2004－ARC－159　(7)
2004／7／30

SMT プロセッサにおける細粒度最適化手法の検討

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．しかし，これらを共有するということは計算機資源の競合も発生し，スループット低下の原因にもなりえる．そこで本研究では，細粒度最適化によって競合を可能な限り回避する方式について検討した．方式は，(1) 並列実行する命令流の命令を並び替える方式，(2) ループ再構成時，ループ分割しヘテロな命令流を生成する方式，(3) 利用が集中する演算器について，代替命令列を利用する方式を検討した．シミュレータによる評価の結果，(1) に関しては，メモリアクセスの時間的局所化により，キャッシュメモリの競合を抑え，性能低下を防ぐことができることを確認した．(2) については，ループ分散したものを並列実行することで性能が向上した．(3) については，乗算命令を代替命令列で実現することで，演算器の利用率が向上し，全体のスループットが最大 60% 向上した．

Fine Grain Optimization Techniques on 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. However, such resource sharing may possibly cause significant performance degradation due to resource competition. In this paper, we proposed methods in order to prevent a program from resource competition by fine grain optimization. (1) Reordering an instruction sequence. (2)Applying loop distribution and create hetero flows. (3)Substituting alternate instructions for some busy instructions. As a result of simulations, (1) Performance degradation caused by competition of cache memory is prevented by time localization of memory access. (2) Loop restructuring affects performance of parallel execution. (3) Substituting simple operations for a multiplier achieves total throughput improvement by 60% due to high usage of execution units.

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するようなマルチスレッドアーキテクチャが注目を集めている．このプロセッサアーキテクチャは，プログラムのスレッドレベル並列性（TLP: Thread Level Parallelism）を利用することで，命令レベル並列性（ILP: Instruction Level Parallelism）に着目してきた従来のプロセッサアーキテクチャの限界を克服することができる．本稿では，マルチスレッドアーキテクチャの中でも，とくに SMT プロセッサに注目する．

SMT プロセッサ[1),8)] 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．

従来の並列処理計算機，たとえば対称型マルチプロセッサ計算機では，同時実行する命令列がどのようなものであれ，それらは独立に実行されるためほかのプロセッサの実行命令流による速度の影響は少ない．

しかし，SMT プロセッサではプロセッサ資源を共有するため，資源の競合が発生する．たとえば，ワーキ

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

ングセットの大きいプログラムを SMT プロセッサ上で並列実行した場合，並列実行するそれぞれの命令流がメモリアクセスを行い，それぞれがキャッシュするため競合が発生し，最悪の場合キャッシュメモリのスラッシングが発生する可能性がある．また，ある演算器に利用が集中した場合，計算機資源の有効利用ができない可能性がある．これらの結果，期待するスループットが得られない，もしくは並列実行により全体性能に悪影響を与えるような場合がある．

そこで，本研究では並列実行する命令流の各命令列に注目し，これらの競合の発生を抑える手法について検討した．従来，ほとんどの SMT プロセッサ向けのソフトウェアレベルでの最適化研究では粗粒度レベルでの研究，たとえばプロセスやスレッドレベルでのスケジューラの検討などや，メモリアクセスを意識したメモリ割付や並列化手法の研究などが行われてきたが，命令レベルでの細粒度最適化の検討はほとんど行われていない．

本稿で検討した方式は次の 3 点である．(1) 同時実行する命令の並び替えを行う (2) ループの再構成を工夫する (3) 代替演算器を利用するようなコードを生成する方式，である．(1)(2) はメモリアクセスや個々の演算器の利用を時間的に局所化することで，細粒度ではあるがヘテロな部分を作り，キャッシュミスや演算器の競合を抑える．(3) は利用が集中している演算器の代替命令列を利用することで演算器の競合を抑えるというアプローチである．

本稿では，2 節で対象とするアーキテクチャについて述べ，3 節で各方式について詳細を述べ，4 節でそれについての評価を行う．5 節で関連研究について述べ，6 節でまとめを行う．

2. 対象とするアーキテクチャ

筆者らは OChiMuS PE プロセッサ[9] について研究，開発をすすめている．本稿ではこのプロセッサを対象に評価を行う．

OChiMuS PE は MIPS プロセッサアーキテクチャをベースにした SMT プロセッサであり，複数の実行コンテキストを持ち，それぞれプログラムカウンタ，レジスタファイル，および実行状態レジスタなどを保持する．演算器などの資源は並列実行する各命令流で共有する．

ここで，並列実行する命令流の単位を実スレッド（AT: Architecture Thread）とする．実スレッドの管理は，ユーザレベル（非特権レベル）で実行可能なスレッド管理命令によって行われる．同時実行する実スレッドはすべて同一アドレス空間で実行される．

3. 最適化手法の検討

命令列レベルでの細粒度最適化手法について述べる．

```
# original          # localised
lw    $2,0($5)      lw    $2,0($5)
lw    $3,12($5)     lw    $3,12($5)
addu  $6,$2,$3      lw    $4,4($5)
...(1)              lw    $5,16($5)
lw    $2,4($5)      addu  $6,$2,$3
lw    $3,16($5)     addu  $7,$4,$5
addu  $6,$2,$3      ...(1)'
...(2)              ...(2)'
```

図 1 命令の局所化の例（MIPS アセンブラ）
Fig. 1 A Sample of instruction localisation

本稿での議論は，対象とするプログラムはループレベルで並列化してあるとし，並列実行する命令列の最内ループに対して各方式を適用する．

SMT プロセッサは並列実行する各命令流で演算器やキャッシュを共有利用するため，並列実行する命令列によって，性能が大きく変わる．とくに，ループ並列化を行った場合，各並列実行単位で利用する演算器が同じ場合が多いので，ひとつの演算器に利用が集中し，演算器の競合が発生しやすい．

本節ではこの問題を解決するために，命令列を操作するという点に着目した手法を検討する．

3.1 命令の並び替え

並列実行する命令流の，各命令列を同じ演算器を利用するものの命令間の距離が近くなるようにそれぞれの命令を計算の意味が変わらない範囲で並び替える（**図 1**）．命令レベルでの並び替えであるため，最も細粒度な最適化であるといえる．

また，C 言語の式のようなブロックレベルでの並び替えも行う．ブロックでボトルネックとなるような演算器の利用を局所化するように並び替える．命令単位で並び替えることに比べると利用する演算器の時間的局所性にばらつきがあるが，本手法は言語処理系の比較的前段階で実現しやすい方式である．

この提案手法により，各スレッドが同時に同じ演算を行う可能性を減少させる．これにより，メモリアクセスを時間的に局所化することでキャッシュメモリのスラッシングの機会を減少させる．また，演算器の利用の集中による競合の発生を抑えることができる．

命令をこのように並び替えるためにはレジスタの数に余裕がある必要がある．そのため，この手法の適用可能な範囲はレジスタ数に左右される．

また，なるべく多くの命令を並び替えるために最内ループを適切にアンローリング，またはソフトウェアパイプライニングなどの適用が有効である．次節で述べる評価に利用したプログラムはループアンローリングを行い，この並び替え手法を適用した．

3.2 並列実行を意識したループ再構成

ループ再構成が可能な場合，ループ分割を行う（**図 2** の (B)）．このとき，実行順序の変更が可能ならば，

```
//(A) fusion          // (B)distribution
for(i=0;i<N;i++){     for(i=0;i<N;i++)
  IterA();              IterA();
  IterB();            for(i=0;i<N;i++)
}                       IterB();
```

図 2 ループ再構成の例
Fig. 2 An example of loop restructuring

ループの実行順序を変更した命令流も生成する．通常，ループの誘導変数増減のコストを削減するため，統合するのが一般的であるが，並列実行した場合，その削減効果よりも命令列の組み合わせによる性能向上，および低下が問題になる．

提案手法は，図 2(B) を例にすると，`IterA()` のループを先に行う命令流 A と，`IterB()` を先に行う命令流 B を用意し，それらの命令流を並列実行する．これにより，並列実行したとき，よりヘテロな命令流を作ることができる．

3.3 代替命令列の利用

ある演算器の利用が集中しているとき，その演算器を利用する命令をほかの演算器を利用する命令列に置き換えるという手法を検討する．

ある演算器が集中して利用され，競合を起こしている，つまりボトルネックになっているということは，ある特定の演算器の利用率だけが高くなり，利用率の少ない演算器が生じるということである．そこで，その特定の演算器の計算をほかの演算器を利用する計算によって代替できる場合，その代替表現に代えることでほかの演算器の利用率を向上させ，全体のスループットを向上させる．

代替命令の具体的な例としては，たとえば実行コストの高い乗除算器（Complex ALU）による乗算命令や除算命令の代わりに加算やシフト，論理演算などの演算器を利用して乗算を行うことが可能である．また，計算の高速化のためにメモリ上に計算結果をキャッシュするようなプログラムも，再度計算を行うほうがスループットが向上する可能性がある．

乗算命令を代替命令列によって実現する方式についてもう少し詳しくみる．乗算を乗算器を利用せず，Simple ALU による演算のみで記述すると，次節で述べる評価環境においては，**表 1** に示すような実行性能を示す．代替命令列の命令数が多く，直感的にはこのような最適化で速くなることなどないように思えるが，SMT プロセッサでは演算器を共有して並列実行するため，空き演算器を有効利用することで全体の速度向上が期待できる．

4. 評　　価

本節では前節で検討した手法についてシミュレータを用いて評価した結果を述べる．

表 1 通常命令と代替命令列の比較
Table 1 Comparison between a normal instruction and substitute instructions

	命令数	必要サイクル数
32bit/16bit 乗算器利用	2	13
32bit 代替命令列	188	84
16bit 代替命令列	93	50

＊ 必要サイクル数は逐次実行時の性能

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

TLP	1	2	4	8
Fetch Buff.	32	16	8	4
Disp. Queue	16			
Reorder Buff.	128	64	32	16
Normal RS	Simple ALU: 8, Complex ALU: 4			
LD/ST RS	8 (RS: Reservation Station)			
Simple ALUs	4(0 cycle delay)			
Comple ALUs	2(Mult delay: 12), Div delay: 32)			
Fetch, Decode, Dispatch, Retire Insns	16	8	4	2
Finish Insns	16			
L1D Cache	32byte/line, 2048lines, 2ways Access Penalty: 0 cycle			
L2D Cache	64byte/line, 8192lines, 8ways Access Penalty: 20 cycle			

4.1 評 価 方 法

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）[9] を用いて行った．MUTHASI は OChiMuS PE をシミュレートし，様々なパラメータを設定することが可能である．また，並列実行する最大実スレッド数を 1 に設定すると，通常のシングルスレッド MIPS プロセッサの挙動を示す．シミュレータの各パラメータは**表 2** のとおりである．キャッシュメモリは各実スレッドが共有する．

プログラムの並列化は手動で行い，マルチスレッドアーキテクチャ向けユーザレベルスレッドライブラリ MULiTh[10] を利用して並列実行させた．MULiTh は Pthread 仕様準拠のスレッドライブラリで，OChiMuS PE 用ユーザレベルスレッド制御命令を利用してソフトウェアレベルでのスレッドの抽象化を実現している．

プログラムの作成には gcc-3.4.0（最適化オプション -O1），binutils-2.14 を評価環境用にカスタマイズしたものを利用した．各種手法の適用は，MIPS アセンブラ命令列を手動で操作するか，C 言語ソースコードを直接操作することで行った．

4.2 命令並び替えの評価

本手法の評価は**表 3** に示すプログラムによって行った．

4.2.1 メモリアクセスについて

提案手法のメモリアクセスに対する効果を評価する．

表 3　評価プログラム一覧
Table 3　Evaluate Programs List

プログラム	説明	ボトルネック
matmult	行列積を求めるプログラム	LD, 乗算
matmult2	行列積を求めるプログラム（共有部分を排除）	LD, 乗算
scalar_op	行列 A, B, 定数 x において, $x(A+B)$ を求めるプログラム	乗算
sutadd	行列 A, B の飽和演算	分岐
twoloop	$A_{ij} = ijB_{ij}$ となる行列 A を求めるプログラムと sutadd を同時実行	乗算･分岐･LD

＊ LD: メモリアクセス（ロード）

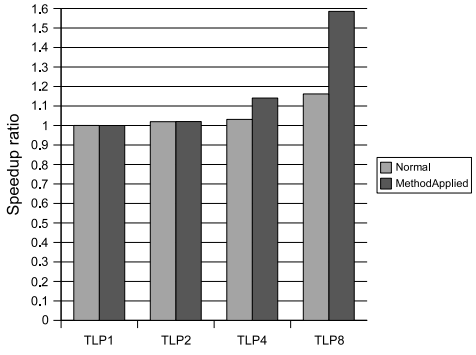

図 3　matmult の命令並び替えによる速度向上率
Fig. 3　Speedup ratio of matmult with instruction reordering

matmult の結果を**図 3** に示す．なお，グラフ中の $TLPx$ という表記は同時実行する命令流が x 個であることを示す．

TLP が低い場合はどちらも性能に差がない．しかし，TLP が高くなるにつれて命令を局所化したもののほうが性能が高くなっている．これは，L1 データキャッシュのヒット率とまったく同じ傾向である．つまり，本手法を適用すると，キャッシュのヒット率が向上，もしくは低下を阻止することがわかる．

次に，matmult2 の結果を**図 4** に示す．これは，行列積を計算するにあたって，計算する行列要素の順番を変更することにより各命令流がデータ共有部分をほとんど持たないように調整したものである．つまり，非常にワーキングセットが大きく，実スレッド間の建設的な干渉が働かないことを意味する．そのため，速度向上がない．しかし，提案方式を適用したものでは，並列度を高くしてもキャッシュメモリの競合による速度低下がある程度抑えられていることがわかる．

最後に，scalar_op の結果を示す（**図 5**）．このプログラムは線形にメモリをアクセスするため，キャッシュメモリのヒット率は高い．しかし，乗算器の利用率が高いため，並列化してもあまり速度向上しない．演算器の競合に対して，命令の並び替えは効果がないことがわかる．

しかし，キャッシュメモリのヒット率（**図 6**）は提案方式を適用したもののほうが並列実行時でもヒット率が高いことがわかる．

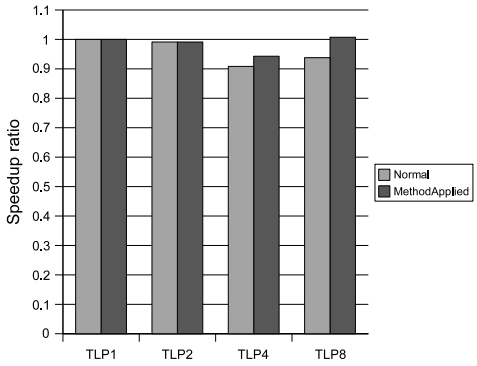

図 4　matmult2 の命令並び替えによる速度向上率
Fig. 4　Speedup ratio of matmult2 with instruction reordering

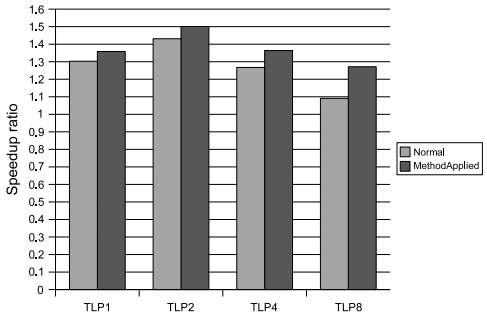

図 5　scalar_op の命令並び替えによる速度向上率
Fig. 5　Speedup ratio of scalar_op program

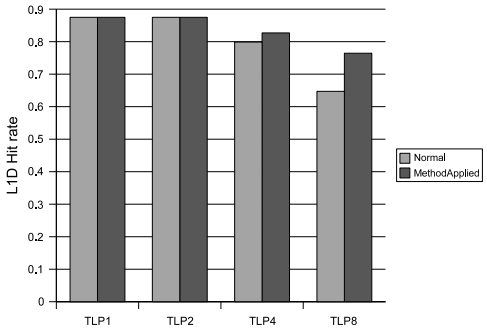

図 6　scalar_op の L1 キャッシュヒット率
Fig. 6　L1 cache hit rate of scalar_op program

4.2.2　演算器の競合について

次に，提案手法が演算器の競合による速度低下に効果があるか，という点について調査する．演算器の利用率に着目するため，シミュレータのキャッシュシステムを無効とした．つまり，メモリアクセスレイテンシは常に 0 である．

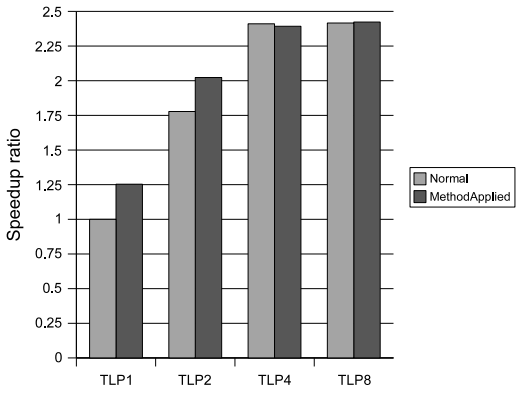

図 7 sutadd の命令並び替えによる速度向上率（キャッシュ無効）
Fig. 7 Speedup ratio of sutadd with instruction reordering(no cache system)

sutadd，二つの行列を飽和加算するプログラムの速度向上率を**図 7** 示す．なお，このプログラムの命令並び替えは C 言語の式レベルの移動で行った．

結果を見ると，並列度が低ければ最適化を行ったものの性能が高い．これは，そもそも式を移動したことで実行命令数自体が減っているからである．これが，並列実行するにしたがって性能差がなくなる．つまり，ある程度の総命令数や最適化の違いなどは SMT プロセッサ上で並列実行した場合，その差がなくなることを示している．

4.3 ループ再構成手法

ループの統合方式について評価した結果を示す（**図 8**）．

P1 はループ統合したものである．P2,P3 はループを分散させたものである．ただし，どちらのループを先にするかによって P2,P3 となっている．P4 は並列実行する命令流の半分を P2 の命令流，半分を P3 の命令流とした．

キャッシュメモリが無効（メモリアクセスレイテンシ 0）の場合，逐次実行時にはループ統合を行ったもの（P1）のほうが性能が良いが，TLP8 になると P4 のほうが速度向上率は高い．これは，ループ分散で，よりヘテロな命令流を同時実行することで並列実行時の効率向上に繋がったと考えられる．

キャッシュメモリを有効にした場合，ループ統合したものが性能が良い．これは，メモリアクセスがそれぞれのループのボトルネックとなり，並列実行する命令流がヘテロでなくなったためと考えられる．

4.4 代替命令の利用の評価

この評価は，matmult にこの方式を適用することで行った．求める行列の各要素の値は符号なし 32bit である．結果を**図 9** に示す．適用するパターンとして，P1〜P4 を用意した．P1,P2,P3,P4 は繰り返し中それぞれ 1/32, 1/16, 1/8, 1/4，乗算命令の代わりに代替命令列を利用した．P0 は代替命令を利用しないものである．速度向上率は P0 の TLP1 を基準とした．

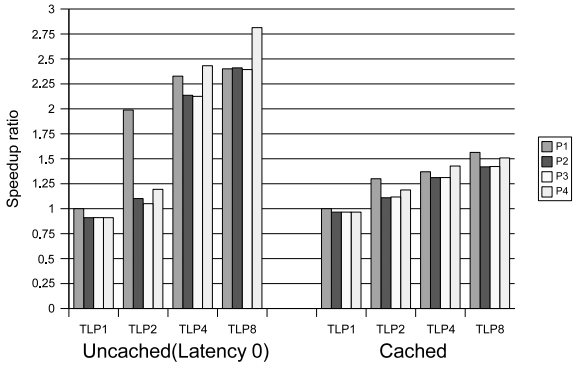

図 8 twoloop のループの再構成方式による速度向上率
Fig. 8 Speedup ratio between loop restructuring methods

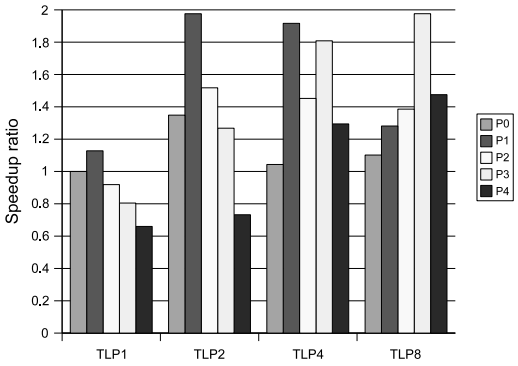

図 9 代替命令の利用による速度向上率（符号なし 32bit 整数乗算）
Fig. 9 Speedup ratio with substituted instructions(unsigned 32bit integer multiply)

並列度を高くすることで適当な量の置き換えを行うことで高いスループットが達成できていることがわかる（TLP8 での P3 は TLP8 での P0 よりも 60%高速）．これは，うまく空き演算器を利用し，乗算演算器の遅延とロード時のキャッシュミスした場合の遅延を多数の SimpleALU の命令が埋めるため，プロセッサ資源を有効利用しているためである．

どの程度の割合で代替命令列を利用すればよいかについてはばらつきがあるが，たとえば実行時にプロファイリングを取ることで最適な割合を選択することができる．

5. 関 連 研 究

SMT プロセッサに関する研究は，粗粒度並列タスクなどをどのように扱うかに着目した研究が多くなされている．たとえば SMT プロセッサにおけるスケジューラの研究としては，文献 5)，6) また，我々の先

行研究として 11) がある．これらの研究の共通点は，実行時プロファイリングによってフィードバックを得て，最適なスケジューリングを行うということである．SMT プロセッサの挙動は予測しづらいため，動的なスケジューリングが有効である．

SMT プロセッサについての言語処理系の研究としては，Lo らの研究[2] や Puppin らの研究[4]，Tian らの研究[7]，などが挙げられる．文献 2) で Lo らは既存のコンパイラによる最適化について，SMT プロセッサではそれらがどのような挙動を示すかについて調査し，このプロセッサにおける最適化の指針を示している．文献 4) ではループ並列化についての手法を適用し，最適な同時実行する命令流の数について考察している．文献 7) では，Intel の Xeon プロセッサなどにおける HyperThreading に適した OpenMP コンパイラによる並列化手法について述べている．

SMT プロセッサについての細粒度の調査として，Mitchell らの研究[3] がある．彼らはこの研究で従来方式で ILP を向上させたものを単純に SMT プロセッサ上で動作させても，TLP をあげるとそれが効果がない，または悪影響を及ぼすことを示した．また，SMT プロセッサについての性能モデルを考察している．

6. ま と め

本稿では SMT プロセッサにおける細粒度最適化方式を検討し，並列実行する命令列について，次の 3 つの最適化手法を提案した．(1) 命令列を演算器の利用が集中するように並び替える，(2) ループ分割を行い，ヘテロな命令流を作る，(3) 集中利用される演算器の代わりに代替命令列と差し替える手法である．

シミュレータによる評価の結果，(1) メモリアクセス命令を局所化したことにより，時間的局所性が向上し，メモリアクセス速度が改善，(2) ループ分散によってヘテロな命令流を生成し，性能向上が可能であることを確認，(3) 利用が集中している演算器を代替命令を利用することで速度向上，という結果を確認した．

今後の課題として，より一般的なアプリケーションでの提案方式の評価，SMT プロセッサにおける並列実行時の詳細なパフォーマンスモデルの構築や，本稿で提案した最適化を実際に言語処理系の最適化ルーチンとして実装することが挙げられる．また，SMT プロセッサにおける言語処理系というテーマでは，これに適した並列化手法の検討が必須である．

謝辞 本稿執筆にあたり，筑波大学大学院システム情報工学研究科前田敦司助教授にアドバイスを頂きました．感謝いたします．

参 考 文 献

1) Hirata, H., Kimura, K., Nagamine, S., Mochizuki, Y., Nishimura, A., Nakase, Y. and Nishizawa, T.: An elementary processor architecture with simultaneous instruction issuing from multiple threads, *Proceedings of the 19th annual international symposium on Computer architecture*, ACM Press, pp. 136–145 (1992).

2) Lo, J. L., Eggers, S. J., Levy, H. M., Parekh, S. S. and Tullsen, D. M.: Tuning compiler optimizations for simultaneous multithreading, *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, pp. 114–124 (1997).

3) Mitchell, N., Carter, L., Ferrante, J. and Tullsen, D.: ILP versus TLP on SMT, *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, p. 37 (1999).

4) Puppin, D. and Tullsen, D.: Maximizing TLP with loop-parallelization on SMT, *Workshop on Multithreaded Execution, Architecture and Compilation* (2001).

5) Snavely, A. and Tullsen, D.M.: Symbiotic Jobscheduling for a Simultaneous Multithreading Processor, *Architectural Support for Programming Languages and Operating Systems*, pp. 234–244 (2000).

6) Snavely, A., Tullsen, D. M. and Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor, *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ACM Press, pp. 66–76 (2002).

7) Tian, X., Bik, A., Girkar, M., Grey, P., Saito, H. and Su, E.: Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance, *Intel Technology Journal*, Vol. 6, pp. 1–13 (2002).

8) 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).

9) 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).

10) 笹田, 佐藤, 河原, 加藤, 大和, 中條, 並木: マルチスレッドアーキテクチャにおけるスレッドライブラリの実現と評価, 情報処理学会論文誌 : ACS, Vol. 44, No. SIG11(ACS3), pp. 215–225 (2003).

11) 内倉, 笹田, 佐藤, 河原, 加藤, 大和, 中條, 並木: SMT プロセッサにおけるスレッドスケジューラの開発, 情報処理学会研究報告 (OS), Vol. 2004, No. 63, pp. 141–148 (2004).