Vol.2013-ARC-206 No.19
2013/8/1

値予測用投機実行回路によるキャッシュコヒーレンシ機構の高速化

大野 純[†1]　平木 敬[†1]

概要：マルチコア・メニーコアシステムにおいてキャッシュコヒーレンシを保つための手法としてディレクトリ機構が一般的になりつつある。しかしながらディレクトリ機構はコア数の増加に伴う容量の問題や3 hop アクセスによるレイテンシの増加といった欠点がある。一方ディレクトリを必要としない手法として、同期のタイミングで各コアが自発的に L1 のキャッシュを無効化する手法が提案されている。この手法では、実際には他のコアによって上書きされていないキャッシュラインが同期のタイミングで無効化されてしまう問題がある。本研究では値予測とともに用いられる投機実行回路により、これらのキャッシュラインの値を用いて投機実行することでキャッシュコヒーレンシ機構の高速化を図った。結果として複雑な予測機構を用いずに平均で 28%高速化することができた。

Accelerating Cache Coherence Mechanism with Speculation Circuit for Value Prediction

Jun Ohno[†1]　Kei Hiraki[†1]

Abstract: Directory is one of the common method to maintain cache coherence in multi/manycore systems. However, directory has problems in area and latency especially in systems with large number of cores. Conversely, directoryless protocol, where each core invalidates their own L1 cache lines at the time of synchronization is proposed. The problem of this method is that the cache lines which are not written by another core are invalidated at synchronization point. We accelerate the coherence mechanism by speculatively executing on these cache lines with speculation circuit for value prediction. Our result shows 28% acceleration on average without complicated prediction schemes.

1. はじめに

近年ではひとつのチップ内に複数のコアを搭載することが一般的になりつつあり、マルチコア・メニーコアにおいては効率的にデータの一貫性を保つ方法が課題となっている。ひとつのコアだけによって使われるプライベートなキャッシュライン (private) や、複数のノードによってアクセスされるキャッシュラインでも読み出し専用でアクセスされるもの (shared RO) には特別な機構は必要ない。しかし、複数のコアによって読み書きされるキャッシュライン (shared RW) については一貫性を保つための機構が必要になる。ページ内に一つでも shared RW のものがあれば shared RW のページ、それ以外で shared RO のデータを含むページを shared RO のページ、private なデータのみを含むページを private のページとして分類した場合のそれらのページへのアクセス数の割合を図 1 に示す。図に示すように shared RW のデータを含むページへのアクセスは 13%から 50%を占め、これらのページへのアクセスのレイテンシは性能に大きく影響する。

データの一貫性を保つ手法としてディレクトリ機構が研究されてきた [1][2][3][4][5]。この手法はバスを用いたス

[†1] 現在，東京大学
Presently with University of Tokyo

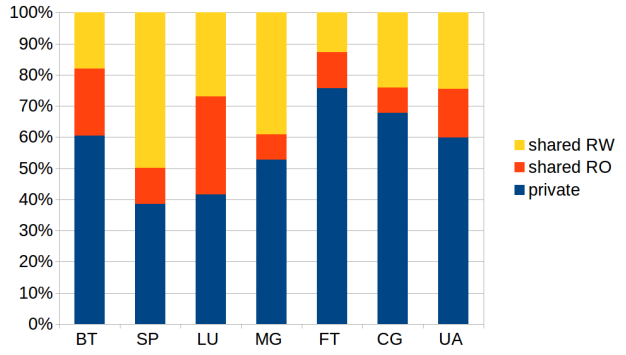

図 1 ページ単位でデータキャッシュを private / shared RO / shared RW に分類した時のそれぞれに対するアクセスの割合

ヌープ方式と比較してトラフィックの量が少なくて済むが、コア数が増えるに従いディレクトリの容量やレイテンシの問題が顕著になる。一方でディレクトリを使わずに一貫性を保つ手法として、一定の条件下で自発的に L1 の shared RW のキャッシュラインを無効化する手法 [6][7][8] が知られている。この手法はトラフィックが少なくディレクトリ特有の問題はないが、他のコアによって上書きされていないキャッシュラインを無効化してしまうことがある。この問題は無効化する頻度を下げることで緩和されるが、一方で頻度を下げることには弊害がある。同期命令のように他のコアがデータをキャッシュしていると実行できない命令は、他のコアが自発的にキャッシュラインを無効化するまで待たなければならないためである。

本研究では値予測とともに用いられる投機実行回路を活用し、自発的に無効化したキャッシュラインの値を用いて投機的に後続命令を実行することでこの問題に対処する。この手法により、本来有効なデータを無効化してしまうことによる性能の低下を抑えることができる。一方、自発的に無効化したキャッシュラインが実際に他のコアにより上書きされているような場合、これを高速化するには consumer-prediction[9] のような予測機構を用いて投機を成功させることが必要になる。本論文ではこれらの予測機構による効果について言及する。

2. 背景

2.1 ディレクトリ機構

マルチコア・メニーコアにおいてキャッシュコヒーレンシを保つ手法としてディレクトリ機構が一般的である。ディレクトリ機構はデータを書き込む際にそのアドレスのデータを L1 にキャッシュしているコアに無効化要求を送る手法で、snooping を用いる手法に比べればトラフィックが少なくて済むという利点がある。しかしディレクトリには容量、レイテンシの 2 つの面で問題がある。まず、容量に関しては単純な実装ではディレクトリの容量がノード数の 2 乗のオーダーで増える [1]。この問題に対処するための手法は多数提案されているが [2][3][4][5]、機構が複雑になるとともに、これらの多くはレイテンシを増加させるという欠点がある。次に、レイテンシに関してはホームノードに最新のデータが存在しなかった場合、最新のデータを保持しているノードにアクセスする (3 hop アクセス) 必要があり、レイテンシが増加する。この問題に対処するための予測手法として owner prediction[10] や consumer prediction[9] があるが、どちらもレイテンシの問題を解決するのは困難である。まず、owner prediction は設計によっては予測がクリティカルパスに入り込む上、予測が外れた場合のレイテンシが 4 hop になる。また、consumer prediction は consumer を多く予測してしまうと後続の書き込み命令で無効化すべきノードが増えるため遅くなる場合がある。以上のようにディレクトリ機構は容量、レイテンシについて解決困難な問題を抱えている。

2.2 Tear-off copy

一方でディレクトリを使わないコヒーレンシ機構 [6][7][8] が提案されている。ディレクトリはデータを書き込む際にどこに無効化要求を出すかを判断するために必要な機構であるため、一定の条件で各々のコアが自発的にデータを破棄するのであればディレクトリは不要になる [7]。特に private や shared RO のキャッシュラインは他のコアによって上書きされることがないため無効化する必要はない。したがって自発的に無効化する対象になるのは shared RW のキャッシュラインだけである。このような self(auto)-invalidation や tear-off copy によるコヒーレンシ機構では他のコアに無効要求を出さずに書き込みを行うため一般には sequencial consistency (SC) を満たせない [11][12] ものの、data race free (DRF) なコードにおいては SC として振る舞う [13]。

この手法の利点はディレクトリ機構で問題であった容量やレイテンシの問題が存在しないという点である。容量に関しては自発的にキャッシュラインを無効化するのでどのノードがデータを持っているかを把握する必要がなく、ディレクトリのような機構は不要である。レイテンシについては最新の値が (write through で) ホームノードに書かれるため 3hop アクセスのような高レイテンシのアクセスがない点、書き込みの際に無効化要求を発行する必要がないため書き込みのレイテンシが小さいという点で優れている。

一方でこの手法では他のコアによって上書きされていなくても一定条件下でキャッシュラインを無効化するため、有効なデータが使えなくなる場合があるという問題がある。ここで、有効なデータが使えなくなることによる性能劣化を軽減する手法として、無効化の頻度を下げることと無効化が必要なキャッシュラインを適切に選択することが考えられる。まず、無効化の頻度を下げる方法は、複雑な

機構を用いずに不必要な無効化を削減することが可能であるが、すでに述べたように同期命令等が遅くなるという弊害がある。また、無効化が必要なものだけを適切に選択する手法 [11] を用いれば性能劣化をさらに抑えられると考えられるが、複雑な機構が必要になるという問題がある。以上のように、この手法はディレクトリ機構特有の問題がないという点で優れているものの、本来有効なキャッシュラインを無効化してしまうという問題がある。

3. 提案手法

3.1 定期的に同期を取る手法の構成と問題

はじめに本研究の対象である tear-off copy によるコヒーレンシ機構 [7] について解説する。この手法ではキャッシュラインを private / shared RO / shared RW に分類し、一定時間ごとに同期を取る。各コアは同期のタイミングで保有している L1 のキャッシュラインのうち shared RW のものをすべて無効化する。また、書き込みは false sharing を防ぐために LLC でマージされる。

まず、キャッシュラインの分類はページ単位で行なう [14][15]。キャッシュラインの分類手法は本研究のテーマではないので概要のみ述べる。各ページは書き込みアクセスが全く無ければは Read-Only (RO)、それ以外は Read-Write (RW) としてマークされ、また、ひとつのコアによってのみアクセスされるページは private、それ以外は shared としてマークされる。TLB ミスが発生した時に必要であれば OS がページの分類を変更する。

続いて、一定時間ごとに同期をとって各ノードが自発的に L1 のキャッシュラインを無効化する方法について述べる。まず、L1 のタグには valid、guard の 2bit を用意する。valid はそのキャッシュラインが有効であることを意味し、guard は同期のタイミングでキャッシュラインを無効化するのを防ぐためのビットである。private や shared RO のページに属するキャッシュラインの guard はセットしておき、shared RW のページに属するキャッシュラインの guard はクリアしておく。そして同期の時に guard がクリアされているキャッシュラインの valid を一斉にクリアする。なお、valid はデータを L1 に持ってきた時にセットする。

次に上記の従来手法の問題点について述べる。上記の方法では、保持しているキャッシュラインが他のコアによって上書きされたかどうかにかかわらず、同期のタイミングで shared RW のデータをすべて無効化する。したがって無効化したものの実際には保持しているデータが有効であるという状況が発生する。さらにデータの private / shared をページ単位で分類するため、ページ内に 1 つでも shared RW のデータがあった場合そのページのすべてのキャッシュラインが自発的に無効化される。この問題に対して無効化が必要なものだけを選択的に無効化する手法が

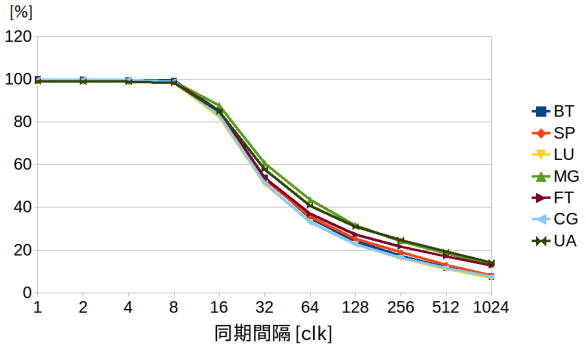

図 2 shared RW へのアクセスのうち、有効なデータがアクセスした時点で無効化されている割合 (16 コア)

提案されているが [11] 機構が複雑になる。図 2 は shared RW のページへのアクセスのうち、アクセスしたワードが有効であるにもかかわらず valid がクリアされている割合を同期間隔を変えて計測した結果を示している。グラフより、同期の間隔を長くすることでこの問題は解決されることがわかるが、同期の間隔を長くすることで同期命令などが遅くなるという問題が発生する。同期命令は他のコアがデータをキャッシュしていると実行できないため、次の同期によって他のすべてのコアで該当するキャッシュラインが無効化されるのを待たなくてはならない。このように、同期のタイミングを長くすれば多くの場合で性能が上がるものの、長くすることには弊害がある。

3.2 投機

本論文ではアクセスしたキャッシュラインが同期により無効化されていた場合でもその値を用いて投機的に実行を継続する手法を提案する。従来手法では L1 の valid がクリアされている場合、実際のデータが有効であっても LLC から最新のデータが到着するまで stall する必要があった。本研究ではこの問題を投機実行回路を用いて解決する。我々の手法では投機実行するためデータが有効であれば L1 ヒットした場合と同じ性能を示す。したがって shared RO や private のラインを無効化対象としても投機によりカバーできるためキャッシュラインの分類を厳密に行う必要はない。しかし、キャッシュラインを分類することには自発的無効化の対象となるキャッシュラインを減らせるためトラフィックを削減できるという利点がある。また、ページの分類を private / shared RO / shared RW の 3 種類にするか、private / shared の 2 種類のみにするかについて、通常のページを自発的無効化の対象のページに変更するときには L1 の Guard をクリアする作業などが必要でオーバーヘッドが大きいため、可能な限りそのような処理は減らしたい。従って 3 種類に分類する方がトラフィック、ページ書き換えのオーバーヘッドの観点から優れていると考えられ、本研究では 3 種類に分類した。

Memory Parameters	
L1 way	4
L2 way	16
Block / Page size	64B / 4KB
L1 latency	4 cyc.
L2 latency	6 + 9 cyc.
Main memory latency	160 cyc.
Network Parameters	
Topology	2D mesh(2x2 / 4x4)
Switching time	2 cyc.
Routing time	2 cyc.
Link time	4 cyc.

表 1 評価に用いたパラメータ

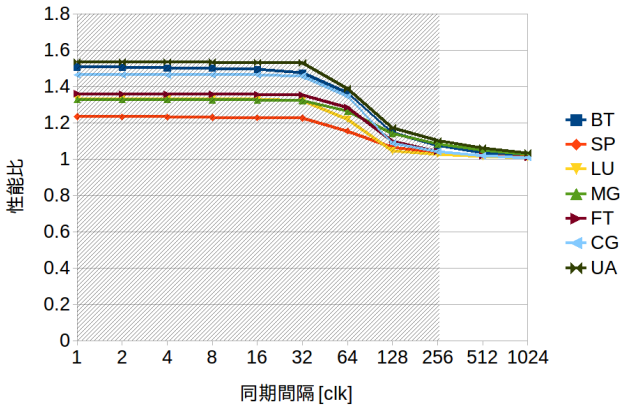

図 3 4 コアにおける同期間隔と性能比の関係

次に投機実行回路の実装による資源の追加について考察する。無効化されたキャッシュラインの値を使うというのはキャッシュラインの値を予測しているという見方ができ、この際に必要な投機実行回路は値予測用のものと同じ回路である。従って値予測機構を搭載するプロセッサにおいては新たな資源の追加は不要である。

4. 評価

4.1 評価環境

NAS Parallel Benchmark のうちの 7 つのベンチマークにおいて、10M 回目のメモリ命令から 100M 回目のメモリ命令までのすべての命令を pin を用いて記録し、本研究で開発したシミュレータ上で実行して評価を行なった。我々のシミュレーションモデルはメモリ以外のすべての命令が 1 サイクルで完結する in-order のプロセッサである。また、ネットワークなどのパラメータを表 1 に示す。

シミュレータの挙動は次のとおりである。一定期間ごとに同期を取り、同期のタイミングで L1 の guard が立っていないキャッシュラインをすべて無効化する。投機しない場合には valid の立っていないキャッシュラインはキャッシュミスとして扱うが、投機する場合には valid が立っていなくても LLC にデータ要求をして実行を続ける。LLC からデータが戻ってきた時点で投機した値と比較し、これらが同一であれば投機成功であり、そのまま実行を続ける。逆に値が一致しなかった場合は投機が失敗しているので roll back を行なう。

データの無効化のタイミングについて、LLC から L1 にデータを持ってくる途中に同期が入ることがある。特に LLC から L1 までのレイテンシよりも同期間隔が短い場合、同期により移送中のデータを無効化する設計では dead lock する。そのため我々のシミュレータでは移送中に同期が入っても、そのリクエストを出したメモリ命令だけは完了できるようにした。

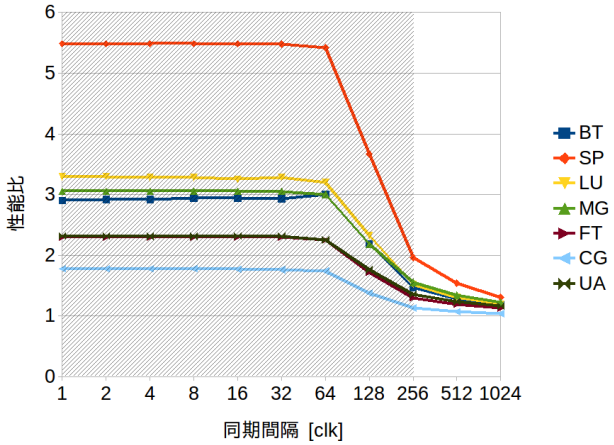

図 4 16 コアにおける同期間隔と性能比の関係

4.2 評価結果

まず、投機による効果を考察する。図 3、図 4 は 4 コア、16 コアにて投機をした場合としない場合の性能比を同期間隔を変えて計測した結果である。同期間隔が短いと持ってきたデータが使用後すぐに無効化されてしまい、次に使うときにはすでに無効になっているという状況が多くなるため投機の有無による差は大きく、4 コアの場合では同期間隔が 32 以下のとき 22%〜54%の差、16 コアの場合には最大 5.48 倍の差がある。しかし 4 コアの場合 32 を超えたあたりから、16 コアの場合 64 を超えたあたりから急速にこの差が縮まっていく。4 コアの場合では同期間隔を 1024 まで上げると投機による高速化は 1〜4%しかなかった。一方 16 コアでの計測結果では同期間隔を 1024 にした場合でも 4%〜31%の高速化が見られた。コア数が増えるにつれて L1 ミスの平均アクセスレイテンシが増加するため、この間に投機を行うかどうかの差が大きくなると考えられる。従ってコア数の多いシステムであれば投機がより重要になると考えられる。

次にアプリケーションごとの高速化の差について考察す

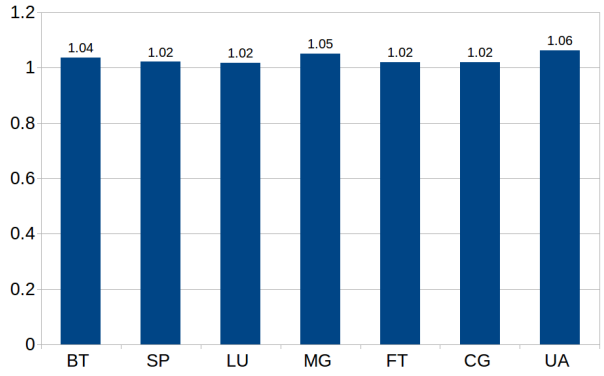

図 5 同期間隔 512、4 コアにおける性能比

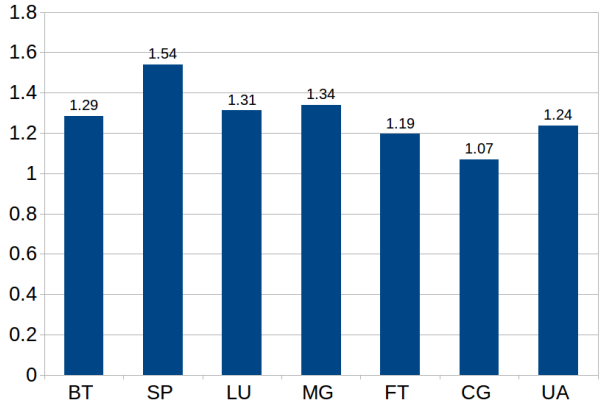

図 6 同期間隔 512、16 コアにおける性能比

る。16 コアにおいて SP の高速化が大きい理由は図 2 に示したように shared RW へのアクセスの割合が多いということが挙げられる。一方 shared RW へのアクセスの割合が最も小さいのは FT であるが、CG の高速化は FT よりも小さい。これは投機をしない場合の実行時間全体における shared RW へのアクセスによる stall が CG においては小さいためと考えられる。データが均等に配置されていると仮定するとアクセス一回あたりの平均レイテンシは同じなので、shared RW へのアクセスがクリティカルパス上にある確率が CG では低いということになる。逆に FT は shared RW へのアクセスの割合自体は低いもののこれらの多くがクリティカルパス上にあり、投機によってこれらを解消することの効果が CG を上回ったと考えられる。

続いて同期間隔について考察する。4 コアの場合には L1 ミスをした際のホームノードへのアクセスの最大 hop 数が 2hop なので往復 32clk であり、LLC の lookup に 15clk かかるため最大 48clk である。一方 16 コアではホームノードへのアクセスの最大 hop 数は 6hop なので往復 96clk であり、LLC の lookup を含めて 111clk である。つまり 4 コアの場合で同期間隔 48 以下、16 コアの場合で 111 以下というのは一部の sharedRW のデータは L1 には置かないということになる。また、平均レイテンシは 4 コアにおいて 31clk、16 コアにおいて 63clk であり、同期間隔がこの付近を上回るとクリティカルパスになるメモリアクセスにおいて L1 のヒット率が上がり、一気に差が縮まり出すと考えられる。しかし同期間隔が 1024〜2048 を超えると、今度は他のコアによって上書きされているにも関わらず古い値を使って実行してしまう場合が増える。本来であれば同期処理によりこのような状況を防ぐため、同期間隔をあまり大きくするとかえって遅くなる。従って同期間隔としては 512〜1024 の前後が妥当と考えられ、図 5、図 6 は 4 コア、16 コアにおいて同期間隔を 512 にした時の性能比を示したものである。このとき 4 コアでは 2%〜6%、16 コアでは 7%〜54%の高速化がみられた。

最後に、我々の手法に consumer prediction のような予測機構を搭載することの必要性について考察する。理想的な予測機構により投機が常に成功したと仮定した場合の実行サイクル数、性能比は 16 コア、同期間隔 512 のとき表 2 のようになった。理想的な予測機構を搭載しても最大 0.7%しか性能が向上しないので複雑な予測機構は投入した資源に見合った高速化が見込めないと言える。

	予測なし	理想予測器	性能比
BT	111445878	111310190	1.0012190079
SP	156163818	155310041	1.0054972428
LU	128809479	128562554	1.0019206604
MG	77290153	76772715	1.0067398685
FT	129314241	129268989	1.0003500608
CG	282977848	282538213	1.0015560196
UA	194756720	194042788	1.0036792504

表 2 理想予測器による高速化 (16 コア、同期間隔 512)

5. 関連研究

ディレクトリを用いないプロトコルとして、private や shared RO のデータは L1 に書き、shared RW のデータは L1 には置かずに直接 LLC に書く手法 [6] がある。shared RW のデータを一切 L1 に置かなかった場合の性能は多くのアプリケーションにおいて MESI より低いものの、これらのデータを L1 にキャッシュできるようにし、一定の条件下で L1 から追い出すようにすると MESI よりも性能が出るようになる。

self-invalidation (auto-invalidation) は他のノードによって write 命令が発行される前に自発的に該当するキャッシュラインを無効化することで後続の write 命令を高速化する手法である。この手法を用いてディレクトリ機構を高速化した研究として [11], [16] が知られている。

一方で、この self-invalidation を用いてディレクトリを使わずに一貫性を保つ研究がある。同期のタイミングで一斉にキャッシュラインを無効化する手法 [7] では一般には SC を保てないが、LLC からデータを持って来る際に何サ

イクル後に無効化するかを決めて L1 にコピーし、すべてのコピーが無効になったタイミングで書き込みを行うようにすれば SC を保つことができる [8]。

コヒーレンシ機構に投機を用いる手法としては [17], [18] が挙げられる。前者は投機的にメモリアクセスを実行し、必要な場合にはロールバックすることで SC を満たしつつ高速化するというものである。後者はディレクトリ機構において無効なラインの値を用いて投機実行するものである。この場合、無効なラインはそのうちの少なくともひとつのワードが他のノードによって上書きされているため、false sharing の解消による分の性能向上しか見込めないという見方 [19] がある。

6. まとめ

同期のタイミングで各コアが自発的に L1 のデータを無効化する手法は容量やレイテンシといったディレクトリ機構特有の解決困難な問題がない反面、同期によって本来有効なキャッシュラインが無効化されてしまうという問題がある。本研究では値予測とともに用いられる投機実行回路を活用し、これらのキャッシュラインの値を用いて投機実行することによりコヒーレンシ機構を高速化した。結果として 16 コア、同期間隔が 512 のとき評価した 7 つのベンチマークにおいて平均で 28%高速化することができた。また、我々の提案手法に理想的な予測機構を追加しても最大 0.7%の性能向上しか見られないことから、consumer prediction のような複雑な予測機構は割に合わないという結論を得た。

参考文献

[1] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal.
Directory-Based Cache-Coherence in Large-Scale Multiprocessors.
In *IEEE Computer, vol. 23, pp. 41-58*, 1990.

[2] Hongzhou Zhao, Arrvindh Shriraman, and Sandhya Dwarkadas.
SPACE : Sharing Pattern-based Directory Coherence for Multicore Scalability.
In *Proceedings of PACT*, 2010.

[3] Jason Zebchuk, Vijayalakshmi Srinivasan, Moinuddin K. Qureshi, and Andreas Moshovos.
A Tagless Coherence Directory.
In *Proceedings of MICRO*, 2009.

[4] Michael Ferdman, Pejman Lotfi-Kamran, Ken Balet, and Babak Falsafi.
Cuckoo Directory: A Scalable Directory for Many-Core Systems.
In *Proceedings of HPCA*, 2011.

[5] Daniel Sanchez and Christos Kozyrakis.
SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding.
In *Proceedings of HPCA*, 2012.

[6] Seth H. Pugsley, Josef B. Spjut, David W. Nellans, and Rajeev Balasubramonian.
SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches.
In *Proceedings of PACT*, 2010.

[7] Alberto Ros and Stefanos Kaxiras.
Complexity-Effective Multicore Coherence.
In *Proceedings of PACT*, 2012.

[8] Keun Sup Shim, Myong Hyon Cho, Mieszko Lis, Omer Khan, and Srinivas Devadas.
Library Cache Coherence.
In *Csail technical report mit-csail-tr-2011-027*, 2011.

[9] Sean Leventhal and Manoj Franklin.
Perceptron Based Consumer Prediction in Shared-Memory Multiprocessors.
In *Proceedings of ICCD*, 2006.

[10] Manuel E. Acacio, José González, José M. García, and José Duato.
The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors.
In *Proceedings of PACT*, 2002.

[11] Alvin R. Lebeck and David A. Wood.
Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors.
In *Proceedings of ISCA*, 1995.

[12] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou.
DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism.
In *Proceedings of PACT*, 2011.

[13] S. V. Adve and K. Gharachorloo.
Shared Memory Consistency Models: A Tutorial.
In *IEEE Computer, vol. 29, pp. 66-76*, 1996.

[14] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki.
Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches.
In *Proceedings of ISCA*, 2009.

[15] Blas Cuesta, Alberto Ros, María E. Gómez, Antonio Robles, and José Duato.
Increasing the Effectiveness of Directory Caches by Deactivating Coherence for Private Memory Blocks.
In *Proceedings of ISCA*, 2011.

[16] Stefanos Kaxiras and Georgios Keramidas.
SARC Coherence: Scaling Directory Cache Coherence in Performance and Power.
In *IEEE Micro, vol. 30, pp. 54-65*, 2011.

[17] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar.
Is SC + ILP = RC?
In *Proceedings of ISCA*, 1999.

[18] Jaehyuk Huh, Jichuan Chang, Doug Burger Gurindar, and S. Sohi.
Coherence Decoupling: Making Use of Incoherence.
In *Proceedings of ASPLOS*, 2004.

[19] Harold W. Cain and Mikko H. Lipasti.
Edge Chasing Delayed Consistency: Pushing the Limits of Weak Memory Models.
In *Proceedings of RACES*, 2012.