Vol.2014-OS-130 No.17
2014/7/29

Xeon Phi搭載計算機における DMA・MMIO併用型CPU間データ通信機構

佐藤 未来子[1,4,a)]　島田 明男[2,4,b)]　吉永 一美[2,4,c)]　辻田 祐一[2,4,d)]　堀 敦史[2,4,e)]　石川 裕[2,3,f)]
並木 美太郎[1,4,g)]

概要：エクサスケールのスーパコンピュータの実現に向けて，マルチコア・メニーコア混在型計算機において，各 CPU の特性を活かしたタスク配置とタスク連携を低オーバヘッドで実現するためのプログラム実行基盤「Multiple PVAS」を提案している．Multiple PVAS は，マルチコア・メニーコア上の各タスクのアドレス空間が単一の仮想アドレス空間を形成する実行モデルであり，双方の CPU で管理する物理メモリに対して Memory Mapped I/O（MMIO）方式によるデータアクセスをサポートして，異なる CPU 上のタスク同士が協調動作可能な実行基盤を実現している．本研究では、Intel Xeon Phi を搭載するマルチコア・メニーコア混在型計算機において，Multiple PVAS のタスクが利用する MMIO データリードの性能を改善する機構について述べる．

1. はじめに

近年では，同一ノード上に高いシングルスレッド性能を指向するレイテンシコアで構成されたマルチコア CPU と，コアを多数搭載してスループット重視のメニーコア CPU を組み合わせて，システムの高性能化を図る，マルチコア・メニーコア混在型計算機が有望視されている [1]．このようなマルチコア CPU とメニーコア CPU が混在するハイブリッド型計算機において高いアプリケーション実行性能を実現するためには，各 CPU コアの特徴を生かした適切なタスク配置や，タスク間の低オーバヘッドな連携が行える実行基盤が必要だと考える．本研究では，マルチコア・メニーコア混在型計算機を対象に，エクサスケールのスーパコンピュータの実現に向けた性能向上を目指してシステムソフトウェアの研究開発を進めている [2][3][4][5]．

1 東京農工大学
Tokyo University of Agriculture and Technology
2 独立行政法人理化学研究所計算科学研究機構
RIKEN AICS
3 東京大学
University of Tokyo
4 独立行政法人科学技術振興機構 CREST
JST CREST
a) mikiko@namikilab.tuat.ac.jp
b) a-shimada@riken.jp
c) kazumi.yoshinaga@riken.jp
d) yuichi.tsujita@riken.jp
e) ahori@riken.jp
f) ishikawa@is.s.utokyo.ac.jp
g) namiki@cc.tuat.ac.jp

本研究では，マルチコア CPU とメニーコア CPU の両方の特性を活かすことを目的としたアプリケーションプログラム実行基盤「Multiple PVAS (Partitioned Virtual Address Space)」を提案している [6]．Multiple PVAS は，一つの仮想アドレス空間に複数の PVAS Task のアドレス空間を配置して実行する独自のタスク実行モデルを対象に，マルチコア CPU およびメニーコア CPU 上の PVAS Task が仮想アドレス空間を共有しながら，双方の CPU に対してタスクを起動・制御，処理委託ができる自由度の高いタスク実行基盤を提供している．Multiple PVAS では，本実行基盤を利用するアプリケーションプログラマが各 CPU へ配置した PVAS Task を少ないオーバヘッドで連携できることを特徴としている．そのため Multiple PVAS の設計では，PVAS Task 間での相互処理依頼に要する数 10 バイト程度の細かいデータ転送を低遅延に実施することを優先している．このことは，アプリケーションプログラム中で双方の CPU に対してまとまったサイズのアクセスを行う際の性能低下を招く原因となる．特に別 CPU で得た演算結果，処理結果をもう一方の CPU でまとめて読みだすといった典型的なプログラミングスタイルにおけるデータ転送性能の向上は解決すべき課題である．

本論文では，Multiple PVAS における CPU 間データ通信方式において，別 CPU のメモリに対する Read 性能の向上を目的とした「高速 Read キャッシュ」を提供するためのシステムソフトウェア設計について述べる．本方式では，アプリケーションプログラムが大量のデータ Read を

行いたい場合に，プログラマが Multiple PVAS に対して高速 Read キャッシュを要求するインタフェースを提供する．本インタフェースを使用しない通常の CPU 間データ通信では，Memory Mapped I/O（MMIO）方式によるダイレクトなメモリアクセスを行う．本インタフェースで指定した領域に関しては，MMIO 方式で別 CPU 上のメモリを Read している最中に，Multiple PVAS がシステムソフトウェアレベルで DMA を活用して別 CPU 上の必要なメモリ領域をローカルメモリへコピーし，コピー後に本ローカルメモリからの Read に切り替えることで高速 Read が可能なキャッシングメモリ領域「高速 Read キャッシュ」を実現する．机上評価において，本設計に要するシステムソフトウェアのメモリ管理のオーバヘッドを見積もり，高速 Read キャッシュ方式が異なる CPU 間でのデータ Read に有効であることを示す．

2. システム構成

2.1 マルチコア・メニーコア混在型計算機の構成

本研究で想定するマルチコア・メニーコア混在型計算機の構成を図 1 に示す．Intel Xeon Phi に代表される並列演算性能の高いメニーコア CPU と，CPU 単体性能の高いマルチコア CPU とが PCI Express バスで相互接続されている．それぞれの CPU が独立にメモリを搭載し，バスを介して双方のメモリ領域を互いのアドレス空間へマッピングして直接アクセスすることができる．

本研究では，ホストとなるマルチコア CPU として Intel Xeon，メニーコア CPU として Intel Xeon Phi で構成されるノード計算機を対象として設計・実装をすすめている．本計算機では，ハードウェアでキャッシュコヒーレンシを保つ機能を備えているため，Memory Mapped I/O（MMIO）方式で別 CPU に搭載されるメモリへアクセスする場合でも、システムソフトウェアによる特別なキャッシュコヒーレンシ制御なしでも直接メモリアクセスが可能である．また，本計算機では Xeon Phi の内部バスと PCI Express バスのブリッジである Transaction Control Unit (TCU) に搭載された DMA コントローラを利用できる．Xeon Phi と Xeon の双方から DMA を要求することが可能であり，転送完了時には Xeon Phi と Xeon のいずれかに割り込みを発生させることができる．

2.2 Multiple PVAS

Multiple PVAS は，マルチコア CPU 側とメニーコア CPU 側でそれぞれ稼働する Linux Kernel をベースとするプログラム実行基盤である．Multiple PVAS の基盤となる PVAS（Partitioned Virtual Address Space）[3] は，プロセスとスレッドの中間に位置するタスクモデルである．PVAS においてコアを割り当てる実行実体であるプロセスを “PVAS Task” と呼び，図 2 中央に示す「PVAS 空間」と

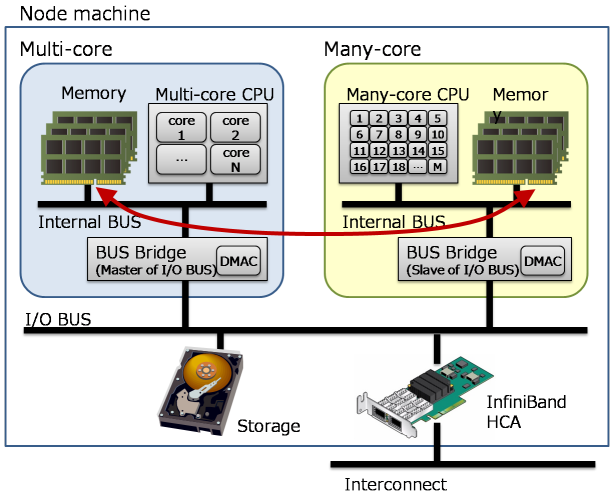


図 1 マルチコア・メニーコア混在型計算機の構成

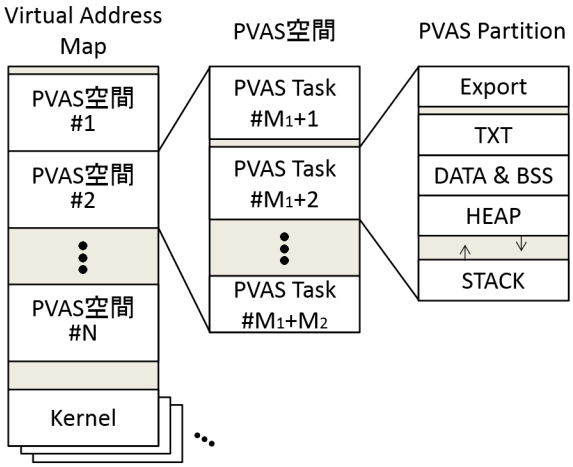


図 2 Multiple PVAS の仮想アドレス空間 [6]

いう一つの仮想アドレス空間に PVAS Task の各アドレス空間として “PVAS Partition” を配置し，PVAS 空間で一つのページテーブルを共有する．これにより，PVAS Task 間では共有メモリを確保することなく，仮想アドレス参照による Task 間データ共有を可能としている．

また，図 2 右側に示すように，PVAS Partition の先頭には “Export” 領域を設けている．この領域は PVAS Task の識別番号からそのアドレスを知ることができる設計とし，他の PVAS Task との情報の受け渡しや同期のための領域として使うことを目的としている．

Multiple PVAS はマルチコア CPU，メニーコア CPU ごとに管理される PVAS 空間を包括する形へ拡張した大域仮想アドレス空間モデルである．すなわち，図 2 左端の Virtual Address Map に示すとおり，異なる CPU の PVAS 空間を単一の仮想アドレス空間へマップし，Multiple PVAS に属する PVAS Task をすべて同じ仮想アドレス空間において実行可能とする．これにより，Multiple PVAS 空間内の仮想アドレスによる情報の受け渡しが，異なる CPU 上の PVAS Task 間でも行えるようになり，複数 CPU で稼働する PVAS Task や Agent システムを自由に構成できる．

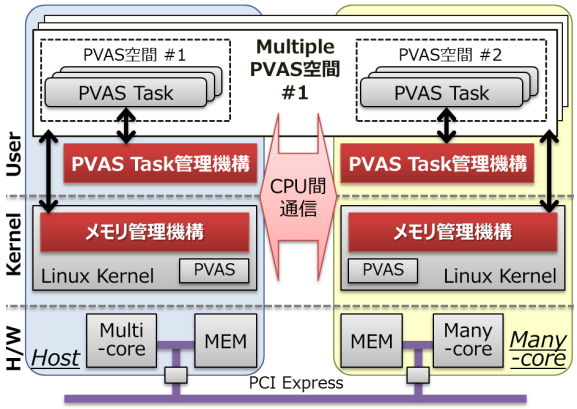
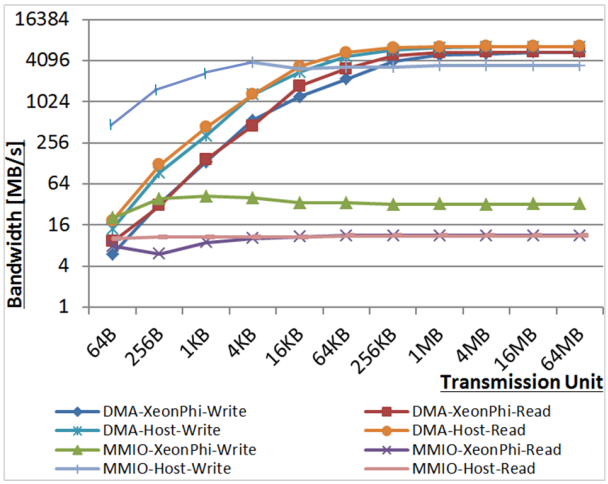


図 3　Multiple PVAS を含む全体構成 [6]



<DMA/MMIO>-<動作主体CPU>-<Read/Write>
（測定環境：Xeon X5680, Xeon Phi 5110P, MPSS2.1.6720-13）

図 4　Xeon Phi - Host 間でのデータ転送帯域

図 3 に Multiple PVAS 空間の概念図とこれを管理するシステム全体構成を示す．Linux Kernel には Multiple PVAS の大域仮想アドレス空間を構築・維持するため “メモリ管理機構” を備え，ユーザレベルには PVAS Task の生成・監視を行なう “PVAS Task 管理機構” を備え，可能な限りユーザレベルで管理制御を行うことで制御オーバヘッドの軽減を図っている．メモリ管理機構では，PVAS 空間を管理するためのページテーブルの管理，ページフォルト時のメモリ管理等の特権レベルで行う必要のある処理を担っている．これらの機構をノード計算機上の各 CPU に設けることで，Multiple PVAS 空間に属する PVAS Task 同士が，どの CPU 上においても PVAS Task に対する起動・制御，処理委託できる自由度の高いタスク実行基盤としている．なお，本論文で提案する高速 Read キャッシュ方式に関しては，メモリ管理機構への追加機能となる．

2.3　CPU 間データ通信の課題と目標

2.1 節で述べたように，対象としている Xeon Phi を搭載する計算機では，MMIO と DMA による CPU 間メモリアクセス方式を利用可能である．MMIO の利点である CPU 命令においてローカルメモリと別 CPU のメモリを等価に扱える点，キャッシュコヒーレンシをハードウェアで保障できる点を重視し，Multiple PVAS では MMIO 方式によるメモリアクセス方式を採用している [6]．MMIO 方式であれば，Multiple PVAS における PVAS Task 間の処理依頼に必要な数 10 バイト程度のデータ転送を低遅延に実施できることも，図 4 に示す予備評価で確認している．

しかし，MMIO 方式の場合，アプリケーションプログラム中で双方の CPU に対してまとまったサイズのアクセスを行う場合には MMIO のメモリアクセス性能が原因でプログラム実行性能の低下を招くことが懸念される．特に別 CPU で得た演算結果，処理結果をもう一方の CPU でまとめて読みだすといった典型的なプログラミングスタイルにおいては，データ転送が性能低下が原因となりうる．

そこで本研究では，システムソフトウェアのサポートにより図 4 に示す DMA 性能を生かし，別 CPU に対するメモリアクセス性能を向上させる方式を検討し，提案する．メモリ Write に関しては，別 CPU のメモリとローカルメモリとのデータ一貫性保証制御が複雑化するため，本論文では対象とはせず，まずはメモリ Read 性能を向上させる方式に特化した検討を行い，CPU 間データ通信性能のさらなる向上に向けた指標を得ることを目標とする．

3.　DMA と MMIO との併用方式の検討

2.3 節で示したとおり，MMIO 方式は小さいサイズのデータを扱うには効率的な方式である．これに対して DMA を用いたデータ通信方式では大量データ転送時に高帯域となるため，CPU 間のデータ転送に活用することは有効である．しかし，ハードウェア初期化オーバヘッドが必要であること，DMA 対象となったメモリに関しては，各 CPU のメモリに同一のデータのコピーが存在するためにデータコヒーレンシを保証する必要があるという欠点があるため，MMIO との使い分けが重要であると考える．

アプリケーションプログラムにおいては，ある程度のまとまったデータ Read を行うことをプログラマは知っており，どのデータの転送性能を向上させたいか，というヒントをシステムソフトウェアへ通知することは可能である．また一方で，システムソフトウェアやコンパイラが転送性能を向上させる領域を自動的に判断する方式も考えられる．プログラムからのヒント情報を用いる方式では，システムソフトウェアやコンパイラがプログラムを解析して MMIO と DMA のどちらを使うべきかを自動的に判断する必要はなくなり，アクセス性能を向上させたい箇所にしぼった効率の良い処理を実施できると考える．

したがって本研究では，アプリケーションプログラム中で一時的に Read アクセス性能を向上させたいメモリ領域

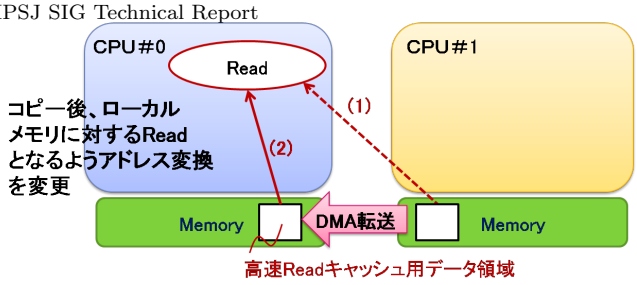


図 5　高速 Read キャッシュ方式の概念図

を明示的に指定するという方法をとる．そして，プログラムに書かれたヒント情報を元に，システムソフトウェアが DMA を活用したコピーを実施することにより，Read 性能を向上させる方式を提案する．

3.1　高速 Read キャッシュ方式の概要

本論文では，MMIO 方式と DMA 方式を併用して別 CPU のメモリに対するメモリ Read 性能を向上させる方式を「高速 Read キャッシュ方式」と呼ぶ．**図 5** に，高速 Read キャッシュ方式の特徴を示す概念図を示す．通常は，MMIO 方式により直接別 CPU のメモリをアクセスすることにより，メモリコピー操作を排除している．本高速 Read キャッシュ方式を利用する場合，プログラムが通常と同じ MMIO 方式のメモリ Read を実施している最中に（図 5 中の (1)），システムソフトウェアが高速 Read キャッシュの対象領域をページ単位でローカルのメモリへ自動的に DMA 転送する．転送が完了した後，同じ仮想アドレスでローカルメモリへのアクセスが行えるようアドレス変換情報を切り替える（図 5 中の (2)）．

本方式の特徴は，システムソフトウェアによる対象領域のコピー中であっても，プログラムは実行継続可能であり，その間は通常の MMIO により対象領域を Read することも可能である点である．ローカルメモリへの転送中にタスクの実行をサスペンドしなければならないのであれば，メニーコア CPU の演算性能を低下させる原因となってしまうが，MMIO 方式を併用することでプログラムの実行を継続可能としている．また，メモリ Read に限定しているので，高速 Read キャッシュ領域に再びデータの Write 処理を行いたい場合でも，システムソフトウェアにより元のアドレス変換情報を復帰させて，通常の MMIO 方式でのデータアクセスへ戻すこともできる．

3.2　高速 Read キャッシュ方式を適用する対象メモリの指定方法

3 章の最初に述べたように，本研究では，高速 Read キャッシュ方式の開始と終了をプログラム中で明記されることを前提とした設計としている．**図 6** に，高速 Read キャッシュ方式を利用する際の擬似コードを示す．擬似コード中に示すように，高速 Read キャッシュ実施期間中には，プログラマが明示的に対象領域への書き込みを避けることに

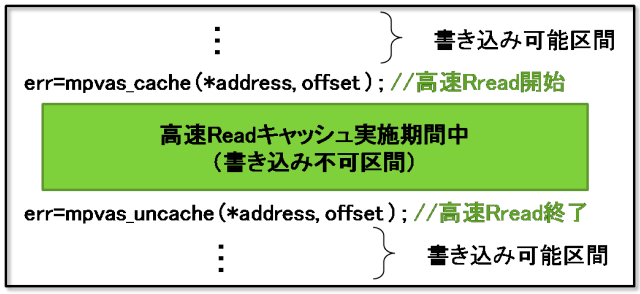


図 6　高速 Read キャッシュの指定方法の擬似コード

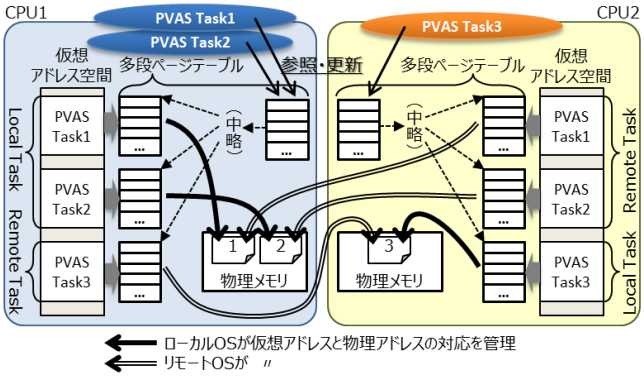


図 7　ページテーブルの構成

より，Read 性能を確保する設計としている．DMA を併用する場合，3 章の最初に述べたとおり複数の CPU でメモリのコピーを持つこととなるため，高速 Read キャッシュ用のデータ領域には Write 制限を特権レベルで設定する．もし，高速 Read キャッシュ実施期間中に Write 処理を行いたい時，あるいは，行った時には，3.1 節にてふれたように，システムソフトウェアにより MMIO 方式のデータアクセスへ戻すことにより，プログラムの実行継続を可能としている．

4.　高速 Read キャッシュ方式のためのメモリ管理の設計

本高速 Read キャッシュ方式を，Multiple PVAS へ適用する場合のメモリ管理の設計について述べる．

4.1　Multiple PVAS の仮想アドレス空間管理

Multiple PVAS では**図 7** に示すように，アプリケーション実行性能を落とすことなく大域仮想アドレス空間を実現するため，大域仮想アドレス空間を構成するページテーブルを CPU ごとに別個に構築する設計としている．すなわち，各 CPU の OS が自身の PVAS 空間のメモリ管理（物理メモリ割り当てやページテーブル管理）を行なっている．なお，別 CPU 上の PVAS 空間の仮想アドレスに対するページフォルト処理に関しては，別 CPU の OS が管理するページテーブルを直接参照し，必要なアドレス変換情報を参照し設定することで，ページ管理のオーバヘッドを

最小限としている．また，図 7 に示すように PVAS では同一 CPU 上の全ての PVAS Task が一つのページテーブルを共有するため，同一 CPU 上の PVAS Task のページフォルト発生回数を最小限に抑える設計となっている．

4.2 高速 Read キャッシュ方式向けの仮想アドレス空間管理

高速 Read キャッシュ方式を適用する場合には，通常の MMIO 方式でアクセスするために物理アドレス空間上にマップしてあるページと，高速 Read キャッシュ用にコピーした物理アドレス空間上のページとを，ページテーブルの物理アドレスを差し替えることで実現できる．ただし，4.1 節で述べたように，PVAS では CPU ごとに PVAS Task がページテーブルを共有する設計となっている．したがって，PVAS 空間を共有する同一 CPU 上の PVAS Task が 3.2 節で述べた高速 Read キャッシュ方式のインタフェースを用いて高速 Read キャッシュを利用している場合，TLB ミス発生により別の PVAS Task も同じ高速 Read キャッシュ利用可能となる．これについては，二つの方式が考えられる．

一つは，指定のインタフェースを利用せずに高速 Read キャッシュ方式の利用を許す方式である．この場合，同じ PVAS であれば，誰かが一度 mpvas_cache/uncache() を行えばよく，システムソフトウェアの負担は軽く済む．しかし，mpvas_uncache() の際に，その PVAS 空間に属する全ての PVAS Task の TLB エントリをフラッシュすることが必要となる．このことは，コア数の多いメニーコア CPU においては問題となる．

もう一つは，mpvas_cache/uncache() を実行した PVAS Task のみに参照を許す方式である．この場合，mpvas_cache/uncache() を利用した PVAS Task の TLB エントリのフラッシュを実施することで一貫性を保障することが可能である。しかし，これらのインタフェースを用いない PVAS Task から高速 Read キャッシュへアクセスされる可能性もあり，それを防ぐことは難しく，その場合の動作は保障できない．

本研究では，マルチコア CPU とメニーコア CPU のハイブリッド型の計算機への適用を目指しているため，PVAS Task を実行するコア数は数十〜数百を想定している．その場合に，TLB エントリのフラッシュのコストが大きいことはシステム性能上問題となりうる．したがって，mpvas_cache/uncache() を実行した PVAS Task のみに参照を許す方針とし，PVAS 空間全域での TLB エントリフラッシュを避ける方針とする．

4.3 高速 Read キャッシュ方式向けのページコピー方式

3.1 節で述べたように，高速 Read キャッシュ方式では別 CPU のメモリ領域をページ単位でコピーする．また，本研究で対象ハイブリッド型のシステムでは，どちらの CPU からも DMA を実行することが可能となっている．したがって，設計として最も効率の良い DMA 転送方式を選択することが妥当な設計といえる．

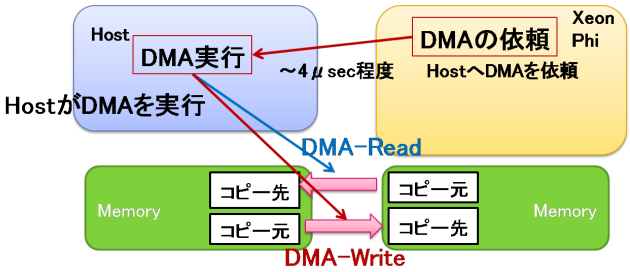


図 8 DMA によるページコピーの方式

2.3 節の図 4 に示す DMA による CPU 間データ転送の予備評価によれば，ホスト CPU である Xeon における DMA 性能が Read/Write ともに高帯域で実現できることがわかる．したがって，Xeon Phi から Xeon へのメモリコピーは Host の DMA-Read を使う設計としている（**図 8** 参照）．また，2.3 節の予備評価によれば，Xeon における DMA-Read/Write が 4 KB ページあたり約 3μsec であり，Xeon Phi から Xeon に対して処理依頼をするための通信コストは文献 [6] より約 4μsec 必要である．一方で，Xeon Phi における DMA-Read/Write が 4 KB ページあたり約 8μsec であることから，処理依頼の通信コストを加味しても，ホストへ DMA を依頼したほうがページコピーの効率がよいということから，Xeon からの Xeon Phi へのメモリコピーは Host の DMA-Write を使う設計としている．

4.4 高速 Read キャッシュ方式利用時のメモリ管理の流れ

4.1 節で述べた Multiple PVAS の仮想アドレス空間管理において，3.2 節で述べた高速 Read キャッシュ方式のインタフェースを用いて高速 Read キャッシュを利用する場合のメモリ管理方式について述べる．**図 9** に “mpvas_cache()” による対象となるメモリのコピー処理の流れを示す．“mpvas_cache()” では，引数に高速 Read キャッシュ方式の対象である仮想アドレスの情報が入っているので，これを元に対象となるメモリをローカルメモリへコピーし，ページテーブルを設定する．まず，mpvas_cache() によりカーネルへ遷移し mpvas_cache() の引数で示されている仮想アドレスからターゲットメモリの物理アドレス情報を得る．対象となるメモリのコピー用の物理メモリを物理アドレス空間上に確保し，ページ単位でのコピーを行う．ページテーブル管理においては，元の MMIO 方式へ戻すための情報として書き換え前のアドレス変換情報をカーネル内に退避しておく．コピー完了後は，対応する仮想アドレスに対するページテーブルエントリの情報をコピーした物理アドレ

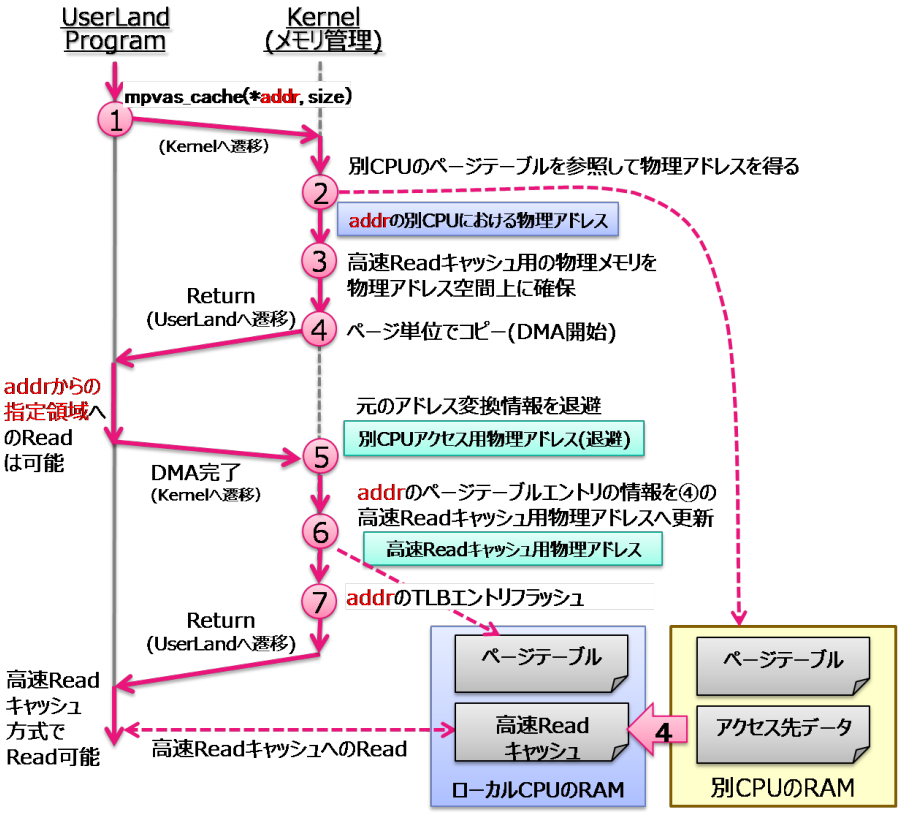


図 9　高速 Read キャッシュ方式利用時のメモリ管理の流れ

スへ変更し，ライトプロテクトを設定し，TLB のエントリフラッシュを実行する．これにより，新しくページテーブルへ設定したエントリが TLB に設定され，プログラムを高速 Read キャッシュ方式で実行され始める．

同様に，3.2 節で述べた “mpvas_uncache()” により元の MMIO 方式へメモリ管理を戻す場合には，mpvas_uncache() によりカーネルへ遷移して，対象の仮想アドレスの TLB エントリをフラッシュする．そして，退避しておいた対象メモリのアドレス変換情報を，元のページテーブルエントリへ書き戻し，高速 Read キャッシュ用に確保した物理メモリを解放した後に，プログラムを続行する．これにより，元のアドレス変換情報が TLB エントリに設定され，MMIO 方式による別 CPU のメモリへのアクセスの状態に戻ることができる．

5. 評価

本論文で提案する高速 Read キャッシュ方式に関して，本設計に要するシステムソフトウェアのメモリ管理のオーバヘッドを机上評価で見積もり，高速 Read キャッシュ方式が別 CPU のメモリ Read に有効であるかどうかを検証する．**表 1** に，高速 Read キャッシュ方式に係る主な制御オーバヘッドを示し，**表 2** に表 1 の項目に要するオーバヘッドを机上で加算して得られた，DMA によるメモリコピー性能と同一サイズの MMIO 方式の性能を示す．

表 1　高速 Read キャッシュ方式の主な制御オーバヘッド項目

#	項目
1	カーネル遷移オーバヘッド
2	INVLPG 命令による TLB エントリフラッシュ
3	TLB ミスオーバヘッド
4	ページテーブルロック獲得・解放
5	ページテーブル参照 (ローカル)
6	ページテーブルエントリ変更
7	元のエントリの退避
8	DAM によるページ転送 (4KB/2MB 等のページ単位で)
9	Xeon Phi から Host への DMA 依頼

表2　高速Readキャッシュ方式を適用した場合の転送性能（単位：μ sec）

	DMAによるメモリコピー性能			MMIO方式によるメモリアクセス性能	
	Xeon	XeonPhi1	XeonPhi2	MMIO-Host	MMIO-Phi
4KB	5	15	16	295	313
8KB	10	24	30	589	625
16KB	19	45	61	1179	1251
32KB	37	80	117	2357	2501
64KB	73	156	233	4715	5003

Xeon: Xeonにおける，XeonからXeon PhiのDMA-Read
XeonPhi1: Xeon PhiからXeonへ依頼し，XeonにおけるDMA-Write
XeonPhi2: Xeon Phiにおける，Xeon PhiからXeonのDMA-Read

高速 Read キャッシュ方式を用いる場合には，システムソフトウェアに対して DMA を用いたページコピーやメモ

リ管理を依頼しなければならない．したがって，それらがXeon 上のカーネルおよび Xoen Phi 上のカーネルにおいてどの程度のオーバヘッドとなるかを検証する必要がある．表 2 では，4 KB のページサイズ以上でいくつかのデータサイズを対象にデータ転送にかかる性能を机上評価した．結果として，どのサイズにおいても，MMIO 方式で同じサイズをアクセスし続けた性能よりも，高速 Read キャッシュ方式を利用する際のデータ転送オーバヘッドは非常に小さいという結果が得られた．このことは，別 CPU のメモリに対してまとまったデータ Read を行いたい場合に，本提案の高速 Read キャッシュ方式をシステムソフトウェアへ適用して，ローカルメモリへのデータ転送をシステムソフトウェア制御で行うことが性能上有効であることを示している．もちろん，アプリケーションプログラムの内容により，ページ単位で転送したうちの何%を読みだすかにより高速 Read キャッシュ方式の効果は異なる．そのため，実際に Multiple PVAS の実行基盤へ本方式を適用し，引き続きアプリケーションプログラムを用いた性能評価を行うことが今後の課題となる．

6. おわりに

本論文では、Intel Xeon Phi を搭載するマルチコア・メニーコア混在型計算機において，本研究が提案する Multiple PVAS のタスク実行基盤における MMIO 方式によるデータ Read 性能を改善する機構について述べた．別 CPU のメモリに対する Read 性能の向上を目的とした「高速 Read キャッシュ」を提供するためのシステムソフトウェア設計について述べ，机上評価において，本設計に要するシステムソフトウェアのメモリ管理のオーバヘッドを見積もり，高速 Read キャッシュ方式が異なる CPU 間でのデータ Read に有効であることを示した．今後は，本方式を Multiple PVAS をはじめとするマルチコア・メニーコア混在型計算機のシステムソフトウェアへの適用を考え，実アプリケーションによる評価検証を行う．

謝辞 本研究は，科学技術振興機構（JST）戦略的創造研究推進事業（CREST）における研究領域「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」によるものである．

参考文献

[1] TOP500 Supercomputer Sites: June 2013, `http://www.top500.org/lists/2013/06/` (2013).
[2] 科学技術振興機構：CREST, `http://www.jst.go.jp/kisoken/crest/`.
[3] Shimada, A., Gerofi, B., Hori, A. and Ishikawa, Y.: Proposing a new task model towards many-core architecture, *Proceedings of the First International Workshop on Many-core Embedded Systems*, MES '13, pp. 45–48 (2013).
[4] 堀　敦史, 島田明男, 並木美太郎, 佐藤未来子, 深沢　豪, 辻田祐一, 石川　裕：メニーコア用 Agent プログラミング環境の提案，情報処理学会「ハイパフォーマンスコンピューティング研究会」第 140 回研究報告，Vol. 2013-HPC-140, No. 32, pp. 1–8 (2013).
[5] Sato, M., Fukazawa, G., Yoshinaga, K., Tsujita, Y., Hori, A. and Namiki, M.: A Hybrid Operating System for a Computing Node with Multi-Core and Many-Core Processors, *International Journal of Advanced Computer Science (IJACSci)*, Vol. 3, No. 7 (2013).
[6] 深沢　豪，佐藤未来子，吉永一美，辻田祐一，島田明男，堀　敦史，並木美太郎：メニーコア混在型並列計算機向け大域仮想アドレス空間モデル Multiple PVAS の提案，情報処理学会「ハイパフォーマンスコンピューティング研究会」第 141 回研究報告，Vol. 2013-HPC-141, No. 7, pp. 1–10 (2013).
[7] Potluri, S., Bureddy, D., Hamidouche, K., Venkatesh, A., Kandalla, K., Subramoni, H. and Panda, D. K. D.: MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, No. 54, pp. 1–11 (2013).
[8] Newburn, C., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chinchilla, F. and McGuire, R.: Offload Compiler Runtime for the Intel Xeon Phi Coprocessor, *Supercomputing*, Lecture Notes in Computer Science, Vol. 7905, Springer Berlin Heidelberg, pp. 239–254 (2013).
[9] Green, R. W.: Native and Offload Programming Models (online), `http://software.intel.com/en-us/articles/native-and-offload-programming-models` (2012).
[10] Reinders, J.: *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors*, Intel (2012).
[11] Hamidouche, K., Potluri, S., Subramoni, H., Kandalla, K. and Panda, D. K.: MIC-RO: Enabling Efficient Remote Offload on Heterogeneous Many Integrated Core (MIC) Clusters with InfiniBand, *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 399–408 (2013).
[12] Zhou, S., Stumm, M., Li, K. and Wortman, D.: Heterogeneous Distributed Shared Memory, *IEEE Trans. Parallel Distrib. Syst.*, Vol. 3, No. 5, pp. 540–554 (1992).
[13] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N. and Hwu, W.-m. W.: An asymmetric distributed shared memory model for heterogeneous parallel systems, *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pp. 347–358 (2010).
[14] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R. and Mendelson, A.: Programming Model for a Heterogeneous x86 Platform, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 431–440 (2009).
[15] Yan, S., Zhou, X., Gao, Y., Chen, H., Wu, G., Luo, S. and Saha, B.: Optimizing a Shared Virtual Memory System for a Heterogeneous CPU-accelerator Platform, *SIGOPS Oper. Syst. Rev.*, Vol. 45, No. 1, pp. 92–100 (2011).