に加え，本 API の `getIRQ`，`disableIRQ`，`ackIRQ`，`enableIRQ` が 1 回ずつ呼ばれる．したがって，割込み 1 回あたりのオーバヘッドはゲスト OS 切替えなしの場合には 60 μs，ありの場合には最大 180 μs 程度増加する☆．

割込み処理に関わるオーバヘッドが全体のスループットに与える影響の大きさは，その割込みの周期に依存する．以降では，本方式を組み込みシステムに応用した場合の影響を，いくつかのユースケースにあてはめて考察する．

動画処理をリアルタイムに行う場合，フレーム単位で割込みが発生すると想定するのが妥当である．動画のフレーム速度が 30 fps の場合，割込み処理の周期は 33 ms となる．割込み処理 1 回あたりのオーバヘッドを最小 60 μs，最大で 180 μs とすると，周期に対するオーバヘッドの割合は最小で 0.18%，最大で 0.54%となり，特に問題とはならない．

ロボット制御等では，制御対象の固有振動数は 1〜50 Hz となることが多い[3)]．サーボモータの周期はその 10 倍程度で安定するので，2〜100 ms の周期を用いる．周期が 100 ms の場合には，周期に対するオーバヘッドの割合は 0.06〜0.18%であり，無視できるレベルである．しかし，周期が 2 ms の場合には，3〜9%となる．制御自体は可能と考えられるが，制御対象の固有振動数によってはオーバヘッドは無視できない大きさとなる．

当然ながら上記の考察は，使用するプロセッサやメモリの動作クロックに依存するものである．しかし，現在のデジタル機器に組み込まれているものと比較して，今回試作に使用した開発ボードの性能が特に優れているわけではない．たとえば携帯電話向けプロセッサでは動作クロックが 100 MHz を超えるものがすでに開発されている．したがって上記の考察は妥当であると考える．

また，ゲスト OS の切替えの時間が相対的に大きくなり無視できない場合は，マルチコアプロセッサを採用して，時間制約の強い処理を単独で専用コアに割り当てるとよい．

6.2 ハードウェア IP の規模

FPGA 上で試作したハードウェア IP の等価ゲート数を**表 3** に示す．

動作モード管理回路および割込み処理監視回路は簡単なオートマトンで構成可能なので，ハードウェア IP は比較的小規模で済む．

メモリアクセス制御回路には制御情報のエントリご

表 3 各ハードウェア IP の等価ゲート数
Table 3 Equivalent gate count for each design.

ハードウェア IP	等価ゲート数
動作モード管理回路	1,632
メモリアクセス制御回路	28,685
デバイスアクセス制御回路	8,519
割込み処理監視回路	350

☆ 本来は直接割込みコントローラを操作する場合の実行時間との差分をとらなければならないが，直接操作する場合は数バイトの I/O で済むため，本 API の実行時間そのものを増加分と見なす．

とに制御レジスタ，判定回路，およびアドレス変換回路が必要なので，比較的規模が大きくなっている．本試作のメモリアクセス制御回路では，1 エントリあたりバスマスタ ID に 3 ビット，ゲストアドレス空間の開始アドレス，終了アドレス，物理アドレスへのオフセットにそれぞれ 20 ビットの計 63 ビットを使用し，このエントリを 8 個用意した．また，デバイスアクセス制御回路では，1 エントリあたりデバイス数に相当する 10 ビットを使用し，このエントリを 8 個用意した．

ハードウェア IP 全体の規模はメモリアクセス制御回路のエントリ数に大きく依存すると考えられる．機能モジュールが多い場合やパーティション間通信のために共有メモリ領域を多く設定する場合には，エントリ数 8 個では足りず，エントリ数を増やす必要がある．この場合にはハードウェア IP の規模がより大きくなる．

なお，試作で実装したシステム全体の等価ゲート数は約 400 万ゲートであった．この値にはプロセッサやデバイスの IP，オンチップメモリがすべて含まれている．システム全体に対するハードウェア IP の割合は約 1%であった．

6.3 様々なプラットフォームへの適用可能性

本方式を既存のプラットフォームに対して適用する場合，プロセッサおよびバスに対する修正は基本的には必要ない．ただし，バスに流れるバスマスタ ID を用いてアクセス制御を行っているため，バスの仕様としてアクセス要求にバスマスタ ID が含まれていることが必要である．そうでない場合には，バス上のアクセス要求の発行元を識別するためにさらに付加的なハードウェアが必要になる．

仮想化のために追加するハードウェア IP は，基本的な機能や構造は特定のプロセッサやバスへの依存性を極力排した設計にし，プラットフォーム独立性を高めている．そのため，各ハードウェア IP のバスへの接続部分とエントリコードの状態管理部分を，使用するバスおよびプロセッサの仕様に合わせて修正することで，本方式を新しいプラットフォームに適用できる．

バスとの接続部分に関しては，基本的には各ハードウェア IP の制御レジスタへのアクセスができるように，接続するバスに合わせて信号線の対応やアドレスのデコード方法を変更するのに加え，必要に応じてアクセス制御のための保護回路でアクセスをブロックする方式の変更が必要である．割込み処理監視回路とメモリおよびデバイスアクセス制御回路は，この部分の修正のみで対応できる．

動作モード管理回路については，使用するプロセッサの命令セットに合わせてエントリコードを設計し，その命令列と命令数および配置アドレスに合わせて，HV モードへの正しい遷移のチェックに必要な状態遷移を決定して組み込むことで，新しいプロセッサに対応できる．

ゲスト OS については，割込み処理ルーチンに修正を加える必要がある．一般に割込みコントローラの実装に依存するコードは OS 内でモジュール化されている．そのコードをハイパーバイザが提供する割込みコントローラ操作用 API の呼び出しに置き換えればよい．この修正量はゲスト OS を対象の SoC に移植するための修正量と比較して十分に小さい．

ゲスト OS としては汎用 OS として Linux，リアルタイム OS として μITRON を想定している．適用対象としては，主にデジタルメディア機器でのアプリケーションプロセッサを想定している．なお，マイクロ秒オーダの応答時間を要求される機械制御のようなハードリアルタイム用途は想定していない．本方式ではデバイスの割込み発生からハイパーバイザへの通知まで時間が，本来その割込みを配送すべきゲスト OS ではなく切替え前のゲスト OS に依存し，割込み応答時間の保証が困難なためである．ハードリアルタイム処理への対応は今後の課題である．

7. 関 連 技 術

従来より，PowerPC[1] や Intel VT-x/i[2] のようなプロセッサにハードウェアによる仮想化支援機構を搭載する技術と，Xen[4] のようにソフトウェアによって仮想化を実現する技術が存在する．これらの仮想計算機技術は，ともにプロセッサ上で動作するゲスト OS に対して仮想計算機を提供している．

前者の仮想化支援機構を持つプロセッサでは，多数の動作モードを持ち，ハイパーバイザにゲスト OS より上位の特権モードを割り当てることができる．さらに，メモリ管理機構や割込みベクタテーブルの修正を監視して，ハイパーバイザに制御を移す設定が可能である．我々の方式では仮想化支援機構を持たないプロセッサもターゲットにしており，プロセッサの外部で動作モードを管理することでハイパーバイザとゲスト OS の実行を区別する．さらに，我々の方式は，仮想化支援機構を持たないプロセッサでは悪意のあるソフトウェアによるプロセッサのメモリ管理機構や割込みベクタテーブルの修正が起こりうることを前提にしており，追加ハードウェア IP によりこれに対処して仮想化を実現している．

本方式は，ゲスト OS の一部の命令を静的に修

正する必要がある点では，Xen に代表される Para-virtualization 技術に属する．Xen では仮想化できない命令を hypercall と呼ばれるハイパーバイザ呼び出しに書き換える．それに対して我々の方式では，主に割込み処理ルーチンに対して修正を行う．ゲスト OS によるプロセッサのメモリ管理機構の操作やページテーブルの更新には制限を加えず，それらの命令をハイパーバイザでエミュレーションする必要はない．

DMA に対して仮想化を提供する技術として Intel VT-d [5] が存在する．VT-x/i がプロセッサの仮想化支援機構であるのに対し，VT-d はチップセットの仮想化支援機構である．VT-d は PCI 接続のデバイスに対してそのデバイスが割り当てられた VM のゲストアドレスから実アドレスへのマッピングを管理し，そのデバイスからの DMA をチップセットでアドレス変換することにより I/O を仮想化する．アドレス変換のマッピング情報およびアクセス権情報を登録しておき，DMA のアクセス元の識別情報をキーにしてアクセス制御を行うことは，DMA に対する仮想化の本質であり，本方式も VT-d も同じである．

そのほか，プロセッサの外部でのアクセス制御技術としては，バス上でマルチプロセッサのアクセス元を区別しフィルタリングを行う技術 [6] が存在する．

8. おわりに

本稿では SoC 全体を仮想化するアーキテクチャを提案し，FPGA を用いた試作と性能評価を行うことで本方式の実現可能性を検証した．様々な機能モジュールを含む SoC 全体のアクセス制御はプロセッサによるメモリ管理機構だけでは解決できず，本方式のようにメモリやデバイスをバスに接続するコントローラにアクセス制御機構を集約する手法が有効である．また，追加ハードウェア IP およびファームウェアの修正で対応する方式は，既存のプラットフォームや IP 資産を大きな変更なく再利用できる点で，開発コストの観点からも望ましい．

今後の課題としては，ハードリアルタイム処理への対応，マルチコアプロセッサへの対応，パーティション間通信機能があげられる．今後はこれらの課題に取り組み，本方式の実用化に向けて検証と機能拡張を続ける予定である．

参考文献

1) IBM: *Book III: PowerPC Operation Environment Architecture* (2003).
2) Intel: *Intel Virtualization Technology Specification for the IA-32 Intel Architecture* (2005).
3) 阪田史郎，高田広章：組込みシステム，オーム社 (2006).
4) Barham, P., et al.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.164–177 (2003).
5) Abramson, D., et al.: Intel Virtualization Technology for Directed I/O, *Intel Technology Journal*, Vol.10, No.3, pp.179–192 (2006).
6) Inoue, H., et al.: FIDES: An Advanced Chip Multiprocessor Platform for Secure Next Generation Mobile Terminals, *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp.178–183 (2005).

(平成 19 年 1 月 19 日受付)
(平成 19 年 5 月 25 日採録)

矢尾　浩（正会員）
平成 3 年京都大学工学部情報工学科卒業．平成 6 年同大学大学院工学研究科修士課程修了．同年（株）東芝入社．以来，計算機アーキテクチャ，システムソフトウェア等の研究に従事．

吉井謙一郎（正会員）
平成 10 年早稲田大学理工学部電気工学科卒業．平成 12 年同大学大学院理工学研究科修士課程修了．同年（株）東芝入社．現在，高信頼システム，システムソフトウェア等の研究に従事．

金井　達徳（正会員）
昭和 59 年京都大学工学部情報工学科卒業．平成元年同大学大学院工学研究科博士後期課程研究指導認定退学．同年（株）東芝入社．以来，高信頼システム，マルチメディアサーバ，データベースシステム，システム LSI 設計支援技術等の研究開発に従事．電子情報通信学会会員．