Tonegawa:
Highly Scalable Distributed Web Server with Data Stream Processing

Tomoaki OIKI *1,
Toyotaro SUZUMURA *1 *2

*1 Tokyo Institute of Technology,
*2 IBM Research - Tokyo
Agenda

- Background
- What is Tonegawa?
- Related Work
- Outline of System S
- Architecture of Tonegawa
- Evaluation
- Conclusion
Background

• The data traffic in the web is increasing.
• Each request demands large computation power. (ex. SSL, Dynamic contents)
• Enhancing single machine power is difficult.
• There are various hardwares such as PS3, SSL acceralator, and FPGA board.

→ We need distributed web servers utilizing various hardwares!
What is Tonegawa?

• A distributed web server on top of SystemS.
  – System S : middleware for data stream processing
• Each request is processed over multiple nodes (read in node A, parse in node B)
  – Feeds specific task to a node which is good at the task (e.g. SSL accelerator)
• Can easily change task assignment across the nodes
  – We can control load imbalance among multiple nodes
Related Work

• Load balancing
  How to distribute load to multiple nodes
  – DNS Round Robin
  – Rewriting Packet

• Web Server Architecture
  How to handle concurrent requests
  – Thread based Concurrency
  – Event-driven Concurrency
  – Staged Event-driven Concurrency
DNS Round Robin

Single hostname is associated with multiple IP addresses.

- Even though the servers range over a large area, this method does not increase overhead.
- Clients' browsers and intermediate DNS servers occasionally cache IP addresses.

→ we often fail to implement intended load balancing with DNS.
Rewriting Packet

- All of requests are sent to the intermediate server, and then requests are passed to each of distributed servers.
- The intermediate server rewrite IP source addresses of responses as if responses were sent from the intermediate server.

All tasks for single request such as parsing request and reading file are processed on single node.
Single thread for single request

- Receives requests by the main thread and transfer each request to each thread.
  - Easy to program
  - Many concurrent requests cause performance degradation.
    - Context switches
    - TLB misses
    - Cache misses
Single thread for multiple requests

Read A → Read B → Parse A → Parse B → Read C ...

Few processes handle parts of requests one by one sequentially to avoid increasing processes and threads.

- Each request has a state such as read and parsed.
- Since there are few processes, contexts switches decrease.

This architecture has problems
- IO blockings causes performance degradation
- Difficult to implement and difficult to achieve modularity
Multiple threads for each stage

- SEDA [Welsh, SOSP'01] (Staged Event-Driven Architecture) splits processing of a request to multiple stages, each of which is processed with a limited number of threads.
- Achieves high throughput by avoiding additional costs for many IO wait and context switches.
- This architecture is used in Haboob [Welsh, SOSP'01].

You create many servers and transfer requests to each server by rotation for load balancing.
Our architecture

- Tonegawa also processes a request by splitting the tasks of the request to multiple stages.
  - Each stage can be processed on various nodes, and single stage can be processed on multiple nodes.
  - This load balancing method can be combined with traditional load balancing methods.
System S
-Stream Processing Infrastructure-

• A stream processing infrastructure developed by IBM Research

• Good at handling continuous data source
  – Stream: data queues maintained during execution of the program
  – Operator: unit of processing in System S

• Program is represented as network of operators and data streams between operators.

• Assign tasks to nodes by assigning operators to nodes.

• We can separate tasks assignment from implementation.
  • We assign tasks to nodes with a dedicated language.
  • Concrete execution logic for operators is written in C++.
SPADE
Stream Processing Application Declarative Engine programming language and compilation infrastructure.

- Dedicated language serves various functions.
  - Draw data-flow graph among nodes
  - Assign operators to nodes
  - Assign multiple operators to single process.
    - In a same process, operators are communicated with each other through memory-pointer.(low cost)
- Has many built-in *Operators*
  - *Functor* - is used for performing tuple-level manipulations such as filtering and mapping.
  - *Split* - is used for routing incoming tuples to different output streams based on a specified condition.
- Users can create user-defined operators(UDOP).
Each code block defines an operator, which has output/input streams.

An operator can output multiple streams.
Tasks assignment in SPADE

- Assign operators to nodes with `node` syntax.
- Assign operators to processes with `partition` syntax.
  - Operators having a same `partition` name are assigned to a same process.
The operators surrounded by a green square are added corresponding to the number of nodes.

- We implemented the server components of Tonegawa by extending *mini_httpd*[^1], which has a small source code.
- We added event handling with a system call named *epoll*, HTTP Keep-Alive and embedded Perl parser to our server.

Evaluation
Software Productivity

• System S hides concrete communication layer among operators.

• Flexible and easy tasks assignment
  • Decide the number of nodes and which node to use by SPADE language.
  • Tonegawa has 92 lines of SPADE code and 3491 lines of C++.

• Since the shared data among operators is described as streams, we get high independence and reusability of operators.

• The operator processing a request corresponds to the state of the request such as parsed and read, so we do not have to explicitly manage the states.
Performance Evaluation
-Configuration-

- Web servers  8 nodes
  - Cent OS 5.2 Linux kernel 2.6.18
  - Athlon 1640B CPU 2.7GHz Memory 1GB
  - IBM InfoSphere Streams Version 1.0.1.0
- Client  1 node
  - Cent OS 5.2 Linux kernel 2.6.18
  - Athlon X2 5050e 2.6GHz Memory 4GB
- Network
  - All nodes are connected with each other through 1Gbps LAN system
- Targets
  - Apache : Apache HTTP Server 2.2.13 with mod_perl
  - Single : Tonegawa on one process in single node
  - Multi : Tonegawa on 8 nodes
Static Contents

The client repeatedly requests a 1KB file from 1 thread to 1024 threads.

- The number of context switches in apache is 40-130 times more than that in single.
- The throughput of multi is only about 1.7 times higher than that of single despite existence of 8 nodes.
- When there are more than 3 nodes, the node running common operator such as Accepter is saturated.
Throughput with various file sizes

We request files from 128 threads which have power of 2 sizes from 1 KB to 256 KB.

- The data traffic approached 1 Gbps in the range from 16 KB to 256 KB at *apache* and *single*.
- Large size files reduce the throughput of *multi*, because data transfer corresponding to the file size occur among operators in *multi*,
Dynamic Contents

- We request Perl scripts from 128 threads that sums from 1 to specified number ranged from 1000 to 10000.

- The throughput of *multi* is increased linearly with the number of nodes.

- The cost of calculation for scripts overwhelm the cost of communication among operators.
Conclusion

- Tonegawa is distributed at smaller granularity level than traditional servers.
- We can delegate tasks assignment and communication among operators to SystemS.
- Since Tonegawa avoids spawning too many threads, it on single node is an efficient web server.
- If the requested file requires large computational effort, Tonegawa on multiple nodes utilizes nodes efficiently.
Future direction

• We have to test various practical situations using standard benchmarks such as SPECWeb2005.
• We expect that Tonegawa on multiple nodes will utilize multiple nodes by using sendfile system call when large size files are required.
• We need to test a heterogeneous environment such as SSL hardware appliance and FPGA.
Related Work

- Web Server Architecture
  - Apache HTTP Web server
  - Lighttpd
  - Haboob (SEDA)
- Distributed Web Server
  - DNS
  - Rewriting Packets
- Stream Processing
  - StreamIt
  - Borelias
以下、省略したスライド
Future direction

- We have to test various practical situations using standard benchmarks such as SPECWeb2005.
- We expect that Tonegawa on multiple nodes will utilize multi nodes by using sendfile system call when large size files are required.
- We need to test a heterogeneous environment such as SSL hardware appliance and FPGA.
分散に関する関連研究

• 従来は DNS などのデータ並列をもちいた分散
• Tonegawa ではタスク並列を用いることで、ヘテロ環境のメリットを利用できる
• Tonegawa のタスク並列はデータ並列と組み合わせることも可能
Operator

- System S ではオペレーターという単位で処理を扱う。
- データを受け取り加工し、次へ送る
- 様々なオペレーターが定義されている
  - Source – ファイルやソケットからデータを読む
  - Aggregate – データの集計を行う
- ユーザーは C++ で定義可能 Udop（ユーザー定義オペレーター）
- オペレーターはノードやプロセスに割り当てられる
ソースコードが小さく移植が容易な、mini_httpd を改変して実装した。

・Virtual Host 、SSH 、POST メソッド対応 、BASIC 認証といった今回、重要でない機能は省いた。

・mini_httpd のサポートしていない、コネクションハンドリングにおける epoll でのイベント管理、Keep-alive 、Perl Parser の組み込みを追加した。

・既存の Web Server と比較して、sendfile 、ヘッダのキャッシュ、ファイル情報のキャッシュ、Perl CGI のパース結果のキャッシュなどの最適化を行っていない。
性能評価

- 以下のテストを実行した
  - 1KB のファイルを 1 から 1024 スレッドから要求した
  - 1KB から 256KB のファイルを 128 スレッドから要求した
  - 負荷を変化させた Perl スクリプトを 128 スレッドから要求した
静的コンテンツ結論

- コンテキストスイッチを抑制できている
- ファイルサイズが大きくなるとかなりはやい段階からマルチノードの性能は低下
- マルチノードがいかせていない
コンテキストスイッチに関する性能分析

• コンテキストスイッチが Apache より Single の方が少ないことが、スループットの減少に貢献していると思われる

<table>
<thead>
<tr>
<th>並列アクセス数</th>
<th>single</th>
<th>apache</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>340</td>
<td>18045</td>
</tr>
<tr>
<td>8</td>
<td>305</td>
<td>38174</td>
</tr>
<tr>
<td>16</td>
<td>296</td>
<td>37834</td>
</tr>
<tr>
<td>32</td>
<td>305</td>
<td>11676</td>
</tr>
<tr>
<td>64</td>
<td>296</td>
<td>11316</td>
</tr>
<tr>
<td>128</td>
<td>305</td>
<td>11341</td>
</tr>
<tr>
<td>256</td>
<td>314</td>
<td>37538</td>
</tr>
<tr>
<td>512</td>
<td>301</td>
<td>37375</td>
</tr>
<tr>
<td>1024</td>
<td>300</td>
<td>37334</td>
</tr>
</tbody>
</table>
Multi のノード数によるスループットの変化

<table>
<thead>
<tr>
<th>ノード数</th>
<th>スループット</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22952</td>
</tr>
<tr>
<td>2</td>
<td>38964</td>
</tr>
<tr>
<td>4</td>
<td>36591</td>
</tr>
<tr>
<td>8</td>
<td>36548</td>
</tr>
</tbody>
</table>

2 ノード使用している時点で
スループットの上昇はストップ

現在の実装では、リクエストの
受信とレスポンスの送信は一つ
のノードが担当しているため、
そこがボトルネックとなっている
動的コンテンツテスト

・クライアントから指定された数まで足し合わせる単純な Perl スクリプトを要求する。
・Apache ではスクリプトのコンパイル結果のキャッシュを行わない設定にしている
ノード数ごとのスループット比較

・ノード数に比例してスループットが上昇

Perl スクリプト内の足し算回数