Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

メニーコア環境における キャッシュアウェア・オペレーティングシステムに向けて

下沢 拓[†1] 堀 敦史[†3] 石川 裕[†1,†2,†3]

我々は，汎用コアによるメニーコアアーキテクチャ上で，メニーコアの特性を生かした，キャッシュを考慮したオペレーティングシステムを設計している．このシステムは，メニーコア上でのキャッシュ汚染を軽減しながら，汎用 OS のカーネルの機能を提供することを目的とする．本稿では，その予備評価として，システムコールやライブラリのキャッシュに対する影響について測定を行った結果を示す．その結果は，このアプローチが効果的となるためには，特定のコア上で，同じシステムコールやライブラリ関数を実行するか，同じアプリケーションのスレッドのシステムコールなどを実行することが必要であることを示唆する．

Towards a Cache-aware Operating System in Many-core Architecture

Taku Shimosawa,[†1] Atsushi Hori[†3] and Yutaka Ishikawa[†1,†2,†3]

We have been designing a cache-aware operating system for many-core architecture, which consists of a number of generic cores. The operating system is aimed to reduce cache pollution in many-core architecture while it provides the full functionality of a general operating system via host multicore CPUs. As a feasibility study on this approach, we have conducted experiments for effects of the operating system calls and the library calls on cache. The results suggest that the particular system calls or library calls should be executed in a dedicated core, or those of the threads of the same application should be executed in a dedicated core for this approach to be effective.

1. はじめに

プロセッサの性能向上の手法としては，コア数を増加させ並列性を向上させることによるものが一般的になり，コモディティ・プロセッサにおいてもマルチコア CPU が一般化している[1),2)]．高性能計算を目的としたスーパーコンピュータでも，これらのコモディティCPU を搭載したクラスタシステムが多く使われており，2010 年 11 月の TOP500[3)] においても，上位 10 システムのうち 7 つが x86_64 アーキテクチャを採用している．さらに GPGPU(General-purpose computing on GPU) による性能向上を図り，GPU を搭載したものも増えつつあり，前述の TOP500 においても，上位 10 システムのうち 3 つが GPU をマルチコア CPU に加えて搭載した構成となっている[4),5)]．

他方，GPU は，浮動小数点の演算に特化し，整数演算命令や制御命令を持たないという欠点がある．そのため，GPU を用いたシステムにおいては，他ノードとの通信などの I/O 処理にはホストとなるプロセッサを用いる必要がある．スーパーコンピュータの全体的な性能を向上させるにあたり，GPU のような一定の演算に特化したコアに加えて，自身で I/O 等の制御を行うことのできる汎用的な CPU コアを搭載したメニーコアプロセッサを搭載することが考えられる．そのようなプロセッサはホストとしての CPU を必要とするかの違いがあるが，いくつかの開発または製造が行われている[6)–8)]．

これのうち，ホストを持つ形のメニーコアプロセッサについて考えると，このようなプロセッサを用いたシステムにおいては，ホストとなるプロセッサだけでなく，メニーコアプロセッサ自身での制御管理が可能となるため，それらを行うシステムソフトウェアが必要となる．現在，クラスタシステムにおいては，Linux のような汎用 OS が多く用いられているが[9),10)]，これらをメニーコアプロセッサが搭載されたシステムで使用するには，いくつかの課題が存在する．第一に，メニーコアプロセッサは，各コアあたりの整数演算性能及びメモリキャッシュ容量が小さく，ホストの汎用 OS と同一のカーネルを起動するには，負荷が高いことである．第二に，ccNUMA や SMP といった，キャッシュコヒーレントな共有メモリを想定したデータの共有が困難，もしくは非常にオーバーヘッドが高いという問題がある．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 東京大学情報基盤センター
Information Technology Center, The University of Tokyo
†3 理化学研究所計算科学研究機構
Advanced Institute for Computational Science, RIKEN

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

本稿では，特にオペレーティングシステムがキャッシュに与える影響について着目し，メニーコアプロセッサ上でのオペレーティングシステム設計を行うにあたり，既存のマルチコアプロセッサシステム上で行った予備評価とその考察を示す．

本稿の構成は，以下の通りである．次節において，対象環境とするシステムとそのシステム上における既存のオペレーティングシステムの問題点について述べる．第 3 節においては，我々が設計をしているメニーコア上のキャッシュ利用に着目したオペレーティングシステムの設計の概略について述べ，検討すべき課題について述べる．第 4 節で，既存のオペレーティングシステムのカーネルやライブラリのキャッシュに対する影響についての予備評価の環境とその方法について述べる．そして，第 5 節でその実験結果と考察について述べる．第 6 節で，スケーラブルなオペレーティングシステムについての関連研究について述べ，第 7 節でまとめる．

2. 背　　景

本節では，設計の対象とするアーキテクチャと，既存のオペレーティングシステムにおけるキャッシュに関するスケーラビリティの問題について述べる．

2.1 対象アーキテクチャ

我々が設計を行っているオペレーティングシステムが対象とするアーキテクチャは，**図 1** で示すように，ホストプロセッサとして，コモディティマルチコア CPU を搭載し，Intel Knights Ferry[7] といったメニーコアプロセッサを I/O バスで接続したヘテロ環境である．このようなメニーコアプロセッサの特徴は，(1) ホストプロセッサと比較して，クロック数が低くシングルスレッドでの演算性能が低い，(2) メモリキャッシュ容量の階層が少なく，また容量も小さい，(3) リングバスなどのレイテンシの高いメモリを採用している，といった特徴をもつ．また，ホストのメモリとメニーコアプロセッサのメモリは独立しており，これらの一貫性はハードウェアでは維持されない．メモリ間のアクセスには，PCI Express といった I/O バスを経由する必要がある．そのため，この二つの間のメモリアクセスは，レイテンシが大きく，また帯域幅もメモリのそれと比較すると小さい．

このようなアーキテクチャ上におけるオペレーティングシステムを実現するためには，第一に，いかにメニーコア上でのカーネル実行の処理時間を減らし，また，キャッシュミスを減らすかが課題となる．さらに，第二に，オペレーティングシステムがホストとの通信を行う場合において，いかに転送を集約し効率よく行うかという課題がある．

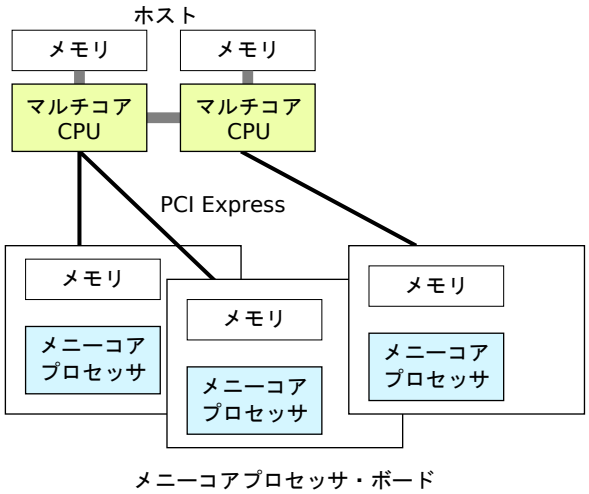

図 1 想定するマシン構成

2.2 汎用 OS の問題点

Linux のような汎用 OS を，メニーコアアーキテクチャに適用するには，前述の課題を解決しなければならないことに加えて，主にキャッシュ競合によるスケーラビリティの問題，キャッシュコヒーレントな共有メモリを仮定している問題が存在する．

スケーラビリティの問題は，主に複数のコアが同じカーネル内のデータ構造にアクセスすることによるキャッシュ競合や，スケジューリングの方法によって引き起こされる問題である．Linux において，少なくとも 48 コアまでは，いくつかの改良で改善することを示した研究[11] が存在するが，メニーコアプロセッサは，少なくとも 64 コアや 128 コアといった数のコア数が想定され，今後さらに増加してくことが予想されるため，既存の設計をそのまま用いるのは難しいと考えられる．

共有メモリを仮定している問題は，ソフトウェアによってカーネルデータ構造を配置したメモリの一貫性を保つ機構を加えるか，一貫性が保たれているメモリ上にカーネルデータを集中配置する必要があり，同様にオペレーティングシステムの設計を変更する必要がある．

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

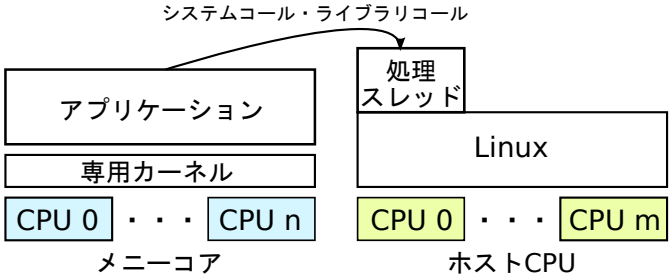

図 2 提案システムの概略図

3. オペレーティングシステムの設計方針と課題

3.1 設 計 方 針

前節で述べたようなシステム上におけるオペレーティングシステムとして，我々は，キャッシュの利用に着目したオペレーティングシステムの設計を行っている．その概略図を**図 2** として示す．

メニーコア上で処理を行うには重いシステムコールやライブラリ関数については，メッセージパッシングによりホストのマルチコア CPU 上で実行させる形をとることで，メニーコア上のアプリケーションに対するキャッシュミスの影響を軽減させる．また，ホスト CPU でも同様にキャッシュミスによる影響を減らすため，メニーコアの処理に当てる CPU コアは固定されるものとする．通信のオーバーヘッドのほうが大きいシステムコール等については，メニーコア自身で実行するものとする．

また，ホスト CPU 間との通信機構を実現するには，(i) 他の CPU に対する通知方式，(ii) データの転送方式を設計する必要がある．まず，通知方式については，できる限りキャッシュミスによる負荷を減らすために，カーネルへの遷移をなく行えるようにする．そのために，一方のメモリの特定のページを割り当て，もう一方でそれをユーザーページに割り当てて，そのページへのメモリ書き込みを以って，相手に対する通知を行う方式とする．データの転送方式については，共有メモリを想定する方式では，ホスト CPU からメニーコアプロセッサ側のメモリへの頻繁なアクセスが必要となる．このようなアクセスは，前節で述べたように I/O バスを経由し効率が悪いため，一度に集約しメモリ内容のコピーやキャッシュを行う機構を通信機構に備えるものとする．

表 1 評価環境

モデル	DELL PowerEdge R410
CPU	Intel Xeon X5550 (2.67GHz, Quad-core) x 2, (ただし 1 コアのみ使用)
Cache Size	L1I: 32KB, L1D: 32KB, L2: 256KB (各コアあたり) L3: 8MB
Cache Line Size	64 byte (全てのキャッシュ)
Memory	DDR3 1333MHz 2GB x 3 (Triple Channel) x 2
Network	On-board (Broadcom bnx2)

3.2 検 討 課 題

システムコールやライブラリ関数の実行については，その実行をホストとメニーコアのどちらでどのように割り当てて行うのか，さらに設計を進める必要がある．また，通信機構についても，通知方式の利害得失およびメモリアクセスのオーバヘッドとキャッシングのバランスについて検討が必要になる．

我々は，これらの設計，検討を進めるために，既存のマルチコアシステムを用いて行える範囲で予備評価を行った．次節以降では，この予備評価について述べる．

4. 予備評価の環境と手法

本節では，次節で結果を述べる既存のマルチコアシステムでの Linux のキャッシュに対する影響についての実験の環境と，パフォーマンスカウンタを使用するために用いた方法について述べる．

4.1 実 験 環 境

実験に使用した環境は，**表 1** に示す通りである．なお，予測不能な挙動や結果の不安定性を除くために，CPU の TurboBoost 機能および HyperThreading を無効化した．OS としては，kernel.org で配布されている Linux 2.6.36 を用い，ユーザーランドには，Ubuntu 10.04 (Lucid Lynx) x86_64 版を使用し，CPU は 1 コアのみを有効として測定を行った．また，OS から，cpufreq 等の CPU のクロック周波数を変動させる機能を無効化し，実験中において周波数が一定となるようにした．

また，実験で使用したアプリケーションは，Ubuntu でパッケージとして提供されているものを用い，パッケージのバージョン番号は，libc 2.11.1-0ubuntu7.2，MPICH2 1.2.1.1-4 である．測定ツールの比較として使用した PAPI のバージョンは，4.1.1 である．

測定には，CPU に用意されているパフォーマンスカウンタを用い，**図 3** に示すようなプログラムの実行により測定を行った．命令キャッシュのクリアとしては 64KB の nop 命令の

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

```
...
clear_i_cache(); /* 命令キャッシュのクリア．クリアしない場合は省略 */
clear_d_cache(); /* データキャッシュのクリア．クリアしない場合は省略 */
start_measure(); /* 測定開始 */

pid = syscall(SYS_getpid); /* 測定対象のシステムコール（例）*/

end_measure(); /* 測定終了 */

get_measure_results(values); /* 測定結果取得 */
...
```

図 3 測定プログラムの例

表 2 パフォーマンスカウンタ名と略号

略号は本稿の以降の節で用いるものである．

項目 (略号)	カウンタ名
L1D キャッシュリクエスト (L1DR)	L1D_ALL_REF.ANY
L1D キャッシュミス (L1DM)	L1D.REPL
L1I キャッシュミス (L1M)	L1I.MISSES
L2 キャッシュミス (L2M)	L2RQSTS.MISS
L3 キャッシュミス (L3M)	LLC Misses
実行命令数 (INS)	Instruction Retired
ストールサイクル数 (INACTIVE)	UOPS_EXECUTED.CORE_ACTIVE_CYCLES (CMask = 1, Invert = 1)
実行サイクル数 (CYCLE)	CYCLES.NOT_HALTED

実行，データキャッシュのクリアとしては 64MB のメモリ領域に対するシーケンシャルアクセスを行った．測定結果は，同じプログラムによって，1000 回繰り返し測定した結果を平均して求めたものである．測定に使用したパフォーマンスカウンタ名は，**表 2** に示す通りである．また，パフォーマンスカウンタの値から導出される値の名前と略号を，**表 3** に示す．これは，第 5 節の測定結果で用いられるものである．パフォーマンスカウンタは，実験に用いた CPU においては最大 4 つまでのパフォーマンス項目を同時に測定できるため，本実験では測定項目を 4 つずつに分けてそれぞれ測定を行った．なお，キャッシュリクエスト及びミス数はその回数で表され，実際のメモリへのアクセス量は，そのキャッシュラインサイズ (64byte) を掛けただけのバイト数となる．

表 3 測定結果で掲げるその他の項目と略号

項目（略号）	算出式
ストール率 (STALL)	INACTIVE/CYCLE
CPI	CYCLE/INS

4.2 パフォーマンスカウンタの使用方法

パフォーマンスカウンタに関する操作は，特権モードでの処理が必要なため OS カーネルのサポートが必要になる．Linux のユーザーアプリケーション上からパフォーマンスカウンタを使用するには，PAPI[12] を使用する方法，Linux カーネルに 2.6.31 から追加された perf_event_open システムコールを使用する方法などがある．

これらは，測定中にコンテキストスイッチによりプロセスが切り替わった場合に，それを検出し，パフォーマンスカウンタを別のプロセスの実行中では一時停止することにより，特定のプロセスのみ活動を測定することができるというメリットがある．しかし，測定の開始・終了の際に，システムコールを使用する必要がある．これは，測定は，**図 4** に示される処理 1,2 の点の間の区間のものとなり，特にカーネル内の短い期間，たとえば一回のシステムコールを測定する場合には，測定結果として開始・終了自体のオーバーヘッドを含んでしまうという問題を引き起こす．そのような短い期間では，busy なプロセスがいない環境であれば，他のプロセスがスケジュールされないことを前提とできるので，コンテキストスイッチによるプロセス切り替えの検出は不要である．

そこで，我々は，ユーザーレベル性能測定ツールを作成した．これは，測定の開始・終了点でユーザーレベルからカウンタの値を読むものである．実装としては，CPU の制御レジスタ CR4 のフラグを操作することで，ユーザーレベルからのパフォーマンスカウンタの読み出しを許可させるカーネルモジュールを組み込むこととした．パフォーマンスカウンタの制御レジスタの取扱い自体にはシステムコールを用いるが，測定の開始・終了と分離しており，測定時にカーネルへの遷移を不要とすることができる．これにより，図 4 の処理 3,4 の点に示す箇所で値の測定が行えるようになり，測定区間以外の不必要な実行が計測結果に含まれることを避けることができる．

このことを確かめるため，測定開始の直後に何もせず測定終了するプログラムを作り，実験を行った．**表 4** はその結果であり，ユーザーレベル性能測定ツールによるもの，カーネルレベルツールとして perf_event_open システムコールのみを用いたもの，PAPI(内部に perf_event_open システムコールを用いる設定) を用いた場合の 3 つの比較を示す．概ね，

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

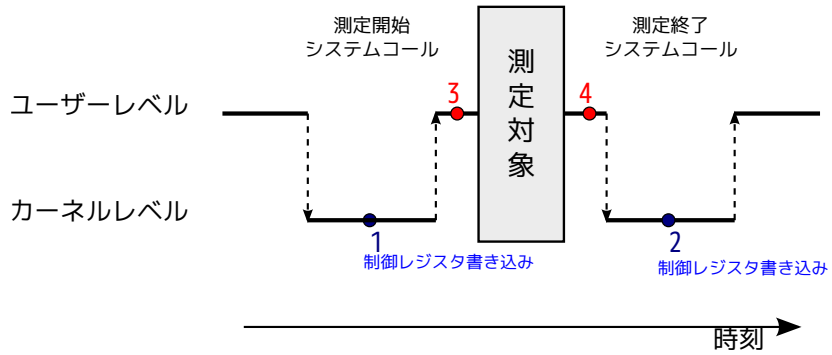

図 4 パフォーマンスカウンタの制御と測定タイミング

表 4 パフォーマンスカウンタの測定方法による値の違い

ユーザーレベル

	L1DR	L1DM	L2M	L3M	STALL	CYCLE	INS
ユーザーレベル性能測定ツール	5	1	5	4	168	305	30
カーネルレベルツールのみ	8	0	4	3	0	329	26
PAPI	755	32	64	65	3817	5741	1469

カーネル

	L1DR	L1M	L2M	L3M	STALL	CYCLE	INS
ユーザーレベル性能測定ツール	0	0	0	0	0	0	0
カーネルレベルツールのみ	326	1	39	23	2968	5141	1114
PAPI	320	12	53	70	5384	7086	1463

ユーザーレベル性能測定ツールでは，他と比較して小さい値となっている．ただし，ユーザーレベルでのキャッシュミス数が 1 回増加している．これは，ユーザーレベルではカウンタを停止させることができないため，カウンタ群からひとつずつ読みだしてメモリ上に順次格納する必要があるためである．そのため，少なくともメモリへのアクセスが一回増えることになる．その一方で，カーネルレベルによるものでは，パフォーマンスカウンタからの値の読み込みの際にカウンタを停止してから読み出せるので，メモリアクセスは発生しない．

5. 評価結果

本節では，前節で述べた方法を用いて，キャッシュに対する影響についての実験を行った結果を示す．

行った実験の概要は以下の通りである．まず，システムコールやライブラリコールによりキャッシュがどれだけ汚染されるかを計測するために，それぞれシステムコール，libc のライブラリ関数による影響，MPI ライブラリの関数自体のキャッシュミス等の値を測定した．

5.1 システムコール

まず，いくつかのシステムコールについて，キャッシュをクリアした場合と，その測定直後に再度システムコールを実行した場合，すなわち，キャッシュ上にすべてキャッシュ容量の許す限り残っていることが期待される場合の二つについて，測定を行った．使用したシステムコールは，最もカーネル内での処理が少ないと考えられる `getpid` システムコール，`getimeofday` システムコール，`open`，`write`，`close` システムコールである．最後の 3 つのシステムコールは，実際のデバイスへの操作による影響を排除するために，RAM ディスクに対して 1 バイトのデータを書き込むことにより測定を行った．これらの結果を**表 5** に示す．

この結果より，getpid システムコールはその処理の内容からカーネルへの遷移および復帰のコストとほぼみなすことができる．これよりカーネルモードへの遷移と復帰で必要となるメモリ量は L2M で 12 回程度と 1KB に満たない値であることがわかる．また，getpid に加えて，gettimeofday といった小さなシステムコールでは，連続してシステムコールを発行する場合には当然キャッシュ内に残るので，新たなミスはほとんど発生していない．このようなシステムコールは，メニーコア内で処理を行ってもよいものと見ることができる．

他方，open や write といった，VFS に関するシステムコールは，1 回の呼び出しについて 20KB 程度のメモリへのアクセスが発生している．これは，測定環境では L1 キャッシュの 1/6 程度にあたり，メニーコア環境ではさらに占める割合が高くなる可能性もあり，実際にデバイス上のファイルへの操作を含む場合は，さらにメモリが必要となると考えられる．一方，キャッシュがクリアされていない場合，write では実行サイクル数が 6.4%程度，CPI が 8.4%程度まで減少している．

5.2 libc ライブラリ関数

libc のライブラリ関数について，同様にキャッシュをクリアした場合と，残っている場合の両方について測定を行った．対象としたライブラリ関数は，ファイル I/O をのぞき，使用頻度の高く比較的複雑と考えられる `fprintf`，`sprintf` の各関数である．なお，`fprintf` はコンソールへの出力を必ず含めるために，標準エラーへの出力とした．また，使用したフォーマットは，1 つの浮動小数点数 (`"%f"`) である．そのユーザーレベル内実行時の値とカーネル内実行時の値を**表 6** に示す．

結果は，システムコールよりもキャッシュミスが大きくなり，L2M で sprintf の場合で

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

表 5 システムコールに伴うパフォーマンスカウンタの値（カーネル実行中の発生数）

キャッシュクリア後

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
getpid	63	6	9	12	13	0.62	22.33	1496
gettimeofday	111	8	25	27	25	0.74	17.05	2710
stat	904	64	183	231	179	0.70	11.26	18215
open	1184	85	240	336	283	0.69	13.03	29584
write	1235	98	301	405	353	0.72	17.02	36018
close	348	21	106	87	77	0.86	13.50	8588

キャッシュクリアなし

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
getpid	38	0	0	0	0	0.25	1.29	87
gettimeofday	74	1	0	0	0	0.12	1.02	162
stat	776	3	26	1	1	0.21	0.86	1383
open	1000	7	107	15	6	0.33	0.97	2200
write	1011	11	240	6	6	0.29	1.10	2315
close	332	1	51	1	1	0.31	1.21	773

表 6 libc 関数呼び出しに伴うパフォーマンスカウンタの値

キャッシュクリア後

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
fprintf (user)	1650	93	325	401	263	0.80	7.14	22512
fprintf (kernel)	1294	75	311	338	255	0.88	12.12	28118
sprintf (user)	1490	72	266	355	215	0.76	5.77	17177
sprintf (kernel)	0	0	0	0	0	-	-	0

キャッシュクリアなし

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
fprintf (user)	1243	5	195	8	8	0.37	1.14	3595
fprintf (kernel)	958	6	148	8	5	0.20	1.18	2476
sprintf (user)	1149	1	85	10	4	0.26	0.91	2712
sprintf (kernel)	0	0	0	0	0	-	-	0

22.7KB，fprintf でカーネルとユーザーモードあわせて 47.3KB に及んでいる．他方，キャッシュのクリアをしない場合には，キャッシュミスの数は極めて小さく，実行サイクル数も 16%程度にまで減少し，CPI も 10%から 16%程度に改善している．

5.3 MPI ライブラリ関数

MPI ライブラリの 1 対 1 通信関数についても前節と同様の測定を行った．MPI ライブラリには，4.1 節で掲げた MPICH2 を用い，通信の相手先は，2.4GHz Opteron 8378 を装備

表 7 MPI 関数呼び出しに伴うパフォーマンスカウンタの値

キャッシュクリア後

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
MPI_Send (user)	818	115	147	272	229	0.59	17.97	19391
MPI_Send (kernel)	557	44	80	142	129	0.91	10.82	12720
MPI_Recv (user)	832	123	204	346	266	0.83	17.01	20996
MPI_Recv (kernel)	582	49	74	138	126	0.90	10.84	12875
MPI_Isend (user)	687	79	131	236	176	0.80	16.99	12370
MPI_Isend (kernel)	0	0	0	0	0	-	-	0
MPI_Irecv (user)	687	107	123	242	202	0.80	15.57	15167
MPI_Irecv (kernel)	295	37	77	127	115	0.94	18.44	10852

キャッシュクリアなし

	L1DR	L1DM	L1IM	L2M	L3M	STALL	CPI	CYCLE
MPI_Send (user)	675	72	299	137	53	0.66	6.83	8195
MPI_Send (kernel)	532	32	113	22	5	0.52	2.00	2348
MPI_Recv (user)	654	80	324	79	16	0.69	4.45	5491
MPI_Recv (kernel)	525	21	41	5	3	0.42	1.52	1792
MPI_Isend (user)	397	15	85	17	8	0.57	1.95	1379
MPI_Isend (kernel)	0	0	0	0	0	-	-	0
MPI_Irecv (user)	547	57	207	50	20	0.71	4.61	4495
MPI_Irecv (kernel)	271	24	100	7	2	0.47	1.98	1160

したマシンであり，1Gbps Ethernet で接続されたものである．それぞれ転送するデータサイズは，1 バイトである．その結果を**表 7** に示す．なお，それぞれ，通信を測定前に行うことで，コネクションを張る等の初期化処理を含まないようにし，受信についてはバリアを用いて確実に配送されているデータについての受信処理としている．

MPI 関数の場合も，libc と同様にかなり多くのキャッシュミスを引き起こしており，キャッシュクリアをしない場合では，関数により大きく異なるものの実行サイクル数が 11%から 42%に，CPI が 10%から 38%に減少している．

5.4 考　　察

これらの結果から，キャッシュクリア後とクリアしない場合を比較すると，当然ながら後者のほうが相当に低いサイクル数で実行ができていることがわかる．したがって，このようなキャッシュミスの多いシステムコールやライブラリ関数は，特定の CPU コアが繰り返し処理したほうが効率がよいと考えられる．これは，違う対象に対する同じシステムコール，例えば write システムコールで異なるファイルディスクリプタに対する操作の場合でも有効と考えられる．なぜならば，システムコールでの L1 ミスの値を比較すると，命令キャッシュのミスのほうがデータキャッシュのミスより大きいからである．ただし，この方法を実

現するには，(i) 関数もしくはライブラリ単位等によって処理するコアを固定する，もしくは，並列アプリケーションの各スレッドはほぼ同じコードを実行することを仮定すると，(ii) 各アプリケーションについて処理するコアを固定するということが必要になる．

また，通信機構のオーバーヘッドがこれらのキャッシュミスを減少させることによる効果を下回らなくてはならない．その評価には，実際の I/O バスを経由したオーバーヘッドの測定が必要である．また，複数のスレッドもしくはコアのシステムコール等の呼び出しを集約させることも，効果的であると考えられる．

本節での評価は，用いたデータサイズはいずれも小さく，データアクセスに関するコストは含まれておらず，主に命令キャッシュの影響によるところが大きい．アプリケーションが関数呼び出しに渡すデータは，キャッシュにあることが多く，データ転送のコストと，関数呼び出しに伴うキャッシュミスのコストとのバランスの考慮が必要である．そのため，I/O 処理のうち，I/O デバイスにデータを直接転送できるものは，メニーコアのみでの処理を行うほうが好ましいことがあると考えられる．

6. 関連研究

多くのコア数をもつ環境でのオペレーティングシステムについては，多くの研究が行われており，そのスケーラビリティの向上の手法としては，主に二つに分類できる．一つは，OS カーネルの機能を分割し一つまたは複数のコアに割り当てるもの，もう一つは，コアごとに OS カーネルを分離し，それらの間で明示的な通信で協調動作することにより一つのオペレーティングシステム機能を提供するものである．

前者の例としては，Corey[13]，fos[14]，GenerOS[15] があげられる．Corey は，アドレス空間や使用するコアをアプリケーションから明示的に指定させることで，false sharing によるキャッシュ競合を削減するとともに，特定のデバイス操作やシステムコールについてアプリケーションの要求で一つのコアで専有させることができ，キャッシュ競合を削減することを目的としたものである．

fos は，メニーコア環境でアプリケーションを動作させるコアとカーネル機能のコアを分割し，スケジューリングをそれらのコアへの割り当て問題とするものである．コアを分割することにより，ロックやキャッシュ汚染によるスケーラビリティの問題を解決しようとするものである．

GenerOS は，Linux をベースとし，システムコールの処理をすべて特定のコアで行うようにすることにより，アプリケーションとのインタフェースを維持したまま，コア数が増加した状況でもキャッシュ競合を減らすものである．

後者の例には，Multikernel[16] がある．Multikernel は，各コアごとに基本的には独立したカーネルを動作させることで，カーネルの扱うデータ構造を分離し，キャッシュ競合を減らすものである．他のコアと協調する必要がある時には，明示的なメッセージパッシングによって行われる．

しかし，これらのオペレーティングシステムには，本稿で想定するようなハードウェア構成を直接の対象としているものはない．

7. おわりに

クラスタ型のスーパーコンピュータの傾向としては，マルチコアの CPU に加えて，一定の計算に特化した GPU を搭載したものが増えている．今後 GPU に代わり，汎用コアを使用したメニーコアが搭載したシステムが増加すると考えられる．我々は，そのようなシステムでのオペレーティングシステムの設計にあたり，カーネルやライブラリがキャッシュに与える影響に着目し，既存のマルチコア環境において予備評価としてパフォーマンスカウンタを用いた実験を行った．その結果は，システムコールの処理やライブラリ関数の実行について，ある程度の関数やシステムコールのまとまりごとにコアを割り当てる，もしくはプロセスごとにコアを割り当てるようにすることで，キャッシュミスを減らしシステムの高速化を図ることができることを示唆している．

今後の課題として，本稿では主にミクロ的な視点から一つのシステムコールや関数の呼び出しのコストについての評価を行ったが，アプリケーションにおいてこれらの影響がどう現れるかの評価が必要と考えられる．また，シングルコアの場合での評価であったが，複数のコアが同時に実行した場合の，カーネル等の影響についての測定・考察を行い，実際のメニーコア環境においてオペレーティングシステムの設計を行う必要があると考える．

参考文献

1) Intel: First the Tick, Now the Tock: Intel(R) Microarchitecture (Nehalem), http://www.intel.com/technology/architecture-silicon/next-gen/319724.pdf.
2) Advanced Micro Devices: AMD Opteron Processor Product Data Sheet, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf.
3) TOP500.org: TOP 500 Supercomputing Sites - November 2010, http://www.top500.org/lists/2010/11.

Vol.2011-ARC-195 No.2
Vol.2011-OS-117 No.2
2011/4/13

4) NVIDIA Corporation: NVIDIA Tesla GPUs Power World's Fastest Supercomputer, http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&prid=678988&releasejsp=release_157.
5) 松岡　聡：TSUBAME2.0の概要, http://www.gsic.titech.ac.jp/~ccwww/TSUBAME20.pdf.
6) 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, New York, NY, USA, ACM, pp.18:1–18:15 (2008).
7) Dubey, P. and Singhal, R.: Architecture Support for HPC Applications, Intel Develop Forum 2010.
8) Dally, B.: GPU Computing To Exascale and Beyond, http://www.nvidia.com/content/PDF/sc_2010/theater/Dally_SC10.pdf.
9) Wallance, D.: Compute Node Linux: Overview, Progress to Date & Roadmap, *Proceedings of the 2007 Cray User Group Annual Technical Conference*.
10) Nakashima, H.: T2K Open Supercomputer: Inter-university and Inter-disciplinary Collaboration on the New Generation Supercomputer, *ICKS '08: Proceedings of the International Conference on Informatics Education and Research for Knowledge-Circulating Society (icks 2008)*, Washington, DC, USA, IEEE Computer Society, pp.137–142 (2008).
11) Boyd-Wickizer, S., Clements, A.T., Mao, Y., Pesterev, A., Kaashoek, M.F., Morris, R. and Zeldovich, N.: An analysis of Linux scalability to many cores, *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, USENIX Association, pp.1–8 (2010).
12) Mucci, P.J., Browne, S., Deane, C. and Ho, G.: PAPI: A Portable Interface to Hardware Performance Counters, *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp.7–10 (1999).
13) 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).
14) Wentzlaff, D. and Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores, *SIGOPS Oper. Syst. Rev.*, Vol.43, pp.76–85 (2009).
15) Yuan, Q., Zhao, J., Chen, M. and Sun, N.: GenerOS: An asymmetric operating system kernel for multi-core systems, *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp.1 –10 (2010).
16) 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).