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

Rob von Behren, Jeremy Condit and Eric Brewer

Presented by: Kristi McBurnie
Outline

- Introduction
- Threads: The Argument
- Compiler Support
- The Test
- Conclusion
- Discussion Questions
Response to Ousterhout’s paper "Why Threads are a Bad Idea (for most purposes)"

Purpose is to show:
- Weaknesses of threads are due to specific implementations
- Threads are more natural
- Threads can achieve the same strengths as events
- Only small improvements to compilers are needed
Introduction

- Lauer and Needham (1978)
  - Duality between message-passing and process-based systems
    - Blocking graph
    - Equivalent performance
  - Message-passing system used does not correspond to today’s event systems
  - Suggests that criticisms for threads are due to problems with thread packages
Introduction

- 4 literature arguments for why events are better:
  - Inexpensive synchronization due to cooperative multitasking
  - Lower overhead for managing state (no stacks)
  - Better scheduling and locality, based on application-level information
  - More flexible control flow
Introduction

- 5 criticisms for threads:
  - Poor performance
    - Preemption and additional kernel crossing
  - Restrictive control flow
    - Complex patterns are too difficult to use well
  - Synchronization mechanisms are heavyweight
    - Cooperative thread systems
  - Ineffective state management
    - Dynamic stack growth
  - Suboptimal scheduling decisions
    - At application level
Threads: The Argument

- Better for high-concurrency servers
  - Independent concurrent requests
  - Sequential code

- Control Flow
  - Events ‘obfuscate’ control flow
    - Mentally matching calls with returns
    - Stack ripping – manually saving and restoring live state
  - Thread systems are more natural
    - Group calls with returns
    - Run-time call stack encapsulates live state
Threads: The Argument

- Exception handling and state lifetime
  - Event based systems can miss deallocation because of branches in the control flow
  - Thread stack naturally tracks the live state for that task

- Existing servers
  - Complex applications
  - Non-high concurrency applications - simpler

- Just fix events?
  - Duplicate the syntax and run-time behaviour of threads
Compiler Support

- Tighter integration between compilers and runtime systems to improve thread performance
  - Minor modifications

- Dynamic Stack Growth
  - Avoids the trade-off between overflow and wasted space
  - Recursive functions and function pointers produce challenges
Compiler Support

- Live state management
  - Purge unnecessary state, reorder or move overlapping variables from the stack
  - Compiler can warn the programmer

- Synchronization
  - Compiler can warm programmer about data races
  - Determine which atomic sections are safe to run concurrently
The Test

- Designed and implemented a simple (5000 line) user-level cooperative threading package for Linux
- Wrote 700 line test web server, Knot
  - Knot-A
    - Accepting new connections over of processing of active ones
    - Closely matches policy used by Haboob
  - Knot-C
    - Processing of active connections over accepting new ones
    - Natural throttling mechanism
- Tested it against SEDA’s event server Haboob
  - Test suite used to evaluate SEDA
The Test

- Same general performance pattern
  - Performance degradation due to poor scalability
  - Well designed thread package can achieve the same scaling behaviour as a well designed event system
The Test

- Steady-state bandwidth in Knot-C is 700Mbit/s
  - Limited by interrupt processing overhead in the kernel
- Haboob’s max bandwidth 500Mbit/s
  - Becomes CPU limited at 512 clients
The Test

- Haboob (because it is event based):
  - Context switching (30,000)
  - Large number of module crossing and queueing operations
  - Temp objects and relies heavily on garbage collection

- Knot avoids these because it is thread based
Conclusion

- Many techniques were introduced in previous work
  - Cooperative user-level threads packages
    - Adya et al.
  - Concurrency and lightweight threading
  - Dynamically linked stacks

- Bring them together in a single package
  - Make them accessible
  - Expand on them that by exploring thread performance issues and compiler support techniques
Conclusion

- Ousterhout’s arguments to not conflict with the authors’
  - Programming with concurrency is fundamentally difficult
  - Cooperatively scheduled events are preferable (for most purposes) because they allow programmers to avoid concurrent code in most cases
  - Cooperative scheduling helps simplify concurrency
    - Authors argue this tool is better used in the context of the simpler programming model of threads
  - Supports threads for true concurrency, which is the case in the target space
Conclusion

- Event and thread systems can be used to obtain good performance in high concurrency systems

- Simpler programming model and wealth of compiler analysis
  - Gives threads an advantage

- Integration between the compiler and thread systems
  - ‘Clean and simple interface to the programmer while achieving superior performance’
Discussion Questions

- In section 2.2 they say that they modified the Pth thread library to perform just as well (or better) than event-driven programming for up to 100,000 concurrent tasks. However, after this point, the thread performance drops off dramatically while the event performance remains constant. Isn't this essentially an argument against threads and for events, especially in *highly* concurrent systems?

- If events can perform as well as threads for up to 100,000 concurrent tasks, and better beyond that, doesn't this essentially destroy the entire paper?
Discussion Questions

- Knot-C favours handling current connections, and as client number goes beyond 1024 the throughput is almost a constant. Does this mean that it provides lucky connected clients with good services, while lots of other clients connections are refused?

- What are the trade-offs between refusing connections and graceful degradation? Which is a better approach or does it depend on the application?
It seems strange that the writers of this paper, when commenting on the "problems" of threads, failed to bring up the difficulties in debugging threads. The complexity of creating/debugging a fine-grained threading mechanism almost seems to be more of an issue than any of the other "problems".

Is this an issue?
In section 4 the authors talk briefly about compiler support for threads and how this is needed. They also touch on the need for methods which will "[warn] the programmer about data races" or "provide an upper bound on the amount of stack space needed when calling each function".

Is this really feasible?
In section 4.1 the authors mention that recursive function and function pointers are a problem for the dynamic growth of stacks. They say that these problems can be addressed with further analyses, without giving any more details.

What are your thoughts on this? Do they seem to downplay the difficulty and importance of this problem?
Given the performance gains of events and the massive amounts of modifications they needed to make to thread packages to try to get it as good as events (and in really highly concurrency, events still outperform threads), is it really worth all the trouble and effort to implement threads?

- Is there any successful implementation of this (ie. compiler support for threads)?