Vol.2013-ARC-205 No.15
Vol.2013-OS-125 No.15
2013/4/26

ヘテロジニアス計算機上の OS 機能委譲機構

佐伯裕治[†1]　清水正明[†1]　白沢智輝[†2]　中村 豪[†3]
高木将通[†4]　Balazs Gerofi[†5]　思 敏[†5]　石川 裕[†5†6]　堀 敦史[†6]

メニーコアプロセッサ向けの OS として，Linux カーネルと軽量カーネルが連携して管理するヘテロジニアス構成の OS を開発している．軽量カーネル上においても Linux カーネルのシステムコールを提供するために，軽量カーネルで実現されない Linux システムコールの処理は Linux カーネルに委譲する．引数がデータ領域を示すシステムコールの場合，転送が必要なデータの構造は API 仕様に依存するため， 300 種類以上の Linux 互換システムコールに個別に対応したデータ転送を実装する必要がある．本稿では，システムコール処理対象となるデータを同一仮想アドレスへのメモリマップを行う方式により，軽量カーネルに個々のシステムコール処理を実装することなく Linux カーネルに委譲する機構と，その基本評価結果について報告する．

Delegation Mechanism of OS Function for System with Heterogeneous Kernel

YUJI SAEKI[†1]　MASAAKI SHIMIZU[†1]　TOMOKI SHIRASAWA[†2]
GOU NAKAMURA[†3]　MASAMICHI TAKAGI[†4]　BALAZS GEROFI[†5]
MIN SI[†5]　YUTAKA ISHIKAWA[†5†6]　ATSUSHI HORI[†6]

We have been developing a heterogeneous OS composed of Linux and lightweight kernels for manycore processor. In order to provide all Linux system calls in the lightweight kernel, those primitives which are not provided by the lightweight kernel are delegated to the Linux kernel. Each system call differs in the number of arguments and argument types, and thus the code transferring arguments and results is implemented in each delegating system call. It is impractical to implement all Linux APIs, i.e., more than 300 system calls. Therefore, we developed a delegation mechanism of system calls without individual implementation to pass the data between the lightweight kernel and Linux using a memory mapping technique. In this technique, a user-level virtual address space in the lightweight kernel is mapped to the same position in a Linux process. We report the result of basic evaluation of system calls on lightweight kernel developed on Intel® Xeon Phi™ Coprocessor.

1. はじめに

電力性能比を向上するためのプロセッサアーキテクチャとして，単純な構造のプロセッサコアを多数並べて，これらを比較的低い周波数で動作させるメニーコアプロセッサの適用が有力視されている．インテル® Xeon Phi™ コプロセッサ[1]はその一例であり，512 ビット幅の SIMD 演算機構を付加したプロセッサコアを 50 コア以上動作させることにより，チップあたりのピーク性能は 1TFlops を超える．

Xeon Phi™は PCI Express 経由でホストシステムに接続されるコプロセッサでありながら，ホストとは独立したメモリにブートされる Linux カーネルが動作する．また，ホストからオフロードされたアプリケーションの一部を処理する実行モデルの他に，Xeon Phi™単独でアプリケーションを実行することが可能である．

†1 (株)日立製作所
Hitachi Ltd.
†2 (株)日立ソリューションズ東日本
Hitachi Solutions East Japan Ltd.
†3 (株)日立ソリューションズ
Hitachi Solutions Ltd.
†4 日本電気株式会社
NEC Corporation
†5 東京大学
University of Tokyo
†6 理化学研究所　計算科学研究機構
RIKEN AICS

このようなメニーコアプロセッサシステムの管理を行う OS として，単体の Linux カーネルが最適であるとは限らない[2][3][4][5]．例えば，一般に並列度が高くなると複数の処理による資源競合が増加するため，同期などにより処理が遂次化され効率が低下するという課題がある．

また，Xeon Phi™の主記憶は，コア間インターコネクトを経由してオフチップ RAM にアクセスする構造でありレイテンシが大きい．このため，コア毎に設けられる２次キャッシュはアプリケーション実効性能にとって貴重であり，OS 処理によるキャッシュライン置き換え等の外乱を抑制する必要がある．

東京大学と理化学研究所では，Xeon Phi™で動作する軽量カーネルを新たに開発し，軽量カーネル上で実行すべきシステムサービスと，ホストシステムの Linux カーネルに委譲すべきシステムサービスの機能分散を検討してきている[8][9][10]．本システムでは，個々のシステムコール毎に Linux 委譲機能を実装しており，実現されているシステムコールの種類は標準入出力とファイル I/O 機能などの基本機能となっている．

しかし，300 個以上存在する Linux の全てのシステムコールについてそれぞれの引数の性質を調べ，処理対象のデータを交換する処理を個別に実装するには，軽量カーネル

の開発コストがかかる．特に ioctl システムコールについては，引数のデータ構造が個々のデバイスドライバに対する操作要求に依存するため，完全な実装が困難である．また，システムコール API 仕様は Linux バージョンに依存するため，軽量カーネルの保守性に問題を生ずる．

そこで我々は，軽量カーネル側と同じ値の引数指定で Linux 側においてシステムコールを代行可能とするため，軽量カーネル上と同一の仮想アドレスを用いて Linux 側からシステムコールの処理対象データを参照可能とする方式を採用した．即ち，同一仮想アドレスへのメモリマップを行う方式により，軽量カーネルに個々のシステムコール処理を実装することなく Linux カーネルに委譲する機構について試作を行い，その基本評価を行った．

本稿では，2 章において Xeon Phi™上で Linux と複数の軽量カーネルが連携動作する OS 構成でのシステムコール委譲処理の概要について述べる．3 章では本稿で提案する同一仮想アドレスへのメモリマップによる委譲機構について述べ，4 章でファイル I/O スループットについて方式による比較結果を示す．5 章で関連研究について述べ，6 章で本稿のまとめを行う．

2. メニーコア向けヘテロジニアス OS

2.1 Linux と軽量カーネルの組合せ

Xeon Phi™が PCI Express 経由でホストに接続される構成のシステムにおける，Linux カーネルと軽量カーネルを組み合わせた OS 構成の例を図 1 に示す．以下，我々が開発を行っている軽量カーネルを McKernel と呼ぶものとする．また，Xeon Phi™上には McKernel だけが動作しており，ホストの Linux に対してシステムコール処理を委譲する OS 構成を Attached 構成と呼ぶ．他方，OS 処理を行う Linux 専用のコア群とアプリケーション実行を行う McKernel が管理するコア群に Xeon Phi™を資源分割する OS 構成を Built-in 構成と呼ぶ．

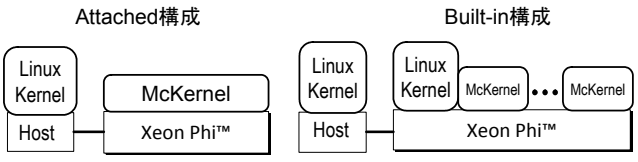

図 1 ヘテロジニアス 0S の構成

Attached 構成は，PCI Express 経由でホストに接続される現在の Xeon Phi™システム向けである．一方，McKernel におけるシステムコールが，同一の Xeon Phi™上の Linux カーネルに処理を委譲される Built-in 構成は，メインプロセッサがメニーコア型のシステムを想定した OS 構成である．

また，Built-in 構成では複数の McKernel が動作するように Xeon Phi™を資源分割することができる．この OS 構成は，コア数増大に伴いメモリ管理等のスケーラビリティが低下する場合や，メニーコアを分割使用するアプリケーションへの適用が期待できる．

Built-in 構成は，資源分割のために小規模な変更を行った Linux カーネルである SHIMOS[6][7][8]を，Xeon Phi™に適用することにより実現している．即ち，Attached 構成の Xeon Phi™システムを SHIMOS により資源分割したマルチコアシステム上でエミュレート可能とする環境[8]を，Xeon Phi™上で動作する MPSS Linux[11]に移植している．

SHIMOS では，予め設定したコア群とメモリ領域を各パーティションに割り当て，それぞれのカーネルは割り当てられた資源の上で動作する．Xeon Phi™上で複数の McKernel が Linux と連携する OS 構成は，複数の McKernel が動作するパーティションを作成し，ELF(Executable and Loadable Format)形式のカーネルイメージを異なる物理アドレスにブートすることにより実現している．ブート時は，はじめに 1 つのパーティションにおいて SHIMOS が起動され，ここから順次 McKernel をブートしていく．

2.2 ヘテロ OS のカーネル間インターフェース

Linux と McKernel のように，異種のカーネルが連携して資源管理を行うシステムにおいて，ブートや通信のようなカーネル間の基本的な処理に関する共通インターフェースを規定しておくことにより，軽量カーネルの開発が容易になる[8]．その結果，Xeon Phi™以外のメニーコアプロセッサへの移植性が向上するとともに，開発された異種のカーネルを同一のメニーコアプロセッサ上で連携動作させることができる．

論文[8]において AAL(Accelerator Abstraction Layer)として定義されているカーネルインターフェースをもとに，我々は，McKernel と Linux カーネルの間で行われるブート及び通信に関連するインターフェースを抽出し，IHK (Interface for Heterogeneous Kernel)として再定義し実装している．図 2 に示すとおり，IHK は，Linux カーネルモジュールとして組み込まれる IHK-Linux driver，McKernel が使用するハードウェア依存部実装である IHK-cokernel，及び，IHK-Linux driver と IHK-cokernel 上に実装されているカーネル間通信モジュール IHK-IKC の 3 つに分けて実装されている．

利用者が McKernel 上でアプリケーションを実行する場合，Linux 上で以下のコマンドを実行する．

```
$ mcexec <実行ファイル> <引数>
```

mcexec は Linux カーネルに実装した mcctrl カーネルモジュール機能を使い McKernel 上にアプリケーションプロセスを生成する．プロセス生成後，mcexec は次節で述べるシステムコール委譲処理を行う．

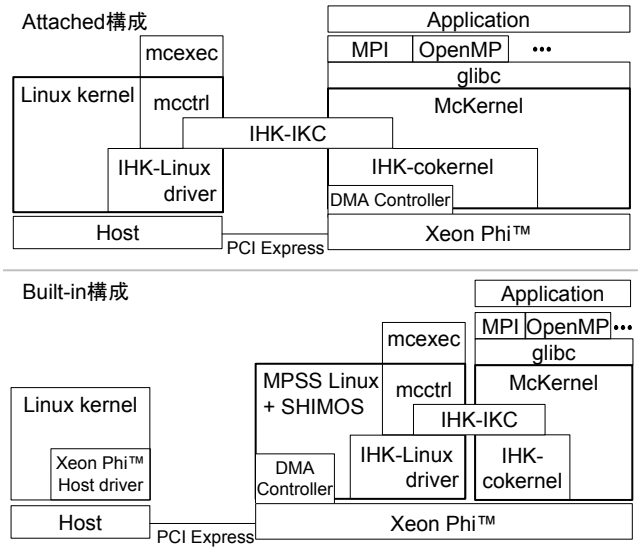

図 2 システムソフトウェアスタック

2.3 システムコール委譲処理の概要

McKernel は，アプリケーション実行のために必要最小限の機能しか持たず，大部分のシステムコールを Linux に委譲している[8][12]．McKernel 上のアプリケーションが Linux に委譲するシステムコールを発行すると，McKernel はカーネル間通信により，システムコール番号を含む引数及び処理に必要なデータを Linux に渡す．mcexec は mcctrl カーネルモジュールを呼び出して McKernel からのシステムコール委譲要求を待っており，mcctrl は McKernel から送られてきた引数とデータを mcexec に渡す．mcexec はシステムコール番号に対応する引数を再構築して，Linux のシステムコールを呼び出す．read や write システムコールのように引数で渡されたサイズに依存したメモリ領域アクセスがある場合，それぞれのシステムコール処理において引数を解釈し McKernel と Linux の間でデータ交換するコードを作成しなければならない．

東京大学で開発された版[8]では，標準入出力，ファイル I/O などの基本的なシステムコールのみを実装していた．McKernel 上で実現されない Linux 互換システムコール全てを Linux に委譲するために，それら委譲するシステムコールごとに引数の性質を調べてデータ移動のためのコードを実装し保守していこうとすると，そのコストは非常に高くなる．

3. メモリマップによる委譲機構

Linux におけるシステムコールは，使用頻度の低いものを含めると 300 個以上ある．特に ioctl システムコールについては，引数のデータ構造が個々の入出力デバイスに対する操作要求に依存する．McKernel におけるシステムコールを Linux に委譲するにあたって転送が必要なデータの構造は，これらの API 仕様によって異なり，Linux カーネルのバージョンにも依存する．従って，2.3 節で述べたシステムコール個別の実装を行うことなく，Linux 側で処理できる方式が望ましい．

そこで，システムコール処理対象のデータを McKernel 上で実行されるアプリケーションと同じ仮想アドレスを用いて，Linux 側からアクセス可能にする．即ち，当該データが格納された McKernel におけるページを，同一仮想アドレスで Linux 側の mcexec の仮想空間にメモリマップする．その結果，引数がポインタの場合にも，McKernel におけるのと同じ引数値で，mcexec がシステムコールを実行できるようになり，システムコール個別の実装が不要となる．

図 3 をもとに，以下本方式による委譲処理の概要を示す．

1) mcexec は ioctl システムコールを呼び出し，mcctrl 内で McKernel からのシステムコール実行要求を待つ
2) アプリケーションがシステムコールを実行した場合，McKernel はシステムコール番号及び引数を

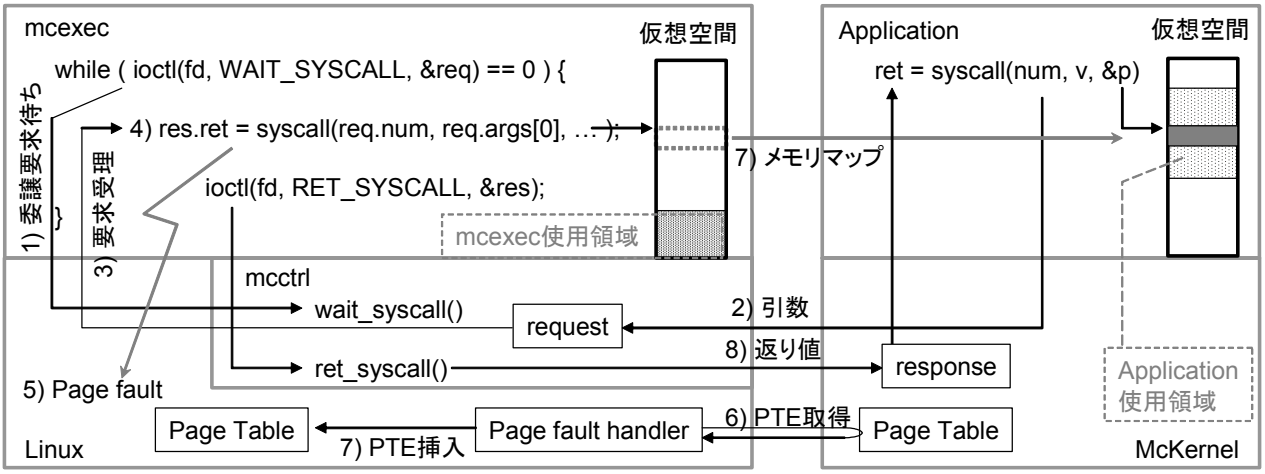

図3 メモリマップによるシステムコール委譲処理

要求ページに格納し，システムコール実行要求を Linux に送信

3) mcctrl 内でシステムコール実行要求を受理し，mcexec にシステムコール番号及び引数を返す
4) mcexec は Linux のシステムコールとして実行
5) 引数がポインタのとき Linux 側にはポインタが指すページがないため，ページフォルトが発生
6) ページフォルトハンドラが McKernel から当該ページのページテーブルエントリ（PTE）を取得
7) 当該ページを同一の仮想アドレスで mcexec の仮想空間にメモリマップ
8) mcexec はシステムコールの返り値を McKernel に送信するために ioctl システムコールを発行し，mcctrl 内で結果が McKernel に送信される

本方式では，システムコール処理対象のデータのメモリコピーを行わないため，システムコールが処理対象とするデータに再利用性があり，5,6,7)におけるページフォルトの処理時間を削減できる場合には，2.3 節で述べたオリジナルの実装方式に対し性能向上が期待できる．

5,6,7)においてページフォルトを発生させて，McKernel 上で実行されるアプリケーションのページを同一の仮想アドレスでマップするために，Linux 上の mcexec スレッドがアプリケーションと重複する仮想空間を使用しないようにする必要がある．これを実現するために，Linux では，mcexec コマンドを Position Independent Executable にしておくと，mcexec スレッドが使用するメモリ領域が仮想アドレスの大きい側にまとまって割り当てられることを利用する．そして，mcexec が使用しない仮想アドレスの小さい側の領域を予約しておき，アプリケーションが使用できるメモリ領域として McKernel に通知することによって重複を回避している．

4. 委譲機構の基本評価

4.1 システムコール性能評価方法

同一仮想アドレスへのメモリマップによって McKernel から Linux へデータを受け渡すシステムコール委譲処理の試作を行い，動作検証と基本性能評価を行った．

性能測定においては，図 4 に示すプログラムを McKernel 上で実行し，キャッシュをクリアした後．

- L1 データキャッシュミス回数
- L1 命令キャッシュミス回数
- L2 キャッシュミス回数
- 実行命令数
- CPU サイクル数

を性能カウンタで測定している．

```
clear_d_cache();                /* Data cache clear */
clear_i_cache();                /* Instruction cache clear */
start_measure();                /* Start of measurement */

pid = syscall(SYS_getpid);      /* System call (example) */

end_measure();                  /* End of measurement */
get_measure_results(values);    /* read the result */
```

図 4 システムコール性能測定プログラム

McKernel における read システムコールを，委譲先の Linux 下の RAM ディスクのファイルに対して行う場合について，CPU サイクル数とデータ長からスループット性能を算出した．McKernel と Linux との間でのデータ受け渡しを同一仮想アドレスへのメモリマップで行う方式（以降の図でメモリマップと表示），2.3 節で述べた処理対象データの受け渡しにメモリコピーを伴うオリジナルの実装方式（以降の図でオリジナルと表示），及び，委譲を行わず Linux カーネル上で測定プログラムを実行した場合（以降の図でホスト Linux 上での計測を Linux local，Xeon Phi™上での計測を Linux on Xeon Phi™と表示）について比較を行った．

4.2 ホスト Linux への委譲処理スループット評価

Xeon Phi™上の McKernel が PCI Express 経由でホストの Linux カーネルにシステムコールを委譲する Attached 構成において，データ長を変えて測定した read スループット性能を図 5 に示す．

Linux local と比較すると McKernel から Linux への委譲処理は，PCI Express 経由でホストから Xeon Phi™にデータ転送を行うことによる性能低下がある．現在の実装では Xeon Phi™搭載の DMA コントローラを使用せず CPU によりデータをコピーしている．McKernel における read の場合，ホストが Xeon Phi™にデータを書き込む際にバッファリングして 64 バイトにまとめて書き込む Write-combining メモリの効果で，PCI Express 性能は極端に低くならないが，DMA コントローラを使用することによりデータ長の大きい転送時の性能改善が見込まれる．

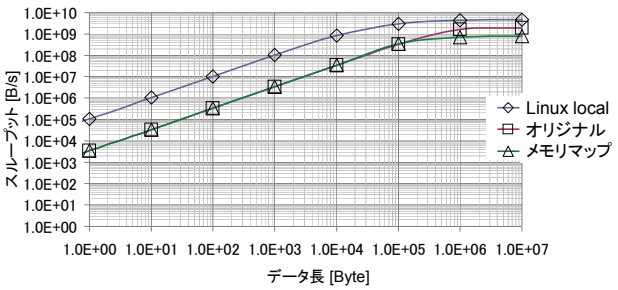

図 5 Attached 構成における read 性能

McKernel と Linux との間でのデータ受け渡しを同一仮想アドレスへのメモリマップで行うと，オリジナルの実装方

式に対して, データ長が大きい場合のスループットは約 2.5 倍低下している．これは 3 章で述べたように，Linux 側で read データを格納する McKernel 上のページにアクセスするたびにページフォルトが発生し，McKernel が管理するページをメモリマップするオーバーヘッドが影響しているものと考えられる．

そこでページフォルトの影響を調べるため，図 4 のプログラムにおいて，あらかじめシステムコールを実行した後にキャッシュをクリアし，システムコール実行時間を測定した．この場合，1 回目の実行時にメモリマップされたページを再利用して，ページフォルトを発生させずにメモリマップされたデータ領域に対する read システムコールの測定を行うことができる．図 6 に示すように，Linux 側でのメモリコピーを伴うオリジナル方式に対して，メモリマップの方式ではスループットが約 30%向上する結果となる．従って，データの再利用性がある場合には，2 回目以降の実行におけるページフォルトの発生頻度減少によって性能向上を見込むことができる．

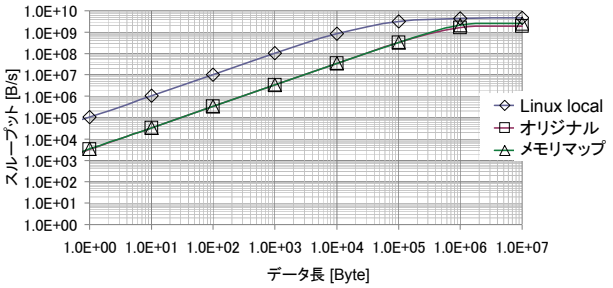

図 6 ページ再利用時の read 性能（Attached 構成）

4.3 Xeon Phi™上での委譲処理スループット評価

次に，Xeon Phi™を Linux のパーティションと McKernel のパーティションに資源分割する Built-in 構成における，read システムコールのスループット性能測定を行った．

図 7 に, Xeon Phi™で動作する Linux における read システムコールのスループット，及び，委譲の際に Xeon Phi™のメモリ上で行われる McKernel と Linux の間での read データ受け渡し方式の違いによるスループットの比較を示す．

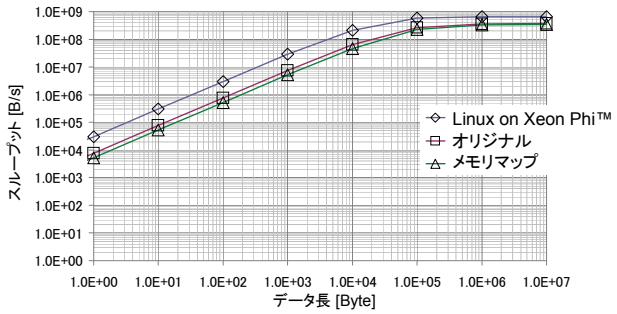

図 7 Built-in 構成における read 性能

McKernel から Linux への委譲処理は Linux 上で処理したときに比べて約 1/2 の性能となっている．McKernel からのデータ受け渡しを同一仮想アドレスへのメモリマップで行う方式は, オリジナルの方式に対して約 90%の性能である．

メモリマップ方式におけるページフォルトの影響を調べるため，Attached 構成における評価と同様の評価を行った．図 8 に示すように，大きいデータ長におけるメモリマップ方式の委譲スループット性能は, 委譲を伴わない Xeon Phi™で動作する Linux において実行した場合と差がなくなることがわかる．

従って，Built-in 構成では，ページマップ方式でのスループット性能低下はページフォルトの影響であり，データの再利用性がある場合には，ページフォルトの発生頻度が減少し，Xeon Phi™で動作する Linux におけるシステムコールと同等のスループットが得られる．

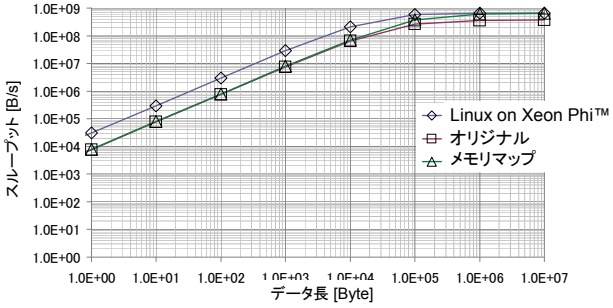

図 8 ページ再利用時の read 性能（Built-in 構成）

4.4 システムコールによる Cache pollution

Xeon Phi™で動作しているアプリケーションがシステムコールを実行したとき，OS 処理によって当該プロセッサコアに付随するキャッシュメモリの内容が置き換えられると，アプリケーションが実行再開したときにキャッシュミスの原因となる．Attached 構成及びメニーコアエミュレーション環境において，オリジナルの実装方式で Linux にシステムコール処理を委譲する McKernel について，キャッシュメモリへの影響を抑制できることが判明している[8].

そこで Built-in 構成において，図 4 の測定プログラムにおいてデータ長を変えて read システムコールを実行したときの, L1 データキャッシュミス, L1 命令キャッシュミス, L2 キャッシュミス回数の測定を行った. 測定開始前にキャッシュクリアを行っているため，測定されたキャッシュミス回数は，置き換えが起こりうるキャッシュライン数に等しい．図 9 に，McKernel における read をメモリマップでデータを受け渡す方式で Xeon Phi™上の Linux カーネルに委譲する場合（図中では mmap）と， Xeon Phi™上の Linux カーネルにおいて read する委譲を伴わない場合（図中では Linux）を比較した結果を示す．

Vol.2013-ARC-205 No.15
Vol.2013-OS-125 No.15
2013/4/26

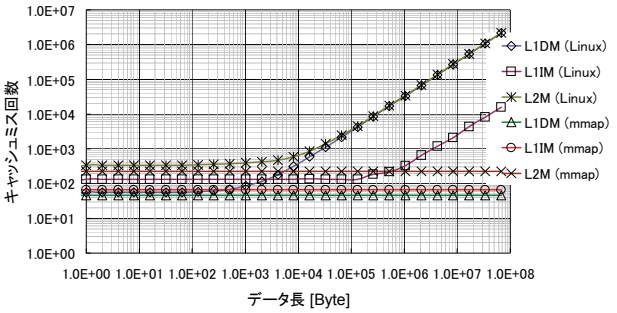

図 9 read システムコールによるキャッシュへの影響

このように，システムコール個別実装が不要なメモリマップ方式により委譲を行う場合についても，キャッシュラインの置き換えはデータ長によらず一定であり，OS 処理がアプリケーション実行に与える影響を抑制できる．

また，Linux に委譲して処理される場合，システムコールの種類によらず, McKernel の処理によって置き換わるキャッシュライン数はほぼ一定となる．表 1 に，read（データ長：10MB）と，間接参照を行う ioctl（引数がポイントする構造体メンバーがポイントするデータにアクセスする）システムコールについて，キャッシュミス回数の測定結果を示す．ここで，Linux との間のデータ受け渡しを同一仮想アドレスへのメモリマップで行っているため，ioctl 個別の処理に対応した実装をすることなく実行できる．但し, カーネル間通信が PCI Express を経由するかに依存して，Attached 構成と Built-in 構成の差が生じている．

表 1 McKernel での read, ioctl によるキャッシュミス回数

	Attached 構成		Built-in 構成	
ミス回数	read	ioctl	read	ioctl
L1D	44	45	49	46
L1I	72	71	66	65
L2	233	232	224	223

5. 関連研究

本稿の対象である軽量カーネルを用いたマルチコア／メニーコアプロセッサにおける OS 処理のスケーラビリティに関連した研究を紹介する．

Multi-kernel[2]は，軽量カーネル群が非同期なメッセージ通信によって連携し，マルチコアシステムを分散システムとして管理する OS 構成である．軽量カーネルが管理するデータを複製することによって，キャッシュメモリにおける競合をなくしスケーラブルな資源管理を行う．また，ハードウェアに依存する部分を CPU driver として分離して実装することで，異種のプロセッサコアへの移植性を考慮している．

AsyMOS[3]は，マルチコアシステムにおいて，入出力デバイスに対応してプロセッサコアを割り当て，当該デバイスが必要とする機能に限定した軽量カーネルに入出力処理をオフロードする．シングルタスクの軽量カーネルの採用によって，当該デバイスに関する競合と同期のオーバーヘッドを削減する．

fos[4]は，マルチコアシステムにおいて，異なる OS 機能を異なるプロセッサコア群に割り当て，OS 機能を並列分散サービスとして提供する．それぞれの OS 機能を提供するサーバは，メッセージ通信によって連携し，管理するデータを複製することにより，スケーラブルな OS サービスの提供を行う．

FusedOS[5]は，IBM BlueGene/Q における計算ノードを，Linux コア群とアプリケーションコア群に資源分割して，ユーザレベルで動作するライブラリがシステムコール処理を Linux カーネルに委譲する．この OS 構成におけるシステムコールが，計算ノードを資源分割せず I/O ノードに委譲する場合（Compute Node Kernel 上でのシステムコール）と，同等の性能が得られることが報告されている．

またシステムコールの委譲に関連して，FlexSC[12]ではシステムコール処理による資源使用量の評価を行い，ユーザモードのスレッドに引数を渡し例外を伴わずにシステムコールを実行することによって，アプリケーション実行に与える影響を抑制している．

6. おわりに

Xeon Phi™上で Linux と複数の軽量カーネルが連携して動作する OS 構成において, 300 個以上存在する Linux 互換のシステムコールについて個別の実装を行うことなく，軽量カーネルから Linux に委譲する機構の開発を行った．本方式では，委譲先である Linux が，ページフォルト発生時に軽量カーネル上のアプリケーションと同一仮想アドレスへのメモリマップを行う．これにより軽量カーネル側と同じ値の引数指定で，Linux 側においてシステムコールが実行可能となる．

本方式の基本評価としてファイル I/O スループットを測定し，システムコール処理対象データの受け渡しをメモリコピーで行う従来方式と比較すると，ページフォルトのオーバーヘッドが原因で性能低下していることがわかる．このことから，データの再利用性がある場合には 2 回目以降の実行においてページフォルト回数が減少し，従来方式以上の性能が達成できる見通しが得られた．結果として，Xeon Phi™の限られた容量のキャッシュメモリの OS 処理による外乱を低減できるため，軽量カーネル上で動作するアプリケーションの実行効率を向上できる．

今後は，同一仮想アドレスへのメモリマップ方式として，アプリケーションの使用したページを予め Linux 側でもマップすることでページフォルトを起こさない方式との比較

を行う．また，使用頻度が高く高性能が要求されるシステムコール，あるいは，McKernel で処理するのが望ましいシステムコールについては，個別の実装を行う必要がある．

謝辞 本研究の一部は，文部科学省「将来のHPCIシステムのあり方の調査研究」のなかの課題名「レイテンシコアの高度化・高効率化による将来のHPCI システムに関する調査研究」からの支援を受けている．

参考文献

1) インテル® Xeon Phi™ コプロセッサ, http://www.intel.co.jp/content/www/jp/ja/processors/xeon/xeon-phi-detail.html
2) Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania: The Multikernel: A new OS architecture for scalable multicore systems, *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles,* pages 29–44, New York, NY, USA, 2009. ACM.
3) Steve Muir and Jonathan Smith: AsyMOS - An Asymmetric Multiprocessor Operating System, *Proceedings of Open Architectures and Network Programming,* pages 25–34, 1998.
4) David Wentzlaff and Anant Agarwal: Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores, *SIGOPS Oper. Syst. Rev., Vol. 43,* pp. 76–85, April 2009.
5) Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, and Kyung Dong Ryu: A Hybrid Approach to Exascale Operating Systems, *SC12 Poster*.
6) Taku Shimosawa, Hiroya Matsuba, and Yutaka Ishikawa: Logical Partitioning without Architectural Supports, *32nd IEEE International Computer Software and Applications Conference (COMPSAC 2008),* pp. 355-364, 2008.
7) Taku Shimosawa and Yutaka Ishikawa: Inter-kernel Communication between Multiple Kernels on Multicore Machines, *IPSJ Transactions on Advanced Computing Systems Vol.2 No.4 (ACS 28),* pp. 64-82, December 2009.
8) Taku Shimosawa: Operating System Organization for Manycore Systems, *A Doctor Thesis submitted to the Graduate School of the University of Tokyo* (2011)
9) Min Si and Yutaka Ishikawa: Design of Direct Communication Facility for Manycore-based Accelerators, *CASS2012 in conjunction with IPDPS2012,* 2012.
10) Balazs Gerofi, Akio Shimada, Atsushi Hori, and Yutaka Ishikawa: Operating System Assisted Hierarchical Memory Management for Heterogeneous Architectures: Preliminary Results on Stencil Computation, *The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013)* (2013 (To appear)).
11) Intel® Manycore Platform Software Stack (MPSS), http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss
12) Livio Soares and Michael Stumm: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association