Capriccio: Scalable Threads for Internet Services (by Behren, Condit, Zhou, Necula, Brewer)

Presented by
Alex Sherman and Sarita Bafna
Main Contribution

• Capriccio implements a scalable user-level thread package as an alternative to event-based and kernel-thread models.
• The authors demonstrate scalability to 100,000 threads and argue the model should be a more efficient alternative for Internet Server implementation.
Key Features

- Scalability with user-level threads
  - Cooperative scheduling
  - Asynchronous disk I/O
  - Efficient thread operations - $O(1)$
- Linked stack management
- Resource-aware scheduling
Outline

• Related Work and “Debate”
• Capriccio Scalability
• Linked Stack Management
• Resource-Aware Scheduling
• Conclusion
Related Work

• Events vs. Threads (Ouserhout, Laura and Needham, Adya, SEDA)
• User-level thread packages (Filaments, NT Fibers, State Threads, Scheduler Activations)
• Kernel Threads (NTPL, Pthreads)
• Stack Management (Lazy Threads)
Debate – event-based side

• Event-based arguments by Ousterhout (Why threads are bad?, 1996)
  – Events are more efficient (context switching, locking overheads with threads)
  – Threads - hard to program (deadlocks, synchronization)
  – Poor thread support (portability, debugging)

• Many event-based implementation (Harvest, Flash, SEDA)
Debate – other arguments

- Neutral argument by Lauer and Needham (On the duality of OS system structures, 1978)
- Pro-thread arguments by Behren, Condit, Brewer (Why events are bad?, 2003)
  - Greater code readability
  - No “stack-ripping”
  - Slow thread performance - implementation artifact
  - High performance servers more sensitive to scheduling
Why user-level threads?

- Decoupling from the OS/kernel
  - OS independence
  - Kernel variation
  - Address application-specific needs
- Cooperative threading – more efficient synchronization
- Less “kernel crossing”
- Better memory management
Implementation

- Non-blocking wrappers for blocking I/O
- Asynchronous disk I/O where possible
- Cheap synchronization
- Efficient O(1) thread operations
Benchmarks

• (left) Capriccio scales to 100,000 threads
• (right) Network I/O throughput with Capriccio only has 10% overhead over epoll
• With asynchronous I/O disk performance is comparable in Capriccio vs. other thread packages
Disadvantages of user-level threads

- Non-blocking wrappers of blocking I/O increase kernel crossings
- Difficult to integrate with multiple processor scheduling
Dynamic Linked Stacks

- Problem: Conservative stack allocations per thread are unsuitable for programs with many threads.
- Solution: Dynamic stack allocation with linked chunks alleviates VM pressure and improves paging behavior.
- Method: Compile-time analysis and checkpoint injection into the code.
Weighted Call Graph

• Each node is a call site annotated with the maximum stack space for that call.
• Checkpoints must be inserted at each recursive frame and well-spaced call sites.
• Checkpoints determine whether to allocate a new stack chunk.
Challenging cases

- Function pointers are only determined at run-time.
- External function calls require conservative stack allocation.
Apache 2.0.44 Benchmark

• Given 2 KB “max-path” only 10.6% call sites required check-pointing code.
• Overhead in the number of instructions was 3-4%.
Resource-Aware Scheduling

- Key idea: View an application as a sequence of stages separated by blocking points.
- Method: Track resources (CPU, memory, file descriptors) used at each stage and schedule threads according to resources.
• Tracking CPU cycles and other resource usage at each edge and node.
• Threads are scheduled so that for each resource, utilization is increased until maximum throughput and then throttled back.
Pitfalls

- Maximum capacity of a particular resource is difficult to determine (e.g., internal memory pools).
- Thrashing is not easily detectable.
- Non-yielding threads lead to unfairness and starvation in cooperative scheduling.
- Blocking graphs are expensive to maintain (for Apache 2.0.44 stack trace overhead is 8% of execution time).
Web Server Performance

- Apache 2.0.44 on a 4x500 MHz Pentium server has 15% higher throughput with Capriccio.
Conclusion

• Capriccio demonstrates a user-level thread package that achieves
  – High scalability
  – Efficient stack management
  – Scheduling based on resource usage

• Drawbacks
  – Performance not comparable to event-based systems
  – High overhead in stack tracing
  – Lack of sufficient multi-processor support
Future Work

- Extending Capriccio to work with multiple processors
- Reducing the kernel crossings with batching asynchronous network I/O
- Disambiguate function pointers in stack allocation
- Improving resource-aware scheduling
  - Tracking variance in resource usage
  - Better detection of thrashing