Why Events Are A Bad Idea (for high-concurrency servers)

Rob von Behren, Jeremy Condit and Eric Brewer
Computer Science Division
University of California, Berkeley
{jrvb, jcondit, brewer}@cs.berkeley.edu

Abstract

Event-based programming has been highly touted in recent years as the best way to write highly concurrent applications. Having worked on several of these systems, we now believe this approach to be a mistake. Specifically, we believe that threads can achieve all of the strengths of events, including support for high concurrency, low overhead, and a simple concurrency model. Moreover, we argue that threads allow a simpler and more natural programming style.

We examine the claimed strengths of events over threads and show that the weaknesses of threads are artifacts of specific threading implementations and not inherent to the threading paradigm. As evidence, we present a user-level thread package that scales to 100,000 threads and achieves excellent performance in a web server. We also refine the duality argument of Lauer-Needham, which implies that good implementations of thread systems and event systems will have similar performance. Finally, we argue that compiler support for thread systems is a fruitful area for future research. It is a mistake to attempt high concurrency without help from the compiler, and we discuss several enhancements that are enabled by relatively simple compiler changes.

1 Introduction

Highly concurrent applications such as Internet servers and transaction processing databases present a number of challenges to application designers. First, handling large numbers of concurrent tasks requires the use of scalable data structures. Second, these systems typically operate near maximum capacity, which creates resource contention and high sensitivity to scheduling decisions; overload must be handled with care to avoid thrashing. Finally, race conditions and subtle corner cases are common, which makes debugging and code maintenance difficult.

Threaded servers have historically failed to meet these challenges, leading many researchers to conclude that event-based programming is the best (or even only) way to achieve high performance in highly concurrent applications. The literature gives four primary arguments for the supremacy of events:

- Inexpensive synchronization due to cooperative multitasking;
- Lower overhead for managing state (no stacks);
- Better scheduling and locality, based on application-level information; and
- More flexible control flow (not just call/return).

We have made extensive use of events in several high-concurrency environments, including Ninja [13], SEDA [14], and Inktomi's Traffic Server. In working with these systems, we realized that the properties above are not restricted to event systems; many have already been implemented with threads, and the rest are possible.

Ultimately, our experience led us to conclude that event-based programming is the wrong choice for highly concurrent systems. We believe that (1) threads provide a more natural abstraction for high-concurrency servers, and that (2) small improvements to compilers and thread runtime systems can eliminate the historical reasons to use events. Additionally, threads are more amenable to compiler-based enhancements; we believe the right paradigm for highly concurrent applications is a thread package with better compiler support.

Section 2 compares event systems and thread systems, and it offers a rebuttal of the common arguments against threads. Next, Section 3 explains why threads are particularly natural for writing high-concurrency servers. Section 4 explores the value of a compiler support for threads. In Section 5, we validate our approach with a simple web server. Finally, Section 6 covers (some) related work, and Section 7 concludes.

2 Threads vs. Events

The debate between threads and events is a very old one. Lauer and Needham attempted to end the discussion in 1978 by showing that message-passing systems and process-based systems are duals, both in terms of program structure and performance characteristics [7]. Nonetheless, numerous papers have been

Events	Threads
event handlers	monitors
events accepted by a handler	functions exported by a module
SendMessage / AwaitReply	procedure call, or fork/join
SendReply	return from procedure
waiting for messages	waiting on condition variables

Figure 1: A selection of dual notions in thread and event systems, paraphrased from Lauer and Needham. We have converted their terminology to contemporary terms from event-driven systems.

written in recent years declaring the need for event-driven programming for highly concurrent systems [8, 9, 14].

2.1 Duality Revisited

To understand the threads and events debate, it is useful to reexamine the duality arguments of Lauer and Needham. Lauer and Needham describe canonical threaded and message-passing (i.e., event-based) systems. Then, they provide a mapping between the concepts of the two regimes (paraphrased in Figure 1) and make the case that with proper implementations, the performance should be equivalent. Finally, they argue that the decision comes down to which paradigm is more natural for the target application; in the case of high-concurrency servers, we believe the thread-based approach is preferable.

The message-passing systems described by Lauer and Needham do not correspond precisely to modern event systems in their full generality. First, Lauer and Needham ignore the cooperative scheduling used by events for synchronization. Second, most event systems use shared memory and global data structures, which are described as atypical for Lauer-Needham's message-passing systems. In fact, the only event system that really matches their canonical message-passing system is SEDA [14], whose stages and queues map exactly to processes and message ports.[1]

Finally, the performance equivalence claimed by Lauer-Needham requires equally good implementations; we don't believe there has been a suitable threads implementation for very high concurrency. We demonstrate one in the next section, and we discuss further enhancements in Section 4.

In arguing that performance should be equivalent, Lauer and Needham implicitly use a graph that we call a *blocking graph*. This graph describes the flow of control through an application with respect to blocking or yielding points. Each node in this graph represents a blocking or yielding point, and each edge represents the code that is executed between two such points. The Lauer-Needham duality argument essentially says that duals have the same graph.

[1] Arguably, one of SEDA's contributions was to return event-driven systems to the "good practices" of Lauer-Needham.

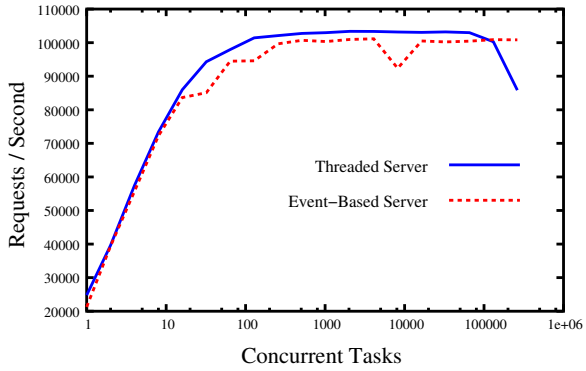

Figure 2: A repeat of the threaded server benchmark from the SEDA paper [14]. The threaded server uses a preallocated thread pool to process requests, while the event server uses a single thread to pull items from the queue. Requests are internally generated to avoid network effects. Each request consists of an 8K read from a cached disk file.

The duality argument suggests that criticisms of thread performance and usability in recent years have been motivated by problems with *specific* threading packages, rather than with threads in general. We examine the most common criticisms below.

2.2 "Problems" with Threads

Performance. *Criticism: Many attempts to use threads for high concurrency have not performed well.* We don't dispute this criticism; rather, we believe it is an artifact of poor thread implementations, at least with respect to high concurrency. None of the currently available thread packages were designed for the combination of high concurrency and blocking operations, and thus it is not surprising that they perform poorly.

A major source of overhead is the presence of operations that are $O(n)$ in the number of threads. Another common problem with thread packages is their relatively high context switch overhead when compared with events. This overhead is due to both preemption, which requires saving registers and other state during context switches, and additional kernel crossings (in the case of kernel threads).

However, these shortcomings are not intrinsic properties of threads. To illustrate this fact, we repeated the SEDA threaded server benchmark [14] with a modified version of the GNU Pth user-level threading package, which we optimized to remove most of the $O(n)$ operations from the scheduler. The results are shown in Figure 2. Our optimized version of Pth scales quite well up to $100,000$ threads, easily matching the performance of the event-based server.

Control Flow. *Criticism: Threads have restrictive control flow.* One argument against threaded programming is that it encourages the programmer to think too linearly about control flow, potentially precluding the use of more efficient control flow patterns. However, complicated control flow patterns are rare in practice. We have examined the code structure of the Flash web server and of several applications in Ninja, SEDA, and TinyOS [5, 9, 13, 14]. In all cases, the control flow patterns used fit three simple cases: call/return, parallel calls, and pipelines. All of these patterns can be expressed more naturally with threads.

We believe more complex patterns are not used because they are difficult to use well. The accidental non-linearities that often occur in event systems are already hard to understand, leading to subtle races and other errors. Intentionally complicated control flow is equally likely to be error prone.

The only patterns we considered that are less graceful with threads are dynamic fan-in and fan-out; such patterns might occur with multicast or publish/subscribe applications. In these cases, events are probably more natural. However, none of the high-concurrency servers that we studied used these patterns.

In retrospect, there is reason to expect that the common patterns should map well onto the call/return mechanism provided by threads. Robust systems tend to need acknowledgements both for error handling and for lifetime management; thus, they need a "return" even in the event model.

Synchronization. *Criticism: Thread synchronization mechanisms are too heavyweight.* Event systems often claim as an advantage that cooperative multitasking gives them synchronization "for free," since the runtime system does not need to provide mutexes, handle wait queues, and so on [8]. However, Adya *et al.* [1] show that this advantage is really due to cooperative multitasking (i.e., no preemption), not events themselves; thus, cooperative thread systems can reap the same benefits. It is important to note that in either regime, cooperative multitasking only provides "free" synchronization on uniprocessors, whereas many high-concurrency servers run on multiprocessors. We discuss compiler techniques for supporting multiprocessors in Section 4.3.

State Management. *Criticism: Thread stacks are an ineffective way to manage live state.* Threaded systems typically face a tradeoff between risking stack overflow and wasting virtual address space on large stacks. Since event systems typically use few threads and unwind the thread stack after each event handler, they avoid this problem. To solve this problem in threaded servers, we propose a mechanism that will enable dynamic stack growth; we will discuss this solution in Section 4.

Furthermore, event systems encourage programmers to minimize live state at blocking points. In a thread system, live state is stored on the stack when a blocking point is reached. However, an event system would require the programmer to manage the same state by hand, which tends to somewhat reduce the overall amount of live state maintained by the program. This problem can also be solved with compiler techniques that we will discuss in Section 4.

Scheduling. *Criticism: The virtual processor model provided by threads forces the runtime system to be too generic and prevents it from making optimal scheduling decisions.* Event systems are capable of scheduling event deliveries at application level. Hence, the application can perform shortest remaining completion time scheduling, favor certain request streams, or perform other optimizations. There has also been some evidence that events allow better code locality by running several of the same kind of event in a row [6]. However, Lauer-Needham duality indicates that we can apply the same scheduling tricks to cooperatively scheduled threads.

2.3 Summary

The above arguments show that threads can perform at least as well as events for high concurrency, and that there are no substantial qualitative advantages to events. The absence of scalable user-level threads has provided the largest push toward the event style, but we have shown that this deficiency is an artifact of the available implementations rather than a fundamental property of the thread abstraction.

3 The Case for Threads

Up to this point, we have largely argued that threads and events are equivalent in power and that threads can in fact perform well with high concurrency. In this section, we argue that threads are actually a more appropriate abstraction for high-concurrency servers. This conclusion is based on two observations about modern servers. First, the concurrency in modern servers results from concurrent requests that are largely independent. Second, the code that handles each request is usually sequential. We believe that threads provide a better programming abstraction for servers with these two properties.

Control Flow. For these high-concurrency systems, event-based programming tends to obfuscate the control flow of the application. For instance, many event systems "call" a method in another module by sending an event and expect a "return" from that method via a similar event mechanism. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. Furthermore, these call/return pairs often require the programmer to manually save and restore live state. This

process, referred to as "stack ripping" by Adya *et al.* [1], is a major burden for programmers who wish to use event systems. Finally, this obfuscation of the program's control flow can also lead to subtle race conditions and logic errors due to unexpected message arrivals.

Thread systems allow programmers to express control flow and encapsulate state in a much more natural manner. Syntactically, thread systems group calls with returns, making it much easier to understand cause/effect relationships, and ensuring a one-to-one relationship. Similarly, the run-time call stack encapsulates all live state for a task, making existing debugging tools quite effective.

Exception Handling and State Lifetime. Cleaning up task state after exceptions and after normal termination is simpler in a threaded system, since the thread stack naturally tracks the live state for that task. In event systems, task state is typically heap allocated. Freeing this state at the correct time can be extremely difficult because branches in the application's control flow (especially in the case of error conditions) can cause deallocation steps to be missed.

Many event systems, such as Ninja and SEDA, use garbage collection to solve this problem. However, previous work has found that Java's general-purpose garbage collection mechanism is inappropriate for high-performance systems [11]. Inktomi's Traffic Server used reference counting to manage the state, but it was an ongoing battle to keep the counts correct, particularly for error handling.[2]

Existing Systems. The preference for threads is subtly visible even in existing event-driven systems. For example, our own Ninja system [13] ended up using threads for the most complex parts, such as recovery, simply because it was nearly impossible to get correct behavior using events (which we tried first). In addition, applications that didn't need high concurrency were always written with threads, just because it was simpler. Similarly, the FTP server in Harvest uses threads [2].

Just Fix Events? One could argue that instead of switching to thread systems, we should build tools or languages that address the problems with event systems (i.e., reply matching, live state management, and shared state management). However, such tools would effectively duplicate the syntax and run-time behavior of threads. As a case in point, the cooperative task management technique described by Adya *et al.* [1] allows users of an event system to write thread-like code that gets transformed into continuations around blocking calls. In many cases, fixing the problems with events is tantamount to switching to threads.

[2]Nearly every release battled with slow memory leaks due to this kind of reference counting; such leaks are often the limiting factor for the MTBF of the server.

4 Compiler Support for Threads

Tighter integration between compilers and runtime systems is an extremely powerful concept for systems design. Threaded systems can achieve improved safety and performance with only minor modifications to existing compilers and runtime systems. We describe a number of ways this synergy can be used both to overcome limitations in current threads packages and to improve safety, programmer productivity, and performance.

4.1 Dynamic Stack Growth

We are developing a mechanism that allows the size of the stack to be adjusted at run time. This approach avoids the tradeoff between potential overflow and wasted space associated with fixed-size stacks. Using a compiler analysis, we can provide an upper bound on the amount of stack space needed when calling each function; furthermore, we can determine which call sites may require stack growth. Recursive functions and function pointers produce additional challenges, but these problems can be addressed with further analyses.

4.2 Live State Management

Compilers could easily purge unnecessary state from the stack before making function calls. For example, temporary variables could be popped before subroutines are called, and the entire frame could be popped in the case of a tail call. Variables with overlapping lifetimes could be automatically reordered or moved off the stack in order to prevent live variables from unnecessarily pinning dead ones in memory. The compiler could also warn the programmer of cases where large amounts of state might be held across a blocking call, allowing the programmer to modify the algorithms if space is an issue.

4.3 Synchronization

Compile-time analysis can reduce the occurrence of bugs by warning the programmer about data races. Although static detection of race conditions is challenging, there has been recent progress due to compiler improvements and tractable whole-program analyses. In nesC [4], a language for networked sensors based on the TinyOS architecture [5], there is support for atomic sections, and the compiler understands the concurrency model. TinyOS uses a mixture of events and run-to-completion threads, and the compiler uses a variation of a call graph that is similar to the blocking graph. The compiler ensures that atomic sections reside within one edge on that graph; in particular, calls within an atomic section cannot yield or block (even indirectly). Compiler analysis can also help determine which atomic sections are safe to run concurrently. This information can then be given to the runtime system to allow safe execution on

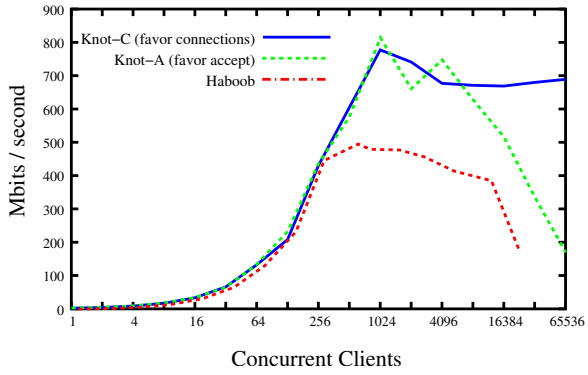

Figure 3: Web server bandwidth versus the number of simultaneous clients. We were unable to run the benchmark for Haboob with more than 16384 clients, as Haboob ran out of memory.

multiprocessors, thus automating the hand-coded graph coloring technique used in libasync [3].

5 Evaluation

To evaluate the ability of threads to support high concurrency, we designed and implemented a simple (5000 line) user-level cooperative threading package for Linux. Our thread package uses the `coro` coroutine library [12] for minimalist context switching, and it translates blocking I/O requests to asynchronous requests internally. For asynchronous socket I/O, we use the UNIX `poll()` system call, whereas asynchronous disk I/O is provided by a thread pool that performs blocking I/O operations. The library also overrides blocking system calls and provides a simple emulation of pthreads, which allows applications written for our library to compile unmodified with standard pthreads.

On top of this thread package, we wrote a simple 700-line test web server, Knot. Knot accepts static data requests, allows persistent connections, and includes a straightforward page cache. The code is written in a clear, straightforward threaded style, and required very little performance tuning.

We compared the performance of Knot to that of SEDA's event-driven web server, Haboob, using the test suite used to evaluate SEDA [14]. The `/dev/poll` patch used for the original Haboob tests has been deprecated, so our tests of Haboob used standard UNIX `poll()` (as does Knot). The test machine was a 2x2000 MHz Xeon SMP with 1 GB of RAM running Linux 2.4.20. The test uses a small workload, so there is little disk activity. We ran Haboob with the 1.4 JVM from IBM, with the JIT enabled. Figure 3 presents the results.

We tested two different scheduling policies for Knot, one that favors processing of active connections over accepting new ones (Knot-C in the figure) and one that does the reverse (Knot-A). The first policy provides a natural throttling mechanism by limiting the number of new connections when the server is saturated with requests. The second policy was designed to create higher internal concurrency, and it more closely matches the policy used by Haboob.

Figure 3 shows that Knot and Haboob have the same general performance pattern. Initially, there is a linear increase in bandwidth as the number of simultaneous connections increases; when the server is saturated, the bandwidth levels out. The performance degradation for both Knot-A and Haboob is due to the poor scalability of `poll()`. Using the newer `sys_epoll` system call with Knot avoids this problem and achieves excellent scalability. However, we have used the `poll()` result for comparison, since `sys_epoll` is incompatible with Haboob's socket library. This result shows that a well-designed thread package can achieve the same scaling behavior as a well-designed event system.

The steady-state bandwidth achieved by Knot-C is nearly 700 Mbit/s. At this rate, the server is apparently limited by interrupt processing overhead in the kernel. We believe the performance spike around 1024 clients is due to lower interrupt overhead when fewer connections to the server are being created.

Haboob's maximum bandwidth of 500 Mbit/s is significantly lower than Knot's, because Haboob becomes CPU limited at 512 clients. There are several possible reasons for this result. First, Haboob's thread-pool-per-handler model requires context switches whenever events pass from one handler to another. This requirement causes Haboob to context switch 30,000 times per second when fully loaded—more than 6 times as frequently as Knot. Second, the proliferation of small modules in Haboob and SEDA (a natural outgrowth of the event programming model) creates a large number of module crossings and queuing operations. Third, Haboob creates many temporary objects and relies heavily on garbage collection. These challenges seem deeply tied to the event model; the simpler threaded style of Knot avoids these problems and allows for more efficient execution.

6 Related Work

Ousterhout [8] made the case in favor of events, but his arguments do not conflict with ours. He argued that programming with concurrency is fundamentally difficult; thus, he concludes, events are preferable (for most purposes) because they allow programmers to avoid concurrency whenever possible. We believe that it is difficult to truly avoid dealing with concurrency in our

target space of highly concurrent servers, and furthermore, we believe that threads are a more appropriate programming model for managing concurrency in these servers. Also, we argue that cooperative threads can provide many of the same benefits as events, as pointed out by Adya *et al.* [1].

There have been many user-level threads packages, such as Filaments [10] and NT's Fibers. Both of these packages use cooperative scheduling, but neither package is targeted at large numbers of blocking threads.

Adya *et al.* [1] cover a subset of these issues better than we have. They identify the value of cooperative scheduling for threads, and they define the term "stack ripping" for management of live state. We explore thread performance and compiler support in addition.

We have looked at many event-driven systems. Inktomi's Traffic Server web proxy uses events for high concurrency, and it is the most complicated event-driven system that we have seen, although Harvest [2] is close. Both of these systems are difficult to modify and debug. Flash [9] and Ninja [13] are a little cleaner in that they better mix threads and events; Flash uses helper processes for disk I/O, and Ninja uses threads in some of the more complex areas, including the recovery code.

SEDA is a hybrid approach between events and threads, using events between stages and threads within them [14]. This approach is quite similar to the message-passing model discussed by Lauer-Needham [7], though Lauer and Needham advocate a single thread per stage in order to avoid synchronization within a stage. SEDA showed the value of keeping the server in its operating range, which it did by using explicit queues; we agree that the various queues for threads *should* be visible, as they enable better debugging and scheduling. In addition, the stage boundaries of SEDA provide a form of modularity that simplifies composition in the case of pipelines. When call/return patterns are used, such boundaries require stack ripping, and are better implemented with threads using blocking calls.

7 Conclusions

Although event systems have been used to obtain good performance in high concurrency systems, we have shown that similar or even higher performance can be achieved with threads. Moreover, the simpler programming model and wealth of compiler analyses that threaded systems afford gives threads an important advantage over events when writing highly concurrent servers. In the future, we advocate tight integration between the compiler and the thread system, which will result in a programming model that offers a clean and simple interface to the programmer while achieving superior performance.

Acknowledgements

We would like to thank George Necula and Feng Zhou for their helpful contributions. We would also like to thank the Berkeley Millennium group for loaning us the hardware for the benchmarks in this paper. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix ATC*, June 2002.

[2] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, January 1996.

[3] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th ACM SIGOPS European Workshop*, September 2002.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[6] J. Larus and M. Parkes. Using cohort scheduling to enhance server performance. Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.

[7] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Second International Symposium on Operating Systems, IR1A*, October 1978.

[8] J. K. Ousterhout. Why Threads Are A Bad Idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[9] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[10] W. Pang and S. D. Goodwin. An algorithm for solving constraint-satisfaction problems.

[11] M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein. Java support for data-intensive systems: Experiences building the Telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, 2001.

[12] E. Toernig. Coroutine library source. http://www.goron.de/˜froese/coro/.

[13] J. R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, , and D. Culler. Ninja: A framework for network services. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[14] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.