The Impact of Event Processing Flow on Asynchronous Server Efficiency

Shungeng Zhang, *Student Member, IEEE*, Qingyang Wang, *Member, IEEE*, Yasuhiko Kanemasa, *Member, IEEE*, Huasong Shan, *Student Member, IEEE*, and Liting Hu, *Member, IEEE*

Abstract—Asynchronous event-driven server architecture has been considered as a superior alternative to the thread-based counterpart due to reduced multithreading overhead. In this paper, we conduct empirical research on the efficiency of asynchronous Internet servers, showing that an asynchronous server may perform significantly worse than a thread-based one due to two design deficiencies. The first one is the widely adopted one-event-one-handler event processing model in current asynchronous Internet servers, which could generate frequent unnecessary context switches between event handlers, leading to significant CPU overhead of the server. The second one is a write-spin problem (i.e., repeatedly making unnecessary I/O system calls) in asynchronous servers due to some specific runtime workload and network conditions (e.g., large response size and non-trivial network latency). To address these two design deficiencies, we present a hybrid solution by exploiting the merits of different asynchronous architectures so that the server is able to adapt to dynamic runtime workload and network conditions in the cloud. Concretely, our hybrid solution applies a lightweight runtime request checking and seeks for the most efficient path to process each request from clients. Our results show that the hybrid solution can achieve from 10 to 90 percent higher throughput than all the other types of servers under the various realistic workload and network conditions in the cloud.

Index Terms—Asynchronous, event-driven, thread-based, internet servers, efficiency

1 Introduction

Modern Internet servers are expected to handle high concurrency workload at high resource efficiency in the cloud [1],[2]. To achieve this goal, many previous research efforts [3], [4] have shown that the asynchronous event-driven architecture could be a superior alternative to the traditional thread-based design. An important reason is that an asynchronous event-driven server can avoid the well-known multithreading overhead, which usually occurs in the thread-based counterpart when facing high concurrency workload. Though conceptually simple, building high-performance asynchronous event-driven servers is challenging because of the obscured non-sequential control flow rooted in the event-driven programming model [4].

In this paper, we study some non-trivial design deficiencies of asynchronous event-driven servers that make them less efficient than the thread-based counterparts when facing high concurrency workload. Through our extensive experiments, we show that constructing good performance and high efficiency asynchronous event-driven servers requires careful design of event processing flow and the capability to adapt to dynamic runtime workload and network conditions. For example, the conventional design practice of one-event-one-handler event processing flow may cause a significant performance loss of an asynchronous server by generating frequent unnecessary intermediate events and context switches, which occur at the transition of control flow between different event handlers. Our further analysis also shows that some runtime workload and network conditions might result in frequent redundant I/O system calls due to the non-blocking nature of asynchronous function calls, causing significant CPU overhead in an asynchronous server, but not in a thread-based one.

The first contribution of the paper is an empirical study illustrating the negative impact of the inefficient event processing flow on asynchronous server performance. Our study is motivated by running a standard 3-tier application benchmark RUBBoS [5] (see Fig. 3), where we observed a significant system throughput drop (28 percent) after we merely upgrade the Tomcat application server in the system from a thread-based version (Version 7) to its asynchronous event-driven version (Version 8). Our analysis reveals that such an unexpected performance degradation stems from the poor design of event processing flow of the asynchronous Tomcat server, causing a significant high CPU overhead due to unnecessary context switches.We further investigate many other representative asynchronous servers/middleware (see Table 1) and

- *S. Zhang and Q. Wang are with the Division of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: {shungeng, qywang}@csc.lsu.edu.*
- *Y. Kanemasa is with Software Laboratory, FUJITSU LABORATORIES LTD, Kawasaki 211-8588, Japan. E-mail: kanemasa@jp.fujitsu.com.*
- *H. Shan is with JD.com American Technologies Corporation, Mountain View, CA 94043. E-mail: huasong.shan@jd.com.*
- *L. Hu is with Computing and Information Sciences, Florida International University, Miami, FL 33199. E-mail: lhu@cs.fiu.edu.*

Manuscript received 28 Oct. 2018; revised 16 Aug. 2019; accepted 20 Aug. 2019. Date of publication 5 Sept. 2019; date of current version 10 Jan. 2020. (Corresponding author: Shungeng Zhang.)
Recommended for acceptance by D. Talia.
Digital Object Identifier no. 10.1109/TPDS.2019.2938500

TABLE 1
Summary of Inefficient Event Processing Flow in Mainstreamed Asynchronous Servers/Middleware

Category	Software Name	Type	Note
Inefficient Event Processing Flow (Section 3 and 4)	Tomcat NIO Connector (refer as `TomcatAsync`)	application server	The reactor thread monitors events while the worker thread processes events, context switch happens between the reactor/worker thread which is similar as `TomcatAsync` design.
	Eclipse Jetty	application server	
	Grizzly/GlassFish	network framework	
	MongoDB Async Driver	database driver	
Improved Event Processing Flow, but with potential optimization overhead (Section 5.1)	Netty	network framework	The worker thread is responsible for both event monitoring and processing, no intermediate context switch, similar as `Netty` design.
	Nginx [9]	web server	
	Lighttpd [10]	web server	Single thread, similar as `SingleT-Async` design.
	Node.js [11]	JavaScript runtime	

find that such a poor design of event processing flow widely exists among Jetty [6], GlassFish [7], and MongoDB Java asynchronous driver [8].

The second contribution is a sensitivity analysis of how different runtime workload and network conditions impact the efficiency of the event processing flow of asynchronous servers. Concretely, we vary the server response size and network latency based on realistic conditions in a typical cloud environment and see their impact on the performance of servers with different architectures. Our experimental results show that an asynchronous server could encounter a severe `write-spin` problem, in which the server makes a large amount of unnecessary I/O system calls when sending a relatively large size of server response (e.g., 100KB), thus wastes the critical CPU resource up to 24 percent. Such a problem is caused by the lack of coordination between the non-blocking nature of asynchronous system calls in the application layer and the TCP wait-ACK mechanism in the OS kernel. Our experiments show that some network conditions (e.g., network latency) could exaggerate the CPU overhead caused by the write-spin problem, leading to a more severe performance drop of the server.

The third contribution is a hybrid solution which exploits the merits of different asynchronous event-driven architectures in order to adapt to dynamic runtime workload and network conditions. We first examined a widely-used asynchronous event-driven network application framework named "Netty [12]", which adopts a write operation optimization technique to alleviate the aforementioned write-spin problem. However, we found that such an optimization introduces non-trivial CPU overhead in the case of the absence of the write-spin problem. Our hybrid solution extends the native Netty by applying a lightweight profiling technique to check whether the write-spin problem exists during the server runtime. Based on the runtime checking results, our solution chooses the most efficient event processing flow for each client request, avoiding both the write-spin problem and the non-trivial optimization overhead.

Overall, our study of asynchronous Internet server efficiency has a potentially significant impact on achieving good performance and high resource efficiency in today's cloud data centers. Plenty of previous research efforts have shown the challenges of achieving high performance at high system utilization, especially for those latency-sensitive interactive web applications [13], [14]. Our work shows that, given the right design of event processing flow, asynchronous Internet servers could continuously achieve stable and high performance under the various runtime workload and network conditions (even at high resource utilization). Our work also provides future research opportunities as many system components (e.g., ZooKeeper [15]) have been shifting from the thread-based architecture to the asynchronous one, thus the similar problems may also occur.

We outline the rest of the paper as follows. Section 2 presents a motivation experiment that merely upgrading an application server from the thread-based version to its asynchronous counterpart causes large system performance loss. Section 3 studies the poor design of event processing flow leading to unnecessary context switch overhead. Section 4 shows the write-spin problem of an asynchronous server sending large size responses. Section 5 introduces our hybrid solution. Section 6 summarized the related work and Section 7 concludes the paper.

2 Background and Motivation

2.1 RPC versus Asynchronous Network I/O

Modern Internet servers generally use either synchronous or asynchronous *connectors* for inter-tier (or between a client and a server) communications. These connectors mainly focus on the following activities: 1) manage network connections from both the upstream and the downstream tiers, 2) read (and write) data through established connections, and 3) parse and route new requests to the application layer (business logic) and vice versa. Although asynchronous and synchronous connectors are similar in functionality, they are very different in the underlying mechanism to interact with the application layer logic.

Synchronous connectors are mostly adopted by the RPC thread-based servers. There are two types of threads in this type connector: the main thread takes care of accepting new connections and dispatching each connection to a dedicated worker thread, and each worker thread will handle all activities of the corresponding connection until the close of it. Accordingly, a large number of worker threads are needed to handle high concurrency workload. Due to the user-perceived sequential processing logic, it is relatively easy for developers to build synchronous thread-based servers, but the overhead associated with multithreading (e.g., locks and context switches) can lead to performance degradation [3].

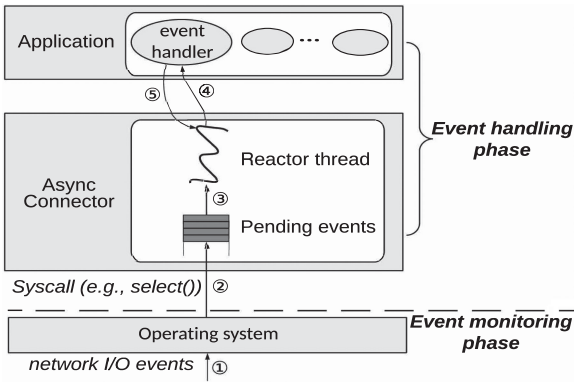


Fig. 1. Interactions of an asynchronous connector between the application server and underlying operating system.

Asynchronous connectors are able to use only one or a few threads for handling high concurrency workload using an event-driven mechanism. Fig. 1 depicts the interactions of an asynchronous connector with the application layer and the underlying operating system. To process a pool of established connections, the asynchronous connector switches between two phases (*event monitoring phase* and *event handling phase*) to handle requests from these connections. The event monitoring phase determines connections with the occurrence of pending network I/O events, such as a readable or writable state of a particular connection. The underlying operating system provides the event notification mechanism (e.g., *select*, *poll*, or *epoll*). The event handling phase will perform the actual business logic by dispatching each event to the corresponding event handler [3], [4], [16].

In practice, there are two typical server designs using the asynchronous connectors. The first design is a single-threaded server which uses only one thread to handle both event monitoring and handling phase (e.g., Lighttpd [10] and Node.js [11]). Previous work [17] shows that such a design is able to minimize multithreading overhead when dealing with in-memory workloads. In the second design, a small size of the worker thread pool is used to concurrently process events in the event handling phase (e.g., the asynchronous Tomcat). Such a design is intended to efficiently exploit a multi-core CPU [16], or deal with complex workload involving transient disk I/O activities. Variants of the second design have been studied, such as the Staged Event-Driven Architecture (SEDA) adopted by Haboob [3].

In general, previous research demonstrates that asynchronous event-driven server is able to outperform the thread-based one in throughput due to the reduced multi-threading overhead, especially for servers facing high concurrency workload. However, our study in the next section will show the contradictory results.

2.2 Experimental Setup

We conduct our experiments using RUBBoS [5], which is a representative n-tier benchmark modeled after the bulletin board applications like *Slashdot* [18]. In our experiments, we configure the benchmark as a typical 3-tier topology as shown in Fig. 2, with one Apache web server, one Tomcat application server, and one MySQL database server. There are 24 servlets providing different interactions, which can be further categorized into browse-only and read/write mixes

(a). Software Stack

Web Server	Apache 2.2.22
Application Server	Tomcat 7.0.57 and Tomcat 8.0.36
Database Server	MySQL 5.5.19
Operating System	RHEL 6.3 (kernel 2.6.32)

(b). Hardware Stack

Processor	Intel Xeon CPU E5-2640, 2.50GHz, Hexa-Core
Memory	16GB
Disk	500GB 7200RPM
Network	1Gbps

(c). 1/1/1 Sample Topology

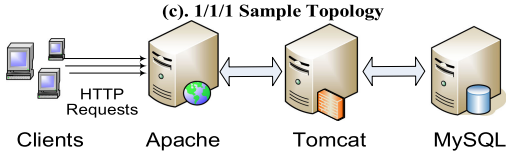


Fig. 2. Details of the experimental setup.

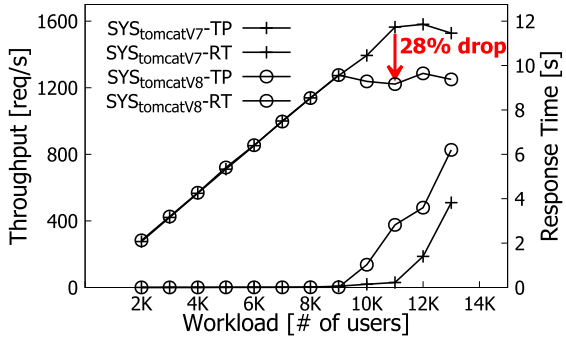


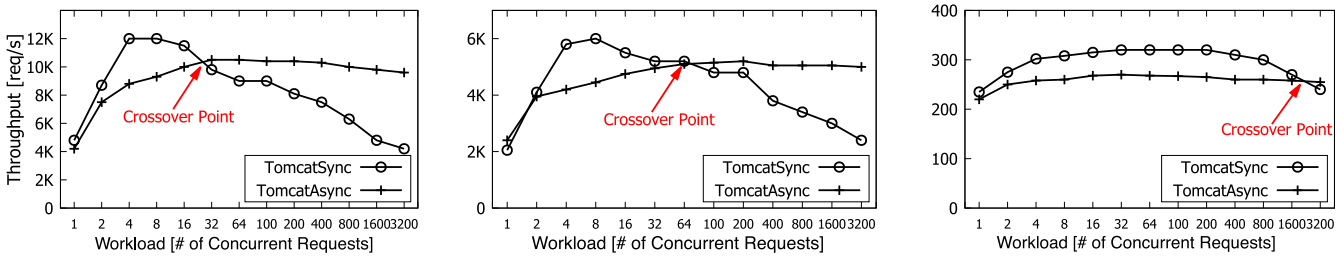
Fig. 3. Significant performance degradation after we merely upgrade Tomcat from the thread-based version 7 to the asynchronous version 8 in a 3-tier system.

workload. We use the former one in this experiment. The response size of each servlet varies from tens to hundreds of kilobytes in a Zipf-like distribution. The default workload generator simulates a number of concurrent users to mimic real user behaviors. Each user browses different pages following a Markov chain model, and the think time between two consecutive requests is averagely 7 seconds. Such a design of workload generator is widely adopted by other typical n-tier benchmarks like RUBiS [19], TPC-W [20], and Cloudstone [21]. We ran the experiments in our private cluster. Fig. 2 shows detailed software configurations, hardware specifications[1] and a sample 3-tier topology.

2.3 Performance Degradation from Tomcat Upgrade

Software upgrade in web-facing n-tier systems is common for system admins due to the rapid application evolvement. In this section, we show a case study that significant system performance loss in a 3-tier RUBBoS benchmark after we upgrade a thread-based application server to its asynchronous counterpart. Concretely, we first adopt Tomcat 7 (noted as `TomcatSync`) as the application server, which adopts a thread-based synchronous connector to communicate with other servers. We then upgrade the Tomcat server to a newer version (version 8, noted as `TomcatAsync`), the default connector of which has changed to the asynchronous one, with the expectation of system performance improvement after the Tomcat upgrade.

1. Only one core is enabled in BIOS unless explicitly mentioned.



(a) The 0.1KB response size case. (b) The 10KB response size case. (c) The 100KB response size case.

Fig. 4. *The concurrency crossover point between* `TomcatSync` *and* `TomcatAsync` *increases as the response size increases from 0.1KB to 100KB.* The throughput of `TomcatSync` surpasses `TomcatAsync` at a wider concurrency range when response size is large (comparing (a) and (c)), suggesting additional overhead for `TomcatAsync` leading to performance degradation.

However, Fig. 3 shows an unexpected system performance drop after the thread-based Tomcat upgrades to its asynchronous counterpart. We use notation $\texttt{SYS}_{tomcatV7}$ and $\texttt{SYS}_{tomcatV8}$ to represent the system with `TomcatSync` and `TomcatAsync`, respectively. The figure shows that the throughput of $\texttt{SYS}_{tomcatV8}$ stops increasing at workload 9000, which is much earlier than $\texttt{SYS}_{tomcatV7}$. At workload 11000, $\texttt{SYS}_{tomcatV7}$ achieves 28 percent higher throughput than $\texttt{SYS}_{tomcatV8}$, and the corresponding average response time is significantly increased by a factor of ten (300ms versus 3s). Considering that we merely upgrade a thread-based Tomcat server to a newer asynchronous one, such a result is counter-intuitive. We note the bottleneck resource in the system is the CPU of the Tomcat server in both cases, while the utilization of the hardware resources (memory, disk I/O, etc.) of all other components is moderate (less than 60 percent).

We also observed another interesting phenomenon that the asynchronous `TomcatAsync` experiences significantly higher frequency of context switches than the thread-based `TomcatSync` when facing the same workload. We monitor system-level metrics using Collectl [22]. For example, `TomcatAsync` encountered more than twice context switches per second than `TomcatSync` (12950 /sec versus 5930 /sec) at workload 11000. Since the high frequency of context switches causes high CPU overhead, it makes sense to suggest that the throughput gap between $\texttt{SYS}_{tomcatV7}$ and $\texttt{SYS}_{tomcatV8}$ (see Fig. 3) is caused by the different level of context switches in Tomcat. We note the Tomcat CPU is the bottleneck in the system. However, significant previous work shows that an asynchronous server is supposed to have much fewer context switches than a thread-based server, so why do we observe the contradictory results here? We will answer this question in the next section.

3 Inefficient Event Processing Flow

In this section, we introduce the inefficient event processing flow problem, which results in the system performance degradation of the 3-tier RUBBoS benchmark after the thread-based Tomcat server upgrades to its asynchronous counterpart. In the following experimental evaluation, we separate out Tomcat for better quantifying our performance analysis on different versions of Tomcat.

3.1 Unnecessary Context Switches between Event Handlers

In this set of experiments, we use JMeter [23] as a workload generator sending HTTP requests to access Tomcat (both the thread-based and the asynchronous) directly (no Apache and MySQL is involved). We divide these HTTP requests into three categories: small, medium, and large, which are based on the response size of each request. Concretely, the Tomcat server will respond with 3 sizes of responses (i.e., 0.1KB, 10KB, and 100KB) according to different types of requests from JMeter. To simulate realistic business logic, the Tomcat server will generate a corresponding response (e.g., a 0.1KB/10KB/100KB random string) on-the-fly during runtime, such generation process (or computation) of each request is proportional to the response size. We choose these three sizes of server response because they are representative of the RUBBoS benchmark. We note that JMeter adopts threads to emulate real users sending requests. To precisely control the workload concurrency to the target Tomcat server, we set the think time between every two consecutive requests from each client thread to be zero.

We first compare the throughput between the thread-based `TomcatSync` and the asynchronous `TomcatAsync` under different workload concurrencies and response sizes, shown in Fig. 4. An interesting phenomenon is that `TomcatSync` outperforms `TomcatAsync` in throughput when the workload concurrency is less than a certain point (referred as the crossover point), and then the throughput superiority of two servers is reversed as the workload concurrency continues to increase. For example, the crossover point workload concurrency is 64 in the 10KB response size case (Fig. 4b), and 1600 in the 100KB response size case (Fig. 4c). We note that the response size for RUBBoS benchmark in Section 2 varies from tens to hundreds of kilobytes in a Zipf-like distribution and the average is about 20KB, and the request processing concurrency in Tomcat is averagely 35 when the system approaches saturation. According to our experimental results here, it is not surprising that `TomcatSync` outperforms `TomcatAsync` in the 3-tier RUBBoS benchmark experiments, leading to higher throughput of $\texttt{SYS}_{tomcatV7}$ than that of $\texttt{SYS}_{tomcatV8}$ since the Tomcat server is the bottleneck. The question is why the thread-based `TomcatSync` outperforms the asynchronous `TomcatAsync` before a certain workload concurrency.

Our further analysis shows that it is the poor design of the event processing flow in `TomcatAsync` that creates a large number of context switches which leads to non-trivial CPU overhead. In Table 2, we show that the frequency of context switches in the asynchronous `TomcatAsync` is significantly higher than that in the synchronous `TomcatSync` when

TABLE 2
TomcatAsync Encounters More Context Switches than TomcatSync in Different Response Size Cases

Workload concurrency	Response size [KB]	TomcatAsync	TomcatSync
		[×1000/sec]	
8	0.1	**40**	16
	10	**25**	7
	100	**28**	2
100	0.1	**38**	15
	10	**26**	5
	100	**25**	2
3200	0.1	**37**	15
	10	**22**	6
	100	**28**	3

The workload concurrency varies from 8 to 3200.

facing the same concurrency workload (e.g., from 8 to 3200). Such results are consistent with our observations in the previous RUBBoS experiments. We note that TomcatAsync uses the second asynchronous server design, in which the server monitors events by a reactor thread (event monitoring) and handles events by a small size of worker thread pool (event handling) (see Section 2.1). To process a new incoming request, Fig. 5 illustrates the event processing flow in TomcatAsync, which includes the following four steps:

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

Accordingly, TomcatAsync needs four context switches between the reactor thread and the worker threads to process one client request. Such an inefficient design of the event processing flow is widely adopted by many representative asynchronous software (see Table 1), showing a

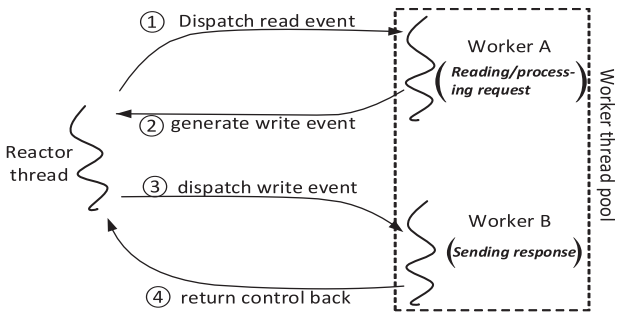


Fig. 5. *Inefficient event processing flow in* TomcatAsync *for one client request processing.* There are four context switches between the reactor thread and the worker threads.

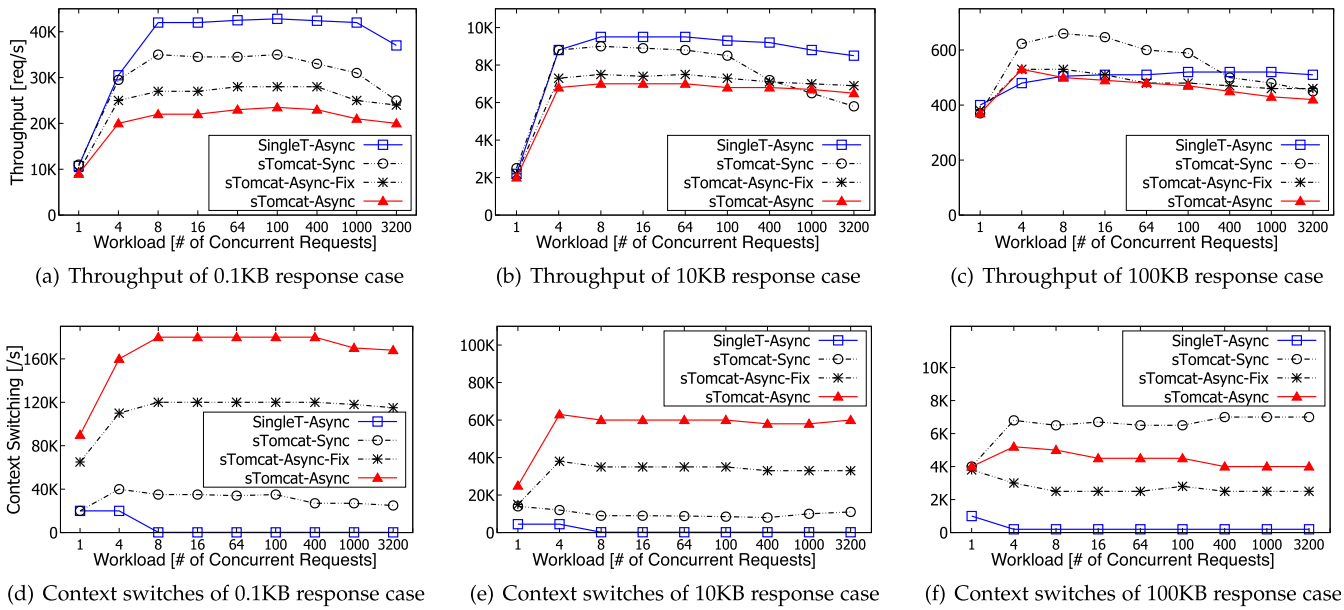
TABLE 3
Summary of Different Types of Servers with Associated Context Switches among the Threads Running in the User-Space (e.g., the Reactor Thread and Worker Threads) when Processing One Client Request

Server type	Context Switch	Note
sTomcat-Async	4	Different worker threads handle read and write events respectively.
sTomcat-Async-Fix	2	The same worker thread handles read and write events.
sTomcat-Sync	0	Dedicated worker thread for each request. Context switches only occur by interrupt or swapped out due to running out of CPU quota.
SingleT-Async	0	One thread handles both event monitoring and processing. No context switches exist.

Other implicit context switches such as interrupt or swapped-out are not counted.

general problem in designing asynchronous software. On the other hand, each client request in the thread-based TomcatSync is dispatched to a dedicated worker thread, which is intended to handle all the activities associated with this request, including reading the request, preparing the response, and sending the response out. Therefore, the context switch only happens when the processing worker thread is interrupted or swapped out by the operating systems due to running out of CPU quota.

To better quantify the performance impact of context switches on servers with different architectures, we simplify the design of TomcatAsync and TomcatSync by ruling out some unnecessary modules (e.g., cache management and logging) and only keep the necessary parts related to the business logic. We refer the simplified TomcatAsync and TomcatSync as sTomcat-Async and sTomcat-Sync, respectively. We also implement two alternative asynchronous servers with reduced context switches as a reference. The first alternative is sTomcat-Async-Fix, which uses the same worker thread to process both the read and the write events of the same request. In this case, the same worker thread, after finishing preparing the response, will continue to send the response out (step 2 and 3 in Fig. 5 are merged with step 4), thus only two context switches are required to process one client request: 1) the reactor thread dispatches a read event to one available worker thread in the thread pool, and 2) the same worker thread returns the control back to the reactor thread after sending response out. The second alternative design is SingleT-Async, which adopts a single thread to process events in both event monitoring and event handling phase. Such a design is supposed to avoid the context switch overhead. We summarize the four types of servers with their associated context switches when processing one client request in Table 3. Readers who are interested can refer to our source code of server implementation from our repository [24].



(a) Throughput of 0.1KB response case (b) Throughput of 10KB response case (c) Throughput of 100KB response case

(d) Context switches of 0.1KB response case (e) Context switches of 10KB response case (f) Context switches of 100KB response case

Fig. 6. *Performance comparison among four architecturally different servers when the size of server response increases from 0.1KB to 100KB.* Subfigure (a) and (d) show the negative correlation between server throughput and corresponding context switch frequency of each server type. However, when the response size is large (100KB), subfigure (c) shows that `sTomcat-Sync` performs the best compared to other types of servers before the workload concurrency 400, suggesting other factors create additional overhead in asynchronous servers.

We show the performance comparison among the four architecturally different servers under different workload concurrencies and response sizes as shown in Fig. 6. Through Figs. 6a and 6d, we observe the negative correlation between the server throughput and the corresponding frequency of context switches of each server type. For example, `sTomcat-Async-Fix` achieves 22 percent higher throughput than `sTomcat-Async` while the context switch frequency is 34 percent less when the server response size is 0.1KB and the workload concurrency is 16 in Figs. 6a and 6d, respectively. In this set of experiments, the computation for each request is proportional to the server response size, which means in the small server response size scenario (e.g., the 0.1KB case), more CPU cycles will be wasted in context switches compared to those consumed in actual request processing. For example, in the 0.1KB response size case, the gap of the context switches between `sTomcat-Async-Fix` and `sTomcat-Async` in Fig. 6d could reflect the throughput difference in Fig. 6a. We further validate such a hypothesis by the other two types of servers `SingleT-Async` and `sTomcat-Sync`, which achieves 91 percent (42K req/sec versus 22 req/sec) and 57 percent (35K req/sec versus 22 req/sec) higher throughput than `sTomcat-Async` at workload concurrency 100, respectively (Fig. 6a). The context switch comparison in Fig. 6d can help explain the throughput difference. For example, `SingleT-Async` only encounters a few hundred per second context switches (due to other daemon processes such as monitoring tools like Collectl [22] and perf [25]), which is three orders of magnitude less than that of `sTomcat-Async`.

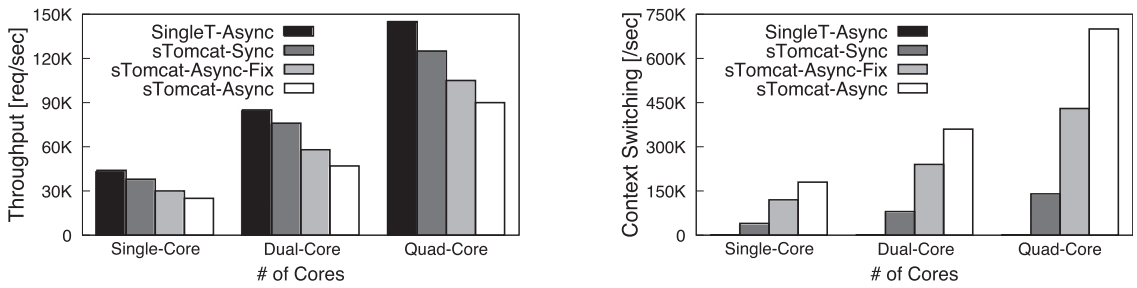
Recall our previous study in Table 3, context switches for the thread-based `sTomcat-Sync` only occur when the processing worker thread is interrupted or swapped out due to running out of CPU quota; on the other hand, context switches for the asynchronous `sTomcat-Async` happen when the events are dispatched between the reactor thread and the processing worker thread (e.g., step 1∼4 in Fig. 5). In this case, the lock operation is required to synchronize the threads (the reactor thread and the worker threads), which introduces lock contention overhead. We then use `perf` [25] (a performance analysis tool) to validate our hypothesis in Table 4. The server response size is 0.1KB and the workload concurrency is 100. Our results show that the lock contention between threads to coordinate information (e.g., connection context) plays a significant role in performance overhead for asynchronous servers with a design of inefficient event processing flow. For example, the futex overhead and cache miss in `sTomcat-Async` is 13.86 and 0.07 percent respectively, which is the highest out of four servers. Such high overhead further results in less CPU efficiency (i.e., the lowest instructions per cycle), leading to significant throughput loss.

We note that the portion of the CPU overhead associated with context switches becomes less when the size of the server response becomes larger. This is because more CPU cycles will be consumed for processing requests and sending responses given the same number of context switches. Fig. 6b and 6c show the throughput comparison of different

TABLE 4
Comparison of Futex Lock Overhead, Cache Miss, and Instructions per Cycle among Four Architecturally Different Servers

	SingleT-Async	sTomcat-Sync	sTomcat-Async-Fix	sTomcat-Async
Throughput [req/sec]	42K	35K	27K	22K
Futex lock [%]	0.49	7.62	11.05	**13.86**
Cache miss [%]	0.054	0.059	0.064	**0.070**
Instructions per cycle [#/sec]	1.15	0.95	0.90	**0.82**

The server response size is 0.1KB and the workload concurrency is 100.



(a) Throughput comparison (b) Context switch comparison

Fig. 7. *The context switch problem caused by inefficient event processing flow also occurs in a multi-core environment.* The workload concurrency keeps 100. The server response size is 0.1KB so that the computation of each request is light. Thus the throughput difference in (a) is mainly caused by the the context switch difference in (b).

servers with the 10KB and the 100KB response size, respectively. The throughput gap among these four types of servers becomes narrower, suggesting that context switches have less impact on server throughput.

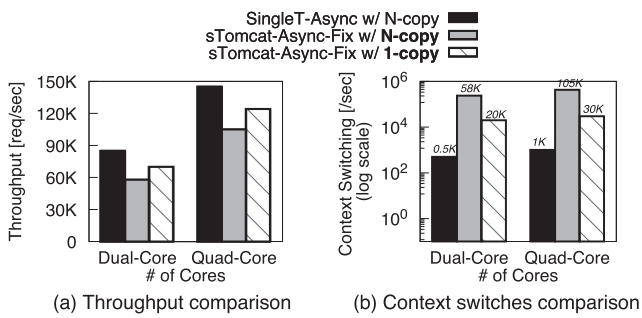
In fact, we also observe another interesting phenomenon that the asynchronous `SingleT-Async` achieves lower throughput than the thread-based `sTomcat-Sync` when the workload concurrency is less than 400 as shown in Fig. 6c. Although the context switches in `SingleT-Async` are much less than those in `sTomcat-Sync` (see Fig. 6f). These results suggest that other factors introduce significant overhead in the asynchronous `SingleT-Async` as the server response size increases (e.g., 100KB). We will explain these factors in Section 4.

3.2 Evaluation in a Multi-Core Environment

Multi-core has been rapidly adopted in cloud data centers, thus one requirement for modern Internet servers lies in the ability to scale-out in a multi-core hardware setting [26], [27]. Our investigation shows that the context switch overhead caused by inefficient event processing flow also has a significant impact on the performance of asynchronous servers in a multi-core environment (see Fig. 7). Previous studies [28], [29] already show that the N-copy model is widely adopted as a successful solution to enabling an asynchronous server to leverage multiple CPUs in a multi-core environment. For example, the asynchronous `SingleT-Async` only uses one thread, we adopt the N-copy model for `SingleT-Async` and each copy consumes one CPU core; N equals the number of cores enabled in the host. To avoid CPU crosstalk penalty [30], we use CPU affinity to launch multiple copies of servers in a multi-core environment. To conduct a fair comparison, we also apply the N-copy model to the other three types of servers. Interested readers can refer to Veal's work [26] which discusses the challenges to scale-up web servers to multi-core. Nevertheless, the N-copy model is a common practice in scaling modern Internet servers, especially in the emerging microservices architecture, where each micro-service can scale-out multiple replicas to handle workload increase [31].

We set the workload concurrency to 100 (high enough to saturate the quad-core CPU) and the server response size to 0.1KB in all cases. Fig. 7a shows that the throughput of each server scales almost linearly as the number of cores increases; Fig. 7b shows the frequency of context switches of different servers. These two figures show the consistent results as in the single-core case (Fig. 6a and 6d), where the inefficient event processing flow causes frequent context switches in asynchronous servers and degrades the server performance in a multi-core environment.

On the other hand, We note that the asynchronous servers such as `sTomcat-Async` and `sTomcat-Async-Fix` delegate event processing to the small size of the worker thread pool (see Section 2.1). Such a design is intended to efficiently exploit multiple CPUs in a multi-core environment since most of the computations (business logic) rely on the worker thread pool. We conduct the same experiments on asynchronous `sTomcat-Async-Fix` with only a single instance running in a multi-core environment in Fig. 8, we refer it as `sTomcat-Async-Fix w/`*1-copy*. An interesting observation is that in Fig. 8a, `sTomcat-Async-Fix w/ 1-copy` outperforms `sTomcat-Async-Fix w/ N-copy` by 13 percent in throughput in a dual-core environment. Such a performance improvement is because that `sTomcat-Async-Fix` encounters less context switches in the `1-copy` case, as shown in Fig. 8b. Recall our previous study in Table 3, context switches for the asynchronous `sTomcat-Async-Fix` happen when the events are dispatched between the reactor thread and the processing worker thread (i.e., step 1 and 4 in Fig. 5). In a dual-core environment, the reactor thread and the processing worker thread could be running on separate CPUs due to the operating system process scheduling, thus such event dispatches between these two threads only involve the thread coordination among CPUs instead of context switches. In this case, the context switch overhead is significantly reduced.



(a) Throughput comparison (b) Context switches comparison

Fig. 8. The `1-copy` model mitigates context switches problem in a multi-core environment for `sTomcat-Async-Fix`, but it cannot solve the problem. The workload concurrency keeps 100 and the server response size is 0.1KB.

TABLE 5
More User-Space CPU Resource is Consumed in SingleT-Async than that in sTomcat-Sync

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

We set the workload concurrency to 100 in all cases.

TABLE 6
Severe Write-Spin Problem Happens in 100KB Response Size Case

Response Size	# req.	# socket.write()	# write() per req.
0.1KB	238530	238530	1
10KB	9400	9400	1
100KB	2971	303795	**102**

The table shows the total number of socket.write() for each request in SingleT-Async under different response size during a one-minute experiment.

However, SingleT-Async w/ N-copy still performs the best in all multi-core cases in Fig. 8a, showing that 1-copy model only mitigates the context switch problem for sTomcat-Async-Fix, it cannot solve the problem completely.

Summary. Through our extensive experiments on four architecturally different servers (Single-Async, sTomcat-Async, sTomcat-Async-Fix, and sTomcat-Sync), we observed that Single-Async is able to achieve the best performance under various workload concurrencies when the response size is small. The main reason is because Single-Async reduces the multithreading overhead (e.g., context switches and lock) caused by inefficient event processing flow, which generates frequent unnecessary intermediate events, as shown in the other three servers (i.e., sTomcat-Async-Fix, sTomcat-Async, and sTomcat-Sync). In the next section, we will discuss other factors that make Single-Async less efficient, for example, when the server response size is large (e.g., 100KB).

4 Write-Spin in Asynchronous Invocation

In this section, we analyze the performance degradation of an asynchronous server in the large response size case. We measure the CPU usage in both the user/kernel space and profile some critical system calls of servers with different architectures using monitoring tools like Collectl [22] and JProfiler [32]. Our measurements show that an asynchronous server could encounter a severe write-spin problem in which the server invokes a large amount of unnecessary I/O system calls when sending a large size server response, thus degrades the server efficiency. We then explore some realistic factors in cloud data centers that could exaggerate the negative effect of the write-spin problem, degrading an asynchronous server performance further.

4.1 Overhead Caused by Write-Spin

Recall the experimental results in Fig. 6a, which shows that the asynchronous SingleT-Async outperforms the thread-based sTomcat-Sync 20 percent in throughput at the workload concurrency 8 in a small response size scenario (i.e., 0.1KB). However, such a throughput superiority is reversed at the same workload concurrency once the server response increases to 100KB (Fig. 6c). Such a result suggests that sending a large size of server response brings a significant overhead for the asynchronous SingleT-Async but not for the thread-based sTomcat-Sync.

To study the throughput drop of SingleT-Async in a large size response scenario, we collect the performance metrics (e.g., CPU) of the server with different response sizes using Collectl. We show the CPU utilization comparison between SingleT-Async and sTomcat-Sync as we increase the response size from 0.1KB to 100KB as shown in Table 5. The workload concurrency is 100 and the CPU of both servers is 100 percent utilized. The table shows that when increasing the 0.1KB response size to 100KB, the CPU consumption in user-space of the asynchronous SingleT-Async increases 25 percent (80-55 percent), which is much less than 34 percent (92-58 percent) of the thread-based sTomcat-Sync. Such a result indicates that SingleT-Async is more sensitive than sTomcat-Sync in user-space CPU utilization as response size increases.

We then profile SingleT-Async in different server response size cases by JProfiler during the experiment runtime, and see the difference of application-level activities. We observed that the frequency of system call socket.write() is exceptionally high when the response size is 100KB in Table 6. In fact, socket.write() will be called when a server tries to send a response out. For example, the synchronous thread-based sTomcat-Sync will call socket.write() only once when processing each client request regardless of the size of the server response. Such a pattern is also true for asynchronous SingleT-Async in the case of the 0.1KB and 10KB response sizes. However, the table shows SingleT-Async requires averagely 102 calls of socket.write() per request in the 100 KB response case. It is well-known that system calls are expensive because of the associated kernel-user switching overhead [33], thus the high CPU overhead in user-space of SingleT-Async sending a large response (in Table 5) can be explained.

We further investigate the exceptionally high socket writes in SingleT-Async, which are caused by a combination of a small size TCP send buffer (16KB by default) and the TCP wait-ACK mechanism. We refer it as the write-spin problem in Fig. 9. Concretely, the processing thread in SingleT-Async invokes a Java library method *java.nio.channels.SocketChanne.write()* [34], which wraps the system call socket.write(). In this case, the method tries to transfer 100KB data to the TCP send buffer, but it is only able to transfer at most 16KB data at first because of the limited size of TCP send buffer, which is structured as a byte buffer ring. A TCP sliding window determines the actual amount of data to be sent to the client and frees up the occupied TCP send buffer space only if the ACKs are received from the previously sent-out packets. Considering the non-blocking nature of the asynchronous servers, such a library method in SingleT-Async immediately returns the total amount of bytes copied to the TCP send buffer once called; in the worst case, it returns zero when the TCP send buffer is already full

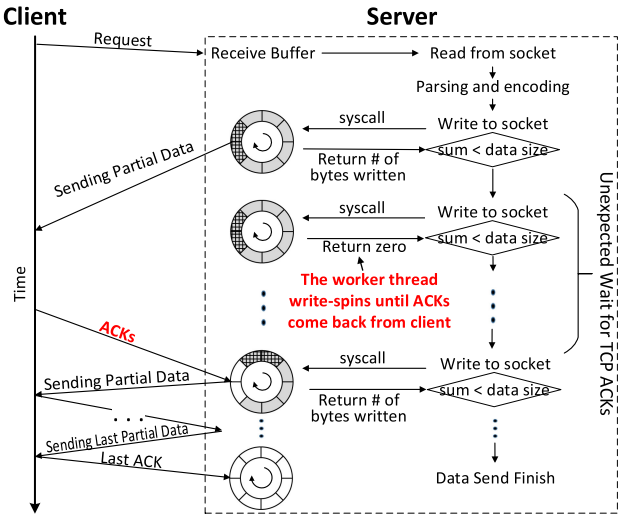


Fig. 9. Write-spin problem causes unnecessary system calls, leading to severe CPU overhead and throughput loss. Due to a small TCP send buffer size and the TCP wait-ACK mechanism, a worker thread write-pins on the `socket.write()` and can only continue to send more data after receiving ACKs of previously sent packets.

(i.e.,system call `socket.write()` returns errno like EWOULDBLOCK, indicating the fullness of TCP send buffer [35]), resulting in a severe write spin problem. In contrast, the method to transfer data in the thread-based `sTomcat-Sync` is blocking; the actual write loop (data transfer) occurs in kernel throughput the limited TCP send buffer, and it is much more efficient than that occurs in user-space, which is what the non-blocking `socket.write()` in the asynchronous `SingleT-Async` does (the write spin-problem). In this case, `sTomcat-Sync` calls only one such a method for each request, avoiding unnecessary spins of the processing worker thread in `SingleT-Async`.

A straightforward solution is to manually set the TCP send buffer to the same size (or even larger) as the server response. However, in practice it is a non-trivial task due to the following three reasons. First, predicting the response size of internet services is difficult since the web application workloads are dynamic by nature. For example, the responses from a Tomcat server can vary from tens of bytes to megabytes since requests may require dynamic content from the downstream databases. Second, HTTP/2 introduces *Server Push*, which allows a server to push several responses for answering one client request [36]. For example, with HTTP/2 Server Push, a typical news website like CNN.com can reply for one request with multiple responses which may easily accumulate up to tens of megabytes (e.g., static and dynamic contents such as images and database query results). Third, setting an oversized size TCP send buffer only for the peak size of server responses could lead to TCP over-buffering, which not only risks running out of the server memory under high concurrency workload but also causes the sluggish interactive response problem [37]. Thus, it is a big challenge to set an appropriate size of the TCP send buffer in advance to avoid the write-spin problem.

In fact, *TCP Auto-Tuning* function already presents in Linux kernel above 2.4, which is supposed to automatically adjust the size of the TCP send buffer to maximize the bandwidth utilization [38] according to the runtime network

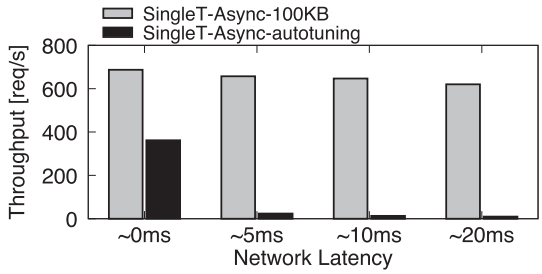


Fig. 10. TCP Auto-Tuning is unable to solve write-spin problem due to insufficient application context. The server response size is 100KB and the workload concurrency is 100.

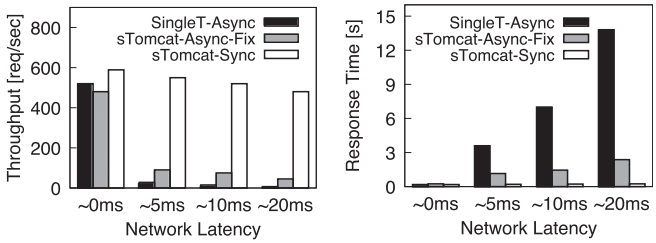
conditions. However, TCP Auto-Tuning mainly focuses on maximizing the utilization of the available bandwidth of the link between the client and the server using Bandwidth-Delay Product (BDP) rule [39] without the knowledge of the application information such as response sizes. Besides, our experiments show that the default TCP Auto-Tuning algorithm is conservative in choosing the send buffer size to avoid frequent packets loss, thus avoid more delay for the application caused by subsequent TCP retransmissions. As a result, the size of the send buffer after auto-tuning may still be deficient for applications, resulting in the write-spin problem for the asynchronous servers. Fig. 10 shows `SingleT-Async` with auto-tuning performs worse than the other case with a fixed large TCP send buffer (100KB), suggesting the occurrence of the write-spin problem. We note that TCP auto-tuning is intended to maximize the bandwidth utilization regardless of different runtime network conditions. In this set of experiments, we also vary network latency between the server node and the client node from 0ms to 20ms. The experimental results show that the performance gap between two servers is notably enlarged when non-negligible network latency exists as shown in Fig. 10. We will discuss more in the next section.

4.2 Write-Spin Exaggerated by Network Latency

Network latency is inevitable in modern cloud data centers. In general, it ranges from a few to tens of milliseconds depending on the location of the component servers, which may run on the different physical nodes located in different racks or even data centers. Our experiments reveal that the non-negligible network latency can exaggerate the overhead caused by the write-spin problem, leading to significant performance loss.

We show the impact of network latency on the performance of the thread-based and asynchronous servers in Fig. 11. The workload concurrency is 100 and the server response size is 100KB. The TCP send buffer size is 16KB by default, with which the write-spin problem occurs in the asynchronous servers. To quantitatively control the network latency, we use the traffic control tool "tc" in the client node. Fig. 11a shows that in a non-negligible network latency scenario, both the asynchronous `SingleT-Async` and `sTomcat-Async-Fix` have a significant throughput drop. For example, the maximum achievable throughput of `SingleT-Async` surprisingly degrades by 95 percent when a small 5-millisecond increase in network latency.

Our further analysis shows that such a significant throughput degradation is caused by the amplification effect



(a) Throughput comparison (b) Response time comparison

Fig. 11. Performance degradation of asynchronous servers is exaggerated when write-spin occurs and non-negligible network latency exists. The server response size is 100KB while the TCP send buffer size is the default 16KB.

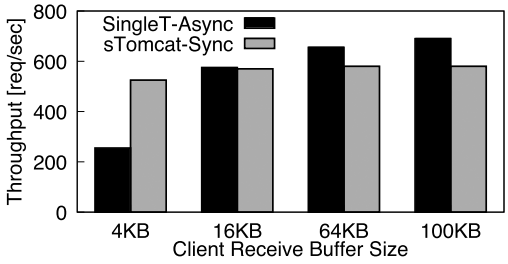


Fig. 12. *Write-spin problem still exists when the client receive buffer size is small, due to the TCP flow control mechanism.* We set the server response size to 100KB. The server TCP Auto-Tuning feature is enabled by default.

on response time when the write-spin problem occurs. In a write-spin scenario, an asynchronous server needs multiple rounds of data transfer to send a large size response (e.g., 100KB) out because of the small TCP send buffer. The server only continues to transfer data until it receives the ACKs from the clients for the previously sent-out packets (see Fig. 9). Therefore, a small increase in network latency can lead to a long delay in the server response time in Fig. 11b. For instance, the response time of `SingleT-Async` is amplified from 0.18s to 3.60s after adding 5-millisecond network latency. Based on Little's Law, the server throughput has a negative correlation with the response time if the workload concurrency (concurrent requests in the server) keeps the same. Thus, 20 times increase in the server response time leads to 95 percent throughput degradation of `SingleT-Async` as shown in Fig. 11a.

4.3 Impact of Client Receive Buffer Size

Other network factors may also trigger the write-spin problem in an asynchronous server, causing significant performance degradation. For example, a client's TCP receive buffer size decides how much data that the sender (the server) can send at one time. Small TCP receive buffer size means the server needs to transfer a large size of response multiple times in order to finish the transfer, which exaggerates the write-spin problem and degrades the server throughput [39]. In fact the receive buffer size is determined by the TCP flow control mechanism between a client and a server (similar to the send buffer size in the server side), however, the diversity of clients in recent years (e.g., cell phones or tablets) may limit a client's send buffer size due to its limited physical resources (e.g., memory). For example, the receive buffer size in the popular mobile OS Android [40] is only 4098 bytes (≈4KB) [41], which is far from sufficient for most modern web applications.

In this set of experiments, we vary the client receive buffer size from 4KB to 100KB to study its impact on the performance of an asynchronous server when sending a relatively large response size (100KB), shown in Fig. 12. The thread-based `sTomcat-Sync` acts as a baseline. The network latency in both cases keeps zero. This figure shows that `SingleT-Async` achieves 170 percent higher throughput when the client receive buffer increases from 4KB to 100KB. The poor performance in the 4KB case is because of the severe write-spin problem caused by small client receive buffer size and the TCP flow control mechanism as mentioned above. On the other hand, `sTomcat-Sync` has a more stable performance under different client receive buffer settings because multithreading mitigates the write-spin problem through parallel data transfer as shown in Fig. 12.

5 Solution

In the previous sections, we have studied two design deficiencies of asynchronous servers due to the inefficient event processing flow: the context switch problem and the write-spin problem. The former one is caused by the poor design of unnecessary event dispatching between the reactor thread and the worker threads (see Table 3), while the latter one results from the unpredictability of the server response size and the limited TCP send buffer size. Although our work is motivated by a 3-tier system throughput drop caused by an inefficient asynchronous Tomcat server (see Section 2.3), we found that many open-sourced asynchronous software packages suffer from the same problems as in the asynchronous Tomcat (see Table 1).

To design a good performance and high efficiency asynchronous server, we should solve the aforementioned two deficiencies under different runtime workload and network conditions. In this section, we first study Netty [12], a widely-used asynchronous event-driven network I/O framework. Netty employs an improved design of event processing flow and provides application-level write operation optimization, with the aim of mitigating the overhead caused by two deficiencies mentioned above, but with a non-trivial optimization overhead. We then present our hybrid solution, which aims to solve the two deficiencies while avoids the Netty optimization overhead by exploiting the merits of the different asynchronous architectures.

5.1 Netty for Reducing Context Switches and Write-Spin

Netty is a widely-used asynchronous event-driven network I/O framework for rapid development of good performance Internet servers. Netty can be categorized into the second design of asynchronous architectures (see Section 2.1), which uses a reactor thread to accept new connections and small size of the worker thread pool to handle established connections with pending events.

Although using worker threads, Netty makes two significant changes to minimize the context switches compared to `sTomcat-Async` and `sTomcat-Async-Fix`. First, the reactor thread and the worker threads in Netty take different

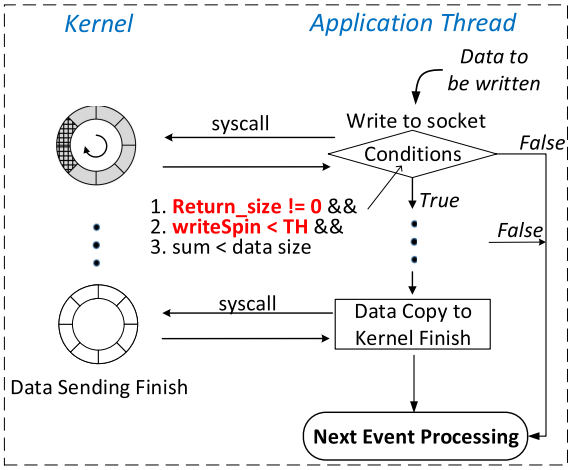


Fig. 13. Netty adopts a runtime checking to mitigate the overhead caused by write-spin problem.

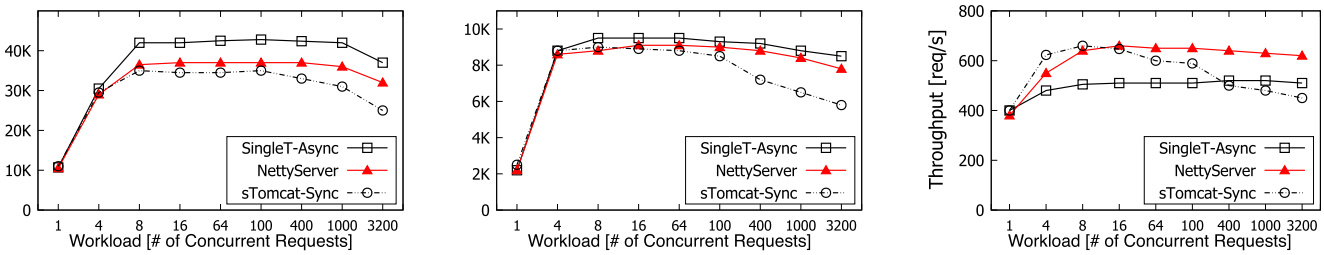
roles compared to those in `sTomcatAsync` and `sTomcat-Async-Fix`. We note that in the case of `sTomcat-Async` and `sTomcat-Async-Fix`, the reactor thread is responsible for accepting new connections and monitoring events, and worker threads only take charge of handling events; the event dispatches between the reactor thread and worker threads involve context switches (step 1~4 in Fig. 5). On the other hand, such frequent event dispatching no longer exists in Netty: the reactor thread only takes charge of accepting new connections and assigning each established connection to a worker thread; each worker thread takes charge of both monitoring and handling events for the assigned connections. As a result, such a role change of the reactor thread and the worker threads in Netty can reduce context switches significantly. Second, Netty adopts a pipeline design of event handlers for business logic, in which the output of a predecessor handler is passed to the next handler in line through a function call (all handlers are processed by the same worker thread), thus avoiding unnecessary intermediate events and the associated context switches between the reactor thread and the worker threads.

To alleviate the write-spin problem, Netty uses a runtime write-spin checking when the processing worker thread tries to send a large amount of data (i.e., server responses) to the kernel using `socket.write()`. Concretely, each Netty worker thread records the total number of `socket.write()` has been called to copy a single response to TCP send buffer, noted as a counter `writeSpin`, shown in Fig. 13. For each `socket.write()`, the processing worker thread keeps track of the total bytes of data which has been sent to the kernel, referred as `return_size`. We note that the processing worker thread will jump out the write spin if either two of the following conditions is met:

- The `return_size` equals to zero, suggesting the fullness of the TCP send buffer;
- The `writeSpin` is greater than a user-defined threshold (16 in Netty-v4 by default), indicating a severe write-spin problem;

When jumping out, the processing worker thread will suspend current data transfer, save the connection context, and resume this data transfer after it loops over other available connections with pending events. Thus Netty is able to prevent the processing worker thread from blocking on a certain connection for copying a large size response to the TCP send buffer in the kernel. However, such an optimization brings the inevitable CPU overhead when no write-spin problem exists in the small size response case.

We demonstrate the effectiveness of a Netty-based server to mitigate the write-spin problem but with the associated optimization overhead in Fig. 14. We build a Netty-based simple application server, noted as `NettyServer`. The figure shows the throughput comparison among three types of servers (`Single-Async`, `NettyServer`, and `sTomcat-Sync`) under different workload concurrencies and response sizes. To evaluate the impact of performance on different servers in both with and without write-spin problem scenarios, the TCP send buffer size is set to 16KB by default. Thus no write-spin problem occurs in the 0.1KB and 10KB response size cases, but a serious write-spin problem in the 100KB response size case. We show that `NettyServer` outperforms other two types of servers when the response size is 100KB, shown in Fig. 14c. For example, `NettyServer` achieves 27 percent higher throughput than `Single-Async` as the workload concurrency is 100, indicating the effectiveness of `NettyServer`'s write optimization technique in mitigating the write-spin problem; `NettyServer` achieves 10 percent higher throughput than `sTomcat-Sync` at the same workload concurrency, suggesting `NettyServer` minimizes the heavy multithreading overhead. However, such performance superiority is reversed as the response size decreases to 0.1KB and 10KB in Fig. 14a and 14b. For example, `NettyServer` performs 17 percent less in throughput



(a) Response size is 0.1KB. (b) Response size is 10KB. (c) Response size is 100KB.

Fig. 14. Netty effectively mitigates the write-spin problem in the large response size case but introduces non-trivial write optimization overhead in the small response size case. We set the size of TCP send buffer to the default 16KB. (a) and (b) show that `NettyServer` has lower throughput than `SingleT-Async`, suggesting non-trivial write optimization overhead, while (c) shows that Netty achieves the best performance out of three types of servers, indicating the effectiveness of alleviating the write-spin problem.

compared to SingleT-Async at the workload concurrency 100 when the response size is 0.1KB in Fig. 14a, suggesting the non-trivial optimization overhead in the case of the absence of the write-spin problem in NettyServer. Thus there is no one-size-fits-all solution that outperforms the other types of servers under various workload conditions.

We also found such a non-trivial optimization overhead widely exists in many mainstreamed asynchronous servers (see Table 1), for example, Nginx [9] and Lighttpd [10].

5.2 A Hybrid Solution

So far we have shown that an appropriately chosen asynchronous solution (see Fig. 14) can always provide better performance than the thread-based counterpart under various runtime workload conditions. However, there is no one-size-fits-all asynchronous solution which always achieves the best performance. Concretely, SingleT-Async encounters the write-spin problem when the response size is large (see Section 4.1); and NettyServer encounters the non-trivial optimization overhead when the response size is small (see Section 5.1). To address such two server design deficiencies, we propose a hybrid solution by exploiting the merits of both SingleT-Async and NettyServer regardless of different runtime workload and network conditions. There are two assumptions for our hybrid solution:

- The response size is unpredictable.
- The workload is in-memory workload.

The first assumption eliminates the case that the server is launched with a large fixed size of TCP send buffer for each connection to avoid the write-spin problem. It is valid due to the difficulty of predicting the server response size and the over-buffering problem that we have discussed in Section 4.1. The second assumption excludes the case of frequent disk I/O blocking the processing worker thread. This is also valid since in-memory stores like Memcached [42] and Redis [43] are widely used by modern internet services due to the strict low-latency requirement [44]. The solution for workloads involving frequent disk I/O is beyond the scope of this paper and requires additional research.

Our hybrid solution integrates the merits of different asynchronous architectures to efficiently handle client requests under various runtime workload and network conditions, shown in Fig. 15. We refer our hybrid solution as HybridNetty. Concretely, HybridNetty extends the native Netty by applying a lightweight profiling technique to check the occurrence of the write-spin problem for each request at the beginning of server runtime (i.e., the initial warm-up phase). In this case, HybridNetty categorizes all incoming requests into two classes: the *writeSpinReq* requests and the *nonWriteSpinReq* requests. The *writeSpinReq* requests cause the write-spin problem while the *nonWriteSpinReq* requests do not. During runtime phase, HybridNetty maintains a map object which keeps a record of category for each request. Thus when a new request comes, HybridNetty checks the category of the request in the map object, and then determines the most efficient event processing flow to process the request (see check req type in Fig. 15). Concretely, HybridNetty will choose the NettyServer execution path to process each *writeSpinReq* request to avoid the write-spin problem and the SingleT-Async execution

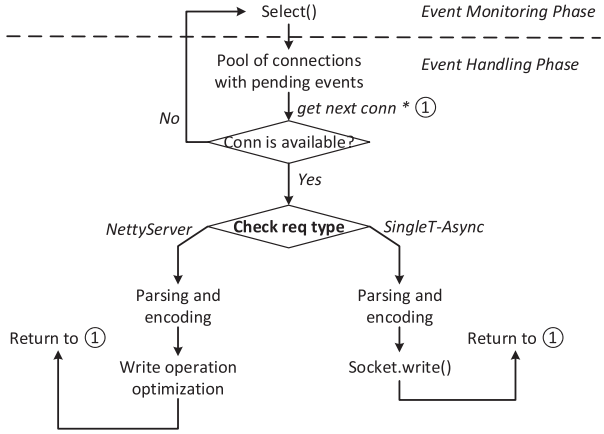


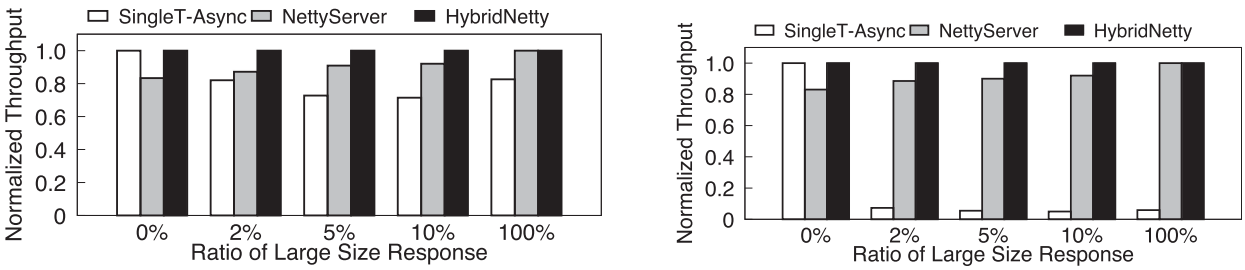
Fig. 15. Illustration of the worker thread processing flow in Hybrid solution.

path to process each *nonWriteSpinReq* request to avoid the overhead caused by the write operation optimization. We note the server response size even for the same request could change over time due to the changes of system state (e.g., the dataset has changed). In this case, the map object will be updated once HybridNetty detects a request has been categorized into a wrong class, thus HybridNetty is able to keep the latest category of each request for future efficient processing. Since our hybrid solution passively profile each incoming request, it cannot completely solve the write-spin problem. The write-spin problem occurs first, and then it is fixed by dynamically choosing the most efficient execution path according to different the request category. Thus the frequency of the write-spin problem is dramatically reduced.

There are two potential extensions of HybridNetty to further improve its performance. The first one is an alternative non-blocking I/O pattern, which blocks the write-spin worker thread using select, poll, or epoll and enables the reactor thread to poll for completed I/O operations. Such a pattern is able to provide potential performance improvement of HybridNetty by completely avoiding the write-spin problem. The second one is that the reactor thread can be removed in HybridNetty, and the limited number of the worker threads can compete for a shared spin-lock to call the system call socket.accept() and accept new connections. Such a extension can further remove the context switch overhead between the main reactor thread and other worker threads, especially under certain workload that the main reactor thread needs to frequently hand over new established connections to worker threads. However, such two extensions of HybridNetty need more investigation, and we will make it as our future work.

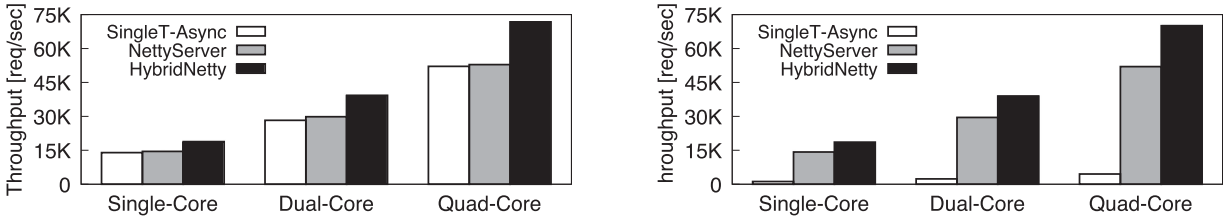
5.3 Experimental Validation

Validation in a Single-Core Environment. We validate the efficiency of our hybrid solution compared to SingleT-Async and NettyServer under various runtime workload conditions and network latencies in Fig. 16. The workload is composed of two categories of requests: the heavy requests and the light requests. These two categories of request differ in the size of the server response; heavy requests, due to their large response size (100KB), are able to trigger the write-spin



(a) No network latency between client and server (b) ~5ms network latency between client and server

Fig. 16. Our hybrid solution achieves the best performance under various mixes of workload and network latencies. We set the workload concurrency to 100 and the TCP send buffer size to the default 16KB. We compare the normalized throughput among the three types of servers and use `HybridNetty` as the baseline.



(a) No network latency between client and server (b) ~5ms network latency between client and server

Fig. 17. Our hybrid solution still performs the best in a multi-core environment. We keep the workload concurrency to be 100; the workload consists of 2 percent heavy and 98 percent light requests. Both (a) and (b) shows that `HybridNetty` performs the best from one-core to quad-core, with or without network latency.

problem while light requests (0.1KB) can not.[2] To simulate different realistic workload scenarios, we vary the ratio of heavy requests from 0 and 100 percent. To clearly show the effectiveness of our hybrid solution, we adopt the normalized throughput comparison and use the `HybridNetty` as the baseline.

Validation in a Multi-Core Environment. We also validate the effectiveness of our hybrid solution in a multi-core environment in Fig. 17. The workload concurrency is 100 and the TCP send buffer size is 16KB by default. In this set of experiments, the workload is composed of 2 percent heavy requests and 98 percent light requests, which follows Facebook's Memcached workload report [46]. To perform a fair comparison, we adopt the *N*-copy model [28] to enable three servers to take advantage of multiple cores in Fig. 17. The figure shows that the maximum achievable throughput of each type of servers scales as the number of cores increases; `HybridNetty` outperforms the other two in all scenarios. For example, Fig. 17b shows that `HybridNetty` performs almost 10X higher than `SingleT-Async` and 35 percent higher than `NettyServer` in throughput respectively in a quad-core environment when the network latency is 5ms, suggesting the effectiveness of our solution in resolving the context switch problem and the write-spin problem without significant write operation optimization overhead.

6 Related Work

Synchronous Thread-Based Server Designs for High Concurrency Support. Many previous research efforts in this category [4], [47], [48], [49] share a similar goal: achieving the same or even better performance compared to the corresponding asynchronous event-driven counterparts. For instance, Behren et al. [47] presents a scalable user-space thread library Capriccio [48] and demonstrates that the threads can provide all the benefits of the events but with a simpler and more natural programming model. They further show a Capriccio-based synchronous server Knot is able to outperform SEDA's event-driven server Haboob [3] under high concurrency workload (up to tens of thousands of concurrent clients). However, Krohn et al. [4] show that the thread library Capriccio uses sophisticated stack management to mimic the event handling to the underlying operating system. In addition, the authors of Capriccio also notice that the thread interface still lacks flexibility compared to the events [47]. These research efforts imply that the asynchronous event-driven architecture still plays an important role in constructing high performance and high-efficiency Internet servers.

There are plenty of research works to show that asynchronous event-driven architecture has been considered as a superior alternative to the thread-based design for high performance systems [50], [51], [52]. For example, Cheng et al. adopt asynchronous design in an I/O efficient graph system to solve problems of poor I/O locality, efficient selective scheduling, and expensive synchronization cost. The optimizations for asynchronous event-driven servers can be further divided into two broad categories.

OS-Level Optimization for Asynchronous Event-Driven Servers. Research work in this category is mainly motivated by mitigating the unnecessary system calls (e.g., event notification mechanisms such as *select*, *poll*, and *epoll*) and the associated CPU overhead [53], [54], [55] when facing high concurrency workload. For example, Lever et al. [47] present a high-performance in-kernel web server TUX, which eliminates the user/kernel crossings overhead by delegating both event monitoring and event handling to the kernel thread. Han et al. [54] present a scalable and

2. Each TCP connection has private send buffer [45], thus the write-spin problem is only caused by connections with the heavy requests.

efficient network I/O named MegaPipe, to lighten application layer socket-related system calls for message-oriented workloads.

Configurations Tuning for Asynchronous Event-Driven Servers. Previous work concludes that an asynchronous web server needs to be well-tuned for the best performance [16], [50], [56], [57], [58], [59], [60], [61]. For example, Brecht et al. [56] study the impact of connection-accepting strategies (either aggressive or passive) on the performance of the asynchronous event-driven μServer. Pariag et al. [16] analyze the impact of maximum simultaneous connections and different socket I/O (either blocking or non-blocking) on the performance of different asynchronous architectures such as the single-threaded μServer and the staged design WatPipe. Google's research group [57] reports that increasing TCP's initial congestion window is able to significantly improve average HTTP response latency in high latency and low bandwidth networks. Our work in the paper is closely related to their research. Previous research efforts focus on either OS-level optimization or configurations tuning for asynchronous event-driven servers. Our approach focuses more on optimizing server architecture, thus our work and their work are complementary. The lessons that we learned from their work may also apply to our proposed solution.

7 Conclusions

In this paper, we show the impact of the event processing flow on the efficiency of asynchronous servers. Through extensive experiments using both realistic macro- and micro-benchmarks, we observe that the inefficient design of the event processing flow in an asynchronous server may cause high CPU overhead and result in significant performance loss in comparison with the thread-based counterpart. Concretely, the inefficient design of event processing flow may either cause high CPU context switch overhead between event handlers (see Section 3) or the write-spin problem when dealing with large size of server responses (see Section 4). Some network-related factors (e.g., network latency and client receive buffer) will exaggerate the degradation of asynchronous server performance. We present a hybrid solution which exploits the merits of different asynchronous event-driven architectures to adapt to dynamic runtime workload and network conditions (see Section 5). In general, our research results provide a solid building block in developing modern Internet servers that can achieve both high performance and high resource efficiency in the cloud.

Acknowledgments

This research has been partially funded by National Science Foundation by CISEs CNS (1566443), Louisiana Board of Regents under grant LEQSF(2015-18)-RD-A-11, and gifts or grants from Fujitsu.

References

[1] R. Hashemian, D. Krishnamurthy, M. Arlitt, and N. Carlsson, "Improving the scalability of a multi-core web server," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 161–172.

[2] Q. Wang, Y. Kanemasa, J. Li, C.-A. Lai, C.-A. Cho, Y. Nomura, and C. Pu, "Lightning in the cloud: A study of very short bottlenecks on n-tier web application performance," in *Proc. USENIX Conf. Timely Results Operating Syst.*, 2014.

[3] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services," in *Proc. 18th ACM Symp. Operating Syst. Principles*, 2001, pp. 230–243.

[4] M. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense," in *Proc. USENIX Ann. Tech. Conf.* , 2007, pp. 7:1–7:14.

[5] *RUBBoS: Bulletin board benchmark*. Feb. 2005. [Online]. Available: http://jmob.ow2.org/rubbos.html

[6] *Jetty: A Java HTTP (Web) Server and Java Servlet Container*. Aug. 2019. [Online]. Available: http://www.eclipse.org/jetty/

[7] *Oracle GlassFish Server*. Aug. 2019. [Online]. Available: http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html

[8] *MongoDB Async Java Driver*. Aug. 2019. [Online]. Available: http://mongodb.github.io/mongo-java-driver/3.5/driver-async/

[9] *nginx: a high performance HTTP and reverse proxy server, as well as a mail proxy server*. Aug. 2019. [Online]. Available: https://nginx.org/en/

[10] *lighttpd*. Aug. 2019. [Online]. Available: https://www.lighttpd.net/

[11] *Node.js*. Aug. 2019. [Online]. Available: https://nodejs.org/en/

[12] *Netty*. Aug. 2019. [Online]. Available: http://netty.io/

[13] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[14] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *Proc. 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 207–217.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Annu. Tech. Conf.*, 2010, pp. 11–11.

[16] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, "Comparing the performance of web server architectures," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 3, pp. 231–243, 2007.

[17] C. Li, K. Shen, and A. E. Papathanasiou, "Competitive prefetching for concurrent sequential i/o," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, pp. 189–202.

[18] S. Adler, "The slashdot effect: An analysis of three internet publications," *Linux Gazette*, vol. 38, 1999, Art. no. 2.

[19] *RUBiS: Rice University Bidding System*. Oct. 2009. [Online]. Available: http://rubis.ow2.org/

[20] *TPC-W: A Transactional Web e-Commerce Benchmark*. Aug. 2019. [Online]. Available: http://www.tpc.org/tpcw/

[21] O. A.-H. Hassan and B. A. Shargabi, "A scalable and efficient web 2.0 reader platform for mashups," *Int. J. Web Eng. Technol.*, vol. 7, no. 4, pp. 358–380, Dec. 2012.

[22] *Collectl*. Oct. 2018. [Online]. Available: http://collectl.sourceforge.net/

[23] *Apache JMeter*™. Aug. 2019. [Online]. Available: http://jmeter.apache.org

[24] *sTomcat-NIO, sTomcat-BIO, and two alternative asynchronous servers*. Mar. 2018. [Online]. Available: https://github.com/sgzhang/AsynMessaging

[25] *perf*. Jun. 2018. [Online]. Available: http://www.brendangregg.com/perf.html

[26] B. Veal and A. Foong, "Performance scalability of a multi-core web server," in *Proc. 3rd ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, 2007, pp. 57–66.

[27] F. Gaud, S. Geneves, R. Lachaize, B. Lepers, F. Mottet, G. Muller, and V. Quéma, "Efficient workstealing for multicore event-driven systems," in *Proc. 30th Int. Conf. Distrib. Comput. Syst.*, 2010, pp. 516–525.

[28] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazieres, and M. F. Kaashoek, "Multiprocessor support for event-driven programs." in *Proc. USENIX Annu. Tech. Conf.*, 2003, pp. 239–252.

[29] A. S. Harji, P. A. Buhr, and T. Brecht, "Comparing high-performance multi-core web-server architectures," in *Proc. 5th Annu. Int. Syst. Storage Conf.*, 2012, pp. 1:1–1:12.

[30] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, Netherlands: Elsevier, 2011.

[31] J. Lewis and M. Fowler, "Microservices: A definition of this new architectural term," Oct. 2018. [Online]. Available: https://martinfowler.com/articles/microservices.html

[32] *JProfiler: The award-winning all-in-one Java profiler*. Aug. 2019. [Online]. Available: https://www.ej-technologies.com/products/jprofiler/overview.html

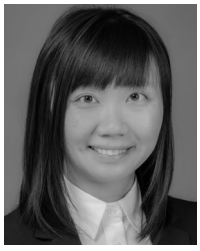
[33] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 33–46.
[34] *Java docs: Class SocketChannel*. Aug. 2019. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/nio/channels/SocketChann el.html#write(java.nio.ByteBuffer)
[35] *Linux Programmer's Manual: SEND(2)*. Aug. 2019. [Online]. Available: http://man7.org/linux/man-pages/man2/send.2.html
[36] M. Belshe, R. Peon, and M. Thomson, "Hypertext transfer protocol version 2 (HTTP/2)," IETF RFC 7540, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7540
[37] B. Constantine, G. Forget, R. Geib, and R. Schrage, "Framework for TCP throughput testing," IETF RFC 6349, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6349.txt
[38] M. Fisk and W.-c. Feng, "Dynamic right-sizing in tcp," 2001, Art. no. 2. [Online]. Available: http://lib-www.lanl.gov/la-pubs/00796247.pdf
[39] M. Allman, V. Paxson, and W. Stevens, "TCP congestion control," IETF RFC 2581, Apr. 1999. [Online]. Available: https://tools.ietf.org/html/rfc2581
[40] *Android*. Aug. 2019. [Online]. Available: https://www.android.com/
[41] *Documentations for Android*. Aug. 2019. [Online]. Available: https://developer.android.com/reference/java/net/SocketOptions/
[42] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al., "Scaling memcache at facebook," in *Proc. 10th USENIX Conf. Networked Syst. Des. Implementation*, 2013, vol. 13, pp. 385–398.
[43] *Redis*. Aug. 2019. [Online]. Available: https://redis.io/
[44] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 167–181.
[45] *The TCP Send Buffer, In-Depth*. [Online]. Available: https://devcentral.f5.com/s/articles/the-send-buffer-in-depth-21845
[46] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. ACM SIGMETRICS Perform. Eval. Rev.*, 2012, vol. 40, no. 1, 2012, pp. 53–64.
[47] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proc. 9th Conf. Hot Top. Operating Syst.*, 2003, pp. 4–4.
[48] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 268–281.
[49] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for oldi microservices," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 177–194.
[50] S. Cheng, G. Zhang, J. Shu, and W. Zheng, "Asyncstripe: I/o efficient asynchronous graph computing on a single server," in *Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2016, Art. no. 32.
[51] M. García-Valdez and J. Merelo, "evospace-js: Asynchronous pool-based execution of heterogeneous metaheuristics," in *Proc. Genetic Evol. Comput. Conf. Companion*, 2017, pp. 1202–1208.
[52] Z. Wang, L. Li, Y. Xu, H. Tian, and S. Cui, "Handover optimization via asynchronous multi-user deep reinforcement learning," in *Proc. Int. Conf. Commun.*, 2018, pp. 1–6.
[53] C. Lever, M. Eriksen, and S. Molloy, "An analysis of the TUX web server," University of Michigan, CITI Tech. Rep. 00-8, Nov. 2000.
[54] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 135–148.
[55] M. Gallo and R. Laufer, "Clicknf: A modular stack for custom network functions," in *Proc. USENIX Annu. Tech. Conf.*, 2018.
[56] T. Brecht, D. Pariag, and L. Gammo, "Acceptable strategies for improving web server performance," in *Proc. Annual Conf. USENIX Annu. Tech. Conf.*, 2004, pp. 20–20.
[57] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing tcp's initial congestion window," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 3, pp. 26–33, Jun. 2010.
[58] S. S. Prakash and B. C. Kovoor, "Performance optimisation of web applications using in-memory caching and asynchronous job queues," in *Proc. Int. Conf. Inventive Comput. Technol.*, 2016, vol. 3, pp. 1–5.
[59] A. Aytekin, H. R. Feyzmahdavian, and M. Johansson, "Analysis and implementation of an asynchronous optimization algorithm for the parameter server," *arXiv preprint*, 2016. [Online]. Available: https://arxiv.org/abs/1610.05507
[60] J. Davis, A. Thekumparampil, and D. Lee, "Node.fz: Fuzzing the server-side event-driven architecture," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 145–160.
[61] J. Davis, G. Kildow, and D. Lee, "The case of the poisoned event handler: Weaknesses in the node.js event-driven architecture," in *Proc. 10th Eur. Workshop Syst. Secur.*, 2017, pp. 8:1–8:6.

Shungeng Zhang received the BS degree from HuaZhong University of Science & Technology in China, in 2014. He is working toward the PhD degree in the Department of EECS, Louisiana State University-Baton Rouge. He is currently working in the cloud computing lab as a research assistant. His research interest include performance and scalability analysis of internet server architecture, aiming to achieve responsive web applications running in the cloud. He is a student member of the IEEE.

Qingyang Wang received the BSc and MSc degrees from Chinese Academy of Sciences and Wuhan University, in 2004 and 2007 and the PhD degree in computer science from Georgia Tech, in 2014. He is an assistant professor with the Department of EECS, Louisiana State University-Baton Rouge. His research is in distributed systems and cloud computing with a current focus on performance and scalability analysis of large-scale web applications (e.g., Amazon.com). He has led research projects with LSU on cloud performance measurements, scalable web application design, and automated system management in clouds. He is a member of the IEEE.

Yasuhiko Kanemasa received the BEng degree in computer engineering from Tokyo Institute of Technology, Tokyo, Japan, in 1996, and the MS degree in computer science from Japan Advanced Institute of Science and Technology, Nomi, Japan, in 1998. He has been working with Fujitsu Laboratories Ltd., Kawasaki, Japan since 1998 and is in the position of research manager currently. His research interests include data processing systems, application performance management, and cloud computing. He is a member of the IEEE, IPSJ, and DBSJ.

Huasong Shan received the PhD degree in computer engineering from Louisiana State University-Baton Rouge, in 2017. He received the MS and BS degree in computer science and technology, Huazhong University of Science and Technology, China, in 2003 and 2006, respectively. He has been working with JD.com American Technologies Corporation, Mountain View, California as staff scientist. His research interests include distributed system, application of AI on system and security, cloud computing, storage system etc. He is a student member of the IEEE.



Liting Hu received the BSc degree in computer science from Huazhong University of Science and Technology, in China and the PhD degree in computer science from Georgia Institute of Technology. Her research is in the general area of distributed systems and its intersection with big data analytics, resource management, power management, and system virtualization. She interned with IBM T.J. Watson Research Center, Intel Science and Technology Center for Cloud Computing, Microsoft Research Asia, VMware, and has been working closely with them. She is a member of the IEEE.