サーバアプリケーションのスループット向上のための Linux スレッドスケジューリング

近藤拓也[†] 日下部茂[††]

近年，インターネットサービスの需要増大に伴い，サーバアプリケーションに要求される並行処理の度合が増大しており，マルチスレッドの重要性が増している．Linux ではスケジューリングのオーバーヘッド削減のため，カーネル 2.5 から定数オーダーのスケジューリングを可能とする $O(1)$ スケジューラが導入されている．しかしながら，$O(1)$ スケジューラはスレッド間のアドレス空間共有を考慮したスケジューリングは行っておらず，近年特に重要視されているメモリ階層の活用を行っていない．我々は Linux 上において，マルチスレッドアプリケーションのスループット向上を図るスケジューリング手法を提案する．本手法では同一アドレス空間を共有するスレッドを集約実行することで，参照の局所性を活用するとともに，コンテキスト切替えのオーバーヘッドを軽減し，高い同時並行度に対応できる．評価実験により，chatroom サーバや Web サーバといったサーバアプリケーションのスループットが向上することを確認した．

Constant-order Scheduling for Process Aggregation Using Context Information

Takuya Kondoh[†] and Shigeru Kusakabe[††]

Recent demands of Internet service increase the degree of concurrency to be supported by server applications and multi-threading plays an important role in supporting concurrent activities. Thus, recent versions of Linux operating system employs $O(1)$ scheduler to realize efficient thread scheduling. However, $O(1)$ scheduler does not take account of the affinity of threads which share the same address space. We propose a scheduling scheme to enhance throughput of multi-threaded applications on the off-the-shelf Linux platforms. Our scheduler aggregates threads which share the same memory address space so that we can exploit locality of reference, reduce context switch overhead, and support high degree of concurrency. Our performance evaluation shows that our scheduler can improve the throughput and scalability of Internet server applications such as chatroom server and web server.

1. はじめに

近年，インターネットサービスの需要増大に伴い，サーバアプリケーションに要求される並行処理の度合が増大している．chatroom サーバや web サーバといったインターネットサーバアプリケーションでは，同時に並行して多数のクライアントをサポートする必要がある．このため，オペレーティングシステムによって多数のスレッドを効率良くサポートすることは，インターネットコンピューティングにおいて非常に重要な問題である．しかし，ユーザレベルのマルチスレッド機構[1] では，軽量なユーザレベルスレッドを利用して高い同時並行性を備えたサーバを実現できるが，SMT プロセッサ[2] 等のマルチプロセッシングハードウェアを十分に活用できない．また，独自のプログラミングモデル[3] を用いた手法では，アプリケーションを記述し直さねばならない．

本稿では，Linux 上においてマルチスレッド化されたアプリケーションを対象としたスループットを向上させるスレッドスケジューラを提案する☆．Linux はカーネルレベルのマルチスレッド機構を備え，カーネル 2.5 からはスケジューリングのオーバーヘッド低減のため，$O(1)$ スケジューラが採用されている．しかし

† 九州大学大学院 システム情報科学府
Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学大学院 システム情報科学研究院
Graduate School of Information Science and Electrical Engineering, Kyushu University

☆ Linux のスケジューラは，スレッドとプロセスをほぼ同一に扱う．以後，特に区別が重要となる場合を除き，スレッドとプロセスを同じ意味で用いる．

ながら，$O(1)$ スケジューラはスレッド間のアドレス空間の共有を考慮していない．近年，CPU とメモリの駆動周波数の速度差が拡大する傾向にあり，より高いスループットを実現するには，メモリ階層を活用する必要がある．そこで本研究では，スレッド間のアドレス空間共有を考慮に入れ，直前に実行されたスレッドのコンテキストに着目することで参照の局所性の向上を図る．我々は，同一アドレス空間を共有するスレッドを集約的に実行するスケジューラを実装した．このスケジューラは $O(1)$ スケジューラ同様，定数オーダーのスケジューリングを実現するだけでなく，アドレス空間を共有するスレッドを集約する．これにより、参照の局所性を向上するとともに，コンテキスト切替え時のオーバーヘッドを削減し，高い同時並行性を効率良く実現する．

次章では，Linux のスケジューラとスレッドについて述べる．3 章では，本研究の目標であるスレッドの集約の概念について述べ，4 章において，目標を実現するためのスケジューラの構成について述べる．5 章では，実装したスケジューラの性能評価について述べ，6 章では，まとめと今後の課題について述べる．

2. Linux のスケジューラとスレッド

2.1 Linux カーネル 2.4 のスケジューラ

カーネル 2.4 のスケジューラはシステム全体で単一の runqueue を用い，スレッド切替えの際に runqueue を線形に探索し，その都度各スレッドの優先度を算出する．カーネル 2.4 のスケジューラでは，このスレッドの優先度算出時において，実行可能な全スレッドと直前に実行されたスレッドを比較し，アドレス空間を共有するスレッドには優先度に+1 の特典を与えることで，メモリ階層の活用を図っている．**図 1** はその構造を示したものである．しかしながらカーネル 2.4 のスケジューラには，多スレッド動作時の runqueue 線形探索における非効率性や，SMP(Symmetric Multi-Processor) における runqueue のロック競合の増大という問題があった．このため,「Priority Queue スケジューラ」[7]，や「Multi Queue スケジューラ」[6]，「タスク構造体のキャッシュカラーリング」[8],「$O(1)$ スケジューラ」[4] といったものが考案された．なかでも「$O(1)$ スケジューラ」はその名が示すとおり，スレッド切替え時において定数オーダーによるスレッド選択を可能としている．

2.2 $O(1)$ スケジューラ

スケジューリングの負荷低減のため，Linux ではカーネル 2.5 から $O(1)$ スケジューラが採用されている．

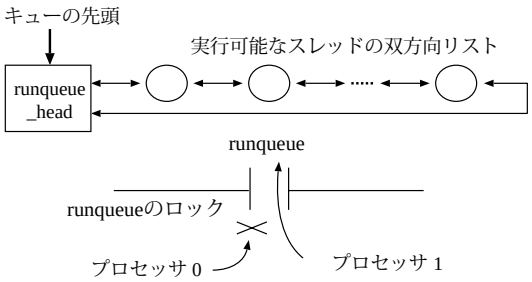

図 1 カーネル 2.4 のスケジューラの構造
Fig. 1 Schematic view of Linux kernel 2.4 scheduler

$O(1)$ スケジューラにおけるスケジューリングのコストは，実行可能な状態にあるスレッドの数に依存せず，常に一定である．また，ロックの競合を抑止するため，プロセッサごとに runqueue を設け，`load_balance()` 関数によってスレッド数を調整して runqueue 間のバランスをとる．**図 2** はその構造を示したものである．$O(1)$ スケジューラの runqueue は，active キューと expired キューという全く同じ構造をした 2 つのキューからなる．active キューでは，実行可能な状態にあり，割り当てられたタイムクォンタムを使いきっていないスレッドを管理する．そしてスレッドがタイムクォンタムを使いきったら，そのスレッドを active キューから expired キューへと移し，expired キューで管理する．active キューが空になったら，active キューと expired キューの役割を交替させる．それぞれのキューはビットマップとビットマップの各ビットと対応する双方向リストの配列で構成されており，ビットマップの各ビットは対応する双方向リストにスレッドが存在するかどうかのフラグとなっている．$O(1)$ スケジューラでは，まずこのビットマップを探索し，最も優先度の高いリストを求める．そして，求めたリストの先頭にあるスレッドを実行する．これによって，実行可能なスレッドの数に依存しない，定数オーダーのスケジューリングを実現している．

しかしながら，プロセッサとメモリの駆動周波数の速度差は拡大する傾向にあるのに対し，$O(1)$ スケジューラはスレッド間のアドレス空間の共有を考慮したスケジューリングを行っていない．高スループットと規模適応性を両立するには，メモリ階層を活用する必要がある．我々は，カーネル 2.4 のスケジューラ同様，次に実行するスレッドを選択する際において，直前に実行されたスレッドのコンテキストに着目することで，局所性の向上を図る．本研究では，スレッド間の親和性を考慮に入れ，アドレス空間を共有するスレッドを集約するスケジューラを開発した．本稿では，こ

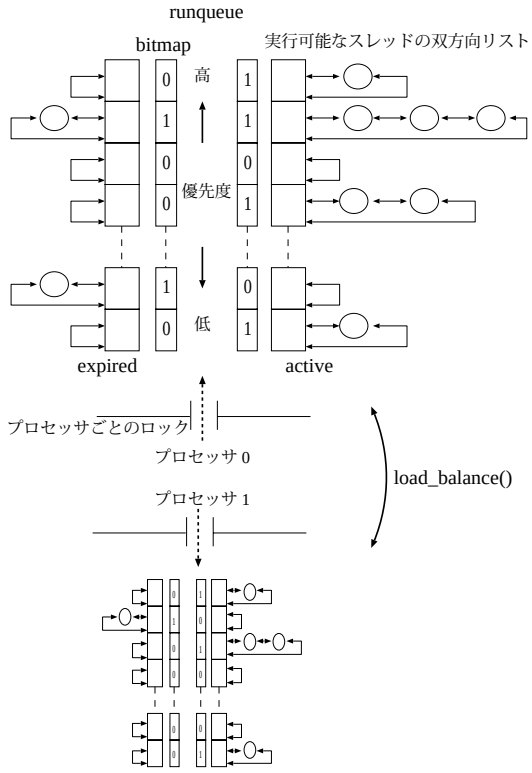

図 2 $O(1)$ スケジューラの構造
Fig. 2 Schematic view of $O(1)$ scheduler

のコンテキストに着目した定数オーダースケジューラを実現するために，Linux カーネル 2.4 の $O(1)$ スケジューラをベースとして用いた．

2.3 Linux のスレッド

スレッドはプログラムの中の一連の制御の流れであり，プロセスと比較して以下のような特徴を備える．

- スレッド生成，削除時の負荷が小さい
- 同一アドレス空間を共有するスレッド間ではコンテキスト切替えが高速
- スレッド間の相互通信が高速

Linux のスレッドは `clone()` システムコールによって実現されている．`fork()` システムコールでは，全てのプロセス資源が複製されるのに対し，この `clone()` システムコールは，親プロセスとプロセス資源を共有する指定が可能である．Linux のスレッドは親プロセスとプロセス資源の一部を共有する特殊な子プロセスとして実装されており，資源の共有をのぞいて Linux のプロセスとほぼ同一である．

2.4 Linux におけるスレッド切替えコストの比較

Linux におけるスレッド切替えコストを比較するために，`sched_yield()` システムコールを用いた実験を行った．実験を行なったシステムの構成を**表 1** に示す．

プロセッサ	Pentium 4 2.6GHz
L1 データキャッシュ	8 KB
L2 キャッシュ	512 KB
メインメモリ	512 MB

表 1 システムの構成
Table 1 system configuration

実験内容	実験に要した時間 (秒)
(A)	9.78
(B)	12.36
(C)	21.31

表 2 実行時間
Table 2 Execution time of each test

`sched_yield()` システムコールを呼び出すと，スレッドは自発的にプロセッサを他のスレッドへと明け渡すことができる．カーネル 2.4 のスケジューラでは，`sched_yield()` は，呼び出したスレッドを `runqueue` の双方向リストの最後尾へと移動することで実現されている．一方，$O(1)$ スケジューラでは，`sched_yield()` は，呼び出したスレッドを `active` キューから `expired` キューへと移動することで実現されている．実験では，次の 3 つの場合における実行時間を計測した．

(A) 単一のスレッドが `sched_yield()` を 2×10^7 回呼び出す
(B) 同一アドレス空間を共有する 2 スレッドがそれぞれ `sched_yield()` を 10^7 回ずつ呼び出す
(C) アドレス空間の異なる 2 スレッドがそれぞれ `sched_yield()` を 10^7 回ずつ呼び出す

厳密には，オペレーティングシステムのカーネルスレッドといった他のスレッドも混在して動作するため，スレッドやアドレス空間の切替えは若干生じるが，(A) の実験では，動作しているスレッドは単体で，スレッドの切替えが生じず，(B) の実験では，スレッドは交互に切替わるが，アドレス空間の切替えは生じず，(C) の実験では，スレッドの切替え時にアドレス空間も切替わる，という動作を想定している．

いずれの場合も `sched_yield()` を呼び出す回数は合計 2×10^7 回である．なお，実験は $O(1)$ スケジューラ上で行った．これは，カーネル 2.4 のスケジューラでは同一アドレス空間のスレッドを優先するため，(C) の実験の場合に必ずしもアドレス空間の異なるスレッドが交互に実行されず，同じスレッドが連続して実行されてしまうことがあるためである．実験結果を**表 2** に示す．

実験では，(B) は (A) と比較しておよそ 1.26 倍の時間を，(C) は (A) と比較して 2.18 倍の時間をその実行に要した．実験 (C) で要した時間は，(A) や (B)

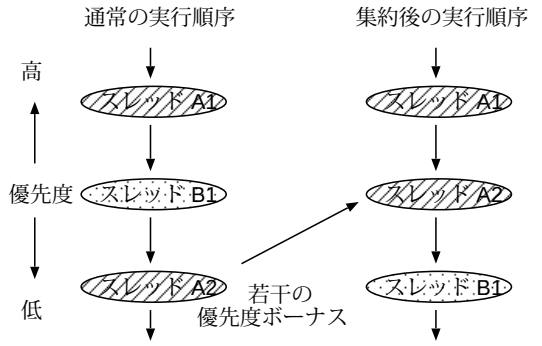

図 3 時間的集約のコンセプト
Fig. 3 Concept of time aggregation

と比較して特に長くなっている．これは，実験 (C) はアドレス空間の切替えが生じ，これに伴って TLB のエントリを無効化するといった処理を行う必要があり，オーバーヘッドが大きくなるからである．

このことから，アドレス空間の切替えも伴うスレッドの切替えでは，アドレス空間の変わらない場合と比較して，スレッドの切替えに伴うコストが大きいことが分かる．したがってカーネル 2.4 のスケジューラ同様に，アドレス空間を共有するスレッドを優先し，アドレス空間の切替えを抑止することで，スレッドの切替えに伴うオーバーヘッドを減少できると考えられる．

3. スレッドの集約

3.1 時間的集約

本研究では $O(1)$ スケジューラを改良し，同一アドレス空間を共有するスレッドを時間的に集約する．具体的には，同一アドレス空間を共有するスレッドの連続的な実行を図る．

アドレス空間を共有するスレッド間では，ユーザ空間のメモリ管理を行なうメモリディスクリプタが共有され，ページテーブルも共通のものとなる．このため，コンテキト切替え時において，メモリディスクリプタを切替える処理が省略でき，コンテキスト切替えのコストを削減できる．また，共有するデータがプロセッサのキャッシュに残っている可能性も高まる．

しかし，先に指摘したように，$O(1)$ スケジューラではアドレス空間を共有するスレッドが優先されることはない．本研究では，直前にプロセッサを割り当てられたスレッドとアドレス空間を共有するスレッドを優先するスケジューリングを定数オーダーで行なえるよう，$O(1)$ スケジューラを改良する．

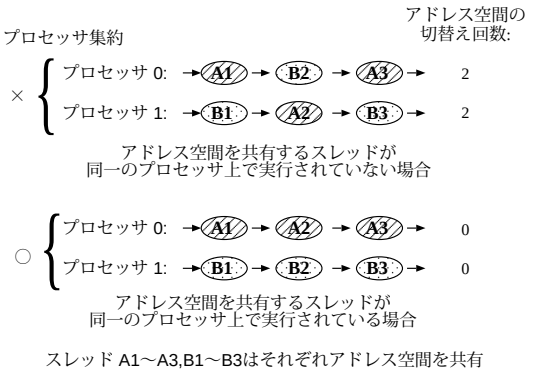

図 4 プロセッサに対する集約
Fig. 4 Concept of processor aggregation

3.2 プロセッサに対する集約

マルチプロセッサ環境を活用できることはマルチスレッドの重要な利点の 1 つである．しかし，アドレス空間を共有する複数のスレッドが異なった runqueue にまたがって存在すると，同一のデータが複数のプロセッサのキャッシュ上にのってしまい，キャッシュの利用効率が低下する可能性がある．また，アドレス空間を共有する複数のスレッドが同時期に異なるプロセッサを割り当てられた場合には，相互のキャッシュの一貫性を維持するための同期処理オーバーヘッドが増加する．そこで，我々はアドレス空間を共有するスレッドの集約をプロセッサに関しても行なう．アドレス空間を共有するスレッドを，できるかぎり各プロセッサごとの同一の runqueue へと集約し，キャッシュの分散を防ぐ．プロセッサに関してこのような集約がなされると，複数のプロセッサが同一データをキャッシュしなくなり，各プロセッサのキャッシュの利用効率が高まる．また，各 runqueue 中のアドレス空間を共有しているスレッド数は増加するので，時間的集約による効果も高まる．

4. 実　　装

4.1 時間的集約の実装

アドレス空間を共有するスレッドを優先し，連続に実行するために，$O(1)$ スケジューラの runqueue とは別に，新たにメモリディスクリプタごとに，アドレス空間を共有するスレッドのための集約用の runqueue を追加する．この runqueue は $O(1)$ スケジューラの runqueue とほぼ同等のもので，定数オーダーによるスレッドの選択が可能であるが，同一のアドレス空間を共有するスレッドのみが管理されている．今，通常の runqueue の中で最も優先度の高いスレッドを X，直前に実行されたスレッドの集約用 runqueue の中で最

も優先度の高いスレッドを Y とおく．$O(1)$ スケジューラの場合，次に実行されるスレッドは当然 X となるが，我々の提案する改良を施したスケジューラでは，次に実行するスレッドを選択する際に次の判定式を用いる．

「X の優先度 $\leq$ Y の優先度+SAME_MM_BONUS」

この式が真であれば Y を，偽であれば X を実行する．この SAME_MM_BONUS は，直前に実行されていたスレッドとアドレス空間を共有するスレッドに対して与える特典にあたるもので，本研究ではこの特典がカーネル 2.4 と同程度になるよう，1 という値を用いている．この値を大きくとれば，それだけ集約の度合を高めることができるが，その分公平さは損なわれることになる．例として，SAME_MM_BONUS=1 とした場合における時間的集約の様子を**図 5** を用いて示す．今，図 5 のように runqueue にスレッドが管理されており，スレッド A にプロセッサが割り当てられているものとする．なお，簡単のため各スレッドは一度プロセッサを割り当てられると終了し，新たなスレッドが実行可能となって runqueue にスレッドが追加されることは無いものとする．この場合，$O(1)$ スケジューラは A-B-C-D-E の順でスレッドにプロセッサを割り当てるが，我々のスケジューラは，A-C-D-B-E という順でプロセッサを割り当てる．$O(1)$ スケジューラでは，A-B 間，B-C 間，D-E 間で 3 回アドレス空間を切替える必要があるのに対し，我々のスケジューラでは，D-B 間の 1 回しかアドレス空間の切替えが生じず，時間的にアドレス空間を共有するスレッドを集約して実行できる．

4.2 プロセッサに対する集約

本研究では，アドレス空間を共有するスレッドをプロセッサに関し集約するために，load_balance() を改良する．この関数は runqueue 間のスレッドの数に偏りが生じた際に，スレッドの数の多い runqueue からスレッドの数の少ない runqueue へとスレッドを移動することで，負荷の調整を行う．我々のスケジューラでは，実行可能な状態にあるアドレス空間を共有するスレッドを集約用の runqueue で管理しているため，この runqueue を用いることで locad_balance() において，比較的低コストでアドレス空間を共有するスレッドを一括して他の runqueue へと移動できる．これによって，プロセッサに対しアドレス空間を共有するスレッドが分散するのを抑止する．

現在，プロセッサに対する集約はまだ実装中であるため，次章の評価においては時間的集約のみを対象とし，シングルプロセッサ環境での評価を行う．

5. 評　　価

実装したスケジューラの評価として，chatroom benchmark[9] と，Apache[10] による HTTP サーバを用いた．評価を行なったシステムの構成は第 2.4 節の表 1 と同様である．

5.1 Chatroom サーバ

chatroom benchmark はチャットルームをシミュレーションする．チャットルーム 1 つあたり 20 人のユーザがおり，TCP ソケット通信によって各ユーザが 1 メッセージあたり 100 バイトのメッセージの送受信を行なう．サーバ側，クライアント側のスレッドそれぞれが，clone() システムコールによって送受信を行なう 2 スレッドを起動するので，1 チャットルームあたり 80 個のスレッドが生成される．評価ではチャットルームの数を 10，1 ユーザあたり 1000 メッセージとし，標準で 1 チャットルームあたり 20 人のユーザの数を 5 から 55 に変化させた時のスループットを計測した．カーネル 2.4 のスケジューラを 1 とした場合における各スケジューラのスループット向上の割合を**表 6** に示す．

実験の結果，我々のスケジューラはカーネル 2.4，カーネル 2.6 いずれの場合においても，$O(1)$ スケジューラよりスループットが高く，1 チャットルームあたりのユーザの数が 20 人の場合において，我々のスケジューラは $O(1)$ スケジューラと比較して，カーネル 2.4 では 22%，カーネル 2.6 では 16%スループットが向上した．1 チャットルームあたりのユーザの数が 5 人の場合，カーネル 2.4 のスケジューラよりカーネル 2.4 の $O(1)$ スケジューラのほうがスループットが低下しているが，これはユーザ数が少ないため，$O(1)$ スケジューラのスレッドの数に依存しないスケジューリングよりも，カーネル 2.4 のスケジューラのアドレス空間を考慮したスケジューリングのほうが効果が大きいからであると考えられる．

5.2 Apache: HTTP サーバ

前節と同様のサーバの構成で，Apache 2.0.48 を用いた HTTP サーバの性能を評価した．Apache は worker というマルチスレッドとマルチプロセスのハイブリッド型ウェブサーバを実装したマルチプロセッシングモジュールを実装している．この worker モジュールを用いると，プロセスごとに一定数のスレッドが生成され，負荷への適応はプロセスの数を増減することにより対応する．Apache は，アイドルなサーバスレッドのプールを常に維持し，それらのスレッドがリクエストに応えるように待機する．worker モジュールを利用す

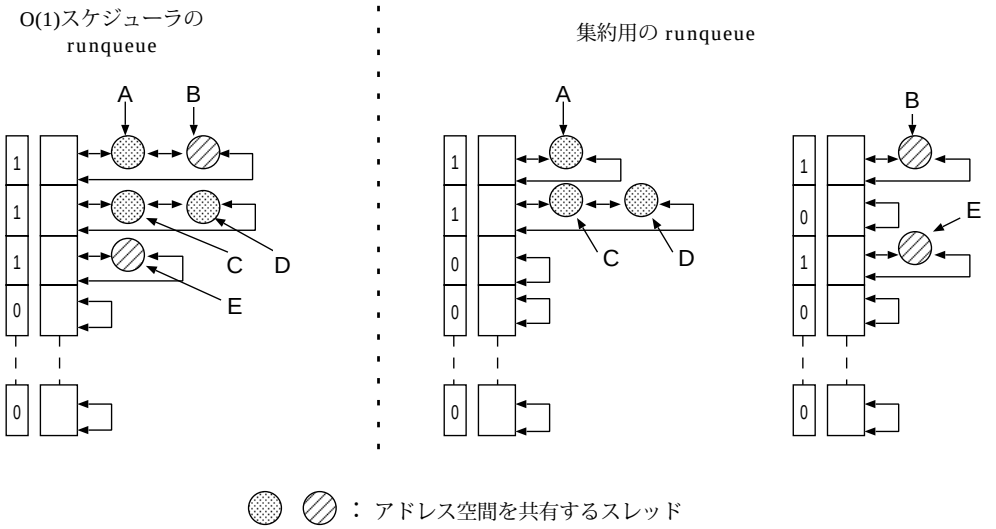

図 5 アドレス空間を共有するスレッドの時間的集約
Fig. 5 Aggregating threads that share the same memory address space

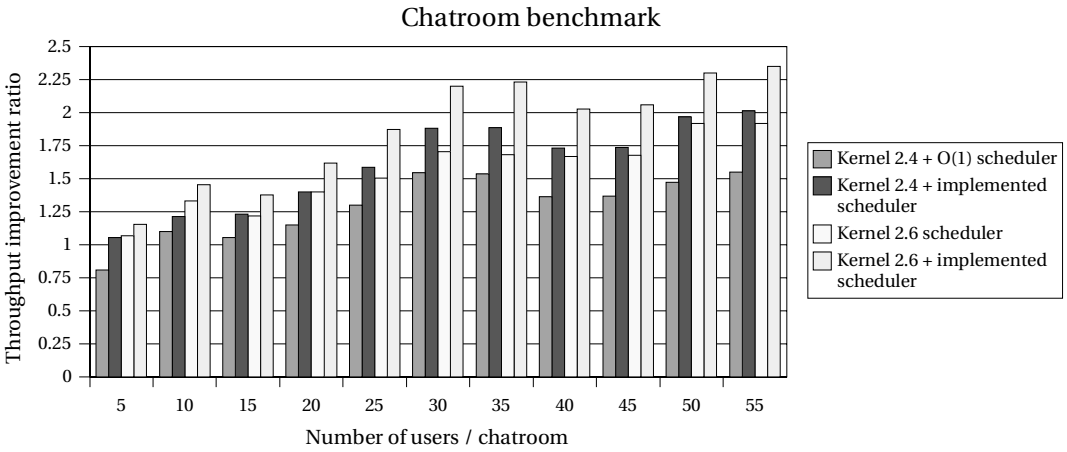

図 6 chatroom benchmark におけるスループット向上の割合
Fig. 6 Throughput improvement ratio of chatroom benchmark

る場合，複数のアドレス空間が生成されるため，アドレス空間の集約による効果が期待できる．Httperf[12] を用いて測定した単位時間当たりのリクエスト数と応答数の関係を**表 7** に，単位時間当たりのリクエスト数とエラーの割合の関係を**表 8** に示す．

実験の結果，単位時間当たりの平均の応答数の最大値はカーネル 2.4 のスケジューラで 4495.2，カーネル 2.4 の $O(1)$ スケジューラで 4510.1，カーネル 2.4 の我々のスケジューラで 4773.7，カーネル 2.6 のスケジューラで 7045.8，カーネル 2.6 の我々のスケジューラで 7315.3 であり，我々のスケジューラでは $O(1)$ スケジューラと比較して，カーネル 2.4 で 5.8%，カーネル 2.6 で 3.8%単位時間あたりの応答数が増加していた．また，単位時間あたりのエラーの割合はリクエストの増加に比例して上昇しているが，我々のスケジューラでは，他のスケジューラよりエラーの発生する割合が低かった．我々のスケジューラでは負荷が低く，runqueue にスレッドが溜まらない状況では，スレッドの集約が意味をなさないが，高負荷時において runqueue にスレッドが溜まりはじめると，スレッドの集約によりキャッシュや TLB を効率良く利用できるため，応答数の向上やエラー発生の抑止に効果があったと考えられる．

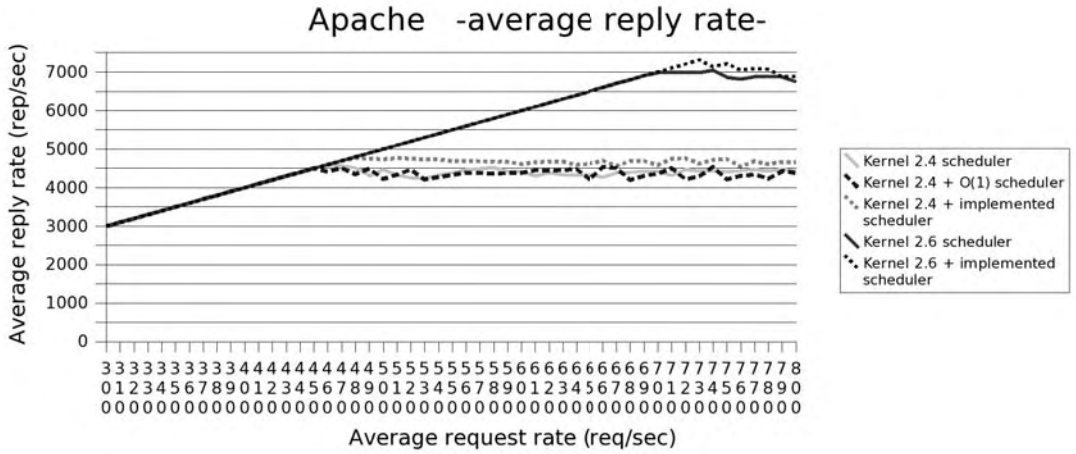

図 7 HTTP サーバの単位時間当たりの平均応答数
Fig. 7 Average reply rate of HTTP server

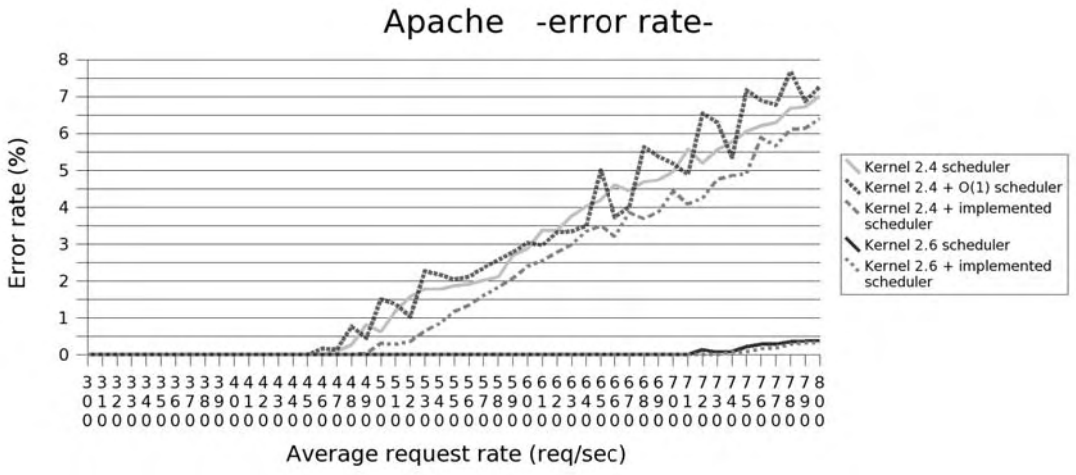

図 8 HTTP サーバのエラーの割合
Fig. 8 Error rate of HTTP server

6. まとめと今後の課題

本研究は，$O(1)$ スケジューラをもとに，アドレス空間を共有するスレッドを連続に実行できるよう集約するスケジューラを実現した．我々のスケジューラでは，アドレス空間を共有するスレッドごとに runqueue をもたせることによって $O(1)$ スケジューラと同様に定数オーダースケジューリングを実現している．キャッシュ利用効率の改善とコンテキスト切替え時のコスト削減によって，カーネル 2.4 において $O(1)$ スケジューラと比較して，chatroom benchmark では 22%スループットが向上し，HTTP サーバのベンチマークでは 5.8%単位時間当たりの応答数が向上し，高い同時並行度を必要とする環境で，効率良くスレッドを実行することができた．

今後は，プロセッサに対する集約の実装を完了し，マルチプロセッサ環境でその有効性を確認する．また，SMT(Simultaneous Multi-Threading) 機能を搭載したプロセッサでは，単一の物理プロセッサ上で複数の論理プロセッサが動作する．論理プロセッサ間ではキャッシュが共有されているので，各論理プロセッサでは実質的に利用可能なキャッシュの量は減少する．本研究のアプローチではキャッシュの集約的利用が期待できるため，この問題の解決に有望と考えられる．今後は SMT 機能を搭載したプロセッサに対しても有効性を確認する．

参 考 文 献

1) Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. ”Capriccio:

Scalable Threads for Internet Services" In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 268-281, 2003.
2) J. Redstone, Joshua Redstone, Hank Levy and Susan Eggers "An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture," ACM SIGPLAN Notices, 35(11), pp. 245-256, 2000.
3) Matt Welsh, David Culler, and Eric Brewer. SEDA: "An Architecture for Well-Conditioned, Scalable Internet Services" In Proceedings of the Eighteenth Symposium on Operating Systems Principles, pp. 230-243, 2001.
4) Ingo Molnar. Ultra-scalable $O(1)$ SMP and UP Scheduler. http://www.uwsg.iu.edu/hypermail/linux/ kernel/0201.0/0810.html
5) J. R. Larus and M. Parkes. "Using Cohort Scheduling to Enhance Server Performance" In Proceedings of Usenix 2002 Technical Conference, pp. 103-114, 2002.
6) Mike Kravetz, Hubertus Franke, Shailabh Nagar and Rajan Ravindram. "Enhancing Linux Scheduler Scalability." Ottawa Linux Symposium 2001. IBM Linux Technology Center.
7) Hubertus Franke and Mike Kravetz. "Implementation of a Priority-Queue Scheduler for Linux." IBM Linux Technology Center, 2000.
8) Shuji Yamamura, Akira Hirai, Mitsuru Sato, Masao Yamamoto, Akira Naruse, and Kouichi Kumon. "Speeding Up Kernel Scheduler by Reducing Cache Misses." In Proceeding of the USENIX Annual Technical Conference 2002 FREENIX Track.
9) Linux Benchmark Suite webpage. http://lbs.sourceforge.net/
10) Apache HTTP Server Project webpage. http://httpd.apache.org/
11) Autobench webpage. http://www.xenoclast.org/autobench/
12) D. Mosberger and T. Tin. "httperf — a tool for measuring web server performance." In Proceedings of WISP '98, Madison, WI, June 1998.