Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

マルチコア環境でのプロセス動作予測によるリソース配分最適化

大野有輝[†1]　菅谷みどり[†2]
秋岡明香[†1]　中島達夫[†1]

近年，CPU のマルチコア化によって処理能力を向上させる手法が一般的になっている．しかしながらマルチコア環境では，共有資源におけるリソースの競合による処理性能低下の問題がある．本研究では，プロセスの動作予測の結果から，実行コアの割当を決めることで，アプリケーションの処理性能を向上させる SPLiT（Scalable Performance Library Tool）を提案する．SPLiT は，(1) PMU（Performance Monitoring Unit）を用いたハードウェアの性能データの収集,(2) アプリケーションの処理に関する情報を元に動作予測を行うシステムを提供する．本研究では，SPLiT システムを Linux 上に実装し，Apache と MySQL に SPLiT lib を組込むことで，ウェブアプリケーションの最適化を行った．評価では必要な開発コストを最小におさえ，Web アプリケーションの性能を最大で 26%向上させたことを示した．

Optimal Resource Assignment by Process Behavior Prediction

Yuki Ohno ,[†1] Midori Sugaya ,[†2] Sayaka Akioka [†1]
and Tatsuo Nakajima[†1]

Recently, multicore processors have become populer, however, the concurrent execution with multicore processors causes resource contentions that can turn into a performance bottleneck. In this research, we present SPLiT (Scalable Performance Library Tool) which optimizes resource assignment by predicting processes behaviors. SPLiT collects the performance data in the kernel with PMU (Performance Monitoring Unit) and in processes of applications through the API of its library. With the result of prediction, it assigns CPU cores to each process and improves usage efficiency and caches.

We implemented SPLiT on Linux, built its library into Apache and MySQL for the optimization of web applications, and evaluated its performance. The result shows SPLiT can improve the performance up to 26% without the development cost of applying SPLiT lib.

1. はじめに

近年，CPU のマルチコア化によって処理能力を向上させる手法が一般的になっている．CPU のマルチコア化は省電力や発熱量の低減にも有効な手段であり，今後もコア数を増やすことで処理能力の向上を目指す傾向が続くと考えられる．この傾向を受け，ソフトウェアによるマルチコアへの対応も進んでいる．

現在のオペレーティングシステム（以下，OS）の多くは，マルチプロセッサ，マルチコアに対応しており，複数のプロセス，スレッドを別々のコアで並列に実行することが可能になっている．また，アプリケーションもマルチコアを意識してプログラミングされているものが増加しており，プロセス内の処理を分割し，複数のスレッドで処理させることで，コアが複数ある場合に並列実行できるようになっている．しかし，OS やアプリケーションによるマルチコアへの対応は，シングルコアで用いていた手法の延長であるものが多く，コア数の増加につれ，様々な問題が顕在化している．

その一つは，並列化時の性能低下があげられる．要因としては，共有リソース競合による処理性能の低下，キャッシュ効率の低下，同一デバイスへの同時アクセスによる効率低下などがあげられる．

一方，これらの問題に対して，ソフトウェアとしては，並列計算ライブラリ, ロックの粒度の細粒度化，ロックフリーアルゴリズム[12)]，並列プログラミングモデル，コンパイラによる自動最適化等，性能低下を防ぐための様々な工夫がされてきた．しかし，個々のケースに合わせて独自の最適化手法を用いることが多く，まだ様々なマルチコア環境に対応するプログラミングモデルが確立されていないのが現状である．

並列環境では，キャッシュのローカリティというハードウェアの構成を考慮したプログラミングが重要である．しかし，キャッシュの最適化のみを考慮すると，そのハードウェア構成独自の最適化になり，汎用的なプログラムを作成することが難しい．また，アプリケーション全体を考慮したスケジューリングを行うことは難しい．現在，利用されている OS のプロセススケジューラの多くは，主に CPU の使用時間だけを考慮しており，そのプロセス

†1 早稲田大学理工学術院
Waseda University, Department of Computer Science
†2 独立行政法人科学技術振興機構 ディペンダブル組込み OS 研究開発センター
Japan Science and Technology Agency (JST), Dependable Embedded OS RD Center

Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

が CPU 以外のどのリソースにアクセスするかは考慮されていない問題がある

本研究では，CPU の割当と, キャッシュの割当の双方のリソースの割り当てを最適化するための手法を提案することを目的としている．また，実現のために，カーネルとアプリケーションの両方での情報収集と最適化を行う SPLiT (Scalable Performance Library Tool) を提案する．カーネルレベルの最適化では，ハードウェアの情報取得や制御ができ，実行されているアプリケーション全体を考慮した最適化が行える利点がある．しかし，欠点としてカーネルの挙動やインタフェースが変わることで，既存のアプリケーションが利用できなくなったり，特殊なプログラミングが要求される問題がある．また，全てのアプリケーションに同様な最適化が適用されるため，アプリケーションによっては効果が無い場合や，逆に性能が低下する可能性がある．

一方，アプリケーションレベルの最適化では，各アプリケーションに合わせた最適化ができるが，ハードウェア構成や他に動作しているアプリケーションを考慮した最適化が難しい．また，カーネル内でのデータ共有やロックがボトルネックになる場合は，アプリケーションをいくら最適化しても性能向上は望めない．

本研究では，カーネルの変更はマルチコア環境での最適化のためのハードウェアサポートを追加するだけにとどめ，最適化はユーザライブラリを通して行う方式とした．これにより，既存のアプリケーションは変更無く使い続けることができ，最適化を行いたいアプリケーションに対しては，ライブラリを用いることで選択的に最適化が行えるようにした．また，データ収集やスケジューリングなどの処理はライブラリ内で行うため，最適化の際のプログラム変更量が少なく抑えるものとした．

論文の残りは下記の構成とする．第 2 節にて設計について述べる．第 3 節にて実装について述べ，第 4 節にて評価について述べる．第 5 節にて関連研究について述べる．最後に第 6 節で現状の課題と結論について述べる．

2. SPLiT

2.1 設計概要

本研究では，マルチコア環境におけるリソース配分最適化のシステムとして，SPLiT（Scalable Performance Library Tool）を提案する．SPLiT は，アプリケーションのキャッシュ効率の情報とサイクル数情報の統計データを元に，アプリケーションの動作予測を行い，最適なコア配分を支援するための仕組みである．本節では，SPLiT の設計について述べる．SPLiT は次の 3 つの処理によって，リソース配分最適化を実現する．

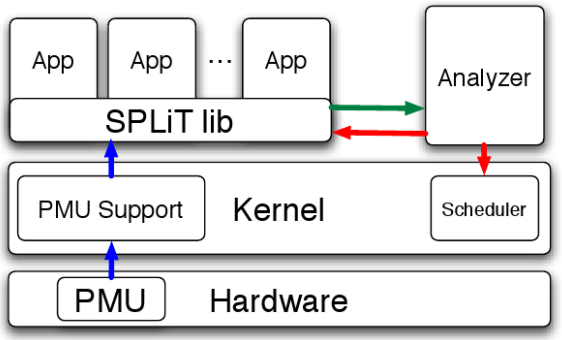

図 1 リソース配分最適化の処理の流れ

- データ収集
- 動作予測
- リソース配分

リソース配分最適化の処理の流れを図 1 に示す．

2.2 データ収集

SPLiT はまず，アプリケーションの動作予測に必要なデータの収集を (1) カーネル，(2) アプリケーションの 2 つのレベルで行う．各アプリケーションは特定の処理の開始時および終了時にライブラリ関数を呼び出し，処理の開始時から終了時までの CPU サイクル数およびキャッシュミスの回数の計測を行う．結果は統計データとしてメモリ上に格納する．

2.2.1 カーネルでのデータ収集

並列化の効果を引き出すためには，できるだけハードウェアのローカリティを意識したスケジューリングを行う必要がある．そのために，本研究では，スレッドの各処理にかかった CPU サイクル数，およびキャッシュミス回数を PMU から取得する．

PMU（Performance Monitoring Unit) は，システムの性能情報を取得するためのプロセッサ機能である[2)]．SPLiT では，この PMU を用いて，各処理にかかった CPU サイクル数，およびキャッシュミス回数の計測を行う．PMU はハードウェアのサポートによる機能であることから，ソフトウェアのみによる性能データの収集よりも低オーバーヘッドでデータを取得することができるという利点がある．

性能データの収集には，カーネルによる PMU のサポートが必要になる．カーネル側でスレッドの切り替わり時に PMU 用レジスタの値の退避や，スレッドの計測対象イベントの設

Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

表 1 ライブラリ API

API 関数		備考
app_id	split_init(app_id, related_app_id, code_key_type)	アプリケーション初期化
code_id	split_start(related_code_id, code_key)	処理開始
void	split_end()	処理終了
void	split_set(setting_type, value)	設定変更
void	split_get(setting_type, value)	設定取得

定を行う．

また，ユーザプログラムから PMU の計測要求を受け付けるための API を用意した．

2.2.2 SPLiT lib でのデータ収集

PMU を用いた CPU サイクル数，キャッシュミスの回数の計測は，アプリケーションが動作するタイミングで行う必要がある．動作タイミングに応じて計測を行うために，SPLiT lib は，ユーザプログラムへの API を提供する．

SPLiT lib が提供する API を表 1 に示した．アプリケーション初期化時，計測対象のプログラムの開始時および終了時，設定値の変更および取得時の API となる．

SPLiT lib は，計測開始から計測終了までの処理動作を 1 つの処理とし，処理毎に処理 ID（code_id), 性能データの統計値の記録と，スレッドスケジューリングを行う．さらに，処理毎の単位の他にも各アプリケーション毎にアプリケーション ID を付与し，アプリケーション ID 毎に性能データを管理する．表 2 に記録するデータの一覧を示した．

2.2.3 アプリケーション処理フロー

初期化：split_init 関数を呼び出す． split_init 関数の引数には，アプリケーション ID（app_id)，関連するアプリケーションの ID（related_app_id)，処理の識別に用いるデータの型（code_key_type）を指定する．

開始：split_start 関数を呼び出す．split_start 関数の引数には，処理の種類を識別するための値（code_key）および関連する処理の ID（related_code_id）を与える．これにより，処理の種類毎に性能データを集計する．

終了：split_end 関数を呼び出す．これにより，性能データの計測を終了し，データの記録を行う．先に述べたカーネルサポートにより PMU にアクセスして統計データを取得する．統計データは，各アプリケーション毎に共有メモリ上に記録する．また，code_key をキーとするハッシュによって，各処理のデータ格納場所を決定する．

変更：split_set 関数を用いて，設定する項目（setting_type）と設定項目の値（value）を指定し変更を可能とした．また，データ収集の有無，データ解析頻度，ハッシュサイズ，平均値の重み，処理回数の閾値の変更が可能とした．

表 2 性能データ

データ			備考
共有			
	アプリケーション毎		
		app_id	アプリケーション ID
		related_app_id	関連するアプリケーションの ID
		hash_size	データ格納用ハッシュサイズ
	処理毎		
		code_id	処理 ID
		count	処理が行われた回数
		cycle_count	処理にかかったサイクル数
		cache_miss	キャッシュミスの回数
		related_code_id	関連する処理の ID
		cpu_mask	処理を行うコア番号
非共有			
		code_key_type	処理識別用データの型
		code_key_table	処理識別用ハッシュデータ

取得：split_get 関数を用いて，設定項目（setting_type）に現在設定されている値を取得する．値を取得できる項目は，split_set 関数で設定できる項目の他に，app_id，related_app_id，code_id の値とした．

2.2.4 サンプル数による統計値の適応

SPLiT は，アプリケーションの処理毎に統計値を更新する．統計値としては，CPU サイクル数，キャッシュミスの回数を集計する．集計は平均値を算出した．ただし，取得するサンプル数に応じて異なる式を用いる．(1) サンプル数が少ない時：相加平均とする．これは，突出した値によるデータの過度な変動を抑えるためである．(2) サンプル数が多い時：指数移動平均とする．指数移動平均は，過去のデータを引き継ぎつつ，現在に近い値に重みづけを行う．具体的には，現在の統計値を a_n，現在の処理回数を n，新たに取得したデータを a，新しい統計値を a_{n+1} としたときの統計値の計算式を式 (1) に示す．

$$a_{n+1} = \begin{cases} (a_n \times n + a)/(n+1) & (n < \theta) \\ a_n \times w + a \times (1-w) & (n \geq \theta) \end{cases} \quad (1)$$

また，閾値 θ を設け，処理の実行回数が θ よりも小さい場合，平均値を統計値とし，実行回数が θ 以上の場合，重みを w とする指数移動平均を用いた．

Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

2.2.5 処理の依存関係の考慮

処理の依存関係を示すために，related_code_id 引数に処理の code_id を渡すものとした．split_start 関数の戻り値には code_id を与えるが，この時，処理 A が実行されると処理 B が必ず実行されるような関連性がある場合には，処理 B の related_code_id 引数に処理 A の code_id を渡す．related_code_id を，スケジューリングのヒントにする事で依存関係を考慮したスケジューリングを可能とした．

2.3 動作予測

SPLiT は，プロセスから収集したデータを元にプロセスへのコアの割当を決める．動作予測部では，データ解析用の Analyzer デーモンが一定時間毎に各アプリケーションでの計測結果を収集し，動作予測を行い，リソースの割り当てを行う．

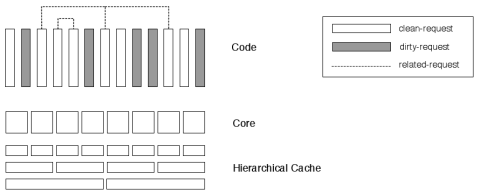

図 2 コア割当前

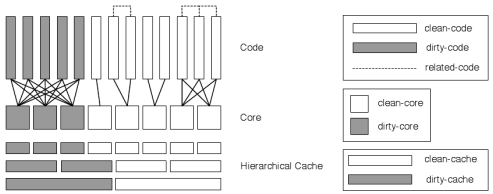

図 3 コア割当後

2.4 動作予測によるリソース割当

Analyzer デーモンによるコア割当時の判定では，統計値によるキャッシュヒット率と CPU 使用時間の二つのデータを用いた．

2.4.1 キャッシュ利用効率による処理の仕分け

キャッシュヒット率の向上のためには，同じデータにアクセスする処理を同一コアで行うようコアの割当を行う必要がある．そのため，本研究ではキャッシュ効率の違いにより処理を区分し，効率が良い処理を優先させることでキャッシュ効率を向上させるものとした．

処理の区分：dirty-code/clean-code ：処理ごとにキャッシュミス回数が閾値より大きいものを，キャッシュ効率の向上が望めない処理 dirty-code とし，それ以外をキャッシュ効率の望める処理 clean-code とした．図 2 にコア割り当て前の clean-code と dirty-code の状態を示した．

コアの区分：dirty-core/clean-core ：コア側でも，clean-code を処理するコア（clean-core）と dirty-code を処理するためのコア（dirty-core）とに区分する．これは，dirty-code によるキャッシュ汚染が clean-code の処理を妨げないことを目的とした．

2.4.2 CPU 使用時間による負荷分散

CPU 負荷分散の観点から，clean-code，dirty-code のそれぞれの処理にかかるクロック数を計算し，その比率から割り当てるコアの数を決めるものとした．

dirty-core の割当: dirty-code の場合，キャッシュの内容が次々と書き換えられるため，キャッシュの効率的な利用は期待できない．そこで，キャッシュの割当については考慮せず，CPU の負荷分散のみを考慮する．CPU 負荷分散を考慮したスケジューリングは，OS のプロセススケジューラが行っているため，SPLiT lib によるコア割当の細かい制御は行わず，OS のスケジューラに委ねる．ただし，dirty-code の clean-core への割当は禁止する．

clean-core の割当: clean-code はコア割当を工夫することで，キャッシュの有効利用が期待できる．そこで，処理同士の関連性をもとに，各処理の clean-core への割当を決める．

処理同士の関連性は，SPLiT で取得するアプリケーション ID，関連するアプリケーションの ID，処理 ID，関連する処理 ID の 4 つである．

(1) 処理 ID が等しい場合，同一コアで処理する．処理 ID が等しいものは，同じコアで処理する．これは，処理 ID が等しい処理は同じデータを扱う可能性が高いことから，妥当な割当だといえる．

(2) アプリケーション ID 毎にコアの割当を行う．同一 ID のアプリケーションは，データを共有している可能性が高く，キャッシュを共有しているコアで実行した方が効率が良いと考えられる．そのため，同じコアに割り当てる．また，関連するアプリケーション ID の情報をもとに，関連するアプリケーション同士が近くに配置されるようコアの割当を行う．

(3) アプリケーションへ割り当てるコアの数を決定する．dirty-core，clean-core の割当時と同様に，各アプリケーションの処理の合計 CPU 使用時間の比率をもとに決定する．

(4) 各処理のコアの割当を決定する．処理にかかるクロック数が多いものから順に割当を

行う．この時，アプリケーションに割り当てられたコア，関連する処理が割り当てられたコア，全てのコア，の順に割当を試みる．

2.4.3 コア割り当ての手順

以上をまとめると，コアの割り当ては次の 1 から 5 までの手順となる．1. 事前に 1 コアに対する平均クロック数（全処理の合計クロック数÷コア数）を求めておく．2. 次に，アプリケーションに割り当てられたコアのうち，それまでに割り当てられた処理の合計クロック数が最も小さいコアを選ぶ．3. もし，そのコアに処理を割り当てた場合に，そのコアの合計クロック数が 1 コアに対する平均クロック数を超えないならば，そのコアへ処理を割り当てる．4. 平均クロック数を超える場合は，関連する処理が割り当てられたコアに対して，同様に割当を試みる．5. 関連する処理が無い場合，あるいは関連する処理が割り当てられたコアへ処理を割り当てた場合に平均クロック数を超える場合は，全てのコアうち，最も合計クロック数の少ないコアへ処理を割り当てる．コア割当後の code と core の関係を図 3 に示す．

2.5 リソース配分

Analyzer が読み込む共有メモリ上には，性能データの他に，割当コアを指定するデータ（表 2 における cpu_mask）がある．Analyzer は，コア割当が決定すると，共有メモリに割当コアを指定するデータを書き込む．

2.2.4 節で述べたように，処理開始時に split_start 関数を呼び出した際，引数として渡された処理の識別子（code_key）に対応する割当コアへスレッドを移動させる．これによって，それぞれの処理を特定のコアで行う．

3. 実　　装

この章では，第 2 節で説明した SPLiT の実装について説明する．

3.1 開 発 環 境

まず，開発環境として使用したハードウェアおよびオペレーティングシステムの概要を表 3 に示す． Core i7 965 は，1 つの CPU の中に 4 つのコアを搭載している．HyperThreading Technology[11] により，各コアは 2 つの論理プロセッサを持ち，1 つのコアで 2 つのスレッドが並列に実行される．キャッシュは 3 段階あり，L1 キャッシュおよび L2 キャッシュはコア毎に存在し，各コアの 2 つのスレッドで共有されている．また，L3 キャッシュおよびメモリは全コアで共有されている階層構造となっている[8]．

また，リソース配分の最適化を行う対象として，ウェブサーバで広く用いられている

表 3 開発に使用したハードウェアおよびオペレーティングシステム

プロセッサ	Intel Core i7 965（3.20 GHz）
メモリ	DDR3 SDRAM 3GB
ネットワークデバイス	Marvell Yukon 88E8056 PCI-E Gigabit Ethernet Controller
OS	Linux 2.6.30

Apache[1] と MySQL[15] を採用した．Apache，MySQL 共にリクエストの種類によって処理内容や計算負荷が大きく異なるため，SPLiT を用いて，処理内容や計算負荷に合わせたリソースを提供することで，効率的な処理が可能になると考えられる．

3.2 システム基盤

本研究では，カーネルの PMU サポートとして Perfmon3[17] を用いた．SPLiT では，Perfmon3 を用いて，各プロセスの CPU クロック数，キャッシュミス回数の取得をしている．さらに，ウェブサーバ向け性能解析フレームワークである mBrace[21] を用いて，SPLiT lib を Linux 2.6.30 上に実装を行った．

SPLiT lib では，スケジューリングに必要な情報の統計データのみをメモリ上に格納し，それをユーザモードの Analyzer が分析して最適化を図ることで，軽量化を実現しオンラインでの性能解析を可能とした．また，コアの配分を最適化する対象として，ウェブサーバの Apache 2.2.11[1] と，MySQL 5.1.32[15] を用いた．

3.3 データ収集部

3.3.1 PMU で収集するデータ

SPLiT では，core イベントとして専用カウンタで Unhalted Core Cycles を，汎用コアで L2_RQSTS.LD_MISS を計測する．Unhalted Core Cycles は，計測の開始から終了までの Unhalted（命令を実行しない状態を除く）コアサイクル数である．コアサイクル数は，コア毎のクロックによってカウントされるため，コアの周波数によって，カウントが増加する速度が異なる．これに対し実装では，処理時間ではなくコアサイクル数を計測することにより，コア周波数の違いによる値の変動を抑制した．変動の少ない値を用いることにより，プロセス動作予測の精度を上げることができる．また，L2_RQSTS.LD_MISS は，L2 キャッシュでのデータキャッシュミス回数である．

3.3.2 Apache におけるデータ収集

Apache では，処理の種類を識別するための引数（code_key）として，リクエストされたページの URL を用いる．性能データの計測開始は，MPM によって予め生成されているプロセスやスレッドが，クライアントからの HTTP リクエストを受け取り，そのリクエスト

Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

をパースした時点とした．また，計測終了は，リクエストされたデータのクライアントへの送信が完了した時点とした．

また，SQL クエリ送信時に，Apache の処理 ID を共に送信するよう変更を行った．本研究での設定では，Apache 上のアプリケーションから MySQL サーバへクエリを送信する場合，MySQL のライブラリを経由してクエリの送信が行われる．そこで，MySQL ライブラリに変更を加え，クエリと共に処理 ID の送信を行うようにした．これによって，Apache 上で Perl，PHP，Ruby 等様々なモジュールに変更を加えることなく，処理 ID の送信を実現することができる．

3.3.3 MySQL におけるデータ収集

MySQL では，処理の種類を識別するための引数（code_key）として，クエリの種類と対象のテーブルの組を用いた．MySQL での性能データの計測開始は，サーバプロセスが SQL クエリを受信して処理用スレッドを生成後，そのスレッド内で SQL クエリのパース開始時とした．このとき，Apache から SQL クエリと共に送信された処理 ID を，関連する処理 ID として split_start 関数に渡す．また，計測終了は，Apache と同様に，リクエストされたデータのクライアントへの送信が完了した時点とした．

Apache および MySQL での処理の流れを図 4 に示す．

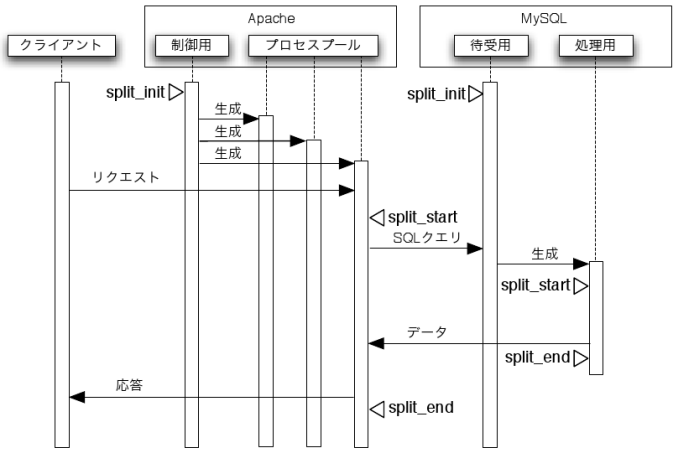

図 4 Apache および MySQL での処理の流れ

3.4 動作予測部

動作予測部では，デーモンとして実装した Analyzer が処理を行う．

3.4.1 各データの集約

性能データはアプリケーション毎に共有メモリ上へ格納されている．Linux で共有メモリを扱う場合，shmget 関数で共有メモリのキーとサイズを指定する．キーとして同じ値を指定することで，別々のプロセスで同じ共有メモリが参照できる他，キーとして IPC_PRIVATE 定数を与え，未使用のキー値から新しいものが 1 つ選ばれ，shmget 関数の戻り値として渡す．SPLiT では，shmget のキー値として，アプリケーション ID（app_id）を用いる．また，各アプリケーション ID は，split_init 関数の呼び出し時に，Analyzer によって用意された共有メモリに書き込まれる．Analyzer は共有メモリに書き込まれたアプリケーション ID リストをもとに，それぞれのアプリケーションの性能データにアクセスし，データの集約を行う．

3.4.2 コア割当情報の書き込み

各アプリケーションの共有メモリに格納されている処理毎のデータのうち，cpu_mask を変更することで，コア割当情報を各アプリケーションの SPLiT lib に伝える．コア割当の際，CPU マスクとして引数に渡す変数の型が cpu_set_t であることから，共有メモリ上にも cpu_set_t 型のデータとして格納する．cpu_set_t 型は構造体であり，各ビットが個々の CPU の許可，不許可に対応している．複数のビットを 1 にすることで，複数の CPU のいずれかで実行されるように設定できる．また，全てのビットを 1 にすることで，コア割当の制限を無くし，通常のスケジューリングに戻すことができる．

3.5 コ ア 割 当

コアの割り当ては，split_start 関数の呼び出し時に，各処理に対応するコアへスレッドを移動させることで実現する．Linux では，プロセスの CPU affinity を設定する sched_setaffinity 関数を用いることで，コア割当が実現できる．本関数では，設定対象のプロセス ID と CPU マスクを引数として与える．sched_setaffinity 関数を呼び出したとき，指定したプロセスが実行可能な CPU のいずれかで実行されていない場合，そのプロセスは CPU マスクで指定された CPU のいずれかに移動を行う．一方，現在実行中の CPU が，CPU マスクによって実行許可されている場合，プロセスの移動は発生しない．

SPLiT の場合，各処理が SPLiT lib 内で CPU マスクを設定するため，sched_setaffinity の設定対象は自分自身（プロセス ID には 0 を指定する）となる．また，CPU マスク設定前に現在の CPU マスク設定を sched_getaffinity 関数で取得，退避しておき，split_end 関

数呼び出し時に設定を元に戻す．

4. 評　　価

本研究では，ウェブアプリケーションに SPLiT を組込み，動作予測精度，応答時間，オーバーヘッドについて評価を行った．

4.1 実験環境

4.1.1 ハードウェアおよびソフトウェア構成

評価実験に用いたクライアント用マシンのハードウェア構成を表 4 に示す．Web サーバ用マシンは，第 3 節で実装に用いたものと同一マシンとした．

表 4　クライアントのハードウェア構成

プロセッサ	Intel Core 2 Duo (2.0 GHz)
メモリ	DDR3 SDRAM 2GB (1067 MHz)
ネットワークデバイス	10/100/1000BASE-T（ギガビット）Ethernet
OS	Mac OS X 10.5.8

4.2 ワークロード

4.2.1 RUBiS

性能の評価対象として RUBiS[20] を利用した．RUBiS(Rice University Bidding System) は，eBay[5] をモデルとしたオークションサイトプロトタイプである．RUBiS によるオークションサイトは，PHP，Java servlets，EJB（Enterprise Java Bean）の 3 種類の実装があり，本研究では，PHP での実装によるオークションサイトを利用した．データベースはいずれの実装でも共通になっており，MySQL で 7 つのテーブルを用いて，オークションのユーザや出品物の情報を管理している．

4.2.2 Apache JMeter

Apache JMeter[10]（以下，JMeter）は，負荷テストや性能計測のためのアプリケーションである．ウェブアプリケーションのほか，データベースやメールサーバ等，様々なサーバのテストを行うことができる．本研究では，クライアント用マシンで JMeter を用いて Web サーバの RUBiS にアクセスし，性能計測を行った．

4.3 動作予測精度

各処理を 1,000 回行って統計データ取得後，再び同じ処理を 1,000 回行い，統計データとの誤差を計測した．また，Analyzer によるコア割当を行わない場合と，コア割当を行った場合での誤差の計測を行った．計測結果は表 5 のとおりである．ここでは誤差を統計値に対する計測値の差の割合とする．誤差を e とし，n 回目の処理において，Analyzer で求めた統計値を s_n，処理時に実際に計測した値を m_n とすると，誤差の計算方法は，式（2）のとおりである．また，サイクル数最大誤差，キャッシュミス回数最大誤差は，各処理における誤差（$(s_n - m_n)/m_n$）のうち最大のものとした．

$$e = \frac{1}{N}\sum_{n=1}^{N}\frac{s_n - m_n}{m_n} \tag{2}$$

表 5　予測誤差

	コア割当なし		コア割当あり	
	Apache	MySQL	Apache	MySQL
サイクル数誤差 [%]	11.2	5.7	9.0	1.3
キャッシュミス回数誤差 [%]	9.0	6.7	4.1	1.8
サイクル数最大誤差 [%]	129.3	230.1	134.2	148.1
キャッシュミス回数最大誤差 [%]	100.4	227.2	94.7	170.0

表 5 より，Apache のサイクル数最大誤差を除いて，コア割当を行った方が誤差，最大誤差共に小さくなっている．理由は，処理を行うコアを限定したため，サイクル数やキャッシュミス回数の変動が小さくなったためであると考えられる．ただし，コア割当を行っても最大誤差は依然として大きく，キャッシュミス回数は最大で統計値の 2.7 倍となっている．

4.4 応答時間

クライアントの JMeter を用いて，ウェブアプリケーションの応答時間の計測を行った．ウェブアプリケーションの応答時間とは，JMeter で設定したシナリオに従いウェブアプリケーションへ HTTP リクエストを送信し，全ての HTTP リクエストへの応答を受信完了するまでの時間とした．応答時間の計測では，次の 2 種類のシナリオを作成した．(1) 一度の処理にかかる時間が短く，総受信されるデータ量が小さいページにアクセスするが，アクセス量が非常に多いケース．(2) アクセス数は少ないが，処理にかかる時間が長く，大量のデータを総受信するページにアクセスするケースである．(1)(2) でのアクセスファイル数，受信ファイルサイズ，SQL クエリ発行数は表 6 のとおりとした．

それぞれのシナリオにおいて，*i*）通常の Apache と MySQL を利用した場合（通常システム），*ii*）SPLiT を組込み性能データを収集するが最適化は行わない場合（予測あり・最適化なし），*iii*）SPLiT で性能データの収集と最適化を行う場合（予測あり・最適化あり），

の 3 つの場合の性能を計測した．

表 6 各シナリオの詳細

	アクセス数の多いシナリオ	データ量の多いシナリオ
クライアント数	1,000	100
合計アクセスファイル数	14,000	1,300
合計発行 SQL クエリ数	10,000	4,542,500
合計アクセスデータベース行数 [百万行]	184.4	23.0
合計受信ファイルサイズ [MB]	218.2	409.7

アクセス数が多い場合の応答時間の計測結果を表 7 に示す．アクセス数が多い場合，SPLiT システムによる性能データ収集・動作予測のオーバーヘッドは 5.0%となっている．また，予測結果をもとにコア割当を行うことによって，通常システムに対して応答時間が 4.7%短縮している．

表 7 アクセス数が多い場合の応答時間

	応答完了時間 [ms]	通常システムとの差 [%]
通常システム	21,788	-
予測あり・最適化なし	22,888	+ 5.0
予測あり・最適化あり	20,759	- 4.7

データ量が多い場合の応答時間の計測結果を表 8 に示す．データ量が多い場合は，SPLiT システムのオーバーヘッドが 2.5%，最適化による応答時間短縮が 26.0%と，アクセス数が多い場合と比べ，良い結果となった．SPLiT システムを適用することにより，アクセス数が多い場合，データ量が多い場合のいずれの場合でも性能向上が可能であるが，特にデータ量が多い場合に有用であったといえる．

表 8 データ量が多い場合の応答時間

	応答完了時間 [ms]	通常システムとの差 [%]
通常システム	152,426	-
予測あり・最適化なし	156,275	+ 2.5
予測あり・最適化あり	112,738	- 26.0

4.5 アプリケーションへのコア割当の検証

2.4 節で述べたように，コア割当の際，同じアプリケーションの処理は距離の近いコアで実行されるように割当を行った．効果を調べるために，上述のデータ量が多い場合のシナリオを用いて，2 種類のコア割当について応答時間を計測し，検証を行った．Core i7 の物理コア毎にアプリケーションの割当を行い，同じ物理コア上の 2 つの論理プロセッサでは，同じアプリケーションが実行されるように割り当てた．2 つ目のコア割当（分散割当）では，Core i7 の同じ物理コア上の 2 つの論理プロセッサのうち，1 つに Apache を，他方に MySQL を割り当てた．また，集中割当，分散割当共に，Apache と MySQL それぞれに 4 つずつ論理プロセッサの割当を行った．2 種類のコア割当での応答時間の計測結果を表 9 に示す．

表 9 より，集中割当の場合はコア割当をしない通常システムと比べて 16.4%応答時間が短縮しているのに対し，分散割当では逆に 60.0%応答時間が遅延している．この結果から，アプリケーションへのコア割当が性能に大きく影響し，本研究で採用したアプリケーションへのコア割当アルゴリズムが有用である事が分かった．

表 9 コア割当アルゴリズムによる応答時間

	応答完了時間 [ms]	通常システムとの差 [%]
通常システム	152,426	-
集中割当	127,462	- 16.4
分散割当	243,898	+ 60.0

4.6 スレッドマイグレーションのオーバーヘッド

sched_setaffinity 関数によって発生するスレッドマイグレーションのオーバーヘッドの計測を行った．スレッドの実行コアとして，2 つのコアを交互に指定することで，100 万回のスレッドマイグレーションを発生させ，実行にかかる時間を計測した．指定した 2 つの CPU 番号（CPU 番号 A および CPU 番号 B）と，100 万回のスレッドマイグレーションにかかった時間（実行時間）を表 10 に示す．CPU 番号 A と CPU 番号 B が両方とも 0 の場合，スレッドマイグレーションは発生しない．CPU 番号 A と CPU 番号 B の両方に 0 を指定した場合の実行時間，0.325 秒は sched_setaffinity 関数呼び出しにかかるオーバーヘッドといえる．

表 10 100 万回のスレッドマイグレーションの実行時間

CPU 番号 A	CPU 番号 B	実行時間 [sec]
0	0	0.325

Vol.2010-ARC-189 No.5
Vol.2010-OS-114 No.5
2010/4/21

次に，スレッドマイグレーション後にデータアクセスを行い，キャッシュミスによるオーバーヘッドの計測を行った．スレッドマイグレーション後，char 型の 2^n 個の配列に線形アクセスし，全ての要素をインクリメントする，という方法でキャッシュミスを発生させた．スレッドマイグレーションとデータアクセスを 1000 回繰り返し，実行開始から実行完了までの時間を計測し，各データサイズにおける，1KB あたりのアクセス時間を計算した．計測結果を図 5 に示す．図 5 中の青い線は，常に 0 番の論理プロセッサを実行コアに指定し，スレッドマイグレーションは発生しないが，sched_setaffinity 関数の呼び出しとデータアクセスを行った場合のアクセス時間である．赤い線は，0 番の論理プロセッサと 1 番の論理プロセッサを交互に移動した場合，緑の線は，0 番の論理プロセッサと 2 番の論理プロセッサを交互に移動した場合のアクセス時間である．

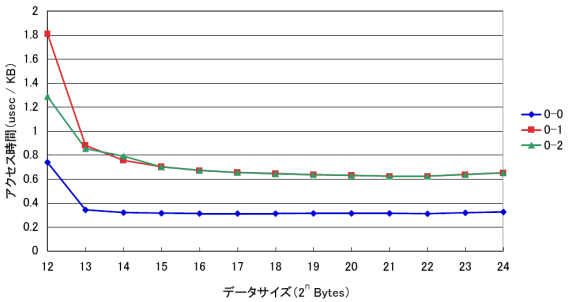

図 5 スレッドマイグレーションにおけるデータアクセスコスト

図 5 より，1KB あたりのデータアクセス時間はデータサイズに関わらず，ほぼ一定である．また，スレッドマイグレーションを発生させずにデータアクセスする場合が一番速く，スレッドマイグレーションが発生する場合は，移動先のコアに関わらず同様のアクセス時間となった．この結果より，アクセスするデータのサイズに関わらず，同じデータにアクセスする処理は，同じコアで行った方が効率的であることがわかる．

5. 関 連 研 究

ハードウェアリソースの共有による性能低下の問題は，Veal ら[22] によって指摘されているように，アドレスバスの通信容量がボトルネックとなっている場合がある．ソフトウェアによって，ハードウェアの能力を向上させることはできないが，アルゴリズムの工夫によって，ハードウェアにかかる負荷を下げられる可能性がある．

スレッドのスケジューリングが不適切である場合は，マルチコアの能力が活かせず，処理能力が低下してしまう．Wentzlaff ら[23] によって指摘されているとおり，マルチコアでは，time sharing（どのスレッドを実行するか）だけでなく，space sharing（どのコアで実行するか）を意識したスケジューリングが必要になる．個々のコアにおける time sharing のスケジューリングに関しても，シングルスレッドのときのように，単に平等性や応答性について考慮するだけでなく，他のコアでどのスレッドが実行されているかを考慮した方が，効率が上がる可能性がある．Gang scheduling[6] のように，通信を行うスレッド同士を同時に実行させることで，通信応答待ちによるブロックを減らし，性能を向上できる場合がある．

Parello[16] は，特定のプロセッサについて，12ヶ月以上かけて手作業でソフトウェアの最適化を行い，それによって収集されたデータをもとに，何が性能のボトルネックになっているかを判断するための決定木を作成した．この手法は，どのようなプログラムにも適用可能であるが，時間がかかる点，他のアーキテクチャには適用できない点が問題となる．

Dryad[9] は，データ並列処理アプリケーションのための分散処理エンジンである．”vertices”と”channels”からなるデータフローグラフを形成し，マルチコアのシングルマシンから，クラスタやデータセンタまで，様々な規模での分散処理に対応する．PC や CPU のスケジューリング，通信やコンピュータの failure の回復，データ通信などの面倒な処理はエンジンが行う．

並列処理用ライブラリとしては，他に OpenMPI[7] をはじめ，MapReduce[4] を単一マシンのマルチコア上で実現させた Phoenix[18] や Threading Building Block[19] など多数存在する．しかし，並列プログラミングの難しさに見合った性能が出るプログラムや環境が多くないことや，ライブラリの多様性によって利用者が分散していることなどから，ライブラリ利用のノウハウが貯まらず，結果として利用者数が伸び悩んでいる．

Meng[13] は，キャッシュにおけるスレッドプライベートなデータの扱いを工夫することで，キャッシュの効率性を向上させた．具体的には，スタックの開始番地がメモリページに合わせて配置されるため，キャッシュミスが多発するという問題に対して，スタックの開始番地をランダムにすることで，キャッシュの競合を低減させている．ただし，スタックの開始番地をランダムにするとメモリ利用効率が落ちるほか，スレッドプライベートなデータの判別やキャッシュの細かい制御が必要となるという問題点がある．

6. 結　　論

本研究では，マルチコア環境におけるリソース配分最適化の困難さを背景に，プロセスの動作予測を行い，予測結果をもとにリソース配分を最適化する手法の提案を行った．動作予測のための情報源として，PMU で取得した性能データと，プログラマによって与えられた処理に関する情報を用いた．カーネルで得られるハードウェア情報と，アプリケーションで得られるソフトウェア情報を組み合わせることで，効率的なリソース配分を目指した．実験の結果，ウェブアプリケーションの性能が最大で 26%以上向上し，提案手法がリソース最適配分に有用であることがわかった．将来課題として，予測精度の向上，他のハードウェア構成への対応，マルチスレッド化の支援などが挙げられる．

参 考 文 献

1) Apache: http://www.apache.org/.
2) Azimi, R., Tam, D.K., Soares, L. and Stumm, M.: Enhancing operating system support for multicore processors by using hardware performance monitoring, *SIGOPS Oper. Syst. Rev.*, Vol.43, No.2, pp.56–65 (2009).
3) Dagum, L. and Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming, *IEEE Comput. Sci. Eng.*, Vol.5, No.1, pp.46–55 (1998).
4) Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *the 6th OSDI*, pp.137–150 (2004).
5) eBay: http://www.ebay.com/.
6) Feitelson, D.G. and Rudolph, L.: Gang Scheduling Performance Benefits for Fine-Grain Synchronization, *Journal of Parallel and Distributed Computing*, Vol.16, pp. 306–318 (1992).
7) Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A. et al.: Open MPI: Goals, concept, and design of a next generation MPI implementation.
8) Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2 (2009).
9) Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks, *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA, ACM, pp.59–72 (2007).
10) JMeter: http://jakarta.apache.org/jmeter/.
11) Koufaty, D. and Marr, D.: Hyperthreading technology in the netburst microarchitecture, *Micro, IEEE*, Vol.23, No.2, pp.56–65 (2003).
12) Lee, P. P.C., Bu, T. and Chandranmenon, G.: A Lock-Free, Cache-Efficient Shared Ring Buffer for Multi-Core Architectures, *ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2009).
13) Meng, J. and Skadron, K.: Avoiding Cache Thrashing due to Private Data Placement in Last-level Cache For Manycore Scaling.
14) Munshi, A.: OpenCL: Parallel Computing on the GPU and CPU, *SIGGRAPH, Tutorial* (2008).
15) MySQL: http://www.mysql.com/.
16) Parello, D., Temam, O., Cohen, A. and Verdun, J.-M.: Towards a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors, *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, IEEE Computer Society, p.15 (2004).
17) Perfmon project: http://perfmon2.sourceforge.net/.
18) Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. and Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems, *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp.13–24 (2007).
19) Reinders, J.: *Intel threading building blocks*, O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007).
20) RUBiS: http://rubis.ow2.org/.
21) vander Zee, A., Courbot, A. and Nakajima, T.: mBrace: action-based performance monitoring of multi-tier web applications, *WDDM '09: Proceedings of the Third Workshop on Dependable Distributed Data Management*, New York, NY, USA, ACM, pp.29–32 (2009).
22) Veal, B. and Foong, A.: Performance scalability of a multi-core web server, *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, New York, NY, USA, ACM, pp.57–66 (2007).
23) Wentzlaff, D. and Agarwal, A.: Factored operating systems (fos): the case for a scalable operating system for multicores, *SIGOPS Oper. Syst. Rev.*, Vol.43, No.2, pp.76–85 (2009).