Improving Asynchronous Invocation Performance in Client-server Systems

Shungeng Zhang[†], Qingyang Wang[†], Yasuhiko Kanemasa[‡]
[†]*Computer Science and Engineering, Louisiana State University*
[‡]*Software Laboratory, FUJITSU LABORATORIES LTD.*

***Abstract*—In this paper, we conduct an experimental study of asynchronous invocation on the performance of client-server systems. Through extensive measurements of both realistic macro- and micro-benchmarks, we show that servers with the asynchronous event-driven architecture may perform significantly worse than the thread-based version resulting from two non-trivial reasons. First, the traditional wisdom of one-event-one-handler event processing flow can create large amounts of intermediate context switches that significantly degrade the performance of an asynchronous server. Second, some runtime workload (e.g., response size) and network conditions (e.g., network latency) may cause significant negative performance impact on the asynchronous event-driven servers, but not on thread-based ones. We provide a hybrid solution by taking advantage of different asynchronous architectures to adapt to varying workload and network conditions. Our hybrid solution searches for the most efficient execution path for each client request based on the runtime request profiling and type checking. Our experimental results show that the hybrid solution outperforms all the other types of servers up to 19%∼90% on throughput, depending on specific workload and network conditions.**

***Index Terms*—Asynchronous, event-driven, threads, client-server applications, performance**

I. Introduction

Asynchronous event-driven architecture for high performance internet servers has been studied extensively before [33] [37] [42] [32]. Many people advocate the asynchronous event-driven architecture as a better choice than the thread-based RPC version to handle high level workload concurrency because of reduced multi-threading overhead. Though conceptually simple, taking advantage of the asynchronous event-driven architecture to construct high performance internet servers is a non-trivial task. For example, asynchronous servers are well-known to be difficult to program and debug due to the obscured control flow compared to the thread-based version.

In this paper, we show that building high performance internet servers using the asynchronous event-driven architecture requires careful design of the event processing flow and the ability to adapt to the runtime varying workload and network conditions. Concretely, we conduct extensive benchmark experiments to study some non-trivial design deficiencies of the asynchronous event-driven server architectures that lead to the inferior performance compared to the thread-based version when facing high concurrency workload. For example, the traditional wisdom of one-event-one-handler event processing flow may generate a large amount of unnecessary intermediate

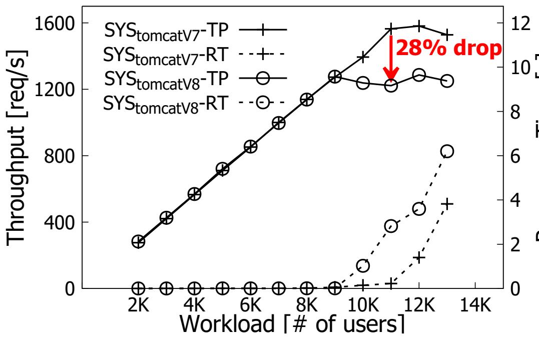


Fig. 1: Upgrading Tomcat from a thread-based version (V7) to an asynchronous version (V8) in a 3-tier system leads to significant performance degradation.

events and context switches that significantly degrade the performance of an asynchronous server. We also observed that some runtime workload and network conditions may cause frequent unnecessary I/O system calls (due to the non-blocking nature of asynchronous function calls) for the asynchronous event-driven servers, but not for the thread-based ones.

The first contribution is an experimental illustration that simply upgrading a thread-based component server to its asynchronous version in an n-tier web system can lead to significant performance degradation of the whole system at high utilization levels. For instance, we observed that the maximum achievable throughput of a 3-tier system decreases by 23% after we upgrade the Tomcat application server from the traditional thread-based version (Version 7) to the latest asynchronous version (Version 8) (see Figure 1) in a standard n-tier application benchmark (RUBBoS [12]). Our analysis reveals that the unexpected performance degradation of the asynchronous Tomcat results from its poor design of event processing flow, which causes significantly higher CPU context switch overhead than the thread-based version when the server is approaching saturation. Our further study shows such poor design of event processing flow also exists in other popular asynchronous servers/middleware such as Jetty [3], GlassFish [10], and MongoDB Java Asynchronous Driver [6].

The second contribution is a detailed analysis of various workload and network conditions that impact the performance of asynchronous invocation. Concretely, we have observed that a moderate-sized response message (e.g, 100KB) for an asynchronous server can cause a non-trivial `write-spin` problem, where the server makes frequent unnecessary I/O system calls resulting from the default small TCP send buffer size and the TCP wait-ACK mechanism, wasting the server CPU resource about 12%∼24%. We also observed some network conditions such as latency can exaggerate the write-spin problem of asynchronous event-driven servers even further.

The third contribution is a hybrid solution that takes advantage of different asynchronous event-driven architectures to adapt to various runtime workload and network conditions. We studied a popular asynchronous network I/O library named "Netty [7]" which can mitigate the write-spin problem through some write operation optimizations. However, such optimization techniques in Netty also bring non-trivial overhead in the case when the write-spin problem does not occur during the asynchronous invocation period. Our hybrid solution extends Netty by monitoring the occurrence of write-spin and oscillating between alternative asynchronous invocation mechanisms to avoid the unnecessary optimization overhead.

In general, given the strong economic interest in achieving high resource efficiency and high quality of service simultaneously in cloud data centers, our results suggest asynchronous invocation has potentially significant performance advantage over RPC synchronous invocation but still needs careful tunning according to various non-trivial runtime workload and network conditions. Our work also points to significant future research opportunities, since asynchronous architecture has been widely adopted by many distributed systems (e.g., pub/sub systems [23], AJAX [26], and ZooKeeper [31]).

The rest of the paper is organized as follows. Section II shows a case study of performance degradation after software upgrading from a thread-based version to an asynchronous version. Section III describes the context switch problem of servers with asynchronous architecture. Section IV explains the write-spin problem of asynchronous architecture when it handles requests with large size responses. Section V evaluates two practical solutions. Section VI summarizes related work and Section VII concludes the paper.

II. Background and Motivation

A. RPC vs. Asynchronous Network I/O

Modern internet servers usually adopt a few connectors to communicate with other component servers or end users. The main activities of a connector include managing upstream and downstream network connections, reading/writing data through the established connections, parsing and routing the incoming requests to the application (business logic) layer. Though similar in functionality, synchronous and asynchronous connectors have very different mechanisms to interact with the application layer logic.

Synchronous connectors are mainly adopted by RPC thread-based servers. Once accepting a new connection, the main thread will dispatch the connection to a dedicated worker thread until the close of the connection. In this case, each connection consumes one worker thread and the operating system transparently switches among worker threads for concurrent request processing. Although relatively easy to program due to the user-perceived sequential execution flow, synchronous connectors bring the well-known multi-threading overhead (e.g., context switches, scheduling and lock contention).

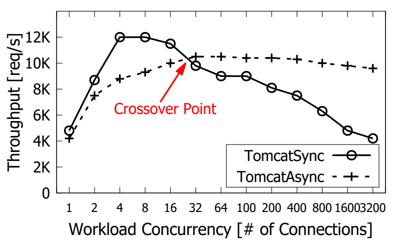
Asynchronous connectors accept new connections and manage all established connections through an event-driven mechanism using only one or a few threads. Given a pool of established connections in a server, an asynchronous connector handles requests received from these connections by repeatedly looping over two phases. The first phase (event monitoring phase) determines which connections have pending events of interest. These events typically indicate that a particular connection (i.e., socket) is readable or writable. The asynchronous connector pulls the connections with pending events by taking advantage of an event notification mechanism such as *select*, *poll*, or *epoll*, supported by the underlying operating system. The second phase (event handling phase) iterates over each of the connections that have pending events. Based on the context information of each event, the connector dispatches the event to an appropriate event handler performing the actual business logic computation. More details can be found in previous asynchronous server research [42] [32] [38] [37] [28].

In practice there are two general designs of asynchronous servers using the asynchronous connectors. The first one is a single-threaded asynchronous server which only uses one thread to loop over the aforementioned two phases, for example Node.js [9] and Lighttpd [5]. Such a design is especially beneficial for in-memory workloads because context switches will be minimum while the single thread will not be blocked by disk I/O activities [35]. Multiple single-threaded servers (also called N-copy approach [43] [28]) can be launched together to fully utilize multiple processors. The second design is to use a worker thread pool in the second phase to concurrently process connections that have pending events. Such a design is supposed to efficiently utilize CPU resources in case of transient disk I/O blocking or multi-core environment [38]. Several variants of the second design have been proposed before, mostly known as the staged design adopted by SEDA [42] and WatPipe [38]. Instead of having only one worker thread pool, the staged design decomposes the request processing into a pipeline of stages separated by event queues, each of which has its own worker thread pool, with the aim of modular design and fine-grained management of worker threads.

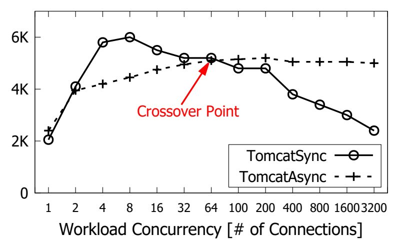
In general, asynchronous event-driven servers are believed to be able to achieve higher throughput than the thread-based version because of reduced multi-threading overhead, especially when the server is handling high concurrency CPU intensive workload. However, our experimental results in the following section will show the opposite.

B. Performance Degradation after Tomcat Upgrade

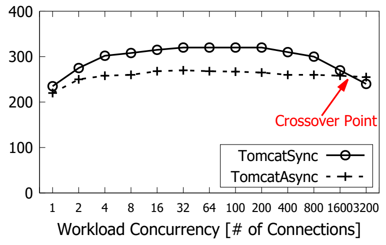
System software upgrade is a common practice for internet services due to the fast evolving of software components. In this section we show that simply upgrading a thread-based component server to its asynchronous version in an n-tier system may cause unexpected performance degradation at high resource utilization. We show one such case by RUBBoS [12], a representative web-facing n-tier system benchmark modeled



(a) The 0.1KB response size case.



(b) The 10KB response size case.



(c) The 100KB response size case.

Fig. 2: Throughput comparison between `TomcatSync` and `TomcatAsync` under different workload concurrencies and response sizes. As response size increases from 0.1KB in (a) to 100KB in (c), `TomcatSync` outperforms `TomcatAsync` at wider concurrency range, indicating the performance degradation of `TomcatAsync` with large response size.

after the popular news website *Slashdot* [16]. Our experiments adopt a typical 3-tier configuration, with 1 Apache web server, 1 Tomcat application server, and 1 MySQL database server (details in Appendix A). At the beginning we use Tomcat 7 (noted as `TomcatSync`), which uses a thread-based synchronous connector for inter-tier communication. We then upgrade the Tomcat server to Version 8 (the latest version at the time, noted as `TomcatAsync`), which by default uses an asynchronous connector, with the expectation of system performance improvement after Tomcat upgrade.

Unfortunately, we observed a surprising system performance degradation after the Tomcat server upgrade, as shown in Figure 1. We call the system with `TomcatSync` as $\texttt{SYS}_{tomcatV7}$ and the other one with `TomcatAsync` as $\texttt{SYS}_{tomcatV8}$. This figure shows the $\texttt{SYS}_{tomcatV7}$ saturates at workload 11000 while $\texttt{SYS}_{tomcatV8}$ saturates at workload 9000. At workload 11000, $\texttt{SYS}_{tomcatV7}$ outperforms $\texttt{SYS}_{tomcatV8}$ by 28% in throughput, and the average response time is one order of magnitude less (226ms vs. 2820ms). Such a result is counter intuitive since we upgrade Tomcat from a lower thread-based version to a newer asynchronous one. We note that in both cases the Tomcat server CPU is the bottleneck resource in the system; all the hardware resources of other component servers are far from saturation ($< 60\%$).

The other interesting phenomenon we observed using Collectl [2] is that `TomcatAsync` encounters significantly higher number of context switches than `TomcatSync` when the system is at the same workload. For example, at workload 10000 `TomcatAsync` encounters 12950 context switches per second while only 5930 for `TomcatSync`, less than half of the previous case. It is reasonable to suggest that high context switches in `TomcatAsync` cause high CPU overhead, leading to inferior throughput compared to `TomcatSync`. However, the traditional wisdom told us that a server with asynchronous architecture should have less context switches than a thread-based server. So why did we observe the opposite here? We will discuss about the cause in the next section.

III. Inefficient Event Processing Flow in Asynchronous Servers

In this section we explain why the performance of the 3-tier benchmark system degrades after we upgrade Tomcat from the thread-based version `TomcatSync` to the asynchronous version `TomcatAsync`. To simplify and quantify our analysis, we design micro-benchmarks to test the performance of both versions of Tomcat.

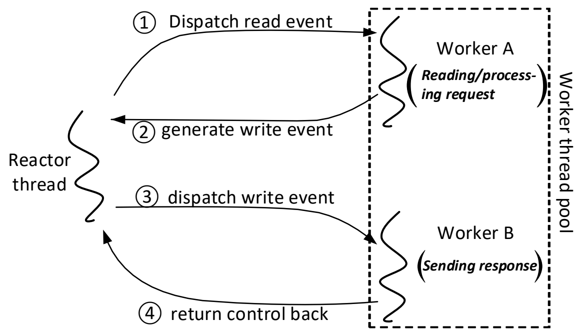


Fig. 3: Illustration of the event processing flow when `TomcatAsync` processes one request. Totally four context switches between the reactor thread and worker threads.

We use JMeter [1] to generate HTTP requests to access the standalone Tomcat directly. These HTTP requests are categorized into three types: small, medium, and large, with which the Tomcat server (either `TomcatSync` or `TomcatAsync`) first conducts some simple computation before responding with 0.1KB, 10KB, and 100KB of in-memory data, respectively. We choose these three sizes because they are representative response sizes in our RUBBoS benchmark application. JMeter uses one thread to simulate each end-user. We set the think time between the consecutive requests sent from the same thread to be zero, thus we can precisely control the concurrency of the workload to the target Tomcat server by specifying the number of threads in JMeter.

We compare the server throughput between `TomcatSync` and `TomcatAsync` under different workload concurrencies and response sizes as shown in Figure 2. The three subfigures show that as workload concurrency increases from 1 to 3200, `TomcatAsync` achieves lower throughput than `TomcatSync` before a certain workload concurrency. For example, `TomcatAsync` performs worse than `TomcatSync` before the workload concurrency 64 when the response size is 10KB; and the crossover point workload concurrency is even higher (1600) when the response size increases to 100KB.

TABLE I: `TomcatAsync` has more context switches than `TomcatSync` under workload concurrencies 8.

Response size	`TomcatAsync` [×1000/sec]	`TomcatSync` [×1000/sec]
0.1KB	40	16
10KB	25	7
100KB	28	2

Return back to our previous 3-tier RUBBoS experiments, our measurements show that under the RUBBoS workload conditions, the average response size of Tomcat per request is about 20KB, and the workload concurrency for Tomcat is about 35 when the system saturates. So based on our micro-benchmark results in Figure 2, it is not surprising that `TomcatAsync` performs worse than `TomcatSync`. Since Tomcat is the bottleneck server of the 3-tier system, the performance degradation of Tomcat also leads to the performance degradation of the whole system (see Figure 1). The remaining question is why `TomcatAsync` performs worse than `TomcatSync` before a certain workload concurrency.

As we found out that the performance degradation of `TomcatAsync` results from its inefficient event processing flow which generates significant amounts of intermediate context switches, causing non-trivial CPU overhead. Table I compares the context switches between `TomcatAsync` and `TomcatSync` at workload concurrency 8. This table shows consistent results as we have observed in the previous RUBBoS experiments: the asynchronous `TomcatAsync` encountered significantly higher context switches than the thread-based `TomcatSync`, given the same workload concurrency and server response size. Our further analysis reveals that the high context switches of `TomcatAsync` is because of its poor design of event processing flow. Concretely, `TomcatAsync` adopts the second design of asynchronous servers (see Section II-A), which uses a reactor thread for event monitoring and a worker thread pool for event handling. To process a new incoming request, Figure 3 shows the event processing flow in `TomcatAsync`, which includes the following four steps:

1) the reactor thread dispatches a read event to a worker thread;
2) the worker thread reads and parses the event, prepares the response, and then generates a write event; the reactor thread is notified the occurrence of the write event;
3) the reactor thread dispatches the write event to a worker thread to send the response out;
4) the worker thread finishes sending the response, and returns the control back to the reactor thread.

So to handle one client request, there are totally 4 context switches among the user-space threads in `TomcatAsync` (see step ①–④ in Figure 3). Such inefficient event processing flow design also exists in many popular asynchronous servers/middleware, including network framework Grizzly [11], application server Jetty [3]. On the other hand, in `TomcatSync` each client request is handled by a dedicated worker thread, from the initial reading of the request to preparing the response to sending the response out. No context switch during the processing of the request unless the worker thread is interrupted, or swapped out by operating system because CPU time slice expires.

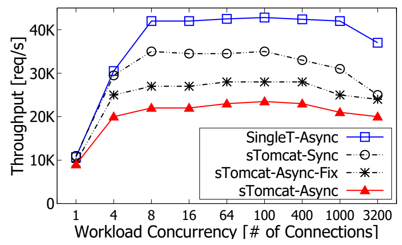
TABLE II: Context switches among user-space threads when the server processes one client request.

Server type	Context Switch	Note
`sTomcat-Async`	4	Read and write events are handled by different worker threads (Figure 3).
`sTomcat-Async-Fix`	2	Read and write events are handled by the same worker thread.
`sTomcat-Sync`	0	Dedicated worker thread for each request. Context switch occurs due to interrupt or CPU time slice expires.
`SingleT-Async`	0	No context switches, one thread handle both event monitoring and processing.

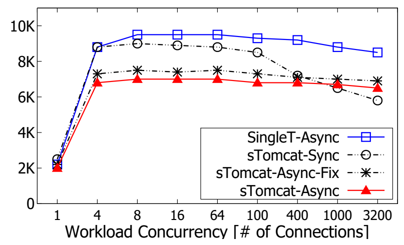
To better quantify the impact of context switches on the performance of different server architectures, we simplify the implementation of `TomcatAsync` and `TomcatSync` by removing out all the unrelated modules (e.g., servlet life cycle management, cache management, and logging) and only keeping the essential code related to request processing, which we refer as `sTomcat-Async` (simplified `TomcatAsync`) and `sTomcat-Sync` (simplified `TomcatSync`). As a reference, we implement two alternative designs of asynchronous servers aiming to reduce the frequency of context switches. The first alternative design, which we call `sTomcat-Async-Fix`, merges the processing of read event and write event from the same request by using the same worker thread. In this case, once a worker thread finishes preparing the response, it continues to send the response out (step ② and ③ in Figure 3 no longer exist); thus processing one client request only requires two context switches: from the reactor thread to a worker thread and from the same worker thread back to the reactor thread. The second alternative design is the traditional single-threaded asynchronous server. The single thread is responsible for both event monitoring and processing. The single-threaded implementation, which we refer as `SingleT-Async`, is supposed to have the least context switches. Table II summarizes the context switches for each server type when it processes one client request.[1] Interested readers can check out our server implementation from GitHub [14] for further reference.

We compare the throughput and context switches among the four types of servers under increased workload concurrencies and server response sizes as shown in Figure 4. Comparing Figure 4(a) and 4(d), the maximum achievable throughput by each server type is negatively correlated with the context switch frequency during runtime experiments. For example, at workload concurrency 16, `sTomcat-Async-Fix` outperforms `sTomcat-Async` by 22% in throughput while the context switches is 34% less. In our experiments, the CPU demand for each request is positively correlated to the response size; small response size means small CPU computation demand, thus the portion of CPU cycles wasted in context switches becomes large. As a result, the gap in context switches between `sTomcat-Async-Fix` and `sTomcat-Async` re-

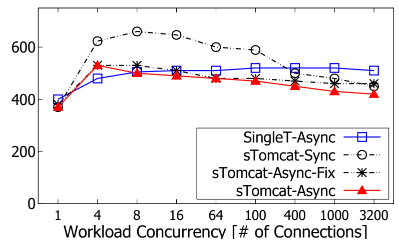
[1] In order to simplify analyzing and reasoning, we do not count the context switches causing by interrupting or swapping by the operating system.



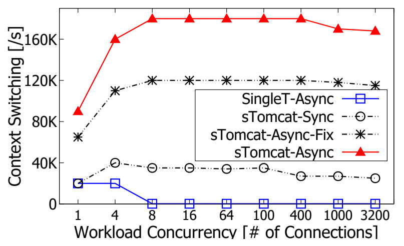
(a) Throughput when response size is 0.1KB



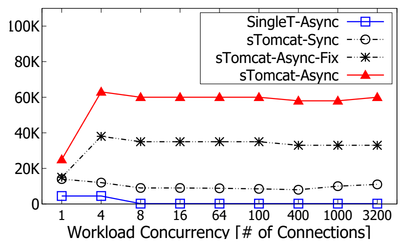
(b) Throughput when response size is 10KB



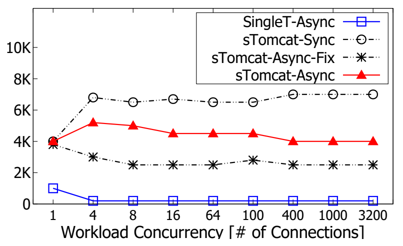
(c) Throughput when response size is 100KB



(d) Context switch when response size is 0.1KB



(e) Context switch when response size is 10KB



(f) Context switch when response size is 100KB

Fig. 4: Throughput and context switch comparison among different server architectures as the server response size increases from 0.1KB to 100KB. (a) and (d) show that the maximum achievable throughput by each server type is negatively correlated with their context switch freqency when the server response size is small (0.1KB). However, as the response size increases to 100KB, (c) shows `sTomcat-Sync` outperforms other asynchronous servers before the workload concurrency 400, indicating factors other than context switches cause overhead in asynchronous servers.

flects their throughput difference. Such hypothesis is further validated by the performance of `SingleT-Async` and `sTomcat-Sync`, which outperform `sTomcat-Async` by 91% and 57% in throughput, respectively (see Figure 4(a)). Such performance difference is also because of less context switches as shown in Figure 4(d). For example, the context switches of `SingleT-Async` is a few hundred per second, three orders of magnitude less than that of `sTomcat-Async`.

We note that as the server response size becomes larger, the portion of CPU overhead caused by context switches becomes smaller since more CPU cycles will be consumed by processing request and sending response. This is the case as shown in Figure 4(b) and 4(c), where the response sizes are 10KB and 100KB, respectively. The throughput difference becomes narrower among the four server architectures, indicating less performance impact from context switches.

In fact, one interesting phenomenon has been observed as the response size increases to 100KB. Figure 4(c) shows that `SingleT-Async` performs worse than the thread-based `sTomcat-Sync` before the workload concurrency 400, even though `SingleT-Async` has much less context switches than `sTomcat-Sync` as shown in Figure 4(f). Such observation suggests that there are other factors causing overhead in asynchronous `SingleT-Async` but not in thread-based `sTomcat-Sync` when the server response size is large, which we will discuss in the next section.

IV. Write-Spin Problem of Asynchronous Invocation

In this section, we study the performance degradation problem of an asynchronous server sending a large size response. We use fine-grained profiling tools such as Collectl [2] and JProfiler [4] to analyze the detailed CPU usage and some key system calls invoked by servers with different architectures. As we found out that it is the default small TCP send buffer size and the TCP wait-ACK mechanism that leads to a severe write-spin problem when sending a relatively large size response, which causes significant CPU overhead for asynchronous servers. We also explored several network-related factors that could exaggerate the negative impact of the write-spin problem, which further degrades the performance of an asynchronous server.

A. *Profiling Results*

Recall that Figure 4(a) shows when the response size is small (i.e. 0.1KB), the throughput of the asynchronous `SingleT-Async` is 20% higher than the thread-based `sTomcat-Sync` at workload concurrency 8. However, as response size increases to 100KB, `SingleT-Async` throughput is surprisingly 31% lower than `sTomcat-Sync` under the same workload concurrency 8 (see Figure 4(c)). Since the only change is the response size, it is natural to speculate that large response size brings significant overhead for `SingleT-Async`, but not for `sTomcat-Sync`.

To investigate the performance degradation of `SingleT-Async` when the response size is large, we first use Collectl [2] to analyze the detailed CPU usage of the server with different server response sizes as shown in Table III. The workload concurrency for both `SingleT-Async` and `sTomcat-Sync` is 100 and the CPU is 100% utilized under this workload concurrency. As the response size for both server architectures increased

TABLE III: `SingleT-Async` consumes more user-space CPU compared to `sTomcat-Sync`. The workload concurrency keeps 100.

Server Type	`sTomcat-Sync`		`SingleT-Async`	
Response Size	0.1KB	100KB	0.1KB	100KB
Throughput [req./sec]	35000	590	42800	520
User total %	55%	80%	58%	92%
System total %	45%	20%	42%	8%

TABLE IV: The write-spin problem occurs when the response size is 100KB. This table shows the measurement of total number of `socket.write()` in `SingleT-Async` with different response size during a one-minute experiment.

Resp. size	# req.	# `socket.write()`	# `write()` per req.
0.1KB	238530	238530	1
10KB	9400	9400	1
100KB	2971	303795	**102**

from 0.1KB to 100KB, the table shows the user-space CPU utilization of `sTomcat-Sync` increases 25% (from 55% to 80%) while 34% (from 58% to 92%) for `SingleT-Async`. Such comparison suggests that increasing response size has more impact on the asynchronous `SingleT-Async` than the thread-based `sTomcat-Sync` in user-space CPU utilization.

We further use JProfiler [4] to profile the `SingleT-Async` case when the response size increases from 0.1KB to 100KB and see what has changed in application level. We found that the frequency of `socket.write()` system call is especially high in the 100KB case as shown in Table IV. We note that `socket.write()` is called when a server sends a response back to the corresponding client. In the case of a thread-based server like `sTomcat-Sync`, `socket.write()` is called only once for each client request. While such one write per request is true for the 0.1KB and 10KB case in `SingleT-Async`, it calls `socket.write()` averagely 102 times per request in the 100KB case. System calls in general are expensive due to the related kernel crossing overhead [20] [39], thus high frequency of `socket.write()` in the 100KB case helps explain high user-space CPU overhead in `SingleT-Async` as shown in Table III.

Our further analysis shows that the multiple socket write problem of `SingleT-Async` is due to the small TCP send buffer size (16K by default) for each TCP connection and the TCP wait-ACK mechanism. When a processing thread tries to copy 100KB data from the user space to the kernel space TCP buffer through the system call `socket.write()`, the first `socket.write()` can only copy at most 16KB data to the send buffer, which is organized as a byte buffer ring. A TCP sliding window is set by the kernel to decide how much data can actually be sent to the client; the sliding window can move forward and free up buffer space for new data to be copied in only if the server receives the ACKs of the previously sent-out packets. Since `socket.write()` is a non-blocking system call in `SingleT-Async`, every time it returns how many bytes are written to the TCP send buffer; the system

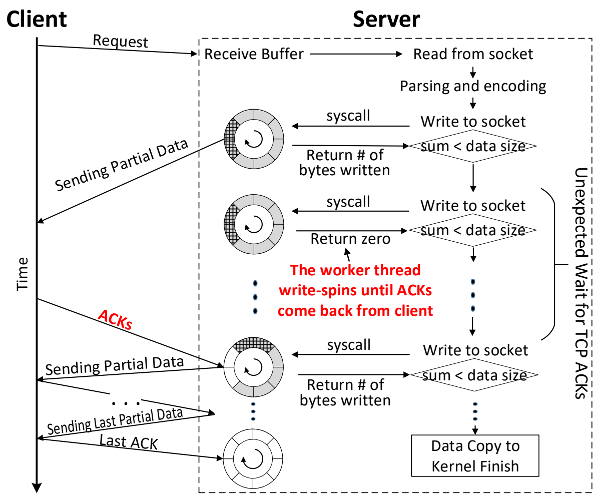


Fig. 5: Illustration of the write-spin problem in an asynchronous server. Due to the small TCP send buffer size and the TCP wait-ACK mechanism, a worker thread write-spins on the system call `socket.write()` and can only send more data until ACKs back from the client for previous sent packets.

call will return zero if the TCP send buffer is full, leading to the write-spin problem. The whole process is illustrated in Figure 5. On the other hand, when a worker thread in the synchronous `sTomcat-Sync` tries to copy 100KB data from the user space to the kernel space TCP send buffer, only one blocking system call `socket.write()` is invoked for each request; the worker thread will wait until the kernel sends the 100KB response out and the write-spin problem is avoided.

An intuitive solution is to increase the TCP send buffer size to the same size as the server response to avoid the write-spin problem. Our experimental results actually show the effectiveness of manually increasing the TCP send buffer size to solve the write-spin problem for our RUBBoS workload. However, several factors make setting a proper TCP send buffer size a non-trivial challenge in practice. First, the response size of an internet server can be dynamic and is difficult to predict in advance. For example, the response of a Tomcat server may involve dynamic content retrieved from the downstream database, the size of which can range from hundreds of bytes to megabytes. Second, HTTP/2.0 enables a web server to push multiple responses for a single client request, which makes the response size for a client request even more unpredictable [19]. For example, the response of a typical news website (e.g., CNN.com) can easily reach tens of megabytes resulting from a large amount of static and dynamic content (e.g., images and database query results); all these content can be pushed back by answering one client request. Third, setting a large TCP send buffer for each TCP connection to prepare for the peak response size consumes a large amount of memory of the server, which may serve hundreds or thousands of end users (each has one or a few persistent TCP connections); such over-provisioning strategy is expensive and wastes computing resources in a shared cloud computing platform. Thus it is challenging to set a proper TCP send buffer size in advance

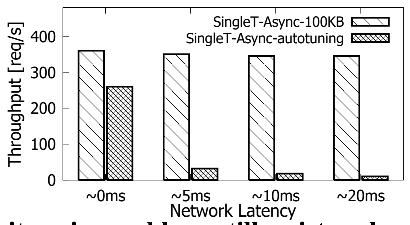


Fig. 6: Write-spin problem still exists when TCP send buffer "autotuning" feature enabled.

and prevent the write-spin problem.

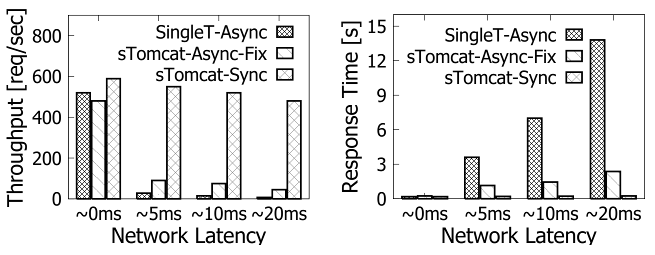
In fact, Linux kernels above 2.4 already provide an auto-tuning function for TCP send buffer size based on the runtime network conditions. Once turned on, the kernel dynamically resizes a server's TCP send buffer size to provide optimized bandwidth utilization [25]. However, the auto-tuning function aims to efficiently utilize the available bandwidth of the link between the sender and the receiver based on Bandwidth-Delay Product rule [17]; it lacks sufficient application information such as response size. Therefore, the auto-tuned send buffer could be enough to maximize the throughput over the link but still inadequate for applications, which may still cause the write-spin problem for asynchronous servers. Figure 6 shows `SingleT-Async` with auto-tunning performs worse than the other case with a fixed large TCP send buffer size (100kB), suggesting the occurrence of the write-spin problem. Our further study also shows the performance difference is even bigger if there is non-trivial network latency between the client and the server, which is the topic of the next subsection.

B. Network Latency Exaggerates the Write-Spin Problem

Network latency is common in cloud data centers. Considering the component servers in an n-tier application that may run on VMs located in different physical nodes across different racks or even data centers, which can range from a few milliseconds to tens of milliseconds. Our experimental results show that the negative impact of the write-spin problem can be significantly exaggerated by the network latency.

The impact of network latency on the performance of different types of servers is shown in Figure 7. In this set of experiments we keep the workload concurrency from clients to be 100 all the time. The response size of each client request is 100KB; the TCP send buffer size of each server is the default 16KB, with which an asynchronous server encounters the write-spin problem. We use the Linux command "`tc`(Traffic Control)" in the client side to control the network latency between the client and the server. Figure 7(a) shows that the throughput of the asynchronous servers `SingleT-Async` and `sTomcat-Async-Fix` is sensitive to network latency. For example, when the network latency is 5ms, the throughput of `SingleT-Async` decreases by about 95%, which is surprising considering the small amount of latency increased.

We found that the surprising throughput degradation results from the response time amplification when the write-spin problem happens. This is because sending a relatively large size response requires multiple rounds of data transfer due to



(a) Throuthput comparison (b) Response time comparison

Fig. 7: Throughput degradation of two asynchronous servers in subfigure (a) resulting from the response time amplification in (b) as the network latency increases.

the small TCP send buffer size, each data transfer has to wait until the server receives the ACKs from the previously sent-out packets (see Figure 5). Thus a small network latency increase can amplify a long delay for completing one response transfer. Such response time amplification for asynchronous servers can be seen in Figure 7(b). For example, the average response time of `SingleT-Async` for a client request increases from 0.18 seconds to 3.60 seconds when 5 milliseconds network latency is added. According to Little's Law, a server's throughput is negatively correlated with the response time of the server given that the workload concurrency (queued requests) keeps the same. Since we always keep the workload concurrency for each server to be 100, server response time increases 20 times (from 0.18 to 3.60) means 95% decrease in server throughput in `SingleT-Async` as shown in Figure 7(a).

V. Solution

So far we have discussed two problems of asynchronous invocation: the context switch problem caused by inefficient event processing flow (see Table II) and the write-spin problem resulting from the unpredictable response size and the TCP wait-ACK mechanism (see Figure 5). Though our research is motivated by the performance degradation of the latest asynchronous Tomcat, we found that the inappropriate event processing flow and the write-spin problems widely exist in other popular open-source asynchronous application servers/middleware, including network framework Grizzly [11] and application server Jetty [3].

An ideal asynchronous server architecture should avoid both problems under various workload and network conditions. We first investigate a popular asynchronous network I/O library named Netty [7] which is supposed to mitigate the context switch overhead through an event processing flow optimization and the write-spin problem of asynchronous messaging through write operation optimization, but with non-trivial optimization overhead. Then we propose a hybrid solution which takes advantage of different types of asynchronous servers aiming to solve both the context switch overhead and the write-spin problem while avoid the optimization overhead.

A. Mitigating Context Switches and Write-Spin Using Netty

Netty is an asynchronous event-driven network I/O framework which provides optimized read and write operations in

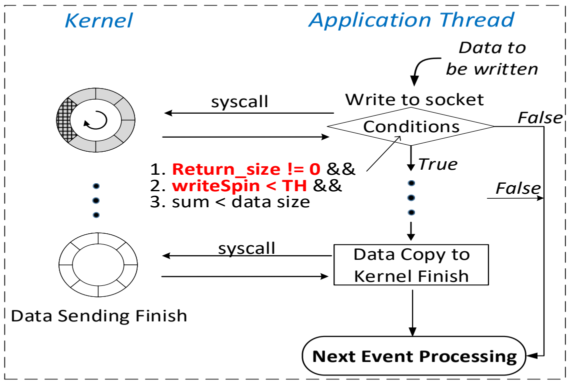
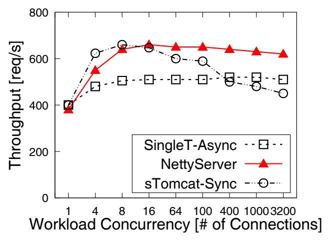
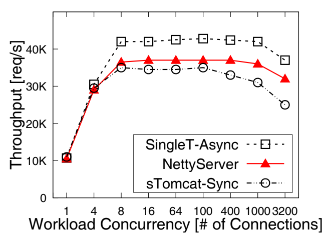


Fig. 8: Netty mitigates the write-spin problem by runtime checking. The write spin jumps out of the loop if any of the three conditions is not met.



(a) Response size is 100KB.



(b) Response size is 0.1KB.

Fig. 9: Throughput comparison under various workload concurrencies and response sizes. The default TCP send buffer size is 16KB. Subfigure (a) shows that `NettyServer` performs the best, suggesting effective mitigation of the write-spin problem, and (b) shows that `NettyServer` performs worse than `SingleT-Async` indicating non-trivial write optimization overhead in Netty.

order to mitigate the context switch overhead and the write-spin problem. Netty adopts the second design strategy (see Section II-A) to support an asynchronous server: using a reactor thread to accept new connections and a worker thread pool to process the various I/O events from each connection.

Though using a worker thread pool, Netty makes two significant changes compared to the asynchronous `TomcatAsync` to reduce the context switch overhead. First, Netty changes the role of the reactor thread and the worker threads. In the asynchronous `TomcatAsync` case, the reactor thread is responsible to monitor events for each connection (event monitoring phase), then it dispatches each event to an available worker thread for proper event handling (event handling phase). Such dispatching operation always involves the context switches between the reactor thread and a worker thread. Netty optimizes this dispatching process by letting a worker thread take care of both event monitoring and handling; the reactor thread only accepts new connections and assigns the established connections to each worker thread. In this case, the context switches between the reactor thread and the worker threads are significantly reduced. Second, instead of having a single event handler attached to each event, Netty allows a chain of handlers to be attached to one event; the output of each handler is the input to the next handler (pipeline). Such a design avoids generating unnecessary intermediate events and the associate system calls, thus reducing the unnecessary context switches between reactor thread and worker threads.

In order to mitigate the write-spin problem, Netty adopts a write-spin checking when a worker thread calls `socket.write()` to copy a large size response to the kernel as shown in Figure 8. Concretely, each worker thread in Netty maintains a `writeSpin` counter to record how many times it has tried to write a single response into the TCP send buffer. For each write, the worker thread also tracks how many bytes have been copied, noted as `return_size`. The worker thread will jump out the write spin if either of two conditions is met: first, the `return_size` is zero, indicating the TCP send buffer is already full; second, the counter `writeSpin` exceeds a pre-defined threshold (the default value is 16 in Netty-v4). Once jumping out, the worker thread will save the context and resume the current connection data transfer after it loops over other connections with pending events. Such write optimization is able to mitigate the blocking of the worker thread by a connection transferring a large size response, however, it also brings non-trivial overhead when all responses are small and there is no write-spin problem.

We validate the effectiveness of Netty for mitigating the write-spin problem and also the associate optimization overhead in Figure 9. We build a simple application server based on Netty, named `NettyServer`. This figure compares `NettyServer` with the asynchronous `SingleT-Async` and the thread-based `sTomcat-Sync` under various workload concurrencies and response sizes. The default TCP send buffer size is 16KB, so there is no write-spin problem when the response size is 0.1KB, and severe write-spin problem in the 100KB case. Figure 9(a) shows that `NettyServer` performs the best among three in the 100KB case; for example, when the workload concurrency is 100, `NettyServer` outperforms `SingleT-Async` and `sTomcat-Sync` by about 27% and 10% in throughput respectively, suggesting `NettyServer`'s write optimization effectively mitigates the write-spin problem encountered by `SingleT-Async` and also avoids the heavy multi-threading overhead encountered by `sTomcat-Sync`. On the other hand, Figure 9(b) shows that the maximum achievable throughput of `NettyServer` is 17% less than that of `SingleT-Async` in the 0.1KB response case, indicating non-trivial overhead of unnecessary write operation optimization when there is no write-spin problem. Therefore, neither `NettyServer` nor `SingleT-Async` is able to achieve the best performance under various workload conditions.

B. A Hybrid Solution

In the previous section, we showed that the asynchronous solutions, if chosen properly (see Figure 9), can always outperform the corresponding thread-based version under various workload conditions. However, there is no single asynchronous solution that can always perform the best. For example, `SingleT-Async` suffers from the write-spin problem for

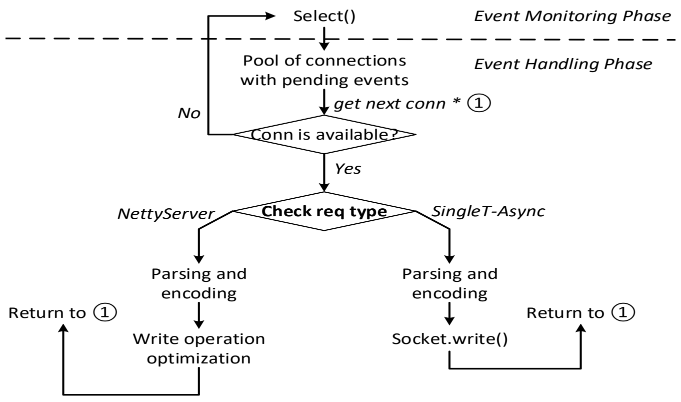


Fig. 10: Worker thread processing flow in Hybrid solution.

large size responses while NettyServer suffers from the unnecessary write operation optimization overhead for small size responses. In this section, we propose a hybrid solution which utilizes both SingleT-Async and NettyServer and adapts to workload and network conditions.

Our hybrid solution is based on two assumptions:

- The response size of the server is unpredictable and can vary during runtime.
- The workload is in-memory workload.

The first assumption excludes the server being initiated with a large but fixed TCP send buffer size for each connection in order to avoid the write-spin problem. This assumption is reasonable because of the factors (e.g., dynamically generated response and the push feature in HTTP/2.0) we have discussed in Section IV-A. The second assumption excludes a worker thread being blocked by disk I/O activities. This assumption is also reasonable since in-memory workload becomes common for modern internet services because of near-zero latency requirement [30], for example, MemCached server has been widely adopted to reduce disk activities [36]. The solution for more complex workloads that involve frequent disk I/O activities is challenging and will require additional research.

The main idea of the hybrid solution is to take advantage of different asynchronous server architectures such as SingleT-Async and NettyServer to handle requests with different response sizes and network conditions as shown in Figure 10. Concretely, our hybrid solution, which we call HybridNetty, profiles different types of requests based on whether or not the response causes a write-spin problem during the runtime. In initial warm-up phase (i.e., workload is low), HybridNetty uses the writeSpin counter of the original Netty to categorize all requests into two categories: the heavy requests that can cause the write-spin problem and the light requests that can not. HybridNetty maintains a map object recording which category a request belongs to. Thus when HybridNetty receives a new incoming request, it checks the map object first and figures out which category it belongs to and then chooses the most efficient execution path. In practice the response size even for the same type of requests may change over time (due to runtime environment changes such as dataset), so we update the map object during runtime once a request is detected to be classified into a wrong category in order to keep track of the latest category of such requests.

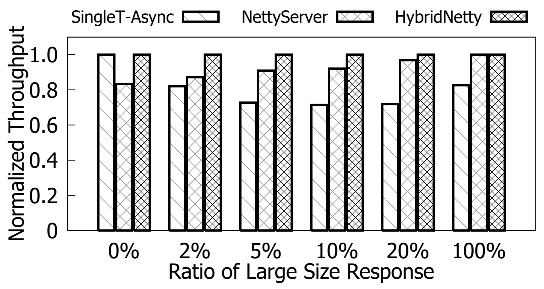
C. Validation of HybridNetty

To validate the effectiveness of our hybrid solution, Figure 11 compares HybridNetty with SingleT-Async and NettyServer under various workload conditions and network latencies. Our workload consists of two classes of requests: the heavy requests which have large response sizes (e.g., 100KB) and the light requests which have small response size (e.g., 0.1KB); heavy requests can cause the write-spin problem while light requests can not. We increase the percentage of heavy requests from 0% to 100% in order to simulating different scenarios of realistic workloads. The workload concurrency from clients in all cases keeps 100, under which the server CPU is 100% utilized. To clearly show the effectiveness of our hybrid solution, we adopt the normalized throughput comparison and use the HybridNetty throughput as the baseline. Figure 11(a) and 11(b) show that HybridNetty behaves the same as SingleT-Async when all requests are light (0% heavy requests) and the same as NettyServer when all requests are heavy; other than that HybridNetty always performs the best. For example, Figure 11(a) shows that when the heavy requests reach to 5%, HybridNetty achieves 30% higher throughput than SingleT-Async and 10% higher throughput than NettyServer. This is because HybridNetty always chooses the most efficient path to process request. Considering that the distribution of requests for real web applications typically follows a Zipf-like distribution, where light requests dominate the workload [22], our hybrid solution makes more sense in dealing with realistic workload. In addition, SingleT-Async performs much worse than the other two cases when the percentage of heavy requests is nonzero and non-negligible network latency exists (Figure 11(b)). This is because of the write-spin problem exaggerated by network latency (see Section IV-B for more details).

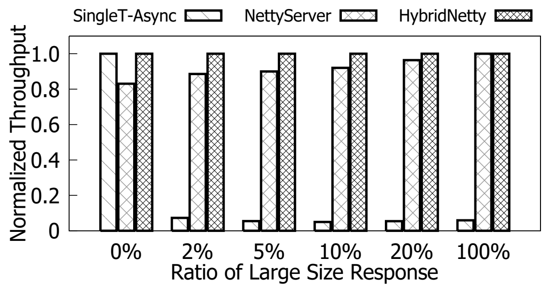
VI. Related Work

Previous research has shown that a thread-based server, if implemented properly, can achieve the same or even better performance as the asynchronous event-driven one does. For example, Von et al. develop a thread-based web server Knot [40] which can compete with event-driven servers at high concurrency workload using a scalable, user-level threading package Capriccio [41]. However, Krohn et al. [32] show that Capriccio is a cooperative threading package that exports the POSIX thread interface but behaves like events to the underlying operating system. The authors of Capriccio also admit that the thread interface is still less flexible than events [40]. These previous research results suggest that the asynchronous event-driven architecture will continue to play an important role in building high performance and resource efficiency servers that meet the requirements of current cloud data centers.

The optimization for asynchronous event-driven servers can be divided into two broad categories: improving operating system support and tuning software configurations.



(a) No network latency between client and server



(b) ~5ms network latency between client and server

Fig. 11: Hybrid solution performs the best in different mixes of light/heavy request workload with or without network latency. The workload concurrency keeps 100 in all cases. To clearly show the throughput difference, we compare the normalized throughput and use `HybridNetty` as the baseline.

Improving operating system support mainly focuses on either refining underlying event notification mechanisms [18] [34] or simplifying the interfaces of network I/O for application level asynchronous programming [27]. These research efforts have been motivated by reducing the overhead incurred by system calls such as *select*, *poll*, *epoll*, or I/O operations under high concurrency workload. For example, to avoid the kernel crossings overhead caused by system calls, TUX [34] implements a kernel-based web server by integrating the event monitoring and event handling into the kernel. Han et al. [27] implement MegaPipe as a new interface (API) for efficient, scalable network I/O by providing lightweight sockets to application level programming.

Tuning software configurations to improve asynchronous web servers' performance has also been studied before. For example, Pariag et al. [38] show that the maximum achievable throughput of event-driven (μServer) and pipeline (WatPipe) servers can be significantly improved by carefully tuning the number of simultaneous TCP connections and blocking/non-blocking `sendfile` system call. Brecht et al. [21] improve the performance of event-driven μServer by modifying the strategy of accepting new connections based on different workload characteristics. Our work is closely related to Google team's research about TCP's congestion window [24]. They show that increasing TCP's initial congestion window to at least ten segments (about 15KB) can improve average latency of HTTP responses by approximately 10% in large-scale Internet experiments. However, their work mainly focuses on short-lived TCP connections. Our work complements their research but focuses on more general network conditions.

VII. Conclusions

We studied the performance impact of asynchronous invocation on client-server systems. Through realistic macro- and micro-benchmarks, we showed that servers with the asynchronous event-driven architecture may perform significantly worse than the thread-based version resulting from the inferior event processing flow which creates high context switch overhead (Section II and III). We also studied a general problem for all the asynchronous event-driven servers: the write-spin problem when handling large size responses and the associate exaggeration factors such as network latency (Section IV). Since there is no one solution fits all, we provide a hybrid solution by utilizing different asynchronous architectures to adapt to various workload and network conditions (Section V). More generally, our research suggests that building high performance asynchronous event-driven servers needs to take both the event processing flow and the runtime varying workload/network conditions into consideration.

(a). Software and Hardware Setup

Software Stack		Hardware Stack	
Web Server	Apache 2.2.22	*Processor*	Intel Xeon CPU E5-2640, 2.50GHz, Hexa-core
Application Server	Tomcat 7.0.57 and Tomcat 8.0.36	*Memory*	16GB
Database Server	MySQL 5.5.19	*Disk*	SATA 500GB 7200RPM
Operating System	RHEL6 (kernel 2.6.32)	*Network*	1Gbps

(b). 1/1/1 Sample Topology

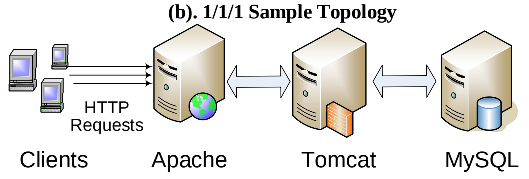


Fig. 12: Details of the RUBBoS experimental setup.

Appendix A
RUBBoS Experimental Setup

We adopt the RUBBoS standard n-tier benchmark, which is modeled after the famous news website *Slashdot*. The workload consists of 24 different web interactions. The default workload generator emulates a number of users interacting with the web application layer. Each user's behavior follows a Markov chain model to navigate between different web pages; the think time between receiving a web page and submitting a new page download request is about 7-second. Such workload generator has a similar design as other standard n-tier benchmarks such as RUBiS [13], TPC-W [15], and Cloudstone [29]. We run the RUBBoS benchmark on our testbed. Figure 12 outlines the software configurations, hardware configurations, and a sample 3-tier topology used in the Subsection II-B experiments. Each server in the 3-tier topology is deployed in a dedicated machine. All other client-server experiments are conducted with one client and one server machine.

References

[1] *Apache JMeter^{TM}*. http://jmeter.apache.org.

[2] *Collectl*. http://collectl.sourceforge.net/.

[3] *Jetty: A Java HTTP (Web) Server and Java Servlet Container*. http://www.eclipse.org/jetty/.

[4] *JProfiler: The award-winning all-in-one Java profiler*. "https://www.ej-technologies.com/products/jprofiler/overview.html".

[5] *lighttpd*. https://www.lighttpd.net/.

[6] *MongoDB Async Java Driver*. http://mongodb.github.io/mongo-java-driver/3.5/driver-async/.

[7] *Netty*. http://netty.io/.

[8] *nginx: a high performance HTTP and reverse proxy server, as well as a mail proxy server*. https://nginx.org/en/.

[9] *Node.js*. https://nodejs.org/en/.

[10] *Oracle GlassFish Server*. http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html.

[11] *Project Grizzly: NIO Event Development Simplified*. https://javaee.github.io/grizzly/.

[12] *RUBBoS: Bulletin board benchmark*. http://jmob.ow2.org/rubbos.html.

[13] *RUBiS: Rice University Bidding System*. http://rubis.ow2.org/.

[14] *`sTomcat-NIO`, `sTomcat-BIO`, and two alternative asynchronous servers*. https://github.com/sgzhang/AsynMessaging.

[15] *TPC-W: A Transactional Web e-Commerce Benchmark*. http://www.tpc.org/tpcw/.

[16] ADLER, S. The slashdot effect: an analysis of three internet publications. *Linux Gazette 38* (1999), 2.

[17] ALLMAN, M., PAXSON, V., AND BLANTON, E. Tcp congestion control. Tech. rep., 2009.

[18] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 45–58.

[19] BELSHE, M., THOMSON, M., AND PEON, R. Hypertext transfer protocol version 2 (http/2).

[20] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 43–57.

[21] BRECHT, T., PARIAG, D., AND GAMMO, L. Acceptable strategies for improving web server performance. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 20–20.

[22] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1999), vol. 1, IEEE, pp. 126–134.

[23] CAÑAS, C., ZHANG, K., KEMME, B., KIENZLE, J., AND JACOBSEN, H.-A. Publish/subscribe network designs for multiplayer games. In *Proceedings of the 15th International Middleware Conference* (New York, NY, USA, 2014), Middleware '14, ACM, pp. 241–252.

[24] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing tcp's initial congestion window. *SIGCOMM Comput. Commun. Rev. 40*, 3 (June 2010), 26–33.

[25] FISK, M., AND FENG, W.-C. Dynamic right-sizing in tcp. *http://lib-www. lanl. gov/la-pubs/00796247. pdf* (2001), 2.

[26] GARRETT, J. J., ET AL. Ajax: A new approach to web applications.

[27] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 135–148.

[28] HARJI, A. S., BUHR, P. A., AND BRECHT, T. Comparing high-performance multi-core web-server architectures. In *Proceedings of the 5th Annual International Systems and Storage Conference* (New York, NY, USA, 2012), SYSTOR '12, ACM, pp. 1:1–1:12.

[29] HASSAN, O. A.-H., AND SHARGABI, B. A. A scalable and efficient web 2.0 reader platform for mashups. *Int. J. Web Eng. Technol. 7*, 4 (Dec. 2012), 358–380.

[30] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 167–181.

[31] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[32] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 7:1–7:14.

[33] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Simplified event programming for busy network applications. In *Proceedings of the 2007 USENIX Annual Technical Conference (Santa Clara, CA, USA* (2007).

[34] LEVER, C., ERIKSEN, M. A., AND MOLLOY, S. P. An analysis of the tux web server. Tech. rep., Center for Information Technology Integration, 2000.

[35] LI, C., SHEN, K., AND PAPATHANASIOU, A. E. Competitive prefetching for concurrent sequential i/o. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 189–202.

[36] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[37] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An efficient and portable web server. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), ATEC '99, USENIX Association, pp. 15–15.

[38] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of web server architectures. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 231–243.

[39] SOARES, L., AND STUMM, M. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 33–46.

[40] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 4–4.

[41] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: Scalable threads for internet services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 268–281.

[42] WELSH, M., CULLER, D., AND BREWER, E. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2001), SOSP '01, ACM, pp. 230–243.

[43] ZELDOVICH, N., YIP, A., DABEK, F., MORRIS, R., MAZIERES, D., AND KAASHOEK, M. F. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference, General Track* (2003), pp. 239–252.