メディアアプリケーションにおけるコンパイラによるI/Oオーバーヘッド隠蔽手法

林　明宏[†1]　関口　威[†1]　間瀬正啓[†1]
和田康孝[†1]　木村啓二[†1]　笠原博徳[†1]

本稿では，相対的に増大するI/Oオーバーヘッドの削減を目指して，連続したファイル入出力を伴うストリーミングデータを扱うメディア処理アプリケーションを対象としたI/Oオーバーヘッド隠蔽手法を提案する．本手法ではI/O処理を並列化コンパイラが生成する通常の粗粒度タスクの1つとして扱い，粗粒度タスク間並列性解析，タスクスケジューリングを行うことでI/Oタスクと演算タスクの並列化を実現する．AACエンコードプログラムを用いて情報家電用マルチコアRP-X及びXeonサーバ上でその性能を評価した結果，提案手法は最大48%の速度向上を実現可能であることが分かった．

Hiding I/O overheads with Parallelizing Compiler for Media Applications

Akihiro Hayashi,[†1] Takeshi Sekiguchi,[†1]
Masayoshi Mase,[†1] Yasutaka Wada,[†1] Keiji Kimura[†1]
and Hironori Kasahara [†1]

In this paper, we propose a novel method which hides I/O overheads in multimedia applications. We propose a compilation technique which realize a I/O task definition, a data dependency analysis among coarse-grain tasks and coarse-grain task scheduling in order to hide I/O overheads for multimedia applications. This paper evaluates processing performance by the proposed methods on RP-X processor and Xeon server. As a result, the proposed method attains speedups to 1.5x for AAC encoding program with 4 SH-4A processors compared with conventional method.

1. はじめに

マルチコアプロセッサの普及は著しく，携帯電話，ゲーム機，自動車，医療機器等の組み込み機器，PC，スーパーコンピュータ等様々な分野で注目を集めており，これまでFRV[1)]，MPcore[2)]，UniPhier[3)]，SPARC 64 VIII fx[4)]，Nehalem[5)]，Phenom[6)]，Power 7[7)]等のホモジニアスマルチコア，Cell BE[8)]やRP-X[9)]等のヘテロジニアスマルチコアが開発されている．

しかしながら，マルチコアでは，プログラムからの並列性抽出，計算資源へのタスクスケジューリング，同期コードやDMAを用いたデータ，電力制御等多くの負担をソフトウェア開発者が負うことになり，プログラム開発コスト及び開発期間が増大する．

この問題を解決するため，筆者らは従来よりOSCAR自動並列化コンパイラ及びAPIを中心としたマルチコア用自動並列化ソフトウェア開発フレームワーク[10)11)12)]を提案しており，ホモジニアスマルチコア及びヘテロジニアスマルチコア上でC言語やFortran言語の自動並列化を実現している．このような自動並列化も含め各種計算処理の並列化に関する取り組みは広く行われている．その一方で，OSCARコンパイラをはじめ，種々の一般的な自動並列化コンパイラは，C言語のfread関数やfwrite関数，Fortranのread文，write文のようなI/O処理に対する依存解析を保守的に行い，これらI/O処理の持つ並列性を積極的に活用することができなかった．しかしながら，一般にプロセッサの動作速度に比べ，ストレージシステムへのアクセスコストは非常に大きく，I/Oボトルネックを引き起こす原因となる[13)]．そのため，I/Oによって生ずるオーバーヘッドを隠蔽することが重要である．

この問題を解決するためには，I/O命令と計算タスクを並列に実行可能な場合に積極的な並列化を自動で行うことが必要となる．そこで本稿では，I/O処理を含めた並列処理の第一歩として，連続したファイル入出力を伴うストリーミングデータを扱うメディア処理アプリケーションを対象としたI/Oオーバーヘッド隠蔽手法を提案する．本手法はOSCARコンパイラの粗粒度タスク並列処理においてI/Oタスクを計算タスクと共に並列性解析し，スケジューリングすることで並列処理によりそのオーバーヘッドを隠蔽することを特徴とする．また，本稿では本手法をAACエンコーダプログラムに適用し，PCサーバ及び情報家電用マルチコアRP-X上で評価した結果についても述べる．

†1 早稲田大学
Waseda University

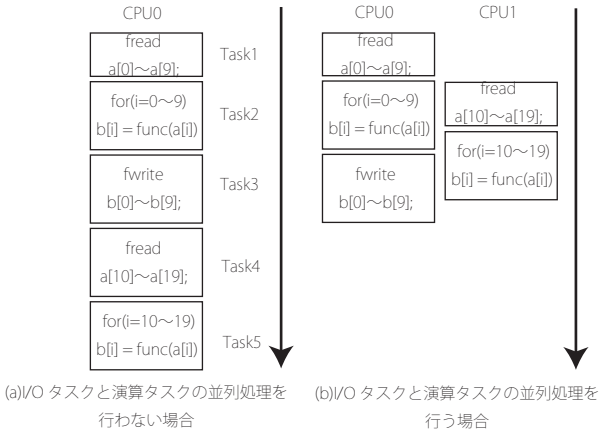
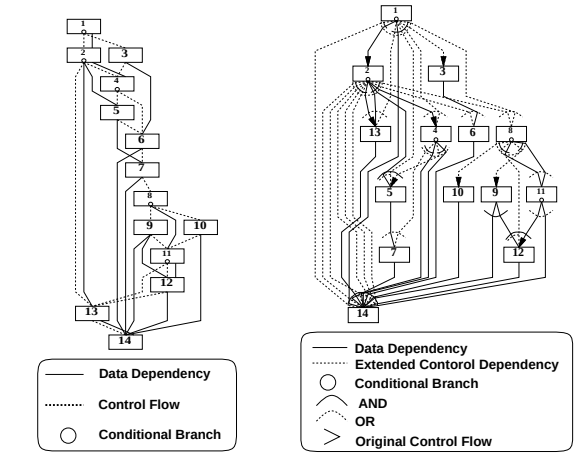


図 1 I/O タスクの並列化例
Fig. 1 Example of Hiding I/O Overheads



(a) Macro Flow Graph (MFG) **(b) Macro Task Graph (MTG)**

図 2 マクロフローグラフ (MFG)，マクロタスクグラフ (MTG) の例
Fig. 2 Example of Macro-Flow Graph and Macro-Task Graph

2. メディア処理における粗粒度 I/O タスクを考慮した並列化

本章では，I/O 命令を考慮した並列化の概要及び，それを実現する OSCAR 並列化コンパイラについて述べる．さらに，本稿で評価したアプリケーションの I/O タスクに，処理コストなどの情報を付与する目的で，一時的に利用した OSCAR コンパイラに対するヒントディレクティブについて説明する．

2.1 I/O タスクの並列化イメージ

本稿では，C 言語の fread や fwrite の様な I/O 命令を対象として並列化方式を説明する．入力と出力でも異なるファイルであれば同時にアクセスすることは可能であり，データ依存あるいはデバイス依存がなければ，データを入力している間に，他のコアで別の処理を実行することは可能である．I/O 命令を考慮した並列化の方針としては，同一ファイルストリーム対する読み込み/書き込みはアクセスするデバイスの衝突が起きるため，プログラムに記述された通りの実行順序を保証するが，I/O 命令によって定義/参照される変数以外の変数に関する計算は，I/O 命令と並列に実行可能であるとすることで処理速度の向上を試みる．図 1 にその例を示す．図 1 において，(a) は I/O タスクと演算の並列処理を行わない例である．この場合，例えば Task2 と Task4 は a と b という異なる配列に対して計算するため，(b) に示す様に並列に実行可能である．このような並列化を実現するために，ここでは OSCAR コンパイラの粗粒度タスク並列化に I/O 粗粒度タスクを組み込む．

2.2 OSCAR コンパイラによる I/O を加えた粗粒度タスク並列処理

OSCAR コンパイラは逐次プログラムを入力とし，(1) 粗粒度タスクの生成及びコストの推定 (2) プロセッサへの粗粒度タスクのスタティックスケジューリング (3) タスク間同期コードを含めた並列化コードの生成を行う．粗粒度タスク並列処理はループ並列処理等他の並列処理方式に比べ，より大きな粒度でプログラムを並列化し，処理性能の向上に効果的であるため，OSCAR 並列化コンパイラでは粗粒度タスク並列処理を適用する．

I/O 並列化の第一段階として，OSCAR コンパイラにおけるソースプログラムの基本ブロックやループやサブルーチンブロック等への分割時に，新たに I/O 粗粒度タスクとして定義する．その後，粗粒度タスクすなわちマクロタスク (MT) 間のコントロールフローとデータ依存関係を表現したマクロフローグラフ (MFG) を生成する．その際，コンパイラはソース中のタスクの入力変数の定義，出力変数の使用を解析することでデータ依存解析を行うが，ソースが解析できない場合，もしくは I/O 処理の様にタスクのコストが推定困難な場合には後述のヒントディレクティブを使用する．さらに MFG から MT 間の並列性を最早実行可能条件解析により引きだした結果をマクロタスクグラフ (MTG) として表現す

る[14)15)]．MFG および MTG の例を図 2 に示す．MFG においてノードは MT を表し，実線エッジはデータ依存を，点線エッジはコントロールフローを表す．また，ノード内の小円は条件分岐を表す．MTG におけるノードも MFG 同様 MT を表し，ノード内の小円は MT 内の条件分岐を表している．また，実線のエッジはデータ依存を表し，点線のエッジは拡張されたコントロール依存を表す．拡張されたコントロール依存とは，通常のコントロール依存だけでなく，データ依存と制御依存を複合的に満足させるため先行ノードが実行されないことを確定する条件分岐を含んでいる．

次に，OSCAR コンパイラは MTG 中の各 MT をチップ上の計算資源の特性を考慮しスケジューリングする[16)]．スケジューリングアルゴリズムはリストスケジューリングにおける優先度及びタスク割り当ての方針をヘテロジニアスマルチコア向けに拡張したものである．最後に OSCAR コンパイラは MT のスケジューリング結果を元に OSCAR API ソースを出力する．

2.3 OSCAR コンパイラへのヒントディレクティブ

```
#pragma oscar_hint accelerator_task (accelerator_type) \
                   cycle (exec_cycle,[trans_mode]) \
                   [workmem(mem_type,mem_size)] \
                   [in(in_list)] [out(out_list)] new-line
```

図 3 OSCAR コンパイラ向けヒント情報
Fig. 3 Hint directive for OSCAR compiler

本節では，OSCAR コンパイラへのヒントディレクティブについて述べる．本稿では，ヒントディレクティブとして OSCAR API におけるヘテロジニアスマルチコアコンパイラ[10)]用に定義されたコンパイラにアクセラレータで処理を行える部分，そのコスト並びに入出力変数を教えるための accelerator_task ディレクティブを用いる．本ディレクティブは前節で述べたように，I/O 処理の様に一般的にコストの推定が困難な場合にも効率的なタスクスケジューリングを実現するため，及び OSCAR コンパイラがソースを解析出来ない場合にタスク間依存解析を実現し，並列性の抽出を容易にするために使用する．本稿では図 3 に示される，accelerator_task ディレクティブを利用し，I/O タスクの並列化に必要なデータ依存解析に必要な情報を与える．本来 accelerator_task はアクセラレータによって実行可能なブロックを指定するヒントディレクティブであり，accelerator_task ディレクティブは引数として，ブロックを実行可能なアクセラレータの種別を accelerator_type に，実行サイクル数を exec_cycle で，アクセラレータへのデータ転送方法 (DMA を使用するか，CPU で行うか等) を trans_mode で，アクセラレータ実行に必要なメモリ種別とそのサイズを mem_type と mem_size で，対象ブロックへの入力となる変数リストを in_list で，出力となる変数リストを out_list で，それぞれ指定することができる[10)]．本稿では特に I/O 処理の実行サイクル数と上記の入出力変数指定を用いて I/O タスクの定義を行う．入出力変数リスト in_list，out_list は複数個のスカラ変数，配列を記述可能である．特に配列に関しては各次元に対して下限 (lower)，上限 (upper)，ストライド間隔 (stride)，連続要素数 (block) を指定可能で，書式は $a[l1:u1:s1:b1][l2:u2:s2:b2]...$ となる．例えば一次元配列 a に対し $a[2:12:3:2]$ という指定を行った場合は，添字が 2 から 12 の範囲で，2 個の連続要素をストライド間隔 3 でアクセスするので $a[2], a[3], a[5], a[6], a[8], a[9], a[11], a[12]$ にアクセスすることとなる．

```
int main() {
  int i, a[N], b[N];
  ...
#pragma oscar_hint accelerator_task (IO_TASK) cycle(100) in(fp) out(a[0:9])
  fread(&a, sizeof(int), 9, fp);
  for (i = 0; i < 9; i++) { b[i] = func(a[i]); }
#pragma oscar_hint accelerator_task (IO_TASK)  cycle(100) in(b[0:9]) out(fp2)
  fwrite(b, sizeof(int), 9, fp2);
#pragma oscar_hint accelerator_task (IO_TASK)  cycle(100) in(fp) out(a[10:19])
  fread(&a, sizeof(int), 9, fp);
  ....
  return 0;
}
```

図 4 OSCAR コンパイラ向けヒントディレクティブ例
Fig. 4 Example of source code with hint directives

サンプルコードを図 4 に示す．サンプルコードは 3 つのヒントディレクティブを含んでいる．1 つ目のディレクティブは「fread 関数の実行に 100 クロックサイクルが必要であり，その実行によって a[0]-a[9] の範囲が定義される」という事を示し，2 つ目のディレクティブは「fwrite 関数の実行に 100 クロックサイクルが必要であり，その実行には b[0]-b[9] の範囲が必要である」という事を示す．同様に最後のディレクティブは「fread 関数の実行に 100 クロックサイクルが必要であり，その実行によって a[10]-a[19] の範囲が定義される」という事を示す．以上の情報により，最後のディレクティブは for ループがアクセスする範囲と

```
for () {
  fread();     // ファイル読み込み
  Encode(); // データ処理
  fwrite();    // データ書き込み
}
```

図 5 一般的なメディア処理のイメージ
Fig. 5 Generic media application

```
for () {
  fread();     // ファイル読み込み ×3
  Encode(); // データ処理 1
  Encode(); // データ処理 2
  Encode(); // データ処理 3
  fwrite();    // データ書き込み ×3
}
```

図 6 粗粒度タスク並列化のためのループアンローリング
Fig. 6 Unrolled version of generic media application

異なる範囲をアクセスすることが分かるため，同時に実行可能と判定可能である．

2.4 ストリーミングデータのメディア処理の並列化イメージ

本節では前節で述べた並列化イメージをベースとし，AAC や MPEG2 エンコーダ等の一般的なメディア処理に対する提案手法の適用について述べる．図 5 に一般的なメディア処理におけるストリーミングデータの処理を示す． 図 5 に示すように，ストリーミングデータのメデイア処理においては，通常ファイルシステムや入力でばいすからのデータ読み込み (fread)，読み込んだデータの処理 (Encode)，処理済みデータのファイルシステムや出力デバイスへの書き出し (fwrite) をループを用いてフレームなどの単位で連続的に行う．このようなプログラムで，粗粒度タスク並列処理を行う為に，まず，図 6 に示す様なループアンローリングを行い，データ処理を行う粗粒度タスクの並列性を高める． しかし，この場合，データ処理を行う粗粒度タスクがコンパイラによって並列に実行されるようにコンパイラがスケジューリングするのみであり，I/O 処理は逐次的に行われるので，I/O を行う粗粒度タスクのオーバーヘッドを隠蔽することは出来ない．そこで，前項で述べた様に，I/O を行う粗粒度タスクのオーバーヘッドを隠蔽するために，ファイル入出力を含めた処理が，図 7 の group1 と group2 の様に 2 回展開されるよう，プログラムのリストラクチャリングを行う．さらに，I/O タスクと演算が並列に実行されるようにコンパイラがスケジューリングし，効率的なストリーム処理を行える可能性を高めている (図 8)．なお，図 8 において

```
for () {
 // group1
  fread();     // ファイル読み込み ×3
  Encode(); // データ処理 1
  Encode(); // データ処理 2
  Encode(); // データ処理 3
  fwrite();    // データ書き込み ×3
 // group2
  read();     // ファイル読み込み ×3
  Encode(); // データ処理 1
  Encode(); // データ処理 2
  Encode(); // データ処理 3
  fwrite();    // データ書き込み ×3
}
```

図 7 I/O タスクを考慮した粗粒度並列化のためのループアンローリング
Fig. 7 Unrolled version of generic media application considering I/O parallelism

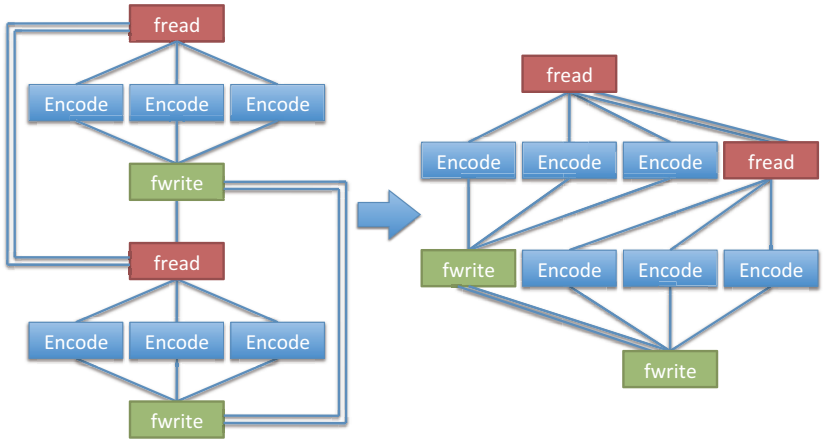


図 8 I/O タスクの並列化を考慮した場合の MTG
Fig. 8 Example of MTG and schdeduled result of I/O parallelization

二重線はデバイス間依存を示している．この例では 2 グループに展開した例を示している．このとき，I/O 処理の並列性抽出とスケジューリングを行うために，必要に応じてヒントディレクティブを利用する．

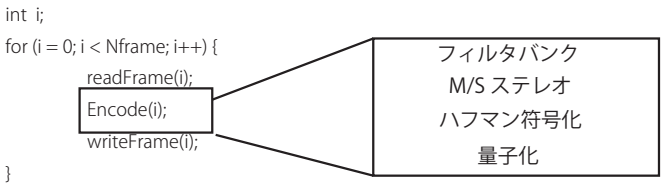


図 9 AAC エンコーダの処理イメージ
Fig. 9 AAC encoding scheme

3. 性能評価

本章では AAC エンコーディングプログラムを例とし提案手法を用いてコンパイルし，性能を評価した結果について述べる．

3.1 評価アプリケーション

AAC エンコーダは，wav 形式の音楽データを AAC 形式に圧縮するプログラムである．本稿で用いる AAC エンコーダはルネサスエレクトロニクスおよび日立製作所提供の AAC-LC エンコードプログラムを Parallelizable C[17] で参照実装したものである．AAC エンコードプログラムの構造を図 9 に示す．図 9 において Nframes はエンコードする wav ファイルのフレーム数を示しており，readFrame 関数で i フレーム目の音声データの読み込み後，Encode 関数の呼び出しにより，1 フレームに対してフィルタバンク処理，MS ステレオ処理，量子化およびハフマン符号化を行い，writeFrame 関数でエンコード結果の書き出しを行う．このようなプログラム構造に対して，2 章で述べた様に，粗粒度タスク並列性の抽出及び，I/O タスク並列性の抽出を目的に手動でリストラクチャリング並びに OSCAR コンパイラへのヒントディレクティブを挿入し I/O タスクと演算タスクの並列化を行った．本評価におけるエンコードパラメータは，サンプリングレートが 44.1[kHz]，ビットレートが 128[kbps] であり，約 19 秒のステレオ PCM データが入力として用いる．

3.2 評価環境

本評価では OSCAR コンパイラによって並列化した AAC エンコーディングプログラムを，Intel の 4 コア CPU である Xeon X5365 を搭載したサーバ，ルネサステクノロジ，日立製作所，早稲田大学で共同開発した情報家電用マルチコア RP-X における SH-4A を 4 コア構成による SMP モードの 2 つの環境で，それぞれ I/O を含めたプログラム実行時間の評価を行った．各環境におけるパラメータを表 1 に示す．

表 1 評価環境
Table 1 Evaluation environment

	Intel Xeon X5365	Renesas/Hitachi/Waseda RP-X
CPU	Xeon X5365 (3.00GHz × 4)	SH-4A (600MHz × 4)
L1 D-Cache	32KB / 1 core	32KB / 1 core
L1 I-Cache	32KB / 1 core	32KB / 1 core
L2 cache	4MB / 1 core	N/A
Native Compiler	GNU C Compiler verion 4.3.2	SH C Compiler + OSCAR API standard Translation
Compile Option	-O3 -fopenmp	

入出力関数に関しては，Xeon サーバー上では C 言語の fread,fwrite 関数を使用した．RP-X 上ではファイルシステムは存在せず，メモリへのリード/ライドとなっているため，現実的なコストを設定するために入力と出力のコストはエンコードと同程度のコストとした．

3.3 Intel Xeon X5365 上での性能評価

図 10 に Xeon X5365 上での性能評価結果を示す．図 10 において横軸はプロセッサ構成で，mCPU は CPU コア m 基を計算資源として使用したことを示す．縦軸は 1CPU での実行時間に対する速度向上率である．図 11 より，2CPU おいて I/O による並列化を行わない場合は 1.45 倍，行った場合は 1.52 倍と 4.8%の性能向上，4CPU において，I/O によるよる並列化を行わない場合は 2.67 倍，行った場合は 2.91 倍となり 8.9%高くなったことがわかる．

3.4 RP-X 上での性能評価

図 11 に RP-X 上での性能評価結果を示す．図 11 において横軸はプロセッサ構成で，mSH は SH-4A コア m 基を計算資源として使用したことを示す．縦軸は 1SH での実行時間に対する速度向上率である．図 11 より，2SH において I/O による並列化を行わない場合は 1.23 倍，行った場合は 1.62 倍と 31.7%の性能向上，4SH において，I/O によるよる並列化を行わない場合は 1.58 倍，行った場合は 2.34 倍となり 48.1%高くなったことがわかる． また，図 12 に 4SH の場合のスケジューリング結果を示す．図 12 は縦方向が時間であり，横方向に各 CPU を表しており，最適にスケジューリングされていることがわかる． また，2 章で述べた I/O タスクのアンローリングを行うことで，処理グループを増やすことでさらなるスケーラビリティの獲得が期待できる．

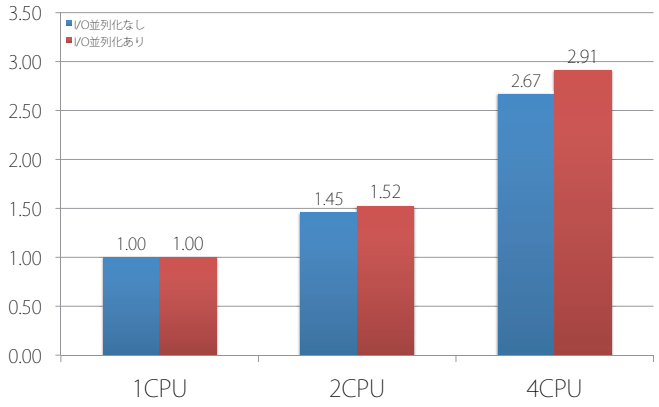


図 10 Xeon X5365 上での性能評価結果
Fig. 10 Evaluation result on Xeon X5365

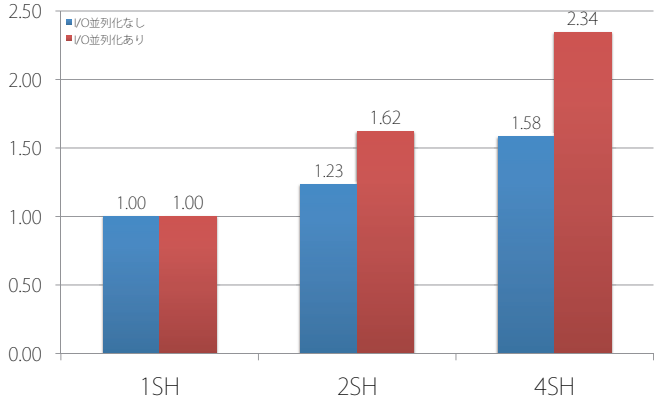


図 11 RP-X 上での性能評価結果
Fig. 11 Evaluation result on RP-X

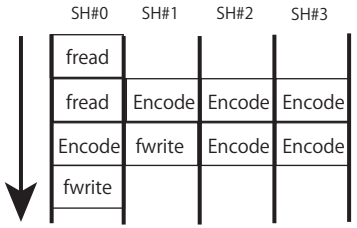


図 12 RP-X の 4SH 向けスケジューリング結果
Fig. 12 Scheduled result for 4SH on RP-X

4. おわりに

本稿では，マルチコアを対象として，入出力命令と計算タスクを並列実行可能となる手法の提案を行った．本手法では，並列化コンパイラが生成する粗粒度タスクグラフ中に，I/O タスクを定義し，スケジューリングを行うことで I/O 部分と計算タスク部分の並列実行を可能とする．AAC エンコーダを用いた評価の結果，Xeon X5365 上で 4 つの CPU を用いた場合，既存の手法と比較して最大 8.9%の性能向上，RP-X 上で 4 つの汎用 CPU を用いた場合，既存の手法と比較して最大 48.1%の性能向上を得ることを確認した．

謝辞 本研究の一部は，経済産業省 “グリーンコンピューティングシステム研究開発” 拠点形成プロジェクト及び NEDO “情報家電用ヘテロジニアスマルチプロセッサ” プロジェクト及び早稲田大学グローバルＣＯＥプログラム「アンビエントＳｏＣ教育研究の国際拠点」（文部科学省研究拠点形成費補助金）の支援により行われた．OSCAR ヘテロジニアス API は早大と日立・ルネサス・富士通・東芝・パナソニック・NEC からなる API 委員会にて策定された．研究を遂行するにあたり，貴重なアドバイスを頂いたプロジェクト関係者の皆様に感謝致します．

参 考 文 献

1) Suga, A. and Matsunami, K.: Introducing the FR 500 embedded microprocessor, *IEEE MICRO*, Vol.20, pp.21–27 (2000).
2) ARM: *ARM11 MPCore Processor Technical Reference Manual* (2005).
3) 木村，藤井，西道，清原：デジタル家電統合プラットフォーム UniPhier におけるメディアプロセッサ，DA シンポジウム (2005).
4) Maruyama, T., Yoshida, T., Kan, R., Yamazaki, I., Yamamura, S., Takahashi, N.,

Hondou, M. and Okano, H.: Sparc64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing, *Micro, IEEE*, Vol.30, No.2, pp.30 –40 (2010).
5) Intel: Intel, http://www.intel.com/.
6) AMD: AMD, http://www.amd.com/.
7) Kalla, R., Sinharoy, B., Starke, W. and Floyd, M.: Power7: IBM's Next-Generation Server Processor, *Micro, IEEE*, Vol. 30, No. 2, pp. 7 –15 (online), DOI:10.1109/MM.2010.38 (2010).
8) Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T. and Yazawa, K.: The design and implementation of a first-generation CELL processor, *2005 IEEE International Solid-State Circuits Conference, ISSCC* (2005).
9) Yuyama, Y., Ito, M., Kiyoshige, Y., Nitta, Y., Matsui, S., Nishii, O., Hasegawa, A., Ishikawa, M., Yamada, T., Miyakoshi, J., Terada, K., Nojiri, T., Satoh, M., Mizuno, H., Uchiyama, K., Wada, Y., Kimura, K., Kasahara, H. and Maejima, H.: A 45nm 37.3GOPS/W heterogeneous multi-core SoC, *IEEE International Solid-State Circuits Conference, ISSCC* (2010).
10) 林, 和田, 渡辺, 関口, 間瀬, 木村, 伊藤, 長谷川, 佐藤, 野尻, 内山, 笠原：情報家電用ヘテロジニアスマルチコア用自動並列化コンパイラフレームワーク，情報処理学会研究報告,Vol.2010-ARC-190, No.7,pp.1-9 (2010).
11) Hayashi, A., Wada, Y., Watanabe, T., Sekiguchi, T., Mase, M., Shirako, J., Kimura, K. and Kasahara, H.: Parallelizing Compiler Framework and API for Power Reduction and Software Productivity of Real-time Heterogeneous Multicores, *Proc of The 23rd International Workship on Languages and Compilers for Parallel Computing(LCPC2010)* (2010).
12) Kimura, K., Mase, M., Mikami, H., Miyamoto, T. and Kasahara, J. S.H.: OSCAR API for Real-time Low-Power Multicores nad Its Performance on Multicores and SMP Servers, *Proc of The 22nd International Workship on Languages and Compilers for Parallel Computing(LCPC2009)* (2009).
13) Lang, S., Carns, P., Latham, R., Ross, R., Harms, K. and Allcock, W.: I/O performance challenges at leadership scale, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09 (2009).
14) 本多, 岩田, 笠原：Fortran プログラム粗粒度タスク間の並列性検出法，信学論 (D-I), Vol.J73-D-I, No.12, pp.951–960 (1990).
15) 笠原, 合田, 吉田, 岡本, 本多：Fortran マクロデータフロー処理のマクロタスク生成手法，信学論, Vol.J75-D-I, No.8, pp.511–525 (1992).
16) 和田, 林, 益浦, 白子, 中野, 鹿野, 木村, 笠原：ヘテロジニアスマルチコア上でのスタティックスケジューリングを用いた MP3 エンコーダの並列化，情報処理学会論文誌コンピューティングシステム (2007).
17) Mase, M., Onozaki, Y., Kimuraa, K. and Kasahara, H.: Parallelizable C and Its Performance on Low Power High Performance Multicore Processors, *Proc. of 15th Workshop on Compilers for Parallel Computing* (2010).