情報処理学会論文誌　コンピューティングシステム　Vol. 2　No. 2　63–74 (July 2009)

# マルチコアプロセッサ上での粗粒度タスク並列処理のためのコンパイラによるローカルメモリ管理手法

中野 啓史[†1]　桃園 拓[†1,*1]　間瀬 正啓[†1]
木村 啓二[†1]　笠原 博徳[†1]

リアルタイム性および高性能，低電力が要求される情報家電機器では，オフチップ共有メモリに加え，小容量高速なローカルメモリを搭載したマルチコアプロセッサが開発されている．しかしながら，プログラマが手動でローカルメモリ容量を考慮しつつローカリティの最適化を行うことはきわめて困難であり，プログラム開発期間の短縮のためにはコンパイラによる自動最適化が必要となる．そこで，本論文では，容量制約のあるローカルメモリを有効に利用するための並列化コンパイル手法を提案する．提案手法ではまず，粗粒度タスク並列処理によりループやサブルーチン間の並列性を抽出する．続いてループ整合分割により，ローカルメモリサイズを考慮した粗粒度タスク分割を行う．従来のデータローカライゼーション手法は，分割されたデータを固定的にローカルメモリに割り当てていた．提案手法では，タスク分割後，データの定義あるいは参照時刻に基づくローカルメモリの割当てと解放を行い，より柔軟なローカルメモリ管理を実現する．オーディオ圧縮に用いられる AAC エンコーダを用いた性能評価の結果，固定的な割当てを行う従来のデータローカライゼーション手法と比較し，SH4A を 4 コア集積した RP1 マルチコア上で，約 2.6 倍，8 コア集積した RP2 マルチコア上で，約 2.5 倍の速度向上がそれぞれ得られた．

# Local Memory Management Scheme by a Compiler on a Multicore Processor for Coarse Grain Task Parallel Processing

Hirofumi Nakano,[†1] Taku Momozono,[†1,*1]
Masayoshi Mase,[†1] Keiji Kimura[†1]
and Hironori Kasahara[†1]

Multicore processors integrating a small fast local memory for each core in addition to an off-chip shared memory has been developed for consumer electronics to meet real-time constraints, high performance and low power demand. However, data locality optimization by hand considering local memory size is much difficult. Therefore automatic compilation optimization is necessary to speed up application development time. This paper proposes a parallelizing compilation scheme which realizes effective use of limited local memory. First, the proposed scheme extracts parallelism among loops or subroutines using coarse grain task parallel processing. Subsequently, a loop is decomposed into smaller loops to fit local memory size using loop aligned decomposition. A conventional data localization scheme allocates decomposed data to fixed local memory address. On the other hand, the proposed scheme effectively allocates and deallocates decomposed data based on data definition and reference time. As the results, the proposed scheme gives us about 2.6 times speedup for AAC encoding program against the conventional scheme which does not manage each array on RP1 4 SH4A multicore processor and about 2.5 on RP2 8 SH4A multicore processor, respectively.

## 1. はじめに

リアルタイム性の要求される組み込みシステムにおいては，キャッシュミスヒット時の実行時不確定性を避けつつ，メモリウォール問題を打開する目的で，ソニー，東芝，IBM の Cell[1)]，ルネサス，日立，早稲田の RP1[2)]，RP2[3)] のように，キャッシュとは別に，ローカルメモリを搭載したマルチコアが開発されている．今後，情報家電，自動車のような組み

†1 早稲田大学
Waseda University
*1 現在，ソニー株式会社
Presently with Sony Corporation

込み用途として，要求されるリアルタイム処理性能向上と消費電力低減を両立させるために，ローカルメモリを搭載した組み込み用マルチコアが採用されると予測される．

しかしながら，プログラマが手動で容量制約のあるローカルメモリを活用する並列化プログラムを開発することは非常に困難である．そこで，コンパイラによる各プロセッサへの負荷分散とローカルメモリへのデータ割当てを自動的に行うローカルメモリ最適化コンパイル手法が望まれる．

ループ並列性を利用し，各並列化ループが終了するたびに，オフチップメモリにデータを書き戻すというコンパイラによるローカルメモリ管理手法[4),5)]が提案されている．しかしながら，マルチコアに搭載されるコア数の増大にともない，ループ並列性のみの利用では，予想される理論性能と得られる実効性能の差が拡大してしまう．そのため，ループ並列性に加え，ループやサブルーチン間といったより粒度の大きな並列性を利用する粗粒度タスク並列処理が重要となる．また，メモリウォール問題の深刻化にともない，いったんローカルメモリに配置したデータは，ループごとにオフチップメモリに書き戻すのではなく，なるべく長時間ローカルメモリに保持することが望ましい．そのため，粗粒度タスク間にわたるデータローカリティ利用が必須となる．

ループやサブルーチンといった粗粒度タスク間の並列性を利用し，粗粒度タスク間でデータローカリティを活用する手法として，ループ整合分割を用いたデータローカライゼーション手法[6),7)]が提案されている．データローカライゼーション手法では，データ依存したループ群を分割し，分割された部分ループでアクセスされる部分配列をローカルメモリ上の領域に固定的に割り当てる．限られたローカルメモリをより効率的に利用するためには，部分配列をローカルメモリ上へ固定的に割り当てるのではなく，粗粒度タスク間にわたるデータの使用状況とローカルメモリの使用状況に応じた柔軟な割当てを行うローカルメモリ管理が重要となる．

そこで，本論文では，従来のデータローカライゼーション手法において，ローカルメモリに固定的に割り当てられていた部分配列を，その定義・参照タイミングに応じ，割当ておよび解放する手法を提案する．

提案手法ではまず，従来と同様に，データ依存したループ群はループ整合分割によって分割され，分割された部分ループを含む粗粒度タスクは各プロセッサにスケジューリングされる．続いて，提案手法では粗粒度タスクのプロセッサに対するスケジューリング結果，各粗粒度タスクがアクセスする配列アクセスの状況をもとにして，ループ整合分割後の部分配列のローカルメモリ割当てを決定し，データ転送を最小化することにより，ローカルメモリ上の時間的データローカリティを高める．

本論文の構成を以下に示す．2 章で関連研究について述べ，課題を明確化する．3 章では提案手法の前提となる粗粒度タスク並列処理とループ整合分割について述べた後，ローカルメモリ上の時間的ローカリティを高めるためのローカルメモリ管理について述べる．4 章では従来のループ整合分割と提案手法の性能評価結果について述べる．5 章で本論文をまとめる．

## 2. 関 連 研 究

プログラム開始時にローカルメモリに割り当てたデータやプログラムコードを実行終了まで同じアドレス上に保持する静的なローカルメモリ管理[8)–10)]では，ローカルメモリサイズを超えるようなデータを，プログラムの実行に従って効率良く割り当てることができない．そのため，提案手法では，プログラムの実行に応じ，ローカルメモリ上に配置するデータを変更する動的なローカルメモリ管理を行う．

コンパイラによる動的なローカルメモリ管理手法としては，ループやサブルーチンといった粗粒度タスクにまたがり，データやプログラムコードをローカルメモリに割り当てる手法[11)–14)]が提案されているが，いずれも単一のプロセッサを対象としており，マルチコアプロセッサに適用することができない．

マルチプロセッサを対象としたコンパイラによるローカルメモリ管理手法として，以下のような研究が行われている．Kandemir ら[4)]は，単一ループにおけるループ並列処理において，プロセッサ間で共有されるデータを，他のプロセッサがオフチップメモリからロードせずに済むように，ループイタレーションの実行順序を変更し，オフチップメモリへのアクセス回数を削減する手法を提案している．また，Issenin ら[5)]は，ループレベル並列性を利用し，プロセッサ間で共有されるデータを共有ローカルメモリに配置する手法を提案している．

上記の 2 手法はループ並列性のみを利用し，ループ終了時にオフチップメモリにデータを書き戻す．しかしながら，今後予想されるコア数の増大に対し，スケーラブルな性能向上を得るには，ループ並列性に加え，より粒度の大きなループやサブルーチンといった粗粒度タスク間の並列性の利用および粗粒度タスク間にわたるデータローカリティの利用が必須となる．

Knight ら[15)]の階層化された粗粒度演算におけるローカルメモリ管理手法や Ruggiero ら[16)]のパイプラインタスクグラフにおけるローカルメモリ管理手法は，いずれもユーザがタスクを適切なサイズに分割することを仮定している．しかしながら，ローカルメモリサイ

ズを考慮しながら，プログラム全域にわたり適切なサイズにタスクを分割することは，ユーザにとって困難であり自動化が望まれる．

吉田ら[6),7)] のループ整合分割を用いたデータローカライゼーション手法は，同一の配列を定義・参照するループ群のインデクス範囲を分割し，部分ループを生成する．同一のインデクス範囲を持つすべての部分ループにわたり，定義・参照される部分配列をローカルメモリ上の領域に固定的に割り当てることで，部分ループ間でのデータローカリティの抽出を実現する．限られたローカルメモリをより効率的に利用するためには，固定的な割当てではなく，プログラムの実行に応じて，ローカルメモリとオフチップメモリとの間で必要最小限のデータ転送を行いながら管理する必要がある．

## 3. ローカルメモリ管理

ローカルメモリを持つマルチコア上でコア数に応じた性能向上を得るには，ループレベル並列性の利用に加え，サブルーチンやループ間の粗粒度タスク並列性を利用することと，粗粒度タスク間でアクセスパタンに応じたデータの割当てと解放を行い，いったんローカルメモリに割り当てたデータをなるべく長期間保持することが必要となる．上記を実現するため，提案手法では粗粒度タスク並列処理の利用を前提として，定義および参照時刻に応じた配列のローカルメモリ割当てと解放を行う．

提案手法の流れは以下のとおりである．まず，粗粒度タスク並列処理[17),18)] により，ループやサブルーチンといった粗粒度タスク間の並列性を抽出する．次に，ループ整合分割[6),7)] により，粗粒度タスク間の並列性とデータローカリティを考慮した粗粒度タスク分割を行う．入力プログラムに対し，これらの従来手法を適用した後，前述したデータの定義および参照情報を考慮したローカルメモリ割当てと解放を行う．この割当てと解放の単位として，ローカルメモリを仮想的に区切ったブロックを使用する．ブロックのサイズは入力プログラムとループ整合分割の状況によって決定される．この割当てと解放により，一時的にアクセスされる部分配列を，その生存区間でのみローカルメモリに配置するなど，より効率的にローカルメモリを利用することができる．

本章ではまず，粗粒度タスク並列処理とループ整合分割の概要を 3.1 節と 3.2 節でそれぞれ述べる．

### 3.1 粗粒度タスク並列処理

本節では，提案手法の前提となる粗粒度タスク並列処理について述べる．粗粒度タスク並列処理とは，マルチグレイン並列処理における最も粒度の粗いタスク並列処理手法であり，対象プログラムを階層的に粗粒度タスク（マクロタスク）に分割し，並列処理を行う[17),18)]．具体的には，対象プログラム全体（第 0 階層とする）を，疑似代入文ブロック（BPA），繰返しブロック（RB），サブルーチンブロック（SB）の 3 種類の第 1 階層マクロタスク（MT）に分割する．第 1 階層マクロタスクのうち，繰返しブロックおよびサブルーチンブロックの内部の第 1 階層において，第 2 階層マクロタスクを定義する．以後，同様に第 $i$ 階層において，第 $(i+1)$ 階層マクロタスクを定義する．

マクロタスクを階層的に生成した後，各階層のマクロタスクのコントロールフローとデータ依存を解析し，マクロフローグラフ（MFG）を生成する．次に，各階層のマクロフローグラフについて，コントロールフローとデータ依存を考慮した最早実行可能条件解析を行い，マクロタスクグラフ（MTG）を生成する．マクロタスクグラフは粗粒度タスク並列性を表す．

解析した並列性に基づき，各階層に割り当てるプロセッサ構成を決定する[18)]．ここで，プロセッサを仮想的にグルーピングした単位をプロセッサグループ（PG）と呼ぶものとし，プロセッサ構成は PG 数と PG 中のプロセッサ数で表される．マクロタスクは，PG に割り当てられて実行される．

### 3.2 ループ整合分割とデータローカライゼーション手法

ループ整合分割[6),7)] は，ループやサブルーチンといった粗粒度タスク間の並列性を抽出し，粗粒度タスク間でデータローカリティを活用する手法である．

**図 1** のマクロタスクグラフを用いてループ整合分割について説明する．図中の $RB_1$，$RB_2$，$RB_3$ はそれぞれマクロタスクを表す．マクロタスク間の実線はマクロタスクどうしが互いにデータ依存していることを表す．ループ整合分割では，同一の配列を定義・参照し，データ依存するループ（RB）を集め，これらのループ群をターゲットループグループ（Target Loop Group: TLG）に選択する．ターゲットループグループ中で最も推定コストの大きなループを標準ループ（Standard loop）と定める．標準ループとターゲットループグループ中の他のループ間で，どのイタレーションどうしが依存するかを解析する．この解析をループ間データ依存解析と呼ぶ．

図中では，$RB_1$，$RB_2$，$RB_3$ がターゲットループグループ $TLG_1$ に選択されている．各関数呼び出し先では，引数である各配列の二次元目全体が，図中に示した網かけのとおり，定義あるいは参照されるものとする．ただし，関数 $func2$ の引数配列 a は，定義の前に参照される前方露出参照とする．また，標準ループ $RB_3$ の $i = K$ のイタレーションと $RB_1$ の $i = K$，$i = K - 1$，$RB_2$ の $i = K$ のイタレーションがそれぞれループ間でデータ依存

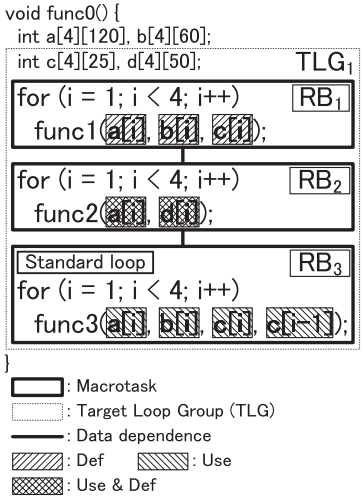



**図 1**　マクロタスクグラフ
Fig. 1　Macrotask graph.

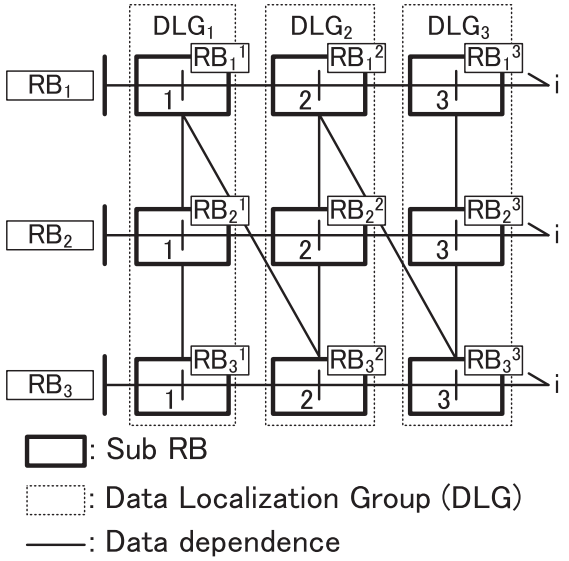



**図 2**　データローカライゼーショングループ
Fig. 2　Data Localization Group.

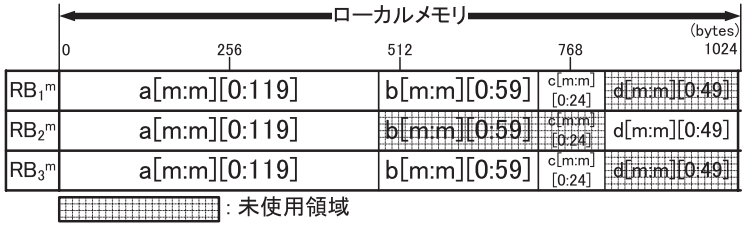



**図 3**　ループ整合分割による $DLG_m$（$1 \leq m \leq 3$）のローカルメモリ割当て
Fig. 3　Local memory allocation of $DLG_m (1 \leq m \leq 3)$ with LAD.

していると解析されるものとする．

解析後，ターゲットループグループ中のループを分割する．そして，多量のデータを共有する，分割された部分ループ群を，同一プロセッサに割り当てるべきマクロタスク集合，データローカライゼーショングループ（DLG）として定義する．

図 1 の $TLG_1$ 中のループを 1,024 bytes のローカルメモリに割り当てる場合，1 イタレーションあたりのデータサイズが 480（配列 a[K] のサイズ）+ 240（配列 b[K] のサイズ）+ 100（配列 c[K] のサイズ）+ 100（配列 c[K-1] のサイズ）+ 200（配列 d[K] のサイズ）= 1,120 bytes とローカルメモリサイズを超えるので，最大分割数（=ループの回転数）である 3 で分割される．ただし，int 型のサイズは 4 bytes とする．分割後のマクロタスクグラフを**図 2** に示す．図中には $DLG_1$，$DLG_2$，$DLG_3$ の 3 つのデータローカライゼーショングループが定義されており，$DLG_m$（$1 \leq m \leq 3$）中には $RB_1^m$，$RB_2^m$，$RB_3^m$ がそれぞれ含まれる．

ループ整合分割を用いたデータローカライゼーション手法では，分割後，各データローカライゼーショングループ中で定義・参照される部分配列を，ローカルメモリ上の一定のアドレスに固定的に割り当てる．**図 3** に $DLG_m$ 中の部分配列をデータローカライゼーション手法を用いてローカルメモリに割り当てた例を示す．図中の網かけ部は，その部分ループでアクセスがない未使用領域を表す．コロンの左右の数字は割り当てられた配列の次元ごとの下限値と上限値をそれぞれ表す．ローカルメモリに割り当てる配列は推定アクセス回数や推定データ転送時間をもとに，ローカルメモリ配置利得の大きな配列から順に選択する．この

とき，未使用領域が存在するにもかかわらず，$RB_3^m$ で必要となる $c[m-1:m-1][0:24]$ が，固定的な割当てのため，ローカルメモリに割り当てられない．また，すべての部分配列をローカルメモリに割り当てるには 1,120 bytes 必要となる．

### 3.3　ローカルメモリ管理適用対象

本節では，提案手法の適用対象について説明する．提案手法は制約付き C（Parallelizable C）[19] で記述されたプログラムを入力として想定している．制約付き C は並列化アプリケーション開発の生産性向上のため，コンパイラが自動で並列化可能な記述を目指した C プログラミング記述のガイドラインである．現状の主な制約として以下があげられる．まず，C で記述されたプログラムの解析を困難なものとしているポインタ変数の使用を制限している．例外的に関数の仮引数におけるポインタの使用は認められるが，仮引数ポインタどうしがエイリアスしていてはならない．また，構造体はメンバ変数ごとにばらされ，動的に確保されるヒープ領域は，多次元配列として宣言される．さらに，再帰呼び出しの利用も制限されている．

提案手法の適用対象データに関しては，プログラム中のグローバル配列とスタック上の自動配列変数を候補としてローカルメモリへ割り当てる．再帰呼び出しが含まれていないプログラムを前提としているので，プログラム中の自動配列変数は，いったんすべてグローバル配列へと変換される．変換された自動配列変数を含むグローバル配列はオフチップメモリ上に配置される．このオフチップメモリ上のグローバル配列を，ローカルメモリ上に確保した領域に動的に割り当てて実行する．入力として制約付き C を想定しているため，実行時に使うサイズが決定されるヒープ領域は提案手法の対象としていない．

粗粒度タスク並列処理では階層的にマクロタスクグラフを生成する．ここで，提案手法の適用対象の粗粒度タスク並列処理階層と粗粒度タスク（マクロタスク）について述べる．コンパイラによる静的な解析情報に基づくローカルメモリ管理を実現するために，粗粒度タスク並列処理階層のうち，提案手法はスタティックスケジューリング可能な，条件分岐を含まない階層を対象とする．また，マクロタスク内で定義・参照される配列を直接プロセッサのローカルメモリに割り当てるために，PG 内プロセッサ数が 1 である階層を対象とする．これらの条件に合致する階層を対象階層と呼ぶ．

従来のループ整合分割を用いたデータローカライゼーション手法では，分割された部分ループをローカルメモリ割当ての対象としているのに対し，提案手法はデータ分割の必要のない小データにアクセスするマクロタスクもローカルメモリ割当ての対象としている．また，ローカルメモリサイズを超えるマクロタスクは内側階層をさらに分割して，内側階層をローカルメモリ割当ての対象としている．内側階層の分割が可能でなければ，ローカルメモリ割当て時の利得を配列ごとに計算し，利得の大きい配列から順に収まる範囲でローカルメモリに割り当てている．

### 3.4　ローカルメモリ管理単位ブロック

配列のような連続的なアドレスを持つデータを割り当てる場合，そのサイズに応じた連続領域をローカルメモリ上に確保する必要がある．サイズや次元数が異なる配列を，ローカルメモリ上の空き領域に割り当てることは，未使用な連続領域を断片化させ，フラグメンテーションを発生させる．そこで，提案手法ではローカルメモリをブロックという単位に仮想的に区切り管理する．各ブロックには配列が 1 つ割り当てられる．また，割り当てられた配列をオフチップメモリに書き戻すことでブロックは解放される．異なるサイズのブロックどうしは，互いに重複してローカルメモリ上にマッピングされる．

各ブロックには以下のように level とブロック番号が割り振られる．ローカルメモリ上に確保できる最大のブロックサイズを $full\_block\_size$ とし，その level を 0 とする．そして，level が $l$ であるブロックの 1/2 のサイズのブロックの level を $l+1$ とする．また，各 level ごとにアドレスの低い方を 0 とし，順にブロック番号を割り振る．

ある配列を割り当てるブロックの level は次のように決まる．アクセス範囲の上下限値から，配列の次元ごとに要素数を求めて掛け合わせる．ループ回転数が変数であるなど，ある次元のアクセス要素数がコンパイル時に決定できなければ，配列次元の宣言要素数で代用する．求めた全要素数に配列要素あたりのサイズを掛けて配列のサイズ $S$ を求める．

$S$ が

$$\frac{full\_block\_size}{2^{l+1}} < S \leq \frac{full\_block\_size}{2^l} \tag{1}$$

である場合，この配列は level $l$ のブロックへ割り当てられる．また，この配列の level は $l$ であるという．

**図 4** に 1,024 bytes のローカルメモリへ level 0 から level 3 までのブロックをマッピングした例を示す．ただし，level が $l$ で，ブロック番号が $n$ のブロックを $Block_n^l$（または $B_n^l$）と表記する．たとえば，図中の $Block_5^3$ は $Block_0^0$，$Block_1^1$，$Block_2^2$ と互いに重複して配置されるので，同時に使用することはできない．

### 3.5　ローカルメモリ割当てと解放

配列単位でブロックへ割当てを行う場合，適切なブロックに割り当てないと，ローカルメモリ内でのデータの移動やオフチップメモリとローカルメモリ間のデータ転送が頻繁に発

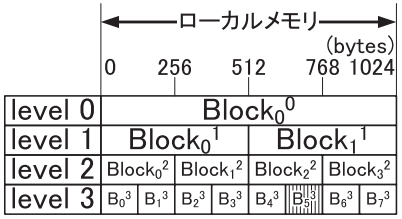



**図 4**　ローカルメモリへのブロック配置
Fig. 4　Block assignment on local memory.

生し，かえって性能が低下しうる．そのような事態を避けるために，あらかじめマクロタスクの実行順序を決定し，各配列の定義・参照時刻を求めることで，ブロック割当てやブロック解放に必要なデータ転送を最適化する．

粗粒度タスクの実行順序の決定を行う粗粒度タスクスタティックスケジューリングについて 3.5.1 項で述べる．そして，配列のブロックへの割当てについて 3.5.2 項で，ブロックの解放について 3.5.3 項でそれぞれ説明する．

#### 3.5.1　データローカリティを考慮した粗粒度タスクスタティックスケジューリング

ループ整合分割によって分割されたループを含む，対象階層中のマクロタスクの実行順序を決めるために，粗粒度タスクスタティックスケジューリングを行う．粗粒度タスクスタティックスケジューラはクリティカルパス長，データ転送時間，そして後続タスク数をプライオリティとしたヒューリスティックなリストスケジューリングである．各プライオリティの組合せのうち，最も短いスケジューリング長が得られたスケジューリング結果を採用する．スケジューリングによって，マクロタスクの実行順序と各マクロタスクのスケジューリング時刻を求める．

#### 3.5.2　ループ整合分割後の部分配列のブロックへの割当て

先行タスクのプロセッサへの割当て状況およびこれらのタスク内でアクセスされるデータのローカルメモリへの割当て結果をもとにして，データ転送を最小化するために，スケジューリング時刻の早いマクロタスクで定義・参照される配列から順にブロックへの割当てを行う．

マクロタスクで定義・参照されるローカルメモリ割当て対象の配列のうち，level の小さな，すなわちサイズの大きな配列から順に，ブロックへ割り当てる．同名の配列がすでに割当て済みのブロックおよび割当て可能な空いているブロックに，それぞれ割り当てたときのデータ転送時間を計算し，データ転送時間が最小となるブロックへ割り当てる．割当て可能なブロックが見つからなければ，3.5.3 項で述べるブロックの解放を行う．

必要なデータ転送時間に差がないブロックが複数見つかった場合，level の小さなブロックがなるべく割当て可能なまま残るように，割り当てるブロックを選択する．たとえば，図 4 において，level が 2 のサイズの配列を割り当てる際に，$Block_0^2$，$Block_1^2$，$Block_3^2$ が割当て可能で，いずれのブロックに割り当ててもデータ転送時間が等しかったとする．このとき，$Block_5^3$ が割当て済みならば，配列を $Block_3^2$ に割り当てることで，$Block_0^1$ を割当て可能なまま残す．

配列をブロックに割り当てたら，そのマクロタスクで参照する配列範囲のうち，ブロック上に載っていない範囲をオフチップメモリからロードする．

#### 3.5.3　ブロックの解放

過去に割り当てられた配列がそのプロセッサ上で再度参照されることがなければ，その配列が割り当てられているブロックは解放する．そうでなければ，参照される配列のサイズを，参照されるまでの時間で割った値である再参照度をブロックごとに求める．参照されるまでの時間はマクロタスクのスケジューリング時刻から求める．単に参照されるまでの時間で解放候補を選択すると，サイズが大きくデータ転送時間の長い配列が頻繁に解放されてしまうことがあるので，サイズもあわせて考慮する．この再参照度が小さいブロックほど将来にわたり参照される頻度が低いことを表すので，再参照度の低いブロックから順に必要に応じ解放する．

ここで，ブロックが割当て済みである場合，ブロック $Block_n^l$ の再参照度 $R_{Block_n^l}$ は

$$R_{Block_n^l} = \sum_j \frac{S_j}{T_j - T_i} \tag{2}$$

とする．ただし，マクロタスク $MT_i$ を割当て中のマクロタスクとし，マクロタスク $MT_j$ を $MT_i$ 以降に同一プロセッサに割り当てられたマクロタスクとする．$T_i$，$T_j$ をそれぞれ $MT_i$，$MT_j$ のスケジューリング時刻とする．ブロックに割り当てられた配列範囲のうち，$MT_j$ で参照されるサイズを $S_j$ とする．

割当て対象の配列のサイズに対応するブロック自体には他の配列は割り当てられていないが，level がより大きな，重複するブロックが割当て済みの場合，再参照度は以下のように下位のレベルに向かって再帰的に求める．

$$R_{Block_n^l} = R_{Block_{2n}^{l+1}} + R_{Block_{2n+1}^{l+1}} \tag{3}$$

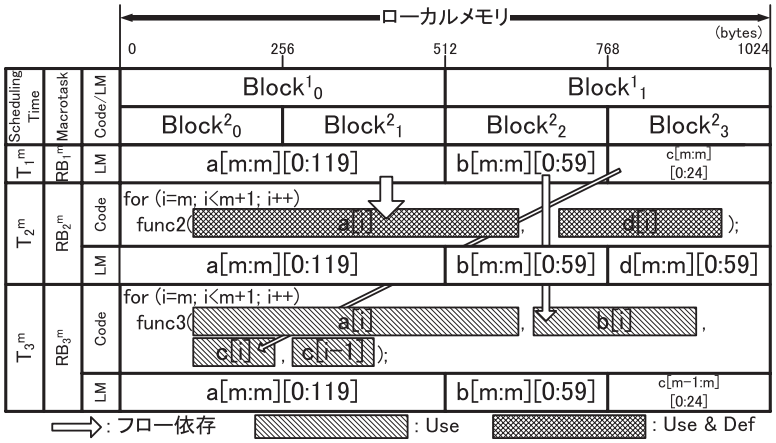



**図 5**　ローカルメモリ割当てと解放
Fig. 5　Local memory allocation and deallocation.

ただし，解放済みの割当て可能なブロックの再参照度は 0 として計算する．

解放するブロックが決定したら，そのブロック上に割り当てられていた配列をオフチップメモリに書き戻す．書き戻すタイミングはその配列を定義したマクロタスクの直後とする．

**図 5** を使って，図 2 で示した $DLG_m$ 中のマクロタスクで定義・参照される配列のブロックへの割当てと解放について説明する．図中の左側の列からスケジューリング時刻，マクロタスクのスケジューリング結果，そして右側の列がプログラムコード（Code）か，ローカルメモリの割当て結果（LM）かの区別をそれぞれ表す．右側の列はローカルメモリ割当て結果と，$RB_2^m$ 以降の配列の定義・参照関係を含むプログラムコードを表す．図中の矢印は，その配列が起点で定義され，終点で参照されることを表すフロー依存を示す．ただし，スケジューリング時刻は $T_1^m < T_2^m < T_3^m$ であるものとする．

今，$RB_1^m$ までの割当てが終わっているものとする．このとき，配列 a，b，c が $Block_0^1$，$Block_2^2$，$Block_3^2$ にそれぞれ割り当てられている．スケジューリング結果に従い，配列 a，d が定義・参照される $RB_2^m$ をブロックに割り当てる．level が 1 の配列 a から割り当てる．$Block_0^1$ に同名の配列が割当て済みなので，そこへ割り当てる．level が 2 の配列 d を割り当てるブロックがないので，$Block_2^2$，$Block_3^2$ について再参照度を計算する．$Block_2^2$ の配列 b と $Block_3^2$ の配列 c は参照されるまでの時間は等しいが，配列 b の方がサイズが大きく，再参照度が大きくなる．そこで，配列 b を残し，$Block_3^2$ を解放する．解放により，空いた $Block_3^2$ に配列 d を割り当てる．最後に，$RB_1^m$ の直後で，配列 c をオフチップメモリに書き戻す．$RB_3^m$ では，配列 a，b はそのまま利用され，空いている $Block_3^2$ に $c[m-1:m][0:24]$ がロードされる．

図 3 に示したとおり，ローカルメモリサイズが 1,024 bytes の場合，固定的な割当てを行う従来のデータローカライゼーション手法では一部の部分配列が割り当てられない．それに対し，提案手法では，同じ 1,024 bytes のローカルメモリを使い，図 5 の LM の各行に示したとおり，$RB_1^m$，$RB_2^m$，$RB_3^m$ にわたり，定義・参照される部分配列がすべてローカルメモリに割り当てられ，利用効率が向上している．このようにターゲットループグループを最大限分割したとしても，データローカライゼーショングループ内の各部分ループで定義・参照される配列サイズの合計がローカルメモリサイズに収まらないような場合，配列の定義・参照に応じたローカルメモリ管理を行う提案手法は，ローカルメモリをより有効に利用することができる．

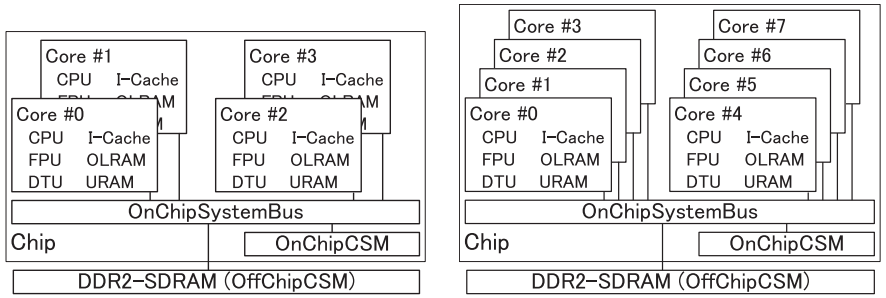
## 4. 性能評価

本章では，提案手法を OSCAR マルチグレイン自動並列化コンパイラ[17] に実装し，4.1 節で述べる RP1 および RP2 マルチコア上で性能評価した結果について報告する．

### 4.1　評価に用いるマルチコア RP1 および RP2

提案手法が任意のプロセッサ数や異なるローカルメモリサイズであっても管理できることを確かめるために，ローカルメモリサイズとプロセッサ数がそれぞれ異なる RP1 および RP2 マルチコア[2),3)] 上で性能評価を行った．RP1 および RP2 マルチコアは NEDO 半導体アプリケーションチップ「リアルタイム情報家電用マルチコア」プロジェクトにおいてルネサステクノロジ，日立製作所，早稲田大学により開発された，コンパイラ協調型マルチコアである．それぞれ，SH4A プロセッサコアを 4，8 コア集積している．

RP1 マルチコアの構成図を**図 6** (a) に，RP2 マルチコアを図 6 (b) にそれぞれ示す．また，RP1 および RP2 の仕様を**表 1** にまとめる．各コアにはローカルデータメモリ OLRAM，複数のコアから共有される分散共有メモリ URAM，プログラムコード用の命令キャッシュ，オフチップ集中共有メモリとローカルメモリ間のデータ転送に使用する DTU をそれぞれ持つ．すべてのコアとオンチップ集中共有メモリ（OnChipCSM）がシステムバス（OnChipSystemBus）で接続されている．チップ外にはオフチップ集中共有メモリ DDR2-SDRAM（OffChipCSM）が接続されている．OLRAM が提案手法が対象とするローカルメモリで




(a) PR1　　(b) PR2

**図 6**　RP1 および RP2 の構成図
Fig. 6　RP1 and RP2 processor architecture.

**表 1**　RP1 および RP2 の仕様
Table 1　RP1 and RP2 specifications.

| | RP1 | RP2 |
|---|---|---|
| プロセッサ数 | 4 | 8 |
| OLRAM レイテンシ | 1–2 クロック (*) | 1–2 クロック (*) |
| OLRAM サイズ | 16 KB | 32 KB |
| URAM レイテンシ | 2 クロック (**) | 2 クロック (**) |
| URAM サイズ | 128 KB | 64 KB |
| I-キャッシュ レイテンシ | 1 クロック | 1 クロック |
| I-キャッシュ サイズ | 32 KB | 16 KB |
| OnChipCSM レイテンシ | 約 12 クロック | 約 12 クロック |
| OnChipCSM サイズ | 128 KB | 128 KB |
| OffChipCSM レイテンシ | 約 55 クロック | 約 55 クロック |

(*)：ページ競合の有無による
(**)：URAM 前段のバッファヒット/ミスヒットによる平均

ある．また，OffChipCSM がオフチップメモリに相当する．なお，本論文における評価では，OnChipCSM を使用せず，集中共有メモリとしては OffChipCSM のみを利用するものとする．

表 1 にあるとおり，OLRAM のサイズが RP1 で 16 KB，RP2 では 32 KB となっている．

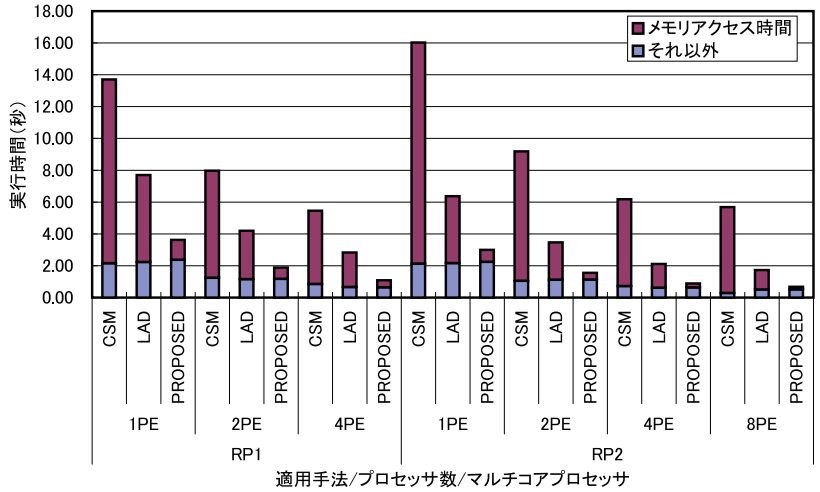



**図 7**　AAC エンコーダの実行時間
Fig. 7　AAC encoder execution time.

これら 2 種類のマルチコアを用いることにより，提案手法のローカルメモリサイズによる効果の違いを評価する．

### 4.2　評価アプリケーション

AAC エンコーダおよび MPEG2 エンコーダを用いて提案手法の性能評価を行った．これらのアプリケーションは制約付き C（Parallelizable C）[19] で記述されている．入力データが OffChipCSM 上に配置された状態から出力結果を OffChipCSM に書き戻すまでの時間を評価の対象とした．

AAC エンコーダは 30 秒の音源を入力データとし，出力データのビットレートは 128 Kbps とした．MPEG2 エンコーダは MediaBench [20] に収録されている “mpeg2encode” を参照実装したプログラムを用いた．入力データには 352 × 256 ピクセルの画像 30 フレームをそれぞれ用いた．

### 4.3　マルチメディア処理に対する性能

AAC エンコーダ，MPEG2 エンコーダに対して，提案手法を実装した OSCAR コンパイラでローカルメモリ管理を行い，RP1，RP2 上で評価した．AAC エンコーダにおける実行時間を**図 7** に，MPEG2 エンコーダにおける実行時間を**図 8** にそれぞれ示す．図中の横軸

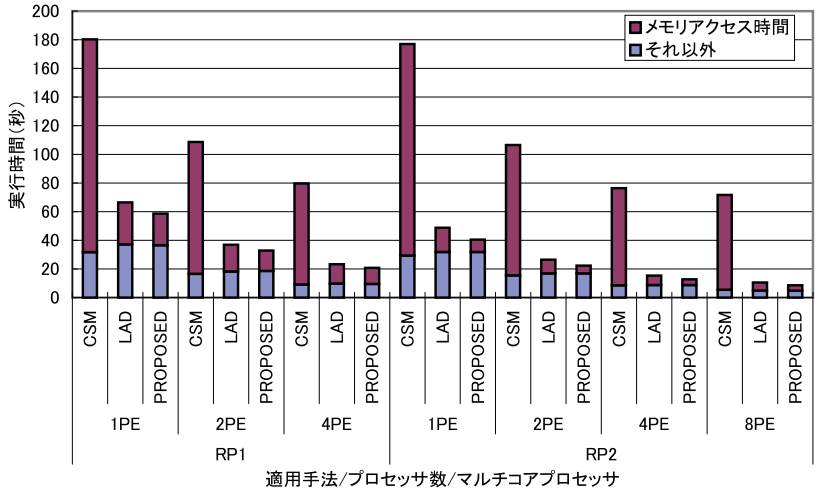



**図 8**　MPEG2 エンコーダの実行時間
Fig. 8　MPEG2 encoder execution time.

は RP1 の 1，2，4PE，RP2 の 1，2，4，8PE の適用手法別の結果をそれぞれ表す．適用手法のうち，CSM はプログラム中のすべての配列をいっさいローカルメモリに配置せず，すべて OffChipCSM に配置した場合の実行時間を表す．こちらはローカルメモリ管理手法がない場合の最も単純な処理方式を想定している．LAD は 3.2 節で述べた従来のループ整合分割を用いたデータローカライゼーション手法[6),7)] の実行時間を表す．PROPOSED は提案手法の実行時間を表す．また，各バーはメモリアクセス時間とその残りに分けられている．メモリアクセス時間はオフチップメモリにあるかローカルメモリにあるかにかかわらず，プログラム中でメモリアクセスにかかった時間および DTU によるデータ転送時間の合計を表す．

図 7 の AAC エンコーダを用いた評価において，実行時間からメモリアクセス時間を引いたそれ以外の時間が，各マルチコア，各 PE 数ごとに，適用手法によらずほぼ一定であることがグラフから読みとれる．このことから，適用手法ごとの実行時間の差はメモリアクセス時間の差に起因することが分かる．

RP1 上の 4PE 実行で，CSM と提案手法を比較すると，約 5.0 倍の速度向上を実現した．同様に RP2 上の 8PE 実行では，提案手法は CSM に対し，約 8.4 倍の速度向上を実現した．CSM ではすべての配列が OffChipCSM に配置されているため，メモリアクセスに長大な時間がかかっているのに対し，提案手法ではコンパイラによる自動的なローカルメモリ管理により，プロセッサ近傍のローカルメモリを有効利用できたことが分かる．

次に，LAD と提案手法を比較すると，RP1，RP2 のいずれのプロセッサ数においても，提案手法がより高い性能を示していることが分かる．RP1 の 4PE 実行で比較してみると，約 2.6 倍の，RP2 の 8PE 実行では，約 2.5 倍の速度向上がそれぞれ得られた．このときのメモリアクセス時間を比較してみると，RP1 の LAD では，約 2.2 秒であるのに対し，提案手法では約 0.5 秒と，約 1.7 秒向上している．同様に RP2 の LAD では，約 1.2 秒であるのに対し，提案手法では約 0.2 秒と，約 1.0 秒向上している．オフチップメモリに割り当てられたままの配列の総サイズを比較すると，プロセッサ数にかかわらず，16 KB のローカルメモリを持つ RP1 では，LAD は約 81 KB，提案手法は 0 KB，32 KB のローカルメモリを持つ RP2 では，LAD は約 50 KB，提案手法は 0 KB であった．これらより，提案手法は固定的な割当てを行う LAD ではオフチップメモリに配置されてしまった配列をすべてローカルメモリに割り当てることで，メモリアクセスにかかる時間を削減し，実行時間の短縮を実現したことが分かる．

提案手法がローカルメモリ割当てによってメモリアクセス時間を削減しているだけでなく，並列性を引き出しているかを見るために提案手法どうしで比較する．RP1 上で逐次実行に対して，4PE で約 3.3 倍，RP2 上で 8PE で約 4.4 倍とプロセッサ数に応じた速度向上が得られていることから，並列性を抽出できていることが分かる．

同一プロセッサ台数の RP1 と RP2 での提案手法のメモリアクセス時間をそれぞれ比較してみると，いずれの結果でも，16 KB のローカルメモリを持つ RP1 に対し，32 KB のローカルメモリを持つ RP2 のメモリアクセス時間は短くなっている．たとえば，4PE どうしで比較してみると，メモリアクセス時間は RP1 では約 0.5 秒であるのに対し，RP2 では約 0.3 秒と約 0.2 秒短縮されている．提案手法ではすべての配列をローカルメモリに割り当てているため，この差はほぼデータ転送時間であり，提案手法は RP2 において RP1 に比べ，より少ないデータ転送ですべての配列をローカルメモリに割り当てている．これより，提案手法はローカルメモリサイズに応じたローカルメモリ管理を実現していることが分かる．

図 8 の MPEG2 エンコーダを用いた評価において，AAC エンコーダ同様，実行時間からメモリアクセス時間を引いたそれ以外の時間が，各マルチコア，各 PE 数で，適用手法によらずほぼ一定である．このことから，適用手法ごとの実行時間の差はメモリアクセス時間の差に起因することが分かる．

RP1 上の 4PE 実行で，CSM と提案手法を比較すると，約 3.8 倍の速度向上を実現した．同様に RP2 上の 8PE 実行では，提案手法は CSM に対し，約 8.4 倍の速度向上を実現した．MPEG2 エンコーダでも AAC エンコーダ同様，コンパイラによる自動的なローカルメモリ管理により，プロセッサ近傍のローカルメモリを有効に利用していることが分かる．

次に，LAD と提案手法を比較すると，RP1，RP2 のいずれのプロセッサ数においても，提案手法がより高い性能を示していることが分かる．RP1 の 4PE 実行で比較してみると，約 1.1 倍の，RP2 の 8PE 実行では，約 1.2 倍の速度向上がそれぞれ得られた．このときのメモリアクセス時間を比較してみると，RP1，4PE の LAD では，約 13.5 秒であるのに対し，提案手法では約 11.3 秒と，約 2.2 秒向上している．同様に RP2，8PE の LAD では，約 5.7 秒であるのに対し，提案手法では約 3.8 秒と，約 1.9 秒向上している．オフチップメモリに割り当てられたままの配列の総サイズを比較すると，プロセッサ数にかかわらず，16 KB のローカルメモリを持つ RP1 上で，LAD は約 9,924 KB，提案手法は約 6,336 KB，32 KB のローカルメモリを持つ RP2 上で，LAD は約 5,032 KB，提案手法は約 3,960 KB であった．これより，提案手法は LAD に比べ，同一サイズのローカルメモリにより多くの配列を配置することで，メモリアクセス時間を削減し，実行時間短縮を達成したことが分かる．

提案手法の並列性能を見るために提案手法どうしで比較すると，逐次実行に対して，RP1 上の 4PE で約 2.8 倍，RP2 上の 8PE で約 4.7 倍の速度向上が得られていることから，プロセッサ数に応じた負荷分散が行われていることが分かる．

また，提案手法において同一プロセッサ台数の RP1 と RP2 でのメモリアクセス時間をそれぞれ比較してみると，16 KB のローカルメモリを持つ RP1 に対し，2 倍の 32 KB のローカルメモリを持つ RP2 の時間の方が短い．たとえば，4PE どうしで比較してみると，メモリアクセス時間は RP1 では約 11.3 秒であるのに対し，RP2 では約 4.1 秒と約 7.2 秒短縮している．また，上述したとおり，RP1 では約 6,336 KB の配列がオフチップメモリに残っているのに対し，RP2 では約 3,960 KB まで削減されている．このことから，提案手法はローカルメモリサイズに応じたローカルメモリ管理を実現していることが分かる．

最後に，AAC エンコーダと MPEG2 エンコーダにおける LAD と提案手法の速度向上率の差について考察する．提案手法は AAC エンコーダにおいて，LAD がローカルメモリに割り当てられなかった配列をすべてローカルメモリに割り当てている．それに対し，MPEG2 エンコーダでは，提案手法は一部の配列をローカルメモリに割り当てることができなかった．いずれの場合も提案手法は LAD に比べ，速度向上を達成しているが，オフチップメモリに残る配列の有無によって大きく速度向上率が変わることが分かる．

## 5. ま と め

本論文では，マルチコアプロセッサ上の容量制約のあるローカルメモリを有効利用するための並列化コンパイル手法を提案した．提案手法はループやサブルーチン間の並列性を利用する粗粒度タスク並列処理において，並列性とデータローカリティを考慮した分割を行うループ整合分割を利用する．従来のデータローカライゼーション手法では，分割された配列はローカルメモリに固定的に割り当てられていたのに対し，提案手法では定義・参照に応じた割当てと解放を行うことで限られたローカルメモリをより有効に利用する．提案手法を OSCAR マルチグレイン自動並列化コンパイラ上に実装し評価した．提案手法を従来のデータローカライゼーション手法と比較したところ，RP1 マルチコア上で 4 コア時，AAC エンコーダで約 2.6 倍，MPEG2 エンコーダで約 1.1 倍，RP2 マルチコア上で 8 コア時，AAC エンコーダで約 2.5 倍，MPEG2 エンコーダで約 1.2 倍の速度向上がそれぞれ得られ，提案手法の有効性が確認された．

**謝辞**　本研究の一部は NEDO 半導体アプリケーションチップ「リアルタイム情報家電用マルチコア」，「情報家電用ヘテロジニアスマルチコア」プロジェクト，早稲田大学グローバル COE「アンビエント SoC」の支援により行われた．

## 参 考 文 献

1) Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., et al.: The design and implementation of a first-generation CELL processor, *Solid-State Circuits Conference, 2005, Digest of Technical Papers, ISSCC, 2005 IEEE International*, pp.184–185, 592 (2005).
2) Yoshida, Y., Kamei, T., Hayase, K., Shibahara, S., Nishii, O., Hattori, T., Hasegawa, A., Takada, M., Irie, N., Uchiyama, K., et al.: A 4320MIPS Four-Processor Core SMP/AMP with Individually Managed Clock Frequency for Low Power Consumption, *Solid-State Circuits Conference, 2007, ISSCC 2007, Digest of Technical Papers, IEEE International*, pp.100–101, 590 (2007).
3) Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K. and Kasahara, H.: An 8640 MIPS SoC with Independent Power-Off Control of 8 CPUs and 8 RAMs by An Automatic Parallelizing Compiler, *Solid-State Circuits Conference, 2008, ISSCC 2008, Digest of Technical Papers, IEEE International*, pp.90–91, 598 (2008).

4) Kandemir, M., Kadayif, I., Choudhary, A., Ramanujam, J. and Kolcu, I.: Compiler-directed scratch pad memory optimization for embedded multiprocessors, *IEEE Trans. Very Large Scale Integration* (*VLSI*) *Systems*, Vol.12, No.3, pp.281–287 (2004).
5) Issenin, I., Brockmeyer, E., Durinck, B. and Dutt, N.: Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies, *DAC '06: Proc. 43rd annual conference on Design automation*, New York, NY, USA, pp.49–52, ACM (2006).
6) Kasahara, H. and Yoshida, A.: A Data-Localization Compilation Scheme Using Partial Static Task Assignment for Fortran Coarse Grain Parallel Processing, *Journal of Parallel Computing*, Vol.Special Issue on Languages and Compilers for Parallel Computers (1998).
7) 吉田明正，越塚健一，岡本雅巳，笠原博徳：階層型粗粒度並列処理における同一階層内ループ間データローカライゼーション手法，情報処理学会論文誌，Vol.40, No.5, pp.2054–2063 (1999).
8) Panda, P.R., Dutt, N.D. and Nicolau, A.: Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications, *1997 European Design and Test Conference* (*ED&TC '97*) (1997).
9) Angiolini, F., Benini, L. and Caprara, A.: An Efficient Profile-Based Algorithm for Scratchpad Memory Partitioning, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.24, No.11, pp.1660–1676 (2005).
10) Suhendra, V., Raghavan, C. and Mitra, T.: Integrated scratchpad memory optimization and task scheduling for MPSoC architectures, *CASES '06: Proc. 2006 international conference on Compilers, architecture and synthesis for embedded systems*, New York, NY, USA, pp.401–410, ACM (2006).
11) Udayakumaran, S., Dominguez, A. and Barua, R.: Dynamic allocation for scratch-pad memory using compile-time decisions, *Trans. on Embedded Computing Sys.*, Vol.5, No.2, pp.472–511 (2006).
12) Udayakumaran, S. and Barua, R.: An integrated scratch-pad allocator for affine and non-affine code, *DATE '06: Proc. conference on Design, automation and test in Europe*, 3001 Leuven, Belgium, European Design and Automation Association, pp.925–930 (2006).
13) Li, L., Nguyen, Q.H. and Xue, J.: Scratchpad allocation for data aggregates in superperfect graphs, *LCTES '07: Proc. 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, New York, NY, USA, ACM, pp.207–216 (2007).
14) Li, L., Wu, H., Feng, H. and Xue, J.: Towards Data Tiling for Whole Programs in Scratchpad Memory Allocation, *Advances in Computer Systems Architecture*, Vol.Volume 4697/2007, pp.63–74, Springer Berlin/Heidelberg (2007).
15) Knight, T.J., Park, J.Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W.J. and Hanrahan, P.: Compilation for explicitly managed memory hierarchies, *PPoPP '07: Proc. 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, pp.226–236, ACM (2007).
16) Ruggiero, M., Guerri, A., Bertozzi, D., Poletti, F. and Milano, M.: Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip, *DATE '06: Proc. conference on Design, automation and test in Europe*, 3001 Leuven, Belgium, European Design and Automation Association, pp.3–8 (2006).
17) Kimura, K., Wada, Y., Nakano, H., Kodaka, T., Shirako, J., Ishizaka, K. and Kasahara, H.: Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor, *Proc. 9th Workshop on Interaction between Compilers and Computer Architectures* (*INTERACT-9*) (2005).
18) 白子　準，長澤耕平，石坂一久，小幡元樹，笠原博徳：マルチグレイン並列性向上のための選択的インライン展開手法，情報処理学会論文誌，Vol.45, No.5, pp.1345–1356 (2004).
19) 間瀬正啓，馬場大介，長山晴美，田野裕秋，益浦　健，深津幸二，宮本孝道，白子　準，中野啓史，木村啓二，笠原博徳：OSCAR コンパイラにおける制約付き C プログラムの自動並列化，情報処理学会研究会報告 2006-ARC-170-01（デザインガイア 2006）(2006).
20) Lee, C., Potkonjak, M. and Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia andcommunications systems, *30th Annual IEEE/ACM International Symposium on Microarchitecture* (1997).

(平成 20 年 10 月 3 日受付)
(平成 21 年 1 月 14 日採録)

**中野　啓史**（正会員）

昭和 52 年生．平成 13 年早稲田大学理工学部電気電子情報工学科卒業．平成 15 年同大学大学院修士課程修了．平成 15 年同大学院博士課程進学．平成 21 年博士課程修了．

**桃園　　拓**

昭和 59 年生．平成 19 年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．平成 21 年同大学大学院修士課程修了．平成 21 年ソニー株式会社入社，現在に至る．

**間瀬　正啓**（学生会員）

昭和 58 年生．平成 17 年早稲田大学理工学部電気電子情報工学科卒業．平成 19 年同大学大学院理工学研究科情報・ネットワーク専攻修士課程修了．平成 19 年同大学院基幹理工学研究科情報理工学専攻博士後期課程進学．平成 20 年早稲田大学基幹理工学部情報理工学科助手．

**木村　啓二**（正会員）

昭和 47 年生．平成 8 年早稲田大学理工学部電機工学科卒業．平成 13 年同大学大学院理工学研究科電気工学専攻博士課程修了．平成 11 年早稲田大学理工学部助手．平成 16 年同大学理工学部コンピュータ・ネットワーク工学科専任講師．平成 17 年同助教授．平成 19 年同大学情報理工学科准教授，現在に至る．マルチコアプロセッサのアーキテクチャとソフトウェアに関する研究に従事．

**笠原　博徳**（正会員）

昭和 55 年早稲田大学理工学部電気工学科卒業，昭和 60 年同大学大学院博士課程修了，工学博士，昭和 61 年早稲田大学理工学部専任講師，平成 9 年同教授，現在情報理工学科教授，アドバンストマルチコアプロセッサ研究所長．昭和 60 年カリフォルニア大学バークレー校，平成元年〜2 年イリノイ大学 Center for Supercomputing R & D 客員研究員．昭和 62 年 IFAC World Congress Young Author Prize，平成 9 年情報処理学会坂井記念特別賞，平成 20 年 LSI オブザイヤー準グランプリ受賞．IEEE Computer Society 理事，情報処理学会 ARC 主査，論文誌 HG 主査，文部科学省地球シミュレータ中間評価委員，経済産業省/NEDO“アドバンスト並列化コンパイラ”“リアルタイム情報家電用マルチコア” 等プロジェクトリーダ歴任．