計算機アーキテクチャ 134−29
（1999. 8. 4）

*MUSCAT*における混在スレッド実行方式の検討

大澤 拓[††] 酒井 淳嗣[†] 鳥居 淳[†]
伊藤 義行[†] 井上 俊明[††] 松下 智[†]
西 直樹[†] 枝廣 正人[†]

複数のスレッド実行方式を階層的に適用することによって各方式の利点を引き出すことを目的とした混在スレッド実行方式を提案する．スレッド・モデルに制約を加えることによって細粒度スレッド処理（制御並列実行方式）を可能にした*MUSCAT*へ本方式の適用を検討した．*MUSCAT*へ少量のハードウェアを追加することにより従来スレッド実行方式を実現し，制御並列方式と動的に切替えることが本方式の特徴である．本方式をmpeg2ソフトウェア・デコーダに適用したところ，制御並列実行方式，従来スレッド実行方式単独の場合よりも良好な，4PE時3.26倍の性能向上率を得た．

Blended multi-thread processing for *MUSCAT*

Taku Ohsawa, Junji Sakai, Sunao Torii, Yoshiyuki Ito,
Toshiaki Inoue, Satoshi Matsushita, Naoki Nishi
and Masato Edahiro

We developed a multi-threaded processing, called *blended multi-thread*, that incorporates different types of multi-threading model in a hierarchical way. By blended, we intended that the model combines the advantages of each type of multi-threaded processing. We examine how to apply this model to *MUSCAT*, which supports a fine-grain multi-threaded processing (called *control-parallel*). Our blended multi-thread can be implemented on *MUSCAT* by adding small hardware. Applying this method to an MPEG2 decoder program, we can achieve a 3.26-fold speedup with 4PEs, which exceeds that of either applying control-parallel or conventional multi-thread alone.

1. はじめに

命令レベル並列性の限界とトランジスタ集積度の飛躍的な向上という背景から，我々はオンチップ・マルチプロセッサ*VIRC*[1)2)]，*MUSCAT*[3)]を提案し，評価を行ってきた．

*VIRC*では“Orderedスレッド・モデル”を導入し，ハードウェアによってメモリ管理，スレッド・スケジューリングを行うことにより細粒度スレッド処理を可能にしている．しかしながら，スレッド・モデルに制約を加えたにも関わらず，ソースレベルの並列化が必要，スケジューリングに自由度がない，データ依存性の強いアプリケーションに有効ではないなどの問題があった．

一方，*MUSCAT*ではOrderedスレッド・モデルに更に厳しい制約を加えた“フォーク1回モデル”を導入している．*MUSCAT*ではハードウェアでスレッド管理，レジスタ継承，データ依存解消，投機実行などを実現することによって，並列化の適用範囲を広げるとともに，コンパイラによる自動並列化を現実的にした[4)]．同様のアーキテクチャとしてSKY[5)]やSuperthread[9)]なども提案されている．

しかしながら，以上のようなスレッド・モデルへの制約は並列性の抽出能力という点で少なからぬ制約を与える．特に，*MUSCAT*やSuperthreadに特有な問題点として，コンテクストが大きい，スレッド粒度が不揃いである場合に性能が低下するなどの問題も挙げられる．

他方，従来のSMP(*Symmetric Multi-Processor*)におけるスレッド・モデル（以下，従来スレッド・モデルと呼ぶ）は動的スケジューリングにより並列性を最大限に抽出可能という長所を持つ．しかしながら，動的スケジューリング故の性能もしくはハードウェア・オーバヘッドは大きい．

そのため我々は，複数のスレッド実行方式を階層的に適用することによって各方式の利点を引き出す処理方式として混在スレッド実行方式を提案する．

以下，各スレッド実行方式の長所短所についてまとめた後，提案する混在スレッド実行方式について述べ

† NEC C&Cメディア研究所
C&C Media Laboratories, NEC Corporation
†† NEC シリコンシステム研究所
Silicon System Laboratories, NEC Corporation

(2章)，本方式の*MUSCAT*への適用方法について検討した後(3章)，本方式をmpeg2ソフトウェア・デコーダに適用した結果を示し(4章)，本論文をまとめる(5章)．

2. 各スレッド実行方式の概要

本節では，各スレッド・モデルの利点，欠点に関して議論するとともに，混在スレッド実行方式の概要について述べる．なお従来スレッド・モデルとOrderedスレッド・モデルの比較に関しては[1]に詳しい．

2.1 準　　備

議論に先立ち，本節で用いる用語を整理する．

スレッド・モデル： スレッド間の論理的な相関関係．下記並列性抽出能力の上限を一意に定める．

スレッド実行方式： あるスレッド・モデルを実行するためのスケジューリング方式(複数存在し得る)

並列性抽出能力： 下記オーバヘッドを0としたときに，並列度をどれだけ引き出せるかの値．スレッド実行方式によって一意に定まる．

オーバヘッド： スレッド実行方式によって一意に定まる，スレッド実行(生成，同期，スケジューリングなど)に際するコスト．具体的にはハードウェア，ソフトウェアによる遅延やハードウェア・コスト(これらはトレードオフ関係にある)．

2.2 従来スレッド実行方式

従来のSMPなどで用いられてきた従来スレッド・モデルでは，スレッド・モデルに制約はなく，スレッドのスケジューリングは任意である．多くの実装では実行可能となったスレッドをキュー(スレッド・ランキュー)に挿入し，スケジューラがキューからスレッドを取り出し，PE(*Processor Element*)に割り当て実行する．このようなスケジューリングの任意性は，

長所 2.2.1 最大の並列性抽出能力

を可能にする．しかしながら，

短所 2.2.1 スレッド実行に伴うオーバヘッドは大きい．

2.3 Orderedスレッド実行方式

Orderedスレッド・モデルは，スレッド・スケジューリングに仮想制御フローによる制約を加えたモデルである．ここで，仮想制御フローとはそれに従えば逐次実行が常に可能，即ち必ず同期が成功するフローである．同期方向を単一方向に限定することにより，以下の利点を得ることができる．

長所 2.3.1 スケジューリング，メモリ管理が容易

長所 2.3.2 同期ミスを起こさないスレッドの逐次実行が可能

*VIRC*ではこれらの利点を利用して，スレッド管理，同期などのハードウェア・サポートを可能にしている．しかしながらこのようなスケジューリングの制約は，

短所 2.3.1 並列性抽出能力に制限を加えてしまう．

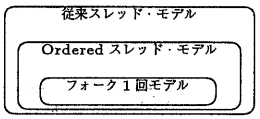

図1 各スレッド・モデルの関係

2.4 制御並列実行方式

Orderedスレッド・モデルにさらに“各スレッドは高々1回しかスレッドを生成しない”という制約を課したのが，フォーク1回モデルである．*MUSCAT*，SKY，Superthreadなどではフォーク1回モデルを導入し，同期ミスによるスレッド切替えを行わない(ウェイトする)ことによって，利点2.3.2に加え，

長所 2.4.1 スケジューリング不要

にした．さらに，この利点を利用しスレッド実行フローを一意にハードウェアにマッピングすることによって，

長所 2.4.2 スレッド生成コストの削減

を可能にしている．具体的には，*MUSCAT*では隣接PEにのみスレッド生成を行うよう制約することによって，ハードウェアでスレッド管理，レジスタ継承を実現し，基本ブロック・レベルの非常に細粒度なマルチスレッド処理を可能にしている．さらに，*MUSCAT*ではスレッドの投機実行によって，並列処理の適用範囲を広げるとともにコンパイラによる自動並列化を現実的にしている．しかしながら，このような制約を加えることは無論

短所 2.4.1 さらなる並列性抽出能力の制限(同期ミスによる計算リソースの浪費など)

を生む．また，同期方向の制限に加えて階層的スレッド生成が行えないことは

短所 2.4.2 プログラミング・インタフェースとしてユーザに見せずらい

という欠点も考えられる．上記に関しては言語上の問題にもかかわるが，少くとも現在の手続き呼びだし型言語にはなじみにくいと考える．また，上記のスレッド実行フローのハードウェアへのマッピングは以下のような欠点も持ち合わせる．

短所 2.4.3 スレッド粒度が不揃いである場合の性能低下(3.1.2節に詳細を示す)

2.5 混在スレッド実行方式

これまで述べた各スレッド・モデルは，各々に加えられた制約関係から図1のように包含できる．この包含関係から，あるスレッド・モデルにおける個々のスレッドの内部を，それが内包しているスレッド・モデルによってスレッド化することが可能であるといえる．なぜならば，スレッドtの内部でスレッド化された各スレッドの制約関係はt内に閉じており，t外への制約関係を持たないことを保証可能だからである．例えば個々の従来スレッドは内部をOrderedスレッド化可能である．このような階層的なスレッド・モデルを本稿では混在スレッド・モデルと呼ぶこととする．

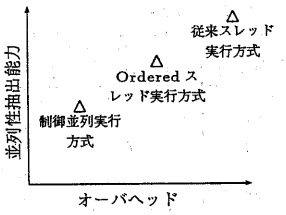

図2 各スレッド実行方式の比較

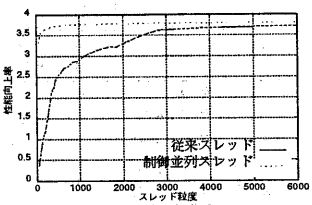

図3 スレッドの粒度と性能向上率の関係

一方，各実行方式を並列性抽出能力，オーバヘッドの観点から比較すると図2のように図示できる．図から各実行方式にはそれぞれ相反する利点と欠点があることが分かる．そこで我々は，混在スレッド・モデルにおける各スレッド・モデルにそれぞれが対応した実行方式を適用することによって，各々の利点をうまく引き出し，より効率良い処理が期待できると考える．これを混在スレッド実行方式と呼ぶこととする．

3. *MUSCAT*への適用

本稿では，混在スレッド実行方式の対象として従来スレッド実行方式と制御並列実行方式を挙げ，*MUSCAT*への適用を検討する．これは，両実行方式が図2に示したようにスレッドレベル並列方式における並列性抽出能力とオーバヘッドの観点から見て，対極に位置すると考えたためである．加えて*MUSCAT*では，制御並列スレッド化に関してコンパイラによる自動並列化を提案しており，従来スレッドはユーザ記述による並列化，制御並列スレッドはコンパイラによる自動並列化という棲み分けが可能ではないかと考えた．即ち，従来スレッド・モデルで記述されたソース・コードを制御並列スレッド化コンパイラでコンパイルすることである．

3.1 従来スレッド実行方式の実現

*MUSCAT*は制御並列実行方式向けに設計されたアーキテクチャであるため，混在スレッド実行方式の実現に先立ち，*MUSCAT*向きの従来スレッド実行方式の実現を検討する．

3.1.1 追加ハードウェア

オンチップ・マルチプロセッサ上での従来スレッドの実現に関しては，スケジューラのハードウェア化[6]などいくつかの提案がなされている．しかしながら我々は*VIRC*における(i)スケジューリング自由度の欠如，(ii)ハードウェア・コストの増大，という反省から

- オンチップ・マルチプロセッサの特徴を生かした最低限のハードウェア・サポート
- 自由度の高いスレッド・スケジューリングを可能にするため，スケジューラはソフトウェアで実現

を基本方針とした．追加したハードウェアを以下に列挙する．

PE固有のタイマとユーザ・レベル例外 PE毎にカウンタを設けプリエンプションを可能にする．さらにプリエンプションはユーザ・レベル例外として実装し，従来SMPの外部割り込みによる実現よりも低オーバヘッドで実現可能にする．

不可分操作命令 PE間の同期や排他制御に必要な不可分操作命令として，指定アドレスのメモリ内容と指定レジスタ値を入れ換える`xchg`(*exchange*)命令を追加する．

halt，wakeup命令 PEの不要なスピン・ウェイトを避けるためPEを一時停止させる`halt`命令，再起させる`wup`(*wakeup*)命令を追加する．

3.1.2 スレッド・スケジューラ

スレッド・スケジューラの実装に関しては，より低オーバヘッドなスケジューラ[8]も提案されているが，今回はアプリケーションの移植性を高めるためにPOSIX規格[7]に準拠した従来型の実装を行った．

3.1.3 予備評価

*MUSCAT*における従来スレッド処理のオーバヘッドを見積もるため予備評価を行った．評価は別途開発した*MUSCAT*の命令レベル・シミュレータを用いてin-order処理のRISCプロセッサ4PEを1チップに集積したモデルで行った．評価の精度の点では問題ではあるが，今回の目的である混在スレッド実行方式の有効性の初期評価は可能であると考えた．

まず，スレッド・スケジューラ自体のオーバヘッドを評価するため，スレッド・スケジューラの主な操作1回あたりの平均実行命令数を調べた(表1)．スレッドの生成，スレッド切替え(プリエンプション＋次スレッドへの切替え)は共に約100命令程度で可能であった．

つぎに，スレッドの粒度と性能向上率との関係を評価するため，ある一定の粒度のスレッド20個を粒度を変えながら実行し，スレッド化していないプログラムに対する性能向上率を調べた(図3)．ただし，各スレッドには相互依存はない．図からスレッド粒度3000命令程度で性能向上率3.6を達成し，それ以上はほぼ頭打ちになっている．一方，制御並列スレッドは従来スレッドよりも常に高い性能を達成しているが，スレッド粒度が不揃いである場合は制御並列スレッドでは十分な性能向上が得られない場合がある．例えば，ランダムな粒度の20スレッドを実行させたところ，図4に示すように制御並列スレッドの性能は2.5～3程度である．ただしこれは一様乱数によるもので，スレッド粒度の分布に偏りがある場合はさらに性能が抑えられる．例えば，乱数がβ分

表1　スケジューラ中の主な関数の呼び出し1回あたりの平均実行命令数

操作	フィボナッチ数		行列積		n-queen	
	呼出回数	平均命令数	呼出回数	平均命令数	呼出回数	平均命令数
スレッド生成	177	86.32	250	84.22	8	91.00
スレッド終了(※)	177	35.88	250	38.98	8	38.25
スレッド待ち合わせ(※)	177	58.03	250	54.01	8	56.00
プリエンプション(※)	275	58.53	0	–	91	59.40
次スレッドへの切替え	597	35.23	251	26.02	150	39.51

※これらの操作は次スレッドへの切替えを伴うこともあり得るが，命令数には含まれていない．

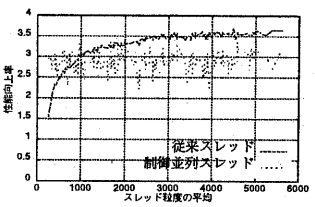

図4　スレッド粒度が不揃いである場合

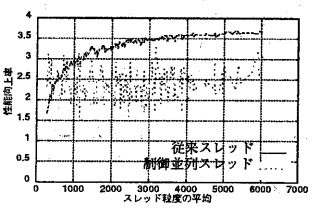

図5　スレッド粒度の分布に偏りがある場合

布(β=(0.01,0.01))に従った場合は図5のようになる．

3.2　混在スレッド実行方式の実現

混在スレッド実行方式の実現にあたり，若干のハードウェアの追加とスレッド・スケジューラの拡張を行う．

3.2.1　追加ハードウェア

ハードウェアの実現性の点から，混在スレッド実行方式はモードの切替で実現した．従来スレッド・モード時に制御並列用命令が現れた場合，制御並列実行方式の性質を利用して自動的に逐次実行する(制御並列スレッド逐次化)ことが可能である(図6)．従来スレッド・モードに`fork`(スレッド生成)命令が現れた場合は生成すべきスレッドをシャドウ・スレッドとして自PE中に保存しておき，`term`(スレッド終了)命令が現れた時点でそのシャドウ・スレッドを実行する☆．これによって，制御並列スレッドと従来スレッドが混在したプログラムを処理することが可能になる(図7参照)．なお，PE毎にモードを持たせ，数PEずつの切替えを実現することも考えられるが，*MUSCAT*が単一方向の隣接PEに対する`fork`に限定したアーキテクチャであることから本稿ではチップ全体で切替えることとした．

☆ 制御並列モードでもスレッドの数がPE数を越えた場合にシャドウ・スレッドとして保存するため，自動逐次実行機構を導入してもハードウェア・コストの増加にはつながらない．

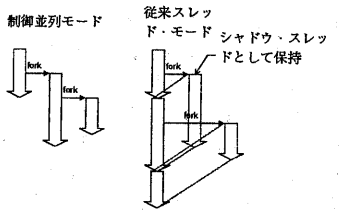

図6　制御並列スレッド逐次化

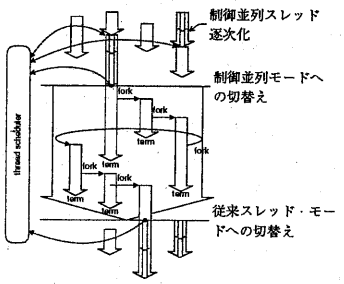

図7　*MUSCAT*における混在スレッド実行の概念図

3.2.2　スレッド・スケジューラの拡張

前3.1.2節で設計したスレッド・スケジューラに切替用のコードを追加し，両者の切替を実現する．頻繁なモードの切替えはオーバヘッドを伴う可能性があるため，切替用コードのオーバヘッドと切替の効果のトレードオフを考慮しながら，適切な頻度で切替えることが必要である．今回は，切替のオーバヘッドを抑えるため，切替用コードを最小限にする方針をとった．即ち，実行可能な従来スレッドの数が1のとき制御並列モードへ移行し，実行可能な従来スレッドの数が2以上のとき従来スレッド・モードへ移行する．

i. 従来スレッド・モードへの移行

スケジューラの入口(特にはランキューに従来スレッドを投入する部分)に従来スレッド・モードへの移行コードを設けることによって移行を自動化する．なお，これによる命令数の増加は5命令以下であった．

ii. 制御並列モードへの移行

同期変数などによりスレッドがブロックされた場合スレッド切替を行うが，スレッド切替中，もしランキューが空なら`halt`命令によりPEは停止する．ここで，停止する前にクリティカル・セクションによってランキュー

の一貫性，他の PE が wup を実行しないことを保証した上で，自 PE 停止後残る動作 PE 数が 1 になる，即ち現在の動作中 PE 数が 2 であることを調べて移行を行う．以上による命令数の増加は 20 命令程度になるが，PE 停止前の処理であることを考えると，全体に対するオーバヘッドはそれほど大きくないと予想される．ただし，クリティカル・セクションが長引くことが問題になる可能性はある．

4. 評　　価

混在スレッド実行方式の有効性を検証するため，mpeg2 ソフトウェア・デコーダを用いた評価を行った．

4.1 並列化方式

並列化は文献[2] を参考に以下の 4 方式で行った．

方式 1　全体の処理の約 35% を占める idct(逆離散コサイン変換) 部と約 10% を占める動き補償部に対して制御並列スレッドを用いて並列化を施した．mpeg2 ソフトウェア・デコーダで用いられる 1 ブロックは 8 × 8 ドットで構成されるため，idct は行側 (8 回) と列側 (8 回) の 2 つのループから成る．1 スレッドの粒度は idct で約 250 命令 (行側) と約 100 命令 (列側)，動き補償で約 50 ～ 300 命令であった．そのため従来スレッドによる並列化は図 3からも推察される通り，スレッド粒度が細かく性能を得られないため断念した．

方式 2　1 ブロック毎の復号処理 (飽和処理，idct, ブロックへの加算処理) を 1 スレッドとして従来スレッドを用いて並列化した．MP@ML 規格ではデコードの基本単位であるマクロ・ブロックは 6 ブロックから構成されるので，6 個の復号スレッド (約 6000 命令) 実行毎に 1 回，VLD(可変長符号復号処理) 及び動き補償 (約 10000 ～ 15000 命令) が挿入される．なお，並列化が難しい VLD+ 動き補償部を含めて高速化するために，2 マクロブロック分の変数領域を用意し，復号処理とのパイプライン化を図っている．また，この方式で用いたスレッド・スケジューラには 3.2.2節で追加した切替用コードは不要なため含まれていない．

方式 3　方式 2 と同様の並列化を制御並列スレッドによって行ったものである．idct は復号処理から，動き補償は VLD+ 動き補償から呼び出す下位関数であるため，フォーク 1 回モデルの制約から方式 1 との併用は不可能である．実際には，完全なパイプライン実行には方式 2 よりも多くのマクロブロック用変数領域を確保することによって更に性能向上は可能になると予想されるが，ここではコーディングが極めて複雑になるので行っていない．また，*MUSCAT*でサポートしているスレッドの投機実行などはスレッド実行方式の差異を確かめるという観点から今回は使用していない．

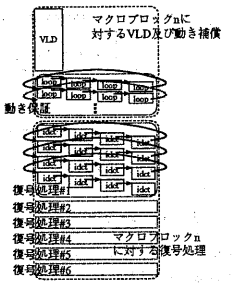

図 8　制御並列実行方式 (方式 1)

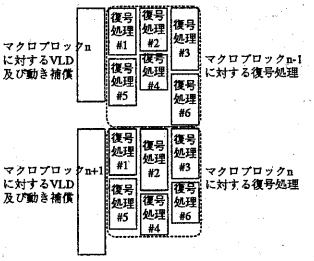

図 9　従来スレッド実行方式 (方式 2)

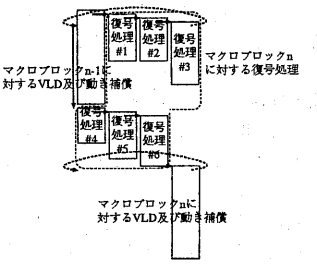

図 10　制御並列実行方式 (方式 3)

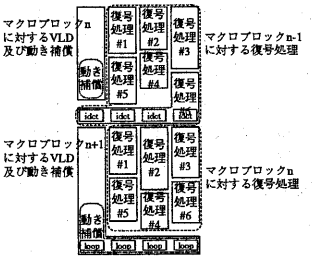

図 11　混在スレッド実行方式 (方式 4)

方式 4　方式 1 の制御並列スレッドと方式 2 の従来スレッドの並列化手法を同時に適用することによって，混在スレッド実行を行うものである．

これらの実行方式のイメージを図 8～図 11に示す．

4.2 評価結果

ワークロードとして MP@ML 標準評価画像の 3 フ

表2 各実行方式の性能向上率

方式	並列化部分		性能向上率	切替の回数
	復号処理, VLD+動き補償	idct, 動き補償内部		
1	–	制御並列スレッド	1.57	
2	従来スレッド	–	2.82	
3	制御並列スレッド	–	2.03	
4	従来スレッド	制御並列スレッド	3.26	3960

レームを用いたときの逐次実行に対する速度向上率を表2に示す．方式4では，制御並列モードと従来スレッド・モードの切り替え回数についても併せて示す．なお，評価は3.1.3節と同様の条件で行った．

方式1では，並列化していない部分が50%を越えているので，全体の性能向上率はさほど高くないが，並列化を行ったidct, 動き補償はスレッド間の粒度が揃っており，制御並列スレッドによる並列化は有効に作用していることが分かった．

次に，方式2と方式3は同様のスレッド化を異なった実行方式で行っているが，方式2の性能向上率が高くなっている．これは，図10で示したように，VLD+動き補償のスレッド粒度が大きく，欠点2.4.3(2.4節)で示した負荷分散が行えないことによる．方式3では，VLD+動き補償に続く3スレッド(図中の復元処理#1～#3)を実行するPEでは，スレッドが完了しているにもかかわらずVLD+動き補償処理スレッドが終了していないために，アイドル状態になってしまう．さらに，次のVLD+動き補償は復号処理#6が完了しないと変数領域が重複するため開始できない．以上の結果は3.1.3節の結果を実証しており，粒度が不揃い，かつある程度の大きさの粒度のスレッドでは従来スレッドが有利であることを示している．

方式4の混在スレッド実行方式は，4種類の並列化手法中もっとも良い性能を示した．これは，図11に示すように，同期点に対して終了の遅れた1従来スレッドを動的に制御並列モードに分解して4PEで実行することによって加速が行なわれたためである．モード切り替えは3.2.2節で示したように，ほとんどコストが生じていない．ただし，今回の並列化ではidct, 動き補償のみ制御並列スレッド化しており，このことが性能を律速させているといえる．方式4においても，idct, 動き補償部以外で1PE動作している部分が全体の20%存在し，これを制御並列スレッド化することによってさらなる高速化が期待できる．

5. おわりに

本稿では制御並列実行方式と従来スレッド実行方式を実行可能にし，両者の利点をうまく引き出すことが可能な処理方式として混在スレッド実行方式を提案し，評価を行った．制御並列実行方式は細粒度の並列化が可能な点で自動並列化には有望な手法ではあるが，(i)並列度抽出能力の制限，(ii)プログラミング・インタフェースとしては見せずらい，などの点からユーザ記述による粗粒度の並列化には不向きであると考える．本稿で提案した混在スレッド実行方式は，従来スレッド実行方式との動的切替えを導入することによってこれらの問題を回避可能であることを確認できた．今後は本方式について，実装法のリファインを検討するとともに，さらなるアプリケーション評価を行い，有効性をより詳細に検証する予定である．

参考文献

1) 本村，井上，鳥居，小長谷: 順序付きマルチスレッド実行モデルの提案とその評価, 情報処理学会論文誌, Vol. 37, No. 7, pp. 1335–1366 (1996).
2) 鳥居，本村，近藤，鈴木，小長谷: 順序付きマルチスレッドアーキテクチャのプログラミングモデルと評価, 情報処理学会論文誌, Vol. 30, No. 9, pp. 1706-1716 (1997).
3) 鳥居, 近藤, 本村, 西, 小長谷: オンチップ制御並列プロセッサMUSCATの提案, 情報処理学会論文誌, Vol. 39, No. 6, pp. 1622–1631 (1998).
4) 酒井，鳥居，近藤，市川，小俣，西，枝廣: 制御並列アーキテクチャ向け自動並列化コンパイル手法, JSPP98, pp. 383-390 (1998).
5) 小林，岩田，安藤，島田: 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャSKY, JSPP98, pp. 87-94 (1998).
6) 飯田，久我，末吉: マルチスレッド制御ライブラリのハードウェア化によるオンチップ・マルチプロセッサの構成, JSPP97, pp. 337-344 (1997).
7) ANSI/IEEE Std 1003.1c, Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) (C Language).
8) Kenjiro, T., Kunio T. and Akinori Y.: Stackthreads/MP: Integrating Futures into Calling Standards, *In Proceedings of ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming(PPoPP)*, (1999).
9) Tsai, J.-Y. and Yew, P.-C.: The Superthreaded Architecutre: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation, *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pp. 35–46 (1996).