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

2006－ARC－170 (14)
2006／11／30

中粒度並列処理用ハードウェア同期処理機構の提案

伊賀崇幸† 佐々木敬泰†
大野和彦† 近藤利夫†

近年，マルチプロセッサ・システムが普及し，プログラムを分割して並列に実行する並列処理手法が広く用いられるようになった．中でもプログラムをスレッド等の細かい粒度に分割する中・細粒度並列処理は，実行するマルチプロセッサ・システムの形態に強く依存しない手法として注目されている．しかしながらこの手法では，プログラムを細かく分割する事によるタスク数の増加により，スケジューリングやコンテキスト・スイッチ，同期処理等に起因するオーバーヘッドが増大し，大幅な性能低下を招く事が問題となっている．この問題を低減する手法として，著者らはハードウェアでスケジューリングを高速化する SSH(Scheduling Support Hardware)，およびハードウェアでコンテキスト・スイッチ処理を行う CSS(Context switch Support System) を提案している．しかしながら，同期処理によるオーバーヘッドが依然問題となっており，これを解決する必要がある．そこで本研究では，同期処理を支援する S3(Synchronization Support System) を提案し，マルチプロセッサ・システムに実装することで，同期処理に起因するオーバーヘッドの増加を抑え更なる高速化を目指す．

Proposal of hardware Synchronization Support System for middle grain parallelism

Takayuki Iga,† Takahiro Sasaki,† Kazuhiko Ohno† and Toshio Kondo†

Today, multiprocessor systems have spread and Parallelism on the system are widely used in every usage. Particularly, Middle-fine grain parallelism, dividing program into a large number of threads, is paid attention to. Because it hardly relies on multiprocessor system environment. However, the overhead of task scheduling, context switching and synchronization process becomes large compared to the execution time of threads and it often makes performance down greatly. In order to reduce these overheads, we have proposed the Scheduling Support Hardware architecture(SSH) and Context switch Support System(CSS). SSH accelerates the performance of the OS with hardware by scheduling threads concurrently with the thread execution on the CPU. CSS saves the registers the former thread used and fetches the registers used in next thread in parallel with the thread execution on the CPU. By using them, we have obtained good results. However, the overhead of the synchronization process still causes performance down. This paper proposes the Synchronization Support System(S3) which supports synchronization process by snooping shared memory bus in parallel with the thread execution on CPU, and adds S3 to the multiprocessor system with SSH and CSS and evaluates them.

1. はじめに

近年，マルチプロセッサ・システムが普及し，プログラムをループやサブルーチン，関数といった粒度の粗い単位で分割して並列で実行する粗粒度並列処理が広く用いられるようになってきた．しかし，粗粒度並列処理ではタスクの粒度が粗いため，均等な負荷分散が難しく，実行するマルチプロセッサ・システムの形態に関わらず，性能を常に最大限引き出せるようなプログラムの記述は極めて困難である．

この問題を解決する手法の１つに，プログラムをスレッド等の細かい単位で分割して並列処理を行う中・細粒度並列処理がある．この処理方式では，プログラムをプロセッサ台数に対して圧倒的に多い数のスレッド集合に分割し，実行条件の整ったスレッドから動的に空きプロセッサに割り当てて実行することで，マルチプロセッサ・システムの形態に強く依存しない並列処理を実現出来る．しかし，この手法では粒度が細かくなる事によるスレッドのスケジューリングやコンテキスト・スイッチ，同期処理等の回数増加に起因するオーバーヘッドの増加が，性能を大幅に低下させてし

† 三重大学大学院工学研究科情報工学専攻
Graduate School of Engineering, Mei University

まう危険性がある．このため，中・細粒度の並列性を有効に利用出来ていないのが現状である．つまり，これらスケジューリングやコンテキスト・スイッチ，同期処理を効率良く実行する事で，中・細粒度並列処理の高速化の実現が可能であると考えられる．

これらの問題を低減するための手法として，著者らは SSH(Scheduling Support Hardware)[1][2] と CSS(Context switch Support System) を提案している．SSH はスケジューリングを支援するハードウェアで，CPU 上のスレッド実行とスケジューリングを並行して行う．これにより，スケジューリングにかかる時間を隠蔽している．また同様に，CSS はコンテキスト・スイッチを支援する機構で SSH と協調して動作する．CSS ではコンテキスト・スイッチにおけるレジスタの復帰／退避処理を CPU 上のスレッド実行と並行に実行する事でコンテキスト・スイッチにかかる時間を隠蔽している．これらの提案により成果は得られた．しかしながら，同期処理に起因するオーバーヘッドが依然問題となっている．そこで本研究では同期処理に着目し，同期処理を支援する S3(Synchronization Support System) を提案し，中・細粒度並列処理の更なる高速化を目指す．

以降，本論文では次のように構成される．まず，次章では関連研究として本研究と同様にハードウェアで同期処理を行う手法を挙げ，本研究との違いについて述べる．次に 3 章では，SSH と CSS について述べ，4 章で S3 の詳細なアーキテクチャ及び動作について述べる．続く 5 章では S3 の性能評価について述べ，6 章で結論と今後の展望を述べる．

2. 関連研究

2.1 Responsive Multithreaded Processor のスレッド間同期機構の設計と実装

村中らによる研究は，リアルタイム処理をハードウェアでサポートする Responsive Multithreaded Processor に，ハードウェア同期機構である Sync Unit を搭載する事で高速化を目指したものである[3]．Sync Unit による同期方式では，全スレッドから書き込める共有レジスタを使用し，そのレジスタにアクセスする専用命令および同期命令を実装している．しかしながら，共有レジスタを用いる方式ではチップ単体での効率は良いが，他のプロセッサとは同期が出来ないため，スケーラビリティに問題がある．

これに対し本研究では，想定するマルチプロセッサ・システムは 1 チップだけではなく汎用的である．また，排他制御に於いて，CPU 資源の浪費削減を図っている点は共通であるが，本研究では更にロックの獲得を自動で行うシステムの実装を目指しており，更に効率的な排他制御が可能となる．

2.2 SMT プロセッサにおける同期方式の検討

笹田らによる研究は，Tullsen らが提案した SMT-Block[4] を独自に改良し，また，それを利用したロック獲得予約という効率の良いロック受け渡し方法による高速化を目指したものである[5]．SMT-Block 方式では lock box というハードウェアをプロセッサ内に用意して，lock box を用いる複雑な専用命令を追加する．笹田らはこの命令の意味を保ったまま，命令の複雑さを解消した新命令を実装した．ロック獲得予約では，ロックを獲得する事を事前に予約する．これにより他のスレッドがロックを解放するとロック獲得予約したスレッドに効率良くロックの受け渡しが行われる．しかしながら，ロック獲得予約には受け渡しが失敗するケースがいくつか存在しており，問題となる場合がある．また，扱う同期処理は排他制御のみである．

これに対し本研究では，扱う同期処理は排他制御とバリア同期を対象としている．また，ロックの受け渡しに関しては，ロックを自動で獲得するシステムにより失敗なく，効率的に実現出来る．

3. SSH と CSS について

3.1 SSH の概要

図 1 に従来の SSH と CSS を用いたマルチプロセッサ・アーキテクチャを示す．提案するマルチプロセッサ・アーキテクチャは，複数の PE(Processor Element)，SSH-m(SSH-master)，共有メモリ (Shared Memory)，メモリ・バス調停器 (Memory Bus Arbiter)，スケジューラ・バス調停器 (Scheduler Memory Arbiter) から成る．メモリ・アーキテクチャとしては，集中あるいは分散共有メモリを想定しており，同一のアドレス空間を共有しているものとする．これは，スレッドを容易に任意の PE に割り当て可能にするためである．

PE は，プログラムの実行を行う CPU と，SSH の一部である SSH-s(SSH-slave) で構成されている．SSH-s は CPU と SSH-m のインタフェースを提供するもので，CPU の動作と並行して次に実行されるスレッドの取得，新規スレッドのキューイング等を行う．CPU と SSH-s はオンチップでの実装を想定しており，32 ビットのアドレス／データ共有のバスで接続されているものとする．また，CPU と SSH-s の通信はメモリマップド I/O により実現する．

スレッドのスケジューリングは，SSH-m で行う．

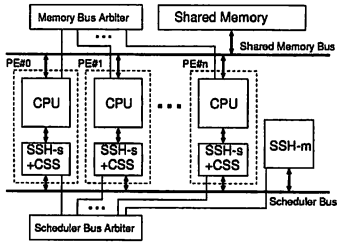

図 1 マルチプロセッサ・アーキテクチャ

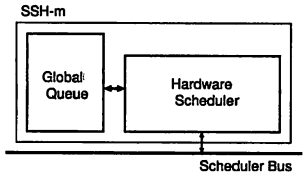

図 2 SSH-m のブロック図

SSH-m は，SSH-s を介して送信された新規スレッドを管理し，スケジューリング・ポリシに従ってスケジュールする．また，SSH-s から要求があった場合，最も優先度の高いスレッドの情報を返す．SSH-s と SSH-m は専用のスケジューラ・バスを介して通信する．

3.2 SSH の詳細

SSH の構成要素には，スケジューリング等の処理を行う SSH-m と CPU と SSH-m とのインタフェースを提供する SSH-s がある．

3.2.1 SSH-m

図 2 に SSH-m のブロック図を示す．SSH-m はスレッドのスケジューリングを行う Hardware Scheduler，及び実行可能キュー等を保持する Global Queue から成る．

3.2.2 SSH-s

図 3 に SSH-s のブロック図を示す．SSH-s では CPU と SSH-m のインタフェースを行う．SSH-s は，CPU とのインタフェースを行う CPU-SSH Interface Unit，スケジューラ・バスを介して SSH-m と通信を行う SSH-SSH Interface Unit，及び Global Queue から先行してロードしたスレッドや Global Queue に登録すべきスレッドをバッファリングするための Register Unit から成る．

3.3 CSS について

CSS は SSH で扱うスレッド管理情報にコンテキスト情報を付加する事により，コンテキスト・スイッチ時に伴うレジスタの復帰／退避処理を CPU 上のスレッド実行と並列に実行する．また，従来は主記憶で行っていたコンテキスト情報の管理をスケジューリング・バスを介して SSH-m で行う事により，メモリ・バスの使用頻度が高くなる事による性能低下を回避している．

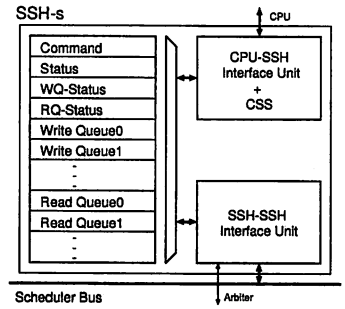

図 3 SSH-s のブロック図

3.4 SSH 及び CSS の動作

ここで，SSH 及び CSS の動作を簡単に説明するために動作の一例を示す．初期条件として，実行時間から十分に時間が経過しており，Global Queue には実行待ちのスレッドが既に存在し，また，当該 SSH-s の Read Queue 及び Write Queue 共に空であるとする．

STEP1: CPU 上ではユーザのスレッドが実行中されている．一方，SSH-s は Read Queue が空であるので，SSH-m に次に実行すべきスレッドの情報を要求する．

STEP2: SSH-m では SSH-s からの要求に従い，Global Queue から最も優先度の高いスレッドの情報をキューから取り出し，SSH-s に送信する．

STEP3: SSH-s は次に実行すべきスレッドの情報を受け取り，Read Queue に書き込む．

STEP4: タイムスライスを使い切る等，スレッドが切り替わるタイミングになると，CPU 上ではスレッド・ライブラリへと制御が移る．

STEP5: スレッド・ライブラリにより Read Queue に次に実行すべきスレッドの情報が到着しているのを確認し，また，Write Queue が空であるのも確認すると，CSS が 1 サイクルで Write Queue にレジスタ退避するのと同時に，Read Queue の情報をレジスタ復帰する．

STEP6: CPU 上ではユーザのスレッドの実行が再開される．また，SSH-s は Write Queue にデータが書き込まれた事を検知し，SSH-m へ送信する．

STEP7: SSH-m では，SSH-s より受け取ったスレッドの情報を再スケジューリングし，適切なキューへ追加する．一方，SSH-s は Read Queue が空になったのを検知し，再び SSH-m へ次に実行す

べきスレッドの情報を要求する．

SSH 及び CSS では，以上のような動作を繰り返し，処理の高速化を実現している．

4. 同期処理支援システム (S3) の提案

4.1 S3 の概要

SSH に CSS を付加することにより，スケジューリングに要する時間及びコンテキスト・スイッチに伴うレジスタの退避／復帰処理時間を隠蔽する事で処理の高速化が実現出来た．しかしながら同期処理の回数増加に起因するオーバーヘッドは依然問題として残っており，性能低下を招く原因となっている．この問題を低減するために，CPU 上でのスレッドの実行と並行して同期処理を行う事により，同期処理に要する時間を隠蔽する同期処理支援システム (S3) を提案する．

4.2 高速化の原理

本節では S3 を実装する事による高速化の原理について述べる．説明を単純化するため，ロック変数を用いる排他制御を例に挙げる．図 4 にスレッド T1，T2，T3 を 2CPU で並列に実行する様子を示す．T1 と T2 は同じロック変数を共有しており，排他的に実行する．また，図 4 の開始時点では既に，T2 がロックを獲得しているものとする．図 4 の (a) では従来手法，(b) では提案手法で実行した流れとしてそれぞれ示す．

まず，従来手法のソフトウェアでの同期処理の場合，図 4(a) に示すように，T1 はロックの獲得に失敗すると複数回ロック獲得を試みる．それでもロックを獲得出来ない場合は CPU を明け渡して次のスレッドに実行を移す．T2 が CPU に割り当てられてロックが解放された後に再び T1 が CPU に割り当てられれば，ロック獲得処理をして続きの処理を実行出来る．この方式では，ロックの獲得が出来ない場合に，獲得出来るまでの獲得失敗回数がそのまま CPU 資源の浪費となり，性能低下の原因となる．

これに対して図 4(b) に示すように提案手法では，ロックの獲得に一度失敗すると T1 はその後のロック獲得処理を S3 に委託して CPU を次のスレッドに明け渡す．S3 では，T1 からの情報を元に Shared Memory Bus を監視して，ロック変数が解放されるのを検知する．ロック解放を検知すると S3 は即座に T1 に代行してロック獲得を試みる．そして S3 がロック獲得に成功すると T1 は実行可能状態となり，CPU に割り当てられるとロック獲得処理を省略して続きの処理に実行を移せる．このように本方式では，CPU の資源浪費を削減し，効率の良いロック獲得処理を行えるため，同期処理の高速化が可能となる．

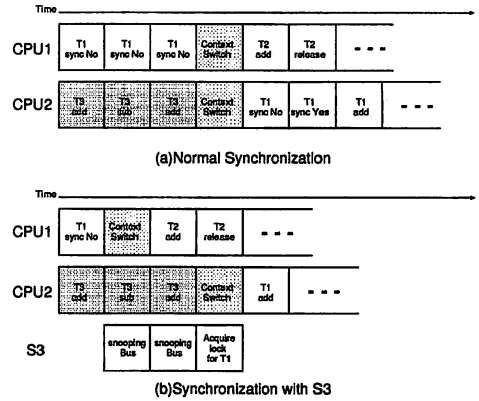

図 4 S3 の高速化の原理

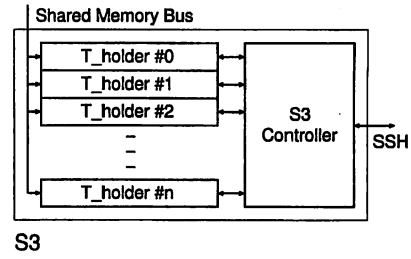

図 5 S3 の構成

4.3 S3 の構成

S3 のブロック図を図 5 に示す．S3 は T_holder と名付けたスレッド情報を保持するユニットを複数持つ構成になっている．S3 Controller は，スレッドの優先度に応じて SSH とのデータ転送を制御する機能を持つ．なお，S3 は図 2 にある SSH-m の Hardware Scheduler と，図 3 にある SSH-s の SSH-SSH Interface Unit にそれぞれ接続する形で実装する．

次に，S3 に搭載されている T_holder ユニットのブロック図を図 6 に示す．T_holder はスレッドの情報を格納する Thread info，格納しているスレッドの優先度を表す Priority，マシンの状態を表す State，ロック変数のアドレスを格納する Lock Address，Shared Memory Bus からのデータとロック変数のアドレスを比較して結果を返す Compare，そして T_holder の動作制御を行う T_holder Controller で構成される．

4.4 S3 の動作

S3 は，図 7 のステート・マシンに従って動作する．S3 は図 7 の 4 つの状態からなっている．以下に 4 つの状態について，その状態における動作と共に述べる．

Idle: S3 の初期状態．S3 がリセットされた時に，こ

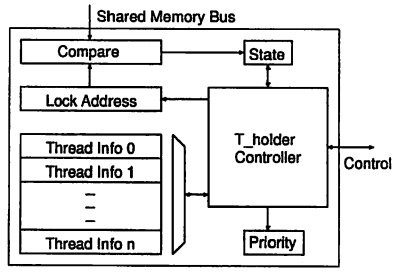

図 6 T_holder の構成

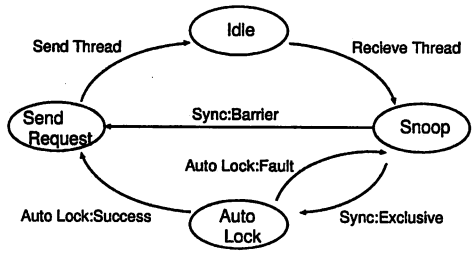

図 7 S3 のステート・マシン

の状態から動作を開始する．SSH からのメモリ・バス監視要求があり，ロック変数のアドレス等の情報を受信すると「Snoop」へと状態が遷移する．

Snoop: メモリ・バス監視状態．全 PE からのロック変数へのアクセスを監視して，ロック解放を検出すると次の状態へ遷移する．同期処理の形式によって遷移先が異なり，排他制御の場合は自動ロック獲得機能が動作する「Auto Lock」へ，バリア同期の場合は「Send Request」へと遷移する．

Auto Lock: 排他制御の場合，他スレッドによりロックが解放されても，自スレッドがロックを獲得出来るまで実行可能とならない．そのため，S3 では自動ロック獲得機能を実装している．これにより，ロック獲得のために CPU 資源を準備するためのタイムラグの削減と，再びロック獲得に失敗した場合の CPU 資源の浪費削減へと結び付く．自動ロック獲得機能でロック獲得に成功した場合は「Send Request」へ，失敗した場合は再び「Snoop」へと状態が遷移する．

Send Request: 保持スレッドの送信要求状態．SSH へとスレッドの情報を送信完了すると「Idle」へと状態が遷移して，新たなメモリ・バス監視要求がくるまで待機する．

5. 予備評価

本章では，S3 の有効性を示すために，一般的な並列処理プログラムをモデル化したものを取り上げ，ソフトウェアのみで実行する方式と S3 を用いて実行する方式との比較を行い，S3 の有効性を示す．以降，本稿では，従来通りソフトウェアのみで実装する場合を「soft-sync」と表し，S3 を用いる方式を「S3」と表す．なお，soft-sync の同期処理に要するサイクル数は文献 1) で設計した Verilog-HDL 環境を用い評価を行い，S3 については soft-sync の実行結果を解析して手計算で実行時間を見積もったものとする．

5.1 対象とするマルチプロセッサ環境

マルチプロセッサ環境は，図 1 のものを対象とする．PE は 1~ 16 台までの任意の台数に設定出来る．Memory Bus は全 PE から等距離で，アクセス時間は均一であり，また，バスの優先順位は全 PE とも等しい．Scheduler Bus も Memory Bus と同様にラウンド・ロビンにより優先順位が決定されるが，SSH-m からの要求には最も高い順位の優先度が割り当てられている．命令メモリは，共有メモリに置かれており，データ・バス，命令・バスはともに 32 ビットのアドレス／データ分離型である．

5.2 評価内容

S3 の性能評価は文献 1) で用いている並列処理モデルを用いて行った．具体的には N 個のスレッドを生成し，各スレッドは途中で 1 度バリア同期をとるモデルを用いて行う．処理に要するサイクル数の内，逐次処理部分である親スレッドのプログラムを初期化する処理と終了する処理は 0 サイクルとする．スレッド生成時間とスレッド結合時間は SSH 及びライブラリに依存するが，オリジナルから変更していないため文献 1) の実測データに従い，それぞれ 690 サイクル，841 サイクルとする．今回はスレッド数を 64 個，及び粒度を 1 スレッドあたり約 500 サイクルとし，本研究では簡単のため，命令キャッシュ及びデータキャッシュのヒット率を 100%とした．このモデルを用いて，PE 数の増加に対する処理速度を評価する．

以上の条件で，スレッドをいくつかのグループに分割する場合の比較を行う．グループ分割は具体的に，全スレッドで 1 度バリア同期をとる方式を「1G-1S」，2 グループに分けて，それぞれのグループ毎に 1 度バリア同期をとる方式を「2G-2S」，2 グループに分けて片方のグループは 1 度バリア同期をとり，もう片方のグループはバリア同期をとらずに処理を終える方式を「2G-1S」，4 グループに分けて，それぞれのグループ

毎に1度バリア同期をとる方式を「4G-4S」とする．

なお，本評価では処理の実行時間を示す指標としてクロック・サイクル数を用いる．

5.3 評価結果

S3の性能評価の結果を図8に示す．横軸はPE数，縦軸はsoft-syncに対するS3の処理速度向上率である．図8より，速度向上率はPE数の増加に伴い高くなっており，PE数が2の時で1.1倍，PE数が16の時に「1G-1S」で最低4.5倍，「4G-4S」で最高7.7倍となっている．この結果について以下に理由を述べる．

今回実装したライブラリでは，同期処理に要するサイクル数について，soft-syncでは1度目の同期失敗時の処理に384サイクル必要とする．具体的にはまず，スピンロックによりロックを獲得してバリア同期のためのカウンタをデクリメントする．この際，ロックを獲得出来るまでスピンロックは複数回試行される．次に，カウンタの値を確認して，その値がバリア同期をとれる値でない場合はロックを解放する．その後，タスク切り替えを行うための処理をしてCPUを明け渡す．2度目以降の同期失敗時の処理には，バリア同期のためのカウンタをデクリメントする処理が不必要となるので，1度目よりオーバーヘッドが少なくなり，258サイクルとなる．

これに対し，S3では1度目の同期失敗時の処理に254サイクルしか必要としない．これは，S3を利用するための準備処理とタスク切り替えを行うための処理のみとなるからである．また，S3では同期処理をCPU上では1度しか行わず，以降はS3がCPUとは独立して処理するため，オーバーヘッドが少なくなっている．つまり，2度目以降は基本的にS3のハードウェア上で同期処理を行うため，CPUから見たオーバーヘッドは0サイクルとなる．ここで，S3で管理出来るスレッド数が上限に達している場合は，1度の同期処理にsoft-syncと同等のサイクル数が必要となるが，今回は簡単のため，S3のスレッド管理数の上限は超えないものとして評価を行った．

以上から，S3ではsoft-syncより同期処理に起因するオーバーヘッドが減り処理時間が短縮出来たと考えられる．この結果から，提案方式は従来のソフトウェアを用いた同期処理方式よりも速度向上率が最大7.7倍を見込める事がわかり，提案方式の有効性を示せた．

6. 結論と今後の展望

本研究では，スケジューリング支援ハードウェア(SSH)と，コンテキスト・スイッチ支援システム(CSS)

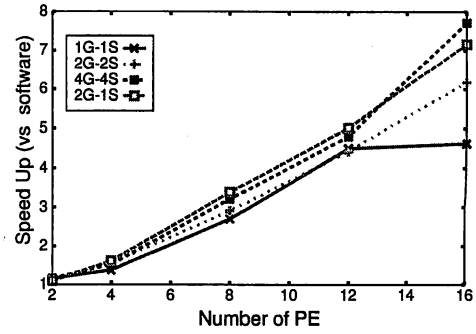

図8 PE数による同期回数

を用いたマルチプロセッサ環境に同期処理支援システム(S3)を追加することで，中・細粒度並列処理を有効に利用可能なシステムの構築を行った．また，今回行った性能評価から，S3を用いた提案方式では従来方式より約7.7倍もの処理速度の向上を見込め，提案方式の有効性を示せた．今後の展望として次の事が考えられる．(1)Verilog-HDLを用いたS3の詳細設計 (2)ベンチマークを用いたシミュレーション評価 (3)同期処理オーバーヘッドの更なる短縮化である．

参考文献

1) 佐々木敬泰, 西村直己, 弘中哲夫, 吉田典可: マルチプロセッサ用スケジューリング支援ハードウェアの提案とシミュレーション評価, 電子情報通信学会論文誌, Vol.J84-D-I, No.11, pp.1515–1531 (2001).

2) Naoki Nishimura, Takahiro Sasaki and Tetsuo Hironaka: "Prototype Microprocessor LSI with Scheduling Support Hardware for Operating System on Multiprocessor System," Asia and South Pacific Design Automation Conference 2000 (ASP-DAC2000), pp.29–30 (2000).

3) 村中延之, 伊藤務, 新井誠一, 山崎信行:Responsive Multithread Processorのスレッド間同期機構の設計と実装, 情報処理学会研究報告 2005-SLDM-119, pp.121-126(2005).

4) 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 Fifth International Symposium on High Performance Computer Architecture, IEEE Computer Society, pp.54-58(1999).

5) 笹田耕一, 佐藤未来子, 内倉要, 加藤義人, 大和仁典, 中条拓伯, 並木美太郎: SMTプロセッサにおける同期方式の検討, 情報処理学会研究報告 2005-ARC-162, pp.31-36(2005).