SMT プロセッサにおけるキャッシュリプレース動的切替え方式

小笠原 嘉泰[†,††] 佐藤 未来子[†]
並木 美太郎[†] 中條 拓伯[†]

SMT プロセッサは，複数のスレッドで演算器やキャッシュメモリを共有し，性能向上を目指している．ところが，キャッシュメモリの共有が原因で，キャッシュラインにおけるスレッド間競合が発生し，性能が低下するという問題がある．スレッド間競合を抑え性能向上を実現する手法として，スレッドごとにリプレース可能な領域を制限するキャッシュリプレース方式があるが，キャッシュ容量やプログラムによっては従来の擬似 LRU と比較し性能低下を引き起こす．そこで本論文では，その SMT プロセッサ向けキャッシュリプレース方式の利点と問題点に着目し，プログラム実行中に擬似 LRU との動的切替えを行うことで，性能向上を目指す．リプレース方式の動的切替えとして，スレッド間競合ミスを切替えパラメータとする動的切替え方式と，セットごとにリプレース方式を切り替える動的切替え方式を提案し，設計した．評価の結果，各動的切替え方式は有効に動作し，SMT プロセッサ向けリプレース方式で発生した性能低下を抑え，さらに擬似 LRU と比べ最大 1.427 倍と SMT プロセッサ向けリプレース方式よりも高い性能向上をもたらした．また，各動的切替え方式を実装しハードウェアコストを見積もった結果，どちらの方式も，プロセッサとキャッシュメモリを含んだチップ全体で 3%未満とわずかなハードウェア増加量で実現できることを示した．

Dynamic Switch Strategies of Cache Replacement for an SMT Processor

Yoshiyasu Ogasawara,[†,††] Mikiko Sato,[†] Mitaro Namiki[†]
and Hironori Nakajo[†]

An SMT processor aims to gain higher performance by sharing resources such as ALUs and cache memory among several threads. However, sharing cache memory causes thread conflict miss which degrades its performance. In order to inhibit thread conflict, a cache replacement strategy that restricts replaceable blocks for each thread is proposed. However, to compare with conventional pseudo-LRU replacement, the replacement algorithm which is suitable for an SMT processor causes performance degradation depending on programs or cache configuration such as cache size. This paper proposes two dynamic switching strategies of cache replacement in order to improve performance. One uses the number of thread conflict miss as invocator of switching, and the other switches replacement algorithm in each set. As a result, dynamic strategy shows 1.427 times as high performance as a conventional replacement strategy. Furthermore, both dynamic switching strategies can be implemented with small additional hardware cost in less than 3%.

1. はじめに

近年，プロセッサアーキテクチャとして，スレッドレベル並列性（TLP: Thread Level Parallelism）を利用したマルチスレッドアーキテクチャに注目が集まっている．マルチスレッド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）プロセッサ[1)] がある．これらのプロセッサは，1 チップ上で複数のスレッドを並列に実行し，性能向上を目指している．特に SMT プロセッサは，1 チップで複数のハードウェアコンテキストを持ちつつ各種演算器やキャッシュメモリなどスレッド間で共有できる資源をできる限り共用することで，資源の有効活用とプロセッサの性能向上を同時に実現している．

しかし，SMT プロセッサには，スレッド間で共有している資源の競合や枯渇が発生するという短所がある．具体的には，スレッドのキャッシュライン競合によるキャッシュミスの増加，各種演算器の枯渇があり，そのため，これらが原因でプロセッサの性能が低下し

† 東京農工大学大学院
Graduate School of Technology, Tokyo University of Agriculture and Technology
†† 日本学術振興会特別研究員 DC2
Research Fellow of the Japan Society for the Promotion of Science DC2

てしまう場合がある．

特にキャッシュラインのスレッド間競合による性能低下は深刻であり，それを解決すべくハードウェア，ソフトウェア両面から様々な解決策が提案されてきた[2),9),13)]．その解決策の1つとして，我々はスレッド間競合ミスを抑える SMT プロセッサ向けキャッシュリプレース方式を提案している[2)]．これは，スレッドごとにリプレース可能な領域を制限し，スレッドの干渉による競合ミスを抑える働きがある．評価の結果，このリプレース方式は有効に動作し，従来の擬似 LRU と比較し全体的に性能向上を実現した．しかしながら，プログラムやキャッシュ容量によっては，一部性能低下を引き起こした．

そこで，本論文では，擬似 LRU と SMT プロセッサ向けリプレース方式のプログラム実行中における動的切替え方式を提案する．プログラム実行中に適切なタイミングで両リプレース方式を切り替えることにより，SMT プロセッサ向けリプレース方式で発生した性能低下を抑え，さらなる性能向上を目指す．また，提案する動的切替え方式を実装し，具体的なハードウェア量を見積もり，その実現可能性を検証する．

なお，本論文では，マルチスレッドプロセッサにおいて1つのスレッドを処理する単位を**実スレッド**（*AT: Architecture Thread*）と定義する．また，マルチスレッドプログラミングやコンパイラによって生成されるスレッドを**論理スレッド**（*LT: Logical Thread*）と定義する．つまり，プログラムの論理スレッドを，SMT プロセッサが保持している実スレッドに順次割り当て，スレッドを並列実行する．論理スレッドの実スレッドへの割当て方法として，スレッドスケジューラなど[13)]を用いることもできるが，本論文では，キャッシュリプレース方式のみの性能を調査するため，生成順に論理スレッドを実スレッドに割り当てる FIFO（First In First Out）を採用する．

次章では，研究背景を示す．3章では，切替え対象のリプレース方式について示す．4章では，プログラム実行中に適切なタイミングでリプレース方式を切り替える動的切替え方式を提案，設計，検討する．5章では評価を，6章では関連研究との比較を行う．

2. 研究背景

本章では研究背景として，本論文で前提とする SMT プロセッサアーキテクチャとそのキャッシュメモリにおける問題点を示す．また，SMT プロセッサ向けリプレース方式の利点，問題点を示す．

2.1 前提とする SMT プロセッサアーキテクチャとキャッシュメモリの問題点

本論文では，SMT プロセッサとして *OChiMuS*（*On Chip Multi SMT*）*PE* [3)] を使用する．OChiMuS PE の特徴の1つとして，スレッドに，**論理スレッド番号（LTN: Logical Thread Number）**という ID を付加し，スレッド管理の効率化を図っている．本論文のキャッシュメモリでは，この LTN を使用するが，OChiMuS PE と LTN でなくとも，論理スレッドを一意に特定できるスレッド ID のようなものが存在すれば，他の SMT プロセッサアーキテクチャやキャッシュメモリを共有した CMP においても対応が可能である．

SMT プロセッサアーキテクチャでは，各スレッドで共有するデータの有効活用を目指し，L1，L2 キャッシュメモリを共有する．ただし，L1-I（Instruction）-キャッシュメモリは，読み込みしか行わず，コヒーレンス維持の必要がないため，OChiMuS PE では実スレッドごとに L1-I-キャッシュメモリを持つ．本論文ではターゲットとするキャッシュメモリとして，L1-D（Data）-キャッシュメモリを扱う．以降，キャッシュメモリという表記は，L1-D-キャッシュメモリを指す．

SMT プロセッサにおけるキャッシュメモリでは，複数のスレッドが並列実行するため，あるスレッドが他のスレッドのデータをリプレースしてしまうスレッド間の干渉が発生する．このスレッド間の干渉によりキャッシュライン競合が多発することは，**破壊的干渉（Destructive Interference）**と定義されており[9),14)]，本論文では，破壊的干渉により発生する競合ミスを**破壊的干渉ミス（Destructive Interference Misses）**と定義する．

2.2 SMT プロセッサ向けリプレース方式の利点と問題点

我々は，SMT プロセッサ向けのキャッシュリプレース方式として，破壊的干渉ミスを抑える **LTN（Logical Thread Number）方式**を提案している[2)]．LTN 方式は，実スレッドが保持する LTN を用いることで，スレッドのリプレース可能なウェイを制限し，スレッド間の干渉を緩和させる方式である．キャッシュアクセス時は従来と変わらず，すべてのウェイに対してアクセスできるようにし，キャッシュミスが発生した場合のみ，リプレース方式として LTN 方式を使用する．そうすることでキャッシュメモリを共有するメリットを損なわず，スレッドの破壊的干渉ミスを抑えることができる．

擬似 LRU（LRU）と LTN 方式（LTN）の性能比

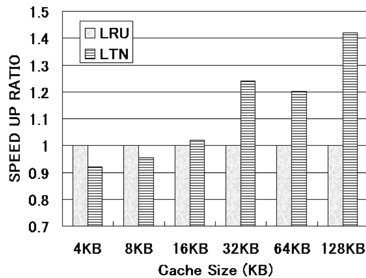

図 1 擬似 LRU と LTN 方式の性能比較
Fig. 1 The performance comparison between pseudo-LRU and LTN strategy.

較として，文献 2) の評価結果の一部を**図 1** に示す．プログラムは，論理スレッドを 8 個に分割した行列乗算を用いている．また，プロセッサの実スレッド数を 2 とし，L1-D-キャッシュメモリはウェイ数：4，ブロックサイズ：32 B とし，キャッシュ容量を 4 KB〜128 KB☆と変化させ，評価している．詳細なパラメータは文献 2) を参照されたい．

図 1 は擬似 LRU の性能を基準とし，LTN 方式の性能向上率を示している．キャッシュ容量 16 KB〜128 KB の場合で性能向上しており，特に 128 KB の場合，1.420 倍と高い性能向上率を示している．これは LTN 方式が有効に働き，破壊的干渉ミスを抑えることができた結果である．

ところが，キャッシュ容量が 4 KB，8 KB の場合，性能向上率が 0.922 倍，0.953 倍となり，擬似 LRU と比較して性能が低下している．これは，キャッシュ容量が少ない場合，LTN 方式を使用することで，各スレッドのリプレースできる領域が減少し，リプレース領域の制限が欠点として働いてしまったためである．また，行列乗算は破壊的干渉ミスが多いプログラムのため，LTN 方式の性能向上率が高いが，破壊的干渉ミスが少ないプログラムは，LTN 方式の性能向上率が低く，場合によっては性能低下を引き起こすことがある．

本論文では，これら LTN 方式の利点と問題点に着目し，それらを考慮した有効なリプレース動的切替え方式を提案，設計し，SMT プロセッサ全体の性能向上を目指す．

3. 切替え対象キャッシュリプレース方式

本章では，まず，動的切替え対象のキャッシュリプレース方式である擬似 LRU と LTN 方式の実現方法について示す．次に，両リプレース方式を同時に実現する方法を示す．

☆ 文献 2) では，8 KB，32 KB，128 KB を評価している．4 KB，16 KB，64 KB の結果は追加評価による．

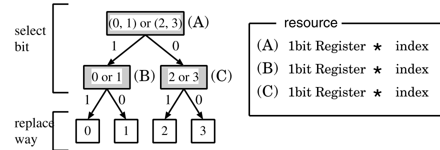

図 2 pseudo-LRU の実現方法
Fig. 2 The realization method of pseudo-LRU.

3.1 擬似 LRU の実現

キャッシュメモリのリプレース方式として，LRU 方式，ランダム方式，ラウンドロビン方式など[7] があるが，性能面において有利である LRU 方式が多用されている．しかし，キャッシュメモリのウェイ数が増加すると，LRU を実現するハードウェア増加量が大きくなってしまうため，ウェイ数が 2 の場合以外は，LRU を擬似的に用いている．本研究では，擬似 LRU として pseudo-LRU [7] を用いる．pseudo-LRU は 2 分木によって，リプレースウェイを決定する．4 ウェイセットアソシアティブの pseudo-LRU の実現方法を**図 2** に示す．

図 2 中の (A)，(B)，(C) は，1 bit × インデックスのレジスタである．(A)，(B)，(C) のレジスタの更新はキャッシュヒット時に次のように行う．

(1) (A) にヒットしたウェイ番号の上位 1 bit を格納する．
(2) 上位ビットが 0 の場合は (B) を，1 の場合は (C) を更新する．格納する数はヒットしたウェイ番号の下位 1 bit とする．

リプレースウェイを決定するとき，まず (A) のレジスタに 1 が格納されていた場合，(B) を選択し，0 が格納されていた場合，(C) を選択する．次に，(B) が選択された場合，(B) に 1 が格納されていたらウェイ 0 をリプレース，0 が格納されていたらウェイ 1 をリプレースする．(C) が選択されたときも同様であり，(C) に 1 が格納されていたらウェイ 2 をリプレース，0 が格納されていたらウェイ 3 をリプレースする．

このアルゴリズムにより，pseudo-LRU は枝の中で最近アクセスしたウェイではない枝を指すことができる．

3.2 LTN 方式の実現

LTN 方式は，論理スレッド番号（LTN）を用いて破壊的干渉ミスを抑えるリプレース方式である[2]．リプレースウェイ制限に LTN を用いるが，LTN から n ビットを取り出しリプレースウェイを制限するとき，それを LTN-n 方式と呼ぶ．たとえば，LTN から 1 ビッ

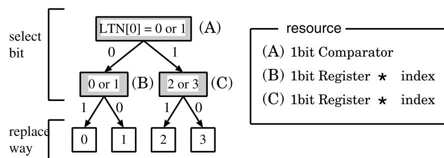

図 3 LTN 方式の実現方法
Fig. 3 The realization method of LTN strategy.

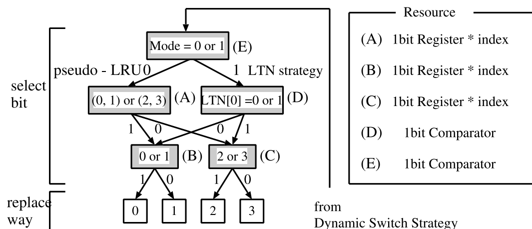

図 4 擬似 LRU と LTN 方式の同時実現方法
Fig. 4 The realization method of both replacement strategies.

トを用いてリプレースウェイを制限するときは LTN-1 方式となり，各実スレッドのリプレース可能ウェイはすべてのウェイの半分となる．以降，本論文の LTN 方式という表記は，LTN の最下位 1 bit を取り出した LTN-1 方式を指す．

4 ウェイセットアソシアティブのときの LTN 方式の実現方法を**図 3** に示す．図 2 の擬似 LRU と比べると，(A) の部分が異なる．LTN 方式の場合，(A) の箇所が，キャッシュアクセスした実スレッドの LTN から取り出したビットの比較器となる．図 3 の場合，LTN の最下位 1 bit を判定しているので，論理スレッド（LT）を 8 個に分割する場合，LT0，LT2，LT4，LT6 が (B) を選択し，LT1，LT3，LT5，LT7 が (C) を選択することになる．

比較器を組み込むことになるが，1 bit × インデックス分のレジスタ領域がなくなることになり，ハードウェア量は擬似 LRU に比べ減少する．

3.3 擬似 LRU と LTN 方式の同時実現

擬似 LRU と LTN 方式の実現方法の違いは，図 2，図 3 の (A) の部分のみとなるため，両リプレース方式を同時に実現することは容易である．両リプレース方式を同時に実現する方法を**図 4** に示す．

図 2 の擬似 LRU と比べると，(D) と (E) の回路が新たに必要となる．(D) は LTN 方式を実現するために必要な 1 bit 比較器である．次に (E) の回路について説明する．次章で提案するリプレース動的切替え方式から，プログラム実行中に適切なタイミングでリプレースモード（Mode）として，0 または 1 が出力されてくる．それを (E) の入力値とし，0 が入力された場合は擬似 LRU を使用するため (A) を選択し，1 が入力された場合は LTN 方式を使用するため (D) を選択する．そのため，(E) として，擬似 LRU と LTN 方式を切り替えるための 1 bit 比較器が必要となる．

つまり，擬似 LRU と LTN 方式を同時に実現する場合，擬似 LRU と比較し，1 bit 比較器が 2 つ追加されるだけであり，ハードウェア増加量は問題とならない．

4. 動的切替え方式

本章では，破壊的干渉ミスを切替えパラメータとする動的切替え方式として **SDI 方式（Switch by miss ratio of Destructive Interference）**を，セットごとにリプレース方式を切り替える動的切替え方式として **SON 方式（Switch by Occupied Number of ways in set）**を提案し，設計する．

4.1 SDI 方 式

4.1.1 SDI 方式の概要

リプレース動的切替え方式として，ここでは破壊的干渉ミスを切替えパラメータとする SDI 方式を提案する．破壊的干渉ミス数をプログラム実行中につねに測定し，破壊的干渉ミス率を計算する．計算した結果，破壊的干渉ミス率が一定値以下の場合は擬似 LRU を，一定値を超えた場合は LTN 方式を選択する．

このように破壊的干渉ミス率を閾値としてとらえリプレース方式をプログラム実行中に切り替えることで，破壊的干渉ミスが多発するプログラムの場合，リプレース方式がすぐに擬似 LRU から LTN 方式に切り替わり，LTN 方式をプログラム開始時から使用した場合に近似する性能向上率を実現することができる．一方，破壊的干渉ミスが少ないプログラムの場合，リプレース方式が擬似 LRU から LTN 方式に切り替わることはなく，LTN 方式を使用することによって発生する性能低下を防ぐことができる．

SDI 方式で重要となるのが，リプレース方式の切替え閾値である破壊的干渉ミス率である．ところが，正確な破壊的干渉ミス率を計算しようとすると除算器を使用しなければならず，それはハードウェア増加量，動作周波数ともに多大な悪影響を及ぼす．そこで SDI 方式では，破壊的干渉ミス数と総メモリアクセス数を用いて，以下の計算を行う．

$$\frac{Miss_\#_DI}{Total_Memory_Access_\#} > \frac{1}{M} \quad (1)$$

$$M = 2^n \quad (2)$$

$Miss_\#_DI$ ：破壊的干渉ミス数
$Total_Memory_Access_\#$ ：総メモリアクセス数

式 (1) を満たす場合は LTN 方式（Mode: 1）を選択し，満たさない場合は擬似 LRU（Mode: 0）を選択する．そして，式 (2) の n を変化させることで，擬似 LRU か LTN 方式かを決定する閾値を変化させる．ただし，n はプログラム実行中に動的に変化させるのではなく，静的に設定する．また，式 (1) の M を 2 のべき乗とすることで，総メモリアクセス数の下位 n ビットをおとして，それを破壊的干渉ミス数と比較するだけなので除算器を使用せずに済む．

一方，切り替える閾値 M を 2 のべき乗とすると，実質的に選択可能な M の候補が少なくなる．そのため，加算器などを配置して M を詳細に設定した方が，より良い性能をもたらす可能性が考えられる．しかし，閾値 M の最適値は実行するプログラムによって異なり，特に SMT プロセッサは異種プログラムの同時実行に適しているので，最適値を求めることが難しい．そのため本論文では，ハードウェア増加量，動作周波数低下の抑制を重視し，リプレース方式を切り替える閾値 M を 2 のべき乗とした．

以上を考慮し，SDI 方式の切替えアルゴリズムを以下に示す．

(1) リプレース方式として，擬似 LRU を初期設定とし，プログラムを開始する．
(2) プログラム実行中，つねに破壊的干渉ミス数を測定し，式 (1) を満たすかどうか確かめる．
(3) 式 (1) を満たす場合（破壊的干渉ミス率がある一定値を超えていた場合），リプレース方式を擬似 LRU から LTN 方式に切り替える．
(4) 一度，LTN 方式に切り替えたら，プログラム終了まで LTN 方式を使用する．

以上のように，リプレース方式が擬似 LRU の状態で破壊的干渉ミス率が増加した場合に限り，LTN 方式への切替えを行う．このような擬似 LRU から LTN 方式への切替えのほかに，リプレース方式が LTN 方式の状態で，破壊的干渉ミス率が減少し一定値以下になった場合，再度擬似 LRU に切り替えることが考えられる．しかしながら LTN 方式は破壊的干渉ミス率そのものを低下させる働きがあるので，破壊的干渉ミス率が一定値以下になったからといって，再度リプレース方式を擬似 LRU に切り替えると，破壊的干渉ミス率も再び増加してしまう可能性が高い．また，双方向の切替えとすると，擬似 LRU から LTN 方式への切替え閾値のほかに，LTN 方式から擬似 LRU への切替え閾値も決定しなければならないため，切替え閾値設定がいっそう難しくなる．これらを考慮し，本論文では，SDI 方式の切替えとして擬似 LRU から LTN 方式への 1 方向のみとした．

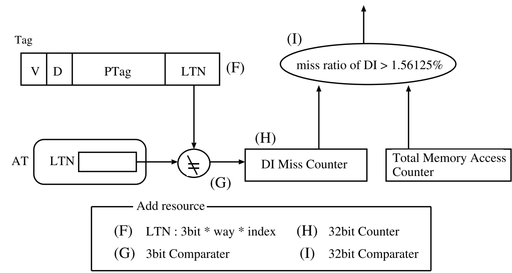

図 5 SDI 方式の実現方法
Fig. 5 The realization method of SDI strategy.

ここで，式 (2) の n を具体的に設定した SDI 方式を **SDI-n 方式**と呼ぶ．たとえば，$n = 6$ を設定した場合，SDI-6 方式となり，式 (1) の分母 M は 64 となり，破壊的干渉ミス率が 1/64（1.56125%）まで上昇したら，擬似 LRU から LTN 方式にリプレース方式を切り替える．

4.1.2 SDI 方式の設計

SDI 方式の切替えアルゴリズムの実現方法を**図 5** に示す．

まず，すべてのキャッシュラインのタグに LTN を記憶する領域を追加する（図 5 の (F)）．次に，キャッシュミスとなり，リプレースウェイが決定したら，そのキャッシュラインに保持している LTN とキャッシュアクセスした実スレッドの LTN を比較する（図 5 の (G)）．比較した結果，LTN が異なっていたら他のスレッドによるリプレースという判定となり，破壊的干渉ミス数を測定しているカウンタ（図 5 の (H)）をインクリメントする．そして，(H) が保持している破壊的干渉ミス数とパフォーマンスカウンタが保持している総メモリアクセス数を用いて式 (1) を計算し，設定した破壊的干渉ミス率（図 5 では 1.56125%）を超えているかどうかを判別する．超えていたら 1，超えていない場合は 0 を出力する（図 5 の (I)）．

この図 5 からの出力は，図 4 の入力となる．SDI 方式のために追加するハードウェア資源は図中の (F)，(G)，(H)，(I) となる．(G)，(H)，(I) の資源はキャッシュメモリにおいて，1 つのみ追加すればよいのでハードウェア増加量は問題ない．唯一，(F) の LTN に注意が必要である．たとえば，プロセッサで 8 個の論理スレッドを実行する場合は，最低 3 bit × ウェイ × インデックス分の領域を用意しなければならないので，ハードウェア増加量は大きくなる．つまり，SDI 方式

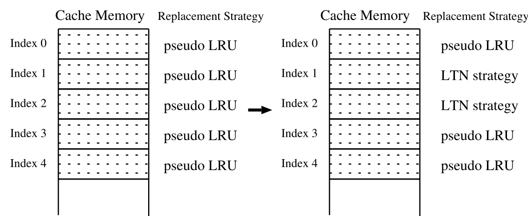

図 6　SON 方式の概要
Fig. 6　The overview of SON strategy.

のハードウェア増加量は (F) に起因する．

4.2　SON 方式

4.2.1　SON 方式の概要

キャッシュメモリには，アクセスが集中する領域（ホットスポット）が存在するため，破壊的干渉ミスが多い場合でも，それが特定のセットのみで多発している場合が考えられる．そこで**図 6** のように，セットごとにリプレース方式を動的に切り替える SON 方式を提案する．図 6 では，インデックス 1，2 に多数のスレッドがアクセスしていると仮定し，それらセットのみのリプレース方式を LTN 方式に切り替えている．このように特定のセットのみ，リプレース方式を LTN 方式とすることで，ホットスポットやスレッドのアクセス局所性に対応することができ，LTN 方式を適当なセットのみで使用することができる．

セットごとにリプレース方式を切り替える手段として，破壊的干渉ミス数をセットごとに測定する方法が考えられる．しかし，その手段による実現には，測定用カウンタがインデックス分必要となり，ハードウェア量が大幅に増加してしまうため現実的ではない．

そこで，セットにアクセスしているスレッド数に注目する．SDI 方式と同様にタグに LTN を持たせ，セット中の各ウェイの LTN を比較し，何種類のスレッドのデータがセット中に格納されているか確認する．そのスレッド数に応じて，使用するリプレース方式を決定する．

セット中に格納されているスレッド数 n を切替えパラメータとし，n 以上になったらリプレース方式を擬似 LRU から LTN 方式に切り替える SON 方式を**SON-n 方式**と呼ぶ．4 ウェイセットアソシアティブの場合，SON-2 方式，SON-3 方式，SON-4 方式が考えられ，各方式の切替えタイミングを**表 1** に示す．セット中のスレッド数が 1 つだけの場合，そのスレッドが 4 つすべてのウェイに格納されていることになる．その場合，LTN 方式を用いる必要がないため，リプレース方式として擬似 LRU を選択する．そのため，SON-1 方式は存在しない．

表 1　セット中のスレッド数によるリプレース方式の切替え
Table 1　The switch by occupied number of ways in set.

スレッド数	各スレッドに割り当てられるウェイ数	選択リプレース方式		
		SON-2	SON-3	SON-4
1	4	擬似 LRU	擬似 LRU	擬似 LRU
2	3：1	LTN 方式	擬似 LRU	擬似 LRU
2	2：2	LTN 方式	擬似 LRU	擬似 LRU
3	2：1：1	LTN 方式	LTN 方式	擬似 LRU
4	1：1：1：1	LTN 方式	LTN 方式	LTN 方式

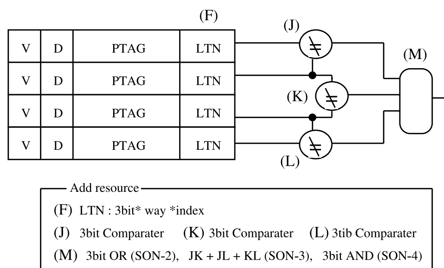

図 7　SON 方式の実現方法
Fig. 7　The realization method of SON strategy.

以上を考慮した SON 方式の切替えアルゴリズムを以下に示す．

(1)　リプレース方式として，擬似 LRU を初期設定とし，プログラムを開始する．
(2)　リプレースが必要なキャッシュミスが発生した場合，そのセット中に存在するスレッド数を確認する．
(3)　確認した結果，設定したスレッド数 n 以上のスレッド数がセット中に存在した場合，擬似 LRU から LTN 方式に切り替える．
(4)　一度，LTN 方式に切り替わっても，セット中のスレッド数が設定したスレッド数 n より小さくなった場合，再度 LTN 方式から擬似 LRU に切り替える．

以上のように，セット中に存在するスレッド数が設定数よりも小さくなった場合，再度擬似 LRU に切り替えることで，適当な期間のみ LTN 方式を活用できる．このように SON 方式は，LTN 方式を適当な場所と期間のみに適用することで，LTN 方式を単体で使用する以上の性能向上率を目指す．

4.2.2　SON 方式の設計

SON 方式の切替えアルゴリズムの実現方法を**図 7** に示す．図 7 の出力は図 4 の入力となる．

まず，既存のタグに LTN（図 7 の (F)）を追加する．次に，各ウェイの LTN を比較するため，比較器を追加する．ここで，各ウェイの LTN を正確に比較するためには，4 ウェイで 6 個，8 ウェイで 28 個の比較器が必要となる．それにともなって，LTN からの

表 2 (M) に追加される回路の真理値表
Table 2 The truth table of (M).

(J)	(K)	(L)	擬似スレッド数	SON-2	SON-3	SON-4
0	0	0	1	0	0	0
0	0	1	2	1	0	0
0	1	0	2	1	0	0
1	0	0	2	1	0	0
0	1	1	3	1	1	0
1	0	1	3	1	1	0
1	1	0	3	1	1	0
1	1	1	4	1	1	1

出力ファンアウト数を増加させなければならず，多くの LTN を比較することは，ハードウェア量，動作周波数の両方に悪影響を及ぼす．そこで，SON 方式では隣のウェイのみの LTN を比較することで，スレッド数を確認する．そうすることで，正確ではないがおおよそのスレッド数を調べることができ，なおかつ比較器や LTN の出力ファンアウト数を抑えることができる．ただし，すべての LTN 比較と，隣のウェイのみの LTN 比較で性能に大きな差が生じるようでは意味がない．しかし，4 ウェイ上で LTN の全比較と隣のウェイのみの LTN 比較の予備評価を行ったところ，性能誤差はわずかであり，ほぼ同等の性能であることが分かった．そのため，SON 方式では隣のウェイのみの LTN 比較とし，4 ウェイの場合は，図 7 のように (J)，(K)，(L) の 3 つの比較器を追加する．

最後に (J)，(K)，(L) からの入力をもとに，0（擬似 LRU）または 1（LTN 方式）を出力する (M) を追加し，SON 方式を実現する．ここでは，(M) に追加する回路を考える．前節で示した表 1 と，(J)，(K)，(L) から入力をもとに，SON-2，3，4 方式の出力を考えると，**表 2** のような真理値表が得られる．なお，表 2 の擬似スレッド数とは，隣のウェイのみの LTN 比較で判断したスレッド数を表す．この真理値表をもとに，回路を簡単化していくと，SON-2 方式は 3 bit OR 回路，SON-3 方式は以下の式 (3)，SON-4 方式は 3 bit AND 回路が得られる．

$$J \cdot K + J \cdot L + K \cdot L \quad (3)$$

つまり，(M) に追加する回路はどの方式でも簡単な組合せ回路となる．SON 方式を実現するためには，(F)，(J)，(K)，(L)，(M) の回路が新たに必要となるが，ハードウェア量増加の原因となるのは，SDI 方式同様 (F) の追加 LTN になると考える．

4.3 動的切替え方式の予備評価と検討

提案した 2 つの動的切替え方式について，本評価の前に予備評価☆を行い，適切なパラメータを検討する．

まず，SDI 方式について検討する．切替えの閾値である破壊的干渉ミス率の適切な値を求めるため，まず，破壊的干渉ミス率として，2 のべき乗ではない 0.5〜20%を設定し，予備評価を行った．その結果，1〜3%がより良い結果をもたらした．その原因として，破壊的干渉ミスが多発するプログラムの場合，そのミス率は 3%以上になる場合が多く，一方，破壊的干渉ミスが少ないプログラムの場合，そのミス率は 1%未満となる場合がほとんどであった．そのため，切替え閾値として 0.5%を設定した場合，破壊的干渉ミスが少ないプログラムまで擬似 LRU から LTN 方式に切り替わってしまい，性能低下が拡大した．また，切替え閾値として 3%以上を設定した場合，破壊的干渉ミスが多いプログラムにおいてもなかなかリプレース方式が切り替わらず，LTN 方式の利点を活かせなかった．これらの結果から，SDI 方式としては，SDI-5 方式（切替え閾値：3.125%），SDI-6 方式（切替え閾値：1.56125%）が適切であり，性能低下の抑制および性能向上が見込めると判断した．

次に SON 方式について検討する．まず，すべての LTN 比較（正確なスレッド数）と隣のウェイのみの LTN 比較（擬似的なスレッド数）の性能誤差を調査するため，4 ウェイセットアソシアティブのキャッシュメモリ上で，それぞれの SON-2，3，4 方式を設計し，予備評価を行った．その結果，両者の性能誤差はわずかであり，ほぼ同等の性能であることが分かった．次に，切替えパラメータとして適切なセット中に存在する擬似スレッド数を求めるため，SON-2，3，4 方式間の性能比較を行った．結果，SON-3，4 方式の性能向上率が良く，SON-4 方式の性能向上率が最も高かった．SON-2 方式では多くのセットで切替えが発生してしまい，SON-3，4 方式と比べると性能は低かった．そのため，適切な切替えパラメータとして，ウェイ数の半分を超える値が妥当と考え，4 ウェイセットアソシアティブでは SON-3，4 方式，8 ウェイセットアソシアティブでは SON-5，6，7，8 方式が適切であり，性能向上が見込めると判断した．

5. 評 価

本章では，提案したリプレース動的切替え方式の性能，ハードウェア量，動作周波数について評価する．

5.1 シミュレーションパラメータ設定

性能評価には，OChiMuS PE をシミュレートする実行駆動型シミュレータ *MUTHASI*（*MUltiTHreaded Architecture SImulator*）[3] を用いた．評価時のプロセッサパラメータを**表 3** に示す．このプロセッサパ

☆ 定量的な予備評価を行ったが，紙面の都合で，予備評価のまとめのみを掲載している．

表 3 シミュレーション時のプロセッサパラメータ
Table 3 The processor configuration of simulation.

PC (AT#)	2
Fetch Buffer Size	16
Dispatch Queue Size	32
Reorder Buffer Size	128
Normal Reservation Station Size	Simple ALU : 8 Complex ALU : 4
LD/ST Reservation Station Size	8
Branch History Table Size	1024 (gshare)
Integer ALU	Simple ALU : 3 Complex ALU : 2
FPU	Simple ALU : 2 (delay 4 cycle) Complex ALU : 1 (Mult 17 cycle, Div 30 cycle)
Branch Unit	1
Speculation Depth	4

表 4 シミュレーション時のキャッシュメモリパラメータ
Table 4 The cache memory configuration of simulation.

Capacity	L1-I-Cache	16 KB
	L1-D-Cache	8 KB, 32 KB, 128 KB
	L2-Cache	512 KB
Way	L1-I-Cache	1
	L1-D-Cache	4
	L2-Cache	8
Line Size	L1-I-Cache	32 B
	L1-D-Cache	32 B
	L2-Cache	64 B
Latency	L1-I-Cache	1 cycle
	L1-D-Cache	2 cycle
	L2-Cache	20 cycle

ラメータは文献 2) と同じ設定である．実スレッド数（AT 数）は 2 である．

評価には，LU 分解（サイズ：128 × 128），行列乗算（サイズ：256 × 256），RADIX ソート（個数：16384）を用いた．LU 分解，RADIX ソートは SPLASH-2 [5] より採用した．これらのプログラムを並列化し，それぞれ単一プログラムから 8 個の論理スレッドを生成し，評価する．次に，異種プログラムの同時実行を評価するため，LU 分解と RADIX ソート，LU 分解と行列乗算，RADIX ソートと行列乗算をそれぞれ実スレッドに割り当てて評価する．異種プログラムの評価では，単一プログラムから 4 個の論理スレッドを生成し，計 8 個のスレッドを実行する．なお，異種プログラムの同時実行では，一方のプログラムのみが実行しているという状況を極力避けるため，行列乗算の計算サイズを 128 × 128 にしている．また，プログラムの作成はスレッドライブラリ MULiTh [4]，binutils-2.13☆，gcc-3.2☆☆，newlib1.9.0 を用いた．各プログラムのスレッド分割方法，スレッド間のデータ共有具合，メモリアクセスパターンは文献 2) を参照されたい．

次に，キャッシュメモリのパラメータを**表 4** に示す．L1-D-キャッシュメモリとして，ウェイ数 4，ラインサイズ 32 B を設定した．4 ウェイ程度の場合，リプレース方式として完全な LRU を実装しても，ハードウェア量は擬似 LRU と大きく変わらない．しかし，完全な LRU と擬似 LRU の性能誤差はわずかであり，ほぼ同一の性能であることが分かっている [2]．よって，本評価では，基準とするリプレース方式として，利用率が高く [6]，なおかつハードウェア量を抑えることのできる擬似 LRU を選択する．

☆ OChiMuS PE のスレッド制御命令を利用可能にしたもの．
☆☆ 最適化オプションは-O2 を設定した．

また，本評価では，キャッシュ容量として 8 KB，32 KB，128 KB を選択した．これらの容量は，本評価のプログラム規模を考慮し設定している．詳細は文献 2) を参照されたい．

他のパラメータについては，文献 2)，6) を参考に設定し，性能に影響しないように各パラメータを比較的大きな値に設定した．このとき，L1-D-キャッシュメモリのリプレース方式として，擬似 LRU，LTN-1 方式（LTN），SDI-6 方式（SDI），SON-4 方式（SON）を実装し，プログラムを実行した．

5.2 実 行 結 果

擬似 LRU と LTN 方式および本論文で提案した動的切替え方式の性能を比較した．各プログラムの性能向上率を**図 8** に示す．このグラフは，擬似 LRU のサイクル数を基準とし，LTN 方式と提案した動的切替え方式の性能向上率を示している．

本評価のプログラム特性として，LU 分解はスレッド間の破壊的干渉が少ないプログラムであり，逆に行列乗算は破壊的干渉が多いプログラム，また RADIX ソートはその中間程度のプログラムである．そのため，LU 分解や LU 分解 & RADIX ソートにおいて，LTN 方式および各動的切替え方式の性能向上率は低く，擬似 LRU と比較し性能が低下してしまった場合もある．逆に行列乗算や RADIX ソート&行列乗算において，各方式の性能が高くなっている．

キャッシュ容量 8 KB では，LTN 方式の性能が低下している場合が多いが，動的切替え方式ではその性能低下を抑制している．たとえば，LU 分解 & RADIX ソートで LTN 方式の性能向上率は 0.924 倍であるが，SDI 方式は 0.982 倍，SON 方式は 0.997 倍となっている．

キャッシュ容量 32 KB では，LTN 方式が行列乗算で 1.240 倍，RADIX ソート&行列乗算で 1.082 倍を示すなど性能向上率が高い．それにともなって動的切替え方式の性能も向上している．また，RADIX ソート&行列乗算の SDI 方式で 1.095 倍，SON 方式で 1.165

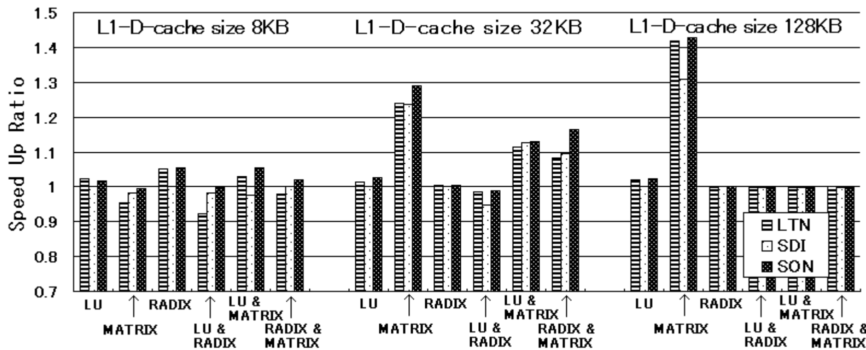

図 8 各リプレース方式による性能向上率

Fig. 8 The speed up ratio by replacement strategies.

表 5 単一プログラムの実行結果

Table 5 The result of all experiments of a program.

		LU			MATRIX			RADIX		
		cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R
L1-D cache size 8 KB	擬似 LRU	20,011,179	1.000	94.93%	304,959,599	1.000	58.62%	39,206,063	1.000	99.93%
	LTN	19,579,758	1.022	94.94%	319,834,358	0.953	55.61%	37,328,615	1.050	99.97%
	SDI	20,011,179	1.000	94.93%	310,845,067	0.981	56.66%	39,206,063	1.000	99.93%
	SON	19,693,420	1.016	94.94%	306,392,265	0.995	57.75%	37,128,389	1.056	99.99%
L1-D cache size32 KB	擬似 LRU	19,248,962	1.000	98.19%	346,332,673	1.000	55.32%	37,256,023	1.000	99.97%
	LTN	19,000,196	1.013	98.21%	279,284,903	1.240	64.41%	37,127,307	1.003	99.96%
	SDI	19,248,962	1.000	98.19%	279,878,305	1.237	64.01%	37,256,023	1.000	99.97%
	SON	18,741,264	1.027	99.13%	268,698,431	1.289	66.55%	37,109,821	1.004	99.97%
L1-D cache size128 KB	擬似 LRU	18,718,283	1.000	99.68%	344,176,602	1.000	58.79%	37,583,351	1.000	99.97%
	LTN	18,338,424	1.021	99.74%	242,458,443	1.420	75.65%	37,659,372	0.998	99.97%
	SDI	18,718,283	1.000	99.68%	262,831,843	1.309	70.33%	37,583,351	1.000	99.97%
	SON	18,300,801	1.023	99.75%	241,187,438	1.427	75.96%	37,573,482	1.000	99.97%

*SUR: Speed up ratio **L1D_H_R: L1 D-cache Hit Ratio

表 6 異種プログラムの実行結果

Table 6 The result of all experiments of programs.

		LU & RADIX			LU & MATRIX			RADIX & MATRIX		
		cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R
L1-D cache size 8 KB	擬似 LRU	39,761,514	1.000	95.13%	60,533,836	1.000	72.69%	49,407,371	1.000	50.58%
	LTN	43,043,571	0.924	89.88%	58,744,706	1.030	73.87%	50,442,593	0.979	49.93%
	SDI	40,497,092	0.982	93.31%	62,090,271	0.975	70.83%	49,407,371	1.000	50.58%
	SON	39,888,445	0.997	95.03%	57,440,914	1.054	75.47%	48,381,425	1.021	51.76%
L1-D cache size32 KB	擬似 LRU	38,943,112	1.000	97.29%	59,801,400	1.000	74.19%	49,346,307	1.000	50.65%
	LTN	39,544,058	0.985	95.55%	53,663,857	1.114	78.24%	45,596,141	1.082	56.64%
	SDI	41,134,110	0.947	93.17%	53,109,429	1.126	78.82%	45,071,018	1.095	56.87%
	SON	39,324,343	0.990	96.50%	52,984,856	1.129	78.57%	42,359,242	1.165	59.08%
L1-D cache size128 KB	擬似 LRU	38,016,458	1.000	99.38%	43,369,476	1.000	99.70%	33,614,955	1.000	99.79%
	LTN	38,120,781	0.997	99.30%	43,404,974	0.999	99.69%	33,626,252	1.000	99.50%
	SDI	38,106,518	0.998	99.29%	43,410,083	0.999	99.55%	33,638,202	0.999	99.48%
	SON	38,140,877	0.997	99.32%	43,415,688	0.999	99.65%	33,633,506	0.999	99.79%

*SUR: Speed up ratio, **L1D_H_R: L1 D-cache Hit Ratio

倍など，プログラムによっては LTN 方式よりも動的切替え方式の方が高い性能向上率を示している．

キャッシュ容量 128 KB では，行列乗算で LTN 方式，動的切替え方式の性能向上率が高くなっている．一方，行列乗算以外のプログラムの性能向上率は大きく変化していない．

5.3 考　　察

実行したプログラムのすべてのサイクル数，性能向上率，L1-D-キャッシュヒット率を**表 5**，**表 6** に示す．

キャッシュ容量が 8 KB と少ない場合，キャッシュミスの要因として，破壊的干渉ミスの割合が低く，容量ミスの割合が高くなる．そのため，リプレースウェイを制限し，破壊的干渉ミスを抑える LTN 方式を用いても性能向上率は低く，逆にリプレースウェイ制限によって性能低下が発生してしまう．しかし，その性能低下を動的切替え方式によって緩和している．特に，SON 方式の効果が強く，擬似 LRU よりも性能が向上している場合が多い．つまり，キャッシュ容量が少ない場合，LTN 方式を用いるよりも，特定のセットを部分的に LTN 方式に切り替える SON 方式の方が有効であり，性能向上が見込める．

キャッシュ容量 32 KB では，もともと LTN 方式の性能が高い．32 KB，128 KB とキャッシュ容量を大きくすると，破壊的干渉ミスの割合が大きくなるため，

表 7 SON 方式の実行経過におけるリプレース方式の採用状況（破壊的干渉ミスの多いプログラム）
Table 7 The replacement strategy condition of execution process of SON strategy (program of much DI).

		Execution Time (cycle)							
		0 (Start)	40,000,000	80,000,000	120,000,000	160,000,000	200,000,000	240,000,000	268,698,431(End)
Index Number	Index 0	0	1	1	1	0	1	0	1
	Index 32	0	1	1	0	1	1	1	1
	Index 64	0	1	1	1	1	1	1	1
	Index 96	0	1	1	1	1	1	1	1
	Index 128	0	1	1	1	1	1	0	0
	Index 160	0	1	0	1	1	1	1	1
	Index 192	0	1	1	1	1	1	1	1
	Index 224	0	1	1	1	0	1	0	1

0：擬似 LRU，1：LTN 方式

各方式が擬似 LRU よりも有効になる．また，キャッシュ容量が増えるとインデックス数が増え，それぞれのセットに適するリプレース方式が変わってくる．異種プログラム実行の場合，アクセスするセットやスレッド間の競合が増えるため，それが顕著になる．そのため，32 KB でも SON 方式が有効に働き，破壊的干渉ミスが多発する行列乗算や RADIX ソート&行列乗算の同時実行などで，LTN 方式よりも高い性能向上率が実現できた．

キャッシュ容量 128 KB では，多くのプログラムでキャッシュヒット率がほぼ 100%となり，リプレース方式が性能に影響しなくなっている．唯一，行列乗算のみキャッシュヒット率が飽和しておらず，擬似 LRU と比較し，LTN 方式および提案した動的切替え方式の性能向上率が高くなっている．

提案した動的切替え方式は，キャッシュ容量の大小，プログラムの種類にかかわらず，LTN 方式以上の性能向上を示した．SDI 方式は，8 KB の LTN 方式による性能低下の抑制について効果は現れたものの，32 KB，128 KB では LTN 方式の性能向上率に達しない場合が多かった．キャッシュ容量が大きい場合，LTN 方式が有効となるが，SDI 方式は破壊的干渉ミス率が設定した値（本評価では，1.56125%）を超えないとリプレース方式が切り替わらないため，LTN 方式の利点をさらに生かすことができなかった．一方，SON 方式の性能向上率は多くプログラムにおいて LTN 方式よりも高く，8 KB では，LTN 方式で発生する性能低下を抑え，さらに 32 KB，128 KB では LTN 方式よりも高い性能向上を実現した．

最後に，提案した 2 つの動的切替え方式の特性を示し，比較する．SDI 方式は，擬似 LRU から LTN 方式への切替えしか行わず，LTN 方式から擬似 LRU への切替えを行わない．本論文で実行したプログラムは，均等にスレッド分割しているため[2)]，破壊的干渉ミスはプログラム全体で比較的均一に発生する．そのため，一度破壊的干渉ミスが多いプログラムと判断された場合，その後も，そのプログラムは破壊的干渉ミスを多発する可能性が高く，1 方向の切替えでも性能向上が期待できる．逆に，LTN 方式の状態で再び擬似 LRU に切り替えてしまうと，低下した破壊的干渉ミス率が再び増加し，結果として性能低下を引き起こしてしまう可能性が高い．しかしながら，1 方向の切替えでは，時間軸方向に対して自由にリプレース方式を切り替えることができないため，実行途中で破壊的干渉ミスが急激に増減するプログラムなどには正確に対処できない．

一方，SON 方式はセットごとにリプレース方式を切り替えることができ，さらに一度 LTN 方式に切り替わってもそのセットの状況に応じ，再度擬似 LRU にリプレース方式を切り替えることができる．この SON 方式のプログラム実行経過におけるセットごとのリプレース方式の採用状況を**表 7**，**表 8** に示す．破壊的干渉ミスの多いプログラムと少ないプログラムを比較するため，破壊的干渉ミスの多いプログラムとして行列乗算を，少ないプログラムとして LU 分解を選択した．行列乗算は 40,000,000 サイクルごとに，LU 分解は 3,000,000 サイクルごとにサンプリングした．また，どちらもキャッシュサイズは 32 KB とし，32 セットごとにサンプリングした．なお，表中の 0 は擬似 LRU を表し，1 は LTN 方式を表す．

表 7 は，破壊的干渉ミスの多いプログラムのため，多くのセットにおいて LTN 方式を採用しており，実行経過によっては若干擬似 LRU を採用している．一方，表 8 は破壊的干渉ミスの少ないプログラムなので，擬似 LRU を採用しているセットが多く，LTN 方式を採用しているセットが少ない．このように，SON 方式はプログラムの特性に応じて，リプレース方式を適切に切り替えていることが分かる．また，セットごとの切替え（空間軸方向の切替え）のほかに，同じセットでも実行経過によって採用しているリプレース方式が変化しており，時間軸方向においても正確に切り替えている．つまり，SON 方式は空間軸方向と時間軸方向についてリプレース方式を自由に切替え可能であるという特徴がある．この特徴を活用することで，SON

表 8 SON 方式の実行経過におけるリプレース方式の採用状況（破壊的干渉ミスの少ないプログラム）
Table 8 The replacement strategy condition of execution process of SON strategy (program of low DI).

		Execution Time (cycle)							
		0 (Start)	3,000,000	6,000,000	9,000,000	12,000,000	15,000,000	18,000,000	18,741,264 (End)
Index Number	Index 0	0	0	0	0	0	0	0	0
	Index 32	0	1	0	1	0	0	0	0
	Index 64	0	0	0	0	0	0	0	0
	Index 96	0	1	1	0	1	0	0	0
	Index 128	0	0	0	0	1	0	0	0
	Index 160	0	1	0	0	0	0	0	0
	Index 192	0	0	1	0	1	1	0	0
	Index 224	0	0	1	0	0	0	0	0

0：擬似 LRU，1：LTN 方式

表 9 各動的切替え方式のハードウェア量
Table 9 The hardware cost of dynamic switch strategies.

スライス数	擬似 LRU	LTN	SDI	SON
SMT	8366			
Cache	2937	2888	3248	3200
Block RAM #	32	32	32	32
SMT + Cache	11303	11254	11614	11566
増加率 (%)	0	−0.44	2.68	2.27

表 10 各動的切替え方式の動作周波数
Table 10 The frequency of dynamic switch strategies.

	擬似 LRU	LTN	SDI	SON
最長パス (ns)	19.769	19.467	19.77	20.033
動作周波数 (MHz)	61.389	62.125	61.125	61.047
低下率 (%)	0	−1.19	0.43	0.56

方式は様々なキャッシュ容量やプログラム特性にも柔軟に対応でき，LTN 方式や SDI 方式以上の性能向上率を実現している．

5.4 ハードウェア量と動作周波数

提案した各動的切替え方式の具体的なハードウェア量と動作周波数を見積もるため，Verilog-2000 と Xilinx 社の ISE6.2.03i を用いて，各方式を実装した．実装したキャッシュメモリ構成は，キャッシュ容量 32 KB，ウェイ数 4，ラインサイズ 32 B，インデックス数 256 であり，リプレース方式は性能評価と同じく擬似 LRU，LTN-1 方式（LTN），SDI-6 方式（SDI），SON-4 方式（SON）を実装した．実装した結果を**表 9** に示す．

SMT プロセッサのハードウェア量は現在著者らが設計・開発している FPGA 向け SMT プロセッサ[8] を参考にした．キャッシュメモリのデータ部分は，各方式とも 32 個の Block RAM を使用した．しかし，Block RAM はその一部分しか使用していない場合でも，1 つと換算されてしまうので，タグ領域を Block RAM で実現すると使用していない無駄な Block RAM 領域が大きくなり，各方式の Block RAM 個数の差が大きくなってしまう．そのため，タグ領域は，スライスを用いて実現する分散 RAM を使用した．

擬似 LRU と比べると，LTN 方式のハードウェア量が減少し，他方式のハードウェア量は増加している．SDI 方式は，タグに追加した LTN およびカウンタが主な原因でハードウェア量が増加している．SON 方式も SDI 方式同様に，タグに追加した LTN が増加の主要因となっている．しかし，プロセッサを含めたチップ全体で考えると，SDI 方式のハードウェア増加率は 2.68%，SON 方式のハードウェア増加率は 2.27%となり，擬似 LRU と比較しても大幅な増加量ではないことが分かる．

次に提案した動的切替え方式の動作周波数を**表 10** に示す．提案方式の実装対象が L1-D-キャッシュメモリであるため，動作周波数の低下は性能に大きな悪影響を及ぼす．しかしながら，擬似 LRU と比較し，SDI 方式の動作周波数低下率は 0.43%，SON 方式は 0.56% となり，どちらも低下率は 1%未満である．つまり，提案した各動的切替え方式の動作周波数の低下について問題はないことが分かる．

6. 関 連 研 究

関連研究として，Suh らによる Partitioning Decision 方式と Modified LRU [11] がある．本論文ではキャッシュリプレース方式を動的に切り替えているが，Suh らはマルチスレッドプロセッサの L2-キャッシュにおいて，スレッドの実行状況に応じて扱うウェイ数を動的に変化させ，最適化している．これは，キャッシュ容量の大きい L2-キャッシュのあるセットにおいて，そこにアクセスする確率の高いスレッドに多くのウェイを割り当て，性能向上を目指している．この方式は，比較的空き容量の多い L2-キャッシュにおいて効果がある．しかし，キャッシュ容量が小さくなると，各スレッドは多くのセットにアクセスするため，動的にウェイを変化させる効果が薄くなる．たとえば，L2-キャッシュ容量が 1 MB の場合，14.2%の性能向上率を示しているが，256 KB の場合はわずか 0.1%の性能向上しか示していない[11]．よって，この方式は本論文で対象とする L1-D-キャッシュのように，キャッシュ容量が小さいキャッシュメモリについて効果は薄いと

考える．

マルチスレッドプロセッサ向けのキャッシュリプレース方式として，文献 9)，10)，12) がある．山崎らによる動的スレッドアソシアティブ方式[9] は．マルチスレッドプロセッサのキャッシュのウェイをスレッドごとに限定することで性能向上を目指しており，5〜8%の性能向上率を示している．しかし，具体的な実現方法が示されておらず，さらに実装したときのハードウェア増加量，動作周波数低下率の検討がない．本研究の動的切替え方式は，性能向上率として最大 42.7%を示し，実装することで具体的なハードウェア増加量，動作周波数低下率を示した．また，佐藤ら[10]，Sigmund ら[12] はスレッドに優先度をつけて，優先度が高いスレッドのデータを優先的にキャッシュに保持するリプレース方式を提案している．これらの研究は，組み込みやメディア処理に特化しており，特定スレッドの処理時間の向上を目標としている．これに対し，我々の動的切替え方式はすべてのスレッドを対象とし，プログラム実行全体の性能向上を実現している．

本論文は性能向上を目標とし，キャッシュリプレース方式を動的に切り替えたが，小宮らは低消費電力を目標とし，キャッシュラインの電力の ON/OFF を動的に切り替えている[15]．キャッシュラインに対する参照密度をプログラム実行中に動的に監視し，一定値を上回るラインのみを活性化し，それ以外を非活性化させることで低消費電力を実現している．今後，これら研究のようにプログラムの実行時情報を用いることで，ハードウェアを動的に最適化する研究がより活発化するのではないかと考える．

7. 終わりに

本論文では，SMT プロセッサの性能低下の原因として，キャッシュラインのスレッド間競合を取り上げた．スレッドの競合ミスを抑えるキャッシュリプレース方式として LTN 方式があるが，プログラムやキャッシュ容量によっては性能低下を引き起こす．そこで，本論文ではプログラム実行中に擬似 LRU と LTN 方式の動的切替えを行うことで，LTN 方式の性能低下の抑制，さらなる性能向上を目指した．動的切替え方式として，破壊的干渉ミス率を切替えパラメータとする SDI 方式，セットごとにリプレース方式を切り替える SON 方式を提案し，設計した．評価の結果，各動的切替え方式は有効に動作し，LTN 方式で発生した性能低下を抑え，さらに擬似 LRU と比べ最大 1.427 倍と LTN 方式よりも高い性能向上をもたらした．また，各動的切替え方式を実装しハードウェアコストを見積もった結果，どちらの方式も，プロセッサとキャッシュメモリを含んだチップ全体で 3%未満とわずかなハードウェア増加量で実現できることを示した．

今後の課題として，OChiMuS PE 以外の SMT プロセッサアーキテクチャにおける各方式の適用，評価がある．また，SMT プロセッサ向けのリプレース方式として，LTN 方式のほかに，スレッド間の共有データの有効活用を目指したリプレース方式がある[2]．今後は，そのリプレース方式を含めた 3 つの方式の動的切替えの検討，評価を行いたい．

謝辞　本研究の遂行にあたり，貴重なご助言をいただいた東京大学大学院笹田耕一助手に感謝いたします．

また，本研究の一部は，日本学術振興会科学研究費補助金（No.19・8474）によるものです．

参 考 文 献

1) Tullsen, D., Eggers, S. and Levy, H.: Simultaneous multithreading: Maximizing on-chip parallelism, *Proc. 22nd Annual International Symposium on Computer Architecture* (*ISCA-22*), pp.392–403 (1995).

2) 小笠原嘉泰，佐藤未来子，笹田耕一，内倉　要，並木美太郎，中條拓伯：SMT プロセッサ向けキャッシュメモリリプレース方式，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG12 (ACS15), pp.119–132 (2006).

3) 河原章二，佐藤未来子，並木美太郎，中條拓伯：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 2002，Vol.2002, No.18, pp.1–8 (2002).

4) 笹田耕一，佐藤未来子，河原章二，加藤義人，大和仁典，中條拓伯，並木美太郎：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：コンピューティングシステム，Vol.44, No.SIG11 (ACS3), pp.215–225 (2003).

5) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Annual International Symposium on Computer Architecture* (*ISCA-22*), pp.24–36 (1995).

6) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishers (2002).

7) Handy, J.: *The Cache Memory book, 2nd Edition*, Academic Press (1998).

8) 加藤義人，大和仁典，小笠原嘉泰，佐藤未来子，笹田耕一，内倉　要，並木美太郎，中條拓伯：SMT プロセッサの FPGA への実装と評価，先進

的計算基盤システムシンポジウム SACSIS 2005, pp.239–240 (2005).

9) 山崎真也，本多弘樹，弓場敏嗣：マルチスレッドアーキテクチャにおけるデータキャッシュ構成方式の提案，情報処理学会研究報告，1998-HPC-93, Vol.1998, No.93, pp.79–84 (1998).

10) 佐藤純一，内山真郷，伊藤　務，山崎信行，安西祐一郎：リアルタイム処理用マルチスレッディングプロセッサの優先度に基づくキャッシュサブシステム，情報処理学会研究報告，2001-ARC-143, Vol.2001, No.143, pp.37–42 (2001).

11) Suh, G.E., Rudolph, L. and Devadas, S.: Dynamic Partitioning of Shared Cache Memory, *The Journal of Supercomputing Architecture*, pp.7–26 (2004).

12) Sigmund, U. and Ungerer, T.: Memory Hierarchy Studies of Multimedia-enhanced Simultaneous Multithreaded Processors for MPEG-2 Video Decompression, *Workshop on Multi-Threaded Execution, Architecture and Compilation 2000* (*MTEAC-2000*), pp.1–9 (2000).

13) 内倉　要，笹田耕一，佐藤未来子，加藤義人，大和仁典，中條拓伯，並木美太郎：SMT プロセッサにおけるスレッドスケジューラの開発，情報処理学会論文誌：コンピューティングシステム，Vol.46, No.SIG12 (ACS11), pp.150–160 (2005).

14) Lo, J.L., Barroso, L.A., Eggers, S.J., Gharachorloo, K., Levy, H.M. and Parekh, S.S.: An Analysis of Database Workload Performance on Simultaneous Multithreaded Processors, *Proc. 25th Annual International Symposium on Computer Architecture* (*ISCA-25*), pp.39–50 (1998).

15) 小宮礼子，井上弘士，村上和彰：待機ラインへの参照密度に基づく低リーク・キャッシュの高性能化，先進的計算基盤システムシンポジウム SACSIS 2006, pp.3–12 (2006).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 7 日採録)

小笠原嘉泰（学生会員）

1982 年生まれ．2003 年育英（現，サレジオ）工業高等専門学校情報工学科卒業．2005 年東京農工大学工学部情報コミュニケーション工学科卒業．2006 年同大学院工学教育部情報コミュニケーション工学専攻博士前期課程修了．現在，同大学院工学府電子情報工学専攻博士後期課程に在籍．2007 年 4 月より，日本学術振興会特別研究員 DC2，サレジオ工業高等専門学校非常勤講師．マルチスレッドプロセッサ，キャッシュメモリ，FPGA，再構成技術に興味を持つ．電子情報通信学会学生会員．

佐藤未来子（正会員）

1966 年生まれ．1990 年東京農工大学大学院工学研究科修了．同年（株）日立製作所入社，サーバシステムの設計・性能評価等に従事．2006 年東京農工大学大学院工学教育部電子情報工学専攻博士後期課程修了．博士（工学）．現在，東京農工大学大学院研究生．マルチスレッドプロセッサ，オペレーティングシステムに関する研究に興味を持つ．

並木美太郎（正会員）

1984 年東京農工大学工学部数理情報工学科卒業．1986 年同大学院修士課程修了．同年 4 月（株）日立製作所基礎研究所入社．1988 年東京農工大学工学部数理情報工学科助手．1989 年電子情報工学科助手．1993 年 11 月電子情報工学科助教授．1998 年 4 月情報コミュニケーション工学科助教授．現在，東京農工大学大学院共生科学技術研究院教授．博士（工学）．オペレーティングシステム，言語処理系，ウィンドウシステム等のシステムソフトウェア，並列処理，コンピュータネットワークおよびテキスト処理の研究・開発・教育に従事．ACM，IEEE 各会員．

中條　拓伯（正会員）

1961 年生まれ．1985 年神戸大学工学部電気工学科卒業．1987 年同大学院工学研究科修了．1989 年同大学工学部助手の後，1998 年より 1 年間 Illinois 大学 Urbana-Champaign 校 Center for Supercomputing Research and Development （CSRD）にて Visiting Research Assistant Professor を経て，現在，東京農工大学大学院共生科学技術研究院准教授．プロセッサアーキテクチャ，並列処理，クラスタコンピューティング，高速ネットワークインタフェースに関する研究に従事．電子情報通信学会，IEEE CS 各会員．博士（工学）．