Linux ftraceを用いたマルチコアプロセッサ上での並列化プログラムのトレース手法

福意　大智[1]　島岡　護[1]　見神　広紀[1]　Dominic Hillenbrand[1]　木村　啓二[1]　笠原　博徳[1]

概要：ソフトウェアの適切な並列化により, マルチコアを搭載したコンピュータシステム上でアプリケーションを高速に動作させることが可能である. 並列化されたソフトウェアの挙動や性能を調査する手法として, ソースコードの解読や実行ダンプファイルの収集, プロファイラの利用, デバッガの利用といった方法が挙げられる. しかしこれらの手法ではどのようなタイミングにおいてコンテクストスイッチが発生したのか, システムで発生する事象に対してソフトウェアがどのような影響を受けているかといった情報を得ることは困難である. そこで, 本稿では並列化されたプログラムが実際に並列実行される様子をソフトウェアからトレースに任意のアノテーションを挿入可能とする拡張を施した Linux ftrace を用いて解析する手法を提案する. 提案手法を用いて, Intel Xeon X7560, ARMv7 の各々のプラットフォームにおいて equake, art, mpeg2enc というベンチマークのトレースを行い, これらのプログラムが実行時に OS からどのような影響を受けているか観測できることが確認できた. また, 1 回のアノテーションの挿入を Intel Xeon で 1.07[us], ARM で 4.44[us] で可能であることが確認できた.

キーワード：自動並列化, Linux, ftrace, マルチコア

1. はじめに

マルチコアプロセッサは組み込み機器からスーパーコンピュータまで様々な場面で活用されている. これらのマルチコアプロセッサを有効に利用するためにはプログラムの適切な並列化が必要である.

並列化プログラムのチューニングやデバッグには, その実行時の挙動の理解が不可欠であるが, 並列化されたプログラムが実行される様子の理解や説明は一般に困難である. ソースコードや実行ダンプファイル, プロファイリング結果, デバッグ情報を読むことも並列化されたプログラムを解析する手段としてよく利用されるが, このような方法でデータを解析することは時間がかかり, コストが大きい. 特にプログラムのコード量が多く, 複雑な動作をする場合において上記手法で解析を行うと膨大な量のテキストを読むことになり, 解析がより困難となる [1].

そこで, プログラムが並列実行される様子をガントチャートとして可視化することが, 並列化されたプログラムを理解するための有効な方法として利用されている. 複雑なデータをガントチャートとして示すことで, 並列化プログラムの挙動や性能の解析が容易となる.

[1] 早稲田大学
Waseda University

プログラムの実行をガントチャートとして可視化するツールのひとつに, Android Systrace が挙げられる. Android Systrace とは Android のデバイスにおいてソフトウェアの実行やオペレーティングシステムの割り込みをガントチャートとして可視化するツールである [2][3]. このツールは Linux ftrace というシステムに基づいており, ftrace は Linux カーネルにおいて実装されている [4][5].

また, プログラムが実行される様子とトレースし, ガントチャートを出力するツールとして Intel VTune[6] や perf timechart[7] が知られている. いずれもシステムレベルのスレッドをトレースすることができ, 時系列に沿ったチャートを取得することが出来る. 特に VTune は OpenCL アプリケーションのトレースに対応しており, GPU の使用効率を調べることが出来る. しかし VTune は Intel プロセッサあるいは Intel と互換性のあるプロセッサ上で動作することを想定しており, その他のプロセッサ上でトレースをすることは困難である. また, perf timechart は基本的に Linux が動作している環境ならばプロセッサに依存せずトレースを行うことが出来るが, ある時点においてプログラムのどの部分がどのように並列実行されているのかをトレースすることは困難である.

そこで, 本稿ではプロセッサに依存せず, Linux ftrace と

連携して並列化プログラムを解析する手法を提案する．本手法では, 並列化プログラムが保有する並列化に関する情報を, キャラクターデバイスファイルを介してカーネルモジュールに送信する．カーネルモジュールでは, 受け取った情報を一旦バッファに保管し, ftrace に転送する．ftrace では並列化プログラムが生成したスレッドの ID や生成時刻, 継続時間といった情報と統合し, ログファイルを作成する．そのログファイルを基に, HTML ファイルを作成する．これにより, 単にスレッドの生成やコンテクストスイッチだけでなく, 並列化されたプログラム内部のタスクがどのような挙動を示すのかということをプロセッサに依存せず解析することが可能となる．本稿では, 本手法を OSCAR 自動並列化コンパイラに適用して本手法の有効性と応用性を確認する．

以降, 第 2 章では本稿の提案手法について説明を行い, 第 3 章では Annotatable Systrace の概要を説明する．第 4 章では OSCAR 自動並列化コンパイラについて説明する．第 5 章において評価環境の説明を行い, 評価対象とするアプリケーションの説明を行う．そして第 6 章では本手法の性能評価の結果を示し, 実際にアプリケーションのトレースを行い, その結果を示す．最後に第 7 章においてまとめについて述べる．

2. Linux ftrace による並列化プログラムのトレース手法

本節では本研究で実際にスレッドのトレースを行うために用いた Linux ftrace 及び ftrace を利用する Android Systrace について説明する．

2.1 Linux ftrace

Linux ftrace とは Linux 内部の情報をトレースするツールであり, カーネルに組み込まれている．ftrace はカーネルのビルドオプションである `CONFIG_FUNCTION_TRACER` を有効にしてビルドすることで利用可能となる．ftrace の使用時には以下のようにして ftrace 用のファイルシステムをマウントする．

```
# mount -t debugfs nodev /sys/kernel/debug/
```

ftrace という名称は”function tracer”に由来しており, 元来は関数呼び出しのトレースを行うツールであった．しかし実際にはトレースの対象は多岐にわたり, トレース対象の一覧は`/sys/kernel/debug/tracing/events/`にて参照することが出来る．本稿ではスレッドの生成とコンテクストスイッチをトレースするため, 以下のようにして各トレース機能を有効にする．

```
# cd /sys/kernel/debug/tracing
# echo 1 > events/sched/sched_wakeup/enable
# echo 1 > events/sched/sched_switch/enable
```

2.2 Android Systrace

Android Systrace とは, Android SDK に標準で含まれるソフトウェアであり, Android デバイス上においてユーザーアプリケーションの実行時間やシステムプロセスの実行時間などを計測し, その結果をもとにチャートを生成するツールである．本ツールはスケジューラやスレッド ID などカーネルから得られる情報を統合し, ある一定時間における Android デバイス上でのプロセスの全体像を HTML ファイルとして得ることが出来る．

3. Annotatable Systrace

Annotatable Systrace とは, Android Systrace をユーザプログラムからトレースに記録するべき文字列を指定できるよう修正を加えたものである．これにより, 例えば並列化コンパイラが生成したタスクが実際にどのように並列実行されるのかということをトレースすることが可能である．すなわち並列化プログラムが持つタスクの情報をカーネルに送信し, カーネル内でスケジューラやスレッド ID などのカーネルの情報と統合する．統合された情報をもとにチャートを作成し, HTML ファイルとして出力する．結果として HTML ファイルにはスケジューラ上で動作したスレッドの情報が表示されるため, 並列化コンパイラによるタスク情報だけでなくスレッド ID やプログラム名なども表示される．

3.1 Annotatable Systrace のシステム構成

以下に, Annotatable Systrace 実装のために Linux ftrace に施した修正について説明する．

まず, Linux カーネルにおいて `task_struct` 構造体に新たに `oscar_mt_str` というメンバーを追加する．これはユーザプログラムから与えられた文字列を保管するためのものである．

次に, ユーザプログラムから systrace に文字列を送信する手段を用意する．そのための方法として, キャラクターデバイスドライバ [8] を使用する．まず, キャラクターデバイスファイルを管理するカーネルモジュールを用意する．このカーネルモジュールにより, `/dev/`以下にキャラクターデバイスファイルが生成される．ユーザアプリケーションはそのデバイスファイルに対し, write 関数を用いて文字列を書き込む．これにより文字列がユーザ空間からカーネル空間に伝えられる．カーネルモジュール内では `trace_sched_switch` という関数に `task_struct` 構造体を渡す．この関数は, ftrace が用意したトレース関数であり, この関数にデータを渡すことによりトレースファイルが生成される．`trace_sched_switch` 関数はスケジューラ内で利用されているため, ここではそのラッパを用意して使用する．

ここで, トレース対象となるアプリケーションは複数ス

レッドから構成されている．また，このような並列化されたプログラムが同時に複数実行される可能性があるため，デバイスファイルをただ一つ用意するだけでは文字列を書き込む際にクリティカルセクションが生じる．このクリティカルセクションを回避するため，デバイスファイルをコア数分用意し，またカーネルモジュール内でも文字列を一時的に保管するバッファをコアの数に応じて用意する．ここで，複数存在するスレッドから複数のデバイスファイルに書き込みを行う際，どのスレッドがどのデバイスファイルに書き込むのか決定する必要がある．そこで，スレッドとデバイスファイルを関連づけるデバイスファイルマネージャを用意する．デバイスファイルマネージャでは，例えば各スレッドの起動時に `sched_getcpu` 関数により CPU-ID を取得し，その ID に基づいて当該 CPU コアのデバイスファイルに対応するファイルディスクリプタの割り当てを行う．ここで，複数のスレッドが同じコアで動作している場合，当該スレッド間でファイルディスクリプタが共有されることになるが，同一コアで動作するスレッドが同時にファイルへ書き込むことは無いので，これはクリティカルセクションにはならない．また，カーネルモジュール内では受け取ったマクロタスク情報を `trace_sched_switch` 関数に渡すため，バッファをコアの数だけ用意している．文字列をどのバッファに格納するかを決定する際は，`smp_processor_id` 関数により決定する．

文字列をカーネルモジュールに伝え，最終的にカーネルに伝えるまでのシステム図を図 1 に示す．まず，ユーザ空間においてプログラム内にある文字列を各スレッドから write 関数によってキャラクターデバイスファイルを通じてカーネルモジュールに送信する．スレッドとデバイスファイルの関連付けはデバイスファイルマネージャによって既に対応づけられている．カーネルモジュール内では受け取った文字列をバッファに格納し，クリティカルセクションを回避する．そして `trace_sched_switch` 関数のラッパを呼び出し，最終的にカーネル，すなわち ftrace に文字列が転送され，トレースファイルを生成するための処理が行われる．

4. OSCAR マルチグレイン自動並列化コンパイラ

本節では，本稿の評価でトレース対象となるプログラムを並列化する OSCAR 自動並列化コンパイラ及びマルチグレイン並列処理技術について述べる [9][10]．OSCAR コンパイラは C もしくは Fortran プログラムを入力し，プログラム全体にわたる解析を通してマルチグレイン並列処理によりプログラムを並列化する，自動並列化コンパイラである．マルチグレイン並列処理とは，ループやサブルーチン，基本ブロック (BB) といった粗粒度タスク間の並列性を利用する粗粒度タスク並列処理，ループイタレーション間の並列性を利用する中粒度並列処理，BB 内部のステートメン

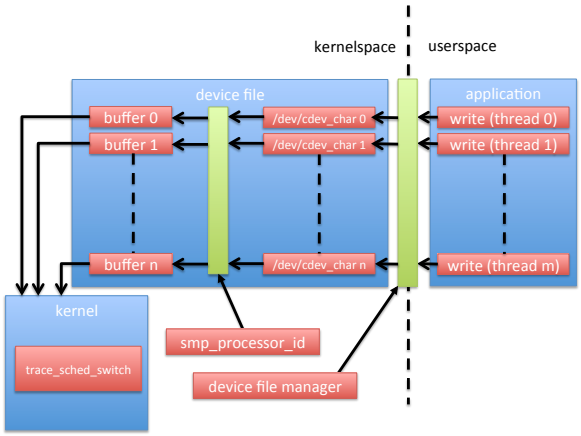

図 1 マクロタスク情報をプログラムからカーネルに伝えるまでのシステム

トレベルの並列性を利用する近細粒度並列処理を階層的に組み合わせてプログラム全域にわたる並列処理を行う手法である．各々の粒度における並列処理の詳細を以下に示す．

粗粒度タスク並列処理とは，ソースプログラムを複数のマクロタスクに分割し，そのマクロタスクを複数のプロセッサエレメント (PE) によって構成されるプロセッサグループ (PG) に割り当てて実行することにより，マクロタスク間の並列性を利用する並列処理手法である．粗粒度タスク並列処理では，ソースとなるプログラムを基本ブロック (BB) 繰り返しブロック (RB)，サブルーチンブロック (SB) の三種類の粗粒度タスク (マクロタスク (MT)) に分割する．なお，ループ並列処理が不可能な実行時間の大きい RB や，インライン展開を効果的に適用することができない SB に対してはその内部を階層的に粗粒度タスクに分割したうえで並列処理を行う．MT が生成されたのち，BB，RB，SB などの MT 間のコントロールフローとデータ依存を解析し，その結果をもとにマクロフローグラフ (MFG) を生成する．さらに最早実行可能条件に基づいた解析により MFG から MT 間の並列性を引き出し，マクロタスクグラフ (MTG) として表現する．もし MTG に条件分岐などの実行時不確定性が無い，すなわちデータ依存しか無い場合，プロセッサ間データ転送および同期オーバーヘッドが最小となるように，コンパイラは MTG 上の MT を静的にプロセッサあるいは PG に割り当て，各プロセッサ用のコードを生成する．MTG 中に条件分岐のような実行時不確定性がありダイナミックスケジューリングを用いる場合には，MT のコードに加えスケジューラのコードを生成し，実行時に動的に MT をプロセッサあるいは PG に割り当てる．SB や RB の内部に MTG が存在する場合，PG 内の PE はさらにグルーピングされて小さな PG が生成される．

中粒度並列処理では，PG に割り当てられた MT が Doall 可能な RB である場合，この RB はイタレーションレベル

表 1　トレース取得マシン RS440 の仕様

Server	HitachiHA8000/RS440
OS	Ubuntu 12.04.2 LTS (64bit, Linux 3.2.52)
CPU	Intel Xeon X7560 (2.27 GHz)
コア数	32
L2 Cache	2048KB
L3 Cache	24576KB
Compiler	GCC-4.6.3
RAM	32GB

表 2　トレース取得マシン Nexus7 2013 の仕様

Server	Nexus7 2013
OS	Android 4.3 (64bit, Linux 3.4.0)
CPU	Qualcomm Snapdragon S4 Pro (1.7 GHz)
コア数	4
L2 Cache	2048KB
Compiler	arm-linux-gnueabihf-gcc-4.6.3
RAM	2GB

で分割され, PG 内の PE に割り当てられ並列実行される.

本稿の評価では, OSCAR コンパイラが生成するコードにおけるマクロタスクと同期待ちのトレースを行う.

5. トレース取得環境

本節では OSCAR コンパイラにより並列化されたプログラムをトレースするための環境について概要を説明する. 本稿では, 評価対象マシンとして Intel Xeon プロセッサを搭載した HA8000/RS440 及び ARM マルチコアを搭載した Nexus7 2013 を使用した. それぞれのシステムの諸元を表 1 と表 2 に示す. 表 1 及び表 2 に示すように各マシンにはプロセッサとして Intel Xeon X7560 及び Qualcomm Snapdragon S4 Pro が搭載されている. また, 評価アプリケーションとして equake, art, mpeg2enc を用いた.

5.1 評価アプリケーション

5.1.1 MPEG2 Encoder.

MPEG2 エンコーダは, メディア処理の性能評価に用いられる MediaBench に含まれるベンチマークである. MPEG2 の規格に従った動画圧縮を行うものであり, メディアアプリケーションでは入力データが計算時間に大きく影響することがしばしばある.

5.1.2 EQUAKE.

equake とは, SPEC (The Standard Performance Evaluation Corporation) が提供するベンチマーク群である, SPEC 2000 に含まれるベンチマークの一つである. このベンチマークでは, メッシュ状にモデリングした地形を伝わる地震波の影響を有限要素法を用いてシミュレーションする.

5.1.3 ART.

art は, SPEC 2000 に含まれるベンチマークの一つであり, Adaptive Resonance Theory 2 (ART2) に基づいたニューラルネットワークを用いて画像認識を行う. このプログラムは, ヘリコプターと飛行機の画像を学習し, より大きな画像の中からそれらの画像を探索するものである.

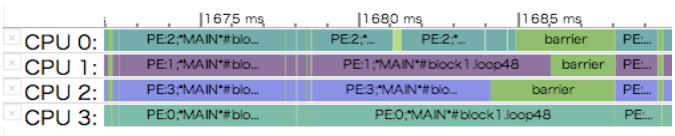

図 2　4PE 並列化した EQUAKE における負荷不均衡

6. トレース取得結果

本節では, OSCAR コンパイラである並列化したプログラムの実行時負荷不均衡, コンテクストスイッチ, スレッドマイグレーションの様子, そして 32PE 並列化プログラムが実際に並列実行される様子が提案ツールにより観測できることを示す. また, 本手法では並列化プログラムからタスク情報をカーネルに伝えるための write 関数が埋め込まれており, これに伴う実行時のオーバーヘッドについても計測した.

6.1 負荷不均衡

Nexus7 上において 4PE 並列化した equake を動作させ, その挙動をトレースした. そのトレース結果において負荷不均衡が生じている部分を図 2 に示す. 図はいずれのコアにおいても loop48 というマクロタスクが動作している様子を示しているが, CPU0, 1, 2, 3 における loop48 の終了時刻が全て異なっていることがわかる. また, PE1 から PE3 では PE0 の処理が終わるまで同期待ちをしていることがわかる. これは, loop48 の中にプログラムを実行しないと回転数が決定しない while ループがあるためであり, そのために各コアに割り当てられた loop48 の実行コストが大きく異なっているためである. このトレース結果により, 当該ループの分割方法を改善し負荷不均衡を解消する, あるいはバリア同期の時間にクロックゲーティングまたはパワーゲーティングを適用することで低消費電力化を図ることが出来ることがわかる.

6.2 コンテクストスイッチ

Nexus 7 において 4PE 並列化した equake 及び art を同時に動作させ, その挙動をトレースし, プログラムにおいてどのようなコンテクストスイッチが行われるのか観測した. その結果を図 3 に示す. 図では, barrier と表示されている部分は art のバリア同期待ち部分であり, それ以外のもの, すなわち PE あるいは pe と表示されている部分は equake が処理を行っている部分である. 図より, CPU0 では主に art の同期待ちが行われている一方で, CPU2 では主に equake が処理を行っていることがわかる. そして,

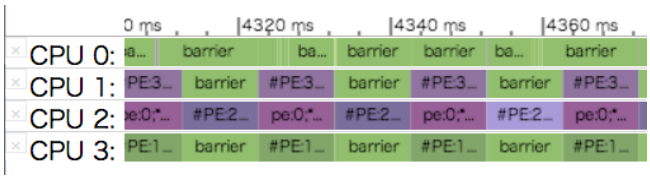

図 3 4PE 並列化した EQUAKE と ART の同時実行

図 4 MPEG2ENC のマイグレーション

CPU1 と CPU3 では art と equake が交互に動作していることがわかる．これにより，art と equake がお互いに CPU 資源を取り合っている様子が確認できる．通常，逐次実行の部分や負荷不均衡の部分でない限り，バリア同期は比較的短時間のうちに終了する．しかし図では CPU0 のようにバリア同期が連続している部分があるが，今回の例では 2 つの並列化プログラムを同時に実行しているため，バリア同期が終了するタイミングがずれてしまう．そのため，必要以上にバリア同期が長くなるのである．本評価では，提案トレース手法により観測が可能な項目を確認するために，人為的にこのような状況を作り出したが，通常はスレッドのアフィニティを設定し，各々のアプリケーションが利用するコアを分割することでこのような現象を回避する．

6.3 スレッドマイグレーション

RS440 において 2PE 並列化した mpeg2enc を動作させ，そのプログラムのスレッドがマイグレーションする様子をトレースした．トレースの結果を図 4 に示す．図では，はじめは CPU10 で動作していたスレッドが CPU8 にマイグレーションされる様子を示している．また，はじめは CPU8 で動作していたスレッドも同様のタイミングで CPU9 にマイグレーションされている．なお，スレッドのバインディングを行えばこの現象は回避することが可能である．

6.4 32PE 並列化プログラム

RS440 において 32PE 並列化した equake を動作させ，そのプログラムをトレースした．プログラム全体の挙動を図 5 に示す．図では，プログラム実行の前半部分においてほとんどの時間をバリア同期が占めていることを示している．これは，メモリの初期化やファイルの読み込みといった逐次部分が実行されているためである．後半では loop48 が実行され，並列処理が行われていることを示している．

loop48 の部分を拡大した図を図 6 に示す．図では，全てのコアにおける loop48 の終了時刻が異なっているため，負荷不均衡が生じていることがわかる．以上のように 32 コア上でもトレースが可能である．

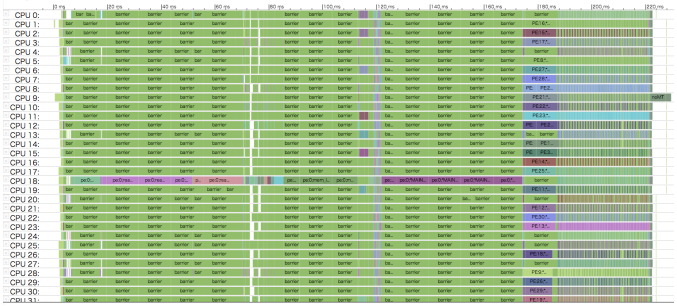

図 5 32PE 並列化した EQUAKE 全体の挙動

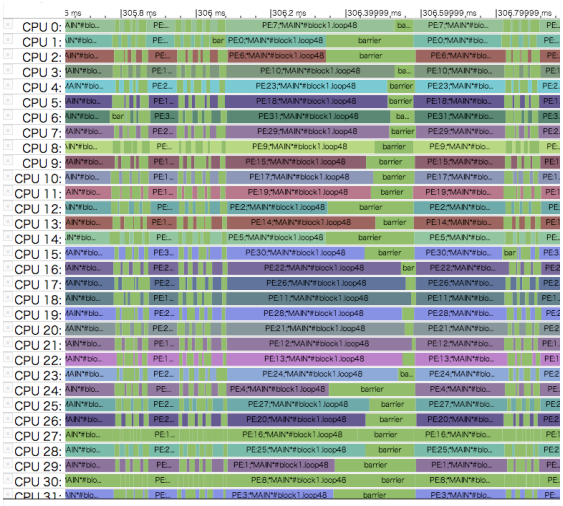

図 6 32PE 並列化した EQUAKE の loop48 の挙動

6.5 オーバーヘッド

RS440 及び Nexus7 において equake, art, mpeg2enc のトレースを行う際に生じるオーバーヘッドについて計測した．また，マクロタスク情報をカーネルに伝えている write 関数自身の呼び出しにかかるオーバーヘッドも測定した．その結果を表 3, 表 4, 表 5 に示す．表 3 及び表 4 の単位は秒であり，OSCAR コンパイラにより RS440 では 32PE 並列化したプログラムを使用し，Nexus7 では 4PE 並列化したプログラムを使用している．systrace, nowrite, original は，それぞれトレース用 write 関数の埋め込まれたプログラムをトレースしている状態，トレース用 write 関数のないプログラムをトレースしている状態，トレース用 write 関数のないプログラムをトレースしないで実行している状態を表す．また，表 5 の結果はトレース用 write 関数の呼び出し 1 回あたりのオーバーヘッドを示しており，単位は `us/call` である．

表 3 及び表 4 より，original に対して，equake は最大で 1.33 倍のオーバーヘッドが発生していることがわかる．art は最大で 1.59 倍のオーバーヘッドが発生し，mpeg2enc については最大で 3.95 倍のオーバーヘッドが発生している．プログラムによってトレースに要するオーバーヘッドが変化する原因として，マクロタスクの粒度の違いがある．マクロタスクの粒度が細かいと，マクロタスクの情報をカーネ

表 3 RS440 におけるアプリケーション毎のオーバーヘッド [秒]

	systrace	nowrite	original
equake	0.101	0.076	0.076
art	1.13	0.717	0.714
mpeg2enc	0.442	0.187	0.185

表 4 Nexus7 におけるアプリケーション毎のオーバーヘッド [秒]

	systrace	nowrite	original
equake	1.47	1.44	1.47
art	2.23	1.90	1.83
mpeg2enc	1.15	0.289	0.291

表 5 各マシンにおける write 関数のオーバーヘッド [us/call]

RS440	1.07
Nexus7	4.44

ルに伝えるための write 関数が呼び出される回数も結果として増えることになる．また, ループ中にマクロタスクが存在し, それにより write 関数が埋め込まれているとループのイタレーションだけ write が呼び出されることになる．マクロタスクの粒度はプログラムによって異なるため, トレースにかかるオーバーヘッドもプログラムによって異なると考えられる．

write 関数呼び出しに要するオーバーヘッドは, 表 5 より, 1 回あたりで RS440 において 1.07[us], Nexus7 において 4.44[us] を要したことがわかる．

7. まとめ

本稿では ftrace を用いて並列化プログラムの挙動をスレッド単位で詳細にトレースし, ガントチャートとして視覚化する手法について述べた．OSCAR コンパイラにより並列化されたプログラムについて, RS440, Nexus7 において評価を行い, コンパイラと協調することで, マクロタスク, 負荷不均衡, コンテクストスイッチ, スレッドマイグレーションのトレースを本手法により取得することが出来た．これにより, 本手法の有効性と応用性を確認することが出来た．Android 4.3 以降に Android API として Trace クラスが用意されているが, これは, 任意の文字列を Android Systrace が出力するチャートに表示するものである．本稿の提案手法と Trace クラスの性能比較が今後の課題である．

参考文献

[1] Eileen Kramer, John T. Stasko: The Visualization of Parallel Systems: An Overview, *Journal of Parallel and Distributed Computing* (1993).
[2] Google: Android Systrace, `http://developer.android.com/tools/help/systrace.html` (2014).
[3] 後藤隆志, 武藤康平, 山本英雄, 平野智大, 見神広紀, 木村啓二, 笠原博徳: プロファイル情報を用いた Android 2D 描画ライブラリ SKIA の OSCAR コンパイラによる並列化, **情報処理学会第** *199* **回** *ARC*・**第** *142* **回** *HPC* **合同研究発表会** (2013).
[4] Jake Edge: A look at ftrace, `http://lwn.net/Articles/322666/` (2014).
[5] Steven Rostedt: Debugging the kernel using Ftrace - part1, `http://lwn.net/Articles/365835/` (2014).
[6] Intel Corporation: Intel VTune Amplifier XE 2013, `https://software.intel.com/en-us/intel-vtune-amplifier-xe`.
[7] Stephane Eranian ,Eric Gouriou, Tipp Moseley, Willem de Bruijn: perf, `http://perf.wiki.kernel.org/index.php/Tutorial` (2014).
[8] Ariane Keller: Kernel Space - User Space Interfaces, `http://people.ee.ethz.ch/~arkeller/linux/kernel_user_space_howto.html` (2014).
[9] 本多弘樹, 岩田雅彦, 笠原博徳: Fortran プログラム粗粒度タスク間の並列性検出手法, **電子情報通信学会論文誌** (1990).
[10] 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).