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

2005－ARC－162 （6）
2005－HPC－101 （6）
2005／3／7

SMT プロセッサにおける同期方式の検討

笹田　耕一 †　佐藤　未来子 †　内倉　要 †
加藤　義人 †　大和　仁典 †
中條　拓伯 †　並木　美太郎 †

Simultaneous MultiThreading（SMT）プロセッサは，複数の命令流が計算機資源（各種演算器やキャッシュメモリ）を共有して並列実行することでプロセッサの利用率を向上させる．並列計算では同期オーバヘッド削減が重要であるが，SMT プロセッサでは並列実行する命令流がチップ内にあるということを利用した同期の最適化が可能であり，すでに SMT-block という軽量な同期手法が提案されている．しかし，これを実現するための命令が複雑である点などから，提案されている手法をそのまま適用することは困難である．そこで，本稿では SMT-block を実現する命令セットについて再検討し，より単純なものにした．また，この機構を利用してさらに同期オーバヘッドを削減する手法であるロック獲得予約を提案する．評価の結果，改良した命令セットでも SMT-block と同様の性能改善が見られた．また，ロック獲得予約を利用した場合，通常の SMT-block を利用したプログラムと比較して最大 6% の性能向上を実現することができた．

Investigation of a Synchronization Method on an SMT Processor

Koichi Sasada ,† Mikiko Sato ,† Kaname Uchikura ,†
Norito Kato ,† Masanori Yamato ,† Hironori Nakajo †
and Mitaro Namiki †

A Simultaneous MultiThreading (SMT) Processor executes multiple instruction flows in parallel with sharing computing resources such as an execution unit or cache memory. In parallel computation, since it is important to reduce synchronization overhead, SMT-block technique which allows light-weight synchronization has been proposed in an SMT processor. However, it is difficult to implement a processor with an SMT-block scheme because of its complex instructions. In this paper, we propose more simple instructions for synchronization to realize SMT-block. Moreover, we also propose Lock Acquire Reservation to optimize synchronization overhead in an SMT processor. As a result of simulations, we achieve equivalent performance improvement as an SMT-block technique with our instructions. With Lock Acquire Reservation, we can gain performance improvement by 6% than a program using a normal SMT-block.

1. はじめに

近年，プロセッサアーキテクチャとしてチップマルチプロセッサ（CMP: Chip Multi Processor）や，Simultaneous MultiThreading（SMT）アーキテクチャなどの 1 プロセッサ上で複数の命令流を同時に実行しスレッドレベル並列性（TLP: Thread Level Parallelism）を利用して性能向上を目指すマルチスレッドアーキテクチャが研究されている．

とくに，SMT プロセッサ[7] は 1 プロセッサ中に複数の実行コンテキスト，すなわちプログラムカウンタ（PC）や汎用レジスタをもち，複数の命令流（実スレッド，AT: Architecture Thread）を並列に実行することが可能なアーキテクチャである．複数の命令流はプロセッサのその他の資源，たとえば演算器やキャッシュメモリなどを共有し実行する．複数の命令流がプロセッサ資源を共有して実行することで，プロセッサの利用率が向上し，資源の有効利用が可能である．

並列計算では多くの場合，各並列単位で同期を取ることが必要である．とくに，細粒度並列処理では同期回数が増えるため，同期コストが全体の計算時間に大きく影響するなど，同期コスト削減は並列計算，とくに細粒度並列計算において重要である．

SMT プロセッサは，複数の実スレッドが演算器を共有しているため，メモリを介すよりも高速な情報伝達が可能であり，これにより高速な同期が可能である．

† 東京農工大学大学院工学教育部
Graduate School Technology, Tokyo University of Agriculture and Technology

```
; acquire lock
retry:
  ll   $r, (addr)
  bnez $r, retry
  li   $r, 1
  sc   $r, (addr)
  beqz $r, retry
  ;; <critical section>
; release lock
  sw $0, (addr)
```

図 1 LL/SC 命令を用いたプログラム
Fig. 1 A Program using LL/SC

Tullsen らによる SMT-block 方式[8] は，これを利用した SMT プロセッサにおける軽量な同期手法である．しかし，提案されている命令は 1 命令でメモリの読み書きを行わなければならず，複雑である．

そこで，本稿では SMT-block と同様に軽量な同期を実現する命令セットを検討する．また，さらなる同期コスト削減手法であるロック獲得予約を新たに考案した．

以下，2 節で既存の同期方法と SMT プロセッサでの同期の問題点を述べ，本稿で改良の対象とする SMT-block の利点と問題点について述べる．3 節でその改善とこれを利用したロック獲得予約について述べる．4 節でこれについての評価を行い，5 節で関連研究を述べる．6 節でまとめる．

2. 従来の同期手法

本節では従来の同期手法について概観し，SMT プロセッサにおける問題点を述べる．そして，その問題点を解決するために既に提案されている SMT-block 方式を説明し，その課題を述べる．

2.1 従来の同期機構と SMT プロセッサにおける問題点

一番単純な同期の実現手法としては，共有メモリにロック変数を用意し，同時にただひとつの並列実行単位のみがある部分（クリティカルセクション）の実行権を持つよう管理する方式がある．

従来の多くのプロセッサでは，ロック変数の獲得を確実にひとつの並列実行単位のみが取得するために，test-and-set や fetch-and-phi などの不可分操作命令が用意されている．これを利用して，ロックを取得できるまで繰り返す手法をスピンロック（ビジーウェイト）という．

MIPS などのプロセッサでは，`ll/sc`（Load-linked / Store Conditional）命令が用意されている．`ll/sc` 命令は二つ組で使い，メモリへの読み込み/書き込み操作における原子性を保証する．具体的には `ll` 命令から `sc` 命令までの間，対象とするメモリへの他の並列実行単位による書き込みがあれば `sc` 命令が失敗する．ロック獲得処理は，ロック変数から読み出した値がすでにロックされている状態であるか，`sc` 命令が失敗した場合，ロック獲得処理を再実行，つまりスピンループを行う．この処理を MIPS アセンブラで記述したのが **図 1** である．

```
; acquire lock
  acquire addr
  ;; <critical section>
; release lock
  release addr
```

図 2 SMT-block を利用したプログラム
Fig. 2 A Program using SMT-block

SMT プロセッサは各実スレッドが計算機内資源を共有するため，ビジーウェイトを行うと，本来無駄な処理であるロック獲得処理を繰り返し行い他の実スレッドの実行を阻害してしまうため，性能に問題がある．

これを防ぐため，Intel のプロセッサでは一定期間実スレッドを停止する `pause` という命令を用意している[3]．ロック獲得のための繰り返しごとにこの命令を挿入することで，無駄な獲得処理をある程度回避できるが，一時停止の時間はどれくらいにすればよいのかという問題がある．とくに，並列実行単位が多い場合，適当な停止時間の予測は困難である．

いくつかのマルチスレッドプロセッサではメモリ[1] や共有レジスタ[4),9)] に full/empty bit を設けて同期を行う．しかし，メモリを介す方式ではメモリへのアクセス遅延が問題になる．また，共有レジスタを用いる方式ではチップ単体での効率はよいが，他のプロセッサ（またはプロセッサコア）とは同期ができないため，スケーラビリティに問題がある．

2.2 SMT-block とその問題点

このような背景のもと，Tullsen ら提案した SMT-block 方式[8] は，少量のハードウェアを追加することで軽量なロック獲得と解放を実現している．これは，`Acquire`，`Release` という命令を新たに用意し，共有メモリをロック変数として利用するロックを実現する．これを利用したプログラムを**図 2** に示す．

まず，lock box というハードウェアをプロセッサ内に用意する．lock box は実スレッド数分のエントリをもち，それぞれ（1）エントリが有効であるか，(2) どのロック変数により待っているか（データのアドレス），(3) どの命令で待ち状態になったか（命令のアドレス）という情報を格納する．lock box はチップ内の実スレッドすべてで共有する．

`Acquire` 命令は `ll/sc` 命令を利用したソフトウェアによるロック獲得処理をすべて 1 命令で行うような命令である．まず指定されたロック変数を読み込み，その値が解放されていることを示していれば，獲得

表 1 同期のために追加する命令
Table 1 New Instructions for Synchronization

命令	説明
`ll2 dr, offset(base)`	ロック獲得確認 + 従来の `ll` + ロック獲得まで休止
`sc2 sr, offset(base)`	ロック獲得確認 + 従来の `sc`
`lrls sr, offset(base)`	ロックの開放
`lrsrv sr, offset(base)`	ロック獲得予約

情報を書き込み，終了する．もしすでに他の並列実行単位にロックが獲得されていれば，その実スレッドの実行を一時停止し，パイプラインなどの計算資源を解放する．このとき，lock box に自実スレッドが停止していることと，そのロック変数のアドレス，そして `Acquire` の命令アドレスを格納する．

`Release` 命令では，そのロック変数を待っている実スレッドあるか lock box により確認し，もしあればその実スレッドの実行を再開する．もし同じロック変数で待っている実スレッドが複数あっても，その中の 1 つのみを再開する．待っているスレッドがなければ，ロック変数のアドレスに解放されたことを示す値を書き込む．

この方式の利点は次の点である．

- ロックの競合によるビジーウェイトを完全に回避するため，計算資源の効率的な利用が可能
- 共有メモリを利用した同期なのでスケーラブル
- lock box 経由でのロックの受け渡しはメモリアクセスなしで可能

しかし，`Acquire` 命令はメモリから読み込み，書き込みを 1 命令で行う必要があることや，実行中に例外が発生した場合などに実行再開アドレスを `Acquire` 命令の直前に戻す必要があるなど，複雑な命令になっているのは問題である．

また，lock box によるロック受け渡しは，ある実スレッドがロック獲得のために待ち状態になっていなければならず，実スレッド再開のオーバヘッドが問題になる．

3. 提 案 手 法

本節では SMT-block を実現する単純な小さい新しい命令を明確な意味とともに定義し，またそれを利用したさらなる高速化の工夫であるロック獲得予約について述べる．

新たに定義した命令一覧を **表 1** に示す．また，これらの命令の意味を **図 4** に擬似コードで示す．以下，これらについて述べる．

3.1 Acquire/Release に代わる新命令

`Aquire` 命令の複雑性は，要するにロードとストアを 1 命令で同時に行っていることと，例外などによる失敗を許さない仕様であることに起因する．そのため，

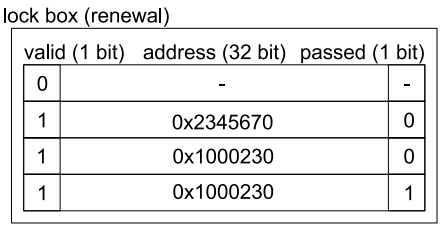

図 3 改良した lock box
Fig. 3 Renewal the lock box

より単純な命令に分割するため `ll2` と `sc2` を設けた（表 1，図 4）．`Acquire` 命令の機能を分割し，`ll/sc` 命令に挙動を追加した，というイメージである．また，lock box もこれを利用するために拡張した（**図 3**）．`Release` 命令に相当するものは `lrls` 命令と改名した．

まず，lock box に `passed` bit を設ける．これは，`lrls` 命令によって立てられる bit であり，ロックが受け渡されたことを示す．また，命令アドレスを格納する領域は不要である．

`ll2` 命令は，`ll` 命令と同様，アドレスの値を読み込み ll bit を立てる．このとき，もし読み出した値がすでに他の並列実行単位にロックが獲得されていることを示す値だった場合（たとえば 1），lock box のエントリ（以下，LBE: Lock Box Entry）にそのアドレスを格納し，一時停止する（図 4 の `sleep` 部分）．

`sc2` 命令は，`sc` と同様，ll bit を確認し，もしこれが立っていればメモリに書き込み，成功を返す（レジスタに書き込む）．そうでなければ失敗を返す．

ただし，LBE に `passed` bit が立っていた場合には，`ll2/sc2` 命令はメモリアクセスをせずに成功する（図 4 の `(*)` 部分）．`sc2` 命令終了後には，LBE を無効化する．

`lrls` 命令は，`Release` 命令と同様，すべての LBE を確認し，そのロック変数で待っている実スレッドを探す．もしあれば，その実スレッドの LBE の `passed` bit を立て，一時停止を解除する（図 4 の `wakeup_ArchitectureThread` 部分）．該当する実スレッドが複数あった場合，その中のひとつを対象とする．もしなければ，解放されたことを示す値（例えば 0）をメモリに書き込む．

`lrls` によって実行再開された実スレッドは `sc2` 命令を実行することになるが，`passed` bit が立っているため何もせずに成功することになる．

`ll2/sc2` 命令は，`ll/sc` 命令と比べ，各所に LBE を確認する機能と，一時停止状態へ遷移する機能を追加しただけなので，`Acquire` 命令に比べ単純である．

これを利用するロック獲得処理は，従来の `ll/sc` を利用したプログラム（図 1）を，それぞれ `ll2/sc2` 命令におきかえるだけでよいため，ソフトウェアの移行は容易である．

```
/* LBE: lock box entry */
instruction ll2(addr){ /* ll2 命令定義 */
  if(LBE.valid == 1 && LBE.passed == 1)
    return 0;                // success (*)
  else{
    LBE.valid = 1; LBE.addr = addr;
    ll_bit  = 1;      /* ll 命令と同じ */
    ll_addr = 1;      /* （従来動作）  */
    val = load(addr); /*           .. */
  }
  if(val == 0 || LBE.passed == 1)
    return 0; /* success */          // (*)
  else{
    sleep(timeout); /* free resouces */
    if(LBE.passed == 1)
      return 0; /* success */
    else
      return 1; /* failed */
  }
}
instruction sc2(val, addr){ /* sc2 命令定義 */
  if(LBE.valid == 1 && LBE.passed == 1){
    LBE.valid = 0; return 1; // success (*)
  } else{
    LBE.valid = 0;
    if(ll_bit == 1){  /* sc 命令と同じ */
      store(val, addr);  /*（従来動作） */
      return 1; /* success */  /*  .. */
    } else                      /*  .. */
      return 0; /* failed */   /*  .. */
  }
}
instruction lrls(addr){ /* lrls 命令定義 */
  for(i=0; i<AT_num; i++)
    if(LBE[i].valid == 1 &&
       LBE[i].addr == addr){
      LBE[i].passed = 1;
      wakeup_ArchitectureThread(i);
      return; /* finish */
    }
  store(0, addr); /* 従来動作 */
}
instruction lrsrv(addr){ /* lrsrv 命令定義 */
  LBE.valid = 1;
  LBE.addr  = addr;
}
```

図 4 設計した命令の定義の擬似コード
Fig. 4 Pseudo-code of new instructions

```
; lock acquire reservation
  lrsrv $0, (addr)
  ;; <some work>
; acquire lock (same as ll2/sc2)
retry:
  ll2  $r, (addr)
  bnez $r, retry
  li   $r, 1
  sc2  $r, (addr)
  beqz $r, retry
  ;; <critical section>
; release lock (instead of sw)
  lrls $0, (addr)
```

図 5 ロック獲得予約を利用したプログラム
Fig. 5 A Program using Lock Acquire Reservation

3.2 ロック獲得予約

lock box を介したロックの受け渡しは，ll2 命令によって一時停止している状態でしか行われない．そのため，この機会を増加するために**ロック獲得予約**という方式を提案する．

lrsrv 命令は，LBE にロック変数のアドレスを格納する．すなわち，自実スレッドがあるロックを獲得することを事前に予約する．これをロック獲得予約という．他の実スレッドが lrls すると，ロック獲得予約した実行中の実スレッドにロックが受け渡される（LBE の passed bit が立つ）．ll2/sc2 命令は，passed bit が立っているため，一切のメモリアクセスなしにロック獲得処理が終了する．

ロック獲得予約を利用したプログラムを**図 5** に示す．ll2/sc2 命令を利用したロック獲得（図 1 を ll2/sc2 にしたもの）に，事前に lrsrv を行うようにする．

本方式の制限は，そのロックを獲得する実スレッドが自実スレッドであることをプログラム上決定している場合にしか適用できない点である．誤った実スレッドがロック獲得予約を行うと，他の実スレッドのロック獲得を阻害する場合がある．

図 6 に，ロック獲得までの命令フローをまとめた．Case 1 ではまだロックが獲得されていない場合，Case 2 ではロック獲得に失敗したために一時停止し，他の実スレッドがロックを解放したので一時停止を解除し，ロックの受け渡しが行われた例である．Case 3 ではロック獲得予約によりメモリアクセスが一切発生せずにロック受け渡しが行われた例である．

3.3 例外の対処

LBE が有効だったとき，プロセス切り替えや例外などが発生した場合，その LBE を無効化する．また，ll2 命令で一時停止状態になっているときには，ll2 命令が失敗を返す．ソフトウェアは失敗が返された場合，ロック獲得処理を再開すればよく，SMT-block で

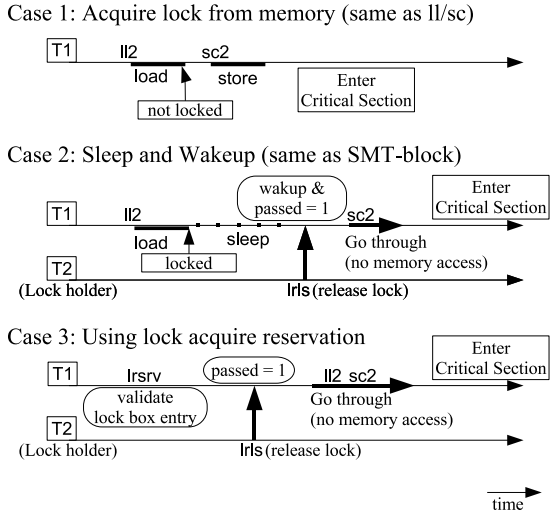

図 6 新命令によるロック獲得フロー
Fig. 6 Lock acquire flow with new instructions

必要だった命令ポインタを戻す特殊な処理は必要ない．同様の機構を利用すれば一時停止のタイムアウトを行うことが可能であり，より柔軟な同期が実現できる．

しかし，`lrls` 命令により，ある実スレッドの LBE が有効で `passed` bit が 1 である状態で例外が発生したとき☆，単純に LBE を無効化すると，ロックが受け渡されたという情報が失われてしまうため問題である．

これを解決するためには，LBE の `passed` bit が 1 で例外が発生した場合，例外処理を行う前に `lrsrv` 相当の処理を行う必要がある．ハードウェアでこれを実装することが困難な場合，OS が例外処理ルーチンの最初に `lrsrv` を行うようにすればよい☆☆．

4. 評　　価

本節では提案した同期の性能について評価する．

4.1 評価環境

評価は我々が開発している実行駆動型シミュレータ MUTHASI（MUltiTHreaded Architecture SImulator）[10] を用いた．MUTHASI は我々が研究開発している SMT プロセッサである OChiMuS PE[10] をシミュレートする．評価時のシミュレータのパラメータを**表 2** に示す．実スレッド数は 4 である．また，L1D キャッシュのアクセス遅延は 2 サイクルでありメモリアクセスごとに最低限このコストがかかる．

プログラムの作成には gcc-3.4.0（最適化オプション -O3），binutils-2.14 を評価環境用に変更したものを利用した．

☆ たとえば，`ll2` 命令で他の実スレッドに `passed` bit を 1 にされた後，`sc2` 命令が実行されるまでに例外が起こった場合．

☆☆ これを実現するためには OS に LBE を確認する方法を提供する必要がある．

表 2 シミュレーション環境
Table 2 An environment of simulation

TLP (AT#)	4
Fetch Buff.	8
Disp. Queue	16
Reorder Buff.	32
Normal RS	Simple ALU: 8, Complex ALU: 4
LD/ST RS	8 (RS: Reservation Station)
Simple ALUs	6 (0 cycle delay)
Comple ALUs	4 (Mult delay: 12, Div delay: 32)
Fetch, Decode, Dispatch, Retire Insns	4
Finish Insns	16
L1D Cache (shared)	32 byte / line, 64 lines, 2ways Access Penalty: 2 cycle
L2D Cache (shared)	64 byte / line, 512 lines, 8ways Access Penalty: 20 cycle

```
single-thread:
  for(i=0; i<N; i++)
    A[i+1] = A[i] + independent_computation()

parallelized:
  for(i=0; i<N; i+= numThreads)
    temp = independent_computation()
    LOCK(lock[thread_id])
      A[i+1] = A[i] + temp // critical section
    UNLOCK(lock[next_thread_id])
```

図 7 評価プログラム
Fig. 7 A Program for Evaluation

4.2 評価結果

図 7 に示す評価プログラムは文献[8] で評価に用いられていたものである．各繰り返しに依存があり，粗粒度並列化がしにくい例になっている．評価は `independent_computation()`（IC） の粒度を変え，各同期手法でどれだけ性能に差があるかを示す．IC は整数の足し算，掛け算，メモリアクセス一度ずつ，ある回数だけ繰り返す．この繰り返し回数が並列実行する命令流の粒度になる．

評価対象は，4 つの実スレッドを並列実行させ，同期手法をそれぞれ `ll/sc` 命令を用いたスピンループ（spin），スピンループの繰り返しごとに 100 cycle 一時停止するもの（sleep），一時停止間隔を 2^i （i: 競合した回数）にするもの（sleep back off）[5]，`ll/sc2` を用いたもの（SMT-block），ロック獲得予約を用いたもの（SMT-block LAR）にしたものである．

実行結果を逐次実行時と比較した速度向上率で示したのが**図 8** である．新命令を利用した SMT-block でも細粒度であれば性能が spin などよりも性能がいいことがわかる．SMT-block（LAR）はごく細粒度の場合 SMT-block に劣るが，ほとんどの場合よい性能を示し，SMT-block にくらべ，最大 6% 性能向上を実現できた．sleep や sleep（back off）などは性能が低く，頻繁に同期が必要なプログラムには向かないことがわかる．

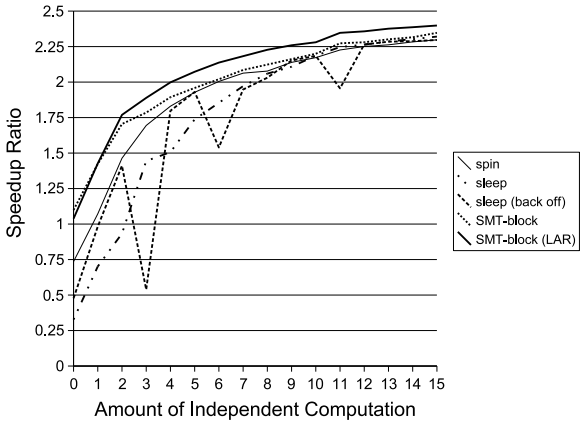

図 8 逐次実行と比較した速度向上率
Fig. 8 The spped of synchronization configurations

5. 関 連 研 究

Tullsen らは SMT-block をより効率的に実行するため，投機一時停止解除[8] を提案している．これは，`Release` 命令をリタイアする前に，一時停止している実スレッドを投機的に再開し，再開のオーバヘッドを隠蔽する．これは，本稿で提案した手法と組み合わせることが可能である．

Intel は `pause` 命令[3] のほかに `monitor/mwait` 命令による，メモリ監視による一時停止とその解除をサポートしている[2]．これを利用すれば適切な時間一時停止することができるが，メモリアクセスは同期のたびに必要なことと，同期処理自体が複雑化するため，細粒度並列処理には向かない．

保護されるクリティカルセクションを投機実行し，もしロック獲得が競合していなければロック獲得処理を省略する方式も提案されている[6] が，ロック獲得を競合しやすい細粒度並列処理では効果が薄い．

6. ま と め

本稿では SMT プロセッサにおいてスケーラブルで軽量な同期を実現する SMT-block に必要な命令を，意味を保ったまま単純な命令に分割し定義した．また，さらに軽量な同期を実現するロック獲得予約という手法を提案し，評価の結果，従来の SMT-block を利用したプログラムよりも最大 6% の性能向上を確認した．

今後は提案した同期機構を利用したマイクロベンチマーク以外のプログラムでの評価を行い，本方式の有用性を確認する．また，ロック獲得予約を活用することにより，より効率的な細粒度並列化を行うことが可能であるが，ロック獲得予約はプログラミングが難しいため，これを自動的に適用する並列化コンパイラの開発が必要である．

参 考 文 献

1) Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A. and Smith, B.: The Tera computer system, *SIGARCH Comput. Archit. News*, Vol.18, No.3, pp.1–6 (1990).
2) Boggs, D., Baktha, A., Hawkins, J., Marr, D. T., Miller, J. A., Roussel, P., Singhal, R., Toll, B. and Venkatraman, K.: The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology, *Intel Technology Journal*, Vol. 8, No. 1, pp. 1–11 (2004).
3) Dichter, C.: Optimizing for the Willamette Processor, *Intel Developer UPDATE Magazine*, pp. 1–5 (2000).
4) Keckler, S. W., Dally, W. J., Maskit, D., Carter, N. P., Chang, A. and Lee, W. S.: Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor, *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, IEEE Computer Society, pp. 306–317 (1998).
5) Mellor-Crummey, J.M. and Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Syst.*, Vol. 9, No. 1, pp. 21–65 (1991).
6) Rajwar, R. and Goodman, J. R.: Speculative lock elision: enabling highly concurrent multithreaded execution, *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 294–305 (2001).
7) Tullsen, D. M., Eggers, S. and Levy, H. M.: Simultaneous Multithreading: Maximizing On-Chip Parallelism, *Proceedings of the22th Annual International Symposium on Computer Architecture*, pp. 392–403 (1995).
8) Tullsen, D.M., Lo, J.L., Eggers, S.J. and Levy, H. M.: Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor, *HPCA '99: Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, IEEE Computer Society, pp. 54–58 (1999).
9) Yankelevsky, M. N. and Polychronopoulos, C.D.: α-coral: a multigrain, multithreaded processor architecture, *ICS '01: Proceedings of the 15th international conference on Supercomputing*, ACM Press, pp. 358–367 (2001).
10) 河原, 佐藤, 並木, 中條: システムソフトウェアとの協調を目指すオンチップマルチスレッドアーキテクチャの構想, コンピュータシステムシンポジウム, Vol. 2002, No. 18, pp. 1–8 (2002).