Vol.2011-OS-118 No.1
2011/7/27

メニーコア向けシステムソフトウェア開発のための実行環境の設計と実装

下沢 拓[†1] 石川 裕[†1,†2] 堀 敦史[†2]
並木 美太郎[†3] 辻田 祐一[†4]

本稿では，メニーコア混在型システム向けのオペレーティングシステムのためのハードウェア抽象化層 AAL を設計する．AAL は，抽象化による移植可能性の向上を目的とし，抽象化層の摸擬環境を作成することによりメニーコア混在型システム向けのオペレーティングシステムの開発を可能にする．さらに，マルチコアシステムでメニーコア混在型システムを摸擬した環境での AAL の実装についても述べる．

Design and Implementation of Development Environment for Systems Software for Manycore Architecture

Taku Shimosawa,[†1] Yutaka Ishikawa ,[†1,†2]
Atsushi Hori ,[†2] Mitaro Namiki [†3]
and Yuichi Tsujita [†4]

AAL, an abstraction layer of operating systems for manycore accelerators is designed in this report. AAL is aimed to provide portability of operating systems for manycore accelerators. AAL enables development of the operating systems without any manycore accelerators by implementing emulation of the layer. An implementation of AAL for a manycore emulation environment on a multicore CPU is also presented.

†1 東京大学 / The University of Tokyo
†2 理化学研究所計算科学研究機構 / Advanced Institute for Computational Science, RIKEN
†3 東京農工大学 / Tokyo University of Agriculture and Technology
†4 近畿大学 / Kinki University

1. はじめに

プロセッサの並列性を極端に増やした形として Echelon[1] や Larrabee[2], Knights Corner[3] といったメニーコアプロセッサがある．このようなプロセッサは，システム上で単独で動作するものと，別にホストとなる CPU をもちアクセラレータという形でシステムに存在するものと二種類がある．我々はアクセラレータ型のメニーコアボードを搭載したシステムに着目する．現在 GPU を搭載したクラスタが数多く存在するが[4]，メニーコアボードは今後 GPU を置き換えていくものと考えている．実際，メニーコアプロセッサをアクセラレータとして装備した高性能計算システムの導入がすでに発表されている[5]．

このようなアクセラレータ型のメニーコアボードを搭載した環境においては，GPU のような他のアクセラレータと異なり，汎用プロセッサをベースとしており，メニーコアプロセッサ自身がデバイスの操作をすることができるという特長がある．この特長を生かすことにより，メニーコアプロセッサのみでの独立した並列計算の実行や，低遅延な通信を行うことができると考えられる．システムソフトウェアの面から，このことを考えたとき，ホストおよびメニーコアプロセッサで資源管理を一元化したオペレーティングシステムが必要になる．しかし，Linux のような汎用 OS をそのまま利用するにはいくつかの課題が存在する．第一に，対象とする環境ではメニーコアとホスト CPU のメモリ空間が独立しており，SMP や ccNUMA のような共有メモリを前提としてデータ構造を配置している OS を使用することは難しく，大きなオーバーヘッドが生じる．第二に，メニーコアプロセッサのコアあたりの演算性能やキャッシュ性能は低く，単純にホストと同様のオペレーティングシステムのコードを実行した場合には，アプリケーションに対するキャッシュ等による性能への影響が大きくなる．

メニーコアを搭載した環境向けのオペレーティングシステムを開発するには，ハードウェアが必要になるが，現在，容易に手に入るハードウェアは少ない．そこで，我々は，抽象化することにより多くの種類のメニーコアへの移植を可能とすることと，その抽象化層を通してメニーコアをエミュレーションすることにより，入手困難なメニーコアボードのない環境でもオペレーティングシステムの開発を目的として，抽象化層 Accelerator Abstraction Layer (AAL) の設計を行う．また，既存のマルチコアシステムにおいて前述した環境を摸擬する実行環境 (Manycore Emulation Environment, MEE) を作成するために，AAL の実装を行う．AAL を用いた MEE の利点は，シミュレータ[6] や FPGA ボード[7,8] を用いた場合と比較して，より高速に OS を実行することができ，また，実際の I/O デバイスを使用

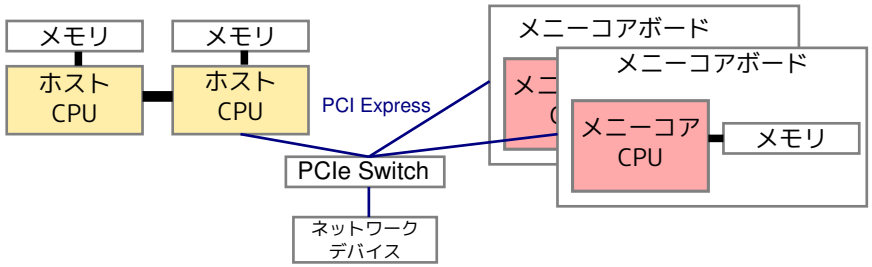

図 1 対象環境の概略図

することができる長所がある．

本稿の構成は，以下のとおりである．次節において，対象とする環境及び AAL の設計の概略について述べ，第 3 節で，AAL のユーザーレベルおよびカーネルレベルのインタフェースについて述べる．第 4 節では，MEE の概要とそのための AAL の実装について明らかにし，第 5 節で，メニーコア向けオペレーティングシステムの関連研究について述べ，第 6 節でまとめる．

2. 抽象化層の概略

2.1 対 象 環 境

本稿で提案する抽象化層は，**図 1** に示すようなハードウェア環境におけるオペレーティングシステムを対象とする．ハードウェア環境は，ホストとなるマルチコア CPU に，メニーコアプロセッサを搭載したボードが PCI Express 等の I/O バス経由で一枚もしくは複数枚が接続されたものである．また，メニーコアボードには，メモリが接続されており，I/O バスを通じてメモリマップされている領域を除いて，ホストと物理メモリ空間は独立であるとする．

2.2 抽象化層の要件

前節で述べたようなハードウェア環境で動作するオペレーティングシステムを，抽象化層上で動作させるには，抽象化層が以下の五点を考慮する必要がある．(i) ホストとボード間でアドレス空間が独立していること．前節で述べたようにホストと各ボード間のメモリアクセスには，I/O バスを経由する必要があるからである．(ii) 異なる種類のメニーコアボードに対してもオペレーティングシステムの移植を容易にすること．特定の種類に限った実装ではなく，アクセラレータ型メニーコアボード全般で使用可能なものであることが必要である．(iii) 複数のボードがシステム内に存在する可能性のあること，全体的な処理能力を得るために，I/O バスの帯域が十分確保できる限りにおいて，複数のボードが一つのノード内に使用されることは十分ありえるためである．(iv) オペレーティングシステムが最適な処理を行うためにメニーコアボードに関する十分な情報を提供するインタフェースを用意すること．抽象化しつつもハードウェアの十分な情報を与えることが，OS がスケジューリング等を考慮するうえで必要であるためである．(v) 開発に必要なデバッグ機能等の開発支援機能を提供すること．この抽象化層の目的を踏まえると，開発にも用いられるものであり，各種デバッグを行うための機能を提供することは，当然要求されることと考えられる．

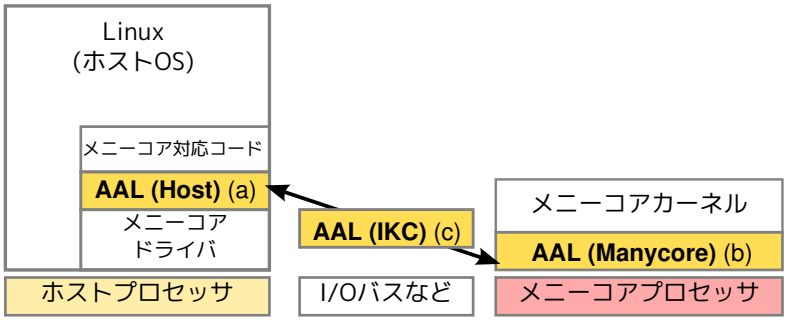

図 2 AAL の概略図

2.3 AAL の概要

本稿で提案する AAL(Accelerator Abstraction Layer) は，**図 2** で示すように，(a) ホスト側で動作する汎用 OS，(b) メニーコアプロセッサで動作するカーネル，(c) ホスト OS とメニーコア OS の通信機構の 3 部分から構成される．これは，前節で述べたようにメモリ空間が独立しているために，同一のカーネルコードをホストとメニーコアで動作させることは困難であるためである．ただし，メニーコアで動作させるカーネルの数は，一つとは仮定しない．また，アクセラレータ型のシステム構成であるため，OS の起動に関する限りにおいては，ホスト OS 側を主たるものとし，メニーコアボードを従たるものとして，メニーコアボード側のカーネルはホストの OS から起動されるものとする．

AAL を用いた OS の動作の概要は以下の通りである．ホスト OS では，AAL のインタフェースを使用し，メニーコアへの対応コードを OS に追加する．このコードは各メニーコアの初期化処理を含み，適宜必要な段階でこのインタフェースを通じ，メニーコアボードを

初期化し，メニーコア OS を起動する．AAL は，対応するハードウェアデバイスもしくは疑似デバイスを用いて，OS 起動要求やその他のホスト OS からの要求に応じて必要な処理を行う．通信の要求がある場合には，通信機構の抽象化層を通り，同様に対応するデバイスを用いて通信を行う．メニーコア側では，ホストに対して処理を行う必要のあるとき，I/O 処理を行うときに，AAL を経由して必要な処理を行う．

前節で掲げた考慮すべき点について述べると，AAL は，以下の特徴を有する．(i) メモリマップや通信機構を通じて独立メモリアドレス空間での明示的な処理を提供する．(ii) 各デバイスに対する操作は，関数ポインタを通じて，実際のハードウェアの種類に応じたコードが実行されるため，複数の種類のハードウェアに対応した処理が行うことができる．(iii) 各メニーコアデバイスに応じて管理構造体および仮想デバイスがホスト OS に作成されるため，それぞれのデバイスへの操作は独立に扱われる．(iv) インタフェースとなる関数には，情報を取得する関数が用意される．(v) メモリのダンプや実行の停止，仮想コンソールといったインタフェースを提供することにより，デバッグ環境を実現している．これらの詳細については，次節で述べる．

3. 抽象化層の設計

本節では，抽象化層である AAL の詳細な設計について述べる．図 2 で示したように，AAL のインタフェースは次の 3 部分からなる．(a) ホスト OS 内で使用されるメニーコアのデバイスドライバに対する操作のインタフェース，(b) メニーコアカーネル内で使用されるメニーコアハードウェアに対する操作のインタフェース，(c) ホスト OS とメニーコアカーネル間の通信機構である．以下では，最初に，ホスト OS 側のインタフェースについて，カーネルレベルとユーザーレベルのインタフェースについて記述する．その後，メニーコア側の関数インタフェース，通信機構のインタフェースについて述べる．

3.1 ホスト側インタフェース

AAL では，ホスト OS からメニーコアデバイスを操作する場合には，AAL で定義されるインタフェースとなる関数を呼び出し，各デバイスに対応したドライバはそのインタフェースに対応する関数を用意する．ホスト側のインタフェースとなる関数群について**表 1** にその一覧を示す．

本節の以降では，まずカーネル内でのインタフェースについて記述し，最後にユーザーレベルとのインタフェースである仮想デバイスドライバについての説明を述べる．

表 1 AAL のホスト側インタフェース関数

aal_host_create_os(cpucores, memory)	メニーコアカーネル構造体の作成・初期化
aal_host_grant_resource(os, resource)	メニーコアカーネルへの資源の付与
aal_host_revoke_resource(os, resource)	メニーコアカーネルからの資源の剥奪
aal_host_load_os_image(os, file_or_memory)	メニーコアカーネルイメージの読み込み
aal_host_boot(os, option)	メニーコアカーネルの起動
aal_host_shutdown(os, reset_or_stop)	メニーコアカーネルの再起動・停止
aal_host_destroy_os(os)	メニーコアカーネル構造体の破棄
aal_host_map_memory(os, physaddr)	メニーコアのメモリのマップ
aal_host_unmap_memory(os, virtaddr)	メニーコアのメモリのマップ解除
aal_host_copy_from_remote_memory(os, hostvirt, mcphys)	メニーコアからのメモリコピー
aal_host_copy_to_remote_memory(os, mcphys, hostvirt)	メニーコアへのメモリコピー
aal_host_issue_interrupt(os, core, vector)	メニーコアへの割り込み発生
aal_host_register_interrupt(os, vector, handler)	メニーコアからの割り込みハンドラ登録
aal_host_unregister_interrupt(os, vector, handler)	メニーコアからの割り込みハンドラ解除
aal_host_dma_initialize(os, channel, opts, handler)	DMA チャネルの初期化
aal_host_dma_add_request(channel, request)	DMA リクエストの追加
aal_host_dma_remove_request(channel, request)	DMA リクエストの削除
aal_host_dma_destroy(channel)	DMA チャネルの終了処理
aal_host_debug_set_breakpoint(os, core, addr, handler)	ブレークポイントの設定
aal_host_debug_unset_breakpoint(os, breakpoint)	ブレークポイントの設定
aal_host_debug_pause_kernel(os, core)	メニーコアカーネルの停止
aal_host_debug_resume_kernel(os, core)	メニーコアカーネルの再開
aal_host_debug_get_register(os, core, info)	レジスタ情報の取得
aal_host_get_info(type, info)	デバイス情報の取得

3.1.1 OS の起動・終了

最初に，メニーコアカーネルの起動や終了にかかわるインタフェースについて述べる．メニーコアカーネルを起動するには，最初に CPU コアとメモリをメニーコアデバイス上で確保する (aal_host_create_os)．このとき，ホスト OS 内では，メニーコアデバイス上の OS に対応する管理構造体が生成され，後述の仮想デバイスが作成される．なお，CPU コアやメモリといった資源の動的な追加や削除が可能な場合は，それぞれ aal_host_grant_resource および aal_host_revoke_resource 関数を用いる．次に，aal_host_load_os_image 関数により，カーネルのイメージをメモリ上に読み込む．イメージとしては，ファイルもしくはメモリを指定する．そして，aal_host_boot 関数により，メニーコアカーネルがブートされる．

メニーコアカーネルを終了させるためには，後述するカーネル間通信機構を用いてシャットダウン要求を送るか，aal_host_shutdown 関数により強制的にリセットもしくは停止させる．カーネルの停止後には，aal_host_destroy_os 関数により OS インスタンスを破棄

する．

3.1.2 メモリ操作

メニーコアのメモリ操作は，メモリマップによるアクセスとメモリ間コピーを行うものがある．前者の関数は，aal_host_map_memory と aal_host_unmap_memory でありメニーコアのメモリ空間の物理アドレスを指定し，その領域をマップする．マップをしない，あるいはマップのできないアドレスに対しては，aal_host_copy_from_remote_memory および aal_host_copy_to_remote_memory 関数によってコピー操作を行う．

3.1.3 割り込み処理

割り込みの操作には，ホストカーネルからメニーコアカーネルに対する割り込みと，その逆に関するものが存在する．前者としては，割り込みを発行する aal_host_issue_interrupt 関数がある．後者には，メニーコアからの割り込みに対するハンドラを追加する aal_host_register_interrupt および解除する aal_host_unregister_interrupt 関数がある．

3.1.4 DMA 処理

DMA 処理は，aal_host_dma_initialize により DMA デバイスの初期化と完了通知割り込みハンドラの登録を行い，aal_host_dma_add_request によりリクエスト構造体を登録し，aal_host_dma_start により DMA の開始を指示する．リクエスト構造体は，実際のハードウェアで用いられる構造体を抽象化したもので構造体内のフラグへのアクセスには，aal_host_dma_request_get_flag 関数等を用いる．また，完了後には aal_host_dma_remove_request により終了したリクエスト構造体を削除する．DMA デバイスの使用後の終了処理は，aal_host_dma_destroy 関数により行う．

3.1.5 デバッグインタフェース

メニーコアカーネルに対するデバッグインタフェースとしては，ブレークポイントの設定と解除を行う aal_host_debug_(un)set_breakpoint 関数がある．カーネルの一時停止を行う aal_host_debug_pause_kernel 関数及び再開を行う aal_host_debug_resume_kernel 関数，CPU のレジスタ等の情報を取得する aal_host_debug_get_register 関数がある．

なお，端末操作のためのコンソールデバイスの実現には，後述する通信機構を用いて実現する．

3.1.6 ハードウェア情報

CPU コア数，メモリのサイズ，可能な割り込みの数などのハードウェアの情報は，aal_host_get_info 関数を通じて取得する．ただし，より詳細なコアやメモリのトポロジなどの情報の仕様に関しては，今後の検討課題である．

3.1.7 仮想デバイスドライバ

仮想デバイスドライバは，各メニーコアデバイスやその上のカーネルに対する起動や停止といった処理を行うインタフェースを提供するものである．

AAL は，各メニーコアデバイスごとに，/dev/mcd*n* と呼ばれるキャラクタデバイスを作成する．また，各デバイス上のカーネルごとに，/dev/mcos*n* と呼ばれるキャラクタデバイスを作成する．

/dev/mcd*n* は，新しいメニーコアカーネルの生成，デバイスに対する直接操作に用いられるキャラクタデバイスである．メニーコアカーネルの生成は，ioctl システムコントロールにより行われ，コア数とメモリ量の指定 (IOCTL_MCD_SET_NUM_CORES，IOCTL_MCD_SET_MEMORY_SIZE) ののちに，OS 管理構造体と仮想デバイスの作成指示 (IOCTL_MCD_CREATE_OS) という流れで実行される．カーネル停止後の構造体と仮想デバイスの破棄指示も，IOCTL_MCD_DESTROY_OS により ioctl を呼ぶことで行われる．また，メニーコアデバイス内のメモリの直接操作は，ファイル操作と同様に read，mmap 等システムコールにより行う．

/dev/mcos*n* は，メニーコアカーネルデバイスに対する操作を行う際に用いられる．主に ioctl システムコールによって制御され，カーネルイメージファイルのロード (IOCTL_MCOS_LOAD_IMAGEFILE)，カーネルの起動，停止・終了 (IOCTL_MCOS_BOOT, IOCTL_MCOS_RESET) カーネルの状態の取得 (IOCTL_MCOS_GET_STATUS)，資源の割り当て変更 (IOCTL_MCOS_ADD_MEMORY, IOCTL_MCOS_ADD_CPU 等) などを行うことができる．また，デバッグ操作として，端末デバイスの確保 (IOCTL_MCOS_CREATE_TTY)，実行の停止 (IOCTL_MCOS_DEBUG_STOP_CPUS)，CPU 状態の取得 (IOCTL_MCOS_DEBUG_GET_REGS 等) が行われる．メモリダンプの取得については，read や mmap により行うことができる．

3.2 メニーコア側インタフェース

メニーコアプロセッサ側の抽象化層としては，前節で挙げたメモリ処理・割り込み処理・DMA 処理は同様であるが，これに加えて，CPU やメモリ，デバイス情報の取得関数が存在する．

また，割り込みコントローラの処理として，初期化を行う aal_mck_intc_init 関数，割り込みの種類および宛先の定義を行う aal_mck_intc_set_interrupt 関数，マスク処理を行う aal_mck_intc_(un)mask_interrupt 関数がある．

3.3 カーネル間通信機構

カーネル間通信機構のインタフェースには，カーネル間通信チャネルを作成する

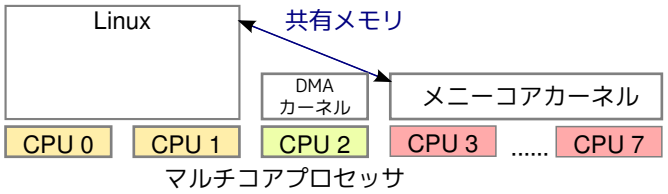

図 3 MEE の概略図

aal_ikc_create_channel 関数がある．この関数では，キューのサイズや通信完了通知ハンドラといったパラメータを指定する．チャネルからデータを取り出す aal_ikc_channel_recv 関数，チャネルにデータを送る aal_ikc_channel_send 関数，また，通信の割り込み禁止等の処理を制御する aal_ikc_channel_set_flag 関数がある．チャネルの破棄には，aal_ikc_destroy_channel 関数を用いる．

4. 摸擬環境の設計と実装

本節では，メニーコア模擬環境 (MEE) の設計とその AAL の実装について述べる．最初に設計の概要について述べ，変更可能なパラメータの種類と，この模擬環境で検証可能な範囲について述べる．最後に，実装の概要を示す．

4.1 設　　計

メニーコア模擬環境 (MEE) の設計概要を**図 3** に示す．MEE は，マルチコアプロセッサのコアを三つの用途に分けて用いる．一つは，ホストプロセッサに相当するものであり，この上で汎用 OS を動作させる．もう一つは，DMA エンジンを模擬するものであり，この上では汎用 OS もしくはメニーコアからの要求に応じてメモリコピーを行うものである．最後のもう一つは，残りのコアをメニーコアプロセッサとみなして，メニーコア用のカーネルを動作させるものである．

4.2 適用範囲とパラメータ

MEE は，エミュレーション環境ではあるが，それぞれのカーネルを各コアでネイティブに実行するために，シミュレータを用いた場合と異なり，OS を高速に動作させることができるため，OS の機能検証を行うことができる．また，I/O デバイスについても，実際のハードウェアを用いることにより，動作の検証を行うことができる．

他方，MEE の制限としては，ホスト CPU と異なるアーキテクチャについての検証は行えないことがある．また，パフォーマンスに関する検証も行うことができない．これは，CPU の処理能力が動作周波数を除いては，基本的に変更不可能であること，また，MEE では，SMP などの共有メモリ型のマルチコアシステムを想定しており，キャッシュ一貫性プロトコル等の動作によって，実際のハードウェアとはキャッシュの挙動が大きく異なることがあることであり，これに関しても明示的な制御は困難である．

MEE で調節可能なパラメータは次のものである．

- $F_{manycore}$: メニーコアカーネルを動作させる CPU の動作周波数（ただし，操作可能なハードウェアに限る）
- L_{DMA} : DMA 実行時の遅延．DMA コアのコピー操作に遅延を加えることで実現する．
- L_{memcpy} : マップを伴わないメモリコピー時やメモリ操作の遅延．
- M_{DMA} : DMA をソフトウェアとハードウェアどちらで行うかの指定．これは，IOAT[9] といった，CPU から使用可能な DMA エンジンのある場合に限る．

いずれもパフォーマンスの調整をある程度行うことができるが，完全な性能の検証を行うことはできない．

4.3 実　　装

MEE 上の AAL を実装するためには，マルチコア CPU のネイティブ環境において複数のカーネルが実行できる必要がある．我々は，SHIMOS[10] をこの機能の実現に使用した．これは，カーネルに対して変更を加えることで資源利用を制限し，仮想マシン等のオーバーヘッドなしに複数のカーネルを実行するものである．

これを用いるために，汎用 OS としては Linux 2.6.32(amd64) を使用した．MEE では，各カーネルは，以下のような形で実行される．

Linux は，カーネルパラメータに従い，CPU コアとメモリの一部のみ使用をして起動をする．ただし，メモリマップに関しては全物理メモリのマップを Linux が保持する．メニーコアカーネルの起動は，メニーコアカーネルに割り当てられたメモリ上に，カーネルコードを展開し，プロセッサ間割り込みを用いて，メニーコアとして割り当てられたコアを起こし，64bit モードへの移行などの初期化処理を通して起動を行う．メモリマップ等は，前述のとおり物理メモリをすべてマップしているために，Linux 側からは，単純なアドレス変換 (__va マクロ) によって行われる．割り込み処理については，Linux，メニーコア側互いにプロセッサ間割り込みを用いる．

DMA コアは，Linux カーネルのメモリ空間で動作するが，Linux カーネルとは完全に独立して動き，リクエストキューと通知領域となるメモリ領域を初期化時に用意し，Linux もしくはメニーコアカーネルの，そのメモリ領域への書き込みを契機としてリクエストの処理

を開始する．メニーコアカーネルは，同様に全物理メモリ領域をマップし，割り込み等の処理もプロセッサ間割り込みを用いることで行う．

5. 関連研究

ヘテロ構成に限らずメニーコア向けのOSの研究はいくつかある．本稿で提案した抽象化層に関連して，本節ではそれらの資源管理について述べる．

Corey[11]では，メモリ領域やカーネル内構造体のプロセス間の共有について，明示的に指定することによりfalse sharingを減らしカーネル内のキャッシュ競合を減らしている．これは，アプリケーションから指定させているものであり，システムコールがそのインタフェースとなっている．有関係にある資源を一つにまとめたshareと呼ばれるオブジェクトを最初に確保し，それに対してメモリ領域やCPUのコアおよびデバイスの割り当てを要求するというインタフェースを提供している．

Tessellation[12]は，マシンの資源を分割し，Cellと呼ばれる単位に専有させ，その上でアプリケーションやシステムサービスを動かし，それらのCellの時間・空間のスケジュールを行うものである．資源管理については，資源分割のアーキテクチャ実装部分であるPartition Mechanism LayerとCellの要求から，決められたポリシーに基づいて実際の割り当てを決定するPartition Managerなどから構成されている．

Barrelfish[13),14]では，OSはCPUコアやメモリ，デバイスの距離等についての知識(system knowledge base)を保持し，その情報を利用して多様なハードウェア環境への対応を可能とすることを目的としている．ただし，これらの適切な収集と保存形式が課題となっている．

6. おわりに

マルチコアプロセッサに加えて，メニーコアプロセッサをアクセラレータとして装備したノード構成のクラスタが今後増えていくと考えられる．本稿ではそのようなクラスタ環境におけるオペレーティングシステムの開発に供するために，ハードウェアの抽象化層AALの設計を示した．この抽象化層の目的は，第一にメニーコアデバイスの抽象化であり，これによりその上で動作するオペレーティングシステムのコードの移植性を高めることである．第二の目的は，メニーコアを直接模擬する代わりに，抽象化層を既存マルチコアCPUで模擬することにより，メニーコア用オペレーティングシステムの機能実証を可能とすることである．さらに，本稿では，マルチコア上で動作する摸擬環境MEEのためのAALの実装を示した．MEEの制限としては，オペレーティングシステムのパフォーマンス等の検証は困難であることが挙げられる．しかし，機能の検証や，実際のデバイスを用いた検証が可能であり，シミュレータよりも高速な検証を行うことができるという利点が存在する．

今後の課題として，抽象化層で必要なものとして，ハードウェア情報の詳細な提供がある．これは，メニーコアの特性を生かしたものである必要があり，その例として，CPUやキャッシュ，メモリの性能，およびそのトポロジやバス構成といった情報が必要になると考えられる．

謝辞 本研究の一部は，科学技術振興機構戦略的創造研究推進事業(CREST)研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」研究課題「メニーコア混在型並列計算機用基盤ソフトウェア」による．

参考文献

1) Dally, B.: GPU Computing To Exascale and Beyond, http://www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf.
2) Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T. and Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing, *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, Vol.1, New York, NY, USA, ACM, pp.18:1–18:15 (2008).
3) Dubey, P. and Singhal, R.: Architecture Support for HPC Applications, Intel Develop Forum 2010.
4) TOP500.org: TOP 500 Supercomputing Sites - June 2011, http://www.top500.org/lists/2011/06.
5) Texas Advanced Computing Center: TACC Collaborates with Intel to Accelerate Open Science Research Using Intel MIC Processor Line, http://www.tacc.utexas.edu/news/press-releases/intel-collaboration.
6) Argollo, E., Falcón, A., Faraboschi, P., Monchiero, M. and Ortega, D.: COTSon: infrastructure for full system simulation, *SIGOPS Oper. Syst. Rev.*, Vol. 43, pp. 52–61 (2009).
7) Tan, Z., Waterman, A., Avizienis, R., Lee, Y., Cook, H., Patterson, D. and Asanović, K.: RAMP gold: an FPGA-based architecture simulator for multiprocessors, *Proceedings of the 47th Design Automation Conference*, DAC '10, Vol.1, New York, NY, USA, ACM, pp.463–468 (2010).
8) Tan, Z., Waterman, A., Cook, H., Bird, S., Asanović, K. and Patterson, D.: A case for FAME: FPGA architecture model execution, *SIGARCH Comput. Archit. News*, Vol.38, pp.290–301 (2010).
9) Intel Corporation: Intel QuickData Technology Software Guide for Linux, http://www.intel.com/technology/quickdata/whitepapers/sw_guide_linux.pdf.

Vol.2011-OS-118 No.1
2011/7/27

10) Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *IEEE International Computer Software and Applications Conference*, pp.355–364 (2008).
11) Wickizer, S.B., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y. and Zhang, Z.: Corey: An Operating System for Many Cores, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp.43–57 (2008).
12) Colmenares, J. A., Bird, S., Cook, H., Pearce, P., Zhu, D., Shalf, J., Hofmeyr, S., Asanovi, K. and Kubiatowicz, J.: Resource Management in the Tessellation Manycore OS, *Second USENIX Conference on Hot Topics in Parallelism (HotPar '10)*, pp.1–6 (2010).
13) Schüpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T. and Isaccs, R.: Embracing diversity in the Barrelfish manycore operating system, *Workshop on Managed Multi-core Systems (MMCS 08)* (2008).
14) Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, New York, NY, USA, ACM, pp.29–44 (2009).