SMT プロセッサ向けキャッシュメモリリプレース方式

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 プロセッサは，複数のスレッドで資源（各種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，スレッドの干渉によるミスが増加し，性能が低下する問題がある．そこで本論文は，SMT プロセッサのキャッシュメモリにおける問題点と利点に着目し，スレッド間の競合ミスを抑えるリプレース方式とスレッドが共有するデータをキャッシュメモリ上に保持するリプレース方式を提案し，設計した．評価の結果，各リプレース方式は有効に動作し，従来の擬似 LRU と比べ，行列乗算で最大 1.42 倍の性能向上をもたらした．また，各リプレース方式を実装しハードウェアコストを見積もった結果，わずかなハードウェア増加量で各方式を実現できることを示した．

Replacement Strategies of Cache Memory for an SMT Processor

Yoshiyasu Ogasawara,[†] Mikiko Sato,[†] Koichi Sasada,[†,☆]
Kaname Uchikura,[†,☆☆] Mitaro Namiki[†] and Hironori Nakajo[†]

An SMT processor aims to gain higher processor performance by executing parallel threads. However, the increasing cache misses caused by sharing the cache memory brings performance degradation. In this paper, we have proposed new cache replacement strategies focus attention on SMT processor's cache memory. As a result, new cache replacement strategy shows 1.42 times as high performance as conventional replacement strategy. And new cache replacement strategies can be implemented with low hardware cost.

1. はじめに

近年，プロセッサアーキテクチャの 1 つとして，スレッドレベル並列性（TLP: Thread Level Parallelism）に注目が集まっている．

マルチスレッドアーキテクチャに，SMT（Simultaneous MultiThreading）プロセッサ[1] がある．これは，1 チップで複数のハードウェアコンテキストを持ち，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有し，資源を有効に利用しつつ性能向上を目指している．

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してしまうという短所がある．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ヒット率の低下，演算器の枯渇がある．そのため，これらが原因でプロセッサの性能が低下してしまう場合がある．

本論文では，SMT プロセッサの性能向上の指針として，キャッシュメモリに着目する．SMT プロセッサの特性を活かしたキャッシュリプレース方式を提案，適用することにより，スレッド間におけるキャッシュライン競合を緩和し，性能向上を図る．また，提案するリプレース方式を実装し，具体的なハードウェア増加量を見積もる．

なお，本論文では，マルチスレッドアーキテクチャにおいて 1 つの命令流を処理する単位を**実スレッド**（*AT: Architecuture 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*）として定義する．

次章では，本論文で前提とする SMT プロセッサとその問題点を示す．3 章では，本論文の目標を述べる．4 章では，まず提案するリプレース方式のベースとなる擬似 LRU について述べ，次に SMT プロセッサの特性を活かした 2 つのリプレース方式を提案，設計，

† 東京農工大学大学院工学府
Graduate School of Technology, Tokyo University of Agriculture and Technology
☆ 現在，東京大学大学院情報理工学系研究科
Presently with Graduate School of Information Science and Technology, The University of Tokyo
☆☆ 現在，株式会社小松製作所
Presently with Komatsu Corporation

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. SMT プロセッサアーキテクチャとその問題点

本章では，本論文で前提とする SMT プロセッサとスレッドライブラリについて述べる．また，SMT プロセッサとキャッシュメモリの構成，その問題点を示す．

2.1 OChiMuS PE と MULiTh

OChiMuS とは，著者らが研究，開発を進めているオンチップマルチ SMT（On Chip Multi SMT: OChiMuS）プロセッサである．*OChiMuS PE* は SMT プロセッサであり，MIPS アーキテクチャをベースにし，マルチスレッドを提供するため，モジュールや命令が新たに拡張されている．この SMT プロセッサの詳細については文献 2) を参照されたい．

また，SMT アーキテクチャ向けのスレッドライブラリとして，POSIX Thread 仕様に準拠した *MULiTh*（*Thread Library for Multithreaded Architecture*）を用いる．MULiTh は OChiMuS PE と協調し，プロセッサ内部の**論理スレッド番号**（*LTN: Logical Thread Number*）を利用することで，実スレッドを仮想化し，スレッドの生成，消滅，および同期などのプロセッサが持つスレッド制御命令を容易に利用することを可能としている．詳細は文献 3) を参照されたい．

2.2 キャッシュメモリ構成と問題点

OChiMuS PE とキャッシュメモリの構成を**図 1** に示す．SMT アーキテクチャでは，各実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，実スレッド分用意する．本論文では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 プロセッサのキャッシュメモリでは，1 スレッドのみ実行する場合には起こらない要因でキャッシュミスが発生する．第 1 に，実スレッドの増加による容量ミス（Capacity misses）があげられる．シングルスレッドであれば十分であるキャッシュ容量も，複数の実スレッドが並列実行するため，不十分になる．第 2 の要因として，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実スレッド間の干渉による競合ミス（Conflict misses）があげられる．SMT プロセッサでは，キャッシュメモリを各実スレッドで共有するため，スレッドの干渉による競合ミスが多発する．

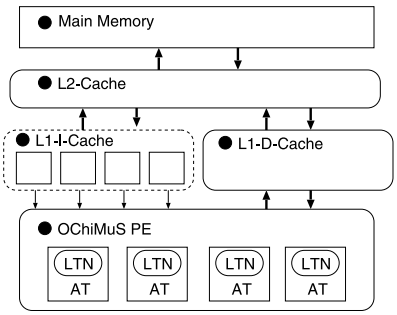


図 1 OChiMuS PE とキャッシュメモリの構成
Fig. 1 The over view of OChiMuS PE and cache memory.

プロセッサの実スレッド数が増加した場合，スレッドの干渉による競合ミスはさらに増加し，性能低下を招く．文献 7) の予備実験において，実スレッド数増加にともなうキャッシュミスの増加，性能低下が指摘されている．

このような問題点が発生する一方で，キャッシュメモリを共有する利点も存在する．ある実スレッドがメモリからキャッシュメモリにデータを読み出し，別の実スレッドがそのデータを利用する場合，後からアクセスした実スレッドはキャッシュミスすることなく，データを利用できる．つまり，プリフェッチと同様の効果があり，ヒット率の向上をもたらす．

以上のような SMT プロセッサのキャッシュメモリにおける問題点と利点に注目し，それらを考慮した有効なリプレース方式を提案する．

3. 目　　標

本論文では，SMT プロセッサのキャッシュメモリにおいて，その特性を活かしたリプレース方式を提案することにより，性能向上をもたらすことが目標となる．SMT プロセッサのキャッシュメモリの問題点と利点をまとめると以下のようになる．

問題点 1 スレッド数増加にともなう容量ミスの増加
問題点 2 スレッド間の干渉にともなう競合ミスの増加
利点 異なるスレッドが同一のキャッシュラインを使用することにともなうキャッシュヒットの増加

問題点 1 に対する有効な手段として，キャッシュ容量の増加がある．キャッシュ容量を増やすことで，容量ミスを緩和できる．また，キャッシュ容量を増やさなくても，文献 7) のようにスケジューラを用い，プロセッサに流入するスレッド数を制限することで容量ミスを緩和できる．問題点 1 をスケジューラを用いず，キャッシュメモリアーキテクチャの視点から考察した

場合，キャッシュ容量の増加しか解決手段がない．

問題点 2 は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ことが原因で発生する．そこで，スレッドごとにリプレース可能な領域を制限する．そうすることで，スレッド間の干渉による相互リプレースを抑え，キャッシュミスを減少させることができる．つまり，問題点 2 はスレッドのリプレースするウェイを制御する新しい方式を提案し，解決する．

SMT プロセッサのキャッシュメモリの利点として，あるスレッドが利用したキャッシュラインを別のスレッドが再利用することで，キャッシュヒットが増加する．複数のスレッドからアクセスされるキャッシュラインは，再びアクセスされる可能性が高い．そこで，複数のスレッドからアクセスされるキャッシュラインをリプレースの対象からはずし，制限する．つまり，リプレース方式を新しく提案し，再度使用されるであろうデータをキャッシュメモリ上に残しておくことで，SMT プロセッサの利点をさらに活かすことができる．

以上のように，SMT プロセッサの問題点 2 と利点について着目する．問題点 2 を解決するリプレース方式と利点をさらに活かすリプレース方式の 2 つの新しい方式を提案，実現し，OChiMuS PE 全体の性能向上を目指す．

これらのリプレース方式は OChiMuS PE と同様に SMT アーキテクチャのプロセッサであれば適用可能である．さらに，各スレッドでキャッシュメモリを共有する場合，CMP（Chip Multi Processor）など他のマルチスレッドアーキテクチャにおいても適用が可能である．

4. 提 案 方 式

本章では，まず，提案するリプレース方式のベースとなる擬似 LRU について説明する．次に，SMT プロセッサのキャッシュメモリの問題点であるスレッド間の競合ミスを抑えるリプレース方式として **LTN（Logical Thread Number）方式**を，利点をさらに活かすリプレース方式として **LSP（Local Shared Priority）方式**を提案し，設計する．

4.1 擬似 LRU

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[6] があるが，性能面において有利である LRU 方式が多く用いられている．しかし，キャッシュメモリのウェイ数が増加すると，LRU を実現することによ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．LRU の擬似的な実現方法には，FRQ（least FReQuently used），NLU（Not Last Used），pseudo-LRU など[6] があるが，本研究では，実現するハードウェア量が少ない pseudo-LRU 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 のハードウェア資源を**図 2** に示す．

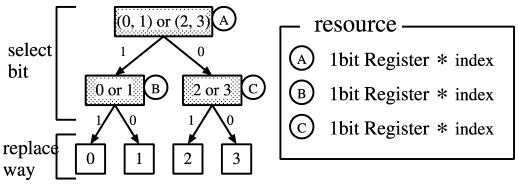


図 2 pseudo-LRU のハードウェア資源
Fig. 2 The hardware resource of pseudo-LRU.

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．

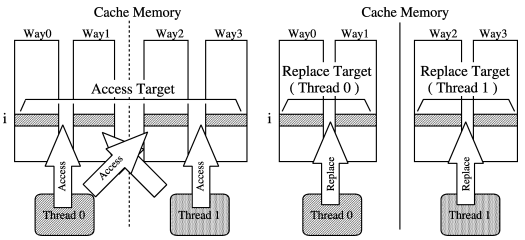
リプレースウェイを決定するとき，まず A のレジスタに 1 が格納されていた場合，B を選択し，0 が格納されていた場合，C を選択する．次に，B が選択された場合，B に 1 が格納されていたらウェイ 0 をリプレース，0 が格納されていたらウェイ 1 をリプレースする．C が選択されたときも同様であり，C に 1 が格納されていたらウェイ 2 をリプレース，0 が格納されていたらウェイ 3 をリプレースする．

このような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．次節以降で新しく提案するリプレース方式は，この pseudo-LRU を変形，拡張させたものとなる．

4.2 LTN 方式

4.2.1 LTN 方式の概要

LTN 方式は，論理スレッド番号（LTN）を用いてスレッド間で発生する競合ミスを抑えるリプレース方式である．LTN 方式の概要を**図 3** に示す．各実スレッドにおけるキャッシュアクセスは図 3 (a) のようになり，従来と変わらずすべてのウェイに対して行えるようにする．しかし，キャッシュミスが発生したときは，図 3 (b) のように，すべてのウェイをリプレースの対象にせず，実スレッドごとにリプレース可能なウェイを制限する．これにより，実スレッド間のリプレース



(a) アクセス時 (b) リプレース時

図 3 LTN 方式の概要
Fig. 3 The view of LTN replacement strategy.

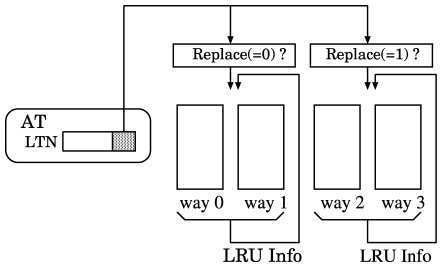


図 4 リプレースの対象となるウェイの選択方法
Fig. 4 The choice for replace way.

における干渉が減少し，相互リプレースを緩和することができる．また，図 3 (a) のようにアクセスの対象となるウェイを制限せず，占有メモリとしていないため，コヒーレンス維持の必要がない．したがってキャッシュメモリを共有するメリットを損なわずにスレッドによる相互リプレースを抑止することが可能となる．

4 ウェイセットアソシアティブのとき，リプレースの対象となるウェイの選択方法を**図 4** に示す．まず，キャッシュメモリにアクセスした実スレッドが保持している LTN より任意のビット（あるいはビット列）を取り出す．次に取り出してきたビットからリプレース可能なウェイを絞り込む．リプレース可能なウェイの中で，さらに候補がある場合は，擬似 LRU にて絞り込んだウェイの中からリプレース対象となるウェイを選択する．

LTN 方式において，LTN から n ビットを用いてリプレースウェイを制限するとき，**LTN-n 方式**と呼ぶ．図 4 のように，LTN から 1 ビットを用いてリプレースウェイを制限するときは LTN-1 方式となる．LTN から 1 ビットを用いる場合，各実スレッドのリプレース可能なウェイは，すべてのウェイの半分となる．

4.2.2 LTN 方式の設計

LTN 方式は，前節で示した擬似 LRU を変形した形となる．4 ウェイセットアソシアティブのとき，LTN-1 方式のハードウェア資源を**図 5** に示す．

図 2 の擬似 LRU と比べると，A の部分が異なる．

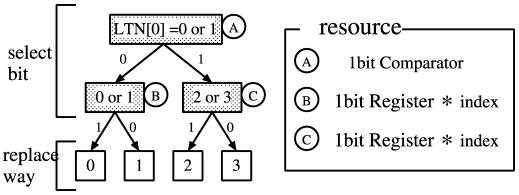


図 5 LTN 方式のハードウェア資源
Fig. 5 The hardware resource of LTN replacement strategy.

LTN 方式の場合，A の箇所が，キャッシュに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．たとえば，LTN-1 方式の場合は 1 bit の 0 or 1 の比較器となる．図 5 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．

4.3 LSP 方式

4.3.1 LSP 方式の概要

SMT プロセッサでは，あるスレッドが利用したキャッシュラインを別のスレッドが再利用することでプリフェッチ効果をもたらし，キャッシュヒット率が向上する．このようにあるインデックスにおいて，複数のスレッドからアクセスされるラインを，本論文では**Local Shared ライン**と呼ぶ．LSP（Local Shared Priority）方式は Local Shared ラインに生存優先度（Priority）を付加し，Local Shared ラインのリプレースを制限し，複数のスレッドからアクセスされるデータをキャッシュメモリ上に残し，スレッド共有型キャッシュメモリの利点をさらに活かすことを目的にしたリプレース方式である．ただし，ここで述べた Local Shared ラインとは，並列プログラミングなどで用いられる共有変数のようなものではなく，単に同一ラインにアクセスしたか否かで決定する．同一キャッシュライン上でまったく関連のないデータに 2 つのスレッドが偶然アクセスした場合も Local Shared ラインに含まれる．

まず，Local Shared ラインの問題点について検討する．Local Shared ラインのリプレースをまったく行わない場合，以下の 2 点の問題が発生する．

問題 1 同一インデックス中のすべてのラインが Local Shared 状態になった場合，そのキャッシュインデックスは完全にリプレース不可能となる．

問題 2 1 度 Local Shared 状態になったキャッシュラインが，以後まったく使用されなくなった場合，キャッシュ容量やウェイが減少したことと等価になり，キャッシュヒット率が低下する．

これらの問題を解決するために，適切なタイミングで Local Shared 状態を無効にする必要がある．まず，キャッシュミス時に同一インデックス中のすべてのラインが Local Shared ラインであるとき，通常の擬似 LRU でリプレースを行う．これにより，問題 1 が解決する．次に，キャッシュタグに数ビットの生存優先度を追加する．あるキャッシュラインにアクセスしたとき，同一インデックス中のヒットした以外のキャッシュラインの生存優先度をデクリメントし，0 になったものは Local Shared 状態を無効にする．これでアクセスされなくなった Local Shared ラインについても検出でき，問題 2 が解決する．

これらを考慮し，LSP 方式を提案する．**図 6** に LSP 方式の概要を示す．まず，キャッシュへアクセスする場合は，図 6 (a) のようになり，次のことが行われる．

(1) ヒットしたラインのタグに含まれる LTN とアクセスした実スレッドの LTN を比較する．

(2) 比較が一致しなかった場合，異なる実スレッドからのアクセスとなり，LS（Local Shared）ビットに 1 をセットし，Local Shared ラインの Priority と実スレッドの LTN を格納する．

(3) 同一インデックス中でヒットした以外のラインについて，Priority の値をデクリメントし，0 かどうか比較する．0 の場合は LS ビットに 0 をセットし，Local Shared を解除する．

キャッシュからデータをリプレースする場合は図 6 (b) のようになり，以下の動作が行われる．

(1) すべてのキャッシュラインで Local Shared になっているかどうかチェックする．

(2) 以下の 2 つを並行して求める．
- 通常の擬似 LRU
- Local Shared ライン以外のラインを対象とした擬似 LRU

(3) すべてのラインが Local Shared だった場合は擬似 LRU を選択し，異なる場合は LSP 方式を選択する．

ここで，生存優先度（Priority）を n とする LSP 方式を **LSP-n 方式**と呼ぶ．たとえば，Local Shared ラインの生存優先度を 4 とすると，LSP-4 方式となる．この場合，同じインデックスのアクセスで，4 回連続ヒットしなかったら，Local Shared 状態を解除する．

4.3.2 LSP 方式の設計

図 7 に LSP 方式で用いるタグを示す．従来のタグのほかに，新たに，Local Shared Bit（LS），論理スレッド番号（LTN），生存優先度（Priority）を付加する．

論理スレッドを 8 個生成する場合，LTN のタグは 3 ビットとなり，また，生存優先度を 8 とすると Priority も 3 bit となる．これらの追加タグは従来のタグと同様にウェイ × インデックス分用意しなければならないので，増加ハードウェア量は大きくなる．そのため，LTN や Priority を少なくする考慮が必要となる．

次に，キャッシュにアクセスしたときのハードウェア資源を**図 8** に示す．キャッシュにアクセスした実スレッドの LTN とタグに保持している LTN の比較を行うため，図の G にあたる不一致比較器が必要となる．不一致回路の結果，Local Shared ラインであることが判別されたら，Priority をセットする．しかし，そのラインがヒットしていない場合，すでにタグの中

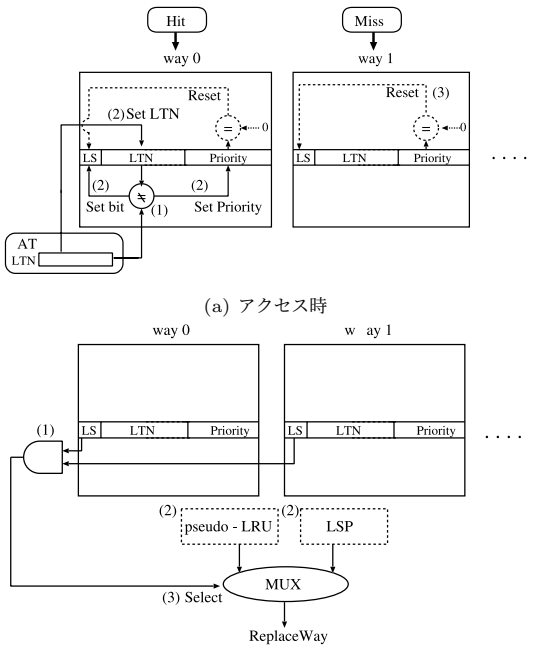


図 6 LSP 方式の概要
Fig. 6 The view of LSP replacement strategy.

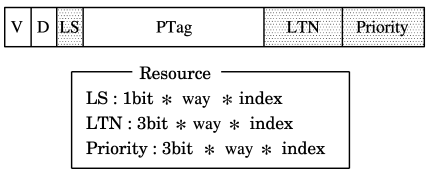


図 7 LSP 方式のタグ
Fig. 7 The tag of LSP replacement strategy.

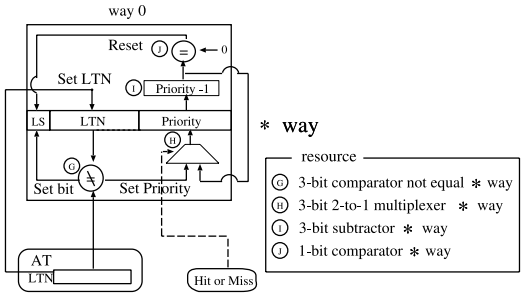


図 8 LSP 方式・アクセス時のハードウェア資源
Fig. 8 The hardware resource of access of LSP replacement strategy.

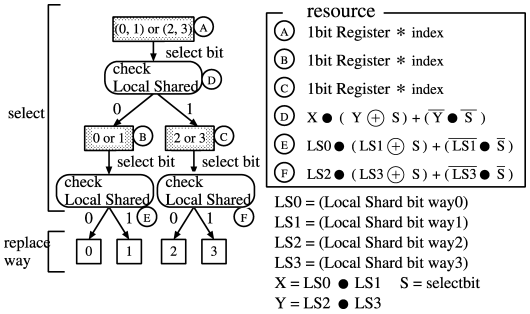


図 9 LSP 方式・リプレース時のハードウェア資源
Fig. 9 The hardware resource of replace of LSP replacement strategy.

に格納されている Priority をデクリメントし，その値を再び Priority にセットしなければならない．よって，図の H のようなキャッシュのヒット/ミスを制御線とする 2 to 1 のマルチプレクサが必要となる．また，Priority を −1 しなければいけないので，I にあたる減算器が必要となる．最後に，Priority が 0 かどうかを判別し，0 ならば LS を 0 にする J の 1 bit の比較器が必要となる．

次に，リプレース時のハードウェア資源を**図 9** に示す．擬似 LRU に D，E，F を追加し，実現する．ここで，D，E，F に追加する回路を考える．

まず，E において，ウェイ 0 の LS を left とし，ウェイ 1 の LS を right とする．F も同様に，ウェイ 2 の LS を left とし，ウェイ 3 の LS を right とする．また，D においてはウェイ 0 とウェイ 1 の LS の論理積を left とし，ウェイ 2 とウェイ 3 の論理積を right とする．このときに，Local Shared ラインを含む枝をリプレースするウェイに選択しないようにすると，**表 1** のような真理値表が得られる．この真理値表をもとに，回路を簡単化していくと式 (1) が得られる．

$$left \bullet (right \oplus select) + (\overline{right} \bullet \overline{select}) \qquad (1)$$

つまり，D，E，F に追加する回路は式 (1) のような

表 1 D，E，F に追加される回路の真理値表
Table 1 The truth table of D, E, F.

Local Shared Bit Left	right	select bit	replace way
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

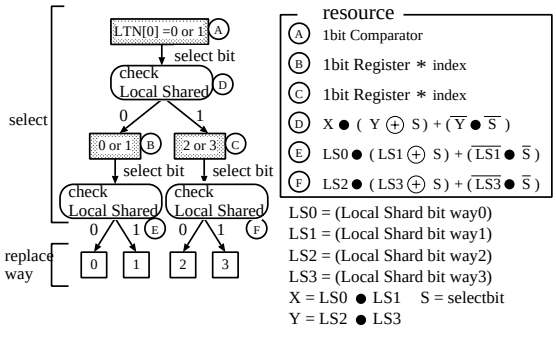


図 10 LTN 方式・LSP 方式組合せのリプレース時におけるハードウェア資源
Fig. 10 The hardware resource of replace of LTN and LSP replacement strategies.

簡単な組合せ回路となる．また，キャッシュアクセス時には図 8 の G，H，I，J の回路がウェイ数分必要になる．しかし，どれも単純な組合せ回路であり，これらの回路は大きなハードウェア増加量にはならない．LSP 方式のハードウェア増加の原因は，図 7 の追加タグ部分になると考える．

4.4 LTN 方式と LSP 方式の組合せ

LTN 方式と LSP 方式を組み合わせたときの実現方法を示す．

キャッシュアクセス時は，LSP 方式を実現するために図 8 のハードウェア資源が必要となる．LTN 方式はキャッシュアクセス時に，追加する回路が必要ないため，LTN 方式と LSP 方式を組み合わせた場合，キャッシュアクセス時の機構は LSP 方式の図 8 と同様になる．

次に，両方式を組み合わせた場合のキャッシュリプレース時のハードウェア資源を**図 10** に示す．図 9 の LSP 方式と比べると A の部分が異なる．LSP 方式では，A の部分が 1 bit× インデックスのレジスタであったが，図 10 では LTN 方式と組み合わせたため，LTN から取り出したビットの比較器となる．まず，A の箇所において LTN 方式を用いてリプレースするウェイを制限し，そして，D，E，F の箇所で LSP 方式によ

る Local Shared 状態の判別を行い，リプレースするウェイを決定する．

両方式を組み合わせた場合，LSP 方式と比べると，A の箇所が 1 bit × インデックスのレジスタから比較器に変わるので，ハードウェア量はその分減少する．

4.5 提案リプレース方式の検討

まず，LTN 方式について検討する．各実スレッドで制御するリプレースウェイの適切な数を求めるため，4 ウェイのキャッシュメモリ上で LTN-1，2 方式を設計し，予備評価を行った．その結果，LTN-2 方式では，実スレッドのリプレースするウェイを制限するあまり，LTN-1 方式と比べて性能低下を及ぼした．そのため，実スレッドで制御するウェイの数は，半分以下程度が妥当と判断した．つまり，4 ウェイでは LTN-1 方式が，8 ウェイでは LTN-1，2 方式が適切であり，性能向上が見込める．

次に LSP 方式について検討する．適切な生存優先度を求めるため，生存優先度として 4〜4096 を格納し，予備評価を行った．その結果，生存優先度を大きな値とすると，Local Shared のロックが長期間解除されず，生存優先度が低い場合と比べて性能低下を及ぼした．また，生存優先度を大きな値とすると，タグのビット数も増加するので，ハードウェア増加量が多くなる．これらの結果から，生存優先度として 4〜16 が適切であり，性能向上が見込めると判断した．

しかしながら，これらの設定はアプリケーションやウェイ数によって変わる可能性がある．今回の設定は一例であり，アプリケーション，ウェイ数に応じて，制限するリプレースウェイ数を決定したり，生存優先度を設定する必要がある．

5. 評　　価

本章では，提案した 2 つのリプレース方式の性能，ハードウェア増加量，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*）[2] を用いた．MUTHASI はプロセッサのパラメータを容易に設定，変更可能である．評価時のプロセッサパラメータを**表 2** に示す．プロセッサパラメータは文献 2)，16) を参考に設定し，SMT プロセッサとして妥当な値とした．実スレッド数（AT 数）は 2，8 であり，各実スレッド数に応じたプロセッサ構成をとる．

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：16,384）を用いた．LU 分解，RADIX ソートは SPLASH-2 [4] より採用した．これらのプログラムは並列マクロによって並列化し，それぞれ 8 個の論理スレッドを生成する．論理スレッド（LT）を多数生成すると，LSP 方式においてタグに付加する LTN のビット長が長くなり，ハードウェア増加量が大きくなってしまう．一方，論理スレッドの生成数を少なくすると，プログラムのスレッドレベル並列性が低くなり，SMT プロセッサの効果が薄くなる．これらを考慮し，生成する論理スレッド数を 8 個とした．また，プログラムの作成は MULiTh [3]，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などは次節に示す．

次に，キャッシュメモリのパラメータを**表 3** に示す．L1-D-キャッシュメモリとして，ウェイ数 4 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．そこで，本評価の前に，擬似 LRU

表 2 シミュレーション時のプロセッサパラメータ
Table 2 The processor configuration of simulation.

PC (AT#)	2	8
Fetch Buffer Size	16	4
Dispatch Queue Size	32	8
Reorder Buffer Size	128	32
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 4cycle) Complex ALU: 1 (Mult 17 delay, Div, 30 delay)	
Branch Unit	1	
Fetch Instructions	8	2
Decode Instructions	8	2
Dispatch Instructions	8	2
Retire Instructions	8	2
Finish Instructions	16	
Speculation Depth	4	

表 3 シミュレーション時のキャッシュメモリパラメータ
Table 3 The cache memory configuration of simulation.

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

☆ 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．

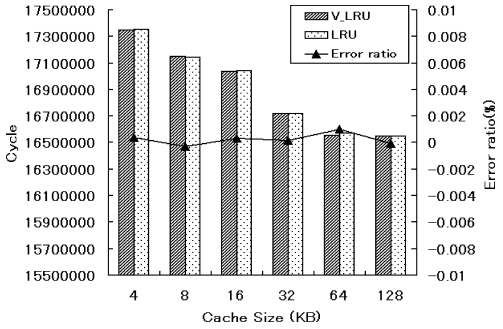


図 11 擬似 LRU と LRU の性能比較
Fig. 11 The Performance comparison between pseudo-LRU and complete LRU.

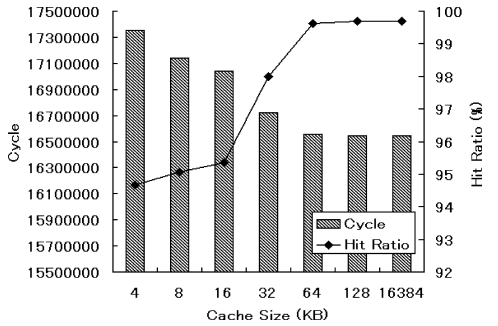


図 12 LU 分解の事前評価
Fig. 12 The prior evaluation of LU.

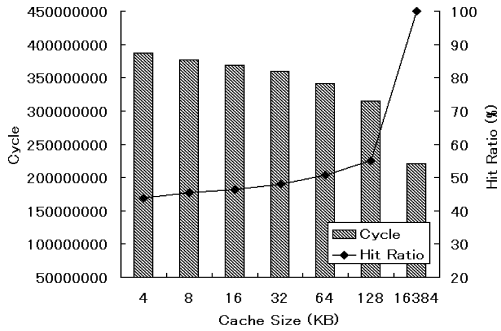


図 13 行列乗算の事前評価
Fig. 13 The prior evaluation of MATRIX.

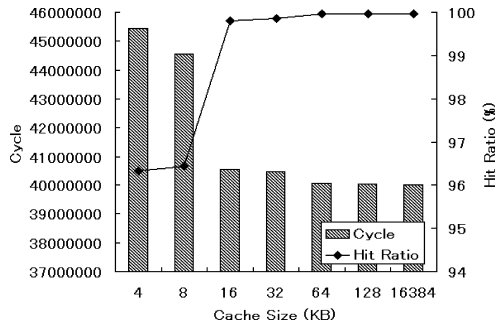


図 14 RADIX ソートの事前評価
Fig. 14 The prior evaluation of RADIX.

と完全な LRU の比較評価を行った．LU 分解の擬似 LRU（V_LRU）と完全な LRU（LRU）の性能比較と誤差率を**図 11** に示す．

比較の結果，擬似 LRU と完全な LRU の性能誤差は ±0.002%未満であり，性能はほぼ同一であることが分かる．他のプログラムの性能比較においても，図 11 の LU 分解と同じく，性能誤差はわずかであった．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[5]，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．

また，実行するプログラム規模に妥当な L1-D-キャッシュ容量を決定するため，本評価の前に，実スレッド数 8，ウェイ数 4，ラインサイズ 32 B，擬似 LRU の L1-D-キャッシュメモリのキャッシュ容量を 4 KB，8 KB，16 KB，32 KB，64 KB，128 KB，16,384 KB と変化させて事前評価を行った．各プログラムの事前評価結果を**図 12**，**図 13**，**図 14** に示す．図 12 の LU 分解において，64 KB 以上はサイクル数，ヒット率が飽和しており，それ以上キャッシュ容量を増やしても性能は変化していない．図 13 の行列乗算は，128 KB においても飽和が発生しておらず，キャッシュ容量を 16,384 KB とヒット率を 100%に近づけるための十分な大きさに設定すると飽和する．図 14 の RADIX ソートは 16 KB で飽和する．

サイクル数，ヒット数が飽和状態のキャッシュ容量では，L1-D-キャッシュメモリのリプレース方式が性能に大きく影響しないため，本評価に適していない．また，評価における妥当なキャッシュ容量として，キャッシュヒット率が 90〜99%になるキャッシュ容量が一般的である[5]．そこで本評価には，キャッシュ容量として，3 つのプログラムの性能が飽和していない 8 KB，LU 分解，行列乗算の性能が飽和していない 32 KB，行列乗算の性能が飽和していない 128 KB が妥当と考え，パラメータとして採用した．行列乗算は，128 KB においてもヒット率が 60%前後であり，一般的なキャッシュ容量のヒット率から外れるが，L1-D-キャッシュメモリのキャッシュ容量として 128 KB より大きい値は，現状の集積技術において現実的ではないと考えられるため，128 KB を選んだ．また，一般的な動向から L1-D-キャッシュ容量をみたとき，8 KB は比較的小さく，32 KB は標準であり，128 KB は大きい[5]．つまり，これら 3 種類のキャッシュ容量は，それぞれプログラム規模に妥当であり，なおかつ一般的なキャッ

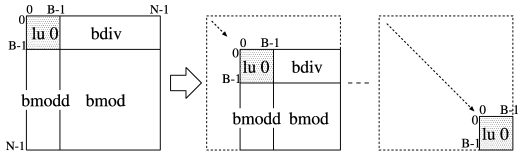


図 15 LU 分解の処理概要
Fig. 15 The over view of LU processing.

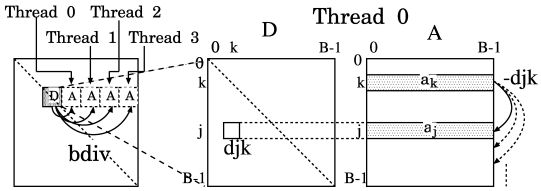


図 16 bdiv の処理概要
Fig. 16 The over view of bdiv processing.

シュ動向からも幅広く提案方式を活用できるかどうかを評価できる．

他のキャッシュメモリのパラメータについては，文献 5)，16) 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．L1-I-キャッシュメモリは表 3 のパラメータのキャッシュメモリが，実スレッド分生成される．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），LSP-8 方式（LSP），LTN-1+LSP-8 方式（LTN+LSP）を実装し，プログラムを実行した．

本評価では，論理スレッド数を 8 としたので，LTN-1 方式において，1 ビットの取り出し方に自由度がある．ここでは，LTN-1 方式として，LTN の最下位 1 ビットを取り出し，評価を行った．

5.2 プログラムのスレッド分割方法

ここでは性能評価に用いる 3 つのプログラムのスレッド分割方法，データ共有具合，メモリアクセスパターンを示す．

5.2.1 LU 分 解

SPLASH2 の LU 分解は，行列を $B \times B$（**図 15** の斜線部分）の小さなブロックに分割し，そのブロックを単位として計算を行う．主な処理は図 15 のように，lu0，bdiv，bmodd，bmod の 4 つに分類される．紙面の都合上，ここでは bdiv のみ扱う．

bdiv は，ある行（k 行）を用いて，それより下側の行（j 行）に対して，**図 16** のように $a_j - d_{j,k}a_k$ のベクトル減算を行う．ただし，処理対象となるブロックを行列 A，対角のブロックを行列 D とする．マルチスレッド化を考えると，bdiv は行列 A ごとにスレッドを割り当て，並列化する．このとき，各スレッドが対角行列 D を参照するため，対角の行列 D はスレッド

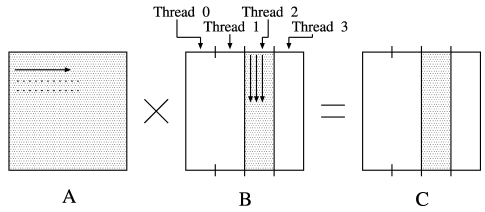


図 17 行列乗算の処理概要
Fig. 17 The over view of MATRIX.

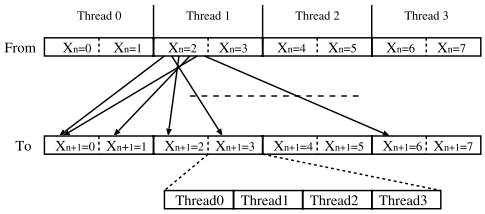


図 18 RADIX ソートの処理概要
Fig. 18 The over view of RADIX.

で共有になる．しかし，行列 D は $B \times B$ の小さなブロックであるため，次項で示す行列乗算などに比べると，各スレッドで共有するデータは少なくなる．また，各スレッドのベクトル減算は行方向のアクセスとなるため，空間的局所性は高くなる．bdiv はバリア同期により，各スレッドの作業を待つ．全スレッドの作業が終了したら，bmodd などの次の処理を行う．

5.2.2 行 列 乗 算

行列乗算は，$N \times N$ の正方行列について，行列乗算の定義式に従い，計算を行う．

スレッドの並列化を考えると，本研究の行列乗算では，行列 B を列方向に M 等分分割し，スレッドを割り当てる．ただし，M は並列化するスレッドの数である．メモリアクセスパターンは**図 17** のようになり，行列 A は行方向，行列 B は列方向にアクセスする．行方向が線形アクセスであるため，空間的局所性は高い．また，マルチスレッド化することにより，複数のスレッドが行列 A を利用するため，スレッドのデータ共有が多く，SMT プロセッサの利点を活用できる．

5.2.3 RADIX ソート

RADIX ソートの主な処理は，ソートとそれを行うときに発生するデータアクセスである．

図 18 の配列 From は，基数 8 において n 桁目までソートしたデータ配列である．ここで，配列 From に格納されているある 1 つのデータ「X」は $X = X_{n+1}X_nX_{n-1}\ldots X_0$ のように添字の値が大きいほど上位桁となる（たとえば，「78564」という値の場合，$X_0 = 4$，$X_1 = 6$，$X_2 = 5\cdots$ となる）．

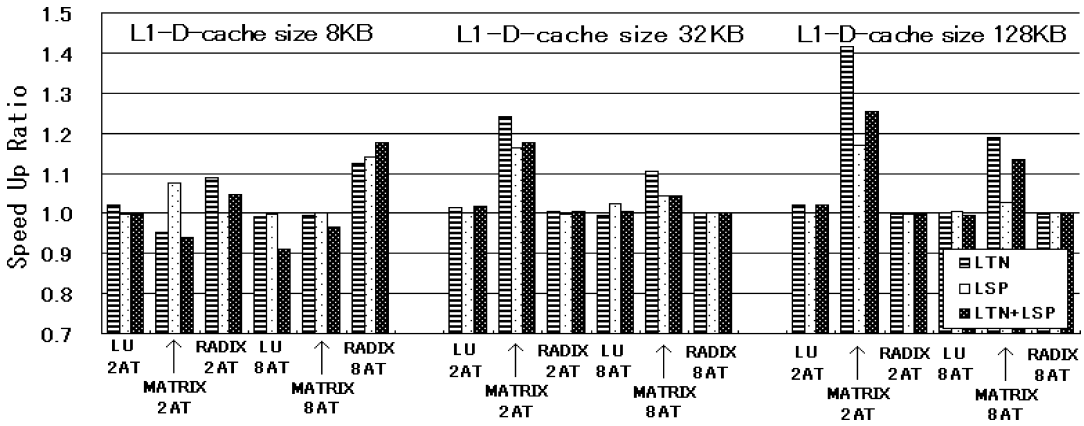


図 19 各リプレース方式による性能向上率
Fig. 19 The speed up ratio by replacement strategies.

図 18 は配列 From の中のデータを n+1 桁目においてソートし，配列 To へ格納するメモリアクセスパターンである．スレッドは配列 From の基数ごとに区切り，割り当てる．また，配列 To は n+1 桁目の値ごとに区切られており，この区間内においてあらかじめ各スレッドごとにアクセスできる範囲を決めている．

このように RADIX ソートは，配列 From の各区間において線形アクセスが発生している．しかし，配列 To において，データが複数の区間に分散するため，ワーキングセットが広がりやすい．そのため，局所性はわずかであり，基数を大きくしていくと区間数が増加し，ランダムアクセスに近い傾向をみせる．また，ソートされるデータは，ソートされる桁の値によってランダムに各スレッドに割り当てられる．つまり，桁によって，ソートされるデータを扱うスレッドが変わる可能性が高い．このように，RADIX ソートはランダムであるが各スレッド間で共有するデータは多い．

5.3 実行結果

擬似 LRU と提案したリプレース方式の性能を比較した．**図 19** は擬似 LRU のサイクル数を基準とし，提案したリプレース方式の性能向上率を示す．

LU 分解は，キャッシュ容量，実スレッド数にかかわらず，どの方式からも高い性能向上率が得られなかった．性能向上率として，8 KB，32 KB，128 KB の実スレッド数 2 で 1.02 倍程度を示した．

行列乗算は 8 KB において，LSP 方式を有効に活用しているが，LTN 方式，LTN+LSP 方式の性能は向上しなかった．32 KB では，2AT で 1.16 倍〜1.24 倍，8AT で 1.044 倍〜1.105 倍の性能向上率を示した．128 KB において，LTN 方式による性能向上率が非常に高く，最大で，実スレッド数 2 のとき，1.420 倍を示した．

RADIX ソートは 8 KB において，各方式を有効に活用し，実スレッド数 8，LTN+LSP 方式で 1.175 倍の性能向上を示した．逆に 32 KB，128 KB では，性能向上も低下も示さなかった．

5.4 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 4** に示す．全体的に，性能向上はヒット率の上昇によってもたらされ，逆に性能低下はヒット率の低下によって引き起こされている．

まず，LU 分解について考察する．LU 分解はキャッシュ容量に関係なく，提案した各方式の性能向上率が低い．その原因として，LU 分解は 5.2.1 項のようにスレッド間のデータ共有が他のプログラムと比べ少なく，また，キャッシュミスの要因もスレッドの競合ミスではなく，容量ミスであることがあげられる．たとえば，マルチスレッド化した LU 分解において，文献 7) の容量ミスを緩和する手法は高い性能向上率を示しているが，スレッドの共有度を求めて競合ミスを緩和する手法は効果がない．そのため，スレッド間の競合ミスの緩和やスレッドが共有するデータを活用することを目的とした各方式は有効に働かず，高い性能向上には至らなかった．また，8 KB では性能向上とともに性能低下を引き起こしていたが，32 KB，128 KB では性能向上のみを示した．これは，キャッシュ容量を大きくしたことで容量ミスが緩和され，競合ミスを改善する各リプレース方式が若干ではあるが有効に作用したことを示している．

行列乗算について考察する．行列乗算は 5.2.2 項に示したとおり，各スレッドで共有するデータが多い．そのため，全体的に LSP 方式はヒット率向上の効果があり，性能も向上した．しかし，8 KB の LTN 方式，

表 4 すべての実行結果
Table 4 The result of all experiments.

		LU			MATRIX			RADIX		
		cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R	cycle	SUR	L1D_H_R
L1-D cache size 8 KB	2AT LRU	20011179	1.000	94.93%	304959599	1.000	58.62%	39206063	1.000	99.93%
	2AT LTN	19579758	1.022	94.94%	319834358	0.953	55.61%	36021814	1.088	99.97%
	2AT LSP	20020463	1.000	94.92%	283632569	1.075	62.86%	39215516	1.000	99.89%
	2AT LTNLSP	20193543	0.991	93.65%	324265675	0.940	54.56%	37453340	1.047	99.95%
	8AT LRU	17141217	1.000	95.06%	376186360	1.000	45.38%	44554932	1.000	96.44%
	8AT LTN	17262666	0.993	94.69%	378560690	0.994	45.03%	39655811	1.124	97.01%
	8AT LSP	17134884	1.000	94.93%	376304669	1.000	45.36%	39031542	1.142	96.60%
	8AT LTNLSP	18798343	0.912	91.04%	390082870	0.964	43.33%	37905786	1.175	97.50%
L1-D cache size32 KB	2AT LRU	19248962	1.000	98.19%	346332673	1.000	55.32%	37256023	1.000	99.97%
	2AT LTN	19000196	1.013	98.21%	279284903	1.240	64.41%	37127307	1.003	99.96%
	2AT LSP	19206246	1.002	98.19%	297785289	1.163	61.58%	37229852	1.001	99.96%
	2AT LTNLSP	18938278	1.016	98.24%	294237701	1.177	61.96%	37053105	1.005	99.96%
	8AT LRU	16718506	1.000	98.00%	359259753	1.000	47.88%	40795238	1.000	99.96%
	8AT LTN	16713745	1.000	97.93%	325100191	1.105	53.37%	40641101	1.004	99.96%
	8AT LSP	16307401	1.025	98.56%	344344549	1.043	51.15%	40701312	1.002	99.97%
	8AT LTNLSP	16663763	1.003	98.10%	343984083	1.044	51.02%	40774106	1.001	99.96%
L1-D cache size128KB	2AT LRU	18718283	1.000	99.68%	344176602	1.000	58.79%	37583351	1.000	99.97%
	2AT LTN	18338424	1.021	99.74%	242458443	1.420	75.65%	37659372	0.998	99.97%
	2AT LSP	18717661	1.000	99.68%	294548861	1.168	64.82%	37763202	0.995	99.97%
	2AT LTNLSP	18340853	1.021	99.79%	274815395	1.252	66.36%	37762733	0.995	99.97%
	8AT LRU	16545703	1.000	99.68%	314912657	1.000	55.04%	40021006	1.000	99.97%
	8AT LTN	16498829	1.003	99.69%	264683661	1.190	63.14%	40054154	0.999	99.97%
	8AT LSP	16475736	1.004	99.68%	306281689	1.028	56.10%	40029149	1.000	99.97%
	8AT LTNLSP	16407969	1.008	99.42%	277636682	1.134	60.70%	40027090	1.000	99.97%

*SUR: Speed up ratio **L1D_H_R:L1 D-cache hit ratio

LTN+LSP 方式において性能低下が発生している．その原因として，8 KB ではキャッシュ容量が少ないため，キャッシュミスの要因として容量ミスが多いことがあげられる．LTN 方式はスレッドによる競合ミスを緩和する方式なので，容量ミスには対処できない．逆にリプレースするウェイを制限してしまうことによって，性能低下を引き起こしている．キャッシュミスにおける容量ミスの割合はキャッシュ容量を増やすと緩和され，競合ミスの割合が多くなる．そのため，競合ミスを緩和する LTN 方式において，32 KB，128 KB とキャッシュ容量を増加させると，性能向上率も上昇する．

RADIX ソートについて考察する．RADIX ソートも 5.2.3 項のようにスレッドで共有するデータが多い．そのため，8 KB で，スレッド間の共有するデータをキャッシュに残す LSP 方式が有効に動作し，LTN 方式と組み合わせた LTN+LSP 方式が高い性能向上を示した．逆に 32 KB，128 KB において，各方式は性能向上も低下も示さなかった．それは事前評価からも分かるように，RADIX ソートは 16 KB の時点で飽和しており，32 KB，128 KB とキャッシュ容量を大きくすることで，どのリプレース方式を採用してもヒット率が 100%に近い値になり，性能にキャッシュメモリが影響しないことが原因である．

提案したリプレース方式は，SMT プロセッサのキャッシュ容量，実スレッド数の大小にかかわらず，従来以上の性能向上を示した．各方式は，行列乗算，RADIX ソートのように，スレッド間でデータを多く共有するプログラムの場合，高い性能向上が見込める．

128 KB の LU 分解，32 KB，128 KB の RADIX ソートの結果から分かるように，提案したリプレース方式は，サイクル数・ヒット率が飽和状態にあるキャッシュ容量のプログラム実行において，若干の性能改善を示しているが，性能改悪には至っていない．また，RADIX ソートでは 8 KB，LU 分解では 32 KB，行列乗算では 32 KB，128 KB で性能向上が顕著に現れている．これらのキャッシュ容量は各プログラムの性能が飽和する前の妥当な値であり，一般的なプロセッサ設計に用いられるキャッシュ容量である．つまり，提案したリプレース方式は，プログラム規模に妥当なキャッシュ容量において性能向上を示し，さらに飽和が発生しているキャッシュ容量においても，性能が向上することは多々あるが低下することは少ない．

5.5 ハードウェア増加量と動作周波数

提案した方式の具体的なハードウェア増加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），LSP-8 方式（LSP），LTN-1+LSP-8 方式（LTN+LSP）を実装した．論理合成結果を**表 5**

表 5 各リプレース方式のハードウェア量
Table 5 The hardware cost of replacement strategies.

スライス数	擬似 LRU	LTN	LSP	LTN+LSP
SMT Processor (P)	8366			
Cache Memory (C)	2937	2853	3565	3489
Block RAM number	32	32	32	32
P+C	11303	11219	11931	11855
ハードウェア増加率	0%	−0.74%	5.56%	4.88%

表 6 各リプレース方式の動作周波数
Table 6 The frequency of replacement strategies.

	擬似 LRU	LTN	LSP	LTN+LSP
最長パス（ns）	19.765	19.467	20.164	20.061
動作周波数（MHz）	60.996	62.145	60.785	60.976
動作周波数低下率	0%	−1.849%	0.347%	0.001%

に示す．

プロセッサのハードウェア量は現在著者らが設計している FPGA 向け SMT プロセッサ[9] を参考にした．また，どの方式もタグ領域として，スライスを用いて実現する分散 RAM を使用したため，ハードウェアスライス数が多くなっている．キャッシュのデータ部分は，各方式とも 32 個の Block RAM を用いた．

LTN 方式は擬似 LRU と比べ，ハードウェア量が 0.74%減少した．LSP 方式はタグ領域が主な原因でハードウェア量が増加している．しかし，プロセッサを含めたチップ全体で考えると増加率は 5.56%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．LTN+LSP 方式は両方式を同時に実現しているので，LSP 方式よりハードウェア量は少なくなる．

次に提案したリプレース方式の動作周波数を**表 6** 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悪影響を及ぼす．しかし，LTN 方式は擬似 LRU と比べ，動作周波数が 1.849%向上した．また，LSP 方式の動作周波数低下率は 0.347%，LTN+LSP 方式の動作周波数低下率は 0.001%となった．LSP 方式，LTN+LSP 方式において若干の動作周波数の低下が生じているが，低下率は 0.5%未満である．つまり，提案した各リプレース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．

性能評価とハードウェア増加量の結果をみると，従来の擬似 LRU と比較し，LTN 方式は 0.74%のハードウェア量の減少かつ最大 1.420 倍の性能向上を示した．LSP 方式は 5.56%のハードウェア増加量に対し，最大 1.168 倍，LTN+LSP 方式は 4.88%のハードウェア増加量に対し，最大 1.252 倍の性能向上を示した．

6. 関 連 研 究

関連研究として，山崎らによる動的スレッドアソシアティブ方式[10),11)] がある．この方式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，LTN 方式と思想が似ている．また，行列乗算のようなデータ共有が多いプログラム実行において，5〜8%の性能向上率を示している[11)]．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，周波数低下率の検討がない．本研究の LTN 方式は，擬似 LRU を変形することで実現し，実装することで具体的なハードウェア増加量，周波数低下率を示した．さらに，LTN 方式は性能向上率が行列乗算で最大 42%を示すなど，動的スレッドアソシアティブ方式よりも高い性能向上率を示している．

Sigmund らはスレッドに優先度をつけて，優先度が高いスレッドのリプレースを制限する Priority 方式[15)] を提案している．この方式と本研究の LSP 方式を比べると，LSP 方式はスレッドのキャッシュライン共有を判定し，スレッド自体に優先度を付加するのではなく，キャッシュラインに生存優先度を付加している点が新しい．

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を可変し制御する Partitioning Decision 方式と Modified LRU[12),13)] を提案している．これは，各スレッドのメモリアクセスの局所性を活用し，キャッシュ容量の大きい L2-キャッシュのあるインデックスにおいて，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て，性能向上を目指している．この方式は，各スレッドでアクセスするインデックスが異なる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インデックスにアクセスするため，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%の性能向上しかない[12)]．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ないと考える．

Tanaka はキャッシュメモリの各ウェイをスレッドに割り当てて占有キャッシュメモリとしている[14)]．占有キャッシュメモリは共有キャッシュメモリと比較し，コヒーレンス維持のコストが発生するため，その分性能が低下する．また，スレッドでデータを共有するプログラムの場合，プリフェッチ効果の利点を活かせなくなってしまう．つまり，本論文のようなマルチスレッドプログラムを実行する場合，L1-D-キャッシュメモ

リとして，占有キャッシュメモリを採用するよりも共有キャッシュメモリを採用する方が効果的である．

キャッシュメモリのヒット率を改善する手法としてスケジューラを活用する方法がある．本研究はリプレース方式について着目したが，文献 7)，8) はシステムソフトレベルのスケジューラについて着目している．プロセッサに流入するスレッド数を制限するスケジューラ[7] やスレッドの相性や親和性を監視するスケジューラ[7),8)] について提案し，ヒット率を改善している．

7. おわりに

本論文では，SMT プロセッサの性能低下の原因として，キャッシュミスを取り上げた．SMT プロセッサにおけるキャッシュメモリの問題点と利点に着目し，それぞれに対応する LTN 方式，LSP 方式，およびそれらを組み合わせた LTN+LSP 方式を提案し，実現した．評価の結果，従来の擬似 LRU と比較し，各方式はキャッシュヒット率を改善させ，性能向上を示した．具体的には，行列乗算において LTN 方式が最大 1.42 倍の性能向上を示した．また，各方式を実装しハードウェアコストを見積もり，わずかなハードウェア増加量で各方式を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 アーキテクチャにおける各方式の適用，評価がある．また，本論文で提案したリプレース方式と文献 7)，8) などのスケジューラを組み合わせた場合の評価，検討を行いたい．たとえば，スレッドの相性を監視するスケジューラ[7] を適用し，キャッシュライン共有度の高いスレッドが並列実行された場合，スレッド間で共有するデータをキャッシュメモリ上に残す LSP 方式はさらに有効に動作すると考えられ，さらなる性能向上が期待できる．

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*, pp.392–403 (1995).
2) 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).
3) 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).
4) 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*, pp.24–36 (1995).
5) Hennessy, J.L. and Patterson, D.A.: *Computer Architecture A Quantitative Approach*, 3rd Edition, Morgan Kaufmann Publishers (2002).
6) Handy, J.: *The Cache Memory book*, 2nd Edition, Academic Press (1998).
7) 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).
8) 小川周吾，平木 敬：プロセスの実行時情報を用いたスケジューラによる高速化手法，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.161–169 (2005).
9) 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，中條拓伯，並木美太郎：SMT プロセッサの FPGA への実装と評価，*SACSIS 2005*, Vol.2005, No.5, pp.239–240 (2005).
10) 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用データキャッシュ—動的スレッドアソシアティブ方式—の評価，情報処理学会研究報告（1999-ARC-132），Vol.1999, No.21, pp.97–102 (1999).
11) 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).
12) Suh, G.E., Rudolph, L. and Devadas, S.: Dynamic Cache Partitioning for Simultaneous Multithreading Systems, *Proc. IASTED International Conference on Parallel and Distributed Computing and Systems* (*PDCS'01*), pp.635–641 (2001).
13) Suh, G.E., Rudolph, L. and Devadas, S.: Dynamic Partitioning of Shared Cache Memory, *The Journal of Supercomputing Architecture*, pp.7–26 (2004).
14) Tanaka, K.: Fast Context Switching by Hierarchical Task Allocation and Reconfigurable Cache, *Innovative Architecture for Future Generation High-Performance Processors and Systems 2004*, pp.20–29 (2003).
15) 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, pp.1–9 (2000).
16) 笹田耕一，佐藤未来子，内倉　要，小笠原嘉泰，中條拓伯，並木美太郎：SMT プロセッサ向けの軽量な同期機構，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.SIG16 (ACS12), pp.14–27 (2005).

(平成 18 年 1 月 27 日受付)
(平成 18 年 5 月 20 日採録)

小笠原嘉泰（学生会員）
1982 年生まれ．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 年同大学大学院工学教育部博士前期課程情報コミュニケーション工学専攻修了．現在，同大学院工学府博士後期課程電子情報工学専攻に在籍．マルチスレッドプロセッサ，キャッシュメモリ，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博士後期課程修了．博士（工学）．現在，東京農工大学大学院特別研究生．オンチップ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．

笹田　耕一（正会員）
1979 年生まれ．2004 年東京農工大学大学院工学研究科博士前期課程情報コミュニケーション工学専攻修了．2006 年同大学院工学教育部博士後期課程電子情報工学専攻退学．同年東京大学大学院情報理工学系研究科特任助手．オペレーティングシステムやシステムソフトウェア，並列処理システム，言語処理系，プログラミング言語に関する研究に興味を持つ．

内倉　要
1981 年生まれ．2004 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 年同大学大学院工学教育部博士前期課程情報コミュニケーション工学専攻修了．現在，(株) 小松製作所に勤務．オペレーティングシステム，システムソフトウェアに興味を持つ．

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．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部助教授．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）．