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

2008－ARC－178（16）
2008／5／14

トランザクショナルメモリにおける並列実行トランザクション数動的制御法の提案とその評価

武田 進† 島崎 慶太† 井上 弘士‡ 村上 和彰‡

†九州大学大学院システム情報科学府 〒819-0395 福岡県福岡市西区元岡744 番地
‡九州大学大学院システム情報科学研究院〒819-0395 福岡県福岡市西区元岡744 番地
E-mail: †{takeda,shimasaki}@c.csce.kyushu-u.ac.jp, ‡{inoue, murakami}@i.kyushu-u.ac.jp

あらまし 本稿では，トランザクショナルメモリにおける性能向上を目的とした並列実行トランザクション数動的制御法を提案する．一般に，並列プログラムにおいては共有変数へのアクセスに関して排他制御を行う必要がある．トランザクショナルメモリでは，複数スレッドに対して共有変数の同時アクセスを許すことで高性能化を実現する．しかしながら，複数スレッドによる共有変数へのアクセスにおいて不都合が発生した場合には，それまでの実行を中断し，トランザクション実行のやり直しを行う必要がある．その結果，期待した並列効果を得ることができないだけでなく，場合によっては性能が低下する．この問題を解決するため，本稿では実行やり直しの発生可能性を事前に検知し，必要に応じて並列に実行されるトランザクション数を抑制する方式を提案する．32コアを搭載したチップマルチプロセッサを前提とした評価を行った結果，最大で1.6倍程度の性能向上を達成することを確認した．

キーワード トランザクショナルメモリ，チップマルチプロセッサ

Adaptive Management of Parallelism on Transactional Memories

Susumu TAKEDA† Keita SHIMASAKI† Koji INOUE‡ and Kazuaki MURAKAMI‡

† Graduate School of Information Science and Electrical Engineering, Kyushu University Motooka 744, Nishiku, Fukuoka, 819–0395 Japan
‡ Faculty of Information Science and Electrical Engineering, Kyushu University Motooka 744, Nishiku, Fukuoka, 819–0395 Japan
E-mail: †{takeda,shimasaki}@c.csce.kyushu-u.ac.jp, ‡{inoue, murakami}@i.kyushu-u.ac.jp

Abstract This paper proposes a technique to improve the performance of CMPs by mans of managing the number of transactions to be executed in parallel. In parallel computing, we need to manage sheared data in order to ensure the exclusiveness. In transactional memories, it is allowed the threads to access the shared data, resulting in higher performance. This is because we can aggressively exploit thread-level parallelisms. However, when a conflict takes place in the transactional memory, the associated thread execution needs to be aborted in order to guarantee the correct execution results. This abort operation degrades the CMP performance. To solve this issue, we propose an adaptive management mechanism to throttle or un-throttle the thread-level parallelism. In our evaluation, it is observed that in the best case we can achieve 1.6x speedup.

Keyword Transactional Memory, Chip Multiprocessor

1. はじめに

チップマルチプロセッサ（CMP）では，スレッドレベル並列性を活用することで高い性能を実現することができる．一般に，並列化されたプログラムの実行においては，複数のスレッドに共有する変数が存在する．このような共有変数に対してあるスレッドがアクセスする場合には排他制御を行う必要がある．従来の排他制御実現方式としてロック方式が用いられる．しかしながら，この手法にはプログラムを作成する際バグが混入しやすいことや，並列実行するスレッド数が増加した際に性能低下の原因となるといった問題点がある．

この問題を解決する1つの手段としてトランザクショナルメモリが注目されている[1]．トランザクショナルメモリでは，複数スレッドに対して共有変数への同時アクセスを許す．これにより，積極的な並列実行が可能となり高い性能を期待することができる．また，トランザクショナルメモリでは，複数スレッドによる共有変数へのアクセスにおいて不都合が発生した場合には関連する処理の中断ならびに再実行を行う．したがって，プログ

ラマはデッドロック等の心配をする必要がなく，比較的容易にプログラム開発を行うことができる．理想的には，並列実行可能なトランザクション数を増加することで CMP 性能を高めることができる．しかしながら，実際には上述した実行の中断やロールバックが発生するため，場合によっては性能が低下するといった問題がある．

そこで本稿では，トランザクショナルメモリをサポートした CMP の高性能化を目的とし，必要に応じて並列実行トランザクション数を動的に変更する実行方式を提案する．また，32 コアを搭載した CMP を前提とした定量的評価を行い，提案方式の有効性を明らかにする．実験の結果，搭載コア数と同数のトランザクションの並列実行を許す従来方式と比較して，提案方式を用いることで最大 1.6 倍程度の性能向上が得られることを確認した．

本稿の構成は以下の通りである．まず，第 2 節にて実験環境を説明する．第 3 節でトランザクショナルメモリとその実装方式の 1 つである LogTM について紹介し，その問題点を述べる．第 4 節では提案手法について説明し，第 5 節にてその評価結果を示す．最後に第 5 章でまとめる．

2. 実験環境

本稿では，フルシステムシミュレータである Virtutech Simics[4]と，GEMS[5]を用いて実験を行う．GEMS は，メモリシステムの詳細なタイミングモデルを実装している．Simics ではシミュレーションのターゲットモデルとして SPARC の ISA を選択した．また，OS は Soraris9 を用いた．

システムモデルパラメータを表 1 に示す．対象モデルは，プロセッサコア数 32 の CMP である．各プロセッサコアはインオーダ実行であり，全てのメモリアクセスが L1 キャッシュにヒットすると仮定した場合には IPC は 1 となる．それぞれのコアは，命令とデータそれぞれに関して 16KB のプライベート L1 キャッシュ（連想度は 4）を有する．また，全てのコアに共有される 4MB の L2 キャッシュ（連想度は 4）を搭載する．なお，L1 キャッシュと L2 キャッシュにおいて排他的にデータを保持する．

評価対象となる CMP モデルにはプロセッサコアが 32 個搭載されることから，キャッシュコヒーレンスによってバスが飽和する可能性がある．したがって，ブロードキャストを行わないディレクトリ方式を前提とした．コヒーレンシプロトコルは MESI であり，オンチップ・ネットワークではファンアウト数 4 の Hierarchical switch インターコネクトを用いてコア間の通信を行う．

表 1　　システムモデルパラメータ

	システムモデル設定
Processor Cores	32core, 1GHz, single-issue, in-order, non-memoryIPC=1
L1 cache	16KB 4-way split, 1-cycle latency
L2 cache	4MB 4-way unified, 12-cycle latency
Memory	4GB 80-cycle latency
Interconnection Network	Hierarchical switch topology, 3-cycle link latency

本稿における実験ではベンチマークプログラムとして Btree を用いた．BTree は，様々な並列アプリケーションにおいて，並列データ構造として用いられている．このプログラムでは，それぞれのスレッドが共有木に対して繰返しアクセスする．具体的には，80%の確率でデータ参照，20%の確率でデータ挿入を行う．木は 9 アレイで，初期値は深さ 6 である．与えるパラメータはプリロード 100,000 個でループ 100,000 回とした．また，並列実行部分（メインループ）のみを計測対象とした．

なお，フルシステムシミュレータを用いて CMP の動作を模擬するには性能のばらつきを考慮しなければならない[6]．そこで，主記憶のアクセスレイテンシに疑似乱数を用いて意図的にばらつきを与え，複数回実験を行う．その後，得られた結果から，95%の信頼区間を求める．この信頼区間は，エラーバーで表示する．

3. LogTM ベーストランザクショナルメモリとその問題点

3.1. トランザクショナルメモリ

トランザクションとは，有限な一連の機械語命令をさす．プログラマは共有変数のコンシステンシを保証するため，共有変数へのアクセスを含む一連の処理をトランザクションとして指定しなければならない．また，トランザクションは以下の 2 つの性質を満たす．

- atomicity：不可分操作．すべての処理が完了するか，何も行われないかのどちらか一方でトランザクションは終了する．
- serializability：並列実行したトランザクションの実行結果が，同じトランザクションを直列実行した結果と同じになる．

トランザクションの性質を満たす制御の一般的な方法として，ロックを用いた排他制御がある．これはトランザクションに対して，ほかのトランザクションからアクセスできないようにカギ（ロック）をかけ排他制御を実現する方式である．トランザクションは完全に排他的に実行されるため，atomicity と

serializability を保証することができる．しかしながら，スレッド数の増加に伴いロック獲得と解放のオーバーヘッドが生じ，実行時間が増加するといった問題がある[2]．

ロックによる排他制御の問題を解決するため，トランザクショナルメモリという制御法が提案されている[1]．ここで，トランザクショナルメモリにおける特有の用語を以下に定義する．

- conflict：トランザクションの性質を満足しない可能性があるメモリアクセスが引き起こす事象．
- isolation：conflict が検出されたアドレスに対してメモリアクセスを許可しない制約．
- abort：トランザクションの実行を中止すること．（これまでのトランザクションを取り消す）
- commit：isolation を無くす（取り払う）処理．

トランザクショナルメモリの制御法では，トランザクションの同期に関する保証を得ずに実行を開始する．つまり，トランザクションの投機的実行を行う．また，同期を保証しないことからトランザクションの性質を満たせない状況である conflict が発生する．conflict が発生すると，共有変数のコンシステンシを保証する必要があるため，共有変数のコンシステンシを保証するための制御を行う．conflict が発生し，結果を取消す必要が生じた場合には，当該トランザクションは abort する．abort する際は，isolation を解放し，トランザクション実行中に書き換えられた値を実行開始前の値に書き戻す処理（いわゆるロールバック）を行う．abort せずトランザクションの実行を終えると，当該トランザクションは commit を行う．

conflict の発生条件，ならびに，abort と commit の実現法は実装によって異なる．トランザクショナルメモリの制御には，conflict を検出する機構と，投機的書き込み，または，投機的書き込み前の値を保持するバッファが必要となる．

3.2. LogTM:Log-based Transactional Memory

Moore らは，LogTM という実装方式を提案している[2]．この手法では，複雑な処理はソフトウェアで，また，単純な処理はハードウェアで実行するというコンセプトで考案された．したがって，複雑な処理が必要とされる abort はソフトウェアで行い，比較的単純な処理である conflict 検出はハードウェアが担当する．

トランザクション実行中の読込みメモリアクセス時，conflict 検出のためセットするフラグを投機的読み込みフラグと呼ぶ．同様に書込みメモリアクセスの際にセットするフラグを投機的書き込みフラグと呼ぶ．conflict の検出は，キャッシュコヒーレンシ制御を拡張して実装され，リクエスト発生時に conflict 検出を行う．コヒーレンシリクエストを送るプロセッサコアと，それを受信するプロセッサコアの両方がトランザクション実行中であるとする．このとき，conflict の発生は以下のうち何れかの事象が検出された場合となる．

- 投機的書込みフラグがセットされているプロセッサコアへの読込みリクエスト
- 投機的読込みフラグがセットされているプロセッサコアへの書込みリクエスト
- 投機的書込みフラグがセットされているプロセッサコアへの書込みリクエスト

conflict が発生するとメモリアクセスを行ったプロセッサは nack（not acknowledge）を受け取る．一方，conflict が発生せず，メモリアクセスが成功した場合には ack（acknowledge）を受取る．

conflict が発生すると，リクエストを送信したプロセッサコアはストールする．このため，デッドロックに陥る可能性が生じる．例えば，トランザクションを実行しているプロセッサコア A と B を仮定した場合，B が A をストールさせ，その後 B が A にストールさせられる場合がある．このような時には何れか一方のプロセッサコアは abort しなければならない．LogTM では abort するプロセッサコアを決定する基準としてタイムスタンプを使用している[2][3]．また，LogTM ではトランザクション実行中に書込みアクセスが生じた場合には，通常の動作時と同様に L1 キャッシュへデータをストアする．このため，commit の処理としては投機的書込みフラグと投機的読込みフラグをリセットする必要がある．

3.3. スレッド数増加による性能低下問題

プロセッサコア数と実行スレッド数が等しく，かつ，コンテキストスイッチが発生しない場合，トランザクション並列実行数の上限はスレッド数と等しい．一般に，トランザクション並列実行数の増加は，conflict 発生確率を高くする原因となる．したがって，スレッド数の増加は，conflict 発生頻度の増加に繋がり，引いては，CMP の性能へ悪影響を及ぼすことが予想される．

そこで，スレッド数の変化が CMP 性能にどのような影響を与えるか調査した．図 1 にスレッド数毎の性能向上率を示す．横軸はスレッド数である．また，縦軸はスレッド数が 31 の場合に対する性能向上率を表す．以後，表 1 の設定におけるスレッド数 31 の BTree 実行時間を BASE とする．図 1 から，スレッド数 14 までは，スレッド数の増加とともに性能が向上

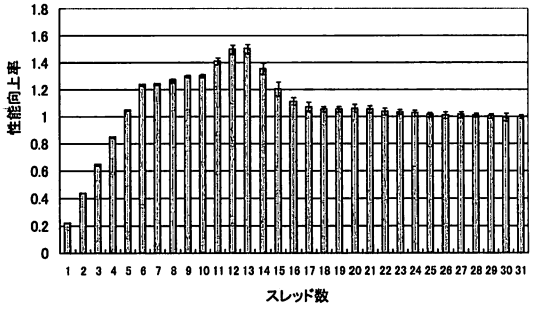

図 1　　スレッド数変更による性能向上率

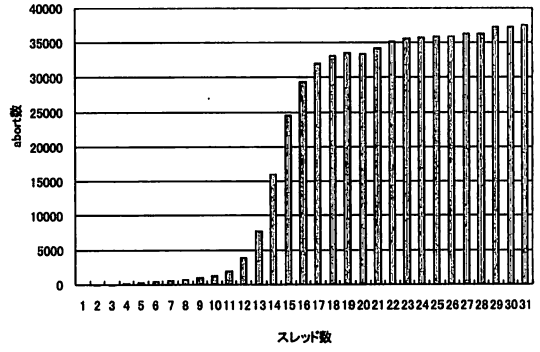

図 2　　スレッド数毎の abort 数

している．しかしながら，それ以上では性能が低下する傾向にある．この原因として abort 処理のオーバーヘッドが増加していることが考えられる．そこで，各スレッド数での実行に関する abort 発生回数を調べた．その結果を図 2 に示す．横軸がスレッド数，縦軸が abort 数を表している．この結果より，スレッド数 14 以上で abort 数が大幅に増加していることがわかる．以上より，スレッド数の増加に伴い性能が低下する原因は，abort の増加にあると推測される．

4. 動的トランザクション並列実行数制限手法の提案

4.1. conflict と abort 発生の関係

LogTM の実装法では，すべての conflict が abort に繋がるわけではない．しかしながら，abort 発生の主な原因は conflict であると考えられる．図 3 は 1,000 トランザクション commit 当たりの conflict 数と abort 数を表している．横軸はトランザクション commit 数であり，縦軸は abort 数と conflict 数である．この図より，conflict と abort の発生には相関があることが分かる．

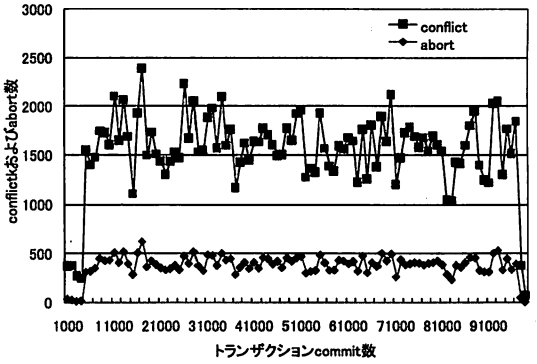

図 3　　スレッド数 31 における conflict と abort の関係

プログラム実行開始時にスレッド数を制限することで，トランザクションの投機実行を制限可能である．この場合，実行前にプログラマがスレッド数を指定しなければならない．つまり，投機実行と conflict 発生のバランスが最適なスレッド数をプログラマが既知である必要がある．したがって，最適なスレッド数を知るためにはプログラムを実行して調査しなければならない．

そこで，動的にトランザクション並列実行数を制限し，トランザクションの投機実行を制限する手法を提案する．本手法により，プログラムの実行中に適応的に投機実行と conflict のバランスをとることが可能になり，スレッド数に関わらず性能向上を達成することができる．

4.2. conflict と abort 発生予測と並列実行制御

abort 数に基づいて同時に並列実行可能な最大トランザクション数（以降，最大トランザクション数と記す）を制限するため，conflict の発生を予測する．そこで，本稿では conflict predictor という機構を用いる．conflict predictor により，最大トランザクション数を増加または減少させる．

まず，最大トランザクション数を増加させる手法について説明する．第 3.2 節で，conflict が発生していない場合は ack を受け取ると述べた．この ack の発生回数に基づいて制御する．最大トランザクション数を 1 減少させるために必要な ack の数を AP，ack の重みを W_{ack}，現在の最大トランザクション数を CTL（Current Transaction Limit）とする．ack の数が式 3.3 を満たすと，conflict predictor は最大トランザクション数を 1 増加させる．

$$AP = AW^{CTL} \tag{3.3}$$

たとえば，AW=2，CTL=2 とすると，ack を 4 回検出し

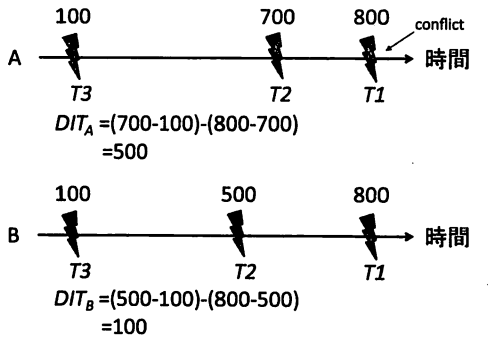

図 4：最大実行トランザクション数制御の例

た場合，*CTL* を 3 に更新する．このとき，これまでカウントした ack 数はゼロにリセットする．その後，ack を 8 回検出すると *CTL* が 4 に設定される．

次に，最大トランザクション数を減少させる手法について述べる．conflict の発生履歴に基づいて最大トランザクション数を決定する．履歴取得の対象となるのは，最新の conflict（*T1*），前回の conflict（*T2*）および前々回の conflict（*T3*）である．ここで，*T1*，*T2*，*T3* は conflict の発生時刻を表す．そして，それぞれの conflict 発生時刻（クロックサイクル数）を記憶しておき，conflict の発生間隔の差分である *DIT*（Difference of Interval Time）を式 3.1 によって求める．

$$DIT = (T2 - T3) - (T1 - T2) \qquad (3.1)$$

DIT の重みを W_{dit} とすると，新しい最大トランザクション数を *NTL*（New Transaction Limit）は式 3.2 で求められる．ただし，$DIT \geqq W_{dit} > 0$ とする．

$$NTL = CTL \times \frac{W_{dit}}{DIT} \qquad (3.2)$$

ここで，W_{dit} は *DIT* が *NTL* に与える影響を調整する役割を担う．つまり，W_{dit} の値が *DIT* の値に近い場合は W_{dit}/DIT の値が 1 に近づくため *NTL* の減少量は小さい．一方，W_{dit} の値が DIT に比べて極めて小さい場合は *NTL* の減少量は大きくなる．conflict の発生ごとに式 3.2 より *NTL* を算出し，必要に応じて最大トランザクション数を減少させる．conflict の発生状況に応じて *NTL* がどのように変化するのか，図 4 に示す例を用いて具体的に説明する．横軸は時間であり，現在の conflict である T1 が発生した時刻を 800 クロックサイクル，T2 を 700 そして T3 を 100 とする A の状況を考える．この場合，区間 *T1-T2* が区間 *T2-T3* よりも狭いことから，*DIT* の値は大きくなる．したがって，conflict が発生すると判断し，式 3.2 の W_{dit}/DIT の値が小さくなるため *NTL* が減少する．一方，B のように conflict の発生間隔がほぼ等しい場合は *DIT* の値が小さくなる．つまり，

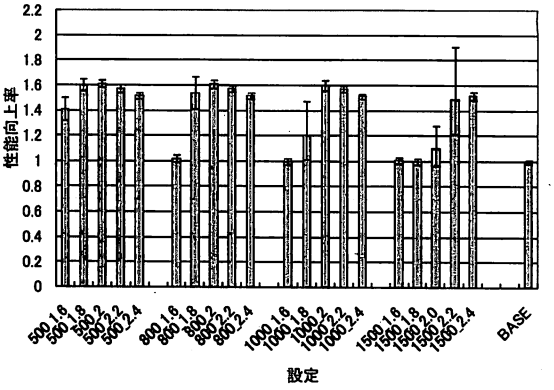

図 5　　動的制限による性能向上率

W_{dit}/DIT の値は 1 に近づくため *NTL* は *CTL* に近い値となる．

最大トランザクション数の増減制御において異なる式を用いた理由は，増加量および減少量に大きな違いがあるためである．一度に最大トランザクション数を大幅に増加させると conflict が増加することによって性能が低下する可能性がある．したがって，増加率を小さくすることで conflict を急増させないよう制御する．一方，減少させる場合は大幅に最大トランザクション数を減少させることで，conflict の発生を抑制し性能低下を防ぐことが可能になる．つまり，減少率を大きくして性能低下を迅速に回避する．

5. 評価

本章では，動的トランザクション並列実行数制限手法の評価を行う．適切な W_{dit} および W_{ack} の値はプログラムによって異なることが考えられる．ここでは W_{dit} は，500，800，1000 および 1500，W_{ack} は 1.6，1.8，2.0，2.2 および 2.4 として実験を行った．

図 5 は本手法による性能向上率を表している．横軸が W_{dit} と W_{ack} の値，縦軸が BASE の実行時間を 1 として正規化した値である．図 5 から，最大で 1.6 倍程度の性能向上を達成していることが分かる．これらの結果と比較して，W_{dit} および W_{ack} の値によっては，提案手法による性能向上率が高いことが分かる．これは，プログラムの振る舞いに応じて適切な最大トランザクション数を選択しているためである．

性能向上が得られた理由として，本手法により abort の発生が抑えられたと考えられる．そこで，abort 数を調査した．図 6 はその結果であり，縦軸は abort 数である．図 5 で性能向上を得られている場合は BASE と比較して abort 数が減少していることが分かる．しかしながら，W_{dit} および W_{ack} の値によっては abort 数が削減できていない．しがって，適切な W_{dit} および W_{ack} の設定が重要である．

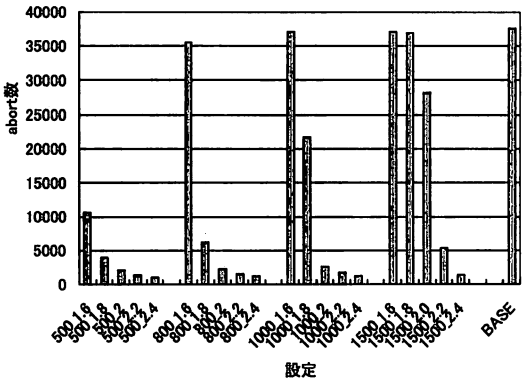

図 6　　動的制限による abort 数

6. おわりに

本稿では，トランザクショナルメモリにおける性能向上を目的とした，並列実行トランザクション数制限手法を提案した．ベンチマーク・プログラムを用いた評価を行った結果，最大で 1.6 倍程度の性能向上を達成することを確認した．

conflict predictor による予測処理のオーバーヘッドや実装に必要なハードウェアコストについて評価することが今後の課題である．また，並列実行トランザクション数制御に必要な W_{dit} および W_{ack} といったパラメータの適切な設定方法についても検討する必要がある．

謝辞

本論文の執筆にあたり，多大なるご協力を頂いた九州大学大学院システム情報科学府の小野貴継氏に心より感謝いたします．また，日頃からご討論頂いております九州大学 安浦・村上・松永・井上研究室ならびにシステム LSI 研究センターの諸氏に感謝します．なお，本研究は主に九州大学情報基盤研究開発センターの研究用計算機システムを利用しました．

文　献

[1] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural support for lockfree data structures. In Proceedings of the 20th Annual International Symposium on Computer Architecture,

[2] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In Proc. of the Twelfth IEEE Symp. on High-Performance Computer Architecture, pages 258-269, Feb. 2006.

[3] R. Rajwar and J. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In Proceedings of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 2002.

[4] Peter S.Magnusson et al.Simics:A Full System Simulation Platform.IEEE Computer, 35(2):50-58, Feb.2002.

[5] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R, Marty, Min Xu,Alla R.Alameldeen,Kevin E.Moore,Mark D.Hill,and David A. Wood.Multifacet's General Execution-driven Multiprocessor Simulator(GEMS)Toolset.Computer Architecture News,pages 92-99,Sept.2005.

[6] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture, pages 7-18, February 2003.