社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2005－ARC－164（18）
2005／8／4

スレッド投機実行のための キャッシュコヒーレンシプロトコルの検証

門馬 太平[†]　ルォン ディン フォン[††]　田代 大輔[‡]　坂井 修一[††]

チップマルチプロセッサ上でシングルスレッドのプログラムを高速に動作させるための技術として、スレッド投機実行という手法が提案されている。スレッド投機実行は、シングルスレッドのプログラムから並列性を抽出し、複数のプロセッサを用いて並列に実行する手法である。しかしながら、スレッド投機実行を行う上で重要な要素となる、メモリ投機を実現するためのキャッシュコヒーレンシプロトコルの設計については、その正しさを示す具体的な検証は行われていない。本稿では、メモリ投機を行う上で必要な要素を備えたキャッシュコヒーレンシプロトコルを設計し、モデル化を行った。そして、そのモデルをモデル検査ツール SPIN を用いて形式的検証を行うことにより、その正しさを示した。

Verification of Cache Coherency Protocol for Speculative Multithreading

Taihei Monma[†] , Luong Dinh Hung[††] , Daisuke Tashiro[‡] and Shuichi Sakai[††]

Speculative multithreading increases performance of a single-threaded program on a Chip-Multiprocessor by exploiting thread-level parallelism. Extending cache coherency protocol is general method to support memory speculation which is important element for speculative multithreading. While several cache coherency protocols supporting memory speculation have been proposed, the correctness of these protocols has not been formally verified. In this paper, we designed and modeled a protocol which supports memory speculation for speculative multithreading. Finally, we verified the protocol by using the SPIN Model Checker and confirmed correctness of the protocol.

1　はじめに

近年、マイクロプロセッサの性能向上はめざましい。しかしながら、回路が複雑化する事による配線遅延の増加や、消費電力、発熱の問題が大きくなってきているため、動作周波数を向上させるのが困難になりつつある。そこで現在注目されているのが、ひと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）である。CMP はスレッドレベル並列性（Thread-Level Parallelism: TLP）を利用するため、マルチスレッドのプログラムでは性能の向上が期待できる一方、シングルスレッドのプログラムは高速化されないという欠点がある。

そこで提案されているのがスレッド投機実行という手法である [1]。スレッド投機実行では、シングルスレッドのプログラムを複数のスレッドに分割し、それらを投機的に並列実行することにより TLP を抽出する。スレッド投機実行の際にはスレッド間の依存関係の解決に様々な工夫が必要であるが、メモリ依存関係の解決はキャッシュコヒーレンシプロトコルを拡張して行う場合が多く、その設計が問題となる。

スレッド投機実行のためのキャッシュコヒーレンシプロトコルはいくつか提案されてはいるものの [2,3]、実際にプロトコルが正しく動作するのかは示されておらず、具体的な検証は行われていない。そこで本研究では、スレッド投機実行のためのキャッシュコヒーレンシプロトコルをできるだけ実装に近い形でモデル化を行う。その上で、モデル検査ツール SPIN を用いて検証を行い、プロトコルの正常動作を確認する事を目的とする。

[†]東京大学大学院 工学系研究科
School of Engineering, The University of Tokyo
[††]東京大学大学院 情報理工学系研究科
Graduate School of Information and Technology, The University of Tokyo
[‡]現在、日立製作所 中央研究所
Hitachi, Ltd., Central Research Laboratory

2　スレッド投機実行とキャッシュコヒーレンシプロトコル

2.1　スレッド投機実行

スレッド投機実行は、シングルスレッドのプログラムを複数のスレッドに分割して、それらを投機的に複数のプロセシングユニット (PU) で実行することにより並列性を抽出する手法である。図 1 に PU 数が 4 の CMP 上におけるスレッド投機実行のイメージを示す。左側が逐次にプログラムを実行した場合、右側がスレッド投機実行を行った場合である。スレッド投機実行を行う場合、まずプログラムの実行命令列は複数のスレッドに分割される。分割されたスレッドは各 PU に割り当てられ、並列に実行される。

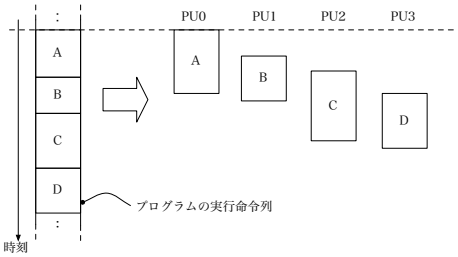


図 1: スレッド投機実行のイメージ

この際に、分割されたスレッド間には制御依存、データ依存が発生する。データ依存はレジスタを介した依存とメモリを介した依存に分けることができ、前者はコンパイラにより静的に解決することができる。一方、後者の静的な解決は困難であり、とりあえず依存がないと仮定してメモリ投機を行うことになる。この際に発生する問題がメモリ依存違反である。

図 2 にメモリ依存違反の発生の例を示す。図中のストア命令とロード命令は本来逐次に実行されなくてはならないが、スレッド投機実行により先行するストア命令を後続のロード命令が追い越してしまっている。そのため、ロードが誤った値を得ることになり、メモリ依存違反が発生する。このとき PU1 以降で実行されていたスレッドは破棄され、再実行される。

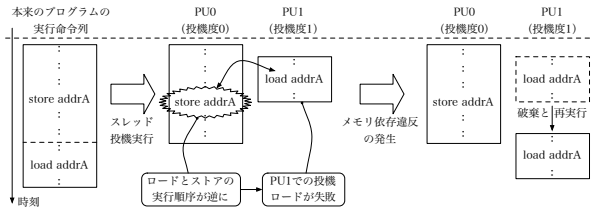


図 2: メモリ依存違反の発生

メモリ投機の支援にはいくつかの方法が提案されているが、分散 1 次キャッシュを利用して、キャッシュコヒーレンシプロトコルの拡張により支援する方法が主流となっている。

2.2　キャッシュコヒーレンシプロトコル

キャッシュコヒーレンシプロトコルとは、共有メモリマルチプロセッサにおいて複数の分散キャッシュ間の整合性を保つためのプロトコルである [4]。メモリを共有した複数のプロセッサが、それぞれ独立したキャッシュを持ち読み書きを行うとき、ただ並列に動作させただけではそれぞれのキャッシュ間で値の整合性が保てないという問題がある。

このような事態を防ぐために、これらのキャッシュに対してはキャッシュコヒーレンシプロトコルが定められている。全てのキャッシュがこのプロトコルに従って動作する限り、キャッシュ間の整合性が保たれることが保証される。

共有メモリ型の比較的結合度の高いマルチプロセッサシステムの多くは、バススヌープ方式のキャッシュコヒーレンシプロトコルを採用している。バススヌープ方式のプロトコルでは、複数のキャッシュコントローラがメモリアクセスの際にバスに流れる信号を「監視」しており、自分のキャッシュに保有されているデータが有効なものであるかをチェックする。有効でない場合は、該当するブロックを無効化、もしくは有効な値で更新する動作が行われる。

3　メモリ投機の支援

キャッシュによりメモリ投機を支援する場合、一般的な共有メモリマルチプロセッサにおけるキャッシュコヒーレンシプロトコルよりもさらに複雑なプロトコルが必要になる。具体的には異なるキャッシュ間のバージョン管理、投機データのバッファリング、そして前述のメモリ依存違反を検知して投機実行の中止、巻き戻しが可能でなければならない。

以下、本節では 1 ライン 1 ワードの分散 1 次キャッシュを考え、そこでメモリ投機支援を行う上で必要となる要素について議論を行う。

3.1　バージョンの管理

スレッド投機実行を行う環境では、各 PU ごとに異なる投機度のスレッドを実行しており、スレッドごとに扱うデータの新しさが異なる。これらのデータを管理することをバージョン管理という。実行されているスレッドのうち、プログラムの実行命令列

上で最も前方にあるスレッドの投機度が最も低く、0である。投機度が0の状態は投機実行ではないので、非投機状態とも呼ぶ。投機度は0, 1, 2, 3…と増大していく。バージョンの管理は、全てこの投機度を参照する事により行う。

3.2 投機ロードと投機ストア

メモリ投機を行う環境では、非投機スレッド以外におけるロード/ストアは全て投機ロード/投機ストアである[1]。そのため、プロトコルではこれらを適切に扱う必要がある。

まず、投機ロードについて考える。投機ロードはその時点では確定していない値を読み出す。そのため、投機ロードが行われたアドレスが、後に先行するスレッドにより書き換えられた場合、投機ロードは誤った値を得ることになり、RAWハザードによるメモリ依存違反が発生する（図2)。投機ロードに失敗した場合、そのスレッドと後続するスレッドは破棄され、巻き戻しを行う必要がある。投機ロードの失敗を検出するためには、投機ロードを行った際に、それをどこかに記録しておかなければならない。

次に投機ストアについて考える。投機ストアにより書き込まれた値は、後続の、すなわち投機度の高いスレッドに反映させる必要がある。そのため、投機度の高いスレッドの同じラインのうち、投機ストアが行われたスレッドと、それよりも投機度の高いスレッド以外を無効化あるいは更新する必要がある。しかしながら、これを行うにはキャッシュラインの持つ値がどのスレッドから転送されたものであるかを知る必要があるため、実装は複雑になる。そのため、効率は低下するが、投機ストアされていないラインを全て無効化/更新する。投機度の低いスレッドのラインに対しては無効化/更新を行う必要はないが、これらのラインはそのスレッドの投機実行が終了し、投機度の高い新たなスレッドが割り当てられるときに無効化する必要がある。これについては3.3節で述べる。

メモリ依存違反が発生するなど、何らかの原因で投機実行が中止されると、そのスレッドと後続のスレッドでそれまで行われた全ての投機ストアは取り消す必要がある。このとき、投機ストアを取り消すために、値の巻き戻しを行わなければならない。しかし、本稿で考える1次キャッシュによる投機支援では、投機ストアされたデータを1次キャッシュでバッファリングし、それ以降の階層には書き込まないため、巻き戻しには投機ストアされたキャッシュラインを無効化するだけでよい。

最後に、投機ロード/ストア時のキャッシュのミスヒットについて考える。ミスヒットの際にキャッシュラインの追い出しが必要になる場合は注意すべきである。なぜなら、投機ストアにより書き込まれた不確定な値を持つキャッシュラインは、投機実行を終えて値が確定するまでメモリへ追い出すことができないからである。

投機ロード時にミスヒットが起こると、もし先行するスレッドのキャッシュが投機ストアされた値を保持していた場合、そのスレッドがミスヒットしたスレッドに対して値を提供する。図3にこのときの動作を示す。複数存在した場合は、最も投機度の高いものが提供する必要がある。もし該当するスレッドが存在しなければ、通常通りメモリから値をロードすることになる。投機ストア時のミスヒットについては、投機ロード時のミスヒットと同じ動作の後、通常の投機ストアを行えばよい。

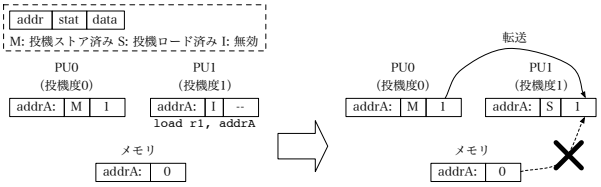


図 3: 投機ロードのミスヒット時の動作

3.3 投機実行の終了と新たな投機スレッドの割り当て

最も投機度の低い（非投機の）スレッドを実行中のPUが、スレッドの最後まで実行すると、そのスレッドの実行は終了し、そのスレッドで投機ストアされたラインは1次キャッシュ以降の階層にライトバックされる。その後、そのプロセシングユニットには最も投機度が高いスレッドが割り当てられる。この際、3.2節で触れたように、投機度の高いスレッドで投機ストアが行われたラインは無効化する必要がある。これは、新たに割り当てられる投機度の高いスレッドでは、最も新しい（最後に投機ストアされた）値をロードする必要があるためである。これを遅延無効化と呼ぶ。遅延無効化を行うには、遅延無効化を行う必要があるラインと、そうでないラインを区別するために、後続スレッドで行われたストア情報を記録しておく必要がある。もし記録を行わない場

[1]実際には、非投機のスレッドにおけるロード/ストアを投機ロード/投機ストアと区別する必要はない。

合、新たな投機スレッドを割り当てる際には全てのラインを無効化しなければならない。

3.4 投機実行の中止と値の巻き戻し

あるスレッドでメモリ依存違反が発生した場合、そのスレッドよりも投機度の高いスレッドは全て破棄されなければならない。破棄の際にはキャッシュ上の投機ストアされたラインを無効化する必要があることは既に述べたが、それだけでなく、他スレッドの投機ストアされたラインから転送されてきた値を保持するラインも無効化する必要がある。ラインの保持する値がメモリからロードされたものなのか、他のスレッドで投機ストアされた値が転送されてきたものなのかを区別するには、投機ロードミスの際にその情報を記録しておく必要がある。もしこの情報が存在しない場合、巻き戻しの際には全てのラインを無効化することになる。

4 SPIN

SPIN (Simple Promela INterpreter) [5–7] はソフトウェアやプロトコルのためのモデル検査ツールである。SPIN で検証するモデルは Promela (PROcess MEta LAnguage) と呼ばれる言語で記述される。Promela は並列動作を記述する C ライクな専用言語であり、複数のプロセスが並列に動作するシステムを記述することができる。変数、配列、構造体といった基本的なデータ構造のほか、プロセス間のメッセージ通信のための機構を標準で備え、実際のプログラムを組む感覚で記述できるのが特徴である。Promela で記述してしまえばそのまま検証が可能であるので、書いたその場で検証が可能なのが大きな利点となっている。

SPIN を用いた検証では、デッドロックや飢餓状態といった並列プログラムにおける基本的な問題を発見できる他、LTL (Linear Temporal Logic: 線形時相論理) と呼ばれる論理を用いてモデルの仕様を記述し、検証対象のモデルが要求されている仕様を満たしているかどうかを検証することもできる。

本稿ではこの SPIN を用いてキャッシュコヒーレンシプロトコルの検証を行う。

5 プロトコルのモデル化と検証

プロトコルの検証は、プロトコルのモデル化→モデルを Promela で記述→ SPIN による検証という手順で行われる。

5.1 検証対象のプロトコル

検証対象のプロトコルは MSI プロトコル [4] をベースに、3 節で述べたメモリ投機に必要な要素を追加したものとなっている。

5.1.1 キャッシュラインの構造

MSI プロトコルのキャッシュラインに対して最低限追加しなければいけないのは、投機ロード情報の記録のためのビットである。このビットは、そのラインに対して投機ロードが実行されたときにセットされる。このビットを投機ロードビットと呼ぶことにする。

これ以外には 3.3 節で述べた遅延無効化の有無を記録するビットと、3.4 節で述べた他のキャッシュからの転送の有無を記録するビットが考えられる。これらは必ずしも必要ではないが、効率を考えるとあった方が良い。前者を遅延無効化ビット、後者を転送ビットと呼ぶ。ここで挙げた 3 つのビットをまとめて投機条件ビットと呼ぶことにする。投機条件ビットは、投機実行の終了/中止の際に全てリセットされる。

図 4 に実際のキャッシュラインを示す。

タグ	状態ビット		投機条件ビット			データ
	Valid	Dirty	Load	Fwd	Delay	Word
	1bit	1bit	1bit	1bit	1bit	

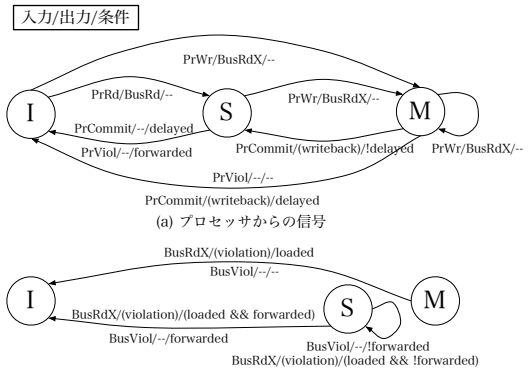
Load：投機ロードビット Fwd：転送ビット
Delay：遅延無効化ビット
図 4: プロトコルのキャッシュライン

5.1.2 信号と状態の遷移

まず、状態に関しては MSI プロトコルと同じく M (Modified), S (Shared), I (Invalid) の 3 状態である。そのため、状態ビットは 2 つである。それぞれの状態の意味は MSI プロトコルとは異なり、Modified 状態は「投機ストアが行われた状態」、Shared 状態は「投機ストアが行われていない状態」という意味になる。

次に、信号についてである。3 節で議論したように、メモリ投機を支援する環境では先行スレッドからの信号なのか、後続スレッドからの信号かによって受信後の動作が変化する場合が多い。そのため、受信側では送信側の投機度を知る必要がある。よって、送信側は信号と同時に自らの投機度を送信する。信号の種類は MSI プロトコルのものに加え、投機実行の終了時に送信する PrCommit/BusCommit 信号、メモリ依存違反等による投機失敗時に送信する PrViol/BusViol 信号が追加される。

図 5 にこのプロトコルの状態遷移図を示す。2 つのスラッシュを挟んで、前は信号の入力、中は信号の出力、後はそのときの条件を示す。なお、遷移先が前の状態と変わらず、バスに信号を流さない場合については省略している。条件の部分の loaded, forwarded, delayed というのは 5.1.1 節で述べた投機条件ビットのことで、それぞれ投機ロードビット、転送ビット、遅延無効化ビットがセットされている状態に対応する。条件の頭に!がついている場合は、逆にそれらのビットがセットされていない状態に対応する。



(a) プロセッサからの信号

(b) バスからの信号（遷移は投機度の低いスレッドからの信号のみ）

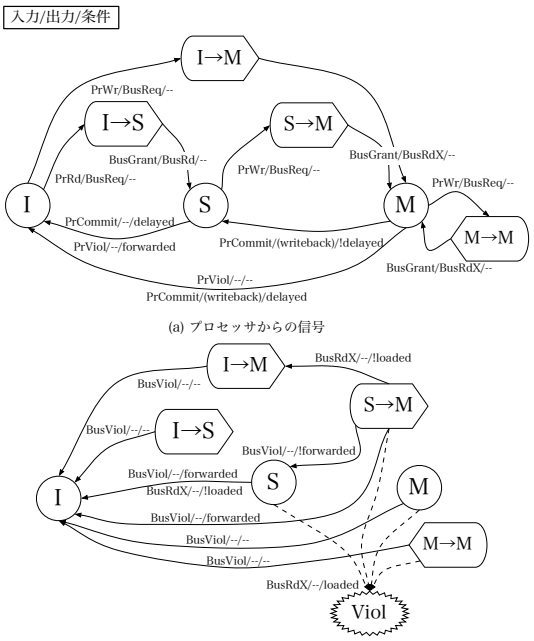
図 5: プロトコルの状態遷移図

5.1.3 中間状態の導入

より実際のシステムに近いモデル化を行うため、プロトコルに拡張を行う。実際のシステムでは、ある時点でバスを利用できるコントローラ（もしくはプロセッサ）は 1 つだけであり、複数のコントローラが同時にバスに信号を流すことはできない。そのため、バスに信号を流したいコントローラは、まずバスの利用要求を出し、その結果利用許可が出て、初めてバスを利用することができる。複数のコントローラからのバスの利用要求の調停を行うロジックは、バス・アービタと呼ばれる。

このような理由のため、実際にキャッシュラインの状態遷移が起こる際には、必ずしもプロセッサからの要求から最終的な状態遷移までが一度に起こるとは限らない。すなわち、バスに信号を流す必要がある場合は、一度バスの利用許可を待つ状態に遷移することになる。このような状態を中間状態と呼ぶ。

中間状態で待機中であるということは、バスが他のプロセッサにより利用されており、利用許可が下りていないということである。従って、待機中にはバスに他のプロセッサからの信号が流れる可能性があるため、中間状態でもスヌープを行う必要がある。その結果、中間状態でも状態遷移が発生し、中間状態から他の（中間）状態への遷移を考慮する必要が生じる。図 5 の状態遷移図は中間状態を考慮していないため、これを考慮した状態遷移図を新たに考える必要がある。図 6 に新たな状態遷移図を示す。BusReq はバスの利用許可を求めることを、BusGrant はバスの利用許可が下りたことを示す。Viol は実際に存在する状態ではなく、メモリ依存違反が発生したことを明らかにするために示してある。



(a) プロセッサからの信号

(b) バスからの信号（遷移は投機度の低いスレッドからの信号のみ）

図 6: 中間状態を導入したプロトコルの状態遷移図

一般にこれらの中間状態はを管理するのはキャッシュコントローラだけであり、キャッシュラインの状態ビットを増やすということは行わない。すなわち、中間状態においても、プロセッサ側からは MSI のいずれかの状態に属しているように見える。

5.2 想定するプロセッサのモデル

図 7 はモデル化の対象とする部分を示したものである。複数のプロセッサとキャッシュが 1 つのバスを共有している状態を考え、そのキャッシュとバスをモデル化の対象とした。モデル化の際には 1 つのキャッシュラインにのみ着目し、アドレスバスに流れる信号と、それに伴う状態遷移のみをモデル化した。実際に起こるデータの読み書きについてはモデル化の対象としていない。5.1 節で述べたプロトコルをこのモデルにあてはめ、Promela で記述を行った。

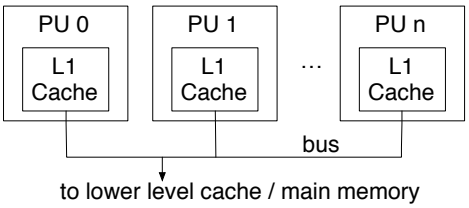


図 7: 想定するプロセッサのモデル

5.3 検証結果

Promela のコードを SPIN に入力すると、C で書かれた検証プログラムが得られる。そのプログラムをコンパイルして実行することにより、検証を行うことができる。検証では、

1. モデルがデッドロックにより停止する事が無いかどうか
2. モデルが常に仕様を満たし、コヒーレンシが維持されるかどうか

の二つを確認する必要がある。

1. については、記述した Promela のコードをそのまま SPIN に入力することにより、デッドロック検出用の検証プログラムが生成される。これをコンパイルして実行したところ、誤りは検出されなかった。

2. については、まず LTL の式でプロトコルの仕様を表現し、その式を SPIN に入力することにより、検証用のコードが出力される。検証用のコードを検証したい Promela コード中に挿入し、それを SPIN に入力することにより、検証プログラムが得られる。ここでは 20 の仕様を考え、それらを全て LTL 式で表現し、前述の作業を行った。全ての検証プログラムコンパイルして実行したところ、誤りは検出されず、プロトコルに仕様の違反がないことが確認できた。検証は、PU 数が 2 から 4 の範囲で行った。検証に要した時間、メモリ量と PU 数の関係を表 1 に示す。

以上により、プロトコルが正しく動作することが確かめられた。

表 1: 検証に要した時間とメモリ使用量

	2PU	3PU	4PU
状態数	4792	6.8×10^5	7.7×10^7
検証時間	$\cong$ 0 秒	5.3 秒	26 分 30 秒
メモリ使用量	1.8MB	52.5MB	8196MB

6 おわりに

CMP は次世代のマイクロプロセッサとして有望視されているが、その能力をさらに生かすためにはスレッド投機実行のような技術が有効である。スレッド投機実行においてメモリ投機を行う場合、それを実現するためのキャッシュコヒーレンシプロトコルが正しく動作するかは示されていない。そのため、プロトコルの正しさを証明するための形式的検証が必要である。本稿では、メモリ投機を行う環境でキャッシュコヒーレンシプロトコルに求められる要素を満たすプロトコルを設計し、モデル化を行った。そして、モデル検査ツール SPIN にモデルとその満たすべき仕様を与えて検証を行うことにより、プロトコルの正しさを確認することができた。

また、研究を通じて、自動検証技術が複雑なプロトコルの検証の際に有効であることも確認できた。本研究ではまずプロトコルの設計を行った後で検証を行ったが、実際にはプロトコルの設計の段階で SPIN のようなツールを活用することにより、より効率よく誤りの少ないプロトコルの設計が可能になるだろう。

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盤研究 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。

参考文献

[1] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414-425, Ligure, Italy, June 1995.

[2] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, NV, February 1998.

[3] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 1-12, June 2000.

[4] D. E. Culler, J. P. Singh and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[5] `http://spinroot.com/`

[6] G. J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[7] G. J. Holzmann. *THE SPIN MODEL CHECKER: Primer and Reference Manual*. Addison-Wesley, 2003.