Vol.2013-DPS-154 No.18
Vol.2013-CSEC-60 No.18
2013/3/14

動的ハイブリッドキャッシュコヒーレンシプロトコル方式

中原　亮[†1]　　金井　敦[†1]

マルチプロセッサにおいて,各プロセッサ間の通信に費やされる時間がその性能向上の妨げとなっている.それを改善するためにこれまでにも多くのキャッシュコヒーレンシプロトコルが考案されてきたが,プログラムの特性に依存して効果が変動してしまう.本論文ではプログラムの特性に依存しない新しいプロトコルを提案する.シミュレーションの結果,我々のプロトコルは評価に使用した全てのアプリケーションに対して性能の向上を達成した.

Study on Dynamic Hybrid Cache Coherency Protocol

AKIRA NAKAHARA[†1]　ATSUSHI KANAI[†1]

In multiprocessor, time spent by the communication between each processor disturbs the performance improvement. So far many cache coherency protocols have been devised to improve it, but the effect fluctuates depending on the characteristic of the program. We suggest a new protocol not depend on the characteristic of the program .As a result of simulation, our protocol achieved a performance gain for all application that we used for an evaluation.

1. はじめに

計算機システムは,現在に至るまで急速な進歩を遂げてきた.それは小型化であり,電力消費量の削減であり,そして実行時間の短縮である.多くの場合,利用者が計算機を選択する際には実行時間の短さ,つまり性能が重要な関心事となる.この性能を向上させるために生まれたのがマルチプロセッサである.これは複数のプロセッサを搭載させて並列計算を行うものであり,最近では１つのチップ内に複数のプロセッサを搭載したオンチップマルチプロセッサが様々な用途で用いられるようになってきている.

マルチプロセッサは複数のプロセッサを並列に動作させ,作業を分担することでデータ処理の速度を向上させる.しかし,単一のプロセッサで動作させた場合に対して,プロセッサを２台３台と増やして動作させれば処理速度も２倍３倍になるかといえばそうではない.何故なら,マルチプロセッサにはプロセッサ間の通信が必要不可欠であり,それぞれのプロセッサが計算をする時間に加え,その通信に要する時間が余計にかかってしまうからである.

プロセッサ間での通信に時間がかかってしまうことはマルチプロセッサの性能向上を妨げる主な原因であり,それを軽減する方法がこれまでにも多く考案されている.主な方法としてはプロセッサ間のネットワーク[1][2]の改善や,キャッシュ・コヒーレンス・プロトコルの改善などがある.

キャッシュ・コヒーレンス・プロトコルとは,キャッシュ間でのデータの一貫性を保つために必要な通信の手順であり,マルチプロセッサが正しい計算結果を導き出すために必要なものである.「データの一貫性を保つ」とは,同じアドレスのデータでも計算途中ではキャッシュ毎に値が違ってくる場合があるので,このようなデータの矛盾を正すことである.データの一貫性を保ちつつプロセッサ間の通信を減らすようなキャッシュ・コヒーレンス・プロトコルを考え出すことができればマルチプロセッサの性能を向上させることができる.代表的なプロトコルとしては MSI プロトコル[3]や TRO プロトコル[4]などが挙げられる.

MSI プロトコルはキャッシュ・コヒーレンス・プロトコルを学習する際にまず習うような基本的なプロトコルである.このプロトコルでは,古いデータが生じた際にそのデータを消去することでデータの一貫性を保つのだが,その消去を行うための信号がプロセッサ間のネットワークを圧迫してしまい,性能向上の妨げとなってしまう.

TRO プロトコルは Self-Invalidation[5]を用いたプロトコルである.このプロトコルでは,各プロセッサが独自に古いデータを消去するため,プロセッサ間の通信を用いることなくデータの一貫性を保つことができる.しかし,このプロトコルで絶対に一貫性を保つためには消去する必要のないデータまでも消去する場合があり,このせいで起こってしまうメモリ・アクセスが性能向上を妨げてしまう.

上の２つのように,どのプロトコルにも有利な場合と不利な場合があり,それはアプリケーション毎に,更にいえばプログラムの特性によって決まってしまう.そこで本論文では,異なるプログラムの特性に対して有効な２つのプロトコルを動的に使い分けるプロトコルを提案する.

2. 関連研究

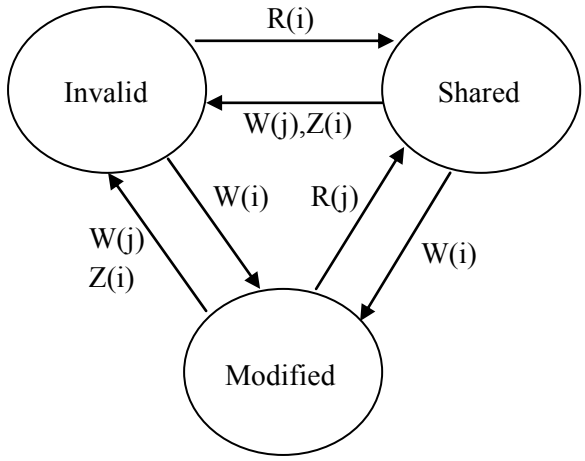
2.1 MSI プロトコル

MSI プロトコル[3]はキャッシュ・ラインが M（Modified）,S（Shared）,I（Invalid）の３状態をとる書き込み時無効化方式のプロトコルである.このプロトコルでは書き込みを

†1 法政大学
Hosei University

行ったキャッシュ・ラインを Modified 状態,読み込みを行ったものを Shared 状態,無効化されたものを Invalid 状態としている.

あるキャッシュのキャッシュ・ラインに書き込みが起こった時,他のキャッシュにある同じデータを持ったキャッシュ・ラインを無効化しなければデータの一貫性が保てなくなる.よって MSI プロトコルではキャッシュ・ライン毎の状態を記憶しておき,キャッシュへの書き込みや読み込みに連動してその状態を図 1 のように遷移させることでデータの一貫性を保つ.なお,図 1 は同じ状態への遷移を省略して書かれたものである.



R:読み込み, W:書き込み, Z:置換
i:ローカル・キャッシュ
j:リモート・キャッシュ

図 1　MSI プロトコルのキャッシュ・ラインの状態遷移図

Figure 1　Cache line state change diagram of MSI protocol

状態遷移はキャッシュ・ラインとディレクトリに保存されている状態を変更することで行われる.つまり,キャッシュとディレクトリの間で信号を送りあう必要がある.書き込みが発生した際に起こる,キャッシュとディレクトリの間で行われる信号のやりとりを図 2 に示す.ここでは全ての Local-Cache が同じキャッシュ・ラインのデータを保存しており,その状態が Shared であるとする.まずは①で PU1 が書き込み命令を行い, Directory1 に対して書き込みのリクエストを送る.次に,②で Directory1 が全ての Shared 状態のキャッシュ・ラインを持つ Local-cache へ Invalidation-request を送る.それを受けた Local-cache はキャッシュ・ラインを Shared 状態から Invalid 状態へ変化させ,③のように Directory1 へ Invalidation が完了した報告として Invalidation-ACK を送信する.そして,全ての Local-cache から Invalidation-ACK を受け取り,データの一貫性が取れることを確認した Directory1 は④で ACK とデータを PU1 へ送信して書き込み命令が完了する.このように書き込み時に

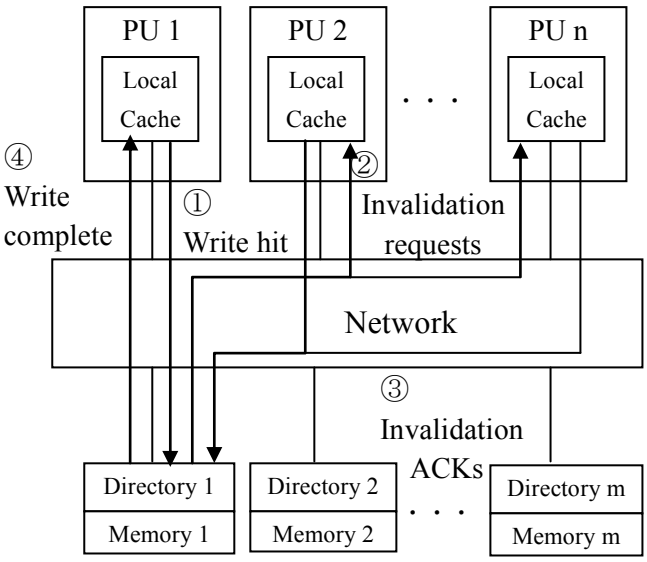


図 2　MSI プロトコルの手順(書き込み時)

Figure 2　Image of MSI protocol (Write)

Shared 状態のキャッシュ・ラインを無効化することを Write-Invalidate と呼ぶ.

ここで②の Invalidation-requests と③の Invalidation-ACKs に注目してもらいたい.これらの信号は Shared 状態のキャッシュ・ラインを持つ全ての Local-cache に対して送受信される.つまり PU の数が増えれば増えるほどに Invalidation に関する大量の信号が Network を同時に流れてしまうのである. よって MSI プロトコルではマルチプロセッサの性能を向上させようとしてプロセッサ数を増やしても,プロセッサを増やしたことによって同時に増えてしまう Invalidation 信号が Network の混雑を生んでしまい性能向上の妨げとなってしまう.

2.2 TRO プロトコル

TRO プロトコル[4]はプロセッサ数を増やしたことによって生じてしまう Network 混雑を解消するために,各プロセッサが独自のタイミングでキャッシュ・ラインに対して無効化を行う Self-Invalidation を使用するプロトコルである.このプロトコルでキャッシュ・ラインがとる状態は Invalid 状態,TRO（tear-off read-only）状態,Read-Write 状態の３つである.Invalid 状態はキャッシュ・ラインが無効な状態,TRO 状態は読み込みが可能な状態,Read-Write 状態は読み込みも書き込みも可能な状態である.その状態遷移図を図 3 に示す.また,この図 3 も同じ状態への遷移は省略して書かれている. 図 3 にも示すように,TRO プロトコルでは TRO 状態のキャッシュ・ラインが同期命令の際に Invalid 状態へ遷移する.この同期命令とは TRO プロトコルのために書き足すものではなく,従来の並列プログラムに含まれているものである.同期命令と同時に Invalidation を行う限

Vol.2013-DPS-154 No.18
Vol.2013-CSEC-60 No.18
2013/3/14

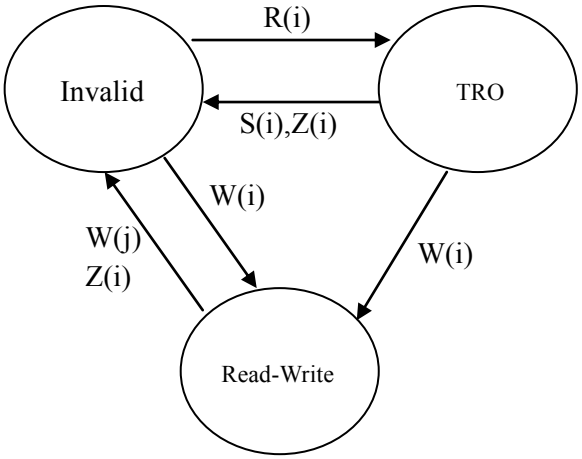


図 3　TRO プロトコルのキャッシュ・ラインの状態遷移図

Figure 3　Cache line state change diagram of TRO protocol

り,キャッシュの一貫性は保たれる.それを共有変数と相互排除を行う例を用いて説明する.

- 相互排除を行う
- 共有変数に読み書きを行う
- 相互排除を解く

複数のプロセッサ間で共有する変数に読み書きを行う場合,競合を防ぐために上のような相互排除,つまり同期命令を読み書きの前と後に行う.よって共有変数へのアクセスの後には必ず同期命令が存在しており,ここで共有変数の保存してあるキャッシュ・ラインを無効化してしまえば,また最新のデータの読み込みを行うのでデータの一貫性は保たれるのである.

書き込みが発生した際のキャッシュとディレクトリの間で行われる信号のやりとりを図 4 に示す.ここでは PUn に Read-Write 状態のキャッシュ・ラインが存在していたとする.①で PU1 が書き込み命令を行い, Directory1 に対して書き込みのリクエストを送る.次に,②で Directory1 が Read-Write 状態のキャッシュ・ラインを持つ PUn の Local-cache に対して Invalidation-request を送信する.PUn の Local-cache は該当キャッシュ・ラインを Invalid 状態に変化させ,③のように Invalidation-ACK を Directory1 へ送信する.そして Invalidation の完了を確認した Directory1 は④で ACK とデータを PU1 に送信し,書き込みが完了する.

TRO プロトコルでは書き込みの際に Invalidation-request を送るのは Read-Write 状態のキャッシュ・ラインだけでよく,Read-Write 状態が存在しなければ Directory との通信だけで書き込みが完了する.つまりプロセッサ数がいくら増えたとしても Invalidation 信号が増えることがなく,それによる Network の混雑を回避することができる.

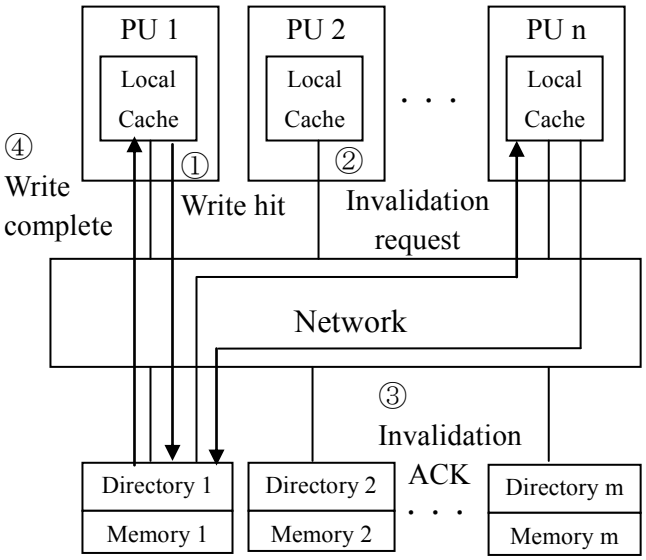


図 4　TRO プロトコルの手順(書き込み時)

Figure 4　Image of TRO protocol (Write)

しかし,TRO プロトコルにも欠点がある.このプロトコルは同期命令に応じて Self-Invalidation を行うのだが,この時に Invalidation の必要がないキャッシュ・ラインまでも無効化してしまい,余計なキャッシュ・ミスが起こってしまうのである.それが起こってしまうプログラムの例を図 5 に示す. 一番上の number_of_processors は大域変数として定義されており,それを読み込んで使用するためキャッシュ・ラインには TRO 状態として保存されている.BARRIER は同期命令なので,この命令の段階で TRO 状態のキャッシュ・ラインである number_of_processors は無効化されてしまう.よってその下の命令にて number_of_processors を読み込んで使用する際にはもう Invalidation が完了しているためキャッシュ・ミスが発生してしまう.

```
int number_of_processors = DEFAULT_P;
                    ・
                    ・
                    ・
BARRIER(global->barrier_rank,number_of_processors);

    if (MyNum != (number_of_processors - 1)) {
                    ・
                    ・
                    ・
```

図 5　TRO プロトコルが不利なプログラム

Figure 5　Disadvantageous program of TRO protocol

Vol.2013-DPS-154 No.18
Vol.2013-CSEC-60 No.18
2013/3/14

TRO プロトコルは Invalidation 信号を減らすことができ, マルチプロセッサの性能向上を達成することができる.しかし,不要なキャッシュ・ミスを引き起こすという欠点も持ち合わせている.TRO プロトコルは複数のプロセッサで共有して読み書きを行うデータに対しては有効だが, 共有しても読み込みしか行わないデータに対しては有効でないといえる.

3. 提案ハイブリッドプロトコル

本章では本論文での提案手法であるプロトコル,MSI プロトコルと TRO プロトコルを組み合わせたハイブリッドプロトコルの説明をする.このプロトコルではプログラムの始めの段階においては MSI プロトコルを使用しているのだが,TRO プロトコルが有効であるメモリ･ブロックを動的に検出してそのキャッシュ･ラインのみに TRO プロトコルへの切り替えを行う.

まずハイブリッドプロトコルのために追加する装置である TRO-bit と Address Buffer についての説明を行い,次に Self-Invalidation,そして状態遷移についての説明を行う.

3.1 TRO-bit と Address Buffer

TRO プロトコルが有効である共有して読み書きを行うメモリ・ブロックはループなどにより複数回に渡り読み書きを行われる傾向にある.よって一度 Write-Invalidate を受けたメモリ･ブロックに対しては TRO プロトコルが有効だと判断し,プロトコルの切り替えを行う.そのプロトコル切り替えのために,TRO-bit と Address Buffer を追加する.

TRO-bit はディレクトリの dirty-bit の隣に追加する新しい bit であり,Local-cache には追加しない.この bit は Write-Invalidate の際に 1 となり,dirty-bit が 0 になると同時に 0 になる.Address Buffer は PU 毎に 1 つずつ追加する.これは TRO プロトコルが有効なメモリ･ブロックのアドレスを保存する記憶装置であり,同期命令に応じた Self-Invalidation を行う装置である. Address Buffer への書き込みは TRO-bit が 1 のメモリ・ブロックに関するアクセスが発生した際に行われ,それに含まれるデータ・アドレスを保存する. 図 6 に Address Buffer のエントリを示す.このエントリはキャッシュ・ラインの置換が発生した際にクリアされ,満杯の場合には LRU アルゴリズムを用いてエントリの置換を行う. また,表 1 に Address Buffer の構成を示す.これは Local-cache と同じアクセス速度の記憶媒体を用いており,TRO が有効なデータが同時期に多く存在することはあまりないのでこのような小さいサイズにしている.

3.2 Self-Invalidation

同期命令時に行う Self-Invalidation の手順と Address Buffer の動作を図 7 に示す.Local-cache に保存されている状態 bit は左が valid-bit であり,valid-bit が 0 ならばそのキャッシュ・ラインの状態は Invalid 状態である.右の状態 bit は dirty-bit であり,valid-bit が 1 で dirty-bit が 0 なら TRO 状態であり,1 なら Read-Write 状態である.Self-Invalidation は同期命令に対応して行うため,同期命令を実行する際には PU は自分の Address Buffer に Invalidation 要求を送る.それを受け取った Address Buffer は自身に格納されているアドレスを取り出し,それによりインデックス付けされた Local-cache のエントリへアクセスする.そして dirty-bit が 0 ならばそれは TRO 状態の無効化するべきキャッシュ・ラインとみなし,Invalidation を行う.

データ・アドレス	LRU
$500	1

図 6 Address Buffer のエントリ
Figure 6 Entries of Address Buffer

表 1 Address Buffer の構成
Table 1 Configuration of Address Buffer

size	ways	latency
32 Byte	8 ways	3 cycle

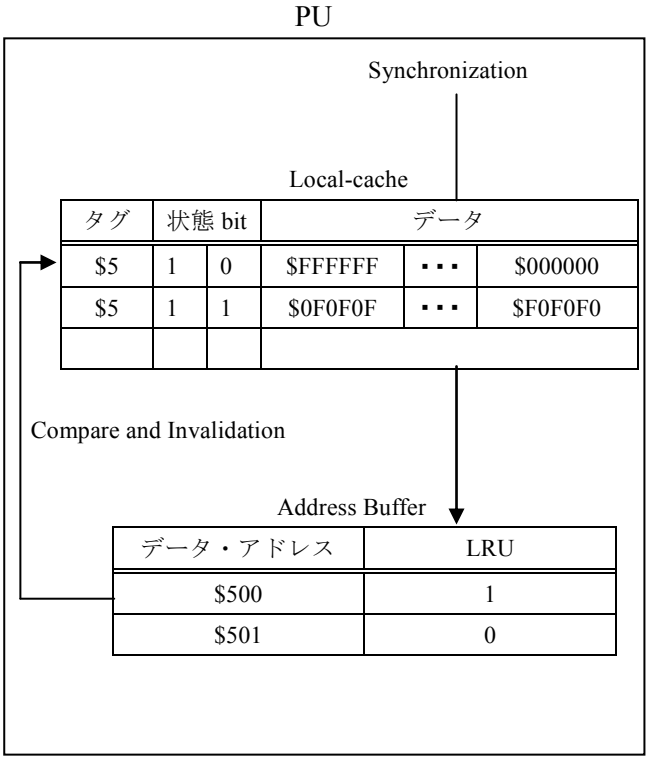


図 7 Self-Invalidation の手順
Figure 7 Process of Self-Invalidation

3.3 キャッシュ・ラインの状態遷移

ハイブリッドプロトコルにおけるキャッシュ・ラインの状態遷移は MSI プロトコルのものと TRO プロトコルのものを図 8 のように切り替えて行われる.ハイブリッドプロトコルは 6 状態を遷移する.しかし,TRO プロトコルの Invalid,Read-wite 状態はそれぞれ MSI プロトコルの Invalid,Modified 状態とほぼ同じ状態であるため,今後の説明は Invalid,Shared,TRO,Modified の 4 状態で行う.

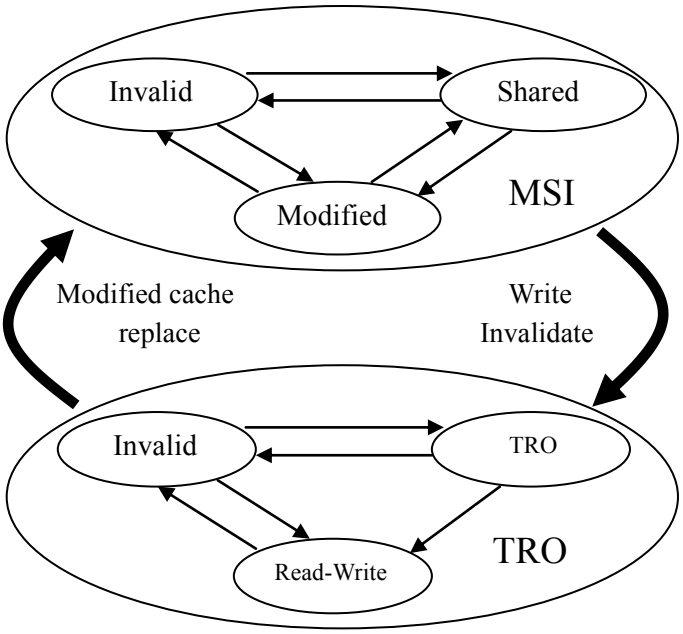


図 8　ハイブリッドプロトコルの状態遷移

Figure 8　Cache line state change diagram of Hybrid protocol

読み込みの際にどのような状態遷移が起こるのかを例を用いて説明する.PU1 で Address Buffer にアドレスが保存されていないが,ディレクトリの TRO-bit が 1 であるメモリ・ブロックに対して読み込みが行われたとする.またこの時のキャッシュ・ラインは Invalid であり, PUn の Address Buffer にもそのアドレスが保存されておらず,キャッシュ・ラインは Modified 状態である.この時の動作を図 9 に示す.PU1 は①のように Directory1 へメモリ・ブロックの読み込みを要求する. ②で Directory1 は最新のデータを持つ PUn へデータを要求する. この時 Local-cache の前に Address Buffer へのアクセスが発生し, そのデータ・アドレスの保存が行われる.この段階でプロトコルが変わり状態遷移が切り替わるため,PUn の Local-cache は状態を Modified から Shared への変更を行わない.次に③で最新のデータを Directory1 に送信する.MSI プロトコルならばここで Directory1 は dirty-bit を 1 から 0 にするのだが,TRO-bit が 1 なので状態遷移は起こらないとして dirty-bit の変更は行わない.そして④で要求されたデータが PU1 に送られ,ここでも Address Buffer へのデータ・アドレス保存が発生する.よって PU1 にもプロトコルの切り替えが起こり,キャッシュ・ラインの状態は TRO として保存されるのである.

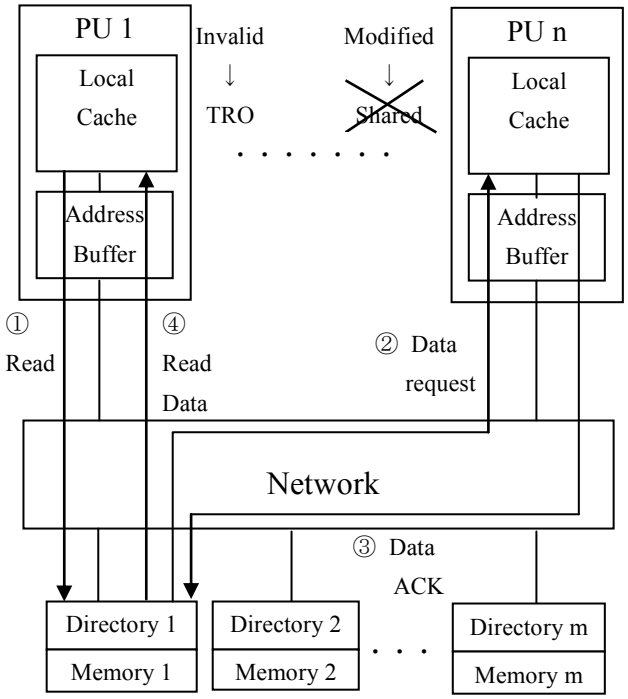


図 9　ハイブリッドプロトコルの手順(読み込み時)

Figure 9　Image of Hybrid protocol (Read)

4. 評価

性能を評価するに当たり MSI プロトコル,TRO プロトコル,そしてハイブリッドプロトコルの 3 つのシミュレータを SimpleScalar[6]を元に作成した.この 3 つのシミュレータのコア数やメモリ階層,ネットワークなどは同じ構成をしている.その構成を表 2 に示す.各プロセッサは Local-cache として L1cache を持ち,全プロセッサの Shared-cache として L2cache が存在する.また,プロセッサ間のネットワークとして 2Dmesh-network を用いている.

シミュレーションにおいて SPLASH2 ベンチマークから 5 つのアプリケーションを評価に使用した.使用したアプリケーション名とその入力データを表 3 に示す.シミュレーションから得られた結果を元に算出し,アプリケーション毎のハイブリッドプロトコルを基準とした実行時間比率を図 10~14 に示す.

実行時間の比率とは,該当プロトコルでの実行時間をハイブリッドプロトコルでの実行時間で割ったものである.従ってハイブリッドプロトコルでの値は常に 1 であり, 値が低いものほどプロトコルとしての性能がよいということがいえる.図からわかることとして,全てのアプリケーションに対してハイブリッドプロトコルは既存のプロトコルと

表 2　シミュレータの構成

Table 2　Configuration of simulator

Processor	
Number of cores	16 cores
issue width	single-issue
issue order	out-of-order
Data and Instruction L1cache (Local-cache)	
size	256 KByte
ways	8 ways
latency	3 cycle
L2cache (Shared cache)	
size	8 MByte
ways	16 ways
latency	45 cycle
Memory	
size	4 GByte
latency	256 cycle
Interconnection network (2D mesh)	
link latency	2 cycle
flit size	16 Bytes

表 3　評価に用いたベンチマーク

Table 3　Benchmarks used in our evaluation

Benchmark	Input data
LU(con.)	256 × 256 matrix
RADIX	2 M keys
FMM	8 K particles
OCEAN (con.)	258 × 258 ocean
OCEAN (n-con.)	258 × 258 ocean

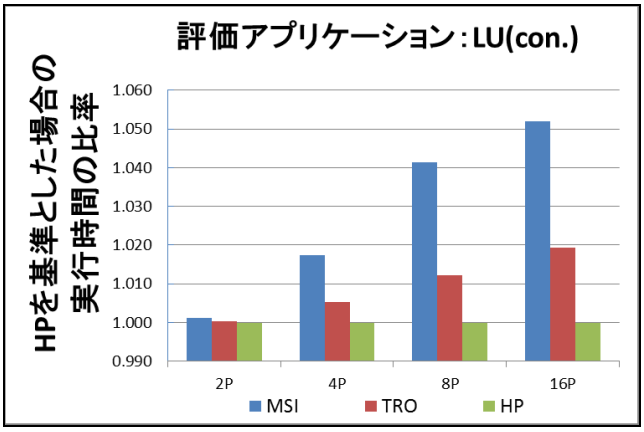


図 10　各プロトコルの実行時間の比率(LU)

Figure 10　The ratio of the execute time of each protocol (LU)

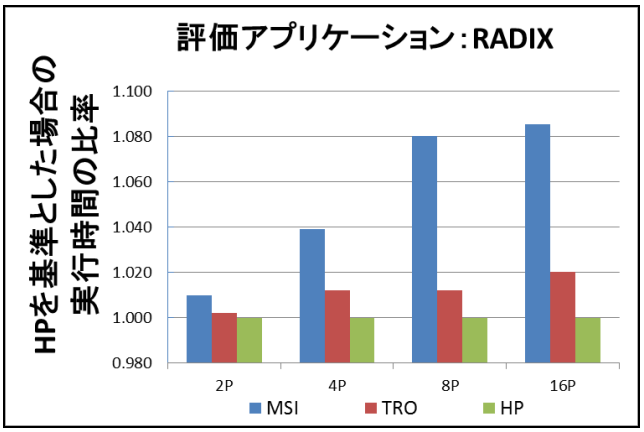


図 11　プロトコルの実行時間の比率(RADIX)

Figure 11　The ratio of the execute time of each protocol (RADIX)

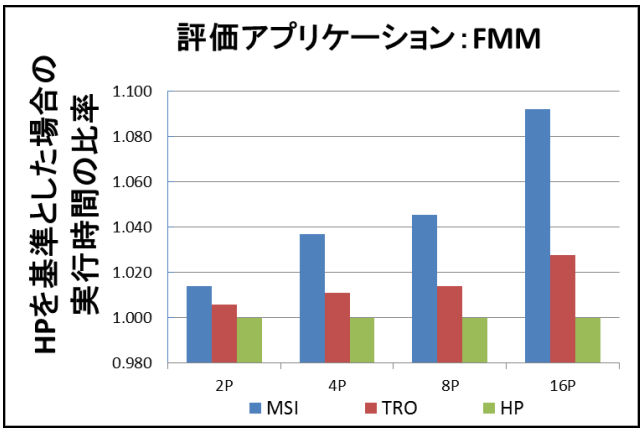


図 12　プロトコルの実行時間の比率(FMM)

Figure 12　The ratio of the execute time of each protocol (FMM)

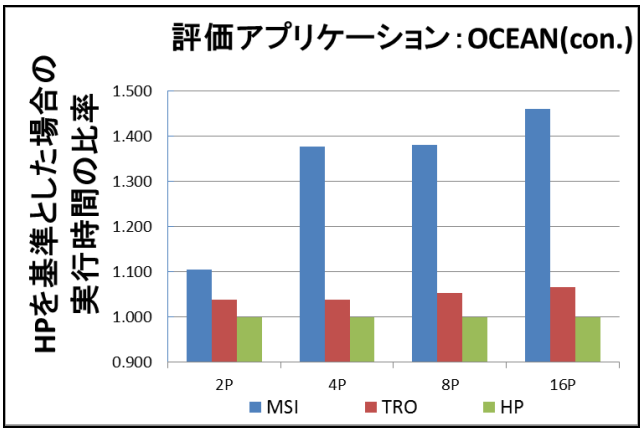


図 13　プロトコルの実行時間の比率(OCEAN (con.))

Figure 13　The ratio of the execute time of each protocol (OCEAN (con.))

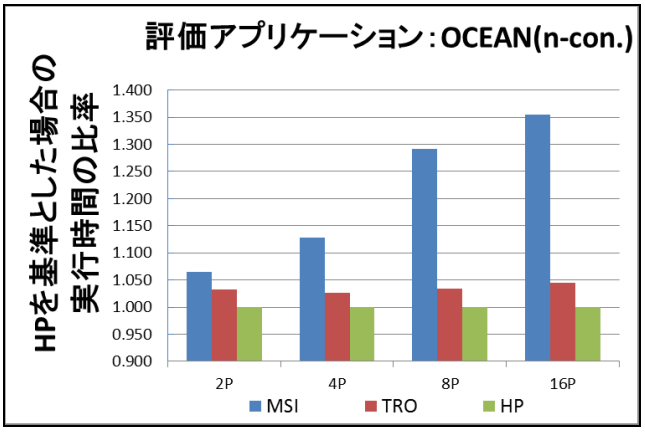


図 14　プロトコルの実行時間の比率(OCEAN (n-con.))
Figure 14　The ratio of the execute time of each protocol (OCEAN (n-con.))

比べその実行時間の短縮に成功している.また,その短縮量はプロセッサ数が増えるほど増加している.また OCEAN(con.),OCEAN(n-con.)に対しては特に大きな実行時間削減ができている.この２つのアプリケーションではプロセッサ間でのデータの受け渡しが頻繁に行われており,TRO プロトコルでの不要な Invalidation の数が多いことが原因だと考えられる． TRO プロトコルにおいて不要な Invalidation が起こった回数をカウントした.それを表 4 に示す.この表からわかるように OCEAN(con.)と OCEAN(n-con.)では不要な Invalidation が多く行われている.ハイブリッドプロトコルはこの不要な Invalidation を防ぐことで TRO に対しても実行時間の短縮に成功している.しかし，不要な Invalidation 回数に対して想定よりも実行時間削減量が少ない.これは Address Buffer を Local-cache と Network の間に追加し,信号を随時監視していることが原因であると考えられる.Address Buffer へのアクセス回数を表 5 に示す.このように Address Buffer へのアクセスは頻繁に行われている.Address Buffer の latency は L1cache と同じ 3cycle であるが,そこへのアクセスが多く行われることによってかなりの時間がかかってしまう.

5. まとめ

本論文では TRO プロトコルにおいて起こる不要なキャッシュ・ミスに注目し,Address Buffer の追加とディレクトリの拡張により MSI プロトコルと TRO プロトコルを融合させたハイブリッドプロトコルを提案した.

シミュレーションの結果,MSI プロトコルと TRO プロトコルの両方と比べて評価した全てのアプリケーションにおいて実行時間の削減に成功した.しかし,追加した装置である Address Buffer へのアクセス回数の多さが性能向上の妨げとなってしまった.よってこの Address Buffer へのアクセスを極力抑えるような配置や手順を考えることが今後の課題となる.また,今回 Address Buffer とディレクトリの TRO-bit を追加したが,このようなハードウェアの拡張をできるだけなくすような方法についても考えなくてはならない.

表 4　TRO プロトコルの不要な Invalidation 数
Table 4　The number of needless Invalidation on TRO protocol

	2P	4P	8P	16P
LU(con.)	2950	3353	3706	3719
RADIX	1346	2330	3860	5769
FMM	8438	9433	10825	11452
OCEAN (con.)	46565	50676	66100	79770
OCEAN (n-con.)	40453	45868	50235	59125

表 5　Address Buffer へのアクセス回数
Table 5　The number of access to Address Buffer

	2P	4P	8P	16P
LU(con.)	7077	8429	10953	11589
RADIX	4592	6350	11284	14864
FMM	16479	20901	28523	32588
OCEAN (con.)	103426	119643	129661	139715
OCEAN (n-con.)	94723	99528	115392	130537

参考文献

1) J.C.Villanueva, J.Flich , J.Duato, H.Eberle, N.Gura , W.Olesinski, ,“A performance evaluation of 2D-mesh, ring, and crossbar interconnects for chip multi-processors”IEEE Press 2009 pp.51-56
2) D. Park, S. Eachempati, R. Das, A. Mishra, Y. Xie et aI., "MIRA: A Multi-layered On-Chip Interconnect Router Architecture," in ISCA, 2008, pp. 251-261
3) J.L. Hennessy and D.A. Patterson, Computer Architecture: A,Quantitative Approach. San Francisco, CA, USA: Morgan Kaufmann,Publishers, Inc., 2003.
4)　S,Kaxiras,. and G,Keramidas., “SARC Coherence: Scaling Directory Cache Coherence in Performance and Power” IEEE Press,2010 pp.54-65
5) A.R. Lebeck and D.A. Wood, “Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors,” Proc. Int’l Symp. Computer Architecture (ISCA-22), IEEE Press, 1995, pp. 48-59.
6) T. Austin, E. Larson , and D. Ernst: SimpleScalar: An　Infrastructure for Computer SystemModeling, Computer, Vol.35, No.2, pp.59–67 (2002).