

Why Events Are A Bad Idea (for high-conc. Servers)

Rob von Behren, Jeremy Condit and Eric Brewer
Proceedings of HotOS III the Ninth Workshop on Hot Topics in Operating Systems, 2003, 19-24

Presented by Jean-Sébastien Légaré

Using events for highly concurrent applications is a mistake

- Threads can achieve all the strength of events
- Threads allow a simpler style of programming than events
- The claimed weaknesses of threads are artefacts of the current thread implementations

CPSC508 2006 Winter Term 1 Jean-Sébastien Légaré 2

Duality of the 2 models

- The “Events versus Threads” debate has been going on for ages.
- The event based and thread based models have been shown to be duals of each other. [LAUER,1978]
 - Using primitives of one model or the other should yield equivalent performance
 - Event-handlers ⇔ monitors
 - Accepted events ⇔ functions exported by a module
 - SendMessage/AwaitReply ⇔ procedure call / fork,join
 - SendReply ⇔ return
 - Waiting for messages ⇔ waiting on condition variables

CPSC508 2006 Winter Term 1 Jean-Sébastien Légaré 3

Highly concurrent applications

- Must use scalable data structures
- Must operate at maximum capacity
- Present many race conditions
 - Programs hard to debug
 - Code hard to maintain

Led many researchers to conclude that event-based programming is the best.

CPSC508 2006 Winter Term 1 Jean-Sébastien Légaré 4

Claimed benefits of events (in literature)

- Inexpensive synchronization due to cooperative multitasking
- Very little stack space required
- Better scheduling and locality, based on application-level information
- More flexible control flow

Also achievable with threads!
(given proper support from the threading implementation)

CPSC508 2006 Winter Term 1 Jean-Sébastien Légaré 5

Criticism regarding threads...

- *Many attempts to use threads for high concurrency have not performed well*
- *Threads have restrictive control flow*
- *Thread synchronization is too heavyweight*
- *Thread stacks are an ineffective way to manage live state*
- *Optimal scheduling cannot be attained with threads*

CPSC508 2006 Winter Term 1 Jean-Sébastien Légaré 6

“Many attempts to use threads for high concurrency have not performed well”

- Due only to the implementations
 - Presence of O(n) operations in schedulers
 - High context switch overhead due to preemption
- Solution
 - Use a user-level thread library
 - Remove/Replace the expensive operations
 - Created a modified version of the GNU pth package that scales up to 100 000 threads [PTH]

SEDA server benchmark repeated

- In this version, one thread is created for each task in the pipeline

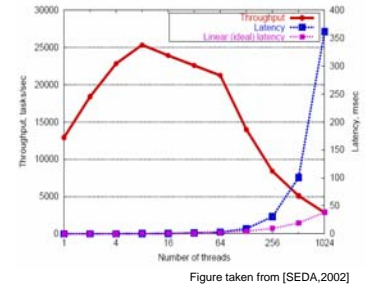
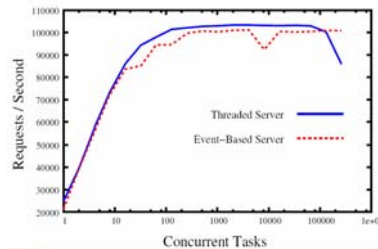


Figure taken from [SEDA,2002]

SEDA benchmark revisited

- The threaded version matches the performance of the event based version



“Threads have restricted control flow”

- Complicated patterns are rare in practice
 - Usually only: call/return, parallel calls, pipelines
- Complex patterns are difficult to use well
 - Hard to understand or error prone
- Only very few patterns are less graceful with threads
 - Dynamic fan-in and fan-out
 - High-concurrency servers use only simple patterns

“Thread synchronization mechanisms are too heavyweight”

- Event systems claim they get synchronization for free.
 - Due to cooperative multitasking and not event themselves [Aya,2002]
- Thread based systems can reap the same benefits
- In either system, free synchronization only comes with uniprocessors

“Thread stacks are an ineffective way to manage live state”

- Large stack space is used
 - Risk overflowing
 - Risk wasting memory
 - Provides automatic state management
 - In contrast, event based systems must manage state by hand at blocking points
- Solution
 - Minimize live state using compiler features
 - Dynamic stack growth

Proposed compiler improvements

- Minimizing live state
 - By popping unused variables before making calls
 - By reorganizing live and dead variables
- Using compiler analysis to determine bounds on the stack size needed
 - Dynamic stack growth
- Determining data races
 - Could reduce synchronization costs on multiprocessor machines

“Optimal scheduling cannot be attained with threads”

- Event system are capable of scheduling event deliveries at application level
- Events allow better code locality
- Same tricks can be applied to cooperatively scheduled threads [LAUER,1978]

Threads are more appropriate for high concurrency servers

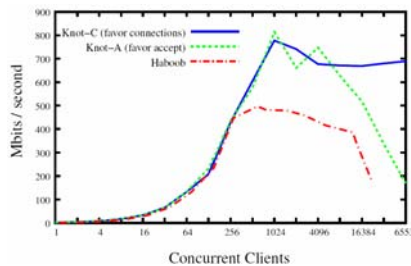
- Event-based programming tends to obfuscate the control flow of the application
 - Manually saving and restoring state makes programs complex (“stack ripping” [ADYA,2002])
- Threads easier to debug
- Stack just needs to be popped to clear state
- Existing event-driven systems use threads for some components
 - Ninja [Ninja,2002] uses them for recovery
 - Adya et Al. [ADYA,2002] uses thread like code to build continuations around blocking calls in cooperative task management

Evaluating performance of threads in presence of high concurrency

- Designed a 5000 line cooperative user-level threading package
 - Used a coroutine library for minimalist context switching
 - Translates blocking IO requests into asynchronous requests using poll() and a thread pool
- Used that package to write a 700 line web server called Knot
- Compared 2 versions of Knot to Haboob, the SEDA web server [SEDA,2001]

Knot compared to Haboob

- Knot A and Haboob drop because of poll()
- Performance spike due to lower overhead
- Haboob switches context 6 times more than Knot
- Garbage collection affects Haboob



Conclusion

- Event systems have been used to obtain good performance in high concurrency systems **but...**
- Threads can achieve similar or higher performance
- Simpler programming model
- Threads can afford help from the compiler

Weaknesses of the paper

- The claim that event-driven applications exhibit a subtle preference for thread based applications is based on experience their own event-based application, Ninja.
- The complete feasibility of all their compiler improvement suggestions is contestable. Especially when it comes to compiler analyses in C.
- The difference in size of code bases for Knot and Haboob should account in the performance evaluations.

References

[ADYA,2002] 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.

[LAUER,1978] H. C.Lauer, R. M.Needham. On the Duality of Operating System Structures. In Second International Symposium on Operating Systems, IR1A, October 1978.

[Ninja,2002] 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.

[PTH] GNU Pth – The GNU Portable Threads. <http://www.gnu.org/software/pth>

[SEDA,2001] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for well-conditioned, scalable Internet services. In Symposium on Operating System Principles, pages 230-243, 2001.

Questions

- If the problems with threads are not "intrinsic" to the model (as the authors suggest) then why do the same problems seem to be so pervasive?
- Knot was tested only in terms of bandwidth. Although the score was impressive, what does this imply in the measure of fairness?
- Knot has a very small codebase. What does this mean in relation to the more sophisticated SEDA model in terms of the fairness of comparisons and the relative merits of the two approaches?

Questions

- The authors said the only patterns are less graceful with threads are dynamic fan-in and fan-out. I want to know the details of the reason for it. Could you explain it ?
- Do you think the scheme provided in this paper has enable threads to be with the 4 points mentioned in the first part ? I think this paper want to solve part of the problem by use of programming language, such as, compiler and dynamic stack. This perhaps could solve some problems. However, it perhaps could not solve most of the problems.

Questions

- We have now seen a paper where a group of people who are very experienced with threads have maligned the use of threads for concurrency, and a paper where a group of people who are very experienced with events have done the same to the event-based model. Could it be that concurrent programming is just inherently difficult, and that these expositions are merely a manifestation of familiarity breeding contempt?
- The 'compiler support' that the authors speak of consists entirely of static analysis: That is, the compiler should signal to the programmer when they've done something silly. What about adding extra constructs to the language itself to support threaded programming? Could that not be equally useful?
- In section 2, the authors invoke the Lauer-Needham duality to 'prove' that certain components of the event-driven model could be implemented in the threading model. (See especially 2.2 under the heading "Scheduling"). However, no discussion is given about the actual practicality of such an implementation. Is this simply a theoretical equivalence, or is it actually the case that the implementation could be done easily and robustly?

Questions

- Which operations are $O(n)$ in the number of threads?
- What is Haboob's strength, compared to Knot? (How about resource management?)

Questions

- What are the $O(n)$ operations in the scheduler that are removed to lessen overhead of threads? Are they vital?
- What is the maximum latency when Knot is used in comparison to that of Haboob? How is fairness and graceful degradation be ensured here?

Questions

- von Behren mentions that threads are better because "... The concurrency in modern servers results from concurrent requests that are largely independent." (Section 3) I would suspect quite the opposite, do you think requests are "largely independent?"
- It seems the author does a lot of hand waving, particularly about compiler optimizations for threads (for instance: "... these problems can be addressed with further analyses" Section 4.1), but is there any evidence to support these claims?

Questions

- They say that they translate blocking I/O requests to asynchronous requests, but that for disk I/O these are implemented with blocking calls performed by a thread pool. Isn't the effect that disk I/O is still a blocking call that the thread must wait on? Why even bother with the thread pool?
- Doesn't cooperative multitasking generally imply sub-optimal performance, at least in the case where there are more threads than processors, because the programmer almost never knows exactly where a thread should yield? It seems likely that you'll either have an over-abundance of context switches or else threads will have to wait for long periods while other threads use the processors.

Questions

- What is the benefit of "stack ripping"?
- What does cooperative multitasking mean for events?
- Ousterhout vs. the authors of "Events Are A Bad Idea" - in a battle to the death, who will come out victorious?
- Do you think that the events vs. threads all comes down to which ever paradigm the programmer is most familiar with?
- Suppose that the threads really work as well as the paper states. How would you cook up a set of benchmarks that would favour event based systems (without leaving the arena of high-concurrency servers)?
- Notice that the subtitle of this paper indicates that this matters mostly for "high-concurrency servers". Does the argument of the paper apply to: modern computer games? embedded systems for space shuttle? embedded systems for set top boxes? firmware for your home office router? word processor and traditional applications?

Questions

- What are the $O(n)$ operations in the scheduler that are removed to lessen overhead of threads? Are they vital?
- What is the maximum latency when Knot is used in comparison to that of Haboob? How is fairness and graceful degradation be ensured here?

Questions

- How do you think the authors could make an estimate on the amount of stack space needed by a recursive function call in order to have a dynamic allocation of stack space? Would you happen to know of any solutions?
- The authors mention that "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". Do you find this a good suggestion to programmers working on big and convoluted projects?
- Do you agree with their opinion that today's thread based systems are badly implemented so they perform poorly when both high concurrency and blocking operations are present?

Questions

- Why do threads present a better sense of abstraction for high-concurrency servers than events?
- What are the improvements proposed in the compiler support for threads?

Questions

- The performance of threads are closely dependent on the thread library and the compiler as well as the underlying operating system. Is the performance of an event-based system more predictable across different operating systems and compilers?
- The paper mentioned that they created a threading package and a web server that outperforms SEDA. It seems that they have made a specific library for a narrow range of applications, do you think that their argument/threading library can be applied to a wider range of applications?

Questions

- In section 3, Exception Handling and State Lifetime, paper says "Cleaning up task state after exceptions...since the thread stack naturally tracks the live state for that task". I am not quite familiar with the threaded system, could you provide a more brief description of this sentence?
- In section 4, the author presented the idea of "Compiler Support for Threads", but these descriptions about functionalities provided by the compiler are very abstract and really hard for me understand how they are related to each other. Could you show me an example and draw a diagram to visualize the process? and I wonder how the "standard libraries thread unsafe" problems could be solved? Can the compiler remedy this problems?

Questions

- What exactly do the authors mean by 'naturally'? They employ the concept in more than one place that Threads are more natural than events (in terms of use, comprehension, etc.), but never actually provided any basis (i.e. psychological reports, etc.) that prove their case.
- Along that line, one of their supporting claims that Threads is 'more natural' is that even in their own Ninja system, they employed threads for complex parts in lieu of using Events. Isn't that kinda like saying that threads is good because we used threads because threads is good so we used threads (ad nauseum)?

Questions

- The authors suggested several modifications on the current thread implementations which are mainly inspired from their counterparts in events. (e.g.cooperative threads, dynamic stack allocation against stack overflow etc.) In general is it fair to say that authors are suggesting event-like threads for performance improvements, specifically for uniprocessors.
- In general it occurs like designing a compiler that's suited for estimating dynamic stack bounds for dynamic allocation in stacks would be a tedious job, especially in nested function calls, mutual recursions etc. Are there any implementations of such compilers? If there are how precise are their estimations?

Questions

- With reference to live state management, how does reordering variables with overlapping lifetimes help ? As in how does it "prevent live variables from unnecessarily pinning dead ones in memory" ?
- Which has higher locality - Batched event processing or Thread processing ?

Questions

- It scales to past 100,000 threads/concurrent tasks. But why does throughput drop off past 200k threads, and not for events?
- Is it an inherent property of threads that "too many" will drop throughput?
- Is a user-level thread package required for such performance levels?
- Does Linux 2.6 kernel threads achieve something similar?

Questions

- The paper says Event is a bad idea and SEDA says event is a good idea. Do both of idea conflict with each other?
- Can we say improving compiler will help improve thread issues?