エンタープライズ向けプロセススケジューラの評価および改良

山村 周史　　平井 聡　　久門 耕一
株式会社 富士通研究所
{yamamura, ahirai, kumon}@flab.fujitsu.co.jp
http://www.labs.fujitsu.com/techinfo/linux/

概要

エンタープライズ分野では過負荷時の安定性や CPU 増加に見合う性能スケーラビリティが重要となる。本論文では、これらに大きな影響を与えるプロセススケジューラに的を絞り、その改良と性能評価・分析を行った。我々の改良により、Pentium III Xeon 550MHz を 8 個搭載するサーバ上で、標準カーネルと比較して、Web サーバ性能が最大 23.3%、チャットルームをシミュレーションするベンチマークプログラムでメッセージスループット性能が最大 89.6%向上した。また、最新の O(1) スケジューラについて評価を行い、高負荷時において良好な性能を示すこと、また、その一方で、このスケジューラを実装した場合、ときとして不平等なスケジューリングが行われるという問題点について述べる。加えて、Hyper-Threading 対応の最新 CPU における各スケジューラの評価を行った。その結果、既存スケジューラは、論理 CPU へのプロセス割り当てについて、共有リソース競合への配慮が十分なされていないことが明らかとなった。本論文ではこれに対応するための簡単な改良についても述べる。

1　はじめに

現在、Linux はデスクトップ用途から Web サーバ、データベースシステムをはじめ幅広く利用されており、ここ数年は、エンタープライズ分野への拡大が重要なテーマとなっている。これを推し進めるために、大規模 SMP や NUMA マシンなどをプラットフォームとして、高速 I/O、プロセススケジューラ、障害解析、大容量メモリ管理などといったシステム構成要素についての様々な改良が LSE [1] を中心に続けられている。

中でも、プロセススケジューラは、エンタープライズ用途におけるスケーラビリティの向上をもたらす重要項目として特に注目が集まっており、短期間に様々な変更が試みられている。我々は、今後エンタープライズ分野に対して Linux をより一層普及していく上で、各種スケジューラの性能評価・比較を行い、それらの挙動・問題点を把握することが重要であると考えた。そこで、本研究では、カーネル 2.4.x/2.5.x における各種スケジューラの詳細な性能評価および問題点の分析を行う。

以降、まず **2.** においてエンタープライズ分野に Linux を適用する上でのプロセススケジューラの要件について述べる。続いて **3.** においてカーネル 2.4.x の標準スケジューラの問題点についてまとめ、それを解決するために我々が行った改良手法について詳述する。**4.** で、カーネル 2.5.x において導入された O(1) スケジューラについて詳述するとともに、その性能を評価した結果を報告する。そして、**5.** において Hyper-Threading 機能を持つ最新 CPU を搭載したシステム上での既存スケジューラの挙動について述べる。最後に **6.** で本論文の総括を行う。

2　エンタープライズ向けプロセススケジューラの開発動向

Linux をエンタープライズ分野に適用するために、プロセススケジューラの要件として大きく以下の 2 点が挙げられる。

- CPU 数の増加に見合った性能スケーラビリティ
- 過負荷時（多プロセス動作時）における急激な性能低下の回避

現在まで幅広く利用されているカーネル 2.4.x は、システム上の全実行可能プロセスを単一の runqueue によって管理する。後の節で詳述するように、この実装が上記要件を満たす上で問題となると従来から指摘されてきた [2, 3]。

これに対して、様々なスケジューラの改良 [3, 4] が提案されたが、我々は、独自にメモリバストラフィックの観測やカーネルプロファイリングなどを使用した分析を行うことで、カーネル 2.4.x の標準スケジューラ内部においてキャッシュミスが多発するという新たな問題点の発見とその解決手法を示した [5]。一方、これと同時期に、「O(1) スケジューラ」[6, 7, 8] が提案された。これもまた、他の改良スケジューラと同様に

カーネル 2.4.x における単一 runqueue による実行可能プロセスの管理を回避しようとするものであり、すでに最新（開発）カーネルでの標準スケジューラとなっている。

しかし、O(1) スケジューラについては他の改良スケジューラと異なり、十分な評価が未だなされておらず、実際の運用に際しては、性能向上が得られるかどうか確認するとともにその問題点の洗い出しが必要である。また、近年、Hyper-Threading 機能を持つ最新 CPU が登場し、これが今後のエンタープライズ市場において急速な利用が進むことを考えると、これらスケジューラが、この CPU で構成されたシステムで有効に機能するか、あるいは、最新 CPU の性能を最大限に引き出すために問題となることはないかについても調査する必要がある。

そこで、我々は、Linux をエンタープライズ分野に展開する上で、以下の 3 点について詳しくまとめることが有用であると考えた。

- カーネル 2.4.x の標準スケジューラをメモリアーキテクチャの観点から調査して明らかとなった問題点とそれに対する解決手法
- カーネル 2.5.x で採用された O(1) スケジューラが実際にどれほどの性能向上をもたらすことができるかの確認と問題点の調査
- Hyper-Threading 対応 CPU に対してカーネル 2.4.x/2.5.x の標準スケジューラを適用した場合のシステムの挙動

本論文の以降の節において、これらについて詳しく述べる。

3 カーネル 2.4.x のプロセススケジューラの問題点とその改良

本節では、高負荷な状態でのカーネル 2.4.x の挙動を、メモリアーキテクチャの観点から分析した結果について述べる。まず、標準スケジューラ内部で多発するキャッシュミスによるスケジューリング性能の低下について詳述し、これを解決するために我々が行った改良手法とその評価結果を報告する。

3.1 標準スケジューラの問題点

カーネル 2.4.x は、デスクトップ用途からフロントエンドマシンなどで広く利用されており、エンタープライズ分野においてもその利用が始まりつつある。しかしながら、SMP マシン上では、高負荷時（多プロセス動作時）におけるスケジューラの挙動について、これまで以下のような問題点が指摘されてきた。

1. 実行可能状態にある全てのタスクを単一のキュー (*runqueue*) で管理する。スケジューラは、次に実行すべきプロセスを選択する際、runqueue を全探索する。そのため、実行可能状態にあるプロセスが増加した場合、走査に要する時間が長くなる。
2. SMP システムにおいて、1. の長い走査時間はロック保持時間の増加とロック競合の増大を招く。これにより、CPU 数が増加した場合の性能向上を妨げる。

現在まで、LSE (Linux Scalability Effort) を中心として、上記問題点に関するスケジューラの改善が試みられており、中でも「Multi Queue スケジューラ」(MQ スケジューラ) [3] が、SMP システムにおける性能スケーラビリティをもらたすものとして注目されてきた。MQ スケジューラは、CPU 毎に runqueue を設けて個別にタスク管理を行うと同時に、runqueue 毎にロック変数を設ける。これにより、runqueue を短くして走査時間を短縮するとともに、ロック競合問題を回避して上記 1, 2 の問題を解決する。

このような問題に対して、我々は、独自に開発したメモリバス観測ツール GATES [9] を用いて、高負荷時における Linux システムの挙動をメモリシステムの観点から評価した。GATES は、共有バス上のメモリトランザクションをリアルタイムに観測することができる。その結果、次のような新たな問題点を確認した [5]。カーネル 2.4.x では、システム上の各プロセスの情報を管理するタスク構造体が、常に物理メモリ上の 8KB 境界に配置される。そのため、スケジューラが runqueue を全探策する際、ある特定のキャッシュラインにおいてアクセス競合が多発する。この場合、runqueue 走査中にキャッシュミスが頻発し、結果として走査に多大な時間を消費する。

図 1 は、4-way Pentium Pro 200MHz (2 次キャッシュ 512KB) サーバ上で apache のプロセスを 256 個起動し、高い負荷をかけているときのメモリバストランザクションである。横軸はページ内のオフセットアドレス、縦軸は 1 リクエストあたりのメモリバストランザクション数である。x86 アーキテクチャでは、物理ページサイズは 4KB であるため、そのオフセットアドレスは、`0x0`〜`0xFFF` をとる。図からわかるように、オフセットアドレス `0x30` におけるメモリバストランザクショ

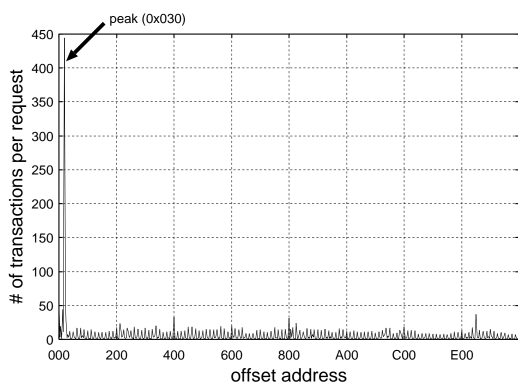


図 1: メモリバストランザクションの測定結果

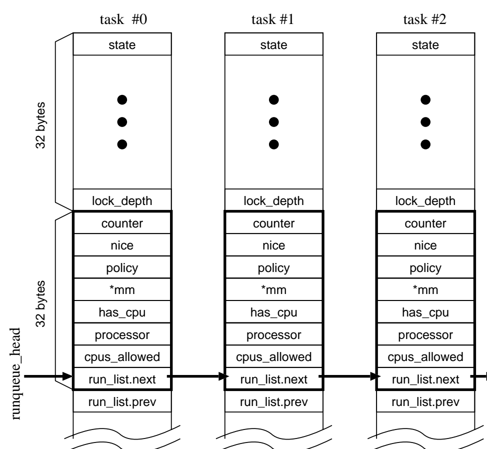


図 2: タスク構造体と runqueue の構造

ン数が突出している。これは、当該アドレスへのアクセスに対してキャッシュミスが多発した結果である。

図 2 にカーネル 2.4.x[1]におけるタスク構造体と runqueue の構造を示す。スケジューラは、次に実行すべきプロセスを選択するために goodness 計算を行う [10]。このときに参照する変数は、8KB 境界に配置されているタスク構造体の先頭から `0x20`〜`0x3F` (32 バイト) の位置に格納されている。従って、当該 32 バイトの物理メモリアドレスは $8\mathrm{KB}(2^{13}) * n + (\mathtt{0x20}\sim\mathtt{0x3F})$ となり、下位 13 ビットが常に同一の値となる。x86 アーキテクチャでは、キャッシュラインとメモリアドレスとのマッピングを下位ビットを用いて決定するため、スケジューラがポインタをたどってタスク構造体に連続してアクセスすると、同一キャッシュライン上で競合する可能性が非常に高くなる。結果、runqueue 走査中にキャッシュメモリが有効に機能せず、図 1 のように特定のメモリアドレスに対して多数のバストランザクションが発生する。

特に、SMP カーネルにおいては、runqueue はロックで保護されている。そのため、キャッシュミスの発生により探索時間が長くなると、それと比例してロック保持時間も長くなる。結果として、CPU 数が増加した場合にロック競合の割合が高くなり、スケーラビリティが得られないという問題が発生していた。これによる性能低下については、次節以降で詳しく述べる。

3.2 キャッシュライン競合の削減によるスケジューラの高速化

3.1 で述べたような、メインメモリ上での配置によって生じるキャッシュコンフリクトの集中という問題を解決する手法として、「カラーリング」がよく知られている [11, 12]。従って、カラーリングをタスク構造体に対して適用することで問題を解決できると考えられる。

しかしながら、現在まで、Linux カーネルにおいてタスク構造体をカラーリングすることはできないと考えられていた。これは、カーネルがカレントプロセスの情報 (例えば `pid` など) を得るときに使用する `CURRENT` マクロ [10] が、「タスク構造体が物理メモリ上で 8KB(2^{13}) 境界に配置されていること」を想定して記述されているからである。従って、タスク構造体のメモリ上での位置をずらしてカラーリングを行うと、カーネルが正常に動作しなくなる。そのため、単純にカラーリングをタスク構造体に対して適用することは困難であり、現在まで行われていなかった。

これに対し我々は `get_current()` にごくわずかな修整を施すことでカラーリングを実現できることに気づいた。図 3 は、修整した `get_current()` である。図からわかるように、太字で示した 1 行のみの修整である。この関数は、キャッシュラインサイズの倍数だけ「ずれた」ベースアドレスを呼び出し元に返す。`get_current()` は、システム内部で頻繁に呼び出される。従って、数回のビット操作命令を追加するだけに留め、オーバヘッドを抑えている[2]。以上のようなわずかな修整でスケジューラ内部で発生していたキャッシュライン競合の問題を解決することが可能となる。

[1] 厳密には、図 2 はカーネル 2.4.4 におけるタスク構造体である。

[2] 実際にカーネルを正常に動作させるためには、これ以外のソースファイルに対しても若干の修正が必要となる。しかし、変更は非常に少なく patch ファイルにして 200 行程度である。

```
static inline struct task_struct * get_current(void)
{
  struct task_struct *current;
  __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
  (unsigned long)current |= ((unsigned long)current >> 10) & 0x00000060;
  return current;
}
```

この1行を追加

図 3: 修正した get_current()

プロセッサ	Pentium III Xeon 550 MHz × 8
2 次キャッシュサイズ	1 MB (4-way set associative)
メインメモリ	1 GB
ネットワークカード	Netgear GA622T (1 GbE) × 4
ディストリビューション	RedHat 7.1 (US)

表 1: サーバの構成

3.3 性能評価

3.3.1 評価環境

評価用サーバとして、富士通製 GRANPOWER 5000 (HS 900) を使用した。サーバ構成を表 1 に示す。また、ベースとなるカーネルのバージョンは 2.4.4 を使用した。

本論文での評価の目的である高負荷時におけるスケジューラの性能について調査するために、ベンチマークプログラムとして以下の 2 つを選択した。

- WebBench 3.0
- Chat micro benchmark

いずれのベンチマークアプリケーションも、サーバ上で多数のサーバプロセスが起動し、クライアントからの多数のリクエストを処理する。そのため、スケジューラの性能について集中的に評価を行うことができる。

【WebBench 3.0】

WebBench 3.0 [13] は、Web サーバ性能の評価を行うベンチマークアプリケーションであり、1 秒あたりに処理したリクエスト数を性能の指標とする。

本実験では、サーバプログラムとして Apache 1.3.19 を用いた。実験中、サーバ上で 256 個のサーバプロセス (*httpd*) を起動する。

サーバに対してリクエストを送信するクライアントマシンとして、28 台の PC/AT を用意した。各クライアントマシンは、OS として Windows NT Workstation 4.0 (SP5) を使用した。WebBench を実行する場合、28 台の PC/AT 上で 256 個のクライアントスレッドを起動し、リクエストを同時に発行する。

【Chat micro benchmark】

Chat ベンチマーク [3, 14] は、上記の WebBench とは異なり、クライアントマシンを必要としない。このベンチマークは、TCP ソケット通信を行う複数のユーザがいるチャットルームをシミュレーションする。各チャットルームには、20 人のユーザがおり、各ユーザは 100 バイトを単位としてメッセージの送受信を行う。サーバ側およびクライアント側にそれぞれ、メッセージを送受信する 2 つのスレッドを起動するので、1 ユーザあたり計 4 スレッド生成される。従って、1 チャットルームあたり 80 個のスレッドが生成されることとなる。

Chat ベンチマークは、チャットルームの数および各ユーザが送受信するメッセージの個数をパラメータとして渡すことができる。本論文の評価では、30 チャットルーム、1 ユーザあたり 300 メッセージに設定した。この場合、2400 プロセス (スレッド) がシステム上に生成される。

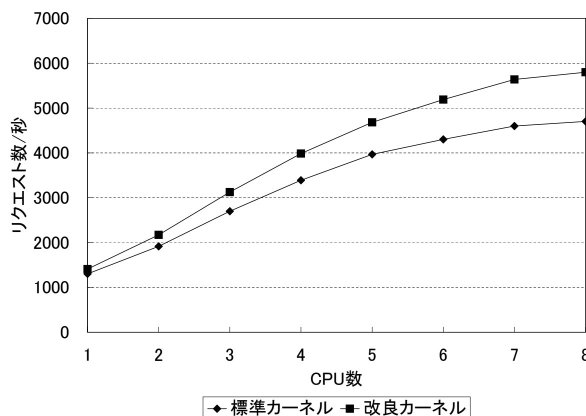


図 4: WebBench 性能

3.3.2 結果

各ベンチマークの測定結果を図 4、5 に示す。これらの図からわかるように、改良カーネルは、WebBench

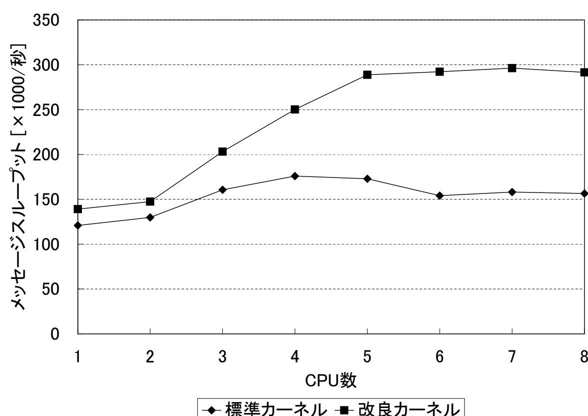


図 5: Chat 性能

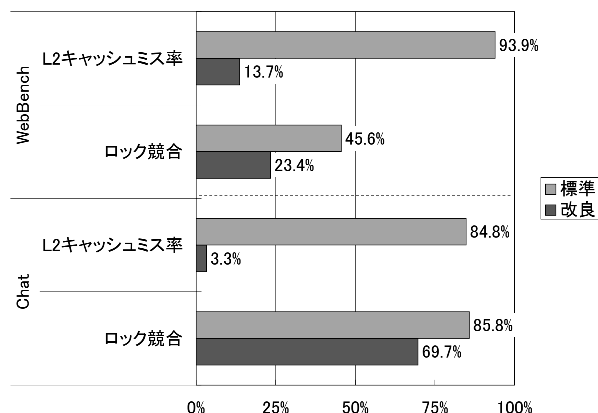


図 6: ロック競合およびキャッシュミス率 (8CPU の場合)

および Chat ともに標準カーネルに比べ大きくスケーラビリティが改善されている。

WebBench では、1 秒あたりに処理したリクエスト数が 8CPU の場合で最大 23.3% 増加している。また、Chat については、6CPU の場合、標準カーネルに対して 89.6% もの性能向上が得られている。標準カーネルは 4CPU まで若干性能が向上し、それより CPU 数が多い場合には性能が劣化しているのに対して、カラーリングしたカーネルでは、6CPU まで性能が大きく向上している。

カラーリングによる本改良手法は、(1) runqueue 走査時のキャッシュミスを減少し、その走査時間を短縮、(2) ロック競合を低減、の 2 つの相乗効果によってシステム全体の性能向上をねらう。実際に、この効果が得られているか確かめるために、8CPU の場合について、以下の 2 項目の計測を行った。

L2 キャッシュミス率 runqueue を走査する `list_for_each` ループ内での L2 キャッシュミス回数および L2 キャッシュへのアクセス回数を測定した。測定には CPU に装備された性能モニタリングカウンタを用いた。L2 キャッシュミス率は、($L2$ キャッシュミス回数)/($L2$ キャッシュアクセス回数) で計算する。

ロック競合率 runqueue を保護するロック変数 (`runqueue_lock`) について、Lockmeter [15] を使用してロックの競合発生率を測定する。

結果を図 6 に示す。この結果から分かるように、標準カーネルでは、85%〜90% 以上と L2 ミス率としては異常な値を示していたものが、改良カーネルでは大幅に改善されている。これにより、runqueue 探策時間が短縮される。同時に、ロック競合率も、WebBench で 45.6% から 23.4%、Chat で 85.8% から 69.7% と大きく減少している。

本節での評価の結果、本改良手法を用いることで標準スケジューラの問題点を解決し、CPU 数の増加に対する性能スケーラビリティを大きく向上できることがわかった。我々の取り組んだスケジューラ内部におけるキャッシュミス多発の問題は、すでに提案されている MQ スケジューラなどの他のスケジューラと着眼点が異なり、それらと互いに直交するものである。従って、本改良手法を他スケジューラと組み合わせることも可能である。

なお、本改良に関する基本的な内容は、USENIX Annual Technical Conference 2002 にて発表を行った。さらに詳細なデータについては、文献 [5] を参照されたい。

4 カーネル 2.5.x のプロセススケジューラの性能評価

前節において、エンタープライズ分野にカーネル 2.4.x を適用する上でのプロセススケジューラの問題点を明らかにし、それを解決するためのカーネルの改良手法を示した。

一方、現在の最新（開発）カーネルであるバージョン 2.5.x においては、標準的なプロセススケジューラとして、「O(1) スケジューラ」が導入されている。O(1) スケジューラは、2002 年初頭に Ingo Molnar が Linux カーネルメーリングリストに投稿したものであり、カーネル 2.4.x における標準スケジューラの問題点を一挙に解決しようとするものである。本節では、O(1) スケジューラを実装したシステムの評価を行い、その性能

を確認する。そして、これまで指摘されていなかった O(1) スケジューラの不平等スケジューリングの問題について述べ、これについて実験および考察を行う。

4.1 O(1) スケジューラの特徴

O(1) スケジューラの概念図を図 7 に示す。O(1) スケジューラは、主として以下のような特徴を有している。

(a) 優先度付き待ち行列による実行可能プロセス管理

実行可能プロセスは、配列構造とリスト構造とを組み合わせて実装された優先度付き待ち行列で管理される。各配列要素は、同一優先度からなるプロセスのリストである。

(b) runqueue_lock の分散

上記 (a) の 2 配列が各 CPU 毎に用意されており、それぞれ独立したロック変数によって保護されている。

(c) CPU 間でのプロセス移動

各 CPU 間で分散された runqueue の間で、適切に負荷バランスを保つための関数 `load_balance()` が実装されている。また、この関数は、キャッシュメモリの利用効率を考慮し、CPU 間のプロセス移動を抑えるように作用する。

O(1) スケジューラでは、図 7 に示すように、2 つの優先度付き待ち行列で runqueue を構成している。それぞれ “active”, “expired” と呼ばれ、active は、実行中のプロセスを管理し、他方 expired は、クォンタム（割り当てられた CPU 持ち時間）を使い切ったプロセスを保持している。クォンタムを使い切ったプロセスは、active から expired に移動される。active が空になった場合は、これらの配列をスワップする。各配列には、プロセスキューの有無を示すビットマップが用意されており、これを参照することで、最も高い優先度を有するプロセスリストの位置（active 配列内のインデックス）を高速に決定することができる。これによって、標準スケジューラのようにプロセスリストを線形探索することなく、次に CPU に割り当てるべきプロセスを直接参照することができる[3]。

また、各 CPU 間で負荷バランスを取るために、`load_balance()` が用意されている。この関数は、自身がアイドル状態にある場合やタイマ割り込みを契機として一定間隔で呼び出され、CPU 間の負荷バランスを保つように他の CPU の runqueue からプロセスを奪う処理を行う。

[3]「計算に要するオーダが 1 である」という名前の由来

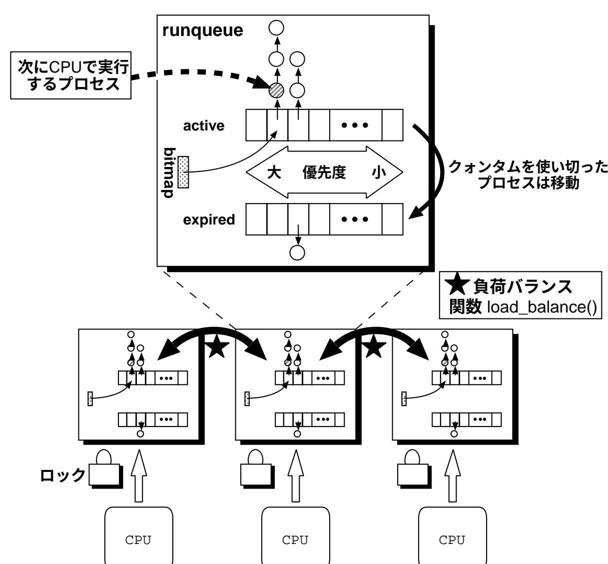


図 7: O(1) スケジューラ概念図

加えて、O(1) スケジューラではキャッシュに転送されたデータの有効利用を図るために、できる限り CPU 間でのプロセスの移動を抑えるようにしている。これにより、キャッシュのヒット率が向上しプロセスの実行時間が短縮されることが期待できる。

以上のような工夫により、O(1) スケジューラは、**3.1** で述べた 2.4.x 標準スケジューラの問題であった単一 runqueue の線形探策とロック競合の問題の解決を図っている。これにより、SMP マシンにおけるスケーラビリティ向上をねらう。

4.2 性能評価

4.2.1 実験環境

O(1) スケジューラの効果を確認するために、**3.** で使用した実験環境およびベンチマークプログラム（WebBench, Chat）を使用して性能評価を行った。ただし、本節の評価では、評価に用いるカーネルのバージョンを 2.4.18 とし、以下の 3 つのカーネルの性能を測定した。

1. 標準カーネル
2. 標準カーネル + O(1) スケジューラパッチ
3. 標準カーネル + Multi Queue スケジューラパッチ

サーバの CPU 数を 1 台から 8 台まで変化させ、これら 3 つのカーネルの性能を測定した。

表 2: WebBench の性能およびスピンロック関数の実行比率

	スケジューラ	1 CPU	2 CPU	4 CPU	8 CPU
性能 [リクエスト/秒]	標準スケジューラ	1,332 (1.00)	2,038 (1.53)	3,613 (2.71)	5,074 (3.81)
	O(1) スケジューラ	1,401 (1.05)	2,314 (1.74)	4,210 (3.16)	6,042 (4.54)
	Multi Queue スケジューラ	1,319 (0.99)	2,192 (1.65)	4,054 (3.04)	6,019 (4.52)
スピンロック	標準スケジューラ	–	1.62 %	6.14 %	18.81 %
	O(1) スケジューラ	–	0.00 %	0.00 %	0.00 %
	Multi Queue スケジューラ	–	0.07 %	0.04 %	0.03 %

表 3: Chat の性能およびスピンロック関数の実行比率

	スケジューラ	1 CPU	2 CPU	4 CPU	8 CPU
性能 [$\times 10^3$ メッセージ/秒]	標準スケジューラ	126 (1.00)	136 (1.08)	201 (1.60)	183 (1.45)
	O(1) スケジューラ	119 (0.94)	144 (1.15)	282 (2.25)	392 (3.11)
	Multi Queue スケジューラ	121 (0.97)	145 (1.15)	269 (2.14)	362 (2.88)
スピンロック	標準スケジューラ	–	5.51 %	15.02 %	31.25 %
	O(1) スケジューラ	–	0.03 %	0.03 %	0.06 %
	Multi Queue スケジューラ	–	1.25 %	1.65 %	1.56 %

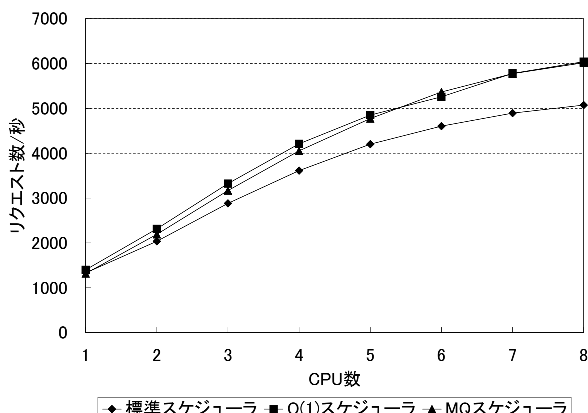


図 8: O(1) スケジューラの WebBench 性能測定結果

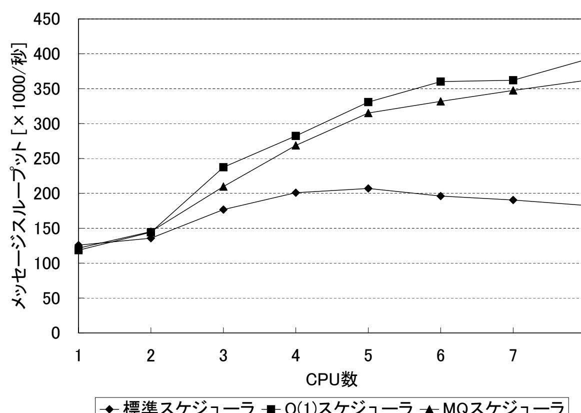


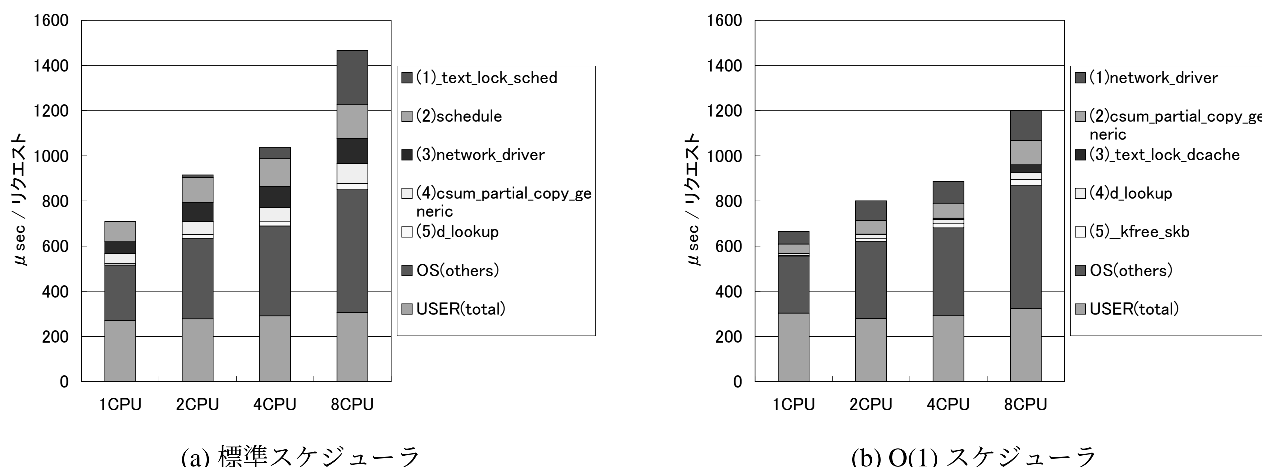
図 9: O(1) スケジューラの Chat 性能測定結果

4.2.2 結果

WebBench の測定結果を図 8、および表 2 に示す。また、Chat の測定結果を図 9、および表 3 に示す。表には、高負荷時におけるスケジューラの性能を大きく左右するスケジューラ内部におけるスピンロック関数 (`_text_lock_sched()`) について、その処理時間が実行時間全体に占める割合もあわせて示した。なお、表中カッコ内の数値は、1 CPU 構成のサーバ・標準スケジューラ上での性能を 1 とした場合の性能比である。

WebBench については、グラフからわかるように、O(1) スケジューラは標準スケジューラに比べ性能が改善されている。1 秒あたりに処理したリクエスト数は、8 CPU の場合で最大 19.0% 増加している。表からわかるように、CPU 数が増加するにしたがって O(1) スケジューラによる性能向上比は大きくなっている。スピンロック処理時間の割合も、標準スケジューラにおいて 8 CPU 時 18.81% であったものが O(1) スケジューラではほぼ 0% にまで抑えられている。しかし、Multi Queue (MQ) スケジューラと比較すると、ほとんど性能差がみられない。WebBench では、MQ スケジューラを用いた場合でも十分にロック競合が減少しており、これら 2 つのスケジューラは同等の性能を示している。

Chat については、グラフから分かるように、O(1) スケジューラは標準スケジューラと比較してスケーラビリティが大幅に向上している。標準カーネルでは、5 CPU 以降性能が劣化するのに対して、O(1) スケジューラでは 8 CPU まで性能が向上している。このとき、標



(a) 標準スケジューラ (b) O(1) スケジューラ

図 10: WebBench 実行時の OS 関数プロファイル

準カーネルに対して 114% の性能向上が得られている。

runqueue に並ぶプロセスの数が、WebBench の場合で最大 256 程度であるのに対して、Chat の場合は 1000 プロセス以上とはるかに多い。MQ スケジューラの場合、CPU 毎にロック変数の分割は行っているが、runqueue 内を線形探策することについては標準スケジューラと同じである。そのため、O(1) スケジューラの特徴である高速なプロセス探策とロック競合の低減とがより有効に機能し、Chat のように非常に多くのプロセスが生成される場合には最も良い結果を示している。

4.2.3 カーネルプロファイリングによる分析

O(1) スケジューラの効果をさらに詳しく調査するために、WebBench 実行中のカーネルプロファイリングを採取し分析を行った。

結果を図 10 に示す。グラフには、OS 内部での処理時間の多い上位 5 関数、その他の OS 処理部分、およびユーザプログラム実行時間に分割して表示している。ただし、“`network_driver`” は、ネットワークドライバ内部での処理時間を示しており厳密には 1 関数ではない点に注意されたい。

図 10 (a) 標準カーネルでは CPU 数が 1〜4 CPU の場合は、`schedule()` 関数の占める割合が最も大きい。これは、標準スケジューラが長い runqueue を線形探策するために多くの時間を消費しているためである。8 CPU の場合は、`_text_lock_sched()` 関数の占める割合が急激に増加している。これは、単一の runqueue_lock における競合によるものである。

一方、図 10 (b) O(1) スケジューラでは、標準スケジューラで問題となっていた上記関数は上位には現れない。WebBench による評価では、性能ボトルネックは、ネットワークパケット処理 (`network_driver` および `csum_partial_copy_generic`) となっている。またロック変数についても、スケジューリングにおける競合よりも、dcache_lock（ディレクトリエントリキャッシュ）における競合が問題となっており、スケジューリングにおける問題は解決できていると言える。

4.3 O(1) スケジューラの問題点

前節までで、多数個のプロセスが生成される場合について、O(1) スケジューラの性能評価・考察を行い、このスケジューラが他と比較して高い性能を示すことを確認した。しかし、その一方で、我々は様々な状況での実験を重ねていく過程で、O(1) スケジューラに問題点があることに気づいた。

O(1) スケジューラを利用した場合、SMP マシン上では不平等なプロセスのスケジューリングが行われることがある。このような現象は、標準スケジューラでは見られない。本節では、この問題について簡単な実験を行った結果について報告する。

【実験方法】

実験では、前節までの評価に用いた環境（8-way Pentium III サーバ、2.4.18 カーネル）と同じものを使用した。実験を簡単にするため、これを 4 CPU 構成で起動し、100×10^6 回の空ループを実行するプロセスを 5 つ同時に起動する。各プロセスの実行時間とともに、実行中に各プロセスがどの CPU 上で実行されているか（カレント CPU）を一定間隔でサンプリング測定する。

【結果・考察】

結果を表 4 に示す。この表は、実行に要した実時間、および各プロセスがどの CPU 上で実行時間の何% を消費したかを示したものである。また、O(1) スケジュー

表 4: O(1) スケジューラによる不平等なプロセススケジューリング（経過時間と CPU 使用率）

	プロセス	経過時間 [秒]	CPU0 [%]	CPU1 [%]	CPU2 [%]	CPU3 [%]
標準スケジューラ	0	5.89	24.5	28.6	28.6	18.4
	1	5.45	12.2	4.1	24.5	59.2
	2	5.93	22.5	16.3	28.6	32.7
	3	5.62	25.3	52.2	0.0	22.4
	4	5.17	27.0	25.2	25.4	22.4
	プロセス	経過時間 [秒]	CPU0 [%]	CPU1 [%]	CPU2 [%]	CPU3 [%]
O(1) スケジューラ	0	6.89	49.0	0.0	0.0	51.0
	1	6.80	0.0	0.0	0.0	100.0
	2	4.55	0.0	0.0	100.0	0.0
	3	4.55	0.0	100.0	0.0	0.0
	4	4.55	100.0	0.0	0.0	0.0

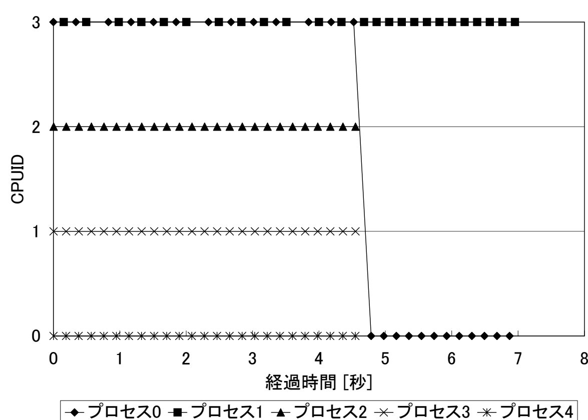


図 11: O(1) スケジューラによる各プロセスのスケジューリング

ラについて、横軸を経過時間、縦軸を CPUID として各プロセスのカレント CPU をプロットしたものを図 11 に示す。

表 4 からわかるように、標準スケジューラに比べ、O(1) スケジューラは各プロセスのスケジューリングに大きな偏りが見られる。標準スケジューラでは各プロセスがほぼ均等に複数の CPU 上で動作しているのに対し、O(1) スケジューラではある特定の CPU でプロセスが動作し続ける。5 プロセスの実行時間について、平均値に対する分散の比率を計算すると、標準スケジューラが 3.8%、O(1) スケジューラが 30.5% となり、実行時間に大きなばらつきがある。図 11 からわかるように、プロセス 2, 3, 4 はそれぞれ 1 台づつ CPU を占有しているのに対して、プロセス 0, 1 は 1 台の CPU を交互に使用していることがわかる。このような不平等なスケジューリングが行われるために、各プロセスの実行時間に大きな差が生じる。本論文では詳しく述べないが、このような少数のプロセスが起動される場合に限らず、より多くのプロセスが生成される場合でもこの問題は発生する。

このような不平等なスケジューリングについては、例えば、ターンアラウンドタイムが重要視されるようなシステムで O(1) スケジューラを使用する場合に注意が必要であるし、また、複数のプロセスが同期を取りながら並列動作する場合には最長実行時間のプロセスによって並列プログラムの実効性能が律速されてしまうことも考えられる。実際には、**4.1** で述べたように、キャッシュメモリの利用効率を考慮して CPU 間のプロセス移動を抑える、という O(1) スケジューラの実装が大きく影響していると考えられるので、これを解決するために、CPU 間でのプロセス移動を行う `load_balance()` の改善が今後必要であると言える。

5 プロセススケジューラの最新 CPU への適用とその問題点

3.、**4.** において、カーネル 2.4.x/2.5.x におけるプロセススケジューラの性能評価･比較を行った。本節では、これらのプロセススケジューラを、近年登場した Hyper-Threading 機能を持つ Intel Xeon プロセッサ [16, 17]（以下 Xeon と略記）に対して適用する場合について調査を行う。Hyper-Threading 機能を利用した場合、実際にシステム性能の向上が得られるかを確認するとともに、既存のプロセススケジューラが Hyper-Threading 機能を有効に利用できていない点について明らかにする。そして、この問題を解決するために我々が行った改良について詳述し、簡単な実験を行った結果について述べる

以下、評価にあたって、まず、Hyper-Threading 機能の概要を説明し、既存のプロセススケジューラを適用

する上で懸念される点について述べる。

5.1 Hyper-Threading 機能の概要

Hyper-Threading 機能とは、一般に SMT (Simultaneous Multithreading) [18, 19] と呼ばれる CPU アーキテクチャの 1 実装である。このアーキテクチャは、複数の命令流（スレッド）を 1 つの物理 CPU 内部で同時に実行することで、CPU 内部の演算器（ALU や浮動小数点演算器など）の稼働率を上げ、CPU のスループット性能を向上しようとするものである。

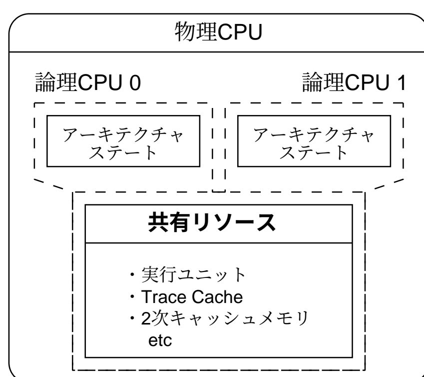


図 12: Hyper-Threading 機能を搭載した CPU の概念図

表 5: 各種計算資源の共有/独立

	共有リソース	独立リソース
レジスタ (EIP, etc)		○
実行ユニット	○	
命令 TLB		○
データ TLB	○	
実行トレースキャッシュ	○	
1 次データキャッシュ	○	
2 次キャッシュ	○	

Xeon では、図 12 のように 1 つの物理 CPU 内部に 2 つの論理 CPU が存在している。論理 CPU は、独立したアーキテクチャステートと論理 CPU 間で共有された演算リソースから成っている。アーキテクチャステートとは、単一スレッドを実行するために保持しておく必要がある CPU の実行環境（各種レジスタの値など）であり、一方、共有リソースとしては、実行ユニットや 2 次キャッシュメモリ、実行トレースキャッシュなどがある。表 5 に各演算リソースの論理 CPU 間での共有/独立をまとめた。

この CPU の本来のシステム性能を引き出すためには、物理 CPU 内部で共有されている演算リソースの競合を考慮したスケジューリングが必要となる。具体的には、ある物理 CPU 上で 2 つのプロセスが同時に走行している場合、各プロセスは実質的に半分の計算資源しか使用できず、1 プロセスのみで走行する場合に比べて性能が大きく低下してしまう。従って、Xeon で構成された SMP システムにおいては、プロセスをスケジューリングする場合には、「可能な限り 1 つの物理プロセッサにつき 1 プロセスが走行するようにスケジューリングを行う」ことが重要となる。

しかしながら、Linux カーネルはこの CPU を 2 CPU が搭載された SMP システムと認識し[4]、従来の SMP カーネルにおけるスケジューリングポリシーを単純に適用する。従って、例えば、システム上で少数のプロセスを起動した場合に、それらが同一物理 CPU に対して割り当てられる場合がある。そのため、同一のプログラムを実行しているにもかかわらず、スケジューリングによって実行時間に大きなばらつきが出る可能性がある。以降の節では、この CPU を複数個搭載したサーバを使用して、Hyper-Threading 機能による性能向上を確認するとともに、上記問題点について検討を行う。

5.2 Hyper-Threading 機能の評価

5.2.1 評価環境

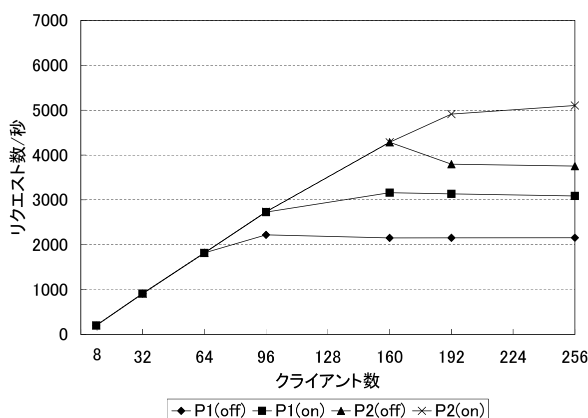
Xeon を 2 台搭載したサーバ上で既存スケジューラの性能について、**4.** までの評価と同じく WebBench を用いて評価を行った。サーバ構成を表 6 に示す。カーネルのバージョンは 2.4.18 を基本とし、標準スケジューラおよび O(1) スケジューラの 2 種類について評価を行う[5]。サーバ上で Web サーバプロセス (httpd) を 256 個起動し、28 台の PC/AT 上で起動されたクライアントスレッドからのリクエストを処理する。実験では、クラインアントスレッドの数を 1〜256 に変化させ、サーバのリクエスト処理性能を測定する。

5.2.2 結果

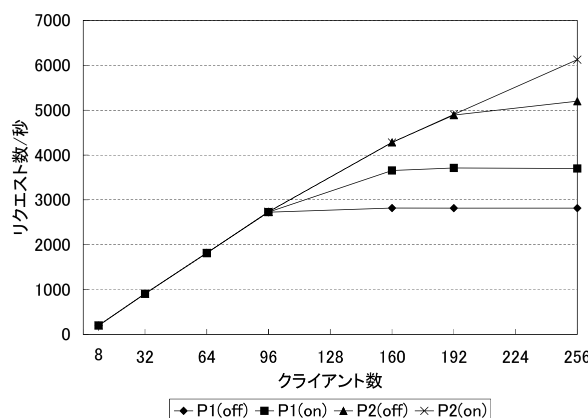
測定結果を図 13 に示す。図中 $Px(on/off)$ は、稼働している物理プロセッサが x 個であり、カッコ内において Hyper-Threading 機能の on または off を示して

[4]厳密には、最新のカーネルは CPU の cpuid をチェックして物理あるいは論理 CPU を区別している

[5]Xeon を複数搭載したサーバでカーネル 2.4.18 を使用した場合、ある CPU に割り込みが集中するという問題がある。実験に使用したカーネルではこれを修正するパッチを別途適用している



(a) 標準スケジューラ



(b) O(1) スケジューラ

図 13: WebBench 性能

表 6: 実験に用いたサーバの構成

プロセッサ	Intel Xeon 2.00 GHz × 2
2 次キャッシュサイズ	512 KB
メインメモリ	512 MB
ネットワークカード	Intel EtherExpress Pro/100 × 2 Netgear GA622T (1GbE) × 2
ディストリビューション	RedHat 7.2（US 版）
カーネルバージョン	2.4.18 2.4.18 + O(1) パッチ

いる。また、表 7 にピーク性能をまとめたものを示す。

結果からわかるように、標準スケジューラ、O(1) スケジューラともに Hyper-Threading 機能を on にした場合に性能が向上している。ピーク性能の向上比は、1CPU の場合 30〜40%、2CPU の場合 18〜19%程度である。標準スケジューラの場合、P2 (off) のときにクライアント数が 192 以上で性能が劣化している。これは、負荷が 100%になってリクエストが処理しきれず、httpd プロセスが runqueue に並び始めたためである。このように、標準スケジューラについては SMP の場合と同様の問題は発生しているものの、いずれのスケジューラについても Hyper-Threading 機能を使用することで性能が向上している。

スケジューラの観点からすると、WebBench は以下のような特徴を有している。

- CPU の個数を越える多数個のプロセスが常に実行可能状態にある（runqueue に並んでいる）。
- 複数のサーバプロセス（httpd）が各々独立してリクエストを処理しており、互いに同期を取るようなことはほとんどない。

この種のアプリケーションの場合、リソース競合の問題について特に考慮せずとも既存スケジューラを適用することで Hyper-Threading 機能を活用できていると言える。

表 7: Hyper-Threading 機能によるピーク性能の向上（カッコ内は off の場合に対する性能比）

物理 CPU 数	スケジューラ	Hyper-Threading	
		off	on
1CPU	標準	2220	3159 (1.42)
	O(1)	2818	3711 (1.32)
2CPU	標準	4284	5104 (1.19)
	O(1)	5201	6127 (1.18)

5.3 Hyper-Threading 機能を活用するためのスケジューラ改良

5.3.1 既存スケジューラの問題点とその改良

5.2 で述べたように、多プロセス動作時においては、従来のスケジューラを用いても大きな問題は発生していなかった。しかし、**5.1** で述べたように、Hyper-Threading 機能を持つ CPU で構成された SMP マシンにおいては、プロセスをスケジューリングする際に「可能な限り 1 つの物理プロセッサにつき 1 プロセスが走行するようにスケジューリングを行う」ことが重要となる。しかしながら、いずれの既存スケジューラも上記の点を十分に考慮できていない。本節では、安定版カーネルとして特に改良が急がれる標準スケジューラについて、まずこの問題点の検討および改良を行う。

標準スケジューラにおいて、具体的に以下の 2 点が問題となる。

(1) schedule() 関数における優先度計算

あるプロセスの優先度 (goodness) を計算する場合、スケジューリングを行う CPU と当該プロセスが以前走行していた CPU とが同じである場合、一定の優先権を与える。そのため、2 つのプロセスが同一物理 CPU 内部の 2 つの論理 CPU で走行し続けてしまうことがある。

(2) reschedule_idle() 関数における CPU の選択

reschedule_idle() 関数は、プロセスの起床時に呼び出される。このとき、当該プロセスの以前走行していた CPU が idle 状態である場合には、その CPU に対して当該プロセスを割り当てる。そのため、図 14 のように、一方の物理 CPU が idle 状態にあるにもかかわらず割り当てられない場合がある。

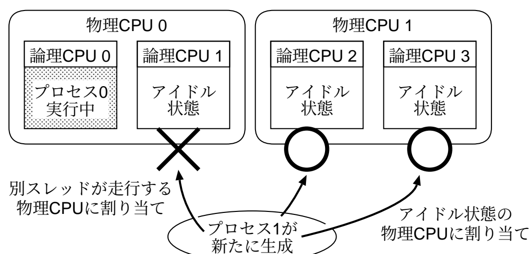


図 14: reschedule_idle() 関数における誤ったプロセススケジューリング

そこで、我々は、この問題点を解決するために標準スケジューラの改良を行った。実際には、schedule() 関数内部と reschedule_idle() 関数にそれぞれ以下のような修整を加えた。

(1) schedule() 関数に対する修整

関数内部において、全プロセスの優先度計算終了後、以下の条件をチェックする。

- runqueue に並んだプロセスがすべてクォンタムを使い切った。
- idle 状態にある物理 CPU が存在する。

この場合、idle している物理 CPU にカレントプロセスを移動する。

(2) reschedule_idle() 関数に対する修整

idle 状態にある物理 CPU が存在するかどうかを他の条件に優先してチェックする。存在すれば、（当該プロセスが以前走行していた CPU と異なっていても）その CPU に当該プロセスを割り当てる。

5.3.2 実験

前節での改良を確かめるために簡単な実験を行った。

【実験方法】

実験では、実行時間の異なるプロセスを 2 つずつ起動し、全プロセスが実行を完了するまでの時間を測定する。各プロセスは、100×10^6 回、500×10^6 回の単純ループを実行する。ループ内部では、事前に確保された 4KB のメモリ領域をサイクリックに read するだけである。これを 10 回繰り返す。各プロセスの実行時間を測定すると同時に、実行中に各プロセスがどの CPU 上で実行されていたか（カレント CPU）を一定間隔でサンプリング測定する。

【結果・考察】

結果を表 8 に示す。また、図 15 に横軸を経過時間、縦軸を論理 CPU ID として、各プロセス実行中のカレント CPU の測定結果をプロットしたものを示す。

表 8: 4 つのプロセスが全て終了するまでの時間

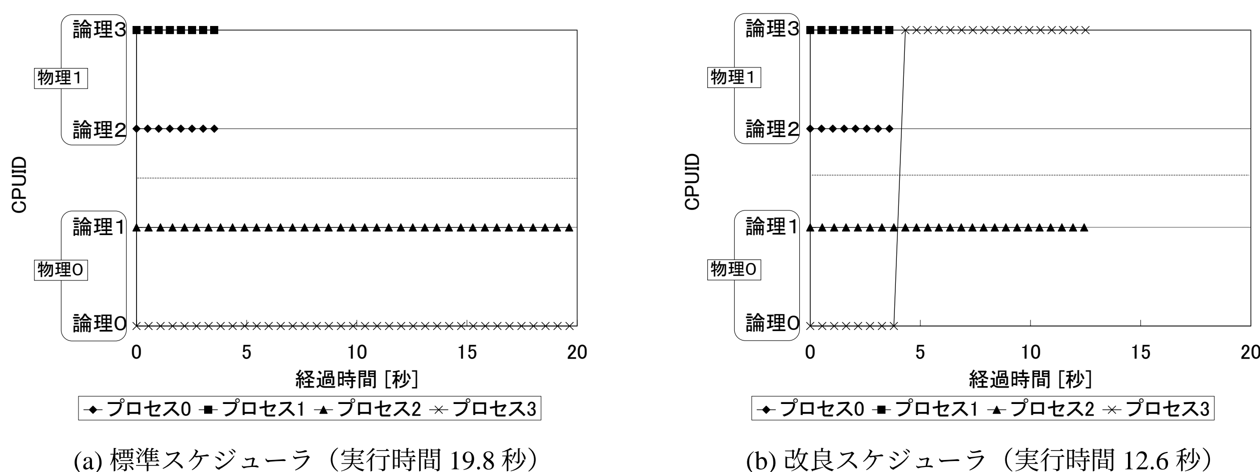
	最短 [秒]	最長 [秒]	平均 [秒]
標準スケジューラ	12.6	19.8	15.5
改良スケジューラ	12.6	12.7	12.6

これらからわかるように、標準スケジューラでは、最短時間と最長時間との差が非常に大きい。これは、図 15 (a) のように実行時間の長いプロセスが同一物理 CPU 内部で動作し続けてしまうためである。一方、最短の時間で終了する場合は、実行時間の長いプロセスが異なる物理 CPU で動作した場合である。標準スケジューラの場合、これらいずれかの状況がランダムに発生するために、動作が安定しない。

一方の改良スケジューラでは、図 15 (b) のように、一方の物理 CPU が idle 状態になったときにプロセスの移動が行われている。これにより、常に最良の状態で 4 つのプロセスの実行が行われる。最長の実行時間について、その性能差は 1.5 倍以上になる。

この実験は、単純なループ処理を行うプログラムを用いた一例であるが、実際の並列アプリケーションや、実稼働するサーバマシンへのバッチジョブの投入などにおいてもこのような状況が発生することは十分に考えられる。また、互いに同期をとりながら動作する並列プログラムなどの場合、標準スケジューラでは最も実行時間の長いプロセスでプログラム全体の実行時間が抑えられてしまうことも考えられる。この場合、今回の改良が有効であると予想できる。

本節では、特にカーネル 2.4.x の標準スケジューラについて言及したが、カーネル 2.5.x における O(1) ス



(a) 標準スケジューラ（実行時間 19.8 秒） (b) 改良スケジューラ（実行時間 12.6 秒）

図 15: 各プロセスが走行していた CPU

ケジューラでも同様の問題が起こり得る。これに対しても、例えば `load_balance()` などのスケジューリングに関連する部分に本節で述べたスケジューリングポリシーを適用することで同様の効果が期待できる。

5.4 Hyper-Threading 対応 CPU における時分割制御に関する考察

これまでで述べたように、Hyper-Threading 機能を搭載した CPU では、あるスレッド（プロセス）の実行が同一物理 CPU 内部で動作している別のスレッドの影響を受ける。そのため、同一のプログラムを実行しても、単独で動作している場合に比べて、見かけ上 CPU 性能が低下してしまう恐れがある。同時に、当該プロセスのユーザ時間（CPU 利用時間）は延びる結果となる。

従来のオペレーティングシステムは、単一の CPU 上において時分割制御を行うことで複数のプロセスの並行実行を実現してきた。このとき、「CPU 性能は一定である」という仮定のもと、各プロセスの使用した CPU 利用時間は、実際に各プロセスが CPU で走行していた細切れの時間を合計することで計算していた。例えば、大型計算機上で CPU 時間に応じて課金するシステムがこれによって構築されていた。しかし、Hyper-Threading 機能を搭載した CPU を利用した場合、他スレッドが動作しているかどうか、あるいはマイクロアーキテクチャレベルで計算資源が競合しているかどうかで動的に CPU 性能が変動することとなる。この結果、単純に上記のような仮定を適用することができなくなる。また、プロセススケジューラの観点からすれば、ある一定の実時間タイムスライスを単位としてスケジューリングを行ったとき、想定したタイムスライスに見合った計算能力をプロセスが実際に使用したかどうか保証できない。

今後は、このような問題を考慮し、CPU 内部の計算資源が割り当てられたかどうかを表す新たな指標が必要になると考えられる。

6 おわりに

本論文では、エンタープライズ用途という観点から、カーネル 2.4.x の標準スケジューラの問題点についてまとめ、これを解決するために、標準カーネルのタスク構造体に対してキャッシュカラーリングを適用した。これにより、8-way Pentium III Xeon サーバにおいて、Web サーバ性能が 23.3%、Chat ベンチマークで 89.6% の性能向上を達成した。また、カーネル 2.5.x で新たに導入された O(1) スケジューラについて詳細な性能評価を行った。その結果、現在まで提案された種々のスケジューラの中では、最新の O(1) スケジューラが良好な結果を示すことが明らかとなった。

また、様々なスケジューラの評価結果を報告するとともに、その問題点についても指摘を行った。特に、O(1) スケジューラを実装したカーネルは、SMP システム上でのプロセスのスケジューリングに大きな偏りを持つ、という問題点が存在することを明らかにした。ピーク性能という観点からすれば、既存のスケジューラの中で O(1) スケジューラが最も有力な選択肢となるが、性能のみにとらわれず、上記問題点にも留意して、サーバマシンのハードウェア構成や使用目的に応じて他のスケジューラを柔軟に選択することも必要である。

加えて、最新の CPU アーキテクチャである Hyper-Threading 機能を搭載した CPU についても評価を行っ

た。ここでは、既存の SMP 向けのスケジューリングポリシーを単純に適用するだけでは、CPU 本来の性能を引き出すことはできないことを述べた。我々の行った簡単な改良により、テストプログラムの性能が最大 1.5 倍向上した。

今後は、本論文で述べたスケジューラのいくつかの問題点について、検討および改良を引き続き行う予定である。

参考文献

[1] "Linux Scalability Effort Project", http://sourceforge.net/projects/lse

[2] "Java Technology, Threads, and Scheduling in Linux", R. Bryant and B. Hartner, Java Technology Update, 4(1), January 2000.

[3] "Enhancing Linux Scheduler Scalability", Mike Kravetz, Hubertus Franke, Shailabh Nagar, Rajan Ravindran, *5th Annual Linux Showcase & Conference*, November 2001.

[4] "Scalable Linux Scheduling", S.Molloy and P. Honeyman, In CITI Technical Report 01-7, University of Michigan, May 2001.

[5] "Speeding Up Kernel Scheduler by Reducing Cache Misses", Shuji YAMAMURA, Akira HIRAI, Mitsuru SATO, Masao YAMAMOTO, Akira NARUSE, and Kouichi KUMON, In *Proc. of the USENIX 2002 Annual Technical Conf. FREENIX Track*, pp.275–285, June 2002.

[6] "[announce] [patch] ultra-scalable O(1) SMP and UP scheduler", Linux kernel mailing list, http://marc.theaimsgroup.com/?l=linux-kernel&m=101010394225604&w=2

[7] "Linux カーネル 2.5 最新開発動向", 後藤 正徳, http://www.atmarkit.co.jp/flinux/special/kernel25/kernel25b.html.

[8] "システム・チューニングで Linux システムを高速化する", 日経 Linux 2002 年 4 月号, pp.56–57.

[9] "GATES（PC サーバ用汎用メモリアクセストレースシステム）の開発". 佐藤 充, 成瀬 彰, 久門 耕一, 情報処理学会 第 59 回全国大会 講演論文集, 1999 年 9 月.

[10] "詳解 LINUX カーネル", Daniel P. Bovet, Marco Cesati 著, 高橋 浩和, 早川 仁 監訳, 岡島 順治郎, 田宮まや, 三浦 広志 訳, オライリー・ジャパン, 2001 年 7 月.

[11] "Solaris Internals Core Kernel Architecture", Jim Mauro and Richard McDougall, SUN MICROSYSTEMS PRESS.

[12] "The Slab Allocator: An Object-Caching Kernel Memory Allocator", Jeff Bonwick, In USENIX Conference Proceedings, pp 87-98, 1994.

[13] "WebBench Homepage", http://etestinglabs.com/benchmarks/webbench/webbench.asp

[14] "Linux Benchmark Suite Homepage", http://lbs.sourceforge.net/

[15] "Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel", Ray Bryant, John Hawkes, 4th Annual Atlanta Linux Showcase & Conference, October 2000.

[16] "Hyper-Threading Technology", http://developer.intel.com/technology/hyperthread/

[17] "Hyper-Threading Technology Architecture and Microarchitecture", Deborah T. Marr, et al., *Intel Technology Journal*, Volume. 6, Issue. 1, February 2002.

[18] "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, In *Proc. of 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, June 1995.

[19] "An elementary processor architecture with simultaneous instruction issuing from multiple threads", H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, In *Proc. of 19th Annual International Symposium on Computer Architecture*, pp. 136–145, May 1992.