ネットワークサーバにおける多重化I/Oの実行間隔制御による性能向上手法

河 合 栄 治[†,☆] 門 林 雄 基[††] 山 口 英[††]

Unix 上のサーバプログラムなどでネットワーク I/O を多重化するのによく用いられるselect()やpoll()（多重化 I/O）には，サーバ負荷の増加に対する性能のスケーラビリティに欠けるという問題がある．従来の解決手法は，これらの関数におけるソケットテーブルの走査を廃止し，特別なイベント通知機構を別途設けるものであった．しかし，これらの手法は，オペレーティングシステムの改造が必要であったり，プログラミングモデルの変更を要したりするため，導入コストが高いという別の問題がある．多重化 I/O において真に問題なのは，ソケットテーブルの走査そのものではなく，多重化 I/O がそのイベント駆動的な処理構造により，必要以上に呼び出されてしまうことである．そこで本論文では，多重化 I/O の呼び出し間隔を制御することで，サーバの性能を向上させる手法を提案する．実際に Web サーバアクセラレータに本手法を実装してベンチマークテストを行った結果，スループットで 11～25%の向上が見られることが判明した．また，サービス遅延時間についても，同時ソケット数に応じて挙動が変化するものの，大きな改善が見られた．また，本手法はselect()やpoll()を用いた従来のプログラミングモデルを踏襲するため，適用コストが非常に小さいという特長も持つ．

Enhancing Network Server Performance by Controlling Execution Intervals of Polling I/O

Eiji Kawai,[†,☆] Youki Kadobayashi[††] and Suguru Yamaguchi[††]

Although network I/O polling with select() or poll() is a strong and popular mechanism to handle concurrent sockets on a network server, there is a problem that they have poor performance scalability. Since the problem is believed to be caused by the overhead of scanning a large socket list, several solutions proposed so far provide special I/O event notification mechanisms that replace the socket list scanning. On the other hand, such solutions raise another problem that they require major modification in the operating system and/or in the programming model of I/O multiplexing, which makes their adoption difficult. The real problem of the I/O polling is its excessively frequent execution, not the processing cost of the socket list scanning. In this paper, we propose a unique and simple solution that improves the scalability of the I/O polling by controlling its execution intervals. This technique decreases the CPU cycles consumed by the I/O polling functions and improves server performance from a viewpoint of throughput and service response time. Because it does not alter the programming model of the traditional I/O polling, the implementation cost is low.

1. はじめに

インターネットサービスを提供するサーバプラットホームには Unix が広く用いられている．その主な理由として，Unix においてネットワークプログラミングのためのソケットモデルがいち早く確立されたことがあげられる．この Unix ソケットを用いて，数多くのネットワークプログラミングモデルが開発された．それらの中で，高い性能が求められるサーバの実装において，多重化 I/O と呼ばれるモデルがよく用いられている．

多重化 I/O とは，select() やpoll() によって実装されており，複数のソケットの状態を 1 回の呼び出しでチェックし，状態の変化をプロセスもしくはスレッド（以後区別なくスレッドと総称する）に通知するものである．この多重化 I/O により，単一のスレッドによる複数ソケットの並行処理が可能となり，ソケット

† 科学技術振興事業団さきがけ研究 21
PRESTO, Japan Science and Technology Corporation
†† 奈良先端科学技術大学院大学情報科学研究科
Graduate School of Information Science, Nara Institute of Science and Technology
☆ 現在，奈良先端科学技術大学院大学附属図書館研究開発室
Presently with Digital Library Research Division, Nara Institute of Science and Technology

ごとにスレッドを割り当てる必要がなくなる．そのため，スレッド切替えのオーバヘッドが削減でき，高い性能を達成することができる．

一方で，近年の技術開発によるネットワーク通信速度のさらなる向上やサービスの複雑化から，サーバで並行処理するソケットの数が増加する傾向にあり，多重化 I/O における性能のスケーラビリティが問題になっている．この問題を解決するために，これまでにいくつかの改善手法が提案された．これらの手法の共通点は，多重化 I/O において呼び出しごとに行われるソケット群に対する状態検索の負荷が問題であるとしていることである．そのため，ソケットテーブルの走査を廃止し，その代わりに特別なインタフェースを通じたイベント通知により効率化を実現している．しかし，これらの解決策はオペレーティングシステムの改変が必要であったり，プログラミングモデルの大きな変更を要したりするため，適用するのが難しいという問題がある．

本研究では，多重化 I/O の呼び出し間隔を制御することにより，従来の多重化 I/O のプログラミングモデルを維持したまま，サーバの性能を向上する技術を提案する．多重化 I/O は，そのイベント駆動的な機構により，引数として与えられたソケットリストに含まれるソケットのうち 1 つでも I/O 可能な状態であれば呼び出し元に実行を戻す．そのため特に多くのソケットを並行処理するようなネットワークサーバでは，必要以上に頻繁に多重化 I/O が呼び出されてしまい，CPU 資源の枯渇を招いてしまう．本手法は，多重化 I/O の呼び出し間隔を制御することにより，多重化 I/O の効率および全体のサービススループットを向上させる．

以下，本論文の構成を述べる．2 章では Unix におけるネットワーク I/O の多重化手法を概観し，3 章では多重化 I/O に関する従来研究についてまとめる．4 章では，本研究で提案する多重化 I/O の呼び出し間隔の制御による効率の改善手法について述べる．本研究では，提案する手法を Web サーバアクセラレータ（リバースプロキシサーバとも呼ばれる）に実装し，ベンチマークテストによる性能評価を行った．また，カーネルプロファイリングを通じて，提案手法におけるカーネルの挙動について分析した．5 章ではそれらの結果について述べる．最後に 6 章で本論文をまとめ，今後の課題について述べる．

2. Unix におけるネットワーク I/O の多重化手法

ネットワーク I/O において高い性能を達成するためには，各ソケットにそれぞれ別のスレッドを割り当てるのではなく，単一のスレッドが複数のソケットを並行して効率良く扱うことが必要となる．本章では，そのためのソケットプログラミングモデルについて概観する．ここでは，ノンブロッキング I/O，多重化 I/O，シグナル駆動 I/O に分類する[1]．

2.1 ノンブロッキング I/O

ノンブロッキング I/O は，対象となるソケットに`fcntl()`☆を用いて`O_NONBLOCK` フラグをセットすることで有効となり，I/O 要求に際してスレッドをブロックしない方式である．いい換えると，I/O 要求時に即時処理可能な部分のみ実行するものである．たとえば，読み込むべきデータが何も到着していない場合は，データの読み込み要求に対して何もデータを返さずに呼び出しスレッドに実行を戻す．

このように，ノンブロッキング I/O ではスレッドをブロックしないため，複数のソケットを並行して扱うにはソケット群に対して順に I/O を呼び出すだけでよい．しかし，読み込むべきデータが到着していない時点で読み込み I/O 要求が発行されることが多く，CPU サイクルを浪費する問題があり実用的ではない．そのため，通常のブロッキング I/O の場合と同様，次にあげる多重化 I/O を併用する必要がある．

2.2 多重化 I/O

多重化 I/O は，ポーリング I/O とも呼ばれ[2]，I/O の前にそれが即時処理可能かどうかをチェックするものである．これは`select()` や`poll()` によって実装されている．多重化 I/O を用いれば，即時処理可能な I/O のみを選択的に実行することができるため，複数のソケットを単一のスレッドによって`read()` や`write()` でブロックすることなく並行して扱うことが可能となる．

`select()` や`poll()` では，1 回の呼び出しで複数のソケットについて状態をチェックするだけではなく，さらにそれらの状態の変化を待つことができる．すなわち，与えられたソケットの集合において，そのいずれかにデータが到着するまでスレッドをブロックすることができる．そのため，高いネットワーク I/O 性能が必要な場合，この多重化 I/O によるイベント通知を基礎としたプログラミングモデルが一般的に用いられ

☆ 本論文では，`fcntl()` を用いたソケット操作について述べるが，同様の操作は`ioctl()` を用いても可能である．

ている．

2.3 シグナル駆動型I/O

シグナル駆動型I/Oとは，対象となるソケットのI/Oが可能な状態に変化する際に発行されるシグナルを受信し，スレッドをブロックすることなくI/Oの実行を可能にするものである．具体的には，`fcntl()`の`F_SETSIG`コマンドで，状態変化の通知に用いるシグナルを指定し（指定しなかった場合は`SIGIO`が用いられる），同じく`fcntl()`の`F_SETOWN`コマンドでプロセス記述子を設定することでシグナルによる通知が有効となる．

このシグナル駆動型I/Oは，Unixにおける従来のシグナルとあわせて用いると多くの問題が発生することが知られている[1]．そのため，現実的にはPOSIX実時間シグナル[3]を用いる必要がある．実時間シグナルは，スレッドによるシグナルキューを介した非同期的な受信が可能である．また，実時間シグナルには情報を持たせることが可能であり，シグナル駆動型I/Oの場合ソケット記述子を通知することができる．

一方で，実時間シグナルを用いる場合にもまだ問題が残されている．実時間シグナルがキューに格納された状態でソケットを閉じた場合，そのシグナルは残ったままとなるため，一度閉じて再度生成されたソケットに関して注意深く取り扱わなければならない．また，キューは有限長であり溢れの問題がある．キューが溢れた場合，イベント情報が失われるため，何らかの回復手段をとらなければならない．

3. 多重化I/Oに関する従来研究

これまで，いくつかの研究論文で多重化I/Oにおけるソケット数の増加にともなう性能の劣化が議論され，さまざまな解決策が提案されている．しかしながらこれらの解決策は，プログラミングインタフェースは異なるものの，与えられたソケットの集合における状態変化情報をカーネルで記録しておき，必要に応じて通知を行うという点で共通している．すなわち，多重化I/Oにおけるソケット群に対する状態検索を廃止し，イベント通知のための特別なインタフェースを新たに用意するのである．本章ではそれらについて概観する．

文献4)では，スレッドがチェックしたいソケットの集合をあらかじめカーネルに通知しておき，カーネルはそれらに関する状態の変化を検知するとイベントとしてその情報を特別なキューに格納し，スレッドはそのキューからイベント情報を取得するというモデルを提案している．

文献5)は，先の文献4)と同様の機能をより汎用的なkqueueと呼ばれるインタフェースに拡張したものを提案している．これは，キューにイベント情報を保持したままソケットを閉じてしまった場合に情報が誤ったものになってしまう問題を解決している．

文献6)では，Linux上における同様の機構として`/dev/poll`インタフェース☆を開発し評価している．また，カーネルからのイベント通知機構として，実時間シグナルを用いたシグナル駆動I/Oの利用を提案し，`/dev/poll`インタフェースとの比較を行っている．さらに同じ著者らによる文献8)では，実時間シグナルによるイベント通知機構の改良を提案しており，一度に複数の実時間シグナルを受信することができるインタフェースを開発している．

文献9)は，実時間シグナルを用いる際の問題点としてシグナルキューの溢れをあげ，イベント情報をすべてキューに格納するのではなく，1つのソケットに対して1つのエントリのみ格納することで，キューの溢れを防止している．

これらの研究で提案されている解決手法の問題点は，オペレーティングシステムへの変更を必要とする特別なインタフェースを用いていたり，プログラミングモデルの大きな変更を要したりすることである．本研究の目標は，従来より広く用いられている多重化I/Oのプログラミングモデルをそのままに，性能を改善する手法を開発することである．

4. 多重化I/Oの呼び出し間隔の制御による効率の改善

本章では，多重化I/Oの性能面における問題点について考察し，本論文で提案する呼び出し間隔制御によるサーバ性能の改善手法について述べる．

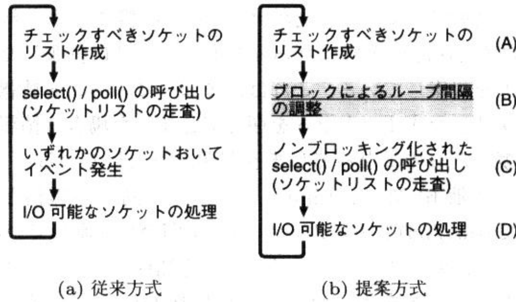
4.1 多重化I/Oの性能面における問題点

多重化I/Oの性能がスケーラブルでない原因は，通信速度が向上すれば多重化I/Oの呼び出し回数も増加することと，同時に扱うソケットの数が大きくなれば多重化I/Oの処理コストも増加することである．これらの2つの原因が同時にサーバの性能に影響を与えるため，クライアントからの要求が増加するに従って，多重化I/Oは性能におけるボトルネックとなってくる．以下，これらの2つの要因について述べる．

4.1.1 多重化I/Oの呼び出し回数の増加

通信速度が向上すれば，同時に扱うソケット集合に

☆ `/dev/poll`インタフェース自体は他のオペレーティングシステムにも実装されている[7]．



(a) 従来方式　(b) 提案方式

図 1 多重化 I/O を用いた処理の流れ
Fig. 1 Processing flow of I/O multiplexing.

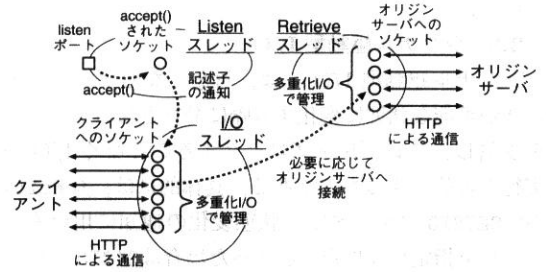


図 2 Chamomile の主要スレッド
Fig. 2 Threads of Chamomile.

おいて，ソケットが I/O 可能へ変化するというイベントの発生頻度が増加する．多重化 I/O は，対象となるソケットのうち 1 つでも I/O 可能になると呼び出しスレッドに制御が戻るため，結局多重化 I/O の呼び出し回数が増加することになる．

4.1.2 多重化 I/O の処理コストの増加

多重化 I/O の 1 回の呼び出しで得られる情報量は小さい．ここでは，多重化 I/O で得られる情報量を，呼び出しの返り値として得られる I/O 可能なソケットの数と定義する．この情報量は一定で小さい（最悪時で 1）にもかかわらず，引数に与えられるソケット集合が大きくなれば多重化 I/O におけるソケット群のチェックに関する処理量が増加することとなる．

4.2 多重化 I/O の呼び出し間隔の制御

3 章で述べた従来研究では，多重化 I/O におけるソケット群の状態チェックに関する処理量が増加する問題について焦点を当てている．一方で本研究では，多重化 I/O の呼び出し回数を削減し，さらに 1 回の多重化 I/O の呼び出しによって得られる情報量を増加させる手法を提案する．

通常の多重化 I/O を用いたプログラミングモデルでは，図 1 (a) に示したような処理の流れとなる．このように，多重化 I/O はそのイベント駆動的な処理機構により，パケットの到着率が上昇するにつれ呼び出し回数が増加する．本研究では，図 1 (b) に示すように非常に短い時間スレッドをブロックする（`nanosleep()` などを用いる）ことにより，多重化 I/O の呼び出し間隔を一定値より大きくする手法を提案する．ここで多重化 I/O は，その引数のタイムアウト値に 0 を与えることでノンブロッキング化しておく．これは，`select()` や `poll()` において，各ソケットを wait channel として呼び出しスレッドを登録する必要がなくなり，処理コストが軽減されるという利点も持つ．本手法により，多重化 I/O の呼び出し間隔が大きくなればその呼び出し回数は少なくなり，さらには 2 つの多重化 I/O 呼び出しの間に状態が I/O 可能に変化するソケットの数が増加することから，1 回の多重化 I/O により得られる情報量が増加する．

一方で，スレッドブロック時間の長さには注意が必要である．多重化 I/O の呼び出し間隔を大きくするということは，サービス応答時間が増大することを意味する．そのため，その間隔は許容範囲に収めなければならない．

4.3 実　装

本研究で提案する I/O モデルは，多重化 I/O ではいっさいスレッドをブロックせず，スレッド自身が能動的に短時間ブロックすることによって多重化 I/O の呼び出し間隔を制御するものである．この機構を，Web サーバアクセラレータ Chamomile[10] に実装した．

Chamomile において実装されているスレッドのうち，主要なものを図 2 に示す．Chamomile には，クライアントからの新しいコネクションを受け付ける Listen スレッドと，受け付けたコネクション上の要求処理を行う 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 スレッドがある．Listen スレッドは `listen()` を実行したポートに対して `accept()` を呼び出し，その結果得られた新しいソケットを実際に多重化 I/O をともなう処理を行う I/O スレッドに渡すのである．本研究では，提案する多重化 I/O の呼び出し間隔の制御機構を，この I/O スレッドおよび Retrieve スレッドに実装した．

図 3 に，多重化 I/O を実行する I/O スレッドの実装を疑似コードで示す．コメント中の記号 (A)–(D) は，図 1 (b) のそれに対応する．また，コード中に `poll()` による多重化 I/O が明示されていないが，`proc_conns()` の中で多重化 I/O および各ソケットに対する I/O を実行している．多重化 I/O は，前に述べたように `poll()` の引数の 1 つであるタイムアウト時間に 0 を与えることでノンブロッキング化している．また，ブロック時間の計算においては，主ループにおける処理時間

```
1 // 主ループ
2 for (;;) {
3   // コネクションリストの準備 (A)
4   prepare_conn_list(&conn_list);
5
6   // ブロックによる調整 (B)
7   gettimeofday(&now, NULL);
8   ptime = time_diff(&now, &prev);
9   if (ptime < min_cycle) {
10    set_time(&stime, min_cycle - ptime);
11    nanosleep(&stime, NULL);
12    slept = 1;
13    gettimeofday(&prev, NULL);
14  } else {
15    slept = 0;
16    prev = now;
17  }
18
19  // リスト中の各ソケットの処理 (C, D)
20  if (!empty(&conn_list))
21    // poll() および read/write I/O
22    proc_conns(&conn_list);
23
24  // ブロックなしの場合はコンテキストスイッチ
25  if (!slept)
26    sched_yield();
27 }
```

図 3 多重化 I/O における間隔制御の実装(疑似コード)
Fig. 3 Implementation of interval control on I/O multiplexing (pseudo code).

(図 3 のptime) を考慮し，ループ間隔の閾値との差分だけブロックしている (図 3 の 10 および 11 行目). この機構により，ループ間隔は一定時間より大きくなるように調整される．一方で，負荷が増大してループの間隔が十分大きくなってしまっている場合には，ブロックしない設計となっている．これは，負荷が高い場合に，サービス応答時間のさらなる増加を招くからである．図 3 に示した実装において，オリジナルの Chamomile への実質的な変更は，6 から 17 行目の追加であり十分小さい．また，poll() などのシステムコールおよびライブラリ関数などにはいっさい変更を加えていない．そのため，本手法により低コストでサーバの性能の改善が可能である．

5. 性能評価およびカーネルプロファイル分析

本研究では提案する手法の評価のために，ベンチマークテストによる性能評価およびカーネルプロファイル分析を行った．本章では，その結果について述べる．

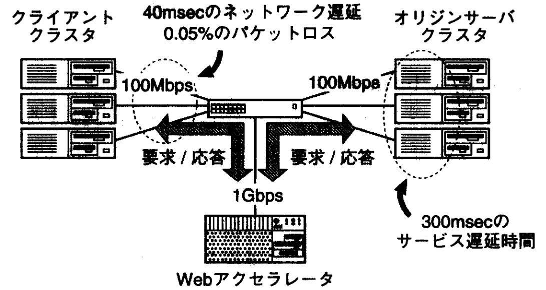


図 4 WebAxe-4 のテスト環境
Fig. 4 Test environment for WebAxe-4 workload.

表 1 Web Polygraph によるテストで用いた機器
Table 1 Specification of the devices for benchmark tests with Web Polygraph.

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

5.1 Web Polygraph によるベンチマークテスト環境

ベンチマークテストでは，Web Polygraph[11] を用いた．3 章であげた従来研究の多くが httperf[12] を用いた評価を行っている．Web Polygraph は，Web キャッシュシステムの評価を行うシステムであり，Web サーバが評価対象である httperf と直接比較することはできない．本研究で Web Polygraph を用いた理由は，より現実的な環境をエミュレートすることができるという利点があるためである．Web Polygraph では，WebAxe-4[13] と呼ばれるワークロードを用いて Web サーバアクセラレータの性能をテストすることができる．WebAxe-4 では，コンテンツサイズや更新頻度，動的なコンテンツの割合，HTTP/1.1 による永続コネクションなどが考慮されている (詳細は付録 A.1 を参照)．このように，Web Polygraph による評価は，多重化 I/O における呼び出し間隔制御がシステム全体に与える影響を総合的に計測することができ，実際にサーバプログラムなどに実装する際の有効な判断指標となる．

5.1.1 ネットワーク構成

ベンチマークテストのネットワーク構成を，図 4 に示す．テストでは，これらのクライアント群およびサーバ群を用い，スイッチに接続されたアクセラレータに

負荷を与える．用いた機材の仕様は**表 1** のとおりである．

5.1.2 Chamomile の設定

ベンチマークの対象となる Chamomile におけるコンテンツキャッシュの容量は 768 MB とし，すべてメモリ上に用意している．キャッシュをメモリ上に用意したのは，ハードディスクを用いるとディスクアクセス速度がシステム性能に大きな影響を与えてしまい，ここでの目的であるネットワーク I/O 性能の評価を適正に行うことができないからである．またキャッシュ容量は，WebAxe-4 がエミュレートするオリジンサーバにおけるコンテンツのワーキングセットサイズが 1 GB であることから，クライアントからのコンテンツ要求の多くがキャッシュヒットする十分な量である．実際には，WebAxe-4 で設定されている最大で 80%程度のキャッシュヒット率が達成可能な要求列に対して，73〜75%程度のキャッシュヒット率を達成した．キャッシュミスした要求については，Chamomile からオリジンサーバ群に対して転送される．

`read()` および `write()`[☆] については，2.1 節で述べたノンブロッキング I/O を用いている．各ソケットは多重化 I/O により事前にチェックされるため，多くの場合はノンブロッキング I/O は不要である．しかし，バッファの空き状況によってはスレッドがブロックされる可能性を排除できないため，今回はノンブロッキング I/O を利用するように設定した．

ベンチマークテストでは，HTTP/1.1 永続コネクションを無効にした場合と有効にした場合の 2 つの場合を検証した．ここで，永続コネクションを有効にした場合のサーバの挙動には注意が必要である．今回の実験では多重化 I/O の負荷を定量的に評価するのが目的であるため，同時コネクション数は一定に保っておくことが望ましい．なぜなら，多重化 I/O では，走査するソケットリストのサイズによって負荷が変化するからである．一方で，同時コネクション数はコネクションタイムアウト時間の設定による影響が大きい．そのため本研究では，同時コネクション数が与えられた閾値を超えないように，コネクションタイムアウト時間を動的に制御する機構を新たに Chamomile に実装し，この閾値を 7,000 に設定した[☆☆]．

☆ 実際には，送受信するデータバッファのタイプに応じて `send()`，`recv()`，`sendmsg()`，`recvmsg()` のいずれかを用いている．

☆☆ Chamomile では同時コネクション数の静的な上限を設定するが，今回の実験ではその値を 10,000 とし，コネクションタイムアウト時間の調整のための閾値をその 70%の 7,000 とした．

表 2 Chamomile における比較パラメータ
Table 2 Parameters for Chamomile.

永続コネクション	無効	有効
多重化 I/O の最小間隔	0〜20 ms	0〜40 ms

5.1.3 Chamomile 実行ホストにおけるカーネルタイマの精度

本研究で提案する多重化 I/O の呼び出し間隔の制御では，`nanosleep()` 呼び出しによるミリ秒単位のスレッドのブロックが必要となる．一方で，これらの関数によるブロック時間は，実際にはオペレーティングシステムのカーネルタイマ機構に依存しており，その精度は今回用いた Linux/i386 では 10 ミリ秒である．そのため，今回はマクロ `HZ` の値を 1,000 とし，カーネルタイマの精度を 10 倍の 1 ミリ秒に高めている．

5.1.4 多重化 I/O 呼び出しの最小間隔

永続コネクションを無効にした場合と有効にした場合における多重化 I/O 呼び出しの最小間隔の設定を**表 2** に示す．2 つの場合の設定の差を設けたのは，永続コネクションを有効にした場合に I/O スレッドが同時に扱わなければならないソケットの数が多く，多重化 I/O を含む主ループの処理時間が長くなるためである．また，どちらの場合においても，多重化 I/O の最小呼び出し間隔を 0 ミリ秒に設定するということは，I/O スレッドにおいて `nanosleep()` によるブロックをいっさい行わないことを意味し，従来の多重化 I/O のモデルに相当する[☆☆☆]．

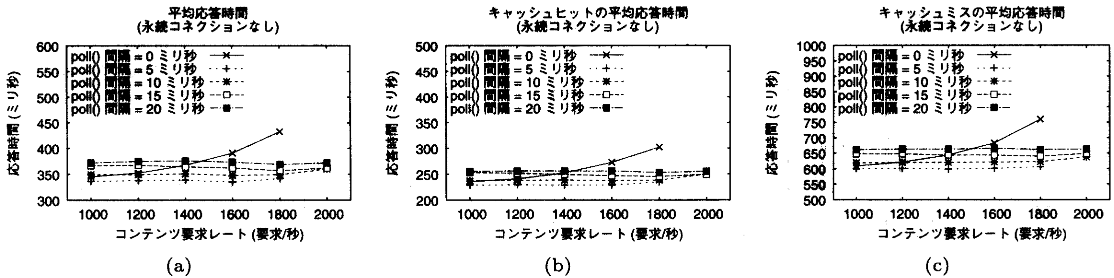
5.2 スループットと応答時間の比較

本節では，多重化 I/O 呼び出しの最小間隔がベンチマークテストの結果に与える影響を，スループットおよび応答時間において考察する．それぞれの分析で用いた実験結果は，WebAxe-4 ワークロードにおける最大要求レートを維持する期間の 1 つである top2 の期間全体の平均値を用いている（WebAxe-4 ワークロードについては付録 A.1 を参照）．

5.2.1 HTTP/1.1 永続コネクションを無効にした場合

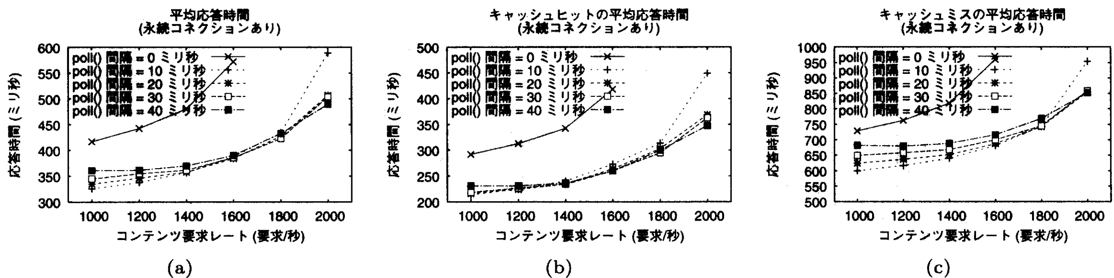
HTTP/1.1 永続コネクションを無効にした場合，1 つのコネクションにつき 1 つのコンテンツ要求が送ら

☆☆☆ 厳密に述べれば，本実装において多重化 I/O の呼び出し間隔を 0 ミリ秒に設定することは，多重化 I/O 自体をノンブロッキング化しているため，従来の I/O モデルとは異なる．しかし，リクエストレートが 1,000 を超えるような高い水準にある場合，主ループの実行中に I/O 可能になるソケットが発生することが多く，実質上は同じか，それよりオーバヘッドが小さい（4.2 節参照）．そのため比較対象として議論の公平性は損なわれない．なお，この点については，5.4 節でも議論する．



(a) (b) (c)

図5 HTTP/1.1 永続コネクションを無効にした場合の応答時間
Fig. 5 Response time in case with HTTP/1.1 persistent connections disabled.



(a) (b) (c)

図6 HTTP/1.1 永続コネクションを有効にした場合の応答時間
Fig. 6 Response time in case with HTTP/1.1 persistent connections enabled.

れ，処理が完了すると同時にコネクションは閉じられる．そのため，コンテンツ要求レートとコネクション確立レートはほぼ一致する．**図5**に，全応答，キャッシュヒット，キャッシュミスにおけるスループットと平均応答時間の関係を示す．

まずこれらのグラフから，スループット特性が多重化I/Oの呼び出し間隔制御によって向上しているのが分かる．制御を行わない場合と間隔を5ミリ秒に設定した場合は，コンテンツ要求レートを毎秒2,000要求に設定するとエラーが発生し，測定不能となった．一方で，間隔を10ミリ秒以上に設定した場合は，毎秒2,000要求のスループットを達成した．

次に，応答時間特性については，多重化I/Oの呼び出し間隔の制御により，コンテンツ要求レートの増加に対する性能のスケーラビリティが大きく向上することが判明した．制御をまったく行わなかった場合は，要求レートを増加させると応答時間も大きく増加しているが，多重化I/Oの呼び出し間隔を制御すると，応答時間は要求レートの増加に対してほぼ一定のまま推移している．これは，実行間隔制御が十分有効に動作していることを示している．一方で，多重化I/Oの間隔を大きくすればするほど，追加されたブロック時間のために応答時間自体は大きくなっている．間隔を10ミリ秒より大きくとると，負荷の比較的軽い局面で制御を行わない場合より応答時間が大きくなっている．

以上の結果からHTTP/1.1永続コネクションを無効にした場合，多重化I/Oの最適な間隔は10ミリ秒であることが判明した．負荷に応じて5ミリ秒と10ミリ秒で動的に間隔を制御する手法も考えられるが，得られる遅延時間特性の改善が10ミリ秒程度とエンドユーザの視点からはほぼ無視できる範囲であるため，多くの場合不要である．

5.2.2 HTTP/1.1 永続コネクションを有効にした場合

HTTP/1.1永続コネクションを有効にした場合，1つのコネクションにつき複数のコンテンツ要求が送られるため，コンテンツ要求レートとコネクション確立レートは一致しない．Chamomileでは，5.1節で述べたように，Web Polygraphとは独立して総コネクション数の閾値を7,000に設定し，この値を超えないようにコネクションタイムアウト時間を制御した．**図6**に，スループットと応答時間の関係を示す．

まず，スループット特性については，永続コネクションを無効にしていた場合と同様に向上しているのが分かる．多重化I/Oの呼び出し間隔制御を行わない場合の最大スループットは毎秒1,600要求となり，間隔を10ミリ秒以上に設定した場合は毎秒2,000要求のスループットを達成した．

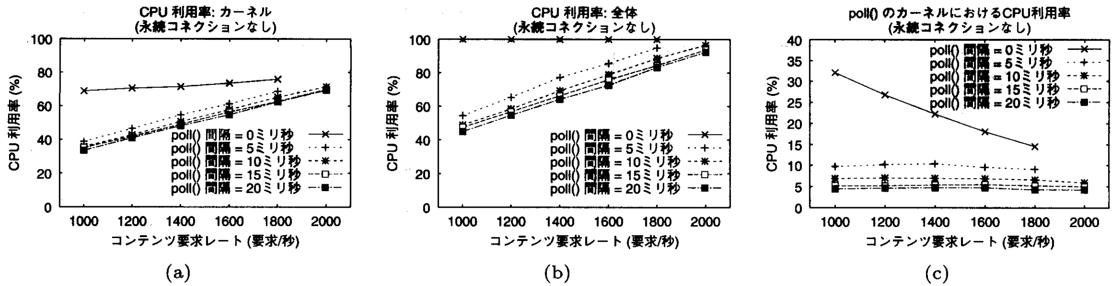


図 7 HTTP/1.1 永続コネクションを無効にした場合の CPU 利用率
Fig. 7 CPU utilization in case with HTTP/1.1 persistent connections disabled.

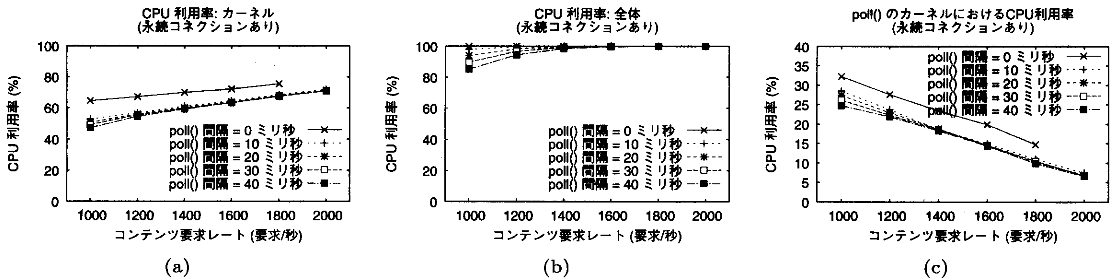


図 8 HTTP/1.1 永続コネクションを有効にした場合の CPU 利用率
Fig. 8 CPU utilization in case with HTTP/1.1 persistent connections enabled.

また，応答時間についても間隔の制御により大幅に改善されていることが分かる．特に，永続コネクションを無効にした場合と異なり，実験を行ったすべてのコンテンツ要求レートにおいて従来の方式と比較して応答時間が低下している．ここで興味深いのは，特に負荷の高い局面において，多重化 I/O の実行間隔の差による応答時間の変化があまり見られなかったことである．この点については後の 5.4 節で議論する．また，制御を行わない場合と比較して緩やかではあるが，要求レートが増加すると応答時間も増加する傾向が見られることが判明した．

以上より，HTTP/1.1 永続コネクションを有効にした場合の多重化 I/O の最適な間隔も 10 ミリ秒であることが判明した．負荷の高い局面では 20 ミリ秒以上に設定することも考えられるが，次の 5.3 節で述べるように，カーネルプロファイルの分析でもほとんど違いはなく，10 ミリ秒が最適値である．

5.3 カーネルプロファイル分析

カーネルプロファイルには，Linux 用カーネルプロファイラ Kernprof[14] バージョン 1.5 を用いた．また，それぞれの実験では，WebAxe-4 ワークロードの top2 の期間に 5 分間のプログラムカウンターのサンプリングを行った．

5.3.1 HTTP/1.1 永続コネクションを無効にした場合

永続コネクションを無効にした場合のカーネルプロファイリング結果を図 **7** に示す．図 7 (a) のグラフがカーネルの CPU 利用率を，図 7 (b) がカーネルおよびユーザプロセスの合計の CPU 利用率を示している．これらのグラフから，多重化 I/O の呼び出し間隔の制御を行わない場合は，カーネルの CPU 利用率が高く，合計ではコンテンツ要求レートに依存せず CPU をほぼ 100%消費していることが分かる．一方，制御を行う場合，要求レートの増加に応じた CPU 利用率の増加を実現している．

図 7 (c) のグラフは，カーネルが消費した CPU サイクルのうち，`poll()` システムコールの処理に要したものの割合を示している．従来の多重化 I/O の方式では，要求レートが上昇するに従って`poll()` に要する CPU サイクルの割合が低下しているものの，高い水準にある．一方で，呼び出し間隔の制御を行った場合は要求レートの上昇と関係なくカーネルによる CPU サイクル消費の 5〜10%程度に収まっており，正しく制御が行われていることを示している．

5.3.2 HTTP/1.1 永続コネクションを有効にした場合

永続コネクションを有効にした場合の CPU 利用率

を図8に示す．永続コネクションを無効にした場合と同様，図8(a)のグラフがカーネルのCPU利用率を，図8(b)がカーネルおよびユーザプロセスの合計のCPU利用率を示している．多重化I/Oの呼び出し間隔の制御を行わない場合の挙動は，永続コネクションを無効にした場合（図7(a)および図7(b)）とほぼ同様に推移している．一方で，制御を行う場合は大きく異なる挙動を示している．特にカーネルにおけるCPUサイクルの消費は，制御を行わない場合より10〜15%程度少ないものの，比較的高い水準で推移している．全体のCPU消費も，コンテンツ要求レートが毎秒1,400要求を超えるとほぼ100%消費している．

また図8(c)のグラフは，カーネルにおいて`poll()`システムコールの処理に要したCPUサイクルの割合を示す．これも，永続コネクションを無効にした場合（図7(c)）と異なる挙動を示している．従来の多重化I/Oの方式と比較して5〜8%程度少なく，要求レートが上昇するに従って`poll()`に要するCPUサイクルの割合が低下しているものの，カーネルにおいて6〜28%を`poll()`の処理が占めている．これらの挙動については次の5.4節で議論する．

5.4 議　論

これまでの評価結果から，本研究で提案する方式は全般的にサービス応答時間を大きく改善することが判明した．しかしながら，細部ではいくつか特異的な挙動を示した点もあった．ここでは，それらについて議論する．

(1) 呼び出し間隔の制御を行わない場合，コンテンツ要求レートが増加すると`poll()`のCPU消費が減少している（図7(c)および図8(c)）

要求レートが上昇すると，ソケットがI/O可能へ変化するイベントの頻度が増加する．そのため，多重化I/Oの呼び出し間隔の制御を行わない場合でも，ある点を超えると多重化I/Oにより得られる情報量が1ではなくなる．その後は，要求レートが上昇すればするほどこの情報量はさらに増加し，実際のI/Oに要する時間も増加する．その結果，制御を行わなくても実際には多重化I/Oを呼び出すレートが徐々に低下していくことになり，`poll()`の呼び出し回数自体は低下していく．

(2) 永続コネクションを有効にした場合，制御を行っても応答時間がコンテンツ要求レートに依存して上昇している（図6）

永続コネクションを有効にした場合，今回の実験では7,000ものソケットを同時に扱うため，多重化I/O自体の処理時間が長くなる．そのため，予期しないコンテキストスイッチが実行中に挿入されてしまう可能性が増加する．また，制御を行う多重化I/Oの実行間隔を大きく設定したが，これが各スレッドにおいて連続実行が許される最大時間であるタイムスライスを超えてしまい，コンテキストスイッチが発生する．こうしたコンテキストスイッチにより，実行間隔制御の有効性が低下し，CPU利用率の大幅な上昇および応答時間の増加につながっている．さらにこのことは，間隔を大きくしても遅延特性に大きな変化が見られなかったことの原因の1つでもある．

6. おわりに

本論文では，高性能サーバにおける多重化I/Oの処理負荷を軽減するために，多重化I/Oの呼び出し間隔を制御することで効率を向上する手法を提案した．本手法は，Chamomileの実装への適用で見たように，従来の多重化I/Oのプログラミングモデルをそのまま用いることが可能であることから，ネットワークサーバなどの実装に適用するのが容易である．

また，ベンチマークテストによる性能評価では，多重化I/Oの呼び出し間隔を10ミリ秒に設定しておけば，ほぼ最適な性能が得られることが判明した．その場合，スループットで11〜25%の向上が得られ，サービス遅延時間についても，同時ソケット数に応じて挙動が変化するものの，大きな改善が見られた．

一方で，多重化I/Oに与えるソケット数が十分大きい場合，オペレーティングシステムのスレッド管理による干渉により，期待したとおりの動作とはなっていないことも判明した．結果として良好な性能が得られているものの，より効率的なプロセッサ利用の実現などのために，リアルタイム指向な制御方式の導入が必要であろう．この点については今後の課題としたい．

謝辞 本論文の執筆にあたり，匿名査読者諸氏から有益なコメントを数多くいただいた．ここに感謝する．

参 考 文 献

1) Stevens, W.R.: *UNIX network programming*, second edition, Vol.1, Prentice Hall PTR (1998).
2) McKusick, M.K., Bostic, K., Karels, M.J. and Quarterman, J.S.: *The Design and Implementation of the 4.4 BSD Operating System*, Addison-Wesley (1996).
3) Gallmeister, B.O.: *POSIX.4: Programming for the Real World*, O'Reilly & Associates, Inc. (1995).

4) Banga, G., Mogul, J.C. and Druschel, P.: A scalable and explicit event delivery mechanism for UNIX, *USENIX Annual Technical Conference*, pp.253–265 (1999).
5) Lemon, J.: Kqueue: A generic and scalable event notification facility, *2001 USENIX Annual Technical Conference* (2001).
6) Provos, N. and Lever, C.: Scalable Network I/O in Linux, *2000 USENIX Annual Technical Conference* (2000).
7) Acharya, S.: Using the devpoll (/dev/poll) Interface, Technical Articles (2002). http://access1.sun.com/techarticles/devpoll.html
8) Provos, N., Lever, C. and Tweedie, S.: Analyzing the Overhead Behavior of a Simple Web Server, *4th Annual Linux Showcase & Conference* (2000).
9) Chandra, A. and Mosberger, D.: Scalability of Linux Event-Dispatch Mechanisms, *2001 USENIX Annual Technical Conference* (2001).
10) Chamomile Web Server Accelerator. http://www.chamomile-proxy.org/
11) The Measurement Factory: Web Polygraph. http://www.web-polygraph.org/
12) Mosberger, D. and Jin, T.: httperf — A Tool for Measuring Web Server Performance, *SIGMETRICS Workshop on Internet Server Performance* (1998).
13) The Measurement Factory: WebAxe-4 Workload. http://www.web-polygraph.org/docs/workloads/webaxe-4/
14) Silicon Graphics: Kernprof (Kernel Profiling). http://oss.sgi.com/projects/kernprof/
15) Rizzo, L.: Dummynet: a simple approach to the evaluation of network protocols, *ACM Computer Communication Review*, Vol.27, No.1, pp.31–41 (1997).

付　録

A.1 Web Polygraph と WebAxe-4 ワークロード

Web Polygraph[11] は，あらかじめ定義されたワークロードに基づいて Web プロキシキャッシュサーバのベンチマークテストを行うツールである．テストでは，クライアントクラスタおよびサーバクラスタが用いられる．Web アクセラレータのテストには，WebAxe-4[13] と呼ばれるワークロードを用いる．ここでは，この WebAxe-4 ワークロードの性質について簡単に説明する．

A.1.1 コンテンツ要求レート

図 9 に，WebAxe-4 を用いたテストにおけるコンテンツ要求レートの変化を示す．主要なテスト期間として，fill，top1，top2 の 3 つがある．

fill は，アクセラレータが十分な量のコンテンツを蓄積し，その後のキャッシュヒット率が定常状態になるようにするための期間である．一般的にテスト開始直後はキャッシュにコンテンツが蓄積されておらず十分なヒット率が達成できない．そのため，オリジン

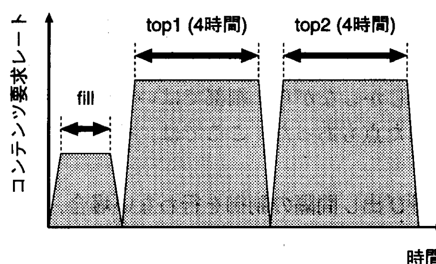


図 9 WebAxe-4 におけるコンテンツ要求レートの時間的変化
Fig. 9 Preset request load of WebAxe-4 workload.

表 3 WebAxe-4 における要求および応答
Table 3 Requests and responses defined in WebAxe-4.

メソッド	GET : 98.4%, POST : 1.5%, HEAD : 0.1%
要求に付加される条件	IMS 要求 † に応答コードが 200 : 5%, IMS 要求に応答コードが 304 : 10%, キャッシュ不可（リロード）: 5%, なし : 80%

†IMS 要求 : ヘッダに “If-Modified-Since” 行がある要求

表 4 WebAxe-4 におけるコンテンツタイプ
Table 4 Content types defined in WebAxe-4.

タイプ	応答サイズ分布	キャッシュ可能率	全要求に占める率
image	指数分布（平均 4.5 KB）	80.0%	65.0%
HTML	指数分布（平均 8.5 KB）	90.0%	15.0%
download	対数正規分布（平均 300 KB，標準偏差 300 KB）	95.0%	0.5%
other	対数正規分布（平均 25 KB，標準偏差 10 KB）	72.0%	19.5%

サーバへ多くの要求を転送しなければならず，アクセラレータの負荷が高くなる．こうした状況を考慮し，WebAxe-4 ではこの fill の間コンテンツ要求レートが低めに設定されている．

top1 および top2 はそれぞれ 4 時間のテストであり連続して 2 回行われるが，基本的には同一の条件である．そこでは，クライアントクラスタは設定されたレートでさまざまなコンテンツ要求を期間中送り続ける．なお，本研究において評価で用いたのは，top2 の期間に計測されたアクセラレータの性能である．

A.1.2　HTTP トランザクション

WebAxe-4 における HTTP トランザクションの特徴については，**表 3** および**表 4** にまとめた．表 3 は，コンテンツ要求・応答についての種類およびその割合であり，表 4 は，クライアントから要求されるコンテンツの特徴である．これらの性質を持つ要求，応答，コンテンツ集合を用いて HTTP トランザクションがベンチマークテストされる．また，ワーキングセット（頻繁にアクセスされるコンテンツ集合）の合計サイズは 1 GB に設定される．

A.1.3　ネットワーク環境

クライアントクラスタとアクセラレータの間には，クライアントクラスタが稼働する FreeBSD 上の Dummynet [15] を用いて，平均 40 ミリ秒の遅延と 0.005%のパケットロス率が双方向にそれぞれ設定される（本文図 4 を参照）．また，オリジンサーバクラスタにおいては平均 300 ミリ秒，標準偏差 100 ミリ秒の正規分布に従うサービス遅延時間が設定されている．テスト中に発生するネットワークトラヒックは，アクセラレータにおけるキャッシュヒット率に左右される．本研究で行ったテストにおいては，キャッシュ容量を一律 768 MB に設定しヒット率を安定させたため，トラヒックはコンテンツ要求レートにほぼ比例する結果となった．具体的には，毎秒 1,000 要求のスループットあたり 45 Mbps 程度のトラヒックがクライアントクラスタとアクセラレータの間で発生した．

(平成 15 年　4 月 14 日受付)
(平成 15 年 12 月　2 日採録)

河合　栄治（正会員）

1996 年京都大学理学部数学科卒業．1998 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．2001 年同大学同研究科博士後期課程修了．2000 年 10 月より，科学技術振興事業団さきがけ研究 21「機能と構成」領域研究員．2003 年 4 月より，奈良先端科学技術大学院大学附属図書館研究開発室科助手．分散計算環境の構築，インターネットにおける大規模情報配信システムの開発，高速 I/O 指向オペレーティングシステムの研究に従事．博士（工学）．

門林　雄基（正会員）

1996 年大阪大学大学院基礎工学研究科物理系専攻博士後期課程中退．同年大阪大学大型計算機センター助手．1997 年大阪大学大学院基礎工学研究科物理系専攻情報工学分野より博士（工学）取得．1999 年大阪大学大型計算機センター講師．2000 年奈良先端科学技術大学院大学情報科学研究科助教授．WIDE プロジェクトボードメンバ．Content Routing Network Forum 代表．レイヤ 7 での QoS 実現を目標とし，セキュリティ，CDN，マルチキャスト等の研究に従事．著書に岩波講座インターネット第 2 巻『ネットワークの相互接続』．

山口　英（正会員）

1990 年 10 月大阪大学大学院基礎工学研究科情報工学専攻博士後期課程を中退し，大阪大学情報処理教育センター助手として着任．1992 年 10 月奈良先端科学技術大学院大学情報科学センター助教授．1993 年 4 月より，同大学情報科学研究科助教授．2000 年 4 月より，同大学情報科学研究科教授．大規模分散処理環境構築，ネットワークセキュリティ等の研究を行う．また，WIDE Project のメンバーとして，広域コンピュータネットワークの構築・研究に従事する．工学博士．