2003－OS－93 (5)
2003／5／8

多重化I/Oの実行間隔制御における スケジュール操作による確定的なプロセッサ利用の実現

河合 栄治[†,‡] 門林 雄基[†] 山口 英[†]

[†] 奈良先端科学技術大学院大学
[‡] 科学技術振興事業団 さきがけ研究21

概要

我々は，ネットワークサーバにおけるサービス応答時間の低減と負荷に応じたプロセッサ利用率の実現を目的として，多重化 I/O の実行間隔制御による効率化手法を提案している．しかし，これまでの手法では，同時コネクション数が非常に多い環境の場合，プロセッサ利用率が比較的負荷の軽い段階で 100%に達してしまうという問題があった．その主な原因は，多重化 I/O を含む主ループにおける処理時間が長く，オペレーティングシステムのスケジューラとの間で干渉が発生することである．本研究では，実時間スケジューリング方式を導入することにより，こうしたスケジューラによる干渉を回避し，同時ソケット数が非常に多い環境でも実行間隔制御を正しく動作させる手法を提案する．

Deterministic Processor Utilization with Schedule Operation for Interval Control on I/O Multiplexing

Eiji Kawai[†,‡] Youki Kadobayashi[†] Suguru Yamaguchi[†]

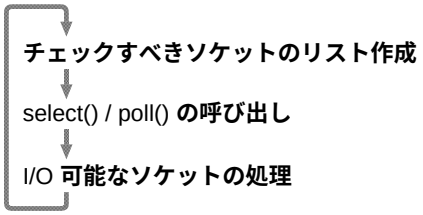
[†]Nara Institute of Scicence and Technology
[‡]PREST, Japan Science and Technology Corporation

Abstract

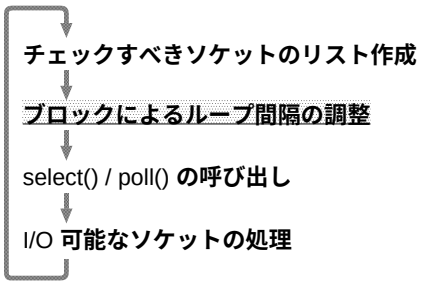
We have proposed interval control mechanisms for I/O multiplexing, whose goal is to achieve both low service delay and moderate processor utilization in proportion to the request arrival rate. However, our previous implementation still consumes a large amount of processor resource especially when the number of concurrent connections is considerably large. The problem is the interference of a thread scheduler with the interval control. To solve this issue, we introduce real-time scheduling into the interval control mechanisms and remove unexpected context switches. The result of benchmarking tests shows that the effectiveness of the interval control is highly improved even when the number of the concurrent sockets is large.

1 はじめに

ネットワークサーバなどを動作させるプラットホームとして Unix が広く利用されている．Unix では，多数のコネクションを効率よく処理するために，ネットワーク I/O イベントの多重化機構として多重化 I/O が用いられることが多い．この多重化 I/O は，ポーリング I/O とも呼ばれ [1]，具体的には `select()` もしくは `poll()` によって実装される機能である．多重化 I/O により，個々のコネクションにプロセスもしくはスレッド（以後スレッドと総称する）を割り当てることなく，単一のスレッドで複数のコネクションを同時に扱うことが可能である．一方で，この多重化 I/O は，扱う同時コネクションの数が増加



(a) 従来方式の主ループ



(b) 提案方式の主ループ

図 1: 多重化 I/O における実行間隔制御

するに従い性能が劣化するという問題点が指摘されている．この問題点を解決するために，これまでに幾つかの手法が提案されている [2, 3, 4, 5, 6, 7]．一方で，これらの手法は，従来の多重化 I/O のプログラミングモデルを大きく変更したり，オペレーティングシステムの改変を要したりするため，実用には大きなコストがかかるという問題点があった．

我々は，多重化 I/O のプログラミングモデルをそのままに，多重化 I/O の実行間隔を細かく制御することにより性能を改善する手法をこれまでに提案した [8]．多重化 I/O の問題点は，同時ソケット数増大に伴う計算量の増大と考えられている．しかし，より本質的には，オーバヘッドによるプロセッサ資源の枯渇を回避できない処理の構造にある．我々の手法は，多重化 I/O を含む主ループの実行間隔を非常に短時間のスレッドブロックにより制御し，多重化 I/O の効率を向上させ，プロセッサ資源の枯渇を防止している（図 1）．

ところが，提案した手法は，非常に多くの同時コネクションを扱うサーバの場合，サービス応答時間の低減やスループットの増大という効果は得られるものの，プログラムの挙動自体には実行間隔制御による変化があまりみられないことが明らかになった．例えば，同時コネクション数が少ない場合には，プロセッサの利用率はサーバへのリクエストレートに対して線形に増加し，負荷に応じたプロセッサ利用が達成された．また，多重化 I/O（ここでは `poll()`）のカーネルにおけるプロセッサ利用率は，負荷の増減に関係なく 10%以下で推移していた．一方で，多数の同時コネクションを扱う場合には，比較的低いリクエストレートの時点でプロセッサ利用率が 100%に達してしまっている．また，多重化 I/O のカーネルにおけるプロセッサ利用率は，実行間隔制御を行わない場合と比較して数%低いものの，そのリクエストレートに対する変化の挙動はほぼ同様となっている．

以上の分析により，提案した実行間隔制御は同時コネクション数が非常に大きい場合には想定した挙動をしていないことが分かった．その主な原因の一つが，処理中のスケジューラの干渉である．同時コネクション数が大きい場合には，ソケットリストの走査およびソケット管理のオーバヘッドにより，この実行間隔（図 1 (a) および (b) におけるループの間隔）がスケジューラが管理するタイムスライスよりも大きくなってしまい，この主ループの実行において予期しないコンテキストスイッチが発生してしまうのである．また，それ以外にも他のブロックされていたスレッド（例えば listen ポートにおいて `accept()` を呼び出すスレッド等）が実行可能状態になった場合などにも発生する．これまでの実装では，こうしたコンテキストスイッチを考慮することなく，実行時間をタイムスタンプの時間差で算出しているため，実際の実行時間より長いループ間隔になってしまい，実行間隔制御が有効に動作しないのである．

本研究では，こうしたコンテキストスイッチを防止し，実行間隔制御の実効性を向上させるために，実時間スケジューリングポリシに基づいたスケジューリング方式 [9] を導入することを検討する．単純に多重化 I/O を実行するスレッドに実時間スケジューリングポリシを適用すると，他のスレッドや他のプロセスに大きな影響を与えることが予想される．そのため，多重化 I/O を含む主ループにおける実際の I/O 処理の間だけ実時間スケジューリングポリシを適用する方式を提案する．これにより，同時コネクション数が大きい場合にも負荷に応じたプロセッサ利用を実現し，負荷の変動がサービス応答時間に与える影響を小さくすることができる．

以下に，本論文の構成を示す．まず 2 節で，我々がこれまでに提案した多重化 I/O における実行間隔制御手法について簡単にまとめ，限界およびその原因について述べる．次に 3 節では，本論文で提案す

```
// 主ループ
for (;;) {
  // コネクションリストの準備
  prepare_conn_list(&conn_list);

  // ループの間隔が短すぎる場合はブロックする
  gettimeofday(&now, NULL);
  ptime = time_diff(&now, &prev);
  if (ptime < min_cycle) {
    set_time(&stime,
             min_cycle - ptime);
    nanosleep(&stime, NULL);
    slept = 1;
    gettimeofday(&prev, NULL);
  } else {
    slept = 0;
    prev = now;
  }

  // リスト中の各ソケットの処理
  if (!empty(&conn_list))
    // poll(), read/write I/O
    proc_conns(&conn_list);

  // ブロックしなかった場合は
  // 他のスレッドに譲る
  if (!slept)
    sched_yield();
}
```

図 2: 多重化 I/O における間隔制御の実装（旧バージョン）

る，実時間スケジューリング方式を用いた実行間隔制御手法について述べる．本研究では，提案手法を実際に Web アクセラレータ[1]に組み込んで，ベンチマークテストによる性能評価およびテスト中のカーネルプロファイリングを行った．4 節ではそれらの結果について詳細に述べる．最後に 5 節で結論を述べる．

2 多重化I/Oの実行間隔制御および問題点

本節では，我々が提案した多重化 I/O の実行間隔制御について簡単にまとめ，同時コネクション数が多い場合の問題点について述べる．

[1]リバースプロキシサーバとも呼ばれる．

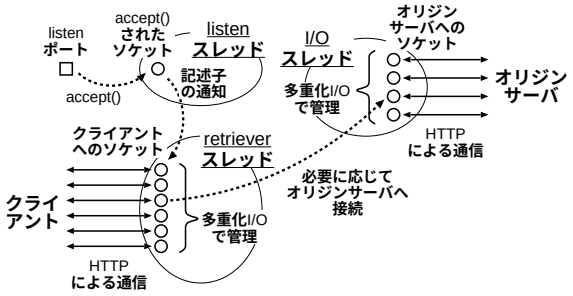


図 3: Chamomile における主要なスレッド

2.1 多重化 I/O の実行間隔制御

これまでに我々は，文献 [8] で多重化 I/O における実行間隔制御手法を提案し，Web アクセラレータ Chamomile [10] に本手法を実装した．ここでは，その実行間隔制御機構の説明のために，疑似コードによる実装を文献 [8] より図 2 に再掲する．

Chamomile における主要なスレッドは 3 つあり，それらは listen ポートにおいて `accept()` を呼び出す listen スレッド，実際のリクエスト処理を行う I/O スレッド，オリジンサーバからコンテンツを取得する retrieve スレッドである（図 3）．図 2 の実装は，I/O スレッド及び retrieve スレッドのものである．

このように，多重化 I/O を含む I/O 処理を行うスレッドは，その主ループにおいてループ間隔を計測し，必要に応じて自身をブロックすることにより，ループ間隔を調整している．これは，多重化 I/O の呼び出し回数を削減することによりプロセッサの利用を軽減し，個々の呼び出しの間隔を増加させることにより返り値として得られる I/O 可能なソケットの数を増加させ，多重化 I/O 呼び出しの効率を向上するものである．

2.2 同時コネクション数が多い場合の問題点

これまでに提案した実行間隔の制御方式は，主ループにおける実行時間を計測し，あらかじめ設定しておいたループ間隔との差分だけスレッドをブロックすることにより，多重化 I/O の実行間隔を制御している．ところが，本実装は処理中のコンテキストスイッチにより他のスレッドの処理が間に挿入されることを考慮していないため，計測された実行時間は不正確である場合がある．特に，同時コネクション

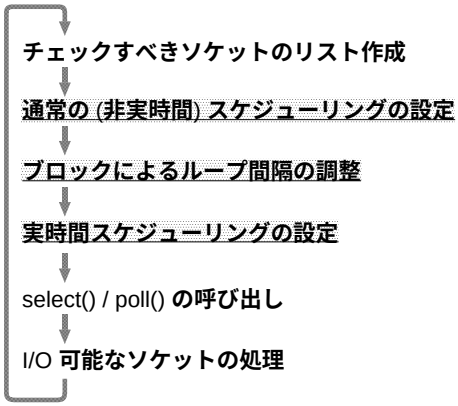


図 4: 実時間スケジューリングを用いた多重化 I/O における実行間隔制御

数が増大するとこの主ループの実際の処理時間自体が長くなるため，このような予期しないコンテキストスイッチによる影響が大きくなる．その場合，計測された見掛け上の処理時間は設定されたループ間隔を越えてしまい，実際にはスレッドはブロックされることなく単に他のスレッドに実行を移すのみで，プロセッサ資源の枯渇を招いてしまう．

3 実時間スケジューリング方式を用いた実行間隔制御

本研究では，プロセッサ資源の枯渇を防ぐために，POSIX 実時間スケジューリング方式 [9] の導入を検討する．実時間スケジューリングポリシは，単純に適用すると他のスレッドやプロセスに大きな影響を与え，問題を発生しやすい．特に，障害発生時に実時間スケジューリングされるスレッドがプロセッサ資源を完全に占有することも考えられる．そのため，こうした障害に対処するために，より高い実時間優先度で動作するコンソールシェルを用意するなど工夫が推奨されている [11].

本研究で提案する多重化 I/O の実行間隔制御では，多重化 I/O を含む主ループにおける I/O 処理の間のみコンテキストスイッチが防止できれば良い．そこで，その処理の間だけ実時間スケジューリングポリシを適用する方式を提案する．本方式により不要なコンテキストスイッチが削減され，実行間隔制御の実効性が向上する．そのため，同時コネクション数が大きい場合にも，プロセッサ利用率およびサービス応答時間をより柔軟に制御できるようになる．さらには，必要な場面にのみ実時間スケジューリングポリシを適用するため，他のスレッドやシステムプロセスへの影響が少なく，問題が発生しにくい．

```
// 実時間スケジューリングポリシの適用
realtime_priority();

// 主ループ
for (;;) {
  // コネクションリストの準備
  prepare_conn_list(&conn_list);
  // ループ間隔が短すぎる場合は
  // 優先度を普通に戻してブロックする
  gettimeofday(&now, NULL);
  ptime = time_diff(&now, &prev);
  if (ptime < min_cycle) {
#ifdef STATIC_MIN_CYCLE
    set_time(&stime,
             min_cycle - ptime);
#else
    set_time(&stime, ptime * ratio);
#endif
    // 非実時間スケジューリングポリシの適用
    normalize_priority();
    // スレッドのブロック
    nanosleep(&stime, NULL);
    // 実時間スケジューリングポリシの適用
    realtime_priority();
  } else {
    sched_yield();
  }
  gettimeofday(&prev, NULL);

  // リスト中の各ソケットの処理
  if (!empty(&conn_list))
    // poll(), read/write I/O
    proc_conns(&conn_list);
}
```

図 5: 多重化 I/O における実時間スケジューリングを導入した間隔制御の実装

3.1 スケジューリングポリシの変更機構の実装

本研究で提案する多重化 I/O の実行間隔制御手法の処理フローが図 4 である．このように，スレッドをブロックする直前にスケジューリングポリシを通常のもの（実際は SCHED_OTHER）に設定し，ブロックが終了したら実時間のもの（実際は SCHED_FIFO）に設定するのである．図 5 に本手法の実装を疑似コー

ドで示す．

実時間スケジューリングを適用することにより，多重化 I/O のより正確な実行間隔制御が可能になる．ここで，この実行間隔制御に二つのポリシを適用することができる．一つは静的な実行間隔の制御であり，もう一つは実行時間に応じた間隔の動的な制御である．

静的実行間隔の制御は，図 5 の実装においてマクロ STATIC_MIN_CYCLE を定義した場合であり，多重化 I/O の実行間隔はあらかじめ設定で与えられた値に調整される．これは，負荷が小さい場合はブロックする時間が長く，負荷が大きい場合はブロックする時間が短くなることを意味する．そのため，負荷の増加に応じたプロセッサ資源の利用が実現され，サービス応答時間も一定に保たれるであろうと考えられる．

動的実行間隔の制御は，STATIC_MIN_CYCLE を定義しない場合であり，これは実際の主ループの実行時間にあらかじめ与えられた定数を掛けた値を実行間隔とする方式である．そのため，負荷が大きくなればブロックする時間も長くなり，結局負荷に依存しない一定のプロセッサ利用が実現され，一方で負荷に応じたサービス応答時間になると考えられる．

4 性能評価

本研究では，提案手法の実効性を評価するために，Web Polygraph [12] の WebAxe-4 ワークロード [13] を用いたベンチマークテストによる性能評価を行った．ベンチマーク環境は，Chamomile の基本設定を含め，比較のために文献 [8] と全く同様のものを用いた．基本構成および機材の仕様をそれぞれ図 6 および表 1 として再掲する．なお，本研究が対象とする多数の同時ソケットを扱う場合について評価するために，HTTP/1.1 永続コネクションを有効にしている．すなわち，Chamomile はテスト中に約 7,000 のソケットを並行して管理する．

多重化 I/O の実行間隔制御方式としては，静的に設定する場合（静的実行間隔制御）および負荷に応じて動的に設定する場合（動的実行間隔制御）の二通りを実験し評価した．以下本節では，スループット及び応答時間の分析と，ベンチマーク中のカーネルプロファイリングによる分析について述べる．

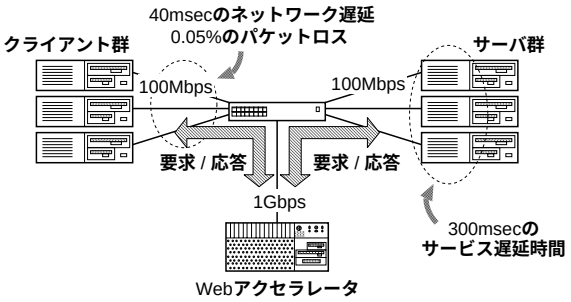


図 6: WebAxe-4 のテスト環境

表 1: Web Polygraph によるテストで用いた機器

サーバ	Pentium III 866MHz，512MB，Intel Pro/100，FreeBSD 4.3
クライアント	Pentium III 1.4GHz，512MB，Intel Pro/100，FreeBSD 4.3
アクセラレータ	Pentium III 800MHz，2GB，NetGear GA620T (1000baseT)，Linux 2.5.17
スイッチ	NetGear FS518T (1000baseT×2，100baseT×16)

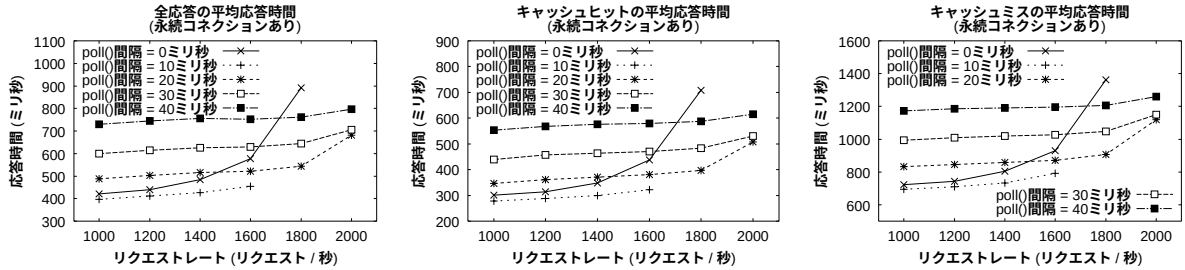
4.1 スループットと応答時間の比較

本節では，ベンチマークテストで得られた結果をもとに，提案する手法が組み込んだアクセラレータのスループットおよびサービス応答時間に与える影響を評価する．

4.1.1 静的実行間隔制御の評価

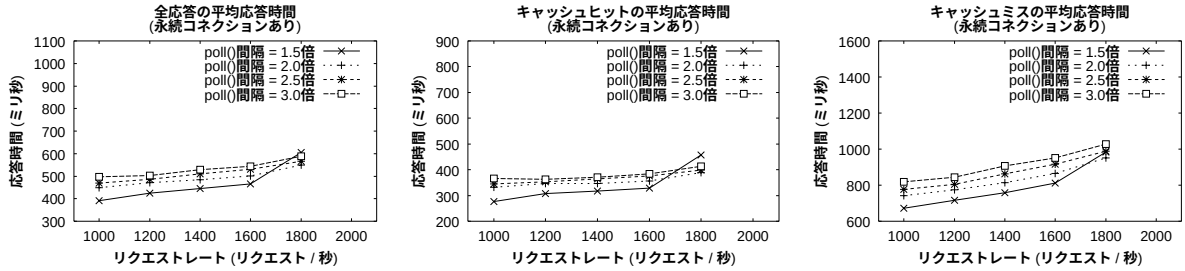
図 7 は，静的実行間隔制御におけるスループットと応答時間の関係を示すグラフである．3 つのグラフは左からそれぞれ，全応答，キャッシュヒット応答，キャッシュミス応答の平均サービス応答時間を表す．グラフ中における poll() 間隔とは，図 5 の実装における主ループの実行間隔のことである．なお，poll() の間隔が 0 ミリ秒とは，実行間隔制御を全く行わない場合（図 1 (a)）である．

まず，これらのグラフから多重化 I/O の実行間隔が最終的なサービス応答時間に与える影響は，従来の実時間スケジューリングを用いない方式と比較して大きいことが判明した．間隔を 30 ミリ秒以上に設定すると，非常に負荷が高い場合を除いて従来方式より応答時間が大きくなってしまう．



(a) 全平均　(b) キャッシュヒット　(c) キャッシュミス

図 7: 静的実行間隔制御の場合のリクエストレートと応答時間の関係（poll() 間隔が 0 の場合は制御を行わないことを意味する)



(a) 全平均　(b) キャッシュヒット　(c) キャッシュミス

図 8: 動的実行間隔制御の場合のリクエストレートと応答時間の関係

また，実行間隔を 10 ミリ秒とすると良好なサービス応答時間が得られるが，最大スループットが従来方式よりも低下してしまうという結果となった．これは，実行間隔が短すぎ，プロセッサ資源の枯渇が発生するためであることが判明した．プロセッサ利用率の分析については後述する．

一方で，サービス応答時間自体の分布はリクエストレートの増加に対して非常に緩やかな増加にとどまっており，実行間隔制御が有効に機能していることが分かる．

4.1.2 動的実行間隔制御の評価

動的実行間隔制御を行った場合のスループットと応答時間の関係を図 8 に示す．3 つのグラフの意味は，静的実行間隔制御の場合と同様である．なお，グラフ中における poll() 間隔とは，主ループの実行間隔をブロック時間を考慮しない実際の処理時間の何倍にするかを意味している．例えば，poll() 間隔が 2 倍の場合，実際の処理に要した時間と同じ時間だけスレッドをブロックする．

動的実行間隔制御を行った場合，静的実行間隔制御の場合と比較して，全般的に応答時間が低く押さえられていることがグラフから分かる．動的実行間隔制御では，負荷が増加するとそれだけ主ループの処理時間が増加し，実行間隔もそれに比例して増加するため，リクエストレートが高まるにつれ応答時間も増加する．しかしながら実験の結果，応答時間の上昇は比較的小さく，リクエストレートの増加に対してほぼ線形の増加に押さえられてる．

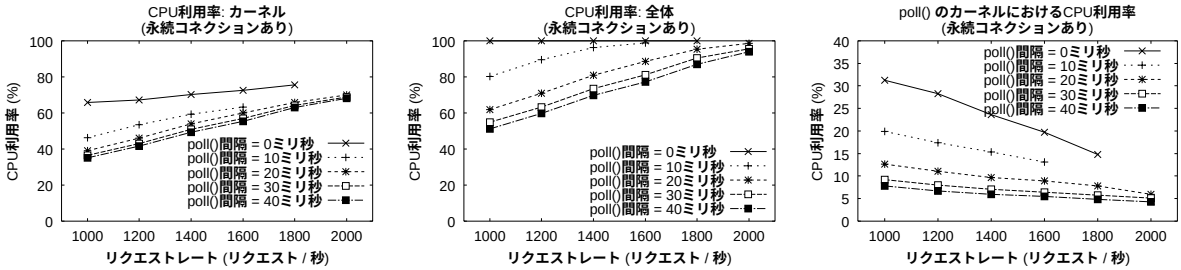
一方で，最大スループットについては，静的実行間隔制御の場合は毎秒 2000 リクエストまで達成していたが，動的実行間隔制御では若干減少し毎秒 1800 リクエストであった．この点については，後程プロセッサ利用率の議論と合わせて考察する．

4.2 カーネルプロファイル分析

本節では，ベンチマーク中のカーネルプロファイリング結果（SGI 社の kernprof [14] を利用）をもとに，プロセッサ利用という観点から提案手法を組み込んだアクセラレータの挙動について分析する．

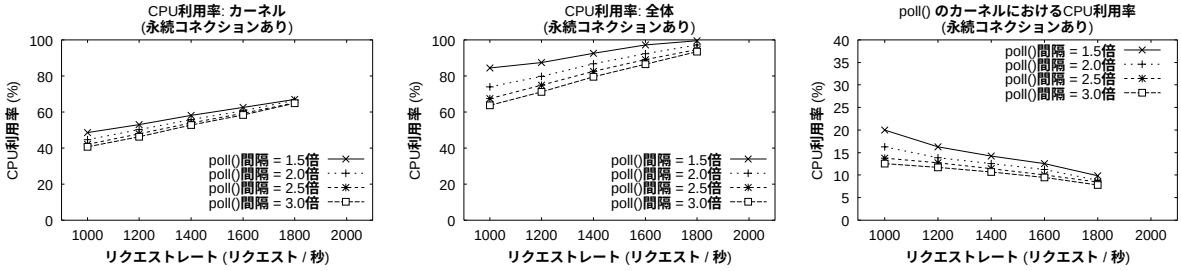
4.2.1 静的実行間隔制御の評価

図 9 に，静的実行間隔制御におけるリクエストレートに対するプロセッサ利用率の変化を示すグラ



(a) カーネル　(b) 全体　(c) カーネルにおける poll()

図 9: 静的実行間隔制御の場合のリクエストレートと CPU 利用率の関係（poll() 間隔が 0 の場合は制御を行わないことを意味する）



(a) カーネル　(b) 全体　(c) カーネルにおける poll()

図 10: 動的実行間隔制御の場合のリクエストレートと CPU 利用率の関係

フを示す．左からそれぞれ，カーネル，システム全体（ユーザプロセス及びカーネル），カーネルにおける poll() のプロセッサ利用率[2]をそれぞれ示す．

カーネルのプロセッサ利用では，制御を行わない（poll() 間隔が 0 ミリ秒）場合と比較して低減されており，さらにはリクエストレートにほぼ比例していることが分かる．また，システム全体でも間隔を 20 ミリ秒以上に設定すればほぼリクエストレートに比例したプロセッサ利用となっている．間隔を 10 ミリ秒に設定した場合，毎秒 1400 リクエストを越えたあたりからプロセッサ資源の枯渇が見られる．これが最大スループットが毎秒 1600 リクエストに低下した原因と考えられる．

一方で，poll() のプロセッサ利用率も実行間隔制御により大きく低下している．実時間スケジューリングを用いない場合，カーネルにおける poll() のプロセッサ利用は，実行間隔制御によって数%程度しか減少しなかった [8]．しかし，実時間スケジューリングの導入により，その減少幅は大きくなり，リクエストレートの影響も小さく押さえられているのが分かる．これは，実行間隔制御がより正確に行われており，poll() の呼び出しレートがほぼ一定で推移していることを示している．

4.2.2 動的実行間隔制御の評価

図 10 は，動的実行間隔制御におけるリクエストレートに対するプロセッサ利用率の変化のグラフである．グラフの意味は静的実行間隔制御の場合と同様である．

グラフから，カーネルのプロセッサ利用率は静的実行間隔制御の場合とほぼ同様の動きを示していることが判明した．動的実行間隔制御を行う場合，負荷が重くなればなるほどブロックする時間が増加するが，カーネルにおける処理はユーザプロセスのブロックとは無関係に大半が TCP/IP 層における処理に費やされており，カーネルのプロセッサ利用はリクエストレートに対して線形に増加する結果となった．このことは，負荷が比較的高い場合には poll() のカーネルにおけるプロセッサ利用率が 8%から 15%程度に押さえられていることからも分かる．

一方で，全体のプロセッサ利用率を見ると，カーネルのプロセッサ利用率の分布と比較して，実行間

[2]システム全体における poll() のプロセッサ利用率ではないことに注意．

隔の設定による差がより大きく出ていることが分かる．これは，カーネルにおいてはネットワーク処理が大半を占めているが，ユーザプロセスにおいてはブロックする時間に応じてプロセッサ利用率が変化するためである．

5 おわりに

本研究では，これまでに提案してきた多重化 I/O における実行間隔制御において，実時間スケジューリングを導入することにより，同時ソケット数が非常に多い場合でも正確に制御を行う手法を提案した．制御方式としては，負荷に関係なく実行間隔を一定に制御する静的実行間隔制御方式と，実際の処理時間に応じて実行間隔を制御する動的実行間隔制御方式を提案した．どちらの場合も，ベンチマークテストによる性能評価では，ほぼ正確に実行間隔制御が動作していることが分かった．

実時間スケジューリングを用いる場合の利点は，非常に多くの同時ソケットを扱う場合でもプロセッサ資源の消費がリクエストレートに対して線形に推移することである．一方で，文献 [8] で議論した通常の非実時間スケジューリング方式と比較して，サービス応答時間の一定の劣化が見られる．そのため，実時間スケジューリングを用いた方式の利点に対する要求が強い場合には，導入する意義があるだろう．大規模サーバクラスタにおいて，リクエストレートに応じた消費電力を実現し，総消費電力を軽減したい場合などがその一例である．

参考文献

[1] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley, 1996.

[2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *USENIX Annual Technical Conference*, pp. 253–265, June 1999.

[3] Jonathan Lemon. Kqueue: A generic and scalable event notification facility. In *2001 USENIX Annual Technical Conference*, June 2001.

[4] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. In *2000 USENIX Annual Technical Conference*, June 2000.

[5] Niels Provos, Chuck Lever, and Stephen Tweedie. Analyzing the Overhead Behavior of a Simple Web Server. In *4th Annual Linux Showcase & Conference*, October 2000.

[6] Abhishek Chandra and David Mosberger. Scalability of Linux Event-Dispatch Mechanisms. In *2001 USENIX Annual Technical Conference*, June 2001.

[7] Shridhar Acharya. Using the devpoll (`/dev/poll`) Interface. Technical Articles, March 2002. `http://access1.sun.com/techarticles/devpoll.html`

[8] 河合栄治, 門林雄基, 山口英. 多重化 I/O の実行間隔制御による効率化手法. 第 6 回プログラミングおよびシステムに関するワークショップ（SPA'03）. 日本ソフトウエア科学会, 2003 年 3 月. `http://spa.jssst.or.jp/2003/program/papers/03001.pdf`

[9] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Inc., 1995.

[10] Chamomile Web Server Accelerator. `http://iplab.aist-nara.ac.jp/~eiji-ka/chamomile/index.html`

[11] Linux manual page of `sched_setscheduler(2)`, August 1999.

[12] Web Polygraph. `http://www.web-polygraph.org/`

[13] WebAxe-4 Workload. `http://www.web-polygraph.org/docs/workloads/webaxe-4/`

[14] Silicon Graphics. Kernprof (Kernel Profiling). `http://oss.sgi.com/projects/kernprof/`