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

2005－ＳＬＤＭ－119 (21)
2005／3／18

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

村中　延之†　伊藤　務††　新井　誠一††　山崎　信行††

† 慶應義塾大学理工学部 情報工学科
†† 慶應義塾大学大学院 理工学研究科 開放環境科学専攻 コンピュータ科学専修
〒 223-8522 横浜市港北区日吉 3-14-1
E-mail: †muranaka@ny.ics.keio.ac.jp

あらまし　リアルタイム制御用に開発されている Responsive Multithreaded Processor においてスレッド間同期をとる際には従来，LL 命令と SC 命令のペアを使用して同期プリミティブを実装していた．しかしながらスピンロックを用いると資源を浪費し，優先度継承の際にデッドロックがおこる問題点がある．そこで本研究では，スレッド間同期をハードウェアでサポートする機構を設計，実装する．模擬的な状況における評価の結果，スレッド間同期における資源の浪費を無くすことができ，ハードウェアで最適な優先度継承が出来ることを確認した．

キーワード　Responsive Multithreaded Processor，同期機構，優先度，マルチスレッド

Design and implementation of the synchronous mechanism among threads on Responsive Multithreaded Processor

Nobuyuki MURANAKA†, Tsutomu ITOU††, Seiichi ARAI††, and Nobuyuki YAMASAKI††

† Department of Information and Computer Science, Faculty of Science and Technology, Keio University
†† Department of Computer Science, Graduate School of Science and Technology, Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama 223-8522, JAPAN
E-mail: †muranaka@ny.ics.keio.ac.jp

Abstract　In Responsive Multithreaded Processor developed for a real-time control primitive to take the synchronization among threads has mounted by using the pair of the LL instruction and SC instruction. However,when the spin lock is used, the resource is wasted and there is a problem in which the deadlock is absorbed when priority is inheritanced. Then, in this papar we design and implement the mechanism to which support the synchronization among threads with hardware. It was confiemed that the waste of the resource is able to be lost as and the priority is able to controled best with hardware as a result of the evaluation.

Key words　Responsive Multithreaded Processor, synchronization mechanism, priority, multithread

1. はじめに

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(RMTP) [1] [2] [3] の特徴として，8way の Simultaneous Multithreading(SMT) [4] [5] に 256 レベルの優先度を設定できる．

マルチスレッディングで複数のスレッドを同時実行するにおいて，スケジューリング上でスレッド間の依存関係が生じる場合や，複数スレッドで 1 つのプログラムを実行する場合に同期をとるプリミティブが不可欠となる．

RMTP においてソフトウェアで同期プリミティブを実装する場合には，Load-Linked(LL) 命令と Store-Conditional(SC) 命令のペアを用いて作成することが多い [6]．しかしこの方式では，ロックの獲得の際に獲得に成功するまで命令がループしてしまうため，特に高優先度スレッドがロック獲得命令を失敗し続けた場合，無駄に実行資源が浪費され，他スレッドの実行に影響を与えてしまう．

これに対して，同期に失敗したスレッドは同期が取れるまでストールさせる機構をハードウェアでサポートすることで，資源の浪費を防ぐ．

また，優先度を扱う際の問題として，低優先度スレッドが高優先度スレッドをブロックする優先度逆転問題が発生ことがある．この対策として，同期に際して優先度を継承するという方法がよく用いられる．しかしコンテキストスイッチが多数発生

すると，どのスレッドがアクティブになるかを静的に把握することは難しく，優先度を考慮したスケジューリングは難しい．そこで同期を取る際にアクティブスレッドの状況に応じた動的な優先度継承をハードウェアでサポートする．

本研究では以上を踏まえた RMTP におけるスレッド間同期を，ハードウェアでサポートする同期機構の設計，実装を行う．

2. 背　　景

2.1 リアルタイムシステム

リアルタイムシステムとは，それぞれの処理に応じて設定される制限時間内 (デッドライン) に処理を終了させることの出来るシステムのことを言う．リアルタイムは大きく別けてハードリアルタイムとソフトリアルタイムにわけられる．ハードリアルタイムは処理がデッドラインを越えてしまうと処理の結果が役に立たず，システム全体に深刻な打撃を与えてしまう処理のことを指す．対してソフトリアルタイムは，処理がデッドラインを越えてしまっても即座に価値が無くなるわけではないが，時間経過とともに価値が減少していく処理を指す．よってソフトリアルタイムでは低確率なミスは容認される場合もある．

2.2 RMTP

RMTP はリアルタイム処理をハードウェアレベルで支援する RMT PU をプロセッシングコアに持ち，リアルタイム通信機構 (Responsive Link)，コンピュータ用周辺機器 (PCI64 I/F, IEEE1394 I/F, DDR SDRAM I/F, DMA コントローラ等)，各種周辺機器 (PWM ジェネレータ，パルスカウンタ等) を 1 つのチップに集積した System on Chip (SoC) である．プロセッシングコアである RMT PU は細粒度マルチスレッディングに加え，各スレッド毎に 256 レベルの優先度を設定できる．

RMTP の特徴は表 1 にまとめる．ブロック図は図 1 のようになる．

表 1　RMTP のパラメータ

スレッド数	アクティブ 8，バックアップ 32
命令フェッチ数	8inst/clock
命令発行数	4inst/clock
命令 commit 数	4inst/clock
functional units	4 integer 1 complex integer 1 64bit integer 2 floating point 1 complex floating point 2 branch 1 memory access 1 synchronization unit 1 SIMD 1 vector int 1 vector FP
パイプライン	13 ステージ
レジスタ	integer 32bit×32entry×8thread FP 64bit×32entry×8thread shared 64bit×32entry
優先度	256 段階

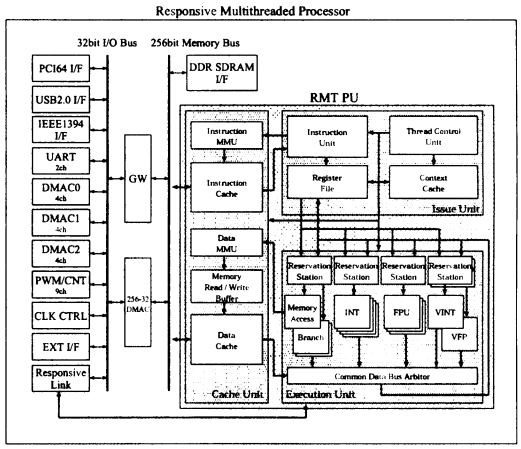

図 1　RMTP のブロック図

2.3 資源共有プロトコル

リアルタイムスケジューリングアルゴリズムとして代表的なものに Rate Monotonic(RM)，Earliest Deadline First(EDF) がある [7]．しかし，これらのアルゴリズムでは複数のタスク間で共有される資源については考慮されていない．ここで問題となるのが共有資源に対するアクセスの競合により，高優先度タスクが低優先度タスクにより実行をブロックされる状態である．また，このような状態のうち，ブロッキング時間に関する上限値が既知でない場合を特に優先度逆転問題という [8]．

この問題に対し，ソフトウェアでの代表的な対策として，Priority Ingeritance Protocol [8] と Priority Ceiling Protocol [8] がある．

2.3.1 Priority Inheritance Protocol

Priority Inheritance Protocol は優先度逆転問題を解決する代表的な資源共有 Protocol である．基本的な考え方は以下の通りである．

- あるタスク T_1 がクリティカルセクション内を実行中により高い優先度をもつタスク T_2 をブロックしたとき，クリティカルセクション内を実行中のタスク T_1 はブロックされたタスク T_2 の優先度を継承する．ただしクリティカルセクションが終了すると，T_1 は元の優先度にもどる．
- ブロックされているタスク T_2 がさらに高優先度のタスク T_3 をブロックした場合，T_2 は T_3 の優先度を継承する．また，クリティカルセクションを実行中の T_1 は T_2 を通して T_3 の優先度を引き継ぐ．

Priority Inheritance Protocol は事前に優先度の情報が無くとも動的に実行でき，汎用性が高いが，1 スレッドが複数のロックを獲得する場合 (入れ子の深さが大きい場合) の制御が難しく，制御不能な優先度逆転問題やデッドロックが生じる可能性がある．そのため，主に簡単かつブロック時間の短いロックに使用される．

2.3.2 Priority Ceiling Protocol

Priority Ceiling Protocol の基本的な考え方は以下の通りで

ある．

- 各ロックに対して，そのロックを獲得を使用する全タスクの最高優先度を優先度シーリングとして設定する．
- あるタスクがロックを獲得しようとするとき，そのタスクの優先度が他のタスクにより保持されている全ロックの優先度シーリングより高い場合のみ，ロックを獲得しクリティカルセクションへ入る．
- クリティカルセクションを実行中のタスクがより高優先度のタスクをブロックするとき，ブロックされた全タスクの最高優先度 (優先度シーリング) をブロックしたタスクが継承する．クリティカルセクションが終了するときにタスクは元の優先度にもどる．

Priority Ceiling Protocol では，Priority Ingeritance Protocol に存在する制御不可能な優先度逆転問題やデッドロックを解決できる．しかし，あらかじめ各ロックを使用するタスクの優先度が分かっていないといけないため，ある程度アプリケーションに特化したプロトコルといえる．

3. 設計，実装

本研究では以上の 2 つのプロトコルの考え方をベースに，ハードウェアで動的に優先度の変更を行う機構を設計，提案する．

ハードウェアで優先度を制御することの利点として，動的に優先度を把握して，それに応じた優先度の変更を行う回路を設計することは比較的容易に出来る．ハードウェアで同期の際に動的に優先度を変更させることで，ソフトウェアの繁雑さを軽減させ，優先度を考慮に入れたスケジューリングをサポートできると考えられる．

3.1 同期方法

RMTP における同期機構については以前にも議論されている [10]．この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て同期を取る．共有レジスタを別に用意するとそれだけ面積は増加する．しかし共有レジスタは同期命令でのみアクセスできるため，他スレッドとの依存は少ない．またメモリを使用するよりもアクセスレイテンシが短く，細粒度の同期が取れるメリットがある．

そこで，本研究でも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機構を設計する．共有レジスタはデータを持つだけでなく，ロックやそれに準じるアクセス状況を示す info bit を持つ．共有レジスタはのフォーマットを図 2 に示す．共有レジスタは 32 個である．これは 32bit の命令列のうち，レジスタ指定は 5bit であるためである．Sync State Buffer のフォーマットは図 3 に示す．Sync State Buffer は各スレッド毎に 2 つのエントリを持つ．1 エントリの場合，1 回同期命令がリザベーションステーションから出されると，その命令がコミットされるまで次の同期命令を出せないからである．

本同期機構の概念は，図 4 のようになる．同期命令は他の命令と同様にリザベーションステーションから出され，独自の命令実行機構へ入る．命令の情報は Sync State Buffer へ書き込まれ，命令が終了するまで保持される．同期命令は必ずライトバックされ，Common Data Bus へデータを送る．

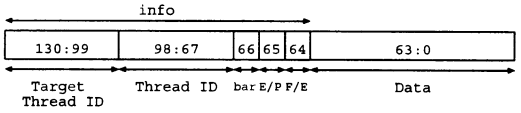

図 2 共有レジスタ

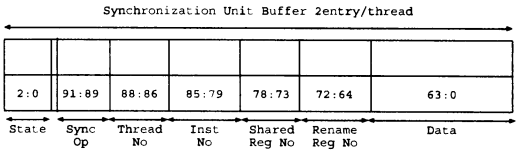

図 3 Sync State Buffer

共有レジスタと Sync State Buffer の情報から同期を取る場合を判断し，それが必要な場合はスレッドコントロールとリオーダーバッファと情報をやりとりし，コミット，フェッチ，プログラムカウンタを制御することで同期を達成する．

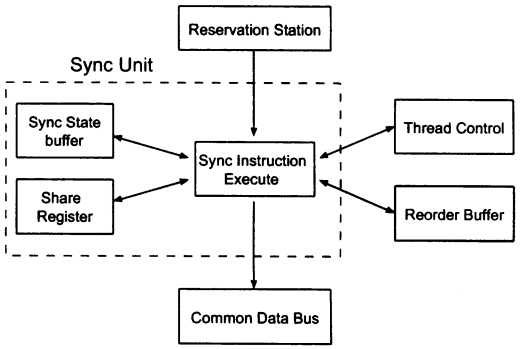

図 4 同期機構の概念

3.2 同期命令セット

共有レジスタにアクセスすることで同期を取る命令として，以下のものを実装した．

- Exclusive 命令

共有レジスタの値を読み込むとともにロックをかけ，書き込むとともにロックを開放する．レジスタにロックがかかっている間は，ロックを解除する命令以外によるアクセスを拒否する．

- Producer-Consumer 命令

2 スレッド間で Pear-To-Pear で同期を取る命令である．Producer 命令でレジスタに値が書き込まれるとともにロックをかけ，同期対象スレッドを指定する．かけられたロックは，指定されたスレッドの Consumer 命令でのみ開放される．

- Barrier 命令

バリア同期をハードウェアでサポートする．この命令セットは Barrier 命令と，バリアに参加するスレッドグループを指定する Pre Barrier 命令からなる．

Barrier 命令はバリア同期を取るスレッドを指定し，指定数分のスレッドが当命令を実行するまで，先に当命令を実行した

スレッドは同期命令失敗時と同様の動作を行う．

Pre Barrier 命令は，バリアに使用するレジスタを宣言することにより，実行したスレッドがそのレジスタを使用するバリアグループであることを宣言する．

- 単純アクセス命令

ロックをかけず，データのみを書き込む命令である．メモリを介してデータを受け渡すよりもアクセスレイテンシが短い利点がある．

3.3 同期失敗時の操作

同期命令の状況により，同期待ちのスレッドが出現する場合がある．同期状況は基本的に Sync State Buffer の状態により行う．Sync State Buffer の状態遷移は図 5 のようになる．

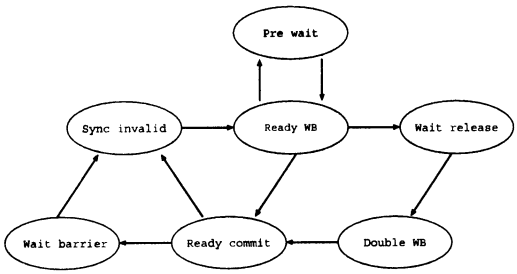

図 5 Sync State Buffer の状態遷移

初期状態は Sync invalid である．同期命令の情報が書き込まれるとただちにライトバックの準備がされ，Ready WB へ遷移する．ただしレジスタの競合が生じた場合は Pre wait へ遷移し，先行命令の終了を待つ．この時に共有レジスタの値などから同期待ちの状況が生じた場合，Wait release へ遷移し同期待ちの状態となる．そうでない場合は通常どおりライトバックされ，Ready commit へ遷移して命令がコミットされるのを待つ．またバリア命令において命令がコミットされた場合，指定したバリアスレッド数に満たない場合は，Wait barrier へ遷移して指定数分のスレッドがバリアに到達するまで同期待ちの状態となる．Sync State Buffer はこのように命令が入ったときから終了するまで命令の情報を保持する．また，ライトバックや同期待ちから復帰するスレッド候補が複数あった場合は，優先度を元に選択する．

ここで重要となるのが，同期待ちのスレッドが資源浪費を抑制するように設計を行うと言うことである．代表的な同期手段であるスピンロックは，ロック獲得に成功するまで命令をループする必要があるため，資源の浪費が生じる．

本方式では，同期待ちスレッドの操作は以下のような流れで行われる．この方法では同期待ちスレッドは資源を浪費することが無い．

（1） リオーダーバッファに最後にコミットされる命令の情報を流す．また，スレッドコントロールユニットにストールする情報を流す．

（2） スレッドコントロールユニットの方で，ストールするスレッドのフェッチを止める．

（3） リオーダーバッファが最後の命令がコミットされたのを確認し，パイプラインをフラッシュする．ただし，同期命令の情報は保持する．

このような方式を取ることで，ストールするスレッドは一切の実行資源を浪費しない．

また，ストールから復帰する場合は復帰スレッドの情報をスレッドコントロールに復帰のシグナルを送ることで，実行再開が可能となる．

3.4 同期の際の優先度継承

本機構では同期の際に必要に応じて優先度を制御する．

スレッドコントロールユニットには各スレッドごとに情報を表すスレッドテーブルがある (図 6)．スレッドの優先度もスレッドテーブルに含まれている．Sync Unit から同期の状況を確認し，それに応じてスレッドテーブルの優先度を一時的に置き換える．

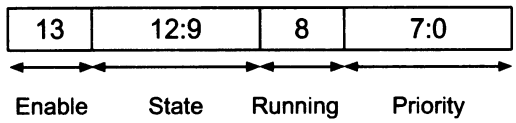

図 6 スレッドテーブル

具体的な制御方法は使用する同期命令により異なる．

- Exclusive アクセス命令

ロックを獲得しているスレッドと，ブロックされているスレッドのうち，最高優先度を求め，その優先度をロック獲得スレッドに一時的に適用する．ロック開放時に変更した優先度は元に戻す．

- Producer-Consumer 命令

Consumer 命令が Producer 命令よりも先に行われた場合，Consumer 命令で指定する同期対象スレッドがアクティブかつ優先度が低い場合，そのスレッドに Consumer 命令の優先度を適用する．ロック開放時に変更した優先度は元に戻す．

- Barrier 命令

あらかじめ専用命令によりバリア命令を行うスレッドグループを宣言しておく必要がある．そのスレッドグループのうち，1 スレッドでもバリア到達したスレッドが発生した場合，そのスレッドグループの最高優先度を求め，その優先度をバリアに参加する全スレッドに適用する．バリア開放とともに優先度は元に戻す．

Exclusive 命令の場合は，Priority Inheritance の考え方を使用したもので，中優先度タスクが高優先度タスクをブロックするといった優先度逆転問題を回避できる．

Producer-Consumer 命令では，必ず同期をとるスレッド数は 2 つなので，上記の機構で優先度逆転問題は解決できる．入れ子レベルの高いロックの獲得は出来ないが，2 スレッド間の Peer-to-Peer の制御が可能であり，シンプルな制御に向く．

Barrier 命令を使用する場合，バリアに参加するスレッドグループの宣言を行っていれば，優先度逆転問題を完全に解決す

ることが出来る．上記の機構により最悪バリア待ち時間を，バリアグループの最高優先度スレッドのセクション実行時間内に抑えることが出来る．

4. 評　　価

4.1 評価環境

RMTP 上でプログラムを実行することにより評価を行う．評価は NC-Verilog を用いて RTL シミュレーションによって行った．

4.2 資源浪費の比較

同期の際に資源浪費の問題がある．ここではスピンロックを対象比較とし，本方式との同期待ちの際に浪費する資源の比較を行う．スピンロックによる同期は，主にクリティカルセクション実行時間の比較的短い場合においてよく用いられる．

クリティカルセクションの長い命令に対してはセマフォを用いられることが多い．しかしセマフォは同期待ちスレッドをバックアップして待ち行列を作成する性質の違いがあるため，本方式と一概に評価出来ない．

以下のようなスレッドからなるプログラムを実行し，評価をとる．

- クリティカルセクション実行に見立てたスレッド

要素数 64 のバブルソートを実行する．

- 待ちスレッド

クリティカルセクション実行スレッドによりブロックされているスレッド．

待ちスレッド数を 1 から 7 まで変化させて，またクリティカルセクションを実行するスレッドの優先度を最高と最低の 2 パターンで，クリティカルセクション実行時間の評価をとる．

クリティカルセクション実行スレッドが最高優先度の場合の実行結果を図 7 に示す．縦軸はクリティカルセクションを終了するまでの時間，横軸は，待ちスレッドの数である．

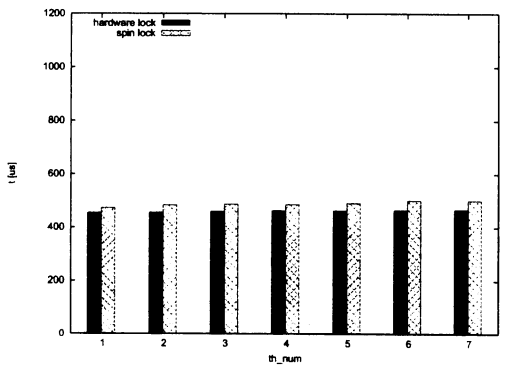

図 7　最高優先度スレッドの資源浪費の影響

スピンロックと本方式の実行時間差は 7%程度であり，低優先度のスレッドがロック獲得命令を失敗し続けたとしても，クリティカルセクション実行に影響は少ないと言える．

クリティカルセクション実行スレッドが最低優先度の場合の実行時間を図 8 に示す．

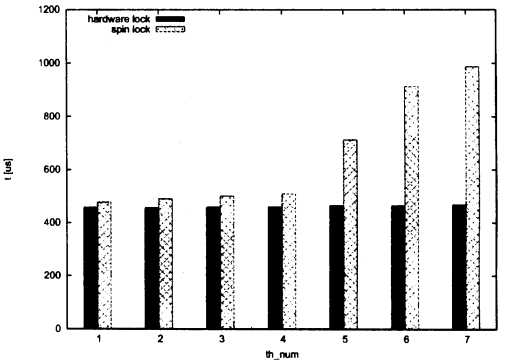

図 8　最低優先度スレッドの資源浪費の影響

待ちスレッド数が 4 以内の場合の実行時間は 7%程度の増加であるが，待ちスレッドが 5 以上になると 50%以上の実行時間の増加が見られる．このことから，スピンロックの場合は高優先度スレッドが複数ロックを獲得しようとすると，同期にかかるオーバーヘッドにより，他スレッドに悪影響を与えることが分かる．本方式では，資源の浪費を抑え，同期のオーバーヘッドを抑えることが出来る．

4.3 ハードウェアによる優先度制御の評価

優先度逆転問題が起こる場合は，例えば次のような状況がある．

各タスクに対し，同期ポイントまでの終了時間を見越して優先度を設定したが，高優先度タスクが低優先度タスクよりも早く終了した．この時間，中間優先度タスクが存在する場合，低優先度タスクを横取りされ，高優先度タスクのブロッキング時間が長くなるという場合である．

このような状況において優先度継承等の対策が有効となる．ここでは，ハードウェアで優先度継承をする場合としないばあいとで最高優先度スレッドのブロッキング時間の差を比較する．

具体的に以下のようなスレッドからなるプログラムを使用して，評価を行う．

- 高優先度スレッド

プログラム開始時に低優先度スレッドにブロックされており，低優先度スレッドと同期が取れるのを待つ．

- 低優先度スレッド

あらかじめロックを獲得しており，16 × 16 の行列計算をした後にロックを解放する

- 中間優先度スレッド

スレッド数を変化させて，どの程度低優先度スレッドの実行を横取りするか，高優先度スレッドのブロッキング時間が長くなるかを評価する．なお，ここでは 16 × 16 の行列計算をする．

優先度をまったく変更しなかった場合の実行結果を図 9 に示す．

中間優先度スレッドが，クリティカルセクション実行中の低

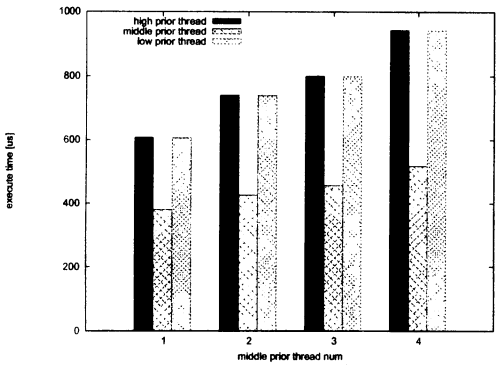

図 9　優先度を変更しない場合の実行時間

優先度スレッドの実行を横取りすることにより，高優先度スレッドのブロッキング時間が長くなっていることが分かる．その影響は中間優先度スレッド数が増加するにつれ，増している．

次に，ハードウェアで優先度を変更させた場合の実行結果を図 10 に示す．低優先度スレッドが高優先度を引き継ぐため，クリティカルセクション実行を中間優先度スレッドに横取りされることはない．その結果，高優先度スレッドのブロッキング時間を，高優先度スレッドのクリティカルセクション実行時間以下に抑えることが可能となる．以上から，この場合における優先度逆転問題をハードウェアで解決することが可能であることが言える．

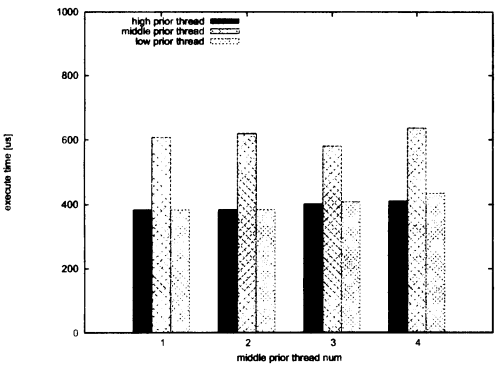

図 10　優先度をハードウェアで変更した場合の実行時間

4.4　論理合成結果

表 2 に同期命令実行ユニットを論理合成した結果を示す．以下が面積の増加になる．

表 2　論理合成結果

面積	453,042um^2
遅延時間	9.31ns

この面積が純粋な増加分となる．RMTP のダイサイズは 1cm 平方で $10^8 um$ であるので，増加面積は 0.45%となる．本実装では各スレッドごとに用意する State buffer と，共有レジスタがクリティカルになったと考えられる．遅延時間は 9.31ns だが，単純に計算すると 107MHz 以上で動作させることは難しいことが言える．これに対しては改善の余地があると言える．

5. ま と め

本研究では，RMTP にハードウェアでスレッド間同期を取る機構と同期プリミティブを設計，実装した．また，ハードウェアでの優先度継承もサポートした．

従来のソフトウェアでの同期ではロックの獲得を失敗し続けると，資源の浪費が起こり，低優先度スレッドに悪影響を及ぼすことがあったが，同期待ちのスレッドをストールさせる本方式では，資源の浪費を防ぐことが可能であることを示した．

また，アクティブスレッドの状況に応じて動的に優先度を変更することにより，ハードウェアでも最適な優先度制御が出来ることを示した．

謝辞　本論文の研究は，文部科学省の科学技術振興調整費の支援による．また本研究の一部は，科学技術振興機構 CREST の支援による．

文　献

[1] 山崎, 安西：“人間支援のための分散リアルタイムネットワーク基板技術プロジェクトから：rmt processor の紹介”, 第 43 巻, pp. 487–491 (2004).

[2] 薄井, 内山, 伊藤, 山崎：“Responsive multithreaded processor の命令供給機構”, 情報処理学会論文誌コンピューティングシステム, 第 45 巻, pp. 105–118 (2004).

[3] 伊藤, 山崎：“Responsive multithreaded processor の命令実行機構”, 情報処理学会論文誌コンピューティングシステム, 第 44 巻, pp. 226–235 (2003).

[4] D.M.Tullsen, S.J.Eggers and H.M.Levy: “Simultaneous multithreading: Maximizing on-chip parallelism”, The 22nd Annual International Symposium on Computer Architecture, pp. 392–403 (1995).

[5] D.M.Tullsen, S.J.Eggers, J.S.Emer, H.M.Levy, J.L.Lo and R.L.Stamm: “Exploiting choice: Instruction fetch and issue on an implementable silultaneous multithreading processor”, 23rd Annual International Symposium on Computer Architecture, pp. 191–202 (1996).

[6] J. L.Hennessy and D. A.Patterson: “Computer Architecture A Quantitative Approach”, Morgan Kaufmann Publishers, 3rd edition (2003).

[7] C.L.Liu and J.W.Layland: “Scheduling algorithms for multiprogramming in a hard read-time environment”, J.ACM, Vol. 20, pp. 46–61 (1973).

[8] L.Sha, R.Rajumar and J.Lehoczky: “Priority inheritance protocols: An aproach to real-time synchronization”, IEEE Transactions on Comupters, Vol. 39, pp. 1175–1185 (1990).

[9] D.M.Tullsen, J. L.Lo, S. J.Eggers and H. M.Levy: “Supporting fine-grained synchronization on a simultaneous multithreading processor”, Technical Report C98-587 (1998).

[10] 薄井, 内山, 伊藤, 山崎：“Responsive multithreaded processor の同期機構のの設計と実装”, 情報処理学会研究報告 2003-ARC-145, pp. 37–42 (2003).