2004－ARC－160　(9)
2004／12／2

メモリ投機を支援するCMPキャッシュコヒーレンスプロトコルの検討

豊島隆志[†]　田代大輔[†]
バルリ ニコ デムス[†,☆]　坂井修一[†]

半導体プロセスの微細化に伴いチップマルチプロセッサが一般化しつつある。複数のプロセッサコアを有効活用する手法としてスレッド投機実行と呼ばれるマルチスレッド化手法が提案されてきた。スレッド投機の実現にはいくつかの付加的なハードウェアが必要となるが、本稿ではメモリ投機を支援する機構としてキャッシュコヒーレンスプロトコルに着目し、スレッド投機実行に起因するキャッシュミスを複数のプロトコルで評価した。その結果、ブロードキャストの適用により性能は約30%向上することがわかった。また、最も高い性能を達成したのは更新方式であるが、更新方式と無効化方式の性能差は6〜9%程度であり、ブロードキャストの適用効果に比べ、設計の複雑な更新方式を採用するメリットは小さいことがわかった。

Cache Coherency Protocols for Memory Speculation on CMP

Takashi Toyoshima,[†] Daisuke Tashiro,[†] Niko Demus Barli[†,☆]
and Shuichi Sakai[†]

Chip Multiprocessors are becoming common with the decreases in size of device dimensions. Speculative Multithreading which effectively uses multiprocessors to improve performance of sequential programs has been proposed. However, additional hardwares are needed for realization of Speculative Multithreading. In this paper, we focus on cache coherency protocols as a mechanism to support memory speculation. We study cache misses caused by Speculative Multithreading on candidate protocols. As a result, we improve performance with about 30% by application of Broadcast. Moreover, although Update-based protocols attained the highest performance, the performance gap between Update-based protocols and Invalidate-based protocols is six to nine%, and it turns out that Update-based protocols have little advantage against Invalidate-based protocols.

1. はじめに

一つのチップ上に複数のプロセッサコアを集積したチップマルチプロセッサ (Chip MultiProcessor—CMP) は、並列化されたマルチスレッド、マルチプロセスのアプリケーションが持つスレッドレベルの並列性 (Thread-Level Parallelism—TLP) を利用することで高い性能を引き出すことができる。近年では半導体プロセスの微細化により、チップあたりで使用可能なトランジスタ数が格段に増加した。それに伴い、より低い設計コストでこれらのトランジスタを有効活用できるCMPも一般的なものとなってきた。

しかしながら、CMPがその性能を発揮するのは並列化されたアプリケーションに対してであり、現在、また将来においても多数用いられるであろう逐次実行のアプリケーションにおいては、単体のプロセッサコアの能力しか引き出すことができない。これら逐次実行のアプリケーションにおいても同様に複数プロセッサコアの能力を引き出すためには、逐次実行のアプリケーションからもTLPを抽出する必要がある。

逐次実行のアプリケーションからTLPを抽出する手法として、スレッド投機実行 (Speculative Multithreading) と呼ばれる技術がある。スレッド投機実行は逐次実行のアプリケーションをスレッドに分割し、それらを投機的に並列実行することでTLPを抽出する。投機的に並列実行を行うことでスレッド間の制御依存、データ依存の制約が緩和される。このため依存関係の有無が曖昧で並列化できなかったアプリケーションにおいてもTLPを抽出することが可能になる。

CMPは、プロセッサコア間の距離が近く、スレッドの実行管理やスレッド間通信のオーバーヘッドを小さくすることが可能であり、スレッド投機実行を行うのに適したアーキテクチャと言える。このためCMPを対象としたスレッド投機実行に関する研究が広く行われてきている [1,7,9,10,12,15]。

† 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
☆ 現在，日本テキサス・インスツルメンツ株式会社
Presently with Texas Instruments Japan Ltd.

スレッド投機実行を行うにあたってはキャッシュのコヒーレンスが重要となる。文献 [8] によれば、共有メモリを用いた代表的な並列アプリケーション (FFT、LU、Barnes、Ocean) のキャッシュミスは、4 個のプロセッサコアそれぞれに 64KB のデータキャッシュ(2-way セットアソシアティブ、32B ライン) を接続した環境を想定すると、キャッシュミスは全体で平均約 6%、共有ミス (sharing/coherence miss) は平均で 1〜2%であり、もっともキャッシュミスの多い、Ocean においても共有ミスは 15%以下となっている。一方、我々の研究 [2] によれば 4 個のプロセッサコアそれぞれに 16KB のデータキャッシュ(2-way セットアソシアティブ、64B ライン) を接続した環境を想定すると、スレッド投機実行によるベンチマーク (SPEC CINT95 [11]) では、キャッシュミスは全体で平均 20%を越え、共有ミスにおいても平均約 15%となっている (図 1)。これらの比較はキャッシュサイズが異なるためフェアではないが、スレッド投機実行における共有ミスは、共有メモリを用いた並列アプリケーションの中でも、特に共有ミスの多いものに匹敵すると予想できる。このため高いパフォーマンスを引き出すためには、キャッシュコヒーレンスプロトコルや、キャッシュの構成についての注意深い検討が必要となる。

キャッシュについての関連研究では SVC [6]、Hydra [7]、STAMPede [12]、IACOMA [5,9] などが挙げられるが、どの研究においてもコヒーレンスプロトコルについては無効化方式 (Invalidation-based) が採用されている。本稿では無効化方式に加え、更新方式 (Update-based) のコヒーレンスプロトコル [2,14] についても検討、比較を行い、更新方式導入に伴うトレードオフを考察する。

以下、2 章では本稿で評価の対象としているスレッド投機実行アーキテクチャNEKO について述べる。3 章前半ではメモリ投機を支援するにあたりキャッシュコヒーレンスプロトコルに必要となる機能について説明し、後半でスレッド投機実行に特有のキャッシュミスについて説明する。4 章で各コヒーレンスプロトコルについての比較評価を行い、5 章でまとめる。

2. NEKO

まず、本稿で評価の対象とするスレッド投機実行アーキテクチャNEKO について述べる。本アーキテクチャの構成を図 2 に示す。

逐次実行のアプリケーションとして記述されたプログラムは、コンパイル時に専用の最適化コンパイラにより静的にスレッドに分割される [13]。このとき分割されたスレッド間には、先行するスレッドから後続するスレッドへの一方向のみ、データ依存、制御依存が許されている。生成された実行コードにはスレッドの投機実行の制御に必要な命令が挿入される。レジスタ

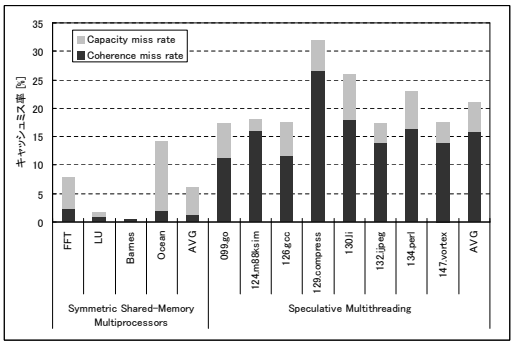

図 1 キャッシュミスの比較

を介したデータ依存に関しては、コンパイラにより静的に解析され、実行コードに挿入されるレジスタ通信命令に従ってレジスタコア間で同期、通信を行い解決する。一方メモリを介したデータ依存はメモリ投機を行う。

分割されたスレッドは実行時、スレッド予測器 (Thread Predictor—TP) により、動的に各プロセッシングユニット (Processing Unit—PU) に割り当てられる。割り当て対象スレッドの選択は予測に基づき投機的に行われ [3]、プロセッシングユニットへの割り当てはラウンドロビン方式で行われる。制御依存やメモリを介したデータ依存の投機に失敗した場合には、不正なスレッドは必要に応じて破棄 (Squash)、または再実行される。

先頭を走るスレッドは処理が投機的になることはないため、非投機スレッドと呼ぶ。一方、後続するスレッドは投機スレッドと呼ぶ。投機スレッドについてはそれぞれ投機レベルを定め、先頭スレッドから見てより後方に位置するスレッドほど投機レベルが高いと表現する。非投機スレッドの投機レベルはゼロと定義する。

個々のプロセッシングユニットは、アウトオブオーダ実行を行うスーパースカラプロセッサであり、それぞれ独立したレジスタファイルを持つ。これらのレジスタファイルは前述のとおり、プロセッシングユニット間で同期、通信を行うために遅延の小さいネットワークで接続されている [4]。

一次キャッシュは命令キャッシュ、データキャッシュそれぞれがプロセッシングユニットごとに個別に用意され、データキャッシュはメモリ投機の支援機構を備えている。メモリ投機を実現するためには、投機情報の保持、投機失敗の検出、及び失敗時の巻き戻しを行う機構が必要となり、NEKO ではこれらをキャッシュコヒーレンスプロトコルにおいて実現している [2,14,16]。

3. メモリ投機支援

3.1 投機支援に必要な機能

メモリ投機を実現するためには、基本的には通常の

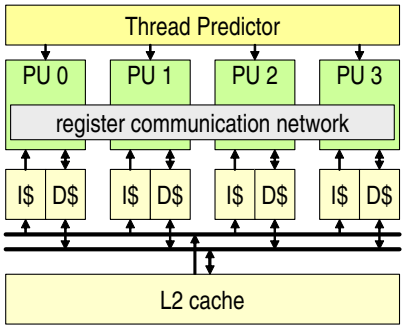

図 2 スレッド投機実行アーキテクチャNEKO の構成

ロード、ストアに加え、投機ロード、投機ストアを実現すれば良い。ただし、投機処理においては投機ミスを検出し、投機ミス時には投機前の状態への巻き戻す機構も実現する必要がある。

まずロードについて説明する。投機スレッドによるロードは全て投機ロードである。投機ロードでは、その時点でまだ値が確定していないメモリの値を、確定したものと見なして投機的に読み出す。そのため、投機的に読み出した値が先行スレッドによって書き換えられてしまった場合には、投機ロードミスを検出し、該当スレッドによる実行結果を破棄し、巻き戻さなければならない。よって、キャッシュはスレッドが非投機状態になるまで、投機ロードが行われたことを記録し、必要に応じて投機ミスを検出しなければならない。

次にストアについて説明する。投機スレッドによるストアは全て投機ストアである。投機ストアは、そのストアが最終的に発生する保証がない。従って、そのストアが不要とわかった際には、速やかに値を破棄し、本来の値に戻す必要がある。本稿では一次キャッシュによる投機支援を考えるため、値の巻き戻しについてはキャッシュを無効化するだけで良い。また、ストアはなるべく早い段階で後続スレッドへ伝達される必要がある。ストアの到着が遅れると、それに伴い投機ロードによる投機ミスの発生する確率が高くなってしまうからである。加えて、ストアは後続するスレッドのみに反映する必要があり、先行するスレッドについては、そのスレッドが終了したタイミングで、次にプロセッシングユニットに割り当てられるスレッドに備えて更新、または無効化する必要がある。新たに割り当てられるスレッドは、ストアを発行したスレッドから見て後続スレッドにあたるためである。このスレッド終了時に行う更新あるいは無効化を、遅延更新 (Delayed Update) あるいは遅延無効化 (Delayed Invalidation) と呼ぶ。

3.2 キャッシュミス

キャッシュミスの要因は、初期ミス (Compulsory miss)、容量ミス (Capacity miss)、競合ミス (Conflict miss) の 3 要因に分類される。共有メモリを用いた並列実行環境では、この 3 要因に共有ミス (Sharing miss) が加わる。

スレッド投機実行を考えた場合、さらにコヒーレンスプロトコルに起因するミスとして、遅延無効化ミス (Delayed Invalidation miss)、破棄ミス (Squash miss) が定義できる。

遅延無効化ミスは前述の遅延無効化により発生するキャッシュミスで、大きく分ければ共有ミスに含むこともできる。しかし通常の共有ミスは更新方式により解消できるが、遅延無効化ミスについては更新方式を導入しても解消することはできない。

破棄ミスはスレッドの破棄に起因するキャッシュミスである。キャッシュに載っていた値が投機ストアにより書き換えられていた場合、スレッドの破棄時に投機ストアを取り消すための無効化が生じる。この無効化が原因で発生するミスが破棄ミスである。

3.3 投機支援の最適化

投機ロードの記録に関しては、ライン単位で行うか、ワード単位で行うかの選択肢がある。ライン単位で行えばハードウェアはシンプルになるが、投機ミスを過剰に誤検出することになる。この比較についてはライン単位で行った場合には投機ミスの誤検出が多くなり、パフォーマンスに大きな影響を与えるという報告があり [9]、NEKO においても同様の結果が確認されている [14]。よって本稿の評価では投機ロードはワード単位で記録、判定している。

遅延無効化の処理についても同様に、ライン単位で行うか、ワード単位で行うかの選択肢がある。投機ロードの場合はワード単位の判定には、ワード毎の判定ビットを設置するだけで良いが、遅延無効化の場合はそれだけでは済まない。ワード単位で無効化を判定すると、最終的に同一ライン上に無効化されたワードと変更済みワードが混在してしまう。そのため、二次キャッシュへの書き戻しの際、特定のワードをマスクして書き戻す機構が必要となる。

キャッシュコヒーレンスプロトコルについては大きく分けて無効化方式と更新方式がある。1 章で示した通り、スレッド投機実行においては共有ミスが多い。このような場合には一般的に更新方式が適しているが、スレッド投機実行特有の事情により、軽率には判断できない。理由の一つとして挙げられるのが、前節で説明した遅延無効化の影響である。。先行しているプロセッシングユニットに更新情報を送ることができないため、更新方式を用いた場合でも、更新情報のおよそ半分 (つまり先行しているプロセッシングユニットに対する更新情報) は無効化情報に置き換える必要がある。遅延更新を実現するために更新情報を保持する特別なバッファを用意することも考えられるが、事実上キャッシュの二重化に等しく、かつスレッド切り替えのタイミングにおいて一斉更新が必要となるため、現実的ではない。また別の理由として、キャッシュミスの低減がロードの発行タイミングを相対的に早めてしまう可能性を挙げることもできる。この場合は投機ミスが増加

し、全体のパフォーマンスに悪影響を与える。

各プロセッシングユニットでアクセスするメモリ領域が近接している場合はブロードキャストが有効である。ブロードキャストは一種の予測に基づく事前更新と考えることもできる。つまり無効化されたラインに対して他のプロセッシングユニットがアクセスしていることから、近い将来接続されたプロセッシングユニットも同じラインにアクセスする可能性が高いと判断し、事前に更新をかける。更新方式による更新とブロードキャストによる更新は、ともに共有ミスを減らす効果があるが、更新のタイミングが異なるため、投機ミスの発生に対して異なる影響を与える。スレッド投機実行におけるブロードキャストの有効性については我々の研究ですでに明らかになっている [2]。

4. 評　　価

4.1 評 価 環 境

サイクルベースシミュレータ上で、専用の最適化コンパイラ [13] によりコンパイルされた SPEC CINT95 のアプリケーションを実行し、評価した。シミュレータは 2 章で述べたスレッド投機実行アーキテクチャを実装している。このシミュレータは、スーパースカラプロセッサのアウトオブオーダ実行、分岐予測などの挙動も含め、サイクル単位で詳細にシミュレートする。プロセッシングユニットの仕様とキャッシュのパラメータを表 1 にまとめた。

キャッシュコヒーレンスプロトコルについては無効化方式、更新方式それぞれについて、3 章で述べた最適化の効果を調べた。つまり遅延無効化の処理単位がキャッシュミスに及ぼす影響、およびブロードキャストがキャッシュミス、投機ミスに与える影響について比較し、本来、遅延無効化により発生していたはずのキャッシュミスが、ブロードキャストの効果でどの程度低減できたかを調べた。

4.2 結　　果

プロトコルの違いによるキャッシュミスの変化を図 3 に示す。upd、inv はそれぞれ更新方式、無効化方式を表し、後ろに_rwbr がついている方式はロードとストアに対してブロードキャストを、_robr がついている方式はロードのみに対してブロードキャストを行った方式を表している。ワード単位で無効化を行った場合、ライン単位で無効化を行った場合は、ともに無効化方式よりも更新方式のほうがキャッスミスは少なく、ブロードキャストもキャッスミスの低減に大きく役立っている。ただし無効化方式を見ると、無効化の処理単位はワード単位よりもライン単位のほうがキャッシュミスが減る。最もキャッシュミスが少ないのは、ブロードキャストを用いた更新方式においてワード単位で無効化処理をした場合で、約 7%のミスとなっている。一方、最もキャッシュミスが多いのは、無効化方式においてワード単位で無効化処理をした場合で約 34%。最少時のおよそ 5 倍となっている。また、ブロードキャストを用いた無効化方式においてライン単位で無効化処理をした場合はキャッシュミスが約 10%となっており、対設計コストも含めて考えると無視できない。

表 1　シミュレーションパラメータ

パラメータ	値
プロセッサユニット数	4 ユニット
パイプライン段数	7 段
フェッチ/発行/リタイア幅	4 命令
物理レジスタ数	128 レジスタ
機能ユニット	ALU × 2、ロード・ストア× 2
リオーダバッファ	64 エントリ
発行キュー	20 エントリ
ロード・ストアキュー	20 エントリ
BTB	1024 エントリ
Bimodal スレッド予測器	4096 エントリ
1 次命令キャッシュ	16KB 64B ライン 2-way セットアソシアティブ アクセスレイテンシ 1 サイクル
1 次データキャッシュ	64KB 64B ライン 2-way セットアソシアティブ アクセスレイテンシ 2 サイクル
2 次キャッシュ	理想化 (常にヒット) アクセスレイテンシ 16 サイクル 無効化レイテンシ 3 サイクル 更新レイテンシ 5 サイクル

次にブロードキャストの影響について検討する。キャッシュミスに対する影響を示したのが図 4、投機ミスに対する影響を示したのが図 5 である。更新方式で平均 7%、無効化方式で平均 33%のキャッシュミスを低減できた。ブロードキャストは無効化方式でより大きな効果が得られた。また図 5 からわかるように、キャッシュミスが減少すると投機ミスが増加する傾向にあることがわかった。

プロトコルごとの性能を実行サイクルの比で表したものが図 6 である。更新方式についてはワード単位で、無効化方式についてはライン単位で無効化処理を行った際のデータで比較している。ともにブロードキャストにより 30%前後の性能向上が見られる。

5. ま　と　め

スレッド投機実行のためのメモリ投機を支援する機構としてキャッシュコヒーレンスプロトコルに着目し、無効化方式に加え、更新方式での実現方法を考案し、シミュレータに実装した。また、それぞれの方式について、いくつか最適化のポイントを挙げ、実際の効果をシミュレータにより評価した。

その結果、無効化の判定は、更新方式ではワード単位、無効化方式ではライン単位で行うべきであることが判明した。また、更新方式、無効化方式ともにブロードキャストが効果的であり、適用した際に全体で約 30%の

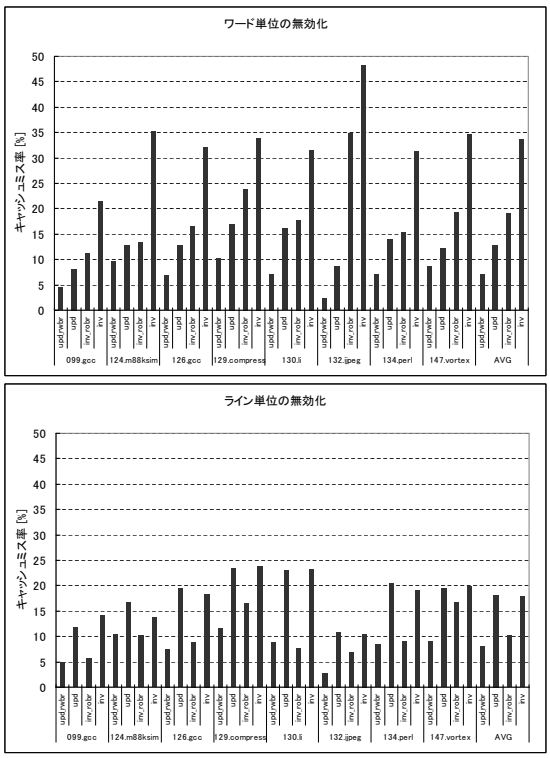

図 3 プロトコルの違いによるキャッシュミスの比較

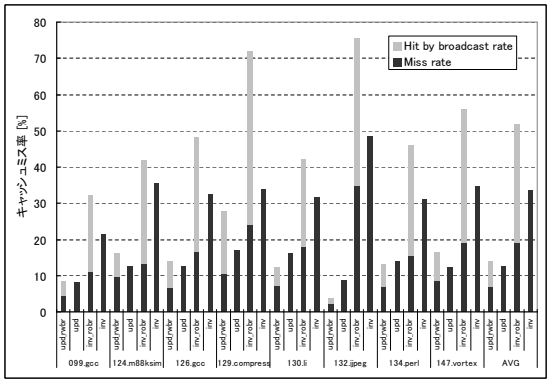

図 4 ブロードキャストがキャッシュミスに与える影響

性能向上が見られることがわかった。

謝　　辞

本稿の研究の一部は、文部科学省科学研究費補助金基板研究 B(2) #13480077 及び B(2) #16300013、半導体理工学研究センター (STARC)、科学技術振興事業団 CREST プロジェクト、日本学術振興会 21 世紀 COE プログラムの支援により行われた。

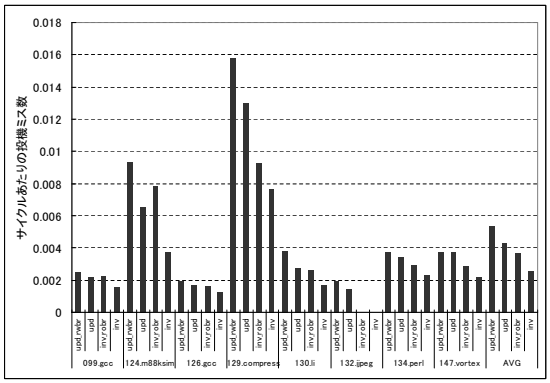

図 5 ブロードキャストが投機ミスに与える影響

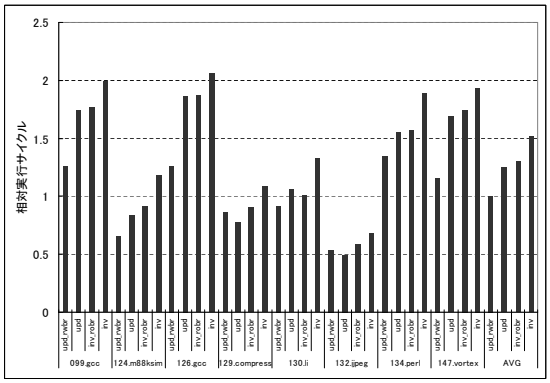

図 6 プロトコルの違いによる相対サイクル速度の比較

参 考 文 献

1) H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st MICRO*, pp. 226–236, 1998.

2) N. D. Barli, L. D. Hung, H. Miura, C. Iwama, D. Tashiro, S. Sakai, and H. Tanaka. Cache Coherence Strategies for Speculative Multithreading CMPs: Characterization and Performance Study. *IPSJ Transactions on Advanced Computing Systems*, 45(SIG 11(ACS 7)):119–132, 2004.

3) N. D. Barli, L. D. Hung, H. Miura, S. Sakai, and H. Tanaka. A Dual-Length Path-Based Predictor for Thread Prediction. *International Workshop on Innovative Architectures 2003*, 2003.

4) N. D. Barli, D. Tashiro, C. Iwama, S. Sakai, and H. Tanaka. A Register Communication Mechanism for Speculative Multithreading Chip Multiprocessors. In *Proc. of the SACSIS 2003*, pp. 275–282, 2003.

5) M. Cintra and J. Torrellas. Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multi-

processors. In *Proc. of the 8th HPCA*, pp. 43–54, 2002.
6) S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th HPCA*, pp. 195–205, 1998.
7) L.Hammond, M.Willey, and K.Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th ASPLOS*, pp. 58–69, 1998.
8) J. L. Hennessy and D. A. Patterson. 6.3 Performance of Symmetric Shared-Memory Microprocessors. In *Computer Architecture A Quantitative Approach 3rd Edition*, pp. 560–576, 2003.
9) V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.
10) P.Marcuello, A.Gonzalez, and J.Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th ICS*, pp. 77–84, 1998.
11) Standard Performance Evaluation Corporation. SPEC CINT95 Benchmarks. http://www.spec.org/cpu95/CINT95/, 1995.
12) J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th ISCA*, pp. 1–12, 2000.
13) D. Tashiro, N. D. Barli, S. Sakai, and H. Tanaka. An Edge-Based Thread Partitioning Method for Speculative Multithreading. In *IPSJ SIG Technical Report ARC-153*, pp. 67–72, 2003.
14) T. Toyoshima. Cache Organization for Memory Speculation. http://www.mtl.t.u-tokyo.ac.jp/, 2004.
15) J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.
16) Y. Yanagawa, L.D. Hung, C.Iwama, N.D. Barli, S. Sakai, and H. Tanaka. Complexity Analysis of A Cache Controller for Speculative Multithreading Chip Multiprocessors. In *Proc. of the 10th HiPC (LNCS 2913)*, pp. 393–404, 2003.