組込みメディア処理向け対称型マルチコアプロセッサにおける階層キャッシュの利用効率を考慮した細粒度スレッドスケジューリング手法

森　達矢[†]　武田　進[††]　松崎秀則[††]

組込みメディア処理向け対称型マルチコアプロセッサにおける階層キャッシュの利用効率を考慮した細粒度スレッドスケジューリング手法を実装した．本手法はアプリケーション開発者がスレッドのコア割り当てを局所化するようスケジューラに指示する API と，指示された情報に基づいてスケジューリングを最適化するためのアルゴリズムで構成されている．本手法を FCFS(first-come first-served) 手法と H.264 デコード処理をベンチマークに用いて比較した結果，最大で約 50%の性能向上を確認した．

A fine-grain thread scheduling method considering efficiency of hierarchical cache on a symmetric multi-core processor for media processing

Tatsuya Mori,[†] Susumu Takeda[††] and Hidenori Matsuzaki[††]

This paper proposes a fine-grain thread scheduling method considering the efficiency of multiple hierarchical cache on a multi-core processor for media processing. The proposed method is composed of APIs providing a framework to specify the order of thread execution intuitively for application programmer and an algorithm to optimize scheduling based on the specified information. Our experiments show the proposed method can improve performance about 50% on H.264 decoder benchmark rather than ordinal FCFS (first-come first-served) method.

1. はじめに

昨今のメディア処理デバイスには，規格が多岐に渡っている映像や音声の再生・記録といった多様かつ性能レンジの幅広い処理をサポートすることが求められており，この要求を満たすためにマルチコアプロセッサの応用が進められている．マルチコアプロセッサは，周波数を抑えつつコア数を増やすことで高性能と低消費電力の両立を目指す手法である．とはいえ，マルチコアプロセッサにおけるソフトウェア開発は性能を引き出すことが一般に困難であるため，コンパイラ・OS・並列処理ライブラリといった開発環境のサポートが不可欠である．同時に，ソフトウェアの開発・運用コストの観点から，ハードウェア構成が異なる場合や同じハードウェア構成であっても使用できるコア数やキャッシュ量といったリソースがソフトウェア実行時に変動する状況においてもプログラムを変更することなく動作できる再利用性と，コア数に比例した性能向上を実現するスケーラビリティの両立が求められている．

これらの目的を実現すべく開発した組込みメディア処理向け対称型マルチコアプロセッサおよび細粒度スレッドによる並列化スキームに対して，本稿では階層キャッシュの利用効率を高める細粒度スレッドスケジューリング手法を考案し，実装した．

以降，本稿が対象とするハードウェアアーキテクチャについて第 2 章で，ソフトウェアアーキテクチャについて第 3 章で説明する．第 4 章で考案した細粒度スレッドスケジューリング手法について述べ，効果を第 5 章の実験で示す．第 6 章で関連研究に触れ，第 7 章でまとめを行う．

2. 対象ハードウェアアーキテクチャ

本稿が対象とするマルチコアプロセッサの諸元を表 1 に示す．このプロセッサは 8 つのコア (media processing engines:MPE) を有し，容量可変の L2 キャッ

† 株式会社東芝セミコンダクター&ストレージ社 半導体研究開発センター
Toshiba Corporation Semiconductor & Storage Products Company, Center for Semiconductor Research & Development
†† 株式会社東芝 研究開発センター
Toshiba Corporation, Research & Development Center

表 1 プロセッサ仕様
Table 1 Processor specification

Technology	65nm CMOS, triple-well, 8-layer-metal	
Die Size	5.06mm x 5.06mm	
Gate Counts	Logic	3.6MGates
	SRAM	5.6Mb
Supply Voltage	2.5V (I/O), 1.2V (Core) 1.2V/0.95V/0V (SVC output)	
Clock Frequency	MPE, L2$ Logic	333MHz (1.2V), 166MHz (0.95V)
	L2$ SRAM, Bus I/F	166MHz (1.2V), 83MHz (0.95V)
L1 Cache	8KB(Instruction)/8KB(Data), 2-way, FIFO, 64ByteLine	
L2 Cache	512KB/256KB/128KB/64KB(unified), 4-way, LRU, 256Byte Line	

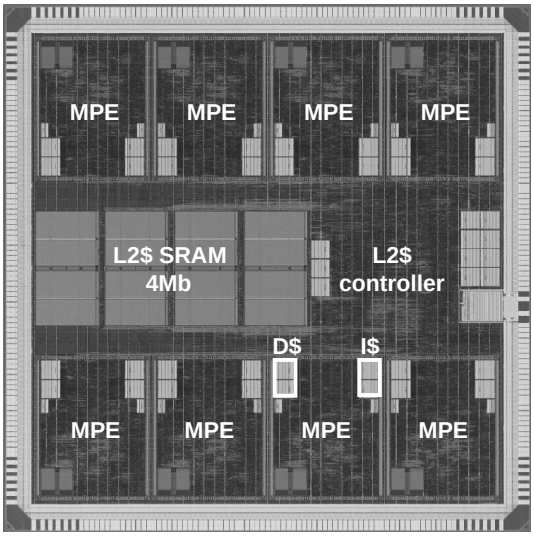


図 1 プロセッサのチップ写真
Fig. 1 Chip micrograph of the processor

シュを共有している．MPE は 32bit 命令長の RISC プロセッサと 64bit 命令長の SIMD 2-way VLIW コプロセッサで構成されている．このコプロセッサにはメディア処理に特化した拡張命令が備わっている[1]．また，MPE は 8KB の L1 命令キャッシュと L1 データキャッシュを有している．65nm CMOS, triple-well, 8-layer metal technology で試作したチップ写真を**図 1** に示す．このプロセッサは，いわゆるごく一般的な対称型マルチコアプロセッサとなっている．なお，より詳細な解説は参考文献 2)，3) を参照願いたい．

3. 対象ソフトウェアアーキテクチャ

スケーラビリティには，プログラムをいかに小さく分割し，効率よくコアへ割り当てるかが重要である．

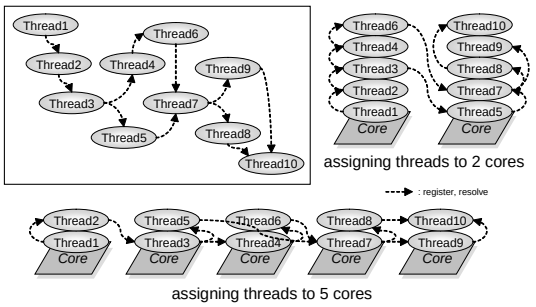


図 2 スレッドのパイプライン並列処理
Fig. 2 Pipeline parallel processing of threads

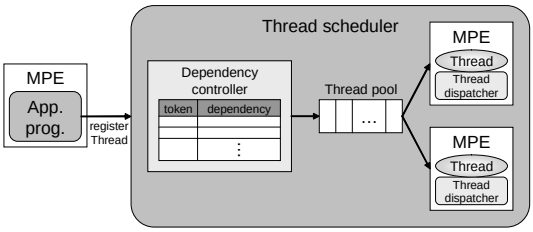


図 3 スレッドスケジューラ
Fig. 3 Thread scheduler

さらに再利用性には，コア数に依存しないソフトウェアであることが求められる．これらの実現を目的とした並列処理スキームと細粒度スレッドスケジューラについて以下に示す．なお，より詳細な解説は参考文献 3)，4) を参照願いたい．

3.1 並列処理スキーム

メディア処理における並列ソフトウェア開発では，パイプライン処理が有効な手段である．本スキームは，**図 2** に示す通りプログラムをスレッドに分割し，スレッド間でデータを受け渡すことでパイプラインを構成するが，コア数の変化に応じてスレッドへの分割とコア割り当てを再設計していてはプログラムの再利用性が得られない．したがって，想定されるコア数に対してなるべく多くの細粒度スレッドに分割してプログラミングしておき，実行時にコア割り当てを決定する on-the-fly スケジューリングを用いる．このとき，プログラマがコア数を意識せず透過的にプログラミングできる API が求められる．なお，以降では特筆しない限り「スレッド」とは「細粒度スレッド」を指すものとする．

3.2 スレッドスケジューラ

スレッドは以下の特徴を持つよう設計されている．

- **non-preemptive**
  スレッドは実行完了までコアを占有する
- **non-blocking**

実行中のスレッドには同期待ちが起きない

- **non-mutex(mutual exclusion)**
  スレッドどうしで排他制御を行わない

この設計に基づいて実装されたスケジューラを**図 3** に示す．登録されたスレッドはスケジューラの *Dependency controller* に収められ，依存解決されるのを待つ．依存数はスレッド登録 API の引数で指定し，依存解決 API で解決先のスレッドを引数で指定する．

- register(token, count, entry, arg)
- resolve(token)

依存がすべて解決されたスレッドは *Thread pool* に移され，実行待機状態となる．すべてのコアには *Thread dispatcher* が常駐し，自コアが空き状態になると *Thread pool* からスレッドを取得する．この *Thread pool* は，FCFS(first-come first-served) のキュー構造になっている．

以上の設計により，スレッドのコンテキストスイッチが不要となり，スケジューリングコストを最小限に抑えている．また，プログラマにとってスレッドの割り当て先であるコアがスケジューラで隠蔽されており，コア数を意識する必要のない透過的なプログラミングが可能となっている．

4. スケジューラの拡張

プログラム全体の性能向上を図るためにスケジューラに求められるのは，

- スレッド間の高い並列度を保つこと
- スケジューリングコストが小さいこと

であり，3.2 節のスケジューラ (以降，既存スケジューラと呼ぶ) はどちらも満たしている．加えて，階層キャッシュを有する対称型マルチコアプロセッサにおいては

- キャッシュ利用効率の向上

も重要であるが，既存スケジューラでは考慮しきれていなかった．細粒度スレッドの場合，FCFS より LCFS(last-come first-served) の方がキャッシュ利用効率が高いことが知られているが，LCFS は FCFS に比べて並列度が低下しやすいという問題がある．3 つのコアと 24 個の実行時間の等しいスレッドを仮定し，FCFS と LCFS でそれぞれスケジューリングした単純なモデルを**図 4** に示す．図 4 のノードはスレッド，ノード中の数字はスレッドの token を表し，エッジはスレッドの依存関係を表している．LCFS は FCFS に比べて実行時間がスレッド 2 つ分長くかかってしまっている．このモデルは最大並列度 4 に対してコア数 3 であるため，LCFS ではスレッド *19* が後回しにされて依存する後続のスレッド *20, 21, 22, 23, 24* が処理

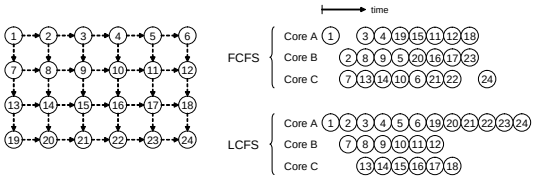


図 4 FCFS と LCFS の並列度
Fig. 4 Concurrency of FCFS and LCFS

される終盤の並列度が低下したことが原因である．また，もしコア数 4 であったならば並列度の低下は顕在化しないため，プログラミング時の性能最適化で見逃す危険性もある．このような見逃しを運用後に発見することは容易ではなく，つまり性能最適化という側面で再利用性が損なわれていることになる．図 4 のモデルはスレッド数が少ないため並列度低下と見逃しの危険性はまだ小さいが，後述する第 5 章の実験で使用する規模のベンチマークではスレッド数が 1 万を超えるため実用は難しい．

とはいえ，LCFS はメモリ参照の時間的局所性を活かすというキャッシュそもそもの理にかなっている点は捨てがたい．そこで，既存スケジューラに LCFS の利点を加えるべく拡張を施す．階層キャッシュを有する対称型マルチコアプロセッサのメモリ参照とキャッシュ利用効率について着目すると，同じ関数を呼び出すスレッドどうしが同じコアで実行されれば L1 命令キャッシュの利用効率が高まる．また，あるスレッドどうしでデータの受け渡しがある場合，同じコアで近い時間に実行されれば L1 データキャッシュの利用効率が高まる．またこのとき，受け渡されるデータが L1 データキャッシュを超えるサイズであっても，コアを問わず近い時間に実行されれば L2 キャッシュの利用効率を高めることができる．このようにキャッシュ利用効率の向上には，スケジューラに

コア割り当ての空間的局所化 ある複数のスレッドを同じコアへ割り当てる

コア割り当ての時間的局所化 ある複数のスレッドを近い時間でコアへ割り当てる

の機能を加えて，スレッドどうしでメモリ参照の時間的局所性を高めることが有効な手段と言える．この 2 つの局所化を実現するために，スケジューラの *Thread pool* 構造，スレッド操作 API，*Thread dispatcher* アルゴリズムを拡張する．このとき，*Thread dispatcher* アルゴリズムの変更によってプログラムの並列度を低下させないよう注意しなければならない．

4.1 *Thread pool* の拡張

図 2 に示した既存スケジューラの *Thread pool* を

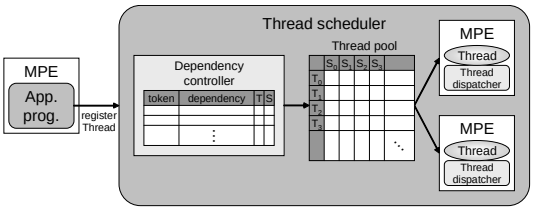


図 5 拡張スケジューラ
Fig. 5 Extended scheduler

図 5 のように拡張する．時間的局所化するスレッドグループを T，空間的局所化するスレッドグループを S で表している．$T \times S$ のマスは，*Ready* 状態のスレッドを保持しておく Ready queue をそれぞれ 1 つずつ有する．次節 4.2 で詳しく述べるが，この T と S に基づいて Ready queue をコアへ割り当てることで 2 つの局所化を実現する．なお，T と S の指定方法については，スレッド登録 API に T と S を意味する引数を 1 つずつ追加することで実現した．

- register(token, count, entry, arg, T, S)

なお，resolve() に変更はない．

4.2 *Thread dispatcher* アルゴリズムの拡張

図 5 の *Thread pool* に対して，各 S グループは 1 つのコアに関連付けされ，コアは関連付けされた S グループに含まれるスレッドを優先的に実行することで空間的局所性を実現する．また，最大 1 つの T グループが実行状態になり，この実行状態にある T グループに含まれるスレッドを優先的に実行することで時間的局所性を実現する．なお，S_0 と T_0 は局所化の対象外として扱う例外グループとする．

Ready 状態に遷移したスレッドは，登録時に API で指定された S グループと T グループに対応する Ready queue に enqueue され，*Thread dispatcher* によってコアへ割り当てられるのを待つ．各コアは，スレッドの実行が終了すると次に実行するスレッドを割り当ててもらうために *Thread dispatcher* を呼び出す．このとき，そのコアに関連付けされている S グループの集合を $\mathbf{S}$，実行状態の T グループを T' とすると，*Thread dispatcher* の拡張アルゴリズムは以下のようになる．

```
1  HEAD :
2      if T' ∩ S ≠ ∅ then
3          target ← (T', S'(∈ S));
4      else if T' ∩ S_0 ≠ ∅ then
5          target ← (T', S_0);
6      else if T' ∩ S̄ ≠ ∅ then
7          target ← (T', S'(∈ S̄));
8          S ← S ∪ {S'};
9      else if T̄' ≠ ∅ then
10         T' ← select(T̄');
11         goto HEAD;
12     else if T_0 ∩ S ≠ ∅ then
13         target ← (T_0, S'(∈ S));
14     else if T_0 ∩ S_0 ≠ ∅ then
15         target ← (T_0, S_0);
16     else if T_0 ∩ S̄ ≠ ∅ then
17         target ← (T_0, S'(∈ S̄));
18         S ← S ∪ {S'};
19     else
20         halt;
21     endif
22     dispatch dequeue(target);
```

8 行目と 18 行目においては当該 S グループに関連付けされているコアを変更し，10 行目においては実行状態の T グループを変更している．

以上の拡張により，同一の S グループに属するスレッドどうしでは L1 命令キャッシュの利用効率が高まり，同一の T グループに属するスレッドどうしでは L2 キャッシュの利用効率が高まり，同一の S グループかつ同一の T グループに属するスレッドどうしでは L1 データキャッシュの利用効率が高まる．また，T を適切に指定することで並列度の低下を防ぐことも可能となる．

5. 実　　験

H.264 デコード処理をベンチマークに用いて既存スケジューラと拡張スケジューラを比較する．

5.1 ハードウェア環境

本実験では，図 1 の LSI チップを搭載した評価ボードを使用した[3]．この評価ボードは，**図 6** に示す通り FPGA，バスブリッジ，128MB DDR-SDRAM，および NTSC/HD 入出力インタフェース，オーディオインタフェースや USB コントローラといった組込みマルチメディアアプリケーションで使用される標準的な周辺回路を含んでいる．

また，スケジューリングコストの詳細な測定にあたっては，サイクル精度のシミュレータを用いた．このシミュレータにおける L1 キャッシュのミスペナルティは一律 10 サイクル，L2 キャッシュのミスペナルティは一律 200 サイクルと設定した．この設定値は，図 1 の

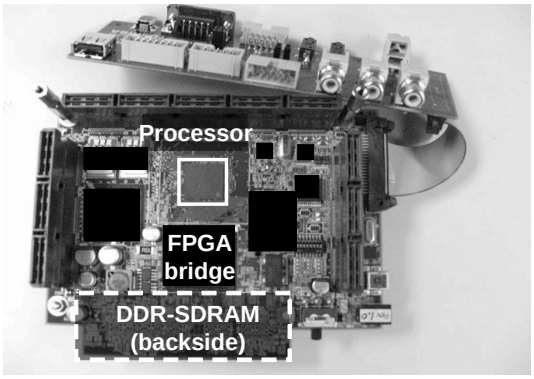


図 6 評価ボード写真
Fig. 6 Evaluation board micrograph

LSI チップにおける各キャッシュやメモリバスでアービトレーションが発生しなかった場合の理想値から算出している．

5.2 ベンチマーク

昨今の組込みメディア処理において最も負荷の大きい処理の 1 つである H.264 デコード処理をベンチマークに用いた．解像度の異なる入力ストリームを使用することで負荷の変動を実現し，同時に再利用性の確認も行った．なお，映像再生にはデコード処理以外にもストリームデータの入力や出力画像のレンダリングといった様々な処理が必要であるが，これらは並列化の対象外とした．

図 7 に示す通り，H.264 デコード処理はおおまかに 3 つのブロックに分割される．本実験では，中段の Video signal processing に対して並列化を行った．Video signal processing は 16x16 ピクセルのマクロブロック単位に処理を分割でき，さらに**図 8** に示す通り 8 つのステージに分割することができる．この図 8 は，8 つのステージがパイプライン処理される様子を表している．ステージごとに並列化したスレッドどうしの同期関係を**図 9** に図示する．青色の矢印はステージ内の同期を表し，赤色の矢印は同マクロブロックにおけるステージ間の同期を表し，緑色の矢印は異なるマクロブロックにおけるステージ間の同期を表している．

5.3 実 験 内 容

8 コアのうち最大 6 コアを Video signal processing で使用し，2 コアを Coded stream processing と Frame output controlling およびその他すべての処理で使用した．評価ボードでは，Video signal processing で使用するコア数を 1〜6 個に切り替え，L2 キャッシュは表 1 の諸元に示した 4 つのサイズを切り替えて実験を

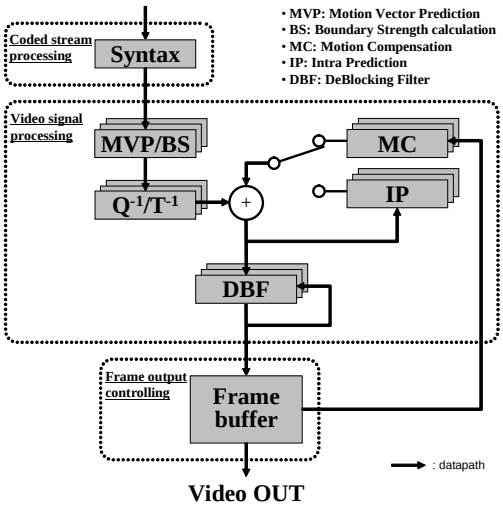


図 7 H.264 デコード処理のシステムブロック図
Fig. 7 System block diagram of H.264 Decoder

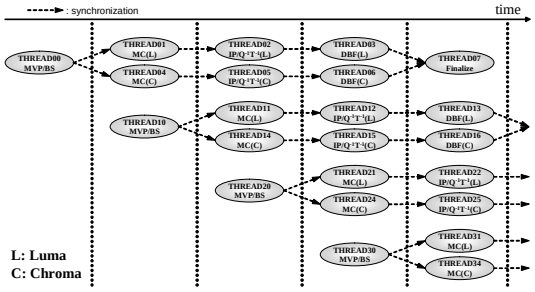


図 8 H.264 デコード処理の Video signal processing におけるマクロブロック処理の関数分割
Fig. 8 Divided functions per macroblock for video signal processing of H.264 decoder

行った．入力には解像度 QVGA/VGA/WVGA/720p の画像をそれぞれ Baseline profile の量子化値 40 でエンコードしたエレメンタリストリームを用い，連続するピクチャ128 枚分を測定の対象とした．既存スケジューラに対しては以上の条件で実験を行い，拡張スケジューラに対しては加えて Video signal processing のスレッドに対して T と S のグループ分けを 2 パターン行った．まず **A パターン**では，S について図 8 のステージごとにグループを分けた (**図 10** 参照)．T については図 8 の Thread0,1,2,4,5 と Thread3,6,7 で前後段に分け，前後段それぞれマクロブロック 2 行ごとにグループを分けた (**図 11** 参照)．後段は，マクロブロック 1 行上の前段に依存している (図 9 における緑色の同期) ため，前段よりも T グループを 1 つ上にずらしている．次に **B パターン**では，A パターンと

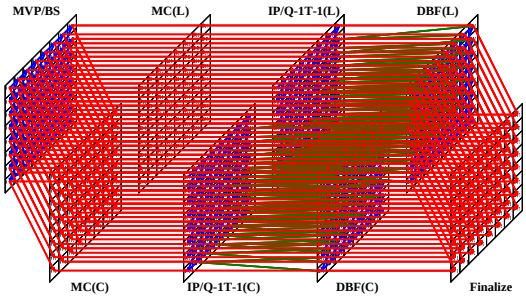


図 9 H.264 デコード処理 1 フレーム内におけるスレッドの同期関係
Fig. 9 Thread synchronization in a frame of H.264 Decoder

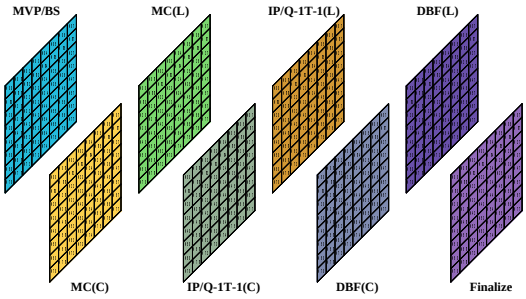


図 10 A パターンおよび B パターンの S グループ分け
Fig. 10 S grouping of Pattern A and Pattern B

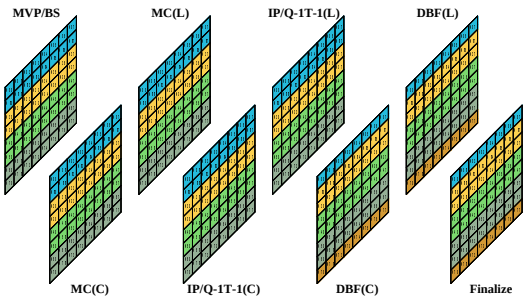


図 11 A パターンの T グループ分け
Fig. 11 T grouping of Pattern A

同様に S をグループ分けし，T についてはフレーム内すべての Thread0,1,2,4,5 を 1 つのグループとし，同様に Thread3,6,7 も 1 つのグループとした．B パターンは，A パターンに比べてフレーム処理全体の並列度低下を防ぐ狙いもある．このように T と S をグループ分けすることで，同じ関数が同一のコアで実行されて L1 命令キャッシュの利用効率が高まり，横並びのマクロブロックを処理する同じ関数どうしで L1 データキャッシュの利用効率が高まり，横並びのマクロブロックを処理する前後段それぞれの異なる関数どうしで L2 キャッシュの利用効率が高まると期待できる．まとめると，実験で用いたパラメータは以下の 4 つである．

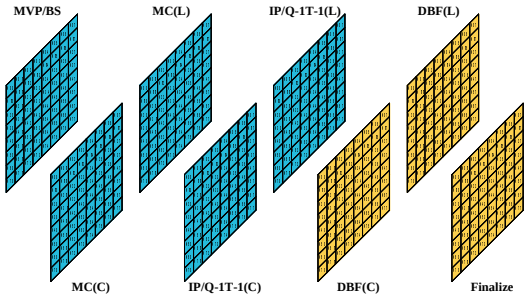


図 12 B パターンの T グループ分け
Fig. 12 T grouping of Pattern B

スケジューラ：
既存スケジューラ, 拡張スケジューラ (A パターン, B パターン)
スレッドを処理するコア数： 1, 2, 3, 4, 5, 6
L2 キャッシュサイズ：
64KB, 128KB, 256KB, 512KB
入力ストリーム： QVGA, VGA, WVGA, 720p
また，測定項目は以下の 4 つとした．

- **Frames per second(FPS)**
- **スレッドの平均実行サイクル数 (CPT)**
- **スレッドによる L1 キャッシュミス回数の総和**
- **スレッドによる L2 キャッシュミス回数の総和**

なお，図 1 のチップには L1 キャッシュミスの測定機能に L1 命令キャッシュと L1 データキャッシュどちらのキャッシュミスなのか分解能がないため，やむなく和で評価した．

シミュレータでは，既存スケジューラと拡張スケジューラの A パターンに対してコア数 4，L2 キャッシュ128KB，入力ストリーム WVGA 1 フレーム分を測定した．測定項目は，結果とともに後述する．

5.4 実 験 結 果

WVGA の入力ストリームに対し，コア数と L2 キャッシュサイズをパラメータとした結果と，L2 キャッシュ 128KB に対し，コア数と入力ストリームをパラメータとした結果を**表 2** にまとめる．

WVGA の入力ストリームに対し，コア数と L2 キャッシュサイズをパラメータとした結果を比較すると，コア数が 1 つの場合は L2 キャッシュサイズにかかわらずどのスケジューリングにおいても FPS の差は小さい．しかし，コア数が増えるにつれ L2 キャッシュサイズの影響が顕著になり，コアで共有するデータを十分に

表 2 実験結果のまとめ
Table 2 Summary of the experimental result

			FCFS				Pattern A				Pattern B			
		MPE	FPS	CPT	L1$ miss	L2$ miss	FPS	CPT	L1$ miss	L2$ miss	FPS	CPT	L1$ miss	L2$ miss
Input stream	L2$ size	1	8.96	11432.28	202380694	18908267	11.27	9058.92	186932650	11447960	11.40	8930.19	187006905	11010550
wvga	64KB	2	11.86	17662.46	208791193	23634770	16.46	12628.81	189336475	15135496	16.94	12414.11	187144588	14221218
		3	12.04	26544.77	213543741	25369559	16.41	19252.99	193712583	17760270	17.29	18432.15	191940692	16754372
		4	11.75	35221.48	219800782	26063110	14.72	25051.94	209725380	19786205	15.50	23853.64	207038184	18668324
		5	10.73	43927.36	239086499	27579473	11.30	36225.41	249975709	24061642	11.51	33645.25	245009832	23684383
		6	9.87	56364.11	308228659	28605721	10.40	45801.08	328496069	27013676	10.36	44731.96	315742808	27215305
	128KB	1	11.01	9348.09	201981163	11873602	12.72	8029.05	187115160	7240078	12.90	7898.94	187204266	6795892
		2	18.34	11404.51	208388930	13052655	22.78	9028.97	189759152	8252357	23.52	8850.74	187503322	7711335
		3	21.38	14586.01	213578452	13798789	28.53	11189.38	193972963	9192671	30.00	10447.68	191591462	8517484
		4	22.10	18784.69	220160535	14040204	30.87	13469.95	205951358	9680135	33.09	12369.85	199452417	8864192
		5	21.53	23084.55	242562103	14170384	27.41	17312.32	229113841	10467851	31.06	16890.14	221451939	9469360
		6	20.09	28032.55	262301108	14991914	23.56	22282.03	296858973	11674528	23.68	20137.81	290994766	11658786
	256KB	1	12.93	7987.10	200392166	6275024	13.67	7484.49	187051414	5021456	13.76	7371.09	187279234	4756698
		2	24.06	8642.17	207229891	6649729	26.08	7841.68	189455459	5110345	26.55	7732.95	187718500	4973244
		3	32.61	9647.28	211817447	6909515	36.46	8524.30	194315622	5280408	37.23	8188.53	191215668	5073921
		4	38.60	10797.06	219560323	6826370	44.21	9361.10	201080104	5398107	45.96	9030.07	197090674	5071532
		5	42.00	12418.55	230478923	6814082	48.56	10736.47	214228170	5499928	52.25	9929.89	209561613	4920724
		6	42.58	14337.31	245905515	6910832	51.62	12313.18	228306186	5516504	54.89	11860.27	223406997	5048111
	512KB	1	14.00	7341.81	198735345	3577896	14.31	7134.23	186579258	3684270	14.14	7152.36	187278380	3983560
		2	26.98	7589.02	206632438	3710955	27.90	7373.06	189038945	3499339	27.48	7310.86	187320672	3969347
		3	38.79	7921.62	211653155	3757592	40.51	7558.69	194423160	3491580	39.97	7575.64	191242264	3881273
		4	48.65	8350.75	219645485	3699064	50.94	8065.90	202063046	3498897	50.90	7816.61	197993347	3704091
		5	57.24	8893.61	232894867	3512014	59.55	8442.48	215853142	3402837	59.87	8476.58	211387305	3535532
		6	63.59	9074.69	249186278	3330826	67.69	8558.15	229152643	3190093	67.09	8789.59	225814615	3316534
L2$ size	Input stream	1	78.59	5896.91	32596516	1145296	80.52	5661.28	32301042	1013186	81.04	5631.29	32300745	965412
128KB	qvga	2	131.04	7110.17	34240065	1348580	140.87	6495.25	33732563	1155888	142.12	6556.34	33519977	1142234
		3	158.66	8577.94	36917051	1412540	178.49	7769.57	35375888	1223167	181.60	7695.31	34814386	1183728
		4	171.70	10688.28	40213378	1469320	196.10	9728.78	37769820	1288504	202.52	9014.09	36757382	1226677
		5	167.87	10303.61	44785596	1535924	193.17	9703.43	41765248	1349893	200.09	9240.50	40263047	1287748
		6	167.94	12467.28	49914504	1595355	188.91	10927.10	45713304	1433022	198.63	10769.40	43796412	1355695
	vga	1	17.44	6982.75	132345970	6950709	19.29	6310.47	123565292	4956049	19.31	6287.78	124609978	4858545
		2	29.80	8293.71	135697273	7375402	34.79	7095.34	125854040	5411276	34.94	7461.07	126363134	5415610
		3	34.98	10488.73	140490469	7959755	44.18	8343.99	129501373	5841757	44.74	8328.69	128134041	5776614
		4	37.31	12975.98	147202464	8138115	47.90	10560.49	135044116	6195903	48.91	10681.72	132599659	6036562
		5	36.25	16134.59	156903172	8506267	47.83	13718.91	145648959	6491379	49.54	13264.04	141221890	6297592
		6	32.56	20725.49	171595406	9155453	44.10	17222.07	155510611	7138396	47.70	17345.26	150321696	6731497
	wvga	1	11.01	9348.09	201981163	11873602	12.72	8029.05	187115160	7240078	12.90	7898.94	187204266	6795892
		2	18.34	11404.51	208388930	13052655	22.78	9028.97	189759152	8252357	23.52	8850.74	187503322	7711335
		3	21.38	14586.01	213578452	13798789	28.53	11189.38	193972963	9192671	30.00	10447.68	191591462	8517484
		4	22.10	18784.69	220160535	14040204	30.87	13469.95	205951358	9680135	33.09	12369.85	199452417	8864192
		5	21.53	23084.55	242562103	14170384	27.41	17312.32	229113841	10467851	31.06	16890.14	221451939	9469360
		6	20.09	28032.55	262301108	14991914	23.56	22282.03	296858973	11674528	23.68	20137.81	290994766	11658786
	720p	1	5.35	8022.73	402557427	26305683	6.37	6824.04	359040734	16310934	6.42	6774.45	355414085	15939992
		2	8.90	9704.08	409108645	27957131	11.40	7697.19	366558754	17749615	11.36	7754.11	361183751	17798584
		3	10.01	13214.88	416597695	30135542	13.83	9788.32	373669097	19837618	13.66	9739.75	370249257	19916610
		4	10.32	17274.40	428028928	30997793	14.74	12161.41	385994351	20850015	14.54	12591.20	381022558	20805390
		5	10.12	21409.77	441588339	31153860	14.20	15825.37	403730151	21615026	14.45	15083.15	394441092	21254250
		6	9.77	25925.98	464296278	31610936	13.62	20346.70	424670806	22663625	14.16	19475.80	411980637	21956036

許容できる L2 キャッシュサイズがなければスケーラビリティを実現できないことが分かる．いずれのスケジューリングにおいても L2 キャッシュサイズが小さいほどスケーラビリティの限界に達するコア数が小さいが，拡張スケジューラは既存スケジューラに比べてスケーラビリティの限界に達するコア数が大きい．L2 キャッシュサイズ 128KB に着目すると，コア数 4 において既存スケジューラの 22.10FPS に対し，A パターンは 30.87FPS，B パターンが 33.09FPS と 50%近く上回っている．また，L2 キャッシュサイズが小さいほどコア数が増えるにつれ CPT が増大している．これは，キャッシュミス回数の増加が影響したためである．既存スケジューラと拡張スケジューラを比較すると，拡張スケジューラはコア数が増えてもキャッシュミス回数の増加が小さく，特に L2 キャッシュサイズが小さい場合の L2 キャッシュミス回数が既存スケジューラの 2/3 程度に抑えられている．この L2 キャッシュミス回数の差が FPS のスケーラビリティに大きく影響を与えている．

次に，L2 キャッシュ128KB に対し，コア数と入力ストリームをパラメータとした結果を比較すると，いずれのスケジューリングにおいても画面サイズにかかわらずコア数 4 までしか FPS が向上していないことが分かる．ただし，拡張スケジューラでは既存スケジューラに比べて 4 コアにおける最大 FPS が 20%ほど上回っている．スケジューリングごとの CPT を見比べると，拡張スケジューラは既存スケジューラに比べてコア数の増加による CPT の増大が小さく，特に画面サイズが大きい場合にその効果が顕著に表れている．これは，拡張スケジューラは既存スケジューラに比べてキャッシュミス回数が著しく少なく，特に B パターンの L2 キャッシュミス回数は場合によって既存スケジューラの半分近くにまで抑えられているためである．これらの効果を裏付けるために，専用のトレースツールを用いてスケジューリング結果を可視化したグラフを A.1 の**図 13** に示す．

このように，拡張スケジューラは既存スケジューラに比べてキャッシュミス回数の削減により CPT が減り，FPS が向上することを確認できた．

最後に，シミュレータによる測定結果を**表 3** にまとめる．平均スケジューリングサイクル数を比較すると，拡張スケジューラは既存スケジューラに比べて 15.08%増加している．対平均スレッド実行サイクル数比で比較すると，既存スケジューラ 7.61%に対して拡張スケジューラ 10.21%と，より顕著である．しかし評価ボードの結果でも述べたように L2 キャッシュミス回数の差が性能に大きく影響を与えるため，この程度のスケジューリングオーバヘッドを犠牲にしてでもキャッシュ利用効率を高めた方が効果が大きいと言える．なお，総実行サイクル数で比較すると，拡張スケジューラは既存スケジューラに比べて 10.96%の性能向上を示した．同条件での評価ボード結果である 39.68%の性能向上と大きくかけ離れているが，これはシミュレータのキャッシュミスペナルティーが均一であるためキャッシュミス回数の多い既存スケジューラでの処理時間が短縮されてしまったためである．

表 3 シミュレーション結果
Table 3 Performance summary

評価項目	FCFS	Pattern A
総実行サイクル数	14,794,509	13,333,567
平均スレッド実行サイクル数	20,716	17,784
平均スケジューリングサイクル数	1,578	1,816
スレッド処理コアの稼働率	84.01%	80.02%
L1 命令キャッシュミス率	1.38%	0.84%
L1 データキャッシュミス率	1.86%	1.83%
L2 キャッシュミス率	6.35%	5.87%

6. 関 連 研 究

汎用マルチコアプロセッサでは，スレッドの並列実行を可能にするために Intel 社の Hyper-Threading[5] や Linux の Order One Scheduler[6]，Completely Fair Scheduler[7] といった手法が用いられている．これらの手法はコアの稼働率向上やリアルタイム性を重視しているがために，場合によってはキャッシュの利用効率が低下してしまいプロセッサ全体での性能低下をまねくことがある．

組込み向けマルチコアプロセッサである Cell Broadband Engine では，メモリバンド幅を効果的に利用するスケジューリング手法が用いられている[8]．この手法は各コアのローカルストレージと共有メモリの間で行われるデータ転送のタイミングをスレッドごとに分散させることでメモリバンド幅が不足することを回避しているが，スレッドどうしのデータ共有にローカルストレージを利用することは考慮されていない．

Snavely らの Symbiotic Jobscheduling では，IPC や演算リソースの競合といった統計情報のサンプリングを行い，この統計情報を基にコンテキストスイッチの際にスレッドの組み合わせを最適化することで性能向上を図っている[9]．また，大河原らの SMT(Simulataneous Multi-threading) プロセッサ向けジョブスケジューリング方式では，一定周期ごとにサンプリングを行って最適なスレッドの組み合わせを探索し，その探索結果に従ってスケジューリングを決定している[10]．しかし，

細粒度スレッドでは入力データがスレッド負荷に及ぼす影響が大きく，サンプリング結果が実態と大きく異なる場合が多々あるため，これら方式の適応は難しい．

内倉らの SMT 向けスレッドスケジューラでは，スレッドどうしのキャッシュアクセスを常に監視しスケジューリングに反映させることでキャッシュ利用効率を高めている[11)]．しかし，監視対象を L1 キャッシュに限定して L2 キャッシュを考慮しておらず，大量のデータを扱うメディア処理への適応が難しい．また，スレッドの数が膨大になる傾向のある細粒度スレッドへの適用も困難である．例えば，本実験の入力ストリーム 720p における 1 フレームあたりのスレッド数は 28,800 である．

7. おわりに

本稿では，階層キャッシュを有する組込みメディア処理向け対称型マルチコアプロセッサにおいて，スレッドのコア割り当てを空間局所化と時間局所化することでキャッシュ利用効率を高めるよう FCFS スケジューラの拡張を行い，実験によってその有効性を確認した．H.264 デコード処理をベンチマークに用いて FCFS スケジューラと比較した結果，最大で約 50%の性能向上を確認した．

しかし，本実験ではスレッドの局所化がプログラム全体の性能スケーラビリティにどのように影響するか定量化まではできていない．一般に，コア数に依存せず性能がスケールするよう並列化を行うことは非常に困難である．拡張スケジューラにおいて，スレッドの局所化とスケーラビリティの関係を定式化できれば並列化の困難さが一層軽減されると期待できる．さらに，この定式化をスレッドの実行中に on-the-fly で適用できればスレッドの局所化を人手で行う手間がなくなることが期待でき，今後の課題である．

参 考 文 献

1) Tanabe, J., Taniguchi, Y., Miyamori, T., Miyamoto, Y., Takeda, H., Tarui, M., Nakayama, H., Takeda, N., Maeda, K. and Matsui, M.: Visconti: multi-VLIW image recognition processor based on configurable processor [obstacle detection applications], *Custom Integrated Circuits Conference, 2003. Proceedings of the IEEE 2003*, pp. 185–188 (2003).

2) Nomura, S., Tachibana, F., Fujita, T., Teh, C. K., Usui, H., Yamane, F., Miyamoto, Y., Kumtornkittikul, C., Hara, H., Yamashita, T., Tanabe, J., Uchiyama, M., Tsuboi, Y., Miyamori, T., Kitahara, T., Sato, H., Homma, Y., Matsumoto, S., Seki, K., Watanabe, Y., Hamada, M. and Takahashi, M.: A 9.7mW AAC-Decoding, 620mW H.264 720p 60fps Decoding, 8-Core Media Processor with Embedded Forward-Body-Biasing and Power-Gating Circuit in 65nm CMOS Technology, *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pp. 262–612 (2008).

3) Mori, T., Ueda, Y., Nonogaki, N., Terazawa, T., Sroka, M., Fujita, T., Kodaka, T., Morita, K., Arakida, H., Miura, T., Okuda, Y., Kizu, T. and Tsuboi, Y.: A Power, Performance Scalable Eight-Cores Media Processor for Mobile Multimedia Applications, *Journal of Solid-State Circuits, IEEE*, Vol. 44, No. 11, pp. 2957 – 2965 (2009).

4) Kodaka, T., Sasaki, S., Tokuyoshi, T., Ohyama, R., Nonogaki, N., Kitayama, K., Mori, T., Ueda, Y., Arakida, H., Okuda, Y., Kizu, T., Tsuboi, Y. and Matsumoto, N.: Design and implementation of scalable, transparent threads for multi-core media processor, pp. 1035–1039 (2009).

5) Koufaty, D. and Marr, D.: Hyperthreading technology in the netburst microarchitecture, *Micro, IEEE*, Vol. 23, No. 2, pp. 56–65 (2003).

6) Aas, J.: Understanding the Linux 2.6.8.1 CPU Scheduler, *SGI*, Vol. 22, p. 05 (2005).

7) Wong, C. S., Tan, I., Kumari, R. D. and Wey, F.: Towards achieving fairness in the Linux scheduler, *SIGOPS Oper. Syst. Rev.*, Vol. 42, No. 5, pp. 34–43 (2008).

8) Maeda, S., Asano, S., Shimada, T., Awazu, K. and Tago, H.: A real-time software platform for the Cell processor, *Micro, IEEE*, Vol. 25, No. 5, pp. 20–29 (2005).

9) Snavely, A., Tullsen, D. M. and Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor, *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, ACM, pp. 66–76 (2002).

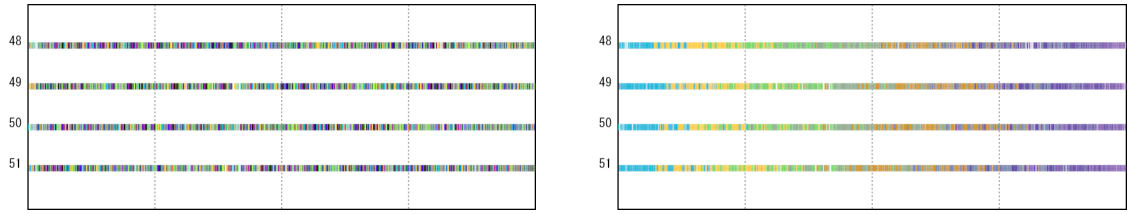
10) 英喜大河原, 彰安里: 最適実行多重度に基づく SMT プロセッサのジョブスケジューリング方式, 技術報告 112(2002-ARC-150), （株）富士通研究所, （株）富士通研究所 (2002).

11) 要内倉, 耕一笹田, 佐藤未来子, 加藤義人, 仁典大和, 中條拓伯, 美太郎並木: 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（ACS）, Vol. 46, No. SIG12(ACS11), pp. 150–160 (2005).

付　　録

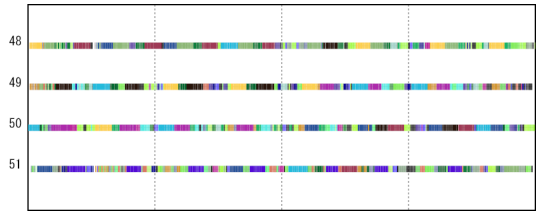
A.1　スケジューリング結果の可視化

コア数 4，L2 キャッシュサイズ 128KB，入力ストリーム WVGA における 1 フレームあたりのスケジューリング結果を可視化したグラフを図 13 に示す．**図 13(c)** は A パターンの S グループごとに色分けしたスレッドの実行状態を図示している．縦軸はコア番号を表し，横軸は左から右へ時間経過を表している．**図 13(d)** は同様に T グループごとに色分けしたスレッドの実行状態を図示している．同様に**図 13(a)** と**図 13(b)** は既存スケジューラにおけるスレッドの実行状態を図示しているが，拡張スケジューラとの違いを分かりやすくするために A パターンと同一 token のスレッドに対して同色の S グループと T グループの色付けを行っている．また同様に，**図 13(e)** **図 13(f)** は B パターンにおけるスレッドの実行状態を図示している．

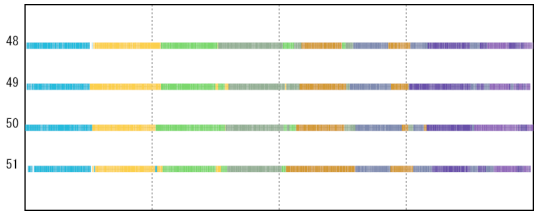


(a) S result of FCFS

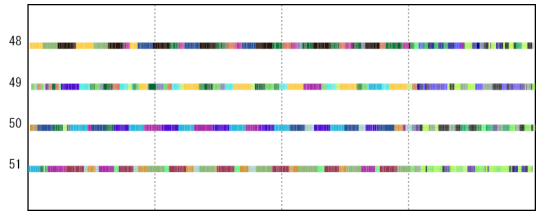
(b) T result of FCFS



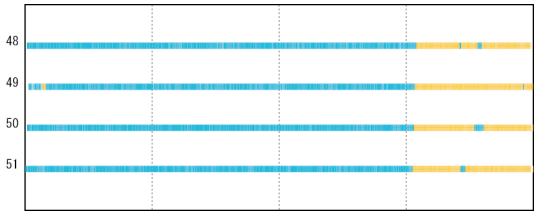
(c) S result of Pattern A



(d) T result of Pattern A



(e) S result of Pattern B



(f) T result of Pattern B

図 13　コア数 4，L2 キャッシュサイズ 128KB，入力ストリーム WVGA における 1 フレームあたりのスレッドスケジューリング結果
Fig. 13　Thread scheculing result around one frame on core numberl 4, L2 cache size 128KB and input stream WVGA