Threads vs Events for Server Architectures

VAARNAN DROLIA, National University of Singapore
CEDRIC ANSLEY CHIN SHEN WANG, National University of Singapore

This paper surveys the different arguments favouring a multi-threaded server versus using an event-driven server architecture. It explores the wide breadth of arguments made by both camps and presents the results in a unified and cohesive way to ascertain the major differences between the two approaches and the relative advantages and disadvantages of each approach.

Categories and Subject Descriptors: [**Computer systems organization**]: Client-server architectures

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Threads, Events, Server Architectures, Thread-Event Duality

1. INTRODUCTION

The Internet has grown to become an integral part of modern life. Services on the Internet have thus experienced massive growth: Facebook, for instance, reports that at peak times they are required to serve 1,000,000 images per second [Beaver et al. 2010]. The vast majority of such websites and other Internet services are provided by *web servers* - a software component designed and deployed for providing services and content over the web.

Modern web servers deployed today face a unique problem: how to provide reliable, responsive Internet systems at scale to millions of users. This problem differs from the challenges that server architectures of the past had to grapple with, primarily because the Internet has reached a scale previously unimaginable by server architects.

1.1. Challenges

The core goal of modern server architectures is to provide *performance at scale*. This means web servers must face and solve the following challenges:

(1) High concurrency - web servers should be able to handle many thousands of connections from web users at the same time.
(2) High throughput - web servers should be responsive: that is, they need to handle large amounts of client requests in a timely manner.
(3) Large variances in server load - due to the nature of the web, web servers may expect huge bursts of user activity, as a site suddenly becomes popular (e.g. the 'Slashdot Effect' [Adler 1999]).

The requirements for web servers are complicated by the fact that commodity operating systems and traditional concurrency models weren't created with these requirements in mind. For instance, traditional concurrency programming involves the use of processes and threads, which are designed with high context-switching costs and memory usage. Such an approach is not ideal for a modern, massively concurrent web server.

As such, most modern server architectures have relied on one of two approaches to achieve the above requirements: threads and events.

1.2. Threads

Threads, in the context of web servers, refer to the strategy of using concurrently-executing threads that each handle one request. An example of a threaded web server is Apache, which spawns new threads to handle each client request (or picks a thread out of a worker thread pool). Naturally, there is a limit to the number of threads a web server may run at any one time; in addition to this, there are high context switching costs associated with spawning or forking a new thread for every request, especially when compared to event-driven servers.

With that said, the threaded model is conceptually easier to understand[1]: programming with threads is essentially expressing linear control flow, like any other sequentially-executing program.

1.3. Events

Events, on the other hand, refer to the strategy of running an event loop in a single thread that handles all client requests as a series of events. No new threads are spawned to handle new client requests, so event-based servers tend to have lower overheads as compared to threaded servers.

However, writing applications in the event-driven paradigm requires doing what is known as 'stack ripping'. As there is only one event loop, it is key to ensure each request does not take up too much time (or resources) in said loop, so as to ensure timely processing of all pending events. Blocking operations are thus handed off to other specialized worker threads, requiring the programmer to manually 'rip' each potentially blocking function into two parts: a call function and a return function. The call function runs in another thread and handles the blocking operation, after which it uses an event mechanism to ensure that the return function be re-entered into the event loop.

This technique obfuscates control flow, complicates memory management in event-driven applications, and demands that the programmer write more code.

1.4. Performance Measurement

Related to the core challenge of web server design (i.e. providing performance at scale) is the need to quantify this performance. Traditional methods of measuring server performance have included throughput, service time, server latency, I/O performance and memory management performance.

However, the nature of modern websites is such that clients surfing the web frequently demand large, complex collections of objects pushed down from the server to the clients. For instance, a user visiting Facebook would need to download the Facebook page, associated style sheets, images, and JavaScript scripts. Those scripts may then request additional components (images and HTML files) from the server, and may block some of the other object downloads. In addition, such objects may be served from different web servers, or from different locations in the Facebook server cluster.

Server concurrency, throughput and load flexibility are thus not the only determining factors for a server's performance. A user's experience of web server performance are also determined by these client-specific factors. Our survey will spend some time examining the different performance measurements from both a server-side and client-side perspective.

2. THREADS VS EVENTS

The literature on server architecture is filled with debates on the pros and cons between threads and events in pursuit of the goal of high throughput, high concurrency and flexible load handling. This debate is not new - we may trace it back to research in the operating systems field. In 1979 [Lauer and Needham 1979] made what is known as the duality argument: that events-based and threaded systems are duals, and are equivalent in both program structure and - assuming good implementations - perfor-

[1]This does not mean threads are easy to use. Programming with threads is notorious for the difficulty associated with synchronization.

mance. However, they made this observation for operating systems, and considered procedure-oriented systems versus message-oriented systems. These systems do not map perfectly to modern thread-based and event-based implementations, but subsequent papers [von Behren et al. 2003a] in the literature have accepted the high-level concepts (in particular, the idea that event-based systems and threaded systems are logically equivalent, and any such system may be converted from one model to the other) and have thus moved on to examine implementation details and characteristics of each approach as opposed to examining the approaches themselves.

An important point to note is that in 2002 [Adya et al. 2002], made the observation that a concurrent program's stack management is orthogonal to its task management. A program can employ manual or automatic stack management, and cooperative or preemptive task management, and make these choices independent from each other. This is an observation that will be used by some of the proposed solutions to the problems faced by threads and events.

We shall examine several problems with threads and events, before examining various proposed solutions.

2.1. Problems with Threads

In brief, here is the case against threads:

— **It is very difficult to write correct concurrent code using threads.** The most famous expression of this argument is John Ousterhout's 1996 presentation [Ousterhout 1996], in which he argues implementing good coordination and synchronization for shared state between multiple threads is too difficult for even expert programmers to do.

— **Threads are not performant.** Threads require large amounts of memory due to thread stacks. We have previously discussed the cost associated with context-switching; in addition, preemption and scheduling may not be optimal if we rely on generic thread management provided to us by the kernel. Lastly, synchronized primitives such as mutexes add additional overhead.

— **Threads provide an oversimplifying abstraction.** While threads are conceptually easier to understand (due to providing procedural semantics to the programmer) actual runtime behaviour of threads are nondeterministic due to interleaving, and do not map to such procedural semantics. This makes it difficult to reason about concurrent code, and tricks the programmer into believing his concurrent code executes in a linear fashion, when it really doesn't.

Proponents for threads, on the other hand, argue that threads are still easier to understand, as they express linear control flow. They are also well-understood elements of operating system design, and cannot be escaped if one is to leverage true CPU concurrency.

2.2. Proposed Solutions for Threads

Many of the solutions for the problems with threads deal with fixing the perceived performance weaknesses of threads, which historically has been the main argument against thread-based servers. [von Behren et al. 2003a] argue that most of the weaknesses of threads are the result of poor thread package implementations. Fix the implementations until performance is comparable with events, they posit, and threads win out because of simpler semantics. They demonstrate this with a couple of core techniques:

2.2.1. Fix basic thread operations. In their May 2003 paper, *Why Events Are A Bad Idea*, von Behren et al argued that most thread libraries were not designed for high concurrency and block operations, thus making them ill-suited for use in modern web servers. In particular, these libraries had *O(n)* operations where n is the number of threads, and had relatively high context switching costs due to preemption and kernel crossings. As a proof of concept, they rewrote and removed most of the *O(n)* operations from the GNU Pth user-level and found that the library scaled quite well up to 100,000 threads, matching SEDA, an event-based server [Welsh et al. 2001].

2.2.2. Cooperative multitasking instead of preemptive multitasking. As mentioned in 2.2.1, the high context switching costs are due to two reasons: (a) preemption, and (b) kernel crossings for kernel threads. (The solution to (b) is avoid using kernel threads, which is detailed in section 2.2.3 below.)

To resolve (a), von Behren et al rely on the observation of [Adya et al. 2002] that the advantage of cheap synchronization (usually described as 'for free') normally conferred to event-based systems is really due to cooperative task management, instead of an intrinsic property of the event-based model. von Behren et al point out that using cooperative thread systems would thus confer the same advantages to threaded systems.

In Capriccio, a thread library that extends many of the ideas in their 2003 paper, von Behren et al implement this idea by using lightweight inter-thread synchronization primitives. For user-space threads, the Capriccio system checks a boolean locked/unlocked flag, and does these checks in a very lightweight manner due to the fact that the thread library is implemented in user-space. This means neither the user threads nor the scheduler can be interrupted in a critical section. In the case of multiple kernel threads (which is allowed alongside Capriccio), Capriccio uses either spin locks or optimistic concurrency control primitives, depending on suitability.

A significant criticism of this approach is that cooperative task management only works for uniprocessor systems (this ties back to the argument that the event model cannot take advantage of true CPU concurrency). For multiprocessor systems, the performance advantage of lightweight synchronization is removed because heavyweight synchronization primitives such as locks and mutexes must be used in the face of true CPU concurrency. von Behren et al's response to this is to argue that tighter integration between compiler and runtime system would alert the programmer to probable data-races [von Behren et al. 2003a] making it easier to use such synchronization primitives. To address the performance hit, they point out that in the vein of TinyOS, a compiler may use a call graph (similar to Lauer and Needham's blocking graph) to identify atomic sections that are safe to execute concurrently. This information may then be passed on to the runtime to allow safe execution on multiple processors. Unfortunately this argument was not yet implemented in Capriccio, their later system.

2.2.3. User-level threads instead of kernel threads. To prevent Capriccio from incurring the costs of kernel crossings identified in their earlier paper [von Behren et al. 2003a] made the decision to use user-level threads instead [von Behren et al. 2003b]. This technique conferred the following benefits:

(1) **No kernel crossings.** Kernel crossings are expensive. For instance, a kernel-level mutex contention often requires a kernel crossing for every synchronization operation, something that is not required if a mutex contention happens in user space. (Note: this is only applicable in the case of preemptive scheduling, whereas Capriccio uses cooperative scheduling, which provides even more performance advantages).
(2) **Extremely lightweight.** User-level threads are lightweight, allowing programmers to use a tremendous number of threads without worry for overhead.

(3) **More efficient memory management.** Kernel threads demand data structures that take up valuable kernel address space, decreasing space available for other kernel resources.
(4) **Increases flexibility of the thread scheduler.** Using user-level threads grants the programmer the option of writing a user-level thread scheduler to be built alongside the application. This allows tailoring of the scheduling algorithm to the demands of the application (e.g. for a web server). This is in contrast to kernel threads, which use a generic kernel-level scheduler that must be fair to all applications. In fact, Capriccio uses a resource aware scheduler to further improve performance (covered in section 2.2.5).

2.2.4. Dynamic Stack Growth. A common criticism with using threads is that thread stacks are relatively ineffective as compared to event-based systems for managing live state. In threaded systems, thread stacks are typically over-allocated or under-allocated, thus either risking stack overflow or causing wasted virtual address space. Event servers don't face this problem as they unwind the thread stack after each event handler, and force the programmer to manually maintain live state.

One way of solving this is to use a mechanism to allow the size of the stack to be adjusted at runtime. In Capriccio, [von Behren et al. 2003b] developed a technique called linked stack management. The technique calls for the compiler to first analyse the amount of stack space that must be preallocated. This is done using a weighted call graph, where each function is represented by a node in the graph weighted by the maximum stack space that a single stack frame for that function would consume. Because this approach may result in an overly-conservative amount of stack space being preallocated, the technique also calls for the stack to grow or shrink according to runtime requirements.

This dynamic allocation is achieved by inserting checkpoints into certain call sites, identified from the initial call graph analysis. These checkpoints are small pieces of code that check if there is enough stack space left to reach the next call site without causing stack overflow. If a stack overflow seems likely, the checkpoint causes a new stack chunk to be allocated, and the stack pointer is changed to point to the new stack chunk.

As a result of this technique, the Capriccio paper [von Behren et al. 2003b] reports increased memory performance, as the system no longer needs to preallocate unnecessarily large stacks. This in turn reduces virtual memory pressure when running large amounts of threads. In addition, Capriccio experiences significantly improved paging behaviour, as stack chunks are reused in LIFO order, allowing these chunks to be shared between threads and reducing the size of the working set.

2.2.5. Resource Aware Scheduler. Lastly, von Behren et al took full advantage of using user-level threads (and therefore the possibility of writing their own thread scheduler) by implementing what they call a resource-aware scheduler. First a blocking graph is created which contains information about the places in the program where threads block. Each node in the graph is a location in the program that blocks, and edges are drawn between two nodes if they are consecutive blocking points. Then, the scheduler is designed to:

(1) Keep track of resource utilization levels in order to determine if each resource is at its limit.
(2) In the blocking graph, annotate each node with resources used on its outgoing edges in order to predict the impact on each resource would be if the thread from that node were to be rescheduled.
(3) Dynamically prioritize threads for scheduling based on (1) and (2).

It is interesting to note that von Behren et al's approach tackles the performance problem with threads, but ignores the other two arguments against threads (difficult to write concurrent programs with threads as compared to the simple parallel model of events, and how threads oversimplify the concurrent model). Their approach to this argument is to simply say that complex control flows are difficult regardless of model, and therefore not their problem to handle [von Behren et al. 2003a].

2.3. Problems with Events

A brief case against events are as follows:

— **The programming model for events is overly-complicated.** The chief argument against event-based systems is that programming for such systems demand 'stack ripping', more commonly known as 'callback hell'. As described earlier, such programming demands that the programmer split each potentially blocking function into two parts, thus resulting in highly fragmented code. Worse, complicated systems often require the programmer to write long, cascading callback chains. This is significantly more complex as compared to the simple semantics of threads.
— **Manual management of state.** A side-effect of the 'stack ripping' model of programming is the necessity for programmers to save and restore state between call and return function pairs. On the one hand, this may be considered an advantage as manual management saves the overhead of using a thread stack and allows the programmer to provide application-specific scheduling; on the other hand, it adds an additional burden on the programmer.
— **Difficulty of taking advantage of real CPU concurrency.** In making the rant against threads, Ousterhout concedes [Ousterhout 1996] that threads should be used when true CPU concurrency is needed, as the model fits perfectly for that use-case. Subsequently, event-driven systems cannot easily take advantage of such CPU concurrency or make use of multiple cores (given that each event loop runs in one thread).

2.4. Proposed Solutions for Events

The proposed solutions for events focus on fixing the problems associated with its programming model. These problems include overly-complicated control flow as a result of stack ripping, and the manual state management required by the events paradigm.

2.4.1. Semantics for events programming. The key idea for the Tame system, introduced by [Krohn et al. 2007], is that events programming can be made manageable if one changes the syntax used[2]. Tame is implemented as a source-to-source translation library in C++, and affords the programmer better expressivity for events programming. It also offers "automatic stack management (like threads) and explicit cooperative task management (like events)." [Krohn et al. 2007]

2.4.2. Events semantics for expressivity. Tame intends to make it easy for a programmer to express events as easy as it is to express threads. As such, it provides four abstractions for handling concurrency:

— *events* - which are future event occurrences.
— *wait points* - which are points in the program that blocks the calling function until events are triggered.
— *rendezvous objects* - to which one wait point and several events may be attached
— *safe local variables* - which is Tame's way of providing variables that are preserved across wait points.

[2]One may think of this as similar to syntactic sugar, but Tame does more than that.

These semantics combine to allow programmer to express event-related control flow without resorting to stack ripping.

2.4.3. Automatic Stack Management. Another common criticism of the events model is that the programmer is forced to maintain state as callback functions are popped from and returned to the stack. Tame seeks to solve this by using closures - a language feature borrowed from functional languages [Krohn et al. 2007], while still allowing the programmer to write ostensibly event-driven code.

The way this is done is that a Tamed function's closure lives until a) control exits the function for the last time or b) at least until events created in the function have been triggered. In addition, events associated with a rendezvous object must trigger exactly once before the rendezvous object is deallocated.

To perform such automatic stack management, Tame uses a reference-counting scheme. The Tame runtime keeps tracks of events and closures and de-allocates objects as they are no longer used.

Krohn et al's system thus allows the programmer the performance properties of an events system without the programing overhead of manual state management.

3. COMPARISON OF THREADS AND EVENTS

As noted in the previous sections, both threads and events have notable benefits as well as drawbacks with substantial parallel tracks of research for us to make a comparative analysis of the thread solutions versus the event solutions in the context of high concurrency server architectures.

In the following sections, we we analyse and summarize the various issues and compare and contrast thread and event based approaches.

3.1. Live State Management

As mentioned earlier, thread have large overheads of automatic stack management and garbage collection while events come with the bane of manual stack management and 'stack ripping'.

For threads, Capriccio manages the stack inefficiencies by using the linked-stack approach, mentioned earlier which allows for dynamic stack allocation. This keeps the approach portable and applicable to programs adopting it's threading library.

Tame deals with these problems by using compiler-generated syntactic sugar which are defined as safe local variables in tvars and they function as closures to avoid the stack ripping problem.

3.2. High Concurrency

The performance of the user-level thread implementation Capriccio against SEDA's Haboob web server and Apache with and without Capriccio is given in Figure 1.

There is an improvement in the supported bandwidth for Apache when using Capriccio by almost 15% and the Knot server has comparable performance to Haboob.

Though the SEDA paper had extensive benchmarks against threading-based servers such as Apache, unfortunately, the Capriccio paper did not conduct the same experiments which leads to a gap in our understanding of their relative performance.

From Table 2, we can see that the same Knot server, in Capriccio and Tame, generates similar throughput but Tame Knot uses ˜30% of the physical memory and ˜20% of the virtual memory used by Capriccio Knot.

Thus, the overheads of threading are very evident in this context due to the high amounts of stack allocation for the 350 threads.

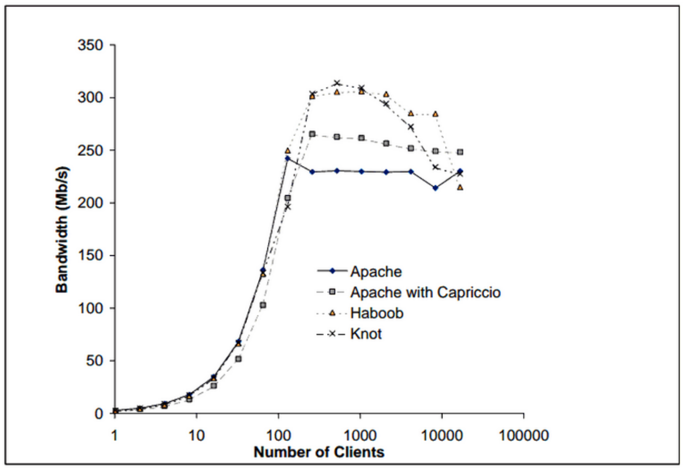
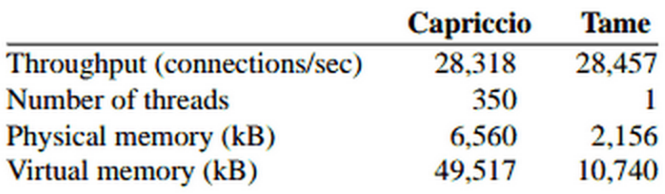


Fig. 1. Figure from [von Behren et al. 2003b]



	Capriccio	Tame
Throughput (connections/sec)	28,318	28,457
Number of threads	350	1
Physical memory (kB)	6,560	2,156
Virtual memory (kB)	49,517	10,740

Fig. 2. Table from [Krohn et al. 2007]

3.3. Flexible Load Handling

Traditional thread-based servers such as Apache are not able to do dynamic resource handling because they allocate one thread to each process and do not distinguish between different requests. This implies that the server cannot degrade "gracefully" in processing large numbers of requests during peak loads.

SEDA adopts a unique use of dynamic resource controllers which work at runtime and can be implemented for any stage. These are highly configurable and allow the allocated resources and scheduling to be handled at runtime. The controllers observe the position of tasks and can prioritise tasks to eventually decrease the load on the system. Also, the length of the task-queues give a lot of information of the server load which can be used for dynamic resource adjustments.

Capriccio, on the other hand, adopts resource aware scheduling where the different blocking points in the program are assumed to be the stage separators. It uses a dynamic blocking graph to automatically detect the nature of the program and make appropriate scheduling decisions. There were a few pitfalls to this approach such as not being able to detect/prevent thrashing.

Thus, SEDA offered more fine-grained control to the programmer as well as manage and prevent thrashing. Capriccio, on the other hand, decreased the scheduling complexity for the programmer at the cost of overheads for the blocking call graph which were 8% of the execution time for Apache and 36% of the execution time for Knot [von Behren et al. 2003b].

3.4. Source Code Expressiveness

The paper on Capriccio comments on the simplicity of the Knot server which is only 1290 lines of C code written in 3 days [von Behren et al. 2003b].

In contrast, the Haboob runtime contains 676 lines of non-commented Java code not including the Sandstorm runtime which contains 7871 non-commented lines of Java code.

A Tame Knot server was implemented by the authors in [Krohn et al. 2007] but there is no discussion on the complexity of that server against the Capriccio Knot server. Another points mentions though that the students at MIT who used Tame to implement an NFS server, wrote 20% less code in their source files, and 50% less code in their header files [Krohn et al. 2007].

The problem with Tame is that the solution requires rewriting of large amounts of existing event code while Capriccio does not have the same problem.

4. PERFORMANCE/WORKLOAD MEASUREMENTS

As mentioned in the introduction, the performance benchmarks used by papers, [Welsh et al. 2001; von Behren et al. 2003b; von Behren et al. 2003a; Krohn et al. 2007] have metrics such as throughput, bandwidth but these are only server-side measurements.

In the context of Internet Services, there is a combination of both dynamically generated content and statically generated content. Loading a full web page involves loading several other elements such as CSS, JavaScript files, and even images. There are several dependencies here such as images which cannot load until certain JavaScript files have loaded, synchronous loading of some resources, etc. Optimizing for high Page Load Times (PLTs) becomes much more complex than running servers which have high data throughput since content is also often served by various different servers including Content Distribution Networks (CDNs).

This makes performance measurements much more complex and hard to do solely on the server side than the traditional measurements such as file-servers which can only be done on the server side. Some of the papers even assume short-lived connections instead of long-running connections which are typical in browser-driven applications where “Keep-Alive” connections are used and user feedback time needs to be taken into account.

The WebProphet paper [Li et al. 2010] proposes a solution to measuring mainly page load times on the client-side and adopts an interesting and unique approach to do so.

It identifies three delays, server computation, communication delay and user page rendering delay. WebProphet consists of three components, a Dependency Extractor, Measurement Engine and Performance Predictor.

The Dependency Extractor constructs the dependency graph for the page by choosing objects of the page and delaying their load time such that objects which depend on that object will be delayed even further. This involves repeatedly loading the page while choosing a different object set to be delayed each time.

Finally, a small set of objects are selected using a greedy approach so that companies can determine which parts of the page should be optimized for the maximum decrease in the PLT.

The Measurement Engine is a set of web agents which are used to load the page and measure the different load times of the page [Li et al. 2010].

The Performance Predictor bundles a trace analyser and page simulator to take the various delays into account and predict the updated PLTs and generate a simulation of the page load [Li et al. 2010].

These are novel solutions to optimize the user experience and help website owners solve the ever growing complexity of web pages so that they can work on optimizing the correct bottlenecks instead of having to over-provision resources and suffering from sub-optimum utilization.

5. OPINIONS

In general, we observe that the papers surveyed here focus on solutions to the problems along two main themes, divided by the paradigm they support. Proponents of threads focus on solutions to improve performance of threading systems, and proponents of events focus on increased expressivity to ensure easier programming for the events paradigm.

5.1. Pipelined Server Architectures

The SEDA paper is one of the most cited papers amongst discussions for server architectures. We observe that SEDA was one of the first few server architectures designed to tackle the problem of large load variances head-on. In addition, its pipeline approach was novel, as were its ideas about dynamic resource controllers, ability to degrade gracefully and explicit event queues. We found the idea that a server should have dynamic resource allocation attractive, especially when compared to the resource containment philosophy of previous papers.

That being said, with the era of low cost computing and large server farms, having a single server implement several stages internally seems to be unnecessary. Rather, the model where multiple web servers each individually perform a few functions (acting as pipelines in a larger cluster) is more prevalent in real world deployments. A good example of this is Twitter's server architecture, which consists of a myriad of services spread out in its server farms.

5.2. Reliance on compiler optimizations

The approaches suggested in Tame involve a rewrite of current event-based server architectures to reap the performance benefits which might not be practical for most servers which have had a considerable amount of closed or open-sourced community investment.

Better threading support on the other hand is much simpler to adapt to, though there is the problem of portability issues related to threads.

5.3. Capriccio's Threading Support

The current implementation of Capriccio is based on using a single processor which handicaps it's ability to be benchmarked in a multiprocessor environment. [Pariag et al. 2007] benchmarks the Capriccio Knot server against server and WatPipe, which have event-driven and hybrid-pipelined architectures respectively, and finds that the thread implementation in Capriccio relies on an event-driven foundation which introduces overheads in the order of 10% of the available CPU time causing the Knot server to perform poorly. This is contrary to the result in [von Behren et al. 2003b] and requires further work.

Also, reimplementing Capriccio in a multiprocessor environment might bring additional inefficiencies and is an area that warrants further work.

In recent times, there are more efficient threading libraries such as the Native POSIX Thread Library for Linux which might be offering better concurrency.

5.4. State of Performance Measurements

WebProphet is a novel approach to client-side benchmarking which had not been considered earlier. There are some drawbacks in it's approach though, since it treats the browsers as a black-box [Wang et al. 2013] and does not take into account different load times for the critical path depending upon the dependency policy of the browser.

This makes the performance measures inaccurate in real browsers where the PLTs actually matter. Also, WebProphet only focuses on network activity and it's delays, while the results in the paper on WProf [Wang et al. 2013] show that computation involving HTML parsing and JavaScript execution comprise 35% of the critical path. Thus, we can see there is still a lot of work left to be done in being able to decrease the Page Load Times for the user.

6. FUTURE WORK

The threading techniques used so far have only focused on single-processor architectures and have not been extended to support the average 2-6 cores found in general computers today. This is definitely a potential area to explore to determine whether concurrent programming becomes even more difficult in the context of multiprocessor systems.

The emergence of newer concurrency constructs in languages such as Go which have "goroutines" functioning as light-weight threads open up new areas where threading-based servers might be able to compete more effectively with their event based counterparts, while maintaining simpler semantics.

The growing popularity of cloud computing also puts factors such as automatic on-demand provisioning and de-provisioning of resources into a much greater perspective where the load might need to be scaled across heterogeneous systems to meet demand. The simplified single server architecture might not work in these emerging scenarios.

REFERENCES

Stephen Adler. 1999. The Slashdot Effect, An Analysis of Three Internet Publications. *Linux Gazette* 59 (March 1999). http://linuxgazette.net/issue38/adler1.html.

Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, Berkeley, CA, USA, 289–302. http://dl.acm.org/citation.cfm?id=647057.713851

Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 1–8. http://dl.acm.org/citation.cfm?id=1924943.1924947

Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. 2007. Events Can Make Sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 7, 14 pages. http://dl.acm.org/citation.cfm?id=1364385.1364392

Hugh C. Lauer and Roger M. Needham. 1979. On the Duality of Operating System Structures. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 3–19. DOI:http://dx.doi.org/10.1145/850657.850658

Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. 2010. WebProphet: Automating Performance Prediction for Web Services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1855711.1855721

John Ousterhout. 1996. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference* 5 (1996).

David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. 2007. Comparing the Performance of Web Server Architectures. *SIGOPS Oper. Syst. Rev.* 41, 3 (March 2007), 231–243. DOI:http://dx.doi.org/10.1145/1272998.1273021

Rob von Behren, Jeremy Condit, and Eric Brewer. 2003a. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=1251054.1251058

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003b. Capriccio: Scalable Threads for Internet Services. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 268–281. DOI:http://dx.doi.org/10.1145/1165389.945471

Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 473–486. http://dl.acm.org/citation.cfm?id=2482626.2482671

Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 230–243. DOI:http://dx.doi.org/10.1145/502059.502057