情報処理学会論文誌　コンピューティングシステム　Vol.5　No.3　86–100 (May 2012)

ディレクトリの余剰エントリを利用した CMP向け分散キャッシュの効率化

藤枝 直輝[1,a)]　吉瀬 謙二[1]

受付日 2011年10月7日, 採録日 2012年2月5日

概要：チップマルチプロセッサ（CMP）においてキャッシュの効率的な利用は重要な課題となってきている．キャッシュの容量を柔軟に利用するアプローチの 1 つとして，Distributed Cooperative Caching（DCC）のように，占有キャッシュをベースとしながらも，あるコアのキャッシュから追い出されたラインを別のコアのキャッシュへと移動可能とする構成が提案されている．DCC では，コア間のコヒーレンスを制御するために分散化されたディレクトリを用いる．本論文では，このディレクトリがキャッシュの利用効率を損なわないようにいくらかの余剰エントリを持っていることに注目し，これらのエントリが持つ情報を利用してキャッシュから追い出されたラインの移動を制御する方式である ASCEND（Adaptive Spill Control with extra ENtries of Directory）を提案する．評価の結果，8 コアで 2 つの並列アプリケーションを同時に動作させた場合，移動可能なラインをすべて移動する場合と比べて性能が平均で 1.5%，最大で 16.9%向上した．また，16 コアで 4 つの並列アプリケーションを同時に動作させた場合では，平均で 1.5%，最大で 14.0%の性能向上を達成し，アプリケーションの組合せごとに最適なラインの移動割合を知っていたと仮定した，DCC の理想的な性能に近い性能を得た．

キーワード：プロセッサアーキテクチャ，チップマルチプロセッサ，キャッシュパーティショニング，ディレクトリキャッシュ

A Method for Efficient Use of CMP Cooperative Caching with Extra Entries of Directory

Naoki Fujieda[1,a)]　Kenji Kise[1]

Received: October 7, 2011, Accepted: February 5, 2012

Abstract: How to use caches in chip multiprocessors (CMPs) efficiently has been an important problem. One of the approaches for efficient use of cache capacity is based on private caches and allows evicted lines from one core transfering (or spilling) to another core, such as Distributed Cooperative Caching (DCC). DCC has distributed directories to maintain coherence among cores. In this paper, we focus on a fact that the directories have some extra entries to prevent efficiency degradation. From this, we propose a new scheme for controlling replication named ASCEND (Adaptive Spill Control with extra ENtries of Directory). Our evaluation shows that the speedup over DCC with 100% spill is 1.5% on average and 16.9% at a maximum when running 2 parallel applications at the same time with 8 cores. ASCEND also improves performance by 1.5% on average and 14.0% at a maximum with 16-core settings, which is near the optimal performance DCC would show if it knew the best spilling rate for each application mix.

Keywords: processor architecture, chip multiprocessor, cache partitioning, directory cache

1. はじめに

プロセッサ内に複数のコアを搭載し，スレッドレベル並列性を利用してプロセッサのスループットを向上させ

[1] 東京工業大学大学院情報理工学研究科
Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[a)] fujieda@arch.cs.titech.ac.jp

© 2012 Information Processing Society of Japan

る CMP（Chip Multiprocessor）は，今や広く用いられている．キャッシュを搭載する CMP では，そのレイテンシを低く抑えつつ，ワークロードごと，あるいはコアごとに異なるキャッシュへの要求に対して，柔軟に対応することが求められる．特に，今後プロセッサ内に搭載されるコア数が増加すれば，それらすべてのコアが単一のワークロードを実行することはまれで，多種多様な要求を持つ複数のワークロードをチップ内で同時に実行するケースがますます増加していくと考えられる．

ラストレベルキャッシュ（LLC）に着目すると，それぞれのコアが独立した LLC を持つ占有キャッシュと LLC をすべてのコアで共有する共有キャッシュとに分類できる．占有キャッシュは共有キャッシュよりもレイテンシを低く抑えることはできるものの，容量の柔軟性に劣る．一方，共有キャッシュは容量を最大限に活用できるものの，リモートのキャッシュアクセスが増加するために，占有キャッシュと比べた平均のレイテンシが大きい．そのため，これらの利点を組み合わせたキャッシュ構成が提案されている [1], [2], [3], [4].

こうした両方の利点を活かした構成のうち，Cooperative Caching（CC）[1] などでとられているアプローチでは，占有キャッシュをベースとしつつ，あるコアを追い出されたキャッシュラインを別のコアへと移動させることで，共有キャッシュの利点を追加する．これにより，自コアのキャッシュに頻繁にアクセスしてレイテンシを低く保ったまま，他のコアの利用していないキャッシュ領域を借りることでキャッシュ容量への要求のばらつきに対応することを可能としている．

CC では，集中化された Coherence Engine と呼ばれる機構がディレクトリを集中管理することでコア間のコヒーレンシを制御している．しかし，集中化された構造へのアクセスの集中から，この構造はコア数のスケーラビリティを阻害する．これを改善するために提案されているのが Distributed Cooperative Caching（DCC）[5] である．分散化された Distributed Coherence Engine（DCE）のそれぞれが，アドレス空間のインタリーブされた一部分のコヒーレンシを制御する．

本論文では，この DCE が持つディレクトリが，キャッシュの利用効率を損なわないためにいくらかの余剰エントリを持っていることに注目する．我々はこれらの余剰エントリが持つ情報を捨てずに保持しつづけ，これを利用してキャッシュから追い出されたラインの移動を制御する方式である **ASCEND（Adaptive Spill Control with extra ENtries of Directory）**を提案する．メニーコアシミュレータを用いて 8 コア・16 コア環境で DCC との性能比較を行う．キャッシュに対する要求が異なる複数のアプリケーションを同時に実行させたときに，ASCEND がアプリケーションやコアごとの異なる要求を正しく検出し，

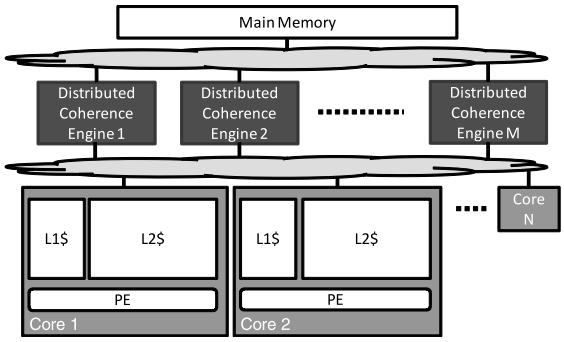


図 1　Distributed Cooperative Caching の構成
Fig. 1　Organization of Distributed Cooperative Caching.

キャッシュ領域を借りたいコアが，利用していないキャッシュ領域を持つコアに対して優先的にラインを移動させることで，キャッシュの利用効率を高め，性能を向上させることができることを示す．これにより，ASCEND の有効性を明らかにする．

本論文の構成を述べる．2 章で提案手法のベースアーキテクチャである DCC について解説し，DCC がディレクトリに余剰エントリを持つことと，その必要性について述べる．3 章で提案手法である ASCEND について述べ，4 章でその評価を行う．5 章で本研究に関連する研究について言及し，6 章で本論文をまとめる．

2. 背景

2.1 Distributed Cooperative Caching

図 1 に，ベースとなる Distributed Cooperative Caching（DCC）[5] の構成を示す．各コアは Processing Element（PE）および L1, L2 のキャッシュを持ち，L2 がラストレベルキャッシュとする．以後特に言及のない場合，単にキャッシュといった場合は L2 キャッシュを意味する．DCC は，分散化された Distributed Coherence Engine（DCE）を持つ．これらは何らかのインターコネクトによってメインメモリ，および各コアへと接続される．DCE はディレクトリキャッシュ（以下，本論文では単にディレクトリと記述する）である．それぞれの DCE はアドレスでインタリーブされたディレクトリの一部分を持っており，すなわち，アドレス空間の一部分のコヒーレンス制御を司っている．DCE の個数はコアの個数とは独立である．

図 2 を用いて，DCC の動作について解説する．コアの数と DCE の数はともに 8 とする．まず，キャッシュおよび DCE のエントリに空きのある場合を仮定する．動作例（a）は，コア 1 が自コアのキャッシュにミスし，必要とするラインを持っているコアも存在しない場合である．コア 1 は，必要とするラインの物理アドレスから一意に定められる DCE へと要求を送信する．ここでは対応する DCE は DCE2 であったとする．DCE2 は要求されたラインに対応するディレクトリを検索する．しかしながら，DCE2 は

© 2012 Information Processing Society of Japan

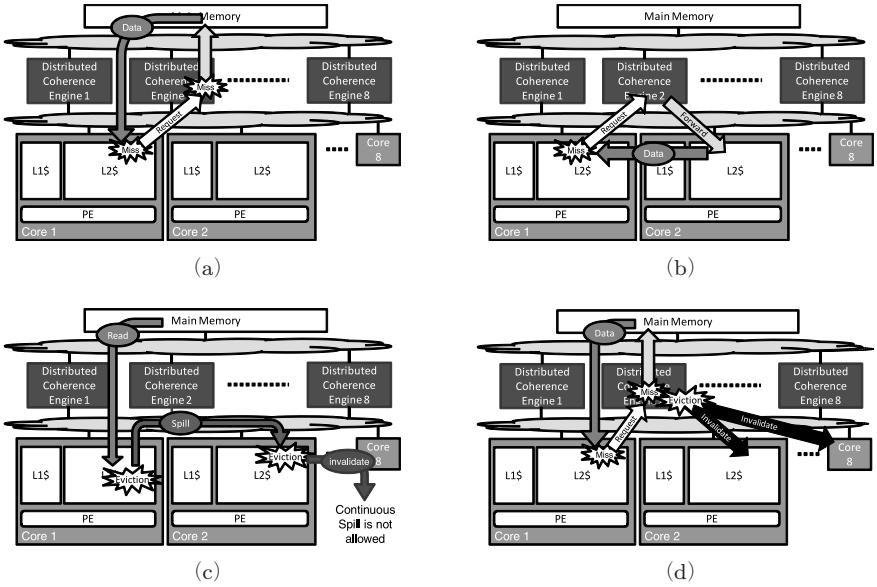


図 2　Distributed Cooperative Caching の動作例
Fig. 2　Working example of Distributed Cooperative Caching.

要求されたラインに対応する有効なエントリを持っていない．そのため，空きエントリを 1 つ割り当て，タグなどの情報を書き込み，メインメモリへと必要とするラインを要求する．その後，メインメモリが必要なライン（データ）をコア 1 へと供給する．

一方，要求されたラインがいずれかのコアに保持されていることもある．動作例（b）は，必要とするラインがコア 2 のキャッシュに存在する場合である．動作例（a）と同様に DCE2 はディレクトリを参照する．今度は有効なエントリが存在しているので，該当するラインを持つコアを調べる．これにより，コア 2 のキャッシュ内に所望のデータが存在することが分かる．したがって，コア 2 へと要求を転送するとともに，コア 1 が当該ラインの共有者となるようにディレクトリを修正する．コア 2 は転送された要求を受信し，キャッシュ間転送によりコア 2 の持つラインをコア 1 へと転送する．この場合，メインメモリへのアクセスが発生せず，コア 1 のアクセスレイテンシが削減される．

次に，コア 1 が自コアのキャッシュにミスし，ラインを格納したいセットに空きのラインが存在しない場合の動作例を（c）に示す．必要とするラインを格納するために，セットのうちある戦略（ここでは LRU（Least Recently Used）とする）に従ってラインが追い出される．Cooperative Caching の特徴は，このとき，追い出されたラインを別のコアへと移動できることである．このラインの移動は Spill とも呼ばれる．ラインの移動を行う場合は，コアがその移動先を定める．キャッシュ間転送を行うと同時に，DCE にラインの移動を通知する．もしこのとき，移動先のコアがラインを格納するセットに空きのラインが存在しなければ，ラインの移動が連鎖しないように，移動されたラインを受け入れるために追い出されるラインは単に無効化またはライトバックされる．動作例（c）ではコア 2 が移動先として選択され，コア 2 がラインを格納するセットに空きのラインが存在しないとする．コア 1 は自らの持つデータをコア 2 へと転送し，そのラインを無効化する（矢印 Spill で示される）．コア 2 はコア 1 からのデータの受け入れ先ラインを無効化またはライトバックして，転送されたラインを格納する（矢印 Invalidate）．コア 1 が要求したラインは，コア 1 が無効化したラインへと格納される（矢印 Read）．

DCC ではある 1 つのラインの連続した移動は 1 回までに限られる．すなわち，ラインがあるコアを追い出されて別のコアに移動した後，使用されないまま再び追い出されたら，そのラインはチップから追い出される．これを実現するために，DCC の各ラインは最後にラインを使用してからラインを移動したかを表す追加のビットを持っている．このビットはラインの移動が行われると 1 にセットされ，PE によって使用されると 0 にクリアされる．もしラインが追い出されるときにこのビットが 1 であれば移動は行われず，そのラインは単に無効化またはライトバックされる．また，他のコアと共有しているラインもラインの移動を行わない．なぜなら，このようなラインは別のコアに移動するまでもなくチップ内の他のコアが持っているためである．以降では，他のコアと共有していないラインで，最後に PE によって使われたあとまだ移動をしていない（ラインを移動したかを表すビットが 0 である）ラインのことを，移動可能なラインと呼ぶ．

2.2　余剰ディレクトリエントリの必要性

前節で述べた 3 つの動作例に続いて，動作例（d）に DCE の対応するセットに空きのエントリが存在しない場合を示

© 2012 Information Processing Society of Japan

す．図 1 の動作例（a）と同様に，コア 1 は DCE2 に要求を送信する．DCE2 は対応するディレクトリのエントリを参照しようとする．しかし，DCE2 は要求されたラインに対応する有効なディレクトリエントリを持っておらず，しかも新たなエントリを格納するための空きエントリも持っていない．この場合，対応するセットの任意のエントリを選択して，エントリに対応したラインを持つすべてのコアに対して対応するラインの無効化，あるいはライトバックの要求を行い，当該のエントリを無効化する．ここでは対応するラインがコア 2 とコア 8 によって共有され，いずれのコアも該当するラインへの書き込みはしていないとすると，DCE2 はコア 2 とコア 8 へと無効化を送信する．DCE2 が必要とする新たなエントリは，この無効化されたエントリへと格納される．なお，以降では特に言及のない場合，その際のエントリの選び方は LRU による．

このようなディレクトリのエントリ不足によるキャッシュラインの無効化は本来起きていなかったものであるから，頻繁に無効化が起きることはキャッシュの利用効率や性能に悪影響を及ぼす．DCC ではディレクトリのエントリ数や連想度はキャッシュのライン数や連想度とは独立して設定できるため，以下にあげる 2 つの方法のいずれかを用いて，こうした無効化を回避する．1 つは，キャッシュの総ライン数に対して，ディレクトリの総エントリ数をある程度大きく設定することである．これは文献 [5] でも採用されている方法であり，文献 [5] ではディレクトリの総エントリ数をキャッシュの総ライン数の 2 倍に設定している．もう 1 つは，ディレクトリの総エントリ数をキャッシュの総ライン数と同じ値に設定し，かつディレクトリの連想度を，キャッシュの連想度 × コア数とすることである．後者の方法では本質的にラインの無効化を防ぐことができるものの，コアの数に比例した大きな連想度を設定する必要があり，スケーラビリティを損なうことになる [5]．そのため前者の方法で無効化を回避することとする．

ディレクトリのエントリ数を大きく設定する際，あまりにも大きな値を設定することは，資源の増加に見合った性能改善を見込めないばかりか，ディレクトリへのアクセスレイテンシを増加させることにつながる可能性さえある．そのため，これらのトレードオフを適切にとる，すなわち適切なディレクトリのエントリ数を選択する必要がある．

キャッシュの総ライン数に対してディレクトリの総エントリ数をどの程度とればよいかを確認するため，予備評価を行った．4.1 節で後述する 8 コア・2 アプリケーション，DCE100 の構成において，DCE のディレクトリエントリ数のみを変化させて，性能がどのように変化するかを測定する．

予備評価の結果を**図 3** に示す．横軸はキャッシュの総ライン数に対するディレクトリの総エントリ数の割合である．ここではキャッシュの総ライン数を 32,768 ラインと設定しているため，ディレクトリの総エントリ数は横軸 75%の場合で 24,576 エントリ，175%の場合で 57,344 エントリである．縦軸には横軸 175%の場合と比較した相対性能を示す．横軸 175%の場合，ディレクトリのエントリ不足に起因するキャッシュラインの無効化はほとんど発生しておらず，ここでの性能はディレクトリのエントリ数を無限大としたときとほぼ等しい．HMean は，全 15 種類のアプリケーションの組合せに対する性能の調和平均を示す．HH は，すべての組合せのうち横軸 138%以上での性能低下が最も大きかった姫野ベンチマーク 2 つの組合せにおける性能を示す．

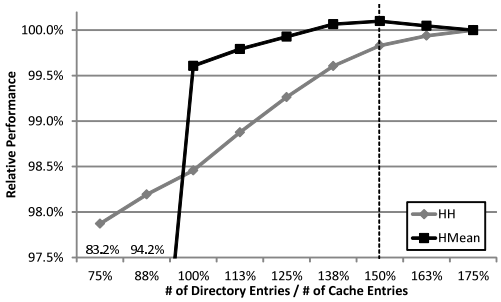


図 3　ディレクトリの容量が性能に与える影響
Fig. 3　Influence of directory size upon performance.

ディレクトリのエントリ数がキャッシュのライン数よりも少ない横軸 75%や 88%の領域では，データの大半が 1 つのコアでのみ使用されることが多い場合，すなわち，ディレクトリのエントリとキャッシュラインとがほぼ 1 対 1 で対応する場合，ディレクトリのエントリ数の減少がほぼそのまま有効なキャッシュラインの減少として現れる．そのため，アプリケーションの組合せによっては性能を大きく損ねる場合がある．また，エントリ数をライン数と同じかやや多くとった場合でも，DCE のインタリーブやセットの割り振りによってセット間で利用されるエントリ数に偏りが生じる．これによりやはり多くの無効化を発生させ，キャッシュの利用率を低下させている．さらにエントリ数を増加させ，ディレクトリのエントリ数をライン数の 1.5 倍（150%）としたときに，初めてすべてのアプリケーションで性能への悪影響が 0.2%を下回る結果となった．

ディレクトリの 1 エントリの大きさは，物理アドレスのビット幅や搭載するコア数に依存するが，予備評価のセッティングでは 50 ビット前後である．一方，キャッシュの 1 ラインはデータだけで 512 ビットに達するので，ディレクトリの 1 エントリとの差は 10 倍以上になる．したがって，ライン数の 1.5 倍のエントリをディレクトリに持たせたとしても，追加の記憶容量はキャッシュの容量に換算すれば 5%以下であり，それほど大きくない．以上のことから，DCE 内のディレクトリにキャッシュライン数の 1.5 倍程度のエントリを持たせることは妥当な選択である．以降では，特に言及のない場合，ディレクトリの総エントリ数

© 2012 Information Processing Society of Japan

はキャッシュライン数の 1.5 倍とする．しかしながら，こうした際に少なくともキャッシュライン数の半分（0.5 倍）のディレクトリエントリは未使用のままとなっている．この領域を有効に利用することができないだろうかという考えが，本論文のモチベーションである．

3. ASCEND の提案

3.1 ASCEND の基本方針

本章では，前節で議論した DCE の余剰エントリを利用して，キャッシュラインの移動を制御する方式である ASCEND（Adaptive Spill Control with extra ENtries of Directory）を提案する．ASCEND では，無効化されたディレクトリエントリへの参照回数などをカウントし，キャッシュラインの移動が良い効果を生むと考えられるコアに移動の権利を与える．また，ラインの移動が悪影響を及ぼすと考えられるコアはラインの移動先として選ばれにくくする．これらを組み合わせることで，キャッシュの利用効率を高め，プロセッサ性能の向上を狙う．

キャッシュラインが無効化されてチップから追い出されるとき，これらのラインに対応するディレクトリのエントリもまた無効化される．このとき，エントリの状態は無効とするものの，そのエントリが持つタグの情報は消去せず保持しつづけるものとする．そのタグは後々新たなエントリが必要となったときに置き換えられる．そうすると，無効化されたディレクトリのエントリが持つタグの集合は，最近チップから追い出されたキャッシュラインに対応するタグの集合となる．そのため，これらのエントリへの再度の参照を検出・計測すると，どのコアがチップから追い出されたデータをすぐに再利用するのかを判定できる．すなわち，こうした再参照の多いコアを検出し，そのコアが追い出したラインを優先的に他のコアへと移動させることにより，プロセッサの性能を向上させることができると考えられる．これが ASCEND の基本方針である．

3.2 ASCEND の構成

ASCEND における DCE の構成を**図 4** に示す．それぞれの DCE に対して，ディレクトリのアレイとコヒーレンス制御のためのコントローラに加え，Spiller Selector と Receiver Controller と呼ばれる 2 つの機構を追加する．Spiller Selector は，ラインの移動が良い効果を生むと考えられるコアに移動の権利を与えるための機構である．Receiver Selector は，ラインの移動が悪影響を及ぼすと考えられるコアがラインの移動先として選ばれにくくするための機構である．これらはコア数に比例したいくつかのカウンタまたはレジスタを持つ．

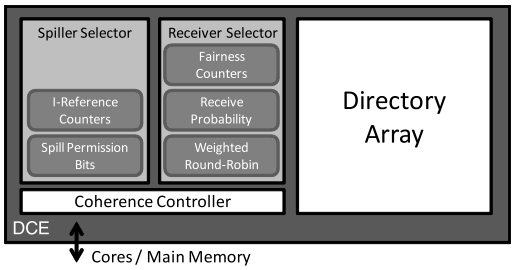


図 4　ASCEND における DCE の構成
Fig. 4　Organization of DCE with ASCEND.

DCC では，追い出されるラインが移動可能なラインであれば，コアがその移動先を決定し，DCE に移動を行うことが通知される．それと異なり，ASCEND では移動可能なラインを実際に移動するかどうか，また移動する場合のラインの移動先は DCE が判定する．追い出されるラインが移動可能であれば，DCE はコアに移動先を通知して，コアはキャッシュ間転送によりラインを移動する．もしそうでなければ，DCE は移動不許可であることをコアに通知する．この場合，追い出されるラインは無効化またはライトバックされる．

ASCEND ではまた，無効化されたエントリの扱い方に軽微な変更と，ディレクトリに 1 つ追加の状態が必要である．これについては 3.3 節で述べる．また，ASCEND によって追加された機構の詳細については，Spiller Selector は 3.4 節，Receiver Selector は 3.5 節で述べる．

3.3 無効化されたエントリの扱いと I 参照

ASCEND では先に述べたとおり，キャッシュラインがチップから追い出されて対応するディレクトリのエントリが無効化されても，そのエントリが持つタグの情報はそのまま保持しておく．そして，ディレクトリが参照されるときには，有効なエントリだけではなく，無効化されたエントリに対してもタグの比較を行う．もしタグが一致するエントリが存在すれば，そのエントリが選択される．そうでなければ，まず無効なエントリが存在するかどうかを調べ，無効なエントリがあれば，その中から任意のエントリが選ばれる．以降では特に言及のない場合，その際のエントリの選び方は LRU による，すなわち最も早く無効化されたエントリを選択することとする．タグが一致するエントリも無効なエントリも存在しない場合は，図 2 (d) で説明した，ディレクトリ不足によるキャッシュラインの無効化が行われる．

こうしてエントリが選択されたときに ASCEND で注目するのは，無効化されたエントリが選択され，かつタグが要求するものと一致した場合である．以降ではこのような無効化エントリへの再度の参照を **I 参照（I-Reference）**と呼ぶこととする．I 参照を多く発生させるコアは，チップから追い出されたデータをすぐに再利用している可能性が高い．したがって，I 参照の回数は 3.4 節で述べる移動を行うコアの選択の際に利用される．

ASCEND ではまた，あるアプリケーションが高速にな

© 2012 Information Processing Society of Japan

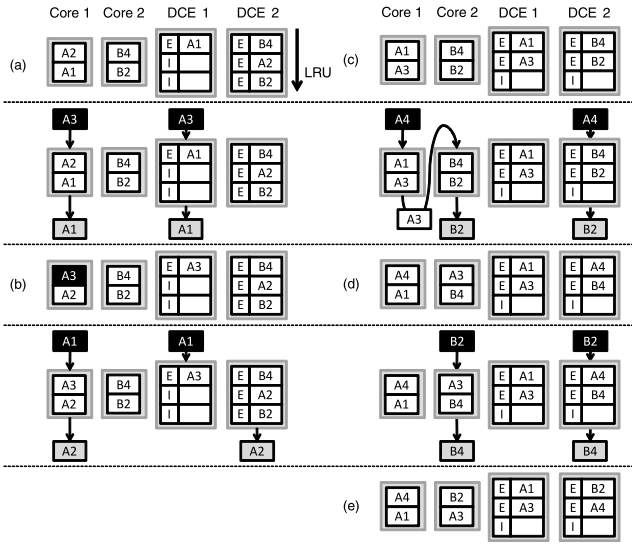


図 5　無効化されたタグを残さない場合の遷移
Fig. 5　Transition of tags when invalidated tags are removed.

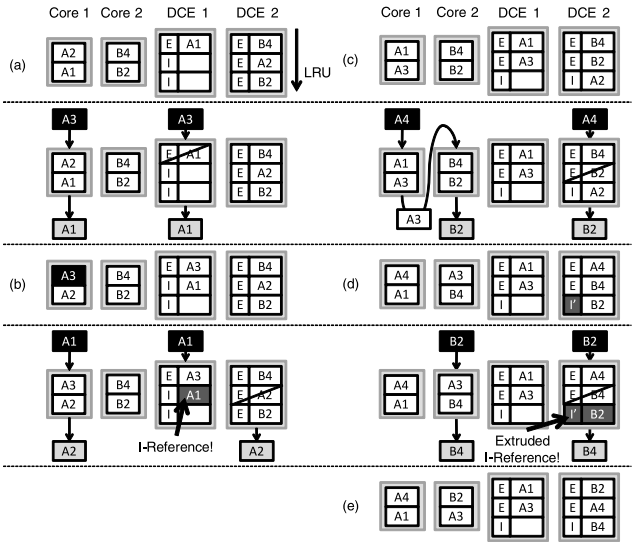


図 6　無効化されたタグを保持しつづける場合の遷移と，I 参照の例
Fig. 6　Transition of tags when invalidated tags are preserved, and example of I-References.

る一方で，別のアプリケーションの性能がきわめて悪化するといった，不公平な性能向上を避けるために，ラインの移動が悪影響を及ぼすと考えられるコアの検出を行う．この悪影響の予測のために，ラインの移動を受け入れたことにより直接的にチップから追い出されるライン，すなわち図 1 の動作例（c）において矢印 Invalidate で表した追い出しに注目する．もしもこのラインが移動可能なラインであれば，これによってチップからラインが追い出されたとき，該当するディレクトリのエントリを無効化すると同時に，そのことが分かるようにマーキングしておく．これは，ディレクトリの状態を 1 つ増やすことにより実現できる．そして，これらのラインへの I 参照を**押し出された I 参照（Extruded I-Reference）**として他の I 参照と区別する．押し出された I 参照を多く発生させるコアは，ラインの移動を受け入れたことにより性能に悪影響を受けているおそれが大きい．なぜならば，押し出された I 参照を引き起こしたエントリに対応するラインは，他のコアからのラインの移動を受け入れなかったとしたら，まだ自らのキャッシュ内に残っていた可能性があるためである．したがって，押し出された I 参照の回数は 3.5 節で述べる移動先コアの選択の際に利用される．

図 5 と**図 6** に，ここまでに述べた無効化エントリのタグ保持と，I 参照検出の例を示す．簡単のために，各コアは 2 つのキャッシュライン，各 DCE は 3 つのディレクトリエントリを持つものとし，各行が 1 つのラインまたはエントリに対応する．各行は最近使われたものが上にくるように並べ替えられている．英大文字 1 字と数字 1 字の組合せでタグを表現し，タグの数字が奇数のラインは DCE1 により，偶数のラインは DCE2 によりコヒーレンスを管理されるものとする．DCE のエントリの左側のフィールドはディレクトリの状態であり，E（Exclusive）が 1 つのコアにのみ所持されている状態，I（Invalid）と I' が無効化された状態を表す．コヒーレンシプロトコルには MOESI を用いるため，ディレクトリはこのほかにも M，O，S の状態を持つが，図には該当するエントリは存在しない．コアからの要求を受けたラインは白黒を反転して示す．なお，以後の説明に必要のないキャッシュの状態などのフィールドは図からは省いている．

初期状態を（a）とする．コア 1 はライン A1，A2 を持ち，コア 2 はライン B2，B4 を持つ．A1 は DCE1 により管理され，A2，B2，B4 は DCE2 により管理されている．また，（a）の段階ではいずれのコアも他のコアへのライン移動を許可されていないとする．

初めに，図 5 に示す，無効化されたタグの情報を保持しない場合の遷移を示す．状態（a）から，コア 1 が A3 への要求を発したとする．A3 がメインメモリから読み出されるとともに，A1 がコア 1 から追い出され無効化される．以上により（b）の状態になる．次に，コア 1 が A1 への要求を発すると，今度は A1 が読み出され，A2 が無効化される．ここまでが（c）の状態である．続いて，コア 1 が A4 への要求を発する．A4 がメインメモリから読み出され，A3 がコア 1 から追い出される．ただし今度は，コア 1 は追い出されたラインの移動を許可されているとしよう．A3 はコア 2 へと移動され，その影響でコア 2 の LRU である B2 は無効化される．以上により状態は（d）に示すとおりとなる．最後に，コア 2 が B2 を要求する．B2 がメインメモリから読み出され，B4 が無効化される．その状態は（e）に示すとおりとなる．

今度は，図 6 に示す，無効化されたタグを保持しつづける場合の遷移を示す．状態（a）と状態（b）の間で A1 がコアから追い出され，無効化される．このとき，無効化さ

© 2012 Information Processing Society of Japan

れたタグの情報はそのまま保持しつづける．すると，状態（b）ではDCE1はタグA1に関する「無効化された」という情報を持っていることとなる．ここから再びコア1からA1への要求が発生し，DCE1を参照すると，無効化されたA1に関するエントリがヒットする．これがI参照である．ASCENDでは，これをコア1によるI参照として記録する．

次に，状態（c）と状態（d）との間でコア2はコア1からのラインA3を受け入れる．これにより，B2がコア2から追い出されて無効化される．このときB2は，ラインの移動を受け入れたことにより直接無効化されたラインであり，なおかつコア2によって利用されていたラインである．したがって，DCE2ではそのことを状態I'を使ってマーキングしておく．この状態から再びコア2がB2を要求すると，DCE2で無効化かつマーキングされたB2に関するエントリがヒットする．これが押し出されたI参照である．ASCENDでは，これをコア2による押し出されたI参照として記録する．（a）から（e）までの各状態を図5と比較すると分かるとおり，ASCENDではディレクトリ中の無効化された部分だけに変更を加えており，有効なエントリには何の影響も与えない．

3.4 Spiller Selector

移動を行うコアを選択するには，I参照の回数を用いる．先に述べたとおり，I参照の回数が多いコアはチップから追い出されたデータをすぐに再利用している可能性が高く，ラインの移動を許可するのに適している．逆に，I参照の回数が少ないコアはワーキングセットの大きさがキャッシュに収まっている小さいアプリケーションを動作させているか，あるいはストリーム処理のように1度使ったデータをほとんど再利用しないアプリケーションを動作させていると考えられる．こうしたコアにラインの移動を許可しても性能向上の可能性は乏しく，逆に他のコアへの悪影響となる可能性すらある．そのため，このようなコアにはラインの移動を許可しない．

この方針に基づき定式化を行う．各DCEが個別に，自らの持つディレクトリで発生したI参照の回数をカウントしていく．ある閾値を定め，一定期間内のI参照の回数がその閾値以上であれば，次の期間ではラインの移動を許可する．そうでなければ，次の期間ではラインの移動を行わない．閾値は，各コアのI参照回数を $I_1, I_2, \ldots, I_N$ とすると，定数 k を用いて，$(\sum_{i=1}^{N} I_i)/N \times k + 1$ と定める．これはI参照回数の平均値の定数倍に1を加えた値である．ただし，N/k が整数となるように k を定めるとする．したがって，コア a がラインの移動を許可する条件は $I_a \geq (\sum_{i=1}^{N} I_i)/N \times k + 1$ である．これを変形すると，

$$(N/k - 1) \times I_a - \left((\sum_{i=1}^{N} I_i) - I_a\right) \geq N/k \qquad (1)$$

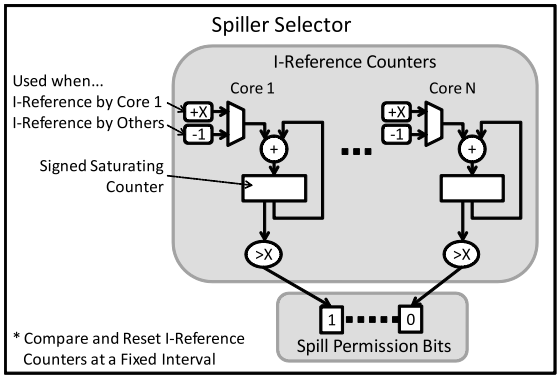


図 7　移動を行うコアの選択機構
Fig. 7　Organization of Spiller Selector.

式(1)で $X = N/k - 1$，$I_o = (\sum_{i=1}^{N} I_i) - I_a$ とおくと，$X \times I_a - I_o \geq X + 1$ となり，両辺が整数であることを利用して，以下の式を得る．

$$X \times I_a - I_o > X \qquad (2)$$

式(2)もまたコア a がラインの移動を許可する条件である．また，式(2)より自コアのI参照回数 I_a と他コアのI参照回数 I_o，それと定数 X とを利用することで，条件の判定が可能である．

たとえば，コア数 $N = 8$，$k = 4/5$ とおき，I参照回数の総和を1,000とすれば，閾値は $1000/8 \times (4/5) + 1 = 101$ となる．また，式(2)にこれらをあてはめると，$X = N/k - 1 = 9$ より，コア a がラインの移動を許可される条件は，$9I_a - I_o > 9$ となる．先に求めた閾値101を I_a に代入すれば，$I_o = 1000 - 101 = 899$ であるから，$9I_a - I_o = 909 - 899 = 10 > 9$ となり，この場合は移動が許可される条件を満たしている．

以上をもとに，**図 7** にSpiller Selectorの構成を示す．各DCEに対して，コアの数と同数のI参照カウンタ（I-Reference Counter），および移動の許可を表す1ビットのフラグ（Spill Permission Bit）を追加する．I参照が発生するごとに，それを引き起こしたコアのI参照カウンタは定数 X だけ加算され，そうでないコアのI参照カウンタは1つデクリメントされる．I参照カウンタのチェックは一定期間ごとに行われる．I参照カウンタの値を X と比較し，X よりも大きければ移動許可のビットを1とする．そうでなければ移動許可のビットを0とする．

コアがDCEにラインの移動要求を発すると，DCEは要求を発したコアに対応する移動許可のビットを確認する．ビットが1であればラインの移動を許可し，移動先を後述するReceiver Selectorで決定する．もしここでビットが0であったらラインの移動は許可されず，この要求はコアからの単なるライン無効化・ライトバックの通知と同じように扱う．いずれの場合でも，DCEは適切にディレクトリを更新する．

© 2012 Information Processing Society of Japan

3.5 Receiver Selector

移動先コアの選択の割合は，主に押し出されたI参照の回数を利用して決定する．押し出されたI参照の回数の多いコアは，他のコアのラインを受け入れることによって性能に悪影響を受けている可能性が高いが，その度合いは元々のオフチップアクセスの回数がどれだけあったかに依存する．たとえば，他のコアのラインを受け入れたことによるミスの増加が100回であっても，元々のミスが100回なのか，あるいは10,000回なのかでは，性能に与える影響は大きく異なる．そのため，判定に用いる閾値はオフチップアクセスの回数に対する割合で定めることとする．

また，コアがラインの移動を許可されている場合，他コアのラインを受け入れることによる性能への悪影響が，自コアのラインを移動することによる性能の向上によってカバーされていることがある．このことを考慮するために，他のコアへと移動したラインがヒットしたことも検出・計測する．DCCでは1つのラインが2回以上別のコアへと繰り返し移動されることがないよう，他のコアへの移動を行ったかどうかの情報は，キャッシュやディレクトリの状態の中にすでに含まれている．そのため，この検出は押し出されたI参照の場合とは違い，追加のディレクトリの状態を必要としない．また，押し出されたI参照が自コアのL2ヒットをミスに変えるものであることに対し，他のコアへと移動したラインのヒットは，ミスを他コアのL2ヒットへと変えるものである．このレイテンシ変化の差分の違いを考慮して，他のコアへと移動したラインのヒットは，押し出されたI参照よりも判定に与える影響が少し小さくなるようにする．

以上の方針に基づいて定式化を行う．あるコアが性能に悪影響を受けていると判定する条件は，各DCEでコアごとに計測した，自らの持つディレクトリで発生した押し出されたI参照の回数（Eと表す），他のコアへと移動したラインのヒットの回数（Hと表す），オフチップアクセスの回数（Oと表す），そして定数lと，閾値のオフチップアクセスの回数に対する割合tとを用いて，

$$E - H \times l > O \times t \tag{3}$$

と表す．この式を変形し，$X = l/t$，$Y = 1/t$とおくことにより，次式を得る．

$$-X \times H - O + Y \times E > 0 \tag{4}$$

ただし，l，tはX，Yがともに整数となるように定めるものとする．すなわち，式(4)よりE，H，Oの各回数，および定数X，Yを用いることで，整数演算のみで条件の判定が可能である．

この条件が真である，すなわち性能に悪影響を受けていると判定されたコアは，ラインの移動先コアとして選択される確率を徐々に小さくしていき，そうでないコアの選択される確率を，逆に徐々に大きくしていく．

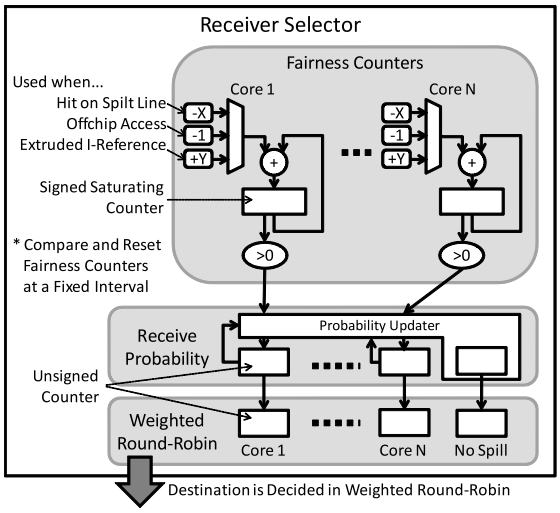


図 8　移動先コアの選択機構
Fig. 8　Organization of Receiver Selector.

以上をもとに，**図 8** にReceiver Selectorの構成を示す．各DCEに対して，コアの数と同数の公平度カウンタ（Fairness Counter），コアの数よりも1つ多い移動先の選択確率カウンタ（Receive Probability）を持つ．また，実際に個々の移動先を決定するための，重み付きラウンドロビン方式の移動先セレクタ（Weighted Round-Robin）が各DCEにつき1つ存在する．公平度カウンタは，他のコアへと移動したラインのヒットのたびに定数Xだけ減算され，オフチップアクセスのたびに1つデクリメントされ，押し出されたI参照のたびに定数Yだけ加算される．公平度カウンタのチェックは一定期間ごとに行われる．公平度カウンタの値が0より大きいかをチェックし，0より大きければそのコアは性能に悪影響を受けていると判定して，選択確率を後述する方法に従って減少させる．もし0以下ならば性能への悪影響はないと判定して，選択確率を増加させる．

選択確率カウンタはコアごとに割り振られる．また，更新機構（Probability Updater）はコアに割り振れる残りのカウンタの値を保持する．このカウンタは，移動先を決定するときには，移動をしない決定をする確率（No Spill）としても用いられる．これらのカウンタの総和は一定である．すなわち，あるコアにカウンタの値を割り当てれば，残りのカウンタの値はその分だけ減少する．逆に，あるコアがカウンタの値を解放すれば，残りのカウンタはそれだけ増加する．

選択確率カウンタの更新の際に考慮する点は2つである．1つは，現在のカウンタの値が大きいものほど大きく変動させることである．カウンタの値が変化したときに与える影響は，現在のカウンタの値が大きくなるほど相対的に小さくなる．そのため，変化に対する反応が鈍化してしまう．それを補正するために大きく変動させる．もう1つは，移動許可を持つコアと持たないコアがともにカウンタ

© 2012 Information Processing Society of Japan

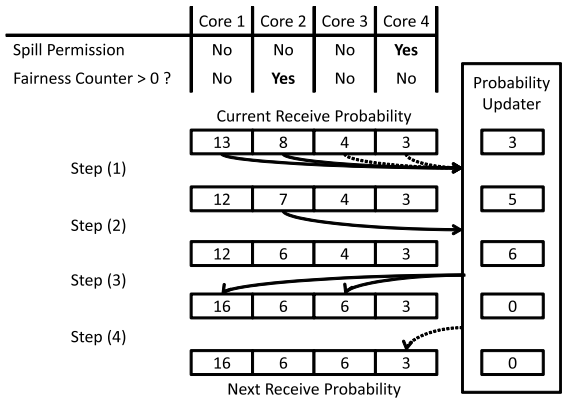


図 9　選択確率カウンタの更新の例
Fig. 9　Example of update process of receive probability.

の増加を求めた場合，移動許可を持たないコアを優先することである．これにより，選択確率カウンタの更新方法を以下のように定める．ただし，手順が早いものほど優先され，同一手順におけるコア間の優先順位はランダムに決定する．

(1) すべてのコアは，現在のカウンタの値 /8 だけカウンタの値を解放する．
(2) 公平度カウンタの値が 0 より大きいすべてのコアは，現在のカウンタの値 /8 + 1 だけカウンタを解放する．
(3) 公平度カウンタが 0 以下で，移動許可を持たないすべてのコアは，現在のカウンタの値 /4 + 1 だけカウンタを割り当てる．
(4) 公平度カウンタが 0 以下で，移動許可を持つすべてのコアは，現在のカウンタの値 /4 + 1 だけカウンタを割り当てる．

除算の結果はすべて剰余を切り捨てる．カウンタの増減値を求める際に 4 や 8 で除算を行うが，これは除算をビットシフトのみの小さなハードウェアで実現するためである．また，初期状態では，すべてのコアの選択確率カウンタの値は 0，残りのカウンタの値は選択確率カウンタのビット幅で表せる最大の値とする．

図 9 に選択確率カウンタの更新の例を示す．ここではコアの数を 4，カウンタのビット幅を 5 ビットとする．そのため，カウンタの総和はつねに 31 となる．更新前のカウンタの値はコア 1 から順に 13，8，4，3 とする．残りのカウンタの値は 3 である．また，コア 4 だけがコアの移動を許可されているとし，コア 2 だけが公平度カウンタの値が 0 より大きかったとする．まず手順 (1) で各コアがカウンタの値をそれぞれ 1，1，0，0 だけ解放する．手順 (2) で公平度カウンタが 0 より大きいコア 2 がカウンタの値を 1 解放する．これにより残りのカウンタの値は 6 になる．手順 (3) で移動許可を持たないコア 1 とコア 3 にそれぞれ 4，2 を割り当てる．手順 (4) では移動許可を持つコア 4 に 1 を割り当てようとするが，すでに残りのカウンタが 0 であるので割当ては発生しない．以上により，更新後のカウンタの値はコア 1 から順に 16，6，6，3 となる．更新前のカウンタの値が大きかったコア 1 の変動が大きく，またコア 1，3，4 がカウンタの増加を求めたときに，移動許可を持たないコア 1 とコア 3 の選択確率が優先的に増加している．よって，この更新方法で先に述べた考慮すべき点をともに解決している．

選択確率カウンタの値をもとに，個々のラインの移動先は重み付きラウンドロビン方式 [6] の移動先セレクタによって決定する．重み付きラウンドロビン方式では，要素が選択される割合をもとに，ある期間の間に要素が選択される回数を割り出し，その回数を守って選択を行う．たとえば，要素 A，B，C が 5:3:2 の選択割合を持つとすれば，5 + 3 + 2 = 10 回の選択を 1 つの期間とし，その間に A が 5 回，B が 3 回，C が 2 回選択されるように要素を選択していく．

重み付きラウンドロビン方式の移動先セレクタは，選択確率カウンタの値で初期化される内部カウンタを持つ．あるラインが 4.4 節で述べた Spiller Selector によって移動を許可されたら，一番大きな内部カウンタを持つものを 1 つ選択し，それに対応したコアを移動先として決定する．あるいは，No Spill に対応するカウンタが選択されたら，移動を行わない決定をする．ただし，ラインの移動元にあたるコアは移動先の選択からは除外される．そのうえで，選択された内部カウンタの値をデクリメントする．もしも内部カウンタの値がすべてゼロになったならば，内部カウンタの値を選択確率カウンタの値でリセットする．移動先（あるいは移動不許可）が決定したら，DCE がコアにそのことを通知するとともに，該当するディレクトリを更新する．

4. 評価

4.1 評価環境

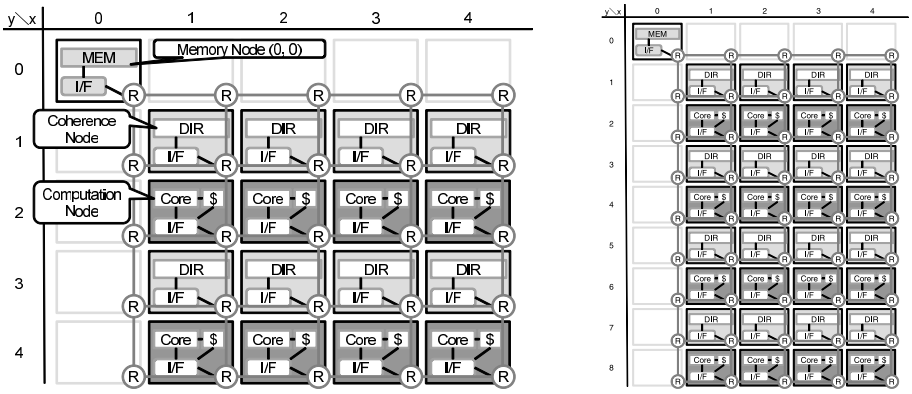
本論文における性能評価には，メニーコアプロセッサシミュレータ SimMc [7] に対し，キャッシュ，ディレクトリその他の拡張を施したものを用いる．**表 1** に，主要な評価パラメータの一覧を示す．

図 10 (a) は，計算コア 8 個の場合における評価対象のメニーコアの構成である．全体は 5 × 5 の 25 ノードから構成され，各ノードはルータによってメッシュ状に接続されている．ノードのうち座標 (0, 0) のノードが主記憶にアクセス可能なメモリノード，(1, 1)〜(4, 1)，(1, 3)〜(4, 3) の計 8 ノードは DCE に相当するコヒーレンスノード，(1, 2)〜(4, 2)，(1, 4)〜(4, 4) の計 8 ノードはコアおよび L1，L2 キャッシュを持つ計算ノードであり，残りのノードはルータだけを持つパスノードである．計算ノードのうち，左側 4 つ ($x \leq 2$) が第 1 のアプリケーションを，右側 4 つ ($x \geq 3$) が第 2 のアプリケーションを，それぞれ並列に実行する．DCE は，キャッシュタグをコヒーレンスノードの個数で割った剰余によりインタリーブされる．な

© 2012 Information Processing Society of Japan

表 1　評価パラメータ
Table 1　Evaluation parameters.

種類	パラメータ	値
計算ノード	プロセッサコア数	8 or 16
	プロセッサ	インオーダ，L1 ヒット時の IPC=1
	ISA	MIPS32
キャッシュ	ブロックサイズ	64 B
	L1 I/D キャッシュ	各 8 KB，2-way
	L2 キャッシュ	256 KB, 8-way, 10 cycles, inclusive
	DCE エントリ	6,144 エントリ，12-way
	コヒーレンスプロトコル	MOESI をもとに変更
主記憶	レイテンシ	200 cycles
ネットワーク	トポロジ	2 次元メッシュ
	ルータ	5 入力 5 出力，2 仮想チャネル
		X-Y 次元順ルーティング
	ホップレイテンシ	3 cycles
	リンク幅	16 B/cycle



(a) 8 計算ノード　　(b) 16 計算ノード

図 10　評価対象となるメニーコアの構成
Fig. 10　Organization of target many-core for evaluation.

お，文献 [5] では，各々の DCE は 1 つのコアとともにルータに接続される構成をとっている（すなわち，ルータが 6 入力 6 出力となる）が，本論文では拡張前の SimMc の計算ノードの構成をそのまま利用できるよう，コアと DCE とを別ノードに離して配置している．

計算コア 16 個の場合では，図 10 (b) に示すとおり，全体が縦方向に拡張された 5 × 9 の 45 ノード構成となる．そして，左上 4 つの計算ノードが第 1 の，右上 4 つが第 2 の，左下 4 つが第 3 の，右下 4 つが第 4 のアプリケーションを実行する．

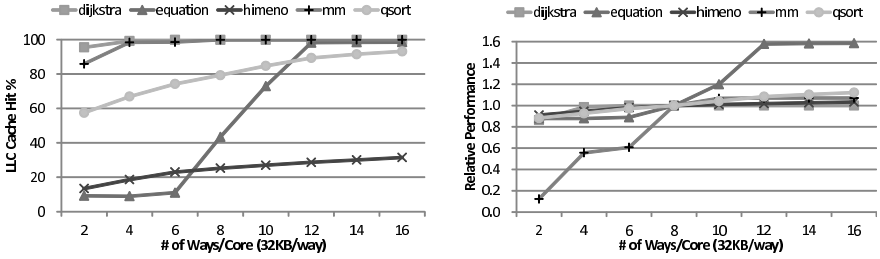
評価では，DCC を比較対象や評価のベースラインとする．移動可能なラインのうち，0%（まったく移動しない），25%，50%，75%，100%（すべて移動させる）のラインを移動させる設定を，それぞれ DCC0，DCC25，DCC50，DCC75，DCC100 と呼ぶこととする．さらに，上記 5 つの設定のうち，各々のアプリケーションの組合せにおいて最高の性能を示すものを，DCCOpt と呼ぶ．これはすなわち，DCC がアプリケーションの組合せごとに最適なラインの移動割合を知っていたと仮定した場合の理想的な性能を表している．本評価において，性能の向上率は Fair Speedup（FS）[8]，すなわち各アプリケーションの実行時間比の調和平均で表す．また，必要なデータの初期化にかかる部分は実行時間から除外する．各アプリケーションは，DCC0 において実行時間がほぼ等しくなるようにループの回数などを調整している．さらに，規定のループ回数を実行し終えたアプリケーションは，それまでの実行サイクル数を記録して，すべてのアプリケーションが規定のループ回数を実行し終えるまで実行を続ける．

なお，ASCEND における移動を行うコアの判定，および移動先コアの選択確率変動の判定は，25 万サイクルごとに行う．選択確率変動の判定周期を短くすると，アプリケーションの挙動変化への対応が素早くなるものの，サンプリングエラーにより判定の正確さが低下する．また，Spiller Selector のパラメータ k を 4/5 とし，Receiver Selector のパラメータ l を 3/4, t を 1/16 と定める．これらのパラメータはラインの移動や受け入れの積極性に影響を与える．k

© 2012 Information Processing Society of Japan

表 2　評価に用いるベンチマーク
Table 2　Benchmarks for evaluation.

名前	内容	主なパラメータ	並列化
dijkstra	最短経路探索	160 ノード	タスク並列
equation	equation solver kernel [9]	384 要素平方	データ並列
himeno	姫野ベンチマーク [10]	サイズ XS	データ並列
mm	行列積	256 要素四方	データ並列
qsort	クイックソート	360 K 要素	データ並列



キャッシュヒット率　　相対性能

図 11　評価に用いるベンチマークのキャッシュ容量特性
Fig. 11　Effects of cache size on benchmarks for evaluation.

を大きくするとラインの移動許可の閾値が大きくなり，ラインを移動する積極性が減少する．l を大きくすると他のコアへと移動したラインのヒットが重視されるようになり，また t を大きくするとラインの移動により性能に悪影響を受けていると判定する閾値が大きくなり，いずれの場合でもラインを受け入れる積極性が増加する．これらの積極性が小さいと，ラインの移動により性能が向上するケースを見落とす場合がある．一方で，あまり積極性を大きくすると，ラインの移動が悪影響を与えて性能を低下させる．これらの判定間隔やパラメータの決定には，このようなバランスを考慮して，いくつかの異なるパラメータを用いて評価を行い，最適なものを採用している．

Spiller Selector，Receiver Selector のカウンタのビット幅は，I 参照カウンタのビット幅を 8 ビット，公平度カウンタのビット幅を 10 ビットとし，選択確率カウンタのビット幅は 8 コア実行時で 6 ビット，16 コア実行時で 7 ビットとする．I 参照カウンタ・公平度カウンタのビット幅は，十分大きな幅をとったときとほとんど同じ判断を下すために必要な小さい値としている．また，選択確率カウンタのビット幅は変化の粒度と変化への対応の速さとのバランスを考慮して決定した．これらのセレクタを追加したことにより増加する記憶容量は，8 コア実行時で DCE あたり 252 ビット，16 コア実行時で DCE あたり 526 ビットである．これはディレクトリのエントリ数に換算すると約 5〜10 エントリに相当し，ディレクトリ全体（DCE あたり 6,144 エントリ）と比べて十分小さい．

ベンチマークには**表 2** で示す 5 種類を用いる．これらはすべて 4 コアの実行に合わせて並列化を施している．これら 5 種類のベンチマークの中から重複を許して 2 つ，または 4 つを選択し，評価のための 1 つのセッティングを構成する．セッティング名は，選択した各々のベンチマークの頭文字を並べたものとする．たとえば，dijkstra と qsort の組合せはセッティング DQ と呼び，mm が 2 つと himeno，qsort の組合せはセッティング MMHQ と呼ぶ．ここで，アプリケーションの順序を入れ替えただけのもの（たとえば，DQ と QD，MMHQ と HMMQ）は同一のセッティングとしてカウントする．したがって，セッティングの数は 2 アプリケーションの場合 15 種類，4 アプリケーションの場合 70 種類となる．

評価に用いるベンチマークにおけるキャッシュの特性は，**図 11** に示すとおりである．これは，L1 サイズを固定し，L2 サイズの容量およびウェイ数を増減させた場合の，キャッシュヒット率，および相対性能の変化についてまとめたものである．横軸にはウェイ数（すなわち容量），縦軸は L2 キャッシュヒット率，または容量 8 ウェイ（256 KB）の場合を 1 とした相対性能である．

各ベンチマークの特性について概説する．

dijkstra　ワーキングセットのサイズが L2 キャッシュと比べて小さく，他のコアからのラインの移動を受け入れやすいベンチマークである．

equation　ワーキングセットのサイズが L2 キャッシュよりも大きく，周期的にアクセスされるデータが多く含まれているベンチマークであり，キャッシュサイズを増加させていくとある点から急激にヒット率・性能ともに上昇する．すなわち，他のコアへラインを移動することで大きな利益を得られるベンチマークである．

himeno　ワーキングセットのサイズが L2 キャッシュよりもずっと大きく，キャッシュサイズの増加に対して緩やかにヒット率・性能が上昇する．したがって，他のコアへラインを移動する利益は equation よりも小

© 2012 Information Processing Society of Japan

さい．

mm　ワーキングセットのサイズが 8 ウェイの場合の容量にほぼ等しく，それよりも小さい容量で性能が大きく低下している．すなわち，他のコアからのラインの移動を受け入れにくいベンチマークである．

qsort　ワーキングセットのサイズが L2 キャッシュよりもやや大きく，その特性がコアによる，あるいは時間経過による変化が比較的大きいベンチマークである．

4.2　8 コア・2 アプリケーションの場合

図 12 に，8 コア・2 アプリケーションの場合における，DCC100 と ASCEND の DCC0 に対する性能向上比を示す．横軸はセッティング名で，ASCEND の DCC0 に対する性能向上比が高いものが右側にくるように並べ替えを行っている．右端は調和平均である．DCC0 から DCC100 までの間で，移動確率を固定した場合に最も性能向上比の調和平均が高かった DCC100 をここでは性能の比較対象としている．

DCC100，すなわち移動可能なすべてのラインを他コアへ移動する DCC と比べて，ASCEND は平均 1.5%の性能向上を得た．特に，セッティング EM，すなわち Equation Solver と行列積の組合せにおいては，DCC100 と比較して 16.9%の性能向上を達成した．

このセッティングについてさらに細かく挙動を分析する．**図 13** に，Equation Solver と行列積とのそれぞれについて，設定を DCC0，DCC100，ASCEND とで変化させたときのキャッシュヒット回数とミス回数と性能の変化を示す．グラフの縦軸は設定名を表す．棒グラフの横軸（上の軸）は，それぞれのコアで自コアでの L2 ヒット（Local Hit），他コアでの L2 ヒット（Remote Hit），L2 ミス（Miss）が発生した回数である．1 設定につき 4 本の棒グラフがあり，それぞれのグラフが 1 つのコアに対応している．単位は 1,000 回である．折れ線グラフの横軸（下の軸）は，DCC0 を 1 としたときの相対性能である．

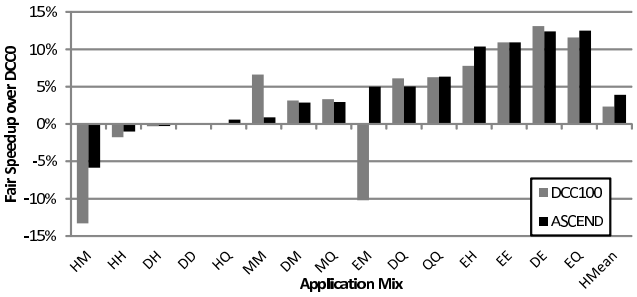


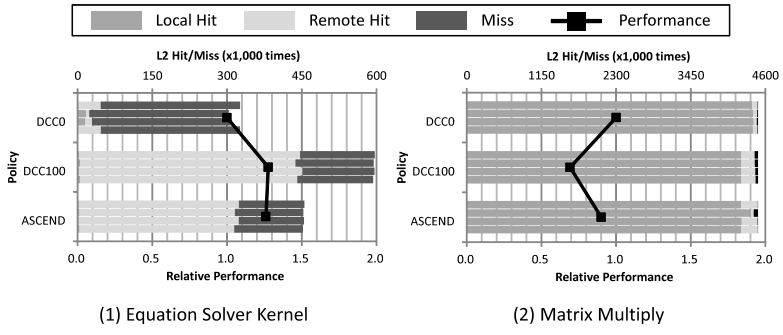
図 12　8 コア・2 アプリケーションにおける，ライン移動を行わない場合に対する性能変化

Fig. 12　Performance over DCC without spilling in 8 cores, 2 applications.

図 13 の DCC0 と DCC100 とを比較すると，Equation Solver ではラインの移動により他コアでのヒットが大幅に増加し，性能が約 28%向上した．しかしながら，行列積ではラインの受け入れによるキャッシュミスの増加が性能に与える影響が大きく，性能は約 31%低下した．これが全体の Fair Speedup では 10%強の低下を招いた原因となっている．一方，DCC100 と ASCEND とを比較すると，Equation Solver の性能は約 26%の向上とほぼ同程度に保ちつつも，行列積の性能低下は約 10%と大きく改善され，Fair Speedup でも約 5%の向上を果たした．ASCEND では，行列積の中でも特に 1 つのコアにのみ重点的にラインの移動を受け入れさせる挙動が見られた．これは，ラインの移動を受け入れることによる性能低下の影響が大きいコアとそうでないコアとが存在しているためと考えられる．前者のコアへのラインの移動を極力避け，行列積への性能の悪影響を最小限に抑えたことが，ASCEND が全体の性能を引き上げた要因である．

4.3　16 コア・4 アプリケーションの場合

図 14 に，16 コア・4 アプリケーションの場合における，ASCEND の DCC100 に対する性能向上比，および ASCEND と DCC25〜DCC100，および DCCOpt の DCC0 に対する性能向上比を示す．横軸はセッティングを表し，それぞれ性能向上比が高いものが右側にくるように並べ替えを行い，各点を折れ線で結んだグラフを示している．縦軸はグラフ（a）が DCC100 を基準とした，グラフ（b）が

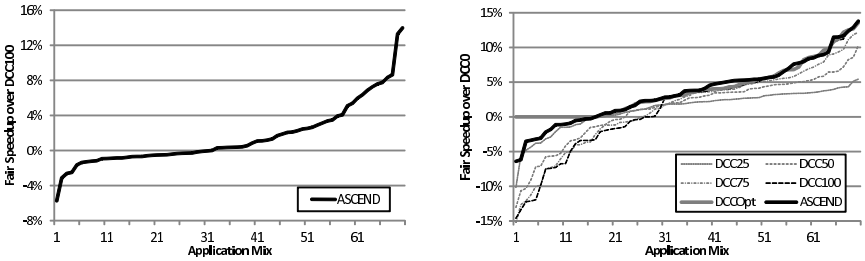


(1) Equation Solver Kernel　　(2) Matrix Multiply

図 13　Equation と MM の組み合わせにおける，設定を変化させたときのキャッシュヒット・ミスと性能の変化

Fig. 13　Relationships of method to cache hit/miss and performance.

© 2012 Information Processing Society of Japan



(a)　(b)

図 14　16 コア・4 アプリケーションにおける，(a) 全ラインを移動する場合，(b) ライン移動を行わない場合，に対する性能変化

Fig. 14　Performance over (a) DCC with spilling all possible lines or (b) DCC without spilling in 16 cores, 4 applications.

DCC0 を基準とした性能向上比である．グラフ（a）では前節と同様に，移動確率を固定した場合に DCC0 と比べた性能向上の調和平均が最も高かった DCC100 を，性能の比較対象としている．

4 アプリケーションの場合での ASCEND の性能向上は，平均で 1.5%，最高では 14.0%（セッティング MMME）となった．グラフ（b）より DCC の各設定における性能の変化を見ると，移動するラインの割合を減少させるにつれて，ベストケースにおける性能向上，ワーストケースにおける性能悪化ともに小さくなっていく様子が見られる．DCCOpt，すなわちセッティングごとにこれらの中から最適な移動割合を知っていると仮定した場合においては，DCC100 を選択するセッティングが全体の約半数の 33 個，DCC0 を選択するものが全体の約 1/4 の 17 個であり，残りの 19 個はその中間の DCC25，DCC50，または DCC75 を選択している．このうち DCC0 以外を選択する領域，すなわちグラフ（b）の右側 3/4 の領域においては，ASCEND は理想的な DCC の設定に近い性能を達成している．また，DCCOpt が DCC0 を選択する領域，すなわち左側 1/4 においては，押し出された I 参照を検出するために少数のラインを移動させることから，性能のオーバヘッドが発生する．しかしながら，こうしたオーバヘッドも DCC25，すなわち全体の 1/4 のラインを移動した場合よりもやや小さな値に抑えることができている．

4.4　DCE の個数による影響

DCC や ASCEND のデザインにおいては DCE の個数，すなわちディレクトリをどれだけ分散して持つかもまた考慮すべきポイントである．DCE の個数を減少させると，DCE のインタリーブによって生じる負荷の偏りが減少するため，必要な余剰エントリは減少する．しかしながら，少数の DCE にアクセスが集中することによりレイテンシの増大が問題となる．このことより，DCC のデザインにおいては適切な DCE の個数を選択する必要がある．ASCEND を適用した場合においては，DCE の個数を少なくすることで，1 つの DCE がより広いアドレス空間のディレクトリ情報を持つために，性能予測の精度が向上すると考えられる．一方で，余剰エントリの割合が減少し，より短い期間に無効化されたラインの情報しか参照できなくなる．これは性能予測の精度に悪影響を及ぼしうる．

DCE の個数を変化させた際の影響を評価するため，図 10 の構成からいくつかのコヒーレンスノード（DCE）をパスノードに置き換え，その代わりにそれぞれが持つディレクトリのエントリ数を適切に増加させる．そのうえで，どのように性能が変化するかを評価する．この評価では，図 10 の 8DCEs 構成，図 10 より（1, 1），(1, 3），(4, 1），(4, 3）をパスノードに置き換えた 4DCEs 構成，4DCE 構成からさらに（2, 3），(3, 3）を置き換えた 2DCEs 構成，そして（2, 1）以外のすべての DCE を置き換えた 1DCE 構成の 4 種類の配置について，それぞれ DCC0，DCC100，ASCEND における実行時間を計測する．DCE あたりのエントリ数は，総エントリ数が等しくなるように変更する．すなわち，4DCEs 構成では 8DCEs 構成の 2 倍の 12,288 エントリ，2DCEs 構成ではその 2 倍の 24,576 エントリ，1DCE 構成ではさらにその 2 倍の 49,152 エントリとする．いずれの場合も DCE のディレクトリの連想度は，8DCEs 構成のときと同じ 12-way とする．

各構成における性能評価を行った結果を**図 15** に示す．グラフのそれぞれの棒の名前は，手法名と DCE 数をスラッシュ区切りで並べて，ASCEND の 8DCEs 構成ならば ASCEND/8 のように表現している．横軸はセッティング名で，図 12 と同じ順に並べ替えを行っている．縦軸は DCC0/8，すなわち 8DCEs 構成でラインの移動を行わない場合と比較した相対性能である．DCC0 の各構成における性能はグラフに現れるほど大きくないため，省略している．各構成の DCC100 と ASCEND との性能を比較した場合，DCE 数を減少させるにつれその差が大きくなり，1DCE 構成では性能差は平均 2.1%，最高で 17.8%となった．これは，DCE の個数が少なくなればなるほど，またラインの移動が加わることによりラインの更新が頻繁に

© 2012 Information Processing Society of Japan

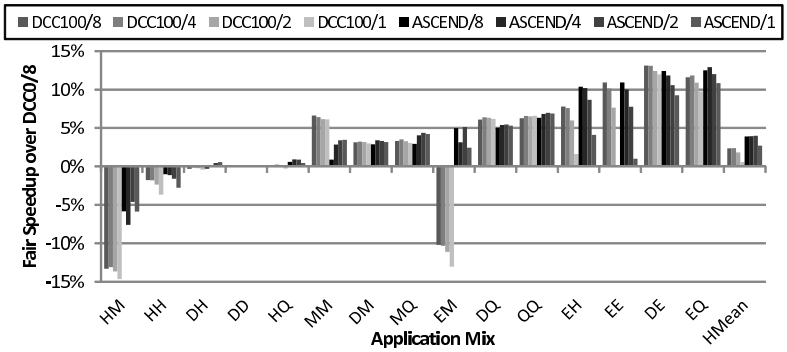


図 15　各構成を 8DCEs 構成・ラインを移動しない場合と比較した相対性能
Fig. 15　Performance over DCC with 8DCEs organization and no spilling.

なればなるほど，ネットワークの混雑が性能に与える影響が大きくなることに起因する．ASCEND がラインの移動が有効と思われるものに移動を限定することで，混雑を緩和する効果が得られたものと考えられる．ネットワークの混雑を考慮した場合，すなわち DCC0/8 をベースとした場合の ASCEND の性能は，ネットワークが混雑する前の 2DCEs 構成の場合が最大となり，8DCEs 構成と比べてさらに 0.1%の性能向上を得た．

5. 関連研究

DCC のように占有キャッシュをベースにラインの移動を可能とするキャッシュ構成においては，移動をどのように制御すればキャッシュの利用効率を高められるかについてさまざまな先行研究がある．

Adaptive Selective Replication（ASR）[11] では，確率的にラインの移動の可否を決定するための機構を持ち，その確率をパフォーマンスカウンタから得られた性能予測をもとに変化させることによって，適応的なラインの移動を行う．確率的に移動を制御する考え方は，ASCEND の Receiver Selector と類似する．ASR ではラインの移動先コアに対する考慮はしていないが，ASCEND の Receiver Selector では，ラインの移動割合と同時に，移動先として選択されるコアの割合を決定することで，これに対応している．Dynamic Spill-Receive（DSR）[12] では，各コアはラインを移動する Spiller と，そのラインを受け入れる Receiver とに二分される．一部のセットを利用してキャッシュミス数を測定し，よりミス数が少なくなるように各コアがいずれかの戦略をとる．DSR はミス数に合わせて細かく戦略を変更し，非常に良い性能を得る．しかしながら，コヒーレンス制御にスヌーピングを前提としており，一部のセットに対してすべてのコアが発したキャッシュミスを検出しなければならず，コア数が増加した際のスケーラビリティが問題となる．Cooperative Cache Partitioning [8] のように，OS と協調してソフトウェアベースの複雑なパーティショニングを行う方式も提案されている．Cooperative Cache Partitioning では空間的だけではなく，時間的にもパーティショニングを行うことが大きな特徴である．すべてのコアがキャッシュラインの移動によって同様に利益を得られる状況で，一時的にわざとパーティションに偏りを持たせることで，公平さを保ちながら全体の性能を向上させることができる．

ASP-NUCA [13] や Elastic Cooperative Caching（ElasticCC）[14] のように，各コアのキャッシュを占有領域と共有領域とに仮想的に分割する方式も提案されている．この方式では，追い出されたラインは必ず移動先コアの共有領域に格納されるため，占有領域のデータは他のコアによって悪影響を受けることがない．そのため，占有領域を適切に設定することで，ラインを受け入れることによる性能低下を防ぎながら，全体の性能を向上させることができる．こうした方式は ASCEND と競合するものではなく，特に ElasticCC は DCC 同様に分散化されたディレクトリを前提とするので，適切にこれらと ASCEND とを組み合わせることで，さらなる性能向上を達成できる可能性がある．

占有キャッシュと共有キャッシュの両方の利点を得るキャッシュ構成については，これまであげてきた占有キャッシュをベースとするもののほかにも，共有キャッシュをベースによく利用するデータを自コア周辺の空いているラインへとコピーする Victim Replication [3] や，データを命令・占有データ・共有データに分類し，その特性によって配置方法を変更する Reactive NUCA [4] などがある．

ASCEND では，多くの場合に使われずに無効化された状態でいるディレクトリのエントリを利用して，無効化される前の情報を残し，それを活用している．こうした余剰ハードウェア資源の持つ情報を利用するという観点からみると，Cached Load/Store Queue [15] がよく類似している．データアクセスを完了して無効化されたロード/ストアキューのエントリの情報を残しておく．直後にそのアドレスの情報が要求されることがあれば，その残された情報を利用することで，キャッシュへのアクセス回数を削減できる．Cached Load/Store Queue では残された情報を直接的に利用しているが，ASCEND ではもっと間接的に，ライン移動のポリシを変更したときの性能予測のためにこう

© 2012 Information Processing Society of Japan

した情報を利用している．

6. おわりに

占有キャッシュに追い出されたラインを移動する機構を追加したキャッシュ構成においては，その移動をどのように制御するかが，キャッシュの利用効率に影響を及ぼす．我々は，こうした構成の 1 つである DCC がいくらかの余剰エントリを持っていることに注目し，これらの持つ情報を利用して，キャッシュから追い出されたラインの移動を制御する ASCEND を提案した．評価の結果，DCC で移動可能なラインをすべて移動する場合と比べて平均で 1.5%，最大で 16.9%の性能向上を，少ないハードウェア資源の追加によって達成した．

今後の課題を述べる．まず，今回は余剰エントリへの特定のアクセス回数の情報を利用して性能予測を行ったが，これらを他の情報と組み合わせることで，より高速かつ正確な性能予測を行えるかどうか検討することがあげられる．また，ASCEND のようにディレクトリの余剰エントリを利用する方法を DCC 以外の方式と組み合わせたときに，どのような情報が新たに利用できて，それをどのように活用すればよいかは，大いに探求の余地があると考えている．

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（JST CREST）の「アーキテクチャと形式的検証の協調による超ディペンダブル VLSI」の支援による．

参考文献

[1] Chang, J. and Sohi, G.S.: Cooperative Caching for Chip Multiprocessors, *Proc. 33rd Annual International Symposium on Computer Architecture*, pp.264–276 (2006).
[2] Chishti, Z., Powell, M.D. and Vijaykumar, T.N.: Optimizing Replication, Communication, and Capacity Allocation in CMPs, *Proc. 32nd Annual International Symposium on Computer Architecture*, pp.357–368 (2005).
[3] Zhang, M. and Asanovic, K.: Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors, *Proc. 32nd Annual International Symposium on Computer Architecture*, pp.336–345 (2005).
[4] Hardavellas, N., Ferdman, M., Falsafi, B. and Ailamaki, A.: Reactive NUCA: Near-optimal block placement and replication in distributed caches, *Proc. 36th Annual International Symposium on Computer Architecture*, pp.184–195 (2009).
[5] Herrero, E., González, J. and Canal, R.: Distributed cooperative caching, *Proc. 17th International Conference on Parallel Architectures and Compilation Techniques*, pp.134–143 (2008).
[6] Katevenis, M., Sidiropoulos, S. and Courcoubetis, C.: Weighted round-robin cell multiplexing in a general-purpose ATM switch chip, *IEEE Journal on Selected Areas in Communications*, Vol.9, No.8, pp.1265–1279 (1991).
[7] 植原　昂，佐藤真平，吉瀬謙二：メニーコアプロセッサの研究・教育を支援する実用的な基盤環境，電子情報通信学会システム開発論文特集号，pp.2042–2057 (2010).
[8] Chang, J. and Sohi, G.S.: Cooperative cache partitioning for chip multiprocessors, *Proc. 21st Annual International Conference on Supercomputing*, pp.242–252 (2007).
[9] Culler, D.E., Gupta, A. and Singh, J.P.: *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann (1999).
[10] 理化学研究所情報基盤センター：姫野ベンチマーク，（オンライン），入手先 ⟨http://accc.riken.jp/HPC/HimenoBMT.html⟩（参照 2011-10-03）．
[11] Beckmann, B.M., Marty, M.R. and Wood, D.A.: ASR: Adaptive Selective Replication for CMP Caches, *Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp.443–454 (2006).
[12] Qureshi, M.: Adaptive Spill-Receive for robust high-performance caching in CMPs, *Proc. 15th IEEE International Symposium on High Performance Computer Architecture*, pp.45–54 (2009).
[13] Dybdahl, H. and Stenstrom, P.: An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors, *Proc. 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp.2–12 (2007).
[14] Herrero, E., González, J. and Canal, R.: Elastic cooperative caching: An autonomous dynamically adaptive memory hierarchy for chip multiprocessors, *Proc. 37th Annual International Symposium on Computer Architecture*, pp.419–428 (2010).
[15] Nicolaescu, D., Veidenbaum, A. and Nicolau, A.: Reducing data cache energy consumption via cached load/store queue, *Proc. 2003 International Symposium on Low Power Electronics and Design*, pp.252–257 (2003).

藤枝 直輝　（学生会員）

2008 年東京工業大学工学部情報工学科卒業．2010 年同大学大学院情報理工学研究科修士課程修了．現在，同大学院情報理工学研究科博士課程在学中．プロセッサアーキテクチャに関する研究に従事．

吉瀬 謙二　（正会員）

1995 年名古屋大学工学部電子工学科卒業．2000 年東京大学大学院情報工学専攻博士課程修了．博士（工学）．同年電気通信大学大学院情報システム学研究科助手．2006 年東京工業大学大学院情報理工学研究科講師．2011 年同准教授．計算機アーキテクチャ，並列処理に関する研究に従事．電子情報通信学会，IEEE-CS，ACM 各会員．

© 2012 Information Processing Society of Japan