組込マルチコア用OSCAR APIを用いた TILEPro64上でのマルチメディアアプリケーションの並列処理

岸本耀平[†] 見神広紀[†] 中野恵一[††]
林 明宏[†] 木村啓二[†] 笠原博徳[†]

組み込み分野においてもマルチコア・メニーコアは広く利用され，そのコア数は今後ますます増加する．しかしながら手動並列化によりコア数の増加に応じたアプリケーションの性能向上を得るのは費用・期間の面から困難となっている．本稿では C 言語で記述されたマルチメディアアプリケーションを OSCAR 自動並列化コンパイラを用い並列化し，情報家電マルチコア用並列化 API である OSCAR API を挿入した並列プログラムを自動生成すると共に、生成プログラムを 64 コアの Tilera 社 TILEPro64 メニーコアプロセッサ上で実行するときにデータのキャッシュへの割り付け方式について検討し，TILEPro64 で並列処理した際の処理性能について報告する．

64 コアを用いた性能評価の結果，OSCAR コンパイラによる並列化により，各スレッドがアクセスするメモリ領域は適切に分割されプロセッサ近接のキャッシュに割当てられるため，TILEPro64 上では，ヒープや.bss のページをローカルなキャッシュ上に適切に配置することにより，1 コアでの実行に対し JPEG XR エンコーダで 55 倍，Optical Flow で 30 倍，MPEG2 エンコーダで 15 倍，AAC エンコーダで 47 倍の性能向上が得られ，OSCAR 自動並列化コンパイラがメニーコアにおいてもコア数増加に応じたスケーラブルな性能向上を得られることが確認できた．また TILEPro64 上で高いスケーラビリティを得るために必要となるキャッシュ利用設定が明らかになった．

Parallel processing of multimedia applications on TILEPro64 using OSCAR API for embedded multicore

Yohei Kishimoto,[†] Hiroki Mikami,[†] Keiichi Nakano,[††]
Akihiro Hayashi,[†] Keiji Kimura[†] and Hironori Kasahara[†]

Multicore processors and many-core processors have been used widely in embedded areas. The number of cores in these multi/many-cores in increasing more and more. However, it is difficult to achieve scalable performance improvement along with the increasing numbers of cores with parallelized applications by hand because of the cost and time. This paper describes the performance of several automatically parallelized multi-media applications with considering cache assignment method on 64-cores TILEPro64 many-core processor. These applications are written in C language, and are parallelized by OSCAR automatic parallelization compiler. OSCAR Compiler generates parallelized C programs by inserting compiler directives of OSCAR API, which enables parallel processing on the multicore for consumers electronics.

Memory regions accessed by threads are devided properly and assigned to the cache near the processor by OSCAR Compiler. By assigning heap/.bss page to the local cache, the evaluation results using 64-cores show 55 times speedup on JPEG XR encoder, 30 times speedup on optical flow calculation, 17 times speedup on MPEG2 encoder and 47 times speedup on AAC encoder compared to sequential execution. These results show that the OSCAR automatic parallelization compiler can achieve scalable performance improvement along with increasing numbers of cores. This also reveal a necessary configuration for cache utilization to achieve higher scalability on TILEPro64.

1. はじめに

マルチコアプロセッサがモバイル機器，カメラから医療機器，スーパーコンピューターまで広く普及しはじめている．さらに並列処理による性能向上をはかるため，チップ内に搭載するコア数を増加させたメニーコアプ

† 早稲田大学
Waseda University
†† オリンパス株式会社
Olympus Corporation

ロセッサが注目を集めており，Tilera 社[1] からは汎用コアを 64 基搭載したメニーコアである TILEPro64[2] が出荷されている．

マルチコアの応用分野としてマルチメディア処理の高速化，低消費電力化の要求は依然として高く，マルチコアにおける並列処理の先行研究が多く存在する．またメニーコアの代表的存在である TILEPro64 および TILE64 の利用事例としては H.264 デコーダのデブロッキングフィルタの並列化[3]，Motion JPEG Decoder の並列化[4] などがある．しかしながら，これらの研究において各アプリケーションは手動で並列化を行なわれており、対象のアプリケーションに固有の並列化を行なわなければならないため汎用性に欠け，また並列プログラムの開発に長期間と大きな開発費を要するという問題点がある．

一般にプログラムの手動による並列化には上記のような問題点があり，その生産性は低く，製品競争力を高めるのにプログラムの自動並列化に期待が集まっている．

マルチコア・メニーコア用に最適化された並列化アプリケーションの生産性を向上するために，我々は OSCAR コンパイラ[5] を開発し，プログラムの自動並列化を行なってきた．OSCAR コンパイラではマルチグレイン自動並列化[6] によるプログラム全域の並列性の抽出，データローカライゼーション[7)8] によるキャッシュ利用の最適化を行うことによりマルチコアプロセッサにおいて高いスケーラビリティを得ることが可能となる．また OSCAR API[9] の利用により，マルチプラットフォームへの対応を行なってきた．特に組み込み情報家電用マルチコア[10] 上においては，OSCAR API を用いることにより電力制御やリアルタイム制御などプロセッサ資源の自動的な利用が実現されている．

メニーコアプロセッサを対象にした自動並列化では，アプリケーションのデータアクセスオーバヘッドを低減するためにキャッシュ配置の制御最適化が課題である．

本稿では OSCAR コンパイラにより，OpticalFlow, JPEG XR[11] エンコーダ，MPEG2 エンコーダ，AAC エンコーダに対し自動並列化を行い，OSCAR API を挿入したコードを自動生成した上で，TILEPro64 のキャッシュ利用設定を変更した際の並列処理性能を評価した．

以下 2 章では OSCAR コンパイラの概要，3 章で OSCAR API の概要，4 章で TILEPro64 の概要，5 章で性能評価について述べる．

2. OSCAR コンパイラ

本章では OSCAR コンパイラの概要について述べる，OSCAR コンパイラは C および Fortran に対応したコンパイラであり，従来利用されてきたループ並列性のみならずプログラム全域の並列性を利用するマルチグレイン自動並列化を行う．また複数ループ間のキャッシュ利用の最適化を行うデータローカライゼーション，OSCAR API によるコード出力を行う． マ

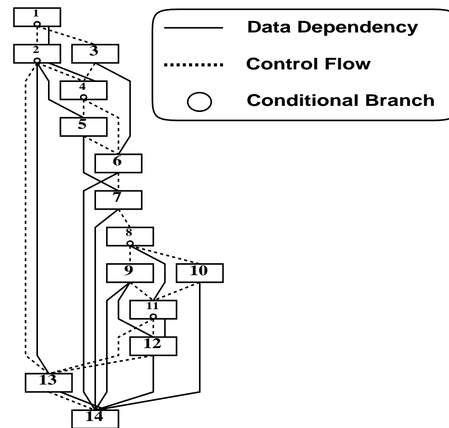

図 1 マクロフローグラフ
Fig. 1 Macro Flow Graph

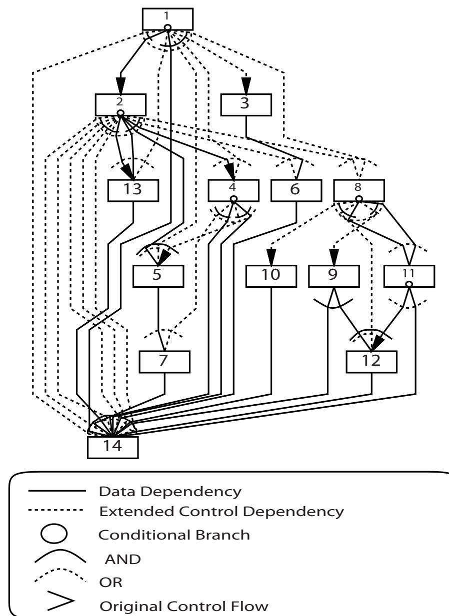

図 2 マクロタスクグラフ
Fig. 2 Macro Task Graph

ルチグレイン自動並列化では，複数の関数呼び出し間に存在する粗粒度並列性，ループ間の中粒度並列性，ステートメント間の近細粒度並列性を組み合わせて並列処理を行う．

粗粒度並列処理においては，ソースプログラムを3種類のマクロタスク (MT) すなわち基本ブロック (BB)，繰り返しブロック (RB)，サブルーチンブロック (SB) に分割し，また MT 内部でも分割を行うことで階層的なマクロタスクを生成する．MT 間の入出力変数を解析することによりマクロフローグラフ (MFG) を生成し，その後各 MT の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．図1に MFG の例，図2に MTG の例をそれぞれ示す．MTG は MT 間の並列性を表現しており，並列実行可能な MT をプロセッサに割り当てることにより並列化を行う．この際 MTG がデータ依存エッジしか持たない場合にはスタティックスケジューリングにより MT の割り当てを行い，コントロール依存エッジを持つ場合にはダイナミックスケジューリングルーチンを生成し，プログラム実行時に MT の割り当てを行う．

データローカライゼーションでは，複数のループに対してデータの利用範囲が一致するように MT を分割するループ整合分割を行った後，MT 間のデータ共有量を計算し，データを共有する MT が同じプロセッサで実行されるようにスケジューリングを行う．これによりキャッシュを有効活用した並列処理を行うことができる．

OSCAR コンパイラが出力する並列ソースコードは OpenMP をベースにした OSCAR API を用いて出力される．このとき，プログラム中一度だけスレッドのフォークを行うワンタイムシングルレベルスレッド生成によりスレッド生成オーバーヘッドを最小化している．

3. OSCAR API

OSCAR API は情報家電用ホモジニアス及びヘテロジニアスマルチコアプロセッサ用並列化プログラム記述 API であり，並列実行指示文，データのメモリ配置指示文，DMA によるデータ転送指示文，電力制御指示文，グループバリア同期指示文，リアルタイム制御指示文から構成されている，OSCAR API は OpenMP をベースとして策定されているため，OpenMP コンパイラに通すことにより並列化実行バイナリを得ることができる．

OpenMP ではサポートされていない電力制御指示文等を利用した並列 C コードを並列バイナリに変換する場合は，OSCAR API 標準解釈系[12] を利用する．OSCAR API 標準解釈系は OSCAR API をランタイム関数に変換する．新規のプロセッサに対して OSCAR API を適用する場合は，この標準解釈系の生成するランタイム関数の定義を，対象プラットフォームに合わせて記述すれば自動並列化された並列 C あるいは Fortran プログラムを各社のマルチコア・メニーコア上で実行できる．このようにして，様々なプラットフォームに対して低コストで標準解釈系の移植が可能となり，逐次コンパイラさえ用意されていれば各社の共有メモリ型マルチコア・メニーコア上で OSCAR コンパイラによる自動並列化が利用できる．

4. メニーコアプロセッサ TILEPro64

本章では，評価対象メニーコアプロセッサ TILEPro64 の基本的なアーキテクチャについて述べる．また並列処理性能に影響を与える要素であるキャッシュホーミングストラテジについて説明する．

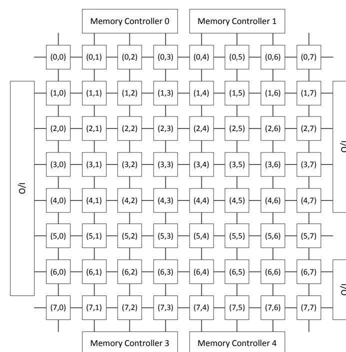

図 3 TILEPro64 ブロック図
Fig. 3 TILEPro64 block diagram

4.1 プロセッサコア

図 3 に TILEPro64 のブロック図[13] を示す．TILEPro64 は 64 個のプロセッサコアを1つのチップに収めたメニーコアプロセッサである．プロセッサコアの命令セットアーキテクチャは MIPS ベースで，3命令同時実行可能の VLIW である．また浮動小数点演算器を持たず，浮動小数点演算はエミュレーションにより実行される．各プロセッサコアは 8×8 のタイル状に配置され，図3に示すようなメッシュ状ネットワークにより接続されている．

4.2 キャッシュホーミングストラテジ

TILEPro64 プロセッサではディレクトリベースのキャッシュコヒーレンシプロトコルが利用されており，

キャッシュコヒーレンシ制御を行うコア (Home tile) においてキャッシュラインの管理が集中的に行なわれる．どのコアが Home tile になるかは図 4 のようにメモリ確保時にページ単位で指定することが可能であり，メモリ確保を行ったコアと Home tile の配置によって以下の 3 つのキャッシュホーミングストラテジが存在する．

```
tmc_alloc_t alloc = TMC_ALLOC_INIT;
//Local Homing に設定
tmc_alloc_set_home(&alloc, MAP_CACHE_HOME_TASK);
p1 = tmc_alloc_map(&alloc, size);

//Remote Homing に設定
tmc_alloc_set_home(&alloc, MAP_CACHE_HOME(n));
//Hash for Home に設定
tmc_alloc_set_home(&alloc, MAP_CACHE_HOME_HASH);
```

図 4 キャッシュホーミングストラテジの明示的な指定方法

Local Homing メモリ確保を行ったコアが Home tile となり、処理中のコアで利用するキャッシュを自身の L2 コントローラで管理する．ローカル L2 キャッシュに要求されたキャッシュラインが存在しなかった場合，ローカル L2 コントローラーは直接メインメモリにアクセスする．

Remote Homing メモリ確保を行ったコアと異なる 1 つの Home tile が指定される．Home tile でないコアにおいてローカル L2 ミスが発生した際，該当キャッシュラインの要求は Home tile に伝えられ，Home tile の L2 コントローラは Home tile の L2 キャッシュに要求されたキャッシュラインが存在するか確認する．存在する場合，リモート L2 ヒットとなり，存在しない場合はメインメモリにアクセスする．

Hash for Home メモリ上の 1 ページをキャッシュライン単位でハッシュ化を行い，複数のコアが Home tile となる．これにより Home tile の L2 キャッシュを分散 L3 キャッシュとして利用可能になり，L2 キャッシュバンド幅を有効活用しリクエストを分散させることができる．

4.3 Hash for Home の制御

プロセスが OS 上で動作する際に使用するメモリ領域はスタック領域，ヒープ領域，.bss 領域，.text 領域および読み取り専用領域に分かれるが，これらの領域に対するキャッシュホーミングストラテジをプログラムの実行時に環境変数 `LD_CACHE_HASH` により大域的に指定できる．以下にそれぞれの `LD_CACHE_HASH` の値がどの領域を含み，どのような場合に有効であるかを示す．

all すべての領域が Hash for Home として確保される．プロセス・スレッドの実行に全てのコアが積極的に利用されない際に，利用されないコアのキャッシュを利用できるため有効である．

allbutstack スタック以外の領域が Hash for Home, スタックは Local Homing として確保される．一般にスタックはスレッドごとに確保され，他のスレッドとデータを共有することは無いため，スタックのデータを分散させるのはキャッシュのサイズを確保する点でしか利点がなく，逆に他のコアのキャッシュを圧迫してしまう．このため，`allbutstack` はシステムのデフォルトに設定されている．

static スタックおよびヒープ領域は Local Homing，その他の領域は Hash for Home として確保される．ヒープ領域がスレッド間・プロセス間で共有されない場合に有効であると考えられる．

ro 読み取りのみのデータ (.rodata セクション) および命令データ (.text セクション) をハッシュ化する．グローバル変数が積極的にスレッド間で共有されない場合に有効であると考えられる．

none すべての領域が Local Homing として確保される．各コアでメモリ領域を共有しないプロセスを動作させる際に有効であると考えられる．

表 1 キャッシュおよびメモリアクセスのレイテンシ
Table 1 latencies of cache and memory access

Level	cycles
L1D	2
Local L2	8
Remote L2	30-60
Main Memory	80

キャッシュおよびメモリアクセスのレイテンシを表 1 に示す．リモートキャッシュへのアクセスレイテンシ (30-60 サイクル) はローカルキャッシュへのアクセスレイテンシ (8 サイクル) と比較して大きいため，適切なキャッシュホーミングストラテジおよび `LD_CACHE_HASH` の選択が．高速なデータアクセスを行うために重要である．

5. 性能評価

本章では 4 章で述べた TILEPro64 プロセッサを OSCAR コンパイラにより並列化されたメディアアプリケーションを用いて評価を行った結果について述べる．さらに性能解析を通し，スケーラビリティに影響を与える要素を明らかにする．

5.1 評価環境

本評価では TILEPro64(TLR36480) を搭載した TILEncore Card を用いた．TILEncore Card はホス

トシステムと PCI-Express により接続されており，ホストシステムからは `tile-monitor` により OS の起動・バイナリの実行等の制御を行うことができる．TILEPro64 上では linux-2.6.36 が動作しており，OS からは各コアが SMP として認識されるが，PCIExpress ドライバが 2 コア占有するため，OS・アプリケーションからは 62 コアまでしか認識されない．このため 64 コア実行時はアプリケーションバイナリを含んだブートイメージから起動する．62 コア未満での実行時は `tile-monitor` を用いる．各アプリケーションは gcc 4.3.3 ベースの `tile-gcc` を用いてコンパイルオプション -O3 -lpthread によりコンパイルを行う．

5.2 対象アプリケーション

以下に今回評価の対象とするメディアアプリケーションの概要を示す．いずれのアプリケーションも Parallelizable C[14) に準拠して記述されている．

Optical Flow　物体の画像間の動きを検出するアプリケーションであり，移動体の追跡や，動体認識で用いられている．画像の速度ベクトルの集合をオプティカルフローといい，本アプリケーションではブロックマッチング法により求める．ブロックシフト演算，差分演算を Y 方向，X 方向に 2 重のループ処理で行うが，Y 方向はイタレーション間に依存がない DOALL ループである．1920×1080 の 2 枚の画像を入力とする．

JPEG XR Encoder[15)]　次世代画像規格 JPEG XR の圧縮を行うアプリケーションである．JPEG XR では，従来画像の圧縮に用いられてきた JPEG に対して高圧縮率で，多様なカラーフォーマットへの対応があることが特徴である．JPEG XR 画像は複数のタイルが画像を構成し，タイルはマクロブロックにより構成されている．画像を複数のタイルに分割して圧縮を行う際，縦方向のタイル間に依存が無いことを利用してタイルレベルで並列化を行なっている．2560×2048 の画像を入力とする．

AAC Encoder　株式会社ルネサス テクノロジ提供のアプリケーションで，フレーム間の処理に依存がないため，OSCAR コンパイラでは中粒度の並列性として抽出可能である．入力には 30 秒の wav ファイルを用い，128kbps で出力する．

MPEG2 Encoder　Media Bench2[16) に収録されているソースコードを Parallelizable C により参照実装したものであり，OSCAR コンパイラではマクロブロック間の並列性を抽出する．マクロブロックレベル処理は複数のループにわたって行なわれるが，イタレーション間に依存があるループが含まれるため，通常の並列化コンパイラによるループ並列処理では各ループで参照するデータの容量がキャッシュサイズを超えてしまう．このため複数ループに対してループ整合分割を行うことでループの並列性を粗粒度タスクに変換し，データローカライゼーションを適用することによってキャッシュ利用率を向上させている．

これらのアプリケーションに対し，OSCAR コンパイラにより自動並列化を行い OSCAR API を用いたコードを出力し，このコードを OSCAR API 標準解釈系に通すことにより各コア用の並列化ソースコードを得た．

本評価においては並列処理性能を評価するために，I/O 処理の時間を除外し，演算処理部分のみを評価の対象とした．

5.3 評価結果

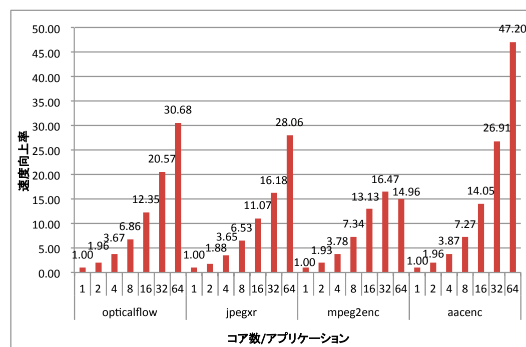

図 5　TILEPro64 における速度向上率
Fig. 5　Speedup ratio on TILEPro64

TILEPro64 における並列処理性能の評価結果を図 5 に示す．ここでは `LD_CACHE_HASH` はデフォルトである `allbutstack` に固定して評価を行った．図中横軸はアプリケーションとコア数を示し，縦軸は逐次実行時に対する速度向上率を示している．図 5 より，64 コア実行時の逐次実行時と比較し，opticalflow で 30.68 倍，JPEG XR エンコーダで 28.06 倍，MPEG2 エンコーダで 14.96 倍，AAC エンコーダで 47.20 倍の性能向上がそれぞれ得られた．

次に，各アプリケーションについて `LD_CACHE_HASH` を変えて評価を行った結果を，図 6 に OpticalFlow，図 7 に JPEG XR エンコーダ，図 8 に MPEG2 エンコーダ，図 9 に AAC エンコーダとして示す．図中横軸はコア数，縦軸は逐次実行時の `allbutstack` に対する速度向上率を示している．

opticalflow では図 6 より，1 コアから 64 コアにおいて `allbutstack`，`static`，`ro`，`none` で同等の速度向上率を示しているが，`all` はこれらに比べ速度

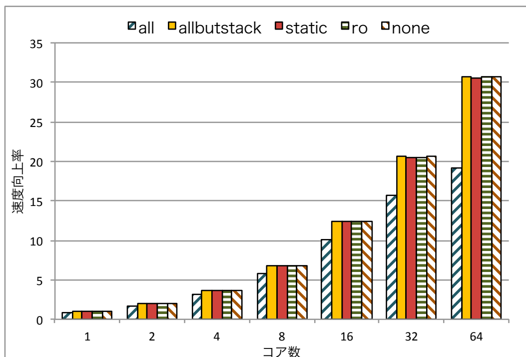

図 6 速度向上率 (opticalflow)
Fig. 6 Speedup ratio(opticalflow)

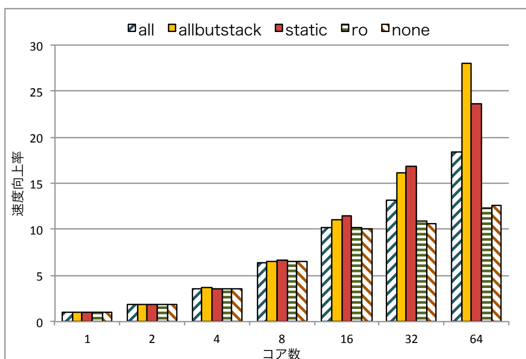

図 7 速度向上率 (jpegxr)
Fig. 7 Speedup ratio(jpegxr)

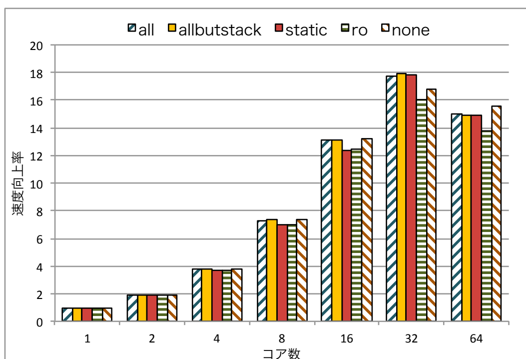

図 8 速度向上率 (mpeg2enc)
Fig. 8 Speedup ratio(mpeg2enc)

向上率が悪化している．例えば 64 コアで all のとき速度向上率は 19.2 倍であるのに対し，`allbutstack`, `static`, `ro`, `none` ではそれぞれ 30.6 倍, 30.6 倍, 30.7 倍, 30.6 倍である．図 7 の jpegxr では 32 コアまでは `static` が最も高い速度向上率を示し，`allbutstack`, `all`, `ro`, `none` の順に速度向上率が高い．64 コアにおいては `allbutstack` が 28.1 倍，`static` が 23.7 倍と

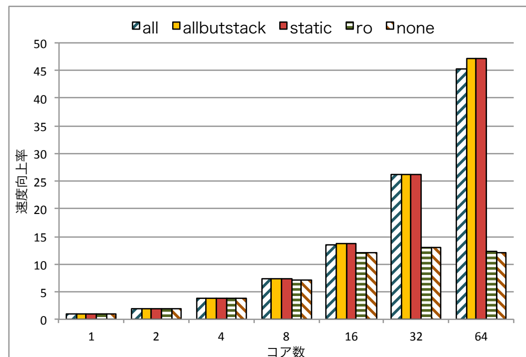

図 9 速度向上率 (aacenc)
Fig. 9 Speedup ratio(aacenc)

`allbutstack` が `static` よりも高い速度向上率を示した．aacenc では図 9 より，`allbutstack` と `static` がほぼ同じ速度向上率を示している．`all` ではこれらに比べわずかに低い速度向上率を示しているが，これはスタック上のデータサイズが小さいためと考えられる．`ro` と `none` は 16 コアから速度向上していない．mpeg2enc では図 8 より，32 コアで `all`, `allbutstack`, `static`, `ro`, `none` の速度向上率はそれぞれ 17.7 倍，17.9 倍，17.9 倍，16.1 倍，16.8 倍に対し 64 コアで 15.0 倍，15.0 倍, 15.0 倍, 13.8 倍, 15.6 倍であり，すべての場合で速度向上率が 32 コアより低くなっている．また，16 コアまでは `static` が `all`, `allbutstack` に対して低い速度向上率であるが，32 コア以上ではほぼ同等の速度向上率を示している．

5.4 性能解析

性能評価結果に対して，データのキャッシュアクセス先に注目した解析を行った．アクセスの測定にはプロファイラ `tile-oprofile` を用い、イベントカウンタの値を取得した．

各アプリケーションについて，`LD_CACHE_HASH` の値を設定することにより各領域のキャッシュホーミングモードを変更し，1 コアで逐次処理を行う場合と 32 コアで並列処理を行う場合で，処理に使われている全てのコアのリード・ライトキャッシュアクセスがローカル・リモートのキャッシュいずれにヒットしたかを測定した．その結果を opticalflow について図 10, jpegxr について図 11，mpeg2enc について図 12，aacenc について図 13 にそれぞれ示す．図中の凡例 `LOCAL_DRD`, `REMOTE_DRD`，`LOCAL_WR`，`REMOTE_WR` はそれぞれローカルキャッシュへのリード，リモートキャッシュへのリード，ローカルキャッシュへのライト，リモートキャッシュへのライトのアクセス回数をそれぞれ示している．横軸はアプリケーション，コア数，`LD_CACHE_HASH` の

値を示し，縦軸はアクセスの割合を示す．

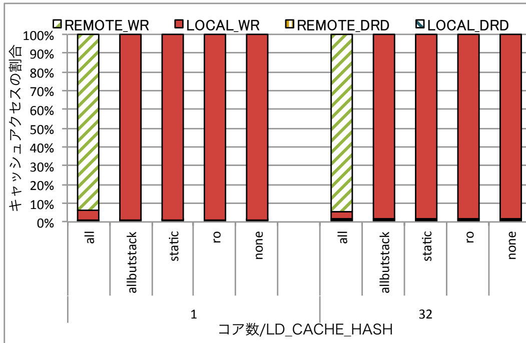

図 10 データのアクセス先 (opticalflow)
Fig. 10 Destination of data accesses(opticalflow)

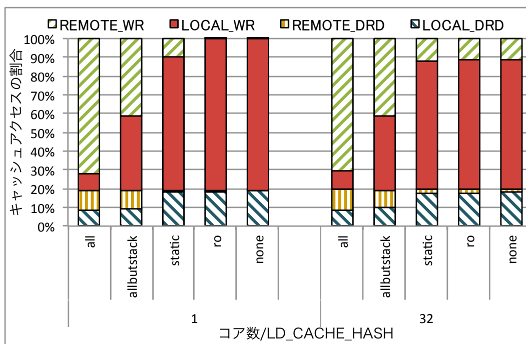

図 11 データのアクセス先 (jpegxr)
Fig. 11 Destination of data accesses(jpegxr)

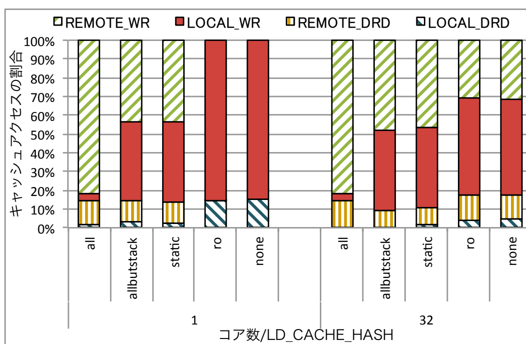

図 12 データのアクセス先 (mpeg2enc)
Fig. 12 Destination of data accesses(mpeg2enc)

図 10 より，opticalflow では，1 コアと 32 コアでの実行時でアクセス割合に大きな差はみられず，`LD_CACHE_HASH` が `all` から `allbutstack` になると

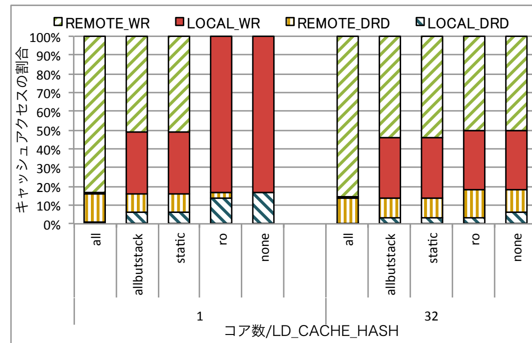

図 13 データのアクセス先 (aacenc)
Fig. 13 Destination of data accesses(aacenc)

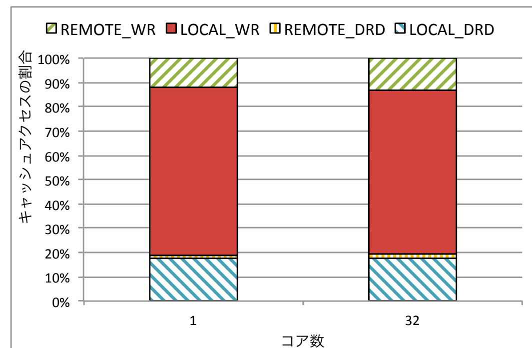

図 14 ヒープ領域を Local Homing に変更した場合のデータのアクセス先 (jpegxr)
Fig. 14 Destination of data accesses(jpegxr) with Local Homing

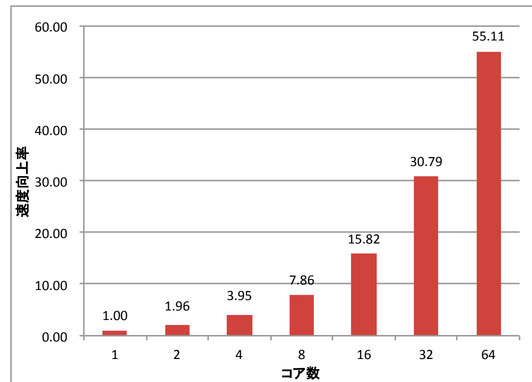

図 15 ヒープ領域を Local Homing に変更した場合の速度向上率 (jpegxr)
Fig. 15 Speedup ratio(jpegxr) with Local Homing

全体の 90%以上を占めていたリモートキャッシュアクセスが 1%以下に減少することから，本プログラムにおけるキャッシュアクセスのほとんどをスタック領域へのアクセスが占めていることがわかる．また，32 コアで `none` の場合，ローカルキャッシュアクセスが

100%を占めることから，opticalflow のメモリアクセスの多くがスレッド間で共有されないスタック上のものであることがわかる．このため，all を除いてはスタック領域のキャッシュ配置が適切に行なわれ，高いスケーラビリティが得られたと考えられる．

図11, 図12, 図13より，jpegxr，aacenc，mpeg2encでは，32コアでnoneの時に，それぞれ12.5%，61.8%，43.7%をリモートキャッシュアクセスが占める．noneではハッシュ化は行なわれず全てのメモリ領域がLocal Homingとなり，メモリ確保を行ったコアからのキャッシュアクセスは全てローカルキャッシュアクセスとなるため，リモートキャッシュアクセスの存在は他のコアからのキャッシュライン要求があり，コア間でデータが共有されることを示している．

図 11 より jpegxr において 32 コア使用時の allbutstack と static の比較をすると，リモートキャッシュアクセスが allbutstack の時 51.2%に対し static の時 14.7%に減少するため，本来コア間で共有されないヒープ領域がハッシュ化によりリモートキャッシュアクセスされてしまっていることが示される．そのため，図7において static が allbutstack より良い性能を示したと考えられる．また図12より mpeg2enc でも 32 コアで static と ro のリモートキャッシュアクセスを比較すると 55.4%から 44.4%に減少するため，共有されない未初期化静的変数領域（.bss）がハッシュ化されていることが同様に示される．

jpegxr と mpeg2enc ではスレッド間非共有データがヒープ領域等，同一の Hash for Home 管理単位上に存在することにより，ハッシュ化されてしまっている可能性がある．共有されない領域のハッシュ化は，利用コア数が少ない場合には他のコアのキャッシュを有効活用できるが，利用コア数が多い場合には他のコアのキャッシュを圧迫し，またキャッシュアクセス時間を増加させるために性能低下を起こすと考えられる．

jpegxr ではタイルレベル処理で用いる特定のヒープ領域が共有されないため，プログラム中このヒープ領域の確保時に明示的に Local Homing と指定することでヒープ上の非共有領域をローカルキャッシュに割り当てることにより性能改善がみられた．図 15 に特定のヒープを Local Homing で確保して性能評価を行った結果を示す．また図 14 に Local Homing として確保した場合のローカルキャッシュアクセス・リモートキャッシュアクセスの割合を示す．図 14 より，jpegxr の 32 コアのアクセス割合が図 11 における static の場合と同等となり，ヒープ領域のデータがローカルキャッシュへ配置されたことがわかる．このようにデータアクセスのローカリティを考慮してキャッシュ配置を行った結果，図 15 より 64 コアでの実行に 1 コアでの実行と比較して 55 倍の速度向上率となり，変更前の allbutstack と比較して 40%の性能向上を得ることができた．

mpeg2enc では，共有されていない.bss 領域へのキャッシュアクセスの割合は比較的大きく，性能に影響を与えていると考えられる．しかしながら今回用いた TILEPro64 用評価環境では.bss 上に配置されたデータをスレッドローカルのキャッシュに適切に配置することが困難であるため，スケーラブルな実行結果が得られていない．

aacencでは，図13より，32コア使用時allbutstack，static，ro，none のローカルキャッシュアクセスはそれぞれ 35.4%，35.8%，35.3%，38.2%とほぼ同等で，キャッシュ配置が適切であることが示される．そのため図 9 より 64 コアで 47 倍とコア数に応じて性能向上が得られたと考えられる．

以上をまとめると，OSCAR コンパイラによる並列化により，各スレッドがアクセスするメモリ領域は適切に分割されており，さらに TILEPro64 のようなメニーコアでコア数増加に応じたスケーラブルな性能向上を得るためには，ヒープや.bss のページをローカルなキャッシュ上に適切に配置することが重要であることが確認できた．また，必要とするコア数がチップ上のコア数より少ない場合，リモートキャッシュアクセスを許容することで性能向上する可能性があることも確認できた．

6. おわりに

本論文では OSCAR コンパイラと OSCAR API を利用して自動並列化が行われたメディアアプリケーションの組み込み向けメニーコアプロセッサ TILEPro64 における性能評価について述べた．評価の結果，64 コア使用時に逐次実行時と比較して Optical Flow で 30 倍，JPEG XR エンコーダで 55 倍，MPEG2 エンコーダで 15 倍，AAC エンコーダで 47 倍の性能向上が得られることが確認できた．また TILEPro64 においてスケーラブルな性能を得るためには、ヒープや.bss のページをローカルなキャッシュ上に適切に配置することが必要であり、適用により最大で 40%の性能向上が得られた．

参 考 文 献

1) Tilera corporation. http://www.tilera.com/.

2) S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C.Ramey, D.Wentzlaff, W.Anderson, E.Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 88 –598, 2008.
3) C. Yan, F. Dai, Y. Zhang, Y. Ma, L. Chen, L. Fan, and Y. Zheng. Parallel deblocking filter for h.264/avc implemented on tile64 platform. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pp. 1–6, 2011.
4) X. Lin, C. Huang, P. Yang, T. Lung, S. Tseng, and Y. Chung. Parallelization of motion jpeg decoder on tile64 many-core platform. In *Proceedings of the Second Russia-Taiwan conference on Methods and tools of parallel programming multicomputers*, pp. 59–68, 2010.
5) H. Kasahara, M. Obata, and K.Ishizaka. Automatic coarse grain task parallel processing on smp using openmp. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, pp. 189–207, 2001.
6) 小幡元樹, 白子準, 神長浩気, 石坂一久, 笠原博徳. マルチグレイン並列処理のための階層的並列処理制御手法. 情報処理学会論文誌, 2003.
7) 吉田明正, 前田誠司, 尾形航, 笠原博徳. Fortran マクロデータフロー処理における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. 35, No. 9, pp. 1848–1860, 1994.
8) 小高 剛, 中野 啓史, 木村 啓二, 笠原 博徳. データローカライゼーションを伴う MPEG2 エンコーディングの並列処理 (コンパイラ技術). Vol. 2004, No. 12, pp. 13–18, 2004.
9) K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako, and H. Kasahara. Oscar api for real-time low-power multicores and its performance on multicores and smp servers. Vol. 5898, pp. 188–202, 2010. 10.1007/978-3-642-13374-9_13.
10) M. Ito, T. Hattori, Y. Yoshida, K. Hayase, T. Hayashi, O. Nishii, Y. Yasu, A. Hasegawa, M. Takada, H. Mizuno, K. Uchiyama, T. Odaka, J. Shirako, M. Mase, K. Kimura, and H. Kasahara. An 8640 mips soc with independent power-off control of 8 cpus and 8 rams by an automatic parallelizing compiler. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 90–598, 2008.
11) ITU-T T.832. Information technology. *jpeg xr image coding system - image coding specification*, 2009.
12) 佐藤卓也, 見神広紀, 林明宏, 間瀬正啓, 木村啓二, 笠原博徳. OSCAR API 標準解釈系を用いた Parallelizable C プログラムの評価. 情報処理学会研究報告, 2011.
13) Tilepro64 processor block diagram. `http://www.tilera.com/products/processors/TILEPRO64`.
14) 木村 啓二, 間瀬 正啓, 笠原 博徳. JISX0180:2011「組込みソフトウェア向けコーディング規約の作成方法」を用いた Parallelizable C の定義. Vol. 2012, No. 22, pp. 1–6, 2012.
15) ITU-T T.832. Information technology. Iso/iec fcd 29199-5: Information technology – jpeg xr image coding system – part 5: Reference software, 2009. `http://www.itscj.ipsj.or.jp/sc29/open/29view/29n10430c.htm`.
16) Media bench 2. `http://euler.slu.edu/~fritts/mediabench/`.