アクセス集中時の Web サーバの性能に対する OS の影響

日比野 秀章 † 松沼 正浩 † 光来健一 † 千葉 滋 †
Hideaki HIBINO Masahiro MATSUNUMA Kenichi KOURAI Shigeru CHIBA

† 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Dept. of Mathematical and Computer Sciences, Tokyo Institute of Technology
{hibino,matsunuma,kourai,chiba}@csg.is.titech.ac.jp

我々は、アクセス集中時においても、安定してサービスを提供することができる Web サーバのアプリケーションレベル・スケジューラを開発している。Web サーバは、様々な環境で稼動されており、サーバの性能はマシン性能や OS に大きく依存する。本論文では、アクセス集中時の Web App サーバの性能を測定することで、性能が実際に OS に大きく影響されることを示す。具体的には、複数の OS 上で Web App サーバとしてよく知られる Tomcat を動作させ、Tomcat に負荷の異なる複数の servlet を処理させる実験を行った。実験の結果、OS の違いにより、servlet の処理性能に無視できない大きな差が出ることを確認した。既存の OS レベルスケジューラだけで安定したサービスを提供するのは困難であり、アプリケーションレベルスケジューラを用いて性能の改善を試みる余地があることを示す。

1 はじめに

現在、インターネット上で様々なサービスを提供するため、Web アプリケーション (App) サーバが広く使われている。例えば、Tomcat[1] のように Java で実装された Web App サーバでは、同じアプリケーションを用いて様々な実行環境で同じサービスを提供することができる。しかし、Web App サーバのハードウェアが同一であっても、その処理性能は OS 等の実行環境の影響を強く受けてしまう。特に、我々の実験によれば、高負荷時の処理性能は実行環境によって大きく異なる。そのため、様々な実行環境において処理性能も含めて等価なサービスの提供を実現するには、開発者および使用者がそれぞれの実行環境に合わせたチューニングをしなければならない。この作業は膨大な時間のかかるものなので、マルチプラットフォームのアプリケーションであれば、このような実行環境による影響は極力避けたい。

本稿では、実行環境のどの部分が高負荷時の Web App サーバの処理性能に最も影響を及ぼしているかの原因追求を行った。各実行環境で考えられる違いはネットワーク処理性能、JVM のガーベジコレクション、サーバ内の servlet 処理部分である。この実験の結果、サーバ内で TCP コネクションを張った後、servlet 処理が開始されるまでの部分がボトルネックになっていることが分かった。この部分に影響を及ぼしているのは主に OS のスケジューラであると推測される。実際、OS のスレッドスケジューラを改良すると、単純な Web サーバの性能が大きく向上することが多くの研究で示されている [2][3]。この点からも我々の推測は妥当なものといえよう。

そこで、我々はアプリケーションレベルのスケジューラを用いて、OS レベルのスケジューラの違いを隠蔽できることを確かめる実験を行った。この実験ではリソースを大量に使用する servlet を一定時間停止させることにより、擬似的にアプリケーションレベルのスケジューリングを実現した。その結果、実行環境による Web App サーバの性能差を縮められる可能性があることが示された。

2 実行環境による処理性能の差異

まず、OS の違いによって、高負荷時の Web App サーバの処理性能にどのような差異があらわれるかを検証するために実験をおこなった。我々は、 2 つのジョブに対して多数のリクエストを異なる割合でサーバに投げ、それぞれのジョブに対するリクエストの処理数を測定した。フィボナッチ数 (引数 25) の計算を行う処理 (単純な数値計算) を job A と呼び、XML ファイルをパースし、DOM ツリーについて全ノードの探索を 100 回行う処理 (大量のメモリを消費する処理) を job B と呼ぶ。測定対象のサーバの OS として、Solaris 9、Linux 2.6.6、Windows 2003 Server Enterprise Edition、FreeBSD 5.2.1 の 4 つを用いた。

実験では、クライアントマシン上で、job A 用に

スレッドを 30 個作り、各スレッドに job A へのリクエストをサーバに投げさせた。各スレッドは、サーバから処理完了の応答があると、再びリクエストをサーバに投げ、これを永遠に繰り返す。以下、本論文では、このような処理を job A に対して常時 30 のリクエストを投げさせる、という。一方、job B に対しては常時 1, 5, 10, 20, あるいは 40 のリクエストを投げさせた。この状態で 2 分間動かし、その間にサーバが処理できたリクエストの数 (処理数) を測定した。なおクライアントマシンは 14 台用い、スレッドを分散配置して実験をおこなった。

実験に使った Web App サーバーは Tomcat (version 5.0.25) である。JVM (Java 仮想機械) としては Sun JDK 1.4.2 を用いた。クライアントマシンとして、Pentium 733MHz (シングル・プロセッサ)、メモリ 512MB、100baseTX ネットワークを搭載したものを 14 台用いた。OS は Linux 2.4.19-Ovl11 (Vine Linux) である。一方、サーバマシンとしては、Sun Fire V60x (Intel Xeon 3.06GHz デュアル、2GB メモリ、1000BaseTX ネットワーク) を 1 台用いた。

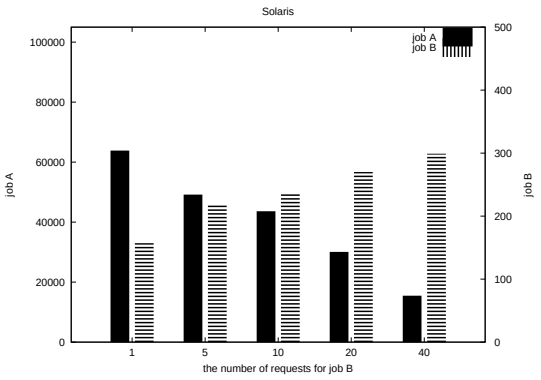

図 1: job A と job B の処理数 (Solaris 9)

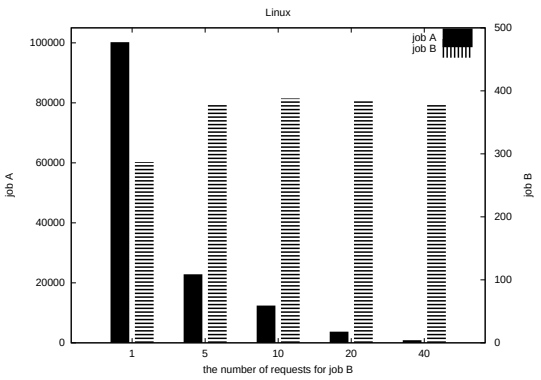

図 2: job A と job B の処理数 (Linux 2.6)

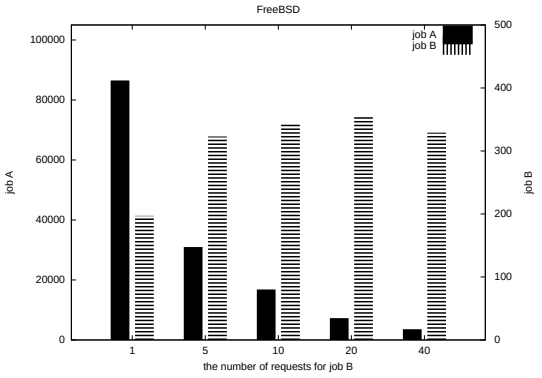

図 3: job A と job B の処理数 (FreeBSD 5.2)

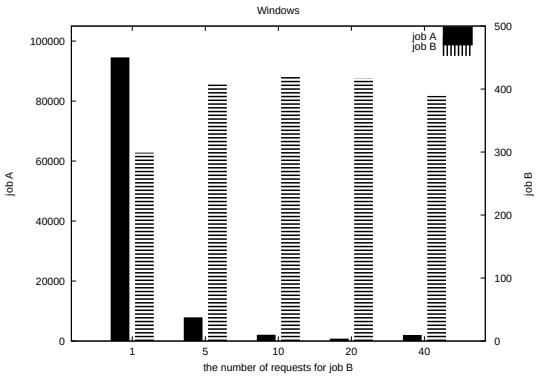

図 4: job A と job B の処理数 (Windows 2003 Server)

OS ごとの測定結果は図 1〜図 4 のようになる。図から、Linux, FreeBSD, Windows の 3 つはほぼ同様の挙動を示し、job B のスレッド数を増やすと、job A の処理数が急激に落ち込み、job B のスレッド数が 40 のときの処理数は 1 のときの処理数の 1/20 以下になることがわかる。一方、Solaris の挙動は異なり、job B のスレッド数を増やしても、job A の処理数はあまり落ち込まない。スレッド数 40 のときの処理数は 1 のときに比べて約 1/4 である。

以上の結果から、Solaris 以外の OS では、高負荷時に重い job B が Solaris より多く処理される一方で、軽い job A がほとんど処理されないという現象が観察される。どちらの job をどのくらい処理するべきかはアプリケーションによるが、Web App サーバの場合、高負荷時にも軽い job が一定数以上処理されるべきである。Web App の利用者は、軽い job はすぐ反応があることを期待するので、高負荷時に反応が著しく悪くなると、より不満を感じやすい。

表 1: netstat による統計情報

	Solaris		Linux		FreeBSD		Windows	
	job A	job B	job A	job B	job A	job B	job A	job B
接続数	28171	669	1977	834	5945	778	3570	958
接続失敗回数	0	0	0	0	0	0	0	0
受信セグメント数	140996	3613	10011	4372	29317	4111	17602	5049
送信セグメント数	141196	3723	10080	4485	29981	4218	18145	5208
再送回数	0	7	0	2	2	4	9	14

表 2: job A と job B における GC の挙動

	job A				job B			
	GC の回数	減少ヒープ量の和 (GB)	停止時間 (秒)	処理数	GC の回数	減少ヒープ量の和 (GB)	停止時間 (秒)	処理数
Solaris	1333	2.73	7.3	37906	5044	19.5	33.7	742
Linux	353	0.51	1.3	70045	5946	19.8	27.3	765
FreeBSD	236	0.36	2.2	52633	4915	15.7	37.8	603
Windows	936	1.65	5.0	52072	10244	21.3	35.3	771

3 実行環境による性能差の要因

本章では、2 章で観察された性能差の要因について追究するために行った実験について述べる。実験はネットワーク性能、JVM のガベージコレクション性能の影響について調べた。また、サーバ内の処理時間を 3 つの区間にわけて測定し、ボトルネックを推測した。

3.1 実行環境によるネットワーク処理への影響

まず、性能差の要因が各 OS でのネットワーク処理の違いによるか否かを検証するための実験をおこなった。実験には、2 章と同じ job A, job B を用い、job A には常時 30 のリクエストを、job B には常時 40 のリクエストを 3 分間投げ続ける。job A には 5 台、job B には 9 台のクライアントマシンを割り当てた。その後、クライアントマシンの側で netstat を用いてパケットの送受信の統計情報を調べた。

得られた統計情報を表 2 に示す。どの OS の場合も再送はほとんど起きておらず、コネクションの失敗も起きていない。したがって、OS によるネットワーク処理の違いが、性能差の原因であるとは考えにくい。

3.2 ガベージコレクション

JVM の実装は OS によって異なるので、GC (ガベージコレクション) の実装も OS ごとに大きく異なる可能性がある。そこで GC の性能が全体の性能差に与える影響を調べるための実験をおこなった。実験では、job A にのみ 80 クライアントから 1 分間常時リクエストを投げつづけた場合と、job B にのみ 80 クライアントから 4 分間常時リクエストを投げた場合について、java の logcc オプションを用いて GC の情報を集め、解析を行った。

解析結果は表 2 のようになった。GC の頻度とリクエストの処理数の関係は、OS 毎にかなり違いがあり、一貫性はみられない。job B へのリクエストの増加により job A へのリクエストの処理数が急激に減少してしまう Linux, FreeBSD, Windows で共通した傾向が見られないので、GC の実装の違いが、OS 間の性能差に大きな影響を与えているとは考えにくい。

3.3 サーバ内でのボトルネック

サーバ内のどの部分の処理に時間がかかっているのか調べるために、処理時間を 3 つの区間に分けて計測した。3 つの区間とは、SYN パケットを受け取ってからシステムコール accept を実行するまで (Syn)、accept からジョブの開始まで (Accept)、ジョブの開

表 3: 処理時間の分布

	Syn			Accept			Job		
	最小	中央	最大	最小	中央	最大	最値	中央	最大
Solaris(job A のみ)	0	0	0	0	0	590	0	0	20
Solaris(job A と job B)	0	270	2140	0	230	3090	0	0	660
Linux(job A のみ)	0	0	350	0	0	10010	0	0	20
Linux(job A と job B)	0	80	2410	20	13180	1650	0	0	120

始から終了まで (Job) である。実験では、1 分間 job A にのみ常時 30 のリクエストを投げた場合と、2 分間 job A に常時 30、job B に常時 80 のリクエストを投げた場合について、job A の処理時間を区間別に測定した。測定は、accept システムコールのトラップが可能であった Solaris と Linux でのみ行った。また SYN パケットの獲得には tcpdump を利用した。

結果を表 3 に示す。job A にのみリクエストを投げた場合と、job A と job B にリクエストを投げた場合とで、各区間の処理時間を比較する。すると、Linux では accept からジョブの開始までの処理時間が大きく増大しているが、Solaris ではこの区間の処理時間はあまり増大していないことがわかる。このことから、この区間の処理のやり方が OS によって大きく異なり、それが全体の性能差につながっているといえる。この区間の処理内容から、2 章で観察された性能差は、主に OS のスケジューリングのやり方に由来すると推測できる。

4 アプリケーションレベルスケジューラ

このような OS レベルでのスケジューリングの問題を解決するために、我々はアプリケーションレベルのスケジューラが有効だと考える。各 OS にサーブレット処理のスケジューリングをまかせてしまうと、2 章で示したように Solaris 以外の OS では急激に処理性能が低下してしまうサービスが出てくる。アプリケーションレベルスケジューラにより、どのような OS を用いていても、各サービスに均等にリソースが割り当てられるようにすれば、OS に依存せずに他のサービスによる影響をなるべく抑えて安定したサービスを提供することが可能になると考えられる。

アプリケーションレベルスケジューラの有効性を確認するために、今回の実験でリソースを大量に使用している job B を一定時間停止させるようにして、job A と job B の処理数の変化を測定した。job B の CPU 時間を job A に譲ることにより、job B のリクエスト数増加による影響を減らすことができる。job B を一定時間停止させるために、job B のコードに sleep メソッドを挿入した。この実験では、sleep メソッドを挿入しない場合 (S0)、job B の前後に sleep メソッドを挿入して 1 つの job B あたり 2、4、6、8、10 秒ずつ停止させる場合 (S2, S4, S6, S8, S10)、job B の毎回の探索の終わりに 20、40、60、80、100 ミリ秒ずつ sleep メソッドを挿入して 1 つの job B あたり 2、4、6、8、10 秒停止させる場合 (S2', S4', S6', S8', S10') について測定を行った。測定は job A を 30 スレッド、job B を 20 スレッド動かし、2 分間計測した。

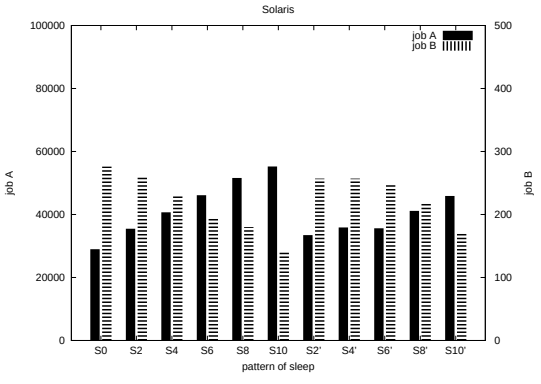

図 5: job B に sleep メソッドを入れた場合の処理数 (Solaris 9)

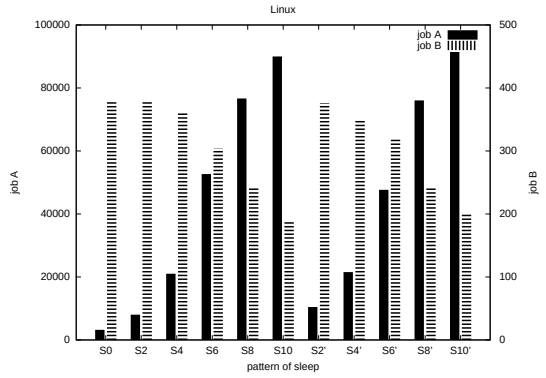

図 6: job B に sleep メソッドを入れた場合の処理数 (Linux 2.6)

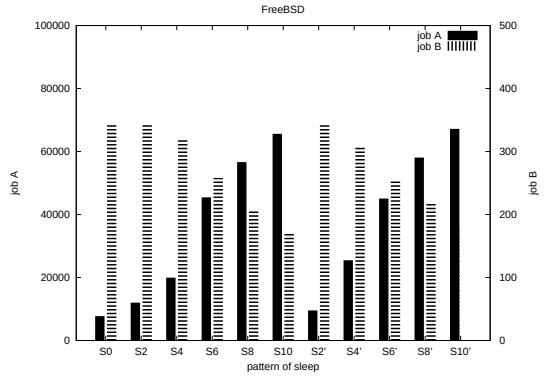

図 7: job B に sleep メソッドを入れた場合の処理数 (FreeBSD 5.2)

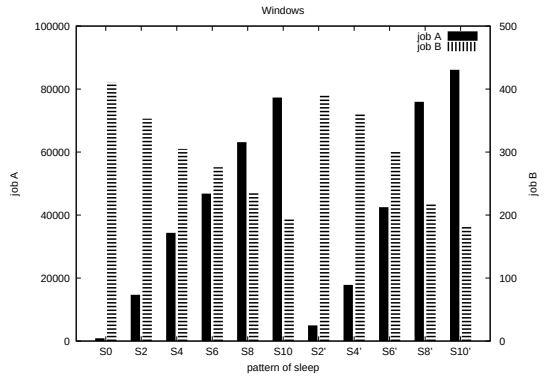

図 8: job B に sleep メソッドを入れた場合の処理数 (Windows 2003 Server)

測定結果を以下の図 5〜図 8 に示す。Linux では job B の前後に sleep メソッドを入れて 4 秒停止させても、job B の処理数は 5%の減少に抑えた上で、job A の処理数を 6.5 倍に増加させることができている。FreeBSD でも job B を 4 秒停止させると、job B の処理数が 8%減少するのに対し、job A の処理数は 2.6 倍になっている。一方、Windows では job B を 2 秒停止させると、job A の処理数は 17 倍になるものの、job B の処理数は 14%低下するという結果になった。また、job B に sleep メソッドを挿入しなくても job A がある程度処理されてていた Solaris では、job B を 4 秒停止させても job A の処理数は 1.4 倍になるにとどまった。この時、job B の処理数は 16% 低下していた。ちなみに、sleep メソッドを job B の前後に入れた場合と、毎回の探索の終わりに細かく入れた場合とではあまり違いは見られなかった。ただし、Solaris では細かく sleep メソッドを入れるとスケジューリングの効果が現れにくくなるという現象が見られた。

5 まとめ

本稿では、Web アプリケーションサーバの処理性能が OS 等の実行環境の影響を強く受けてしまう場合があることを実験により示した。そして、いくつかの実験を行うことにより、その原因が OS のスケジューラの違いによる可能性が高いと結論づけた。さらに、OS レベルのスケジューラの違いを隠蔽するのに、アプリケーションレベルスケジューラが有効であること示した。

今後の課題は、さらに詳細な実験を行うことにより Web アプリケーションサーバの処理性能の違いが OS のスケジューラにあるという確証をつかむことである。さらに、アプリケーションレベルスケジューラを開発し、どのような実行環境でも希望する性能を得られるように、各サービスへの CPU 割り当てを自動的にコントロールできるようにしていく。

参考文献

[1] Tomcat. http://jakarta.apache.org/tomcat/.

[2] Rob von Behren, Jermy Condit, Feng Zhou, George Necula, and Eric Brewer, “Capriccio: Scalable Threads for Internet Services”, In *Proc. of 19th ACM Symp. on Operating Systems Principles (SOSP’03)*, pp. 268–281, 2003.

[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 (SOSP-18)*, 2001.