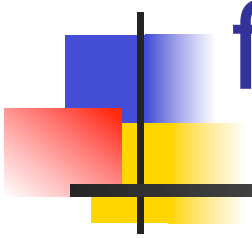


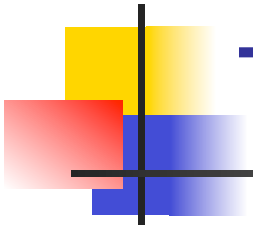
Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou,
Geroge Necula and Eric Brewer

University of California at Berkeley

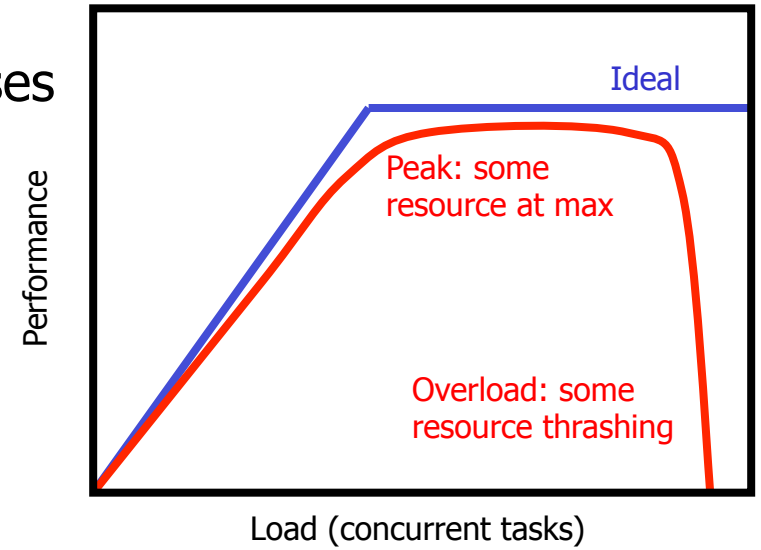
{jrvb,jcondit,zf, necula, brewer}@cs.berkeley.edu

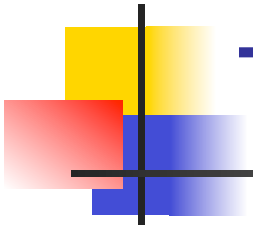
http://capriccio.cs.berkeley.edu



The Stage

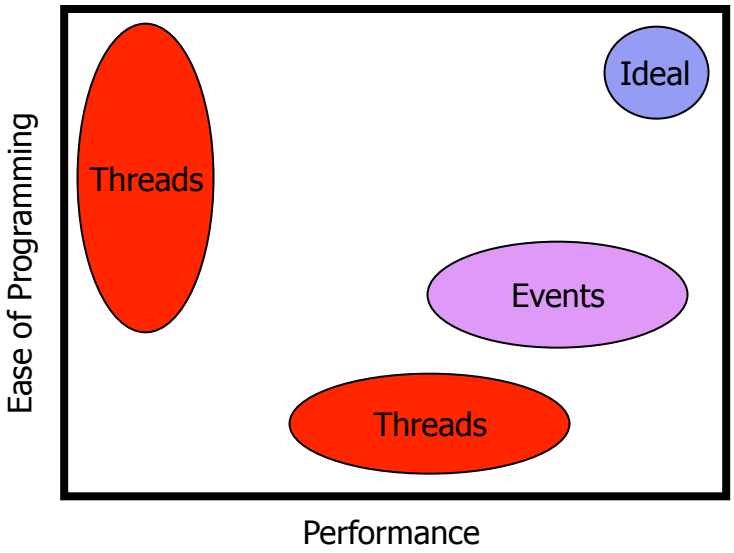
- Highly concurrent applications
  - Internet servers & frameworks
    - Flash, Ninja, SEDA
  - Transaction processing databases
- Workload
  - High performance
  - Unpredictable load spikes
  - Operate “near the knee”
  - Avoid thrashing!

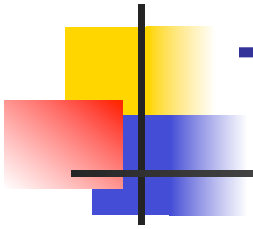




The Price of Concurrency

- What makes concurrency hard?
  - Race conditions
  - Code complexity
  - Scalability (no *O(n)* operations)
  - Scheduling & resource sensitivity
  - Inevitable overload
- Performance vs. Programmability
  - No current system solves
  - Must be a better way!



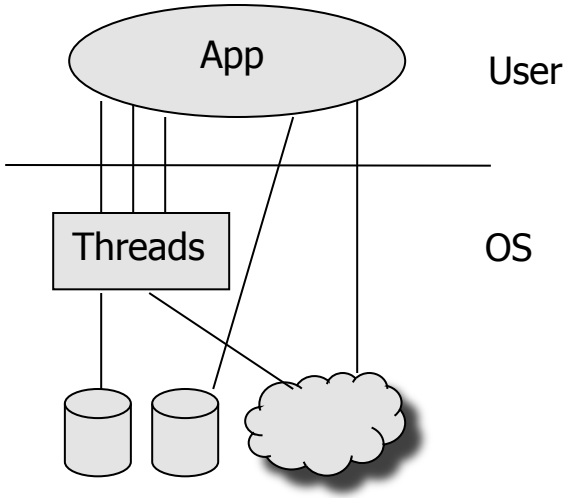


The Answer: Better Threads

- Goals
  - Simplify the programming model
    - Thread per concurrent activity
    - Scalability (100K+ threads)
  - Support existing APIs and tools
  - Automate application-specific customization
- Tools
  - Plumbing: avoid *O(n)* operations
  - Compile-time analysis
  - Run-time analysis
- Claim: User-Level threads are key

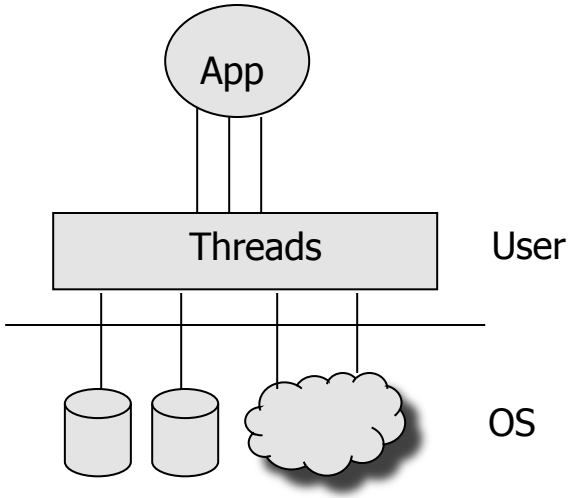
The Case for User-Level Threads

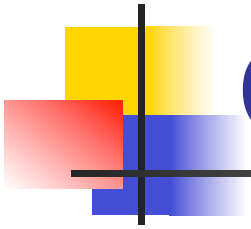
- Decouple programming model and OS
  - Kernel threads
    - Abstract hardware
    - Expose device concurrency
  - User-level threads
    - Provide clean programming model
    - Expose logical concurrency
- Benefits of user-level threads
  - Control over concurrency model!
  - Independent innovation
  - Enables static analysis
  - Enables application-specific tuning



The Case for User-Level Threads

- Decouple programming model and OS
  - Kernel threads
    - Abstract hardware
    - Expose device concurrency
  - User-level threads
    - Provide clean programming model
    - Expose logical concurrency
- Benefits of user-level threads
  - Control over concurrency model!
  - Independent innovation
  - Enables static analysis
  - Enables application-specific tuning



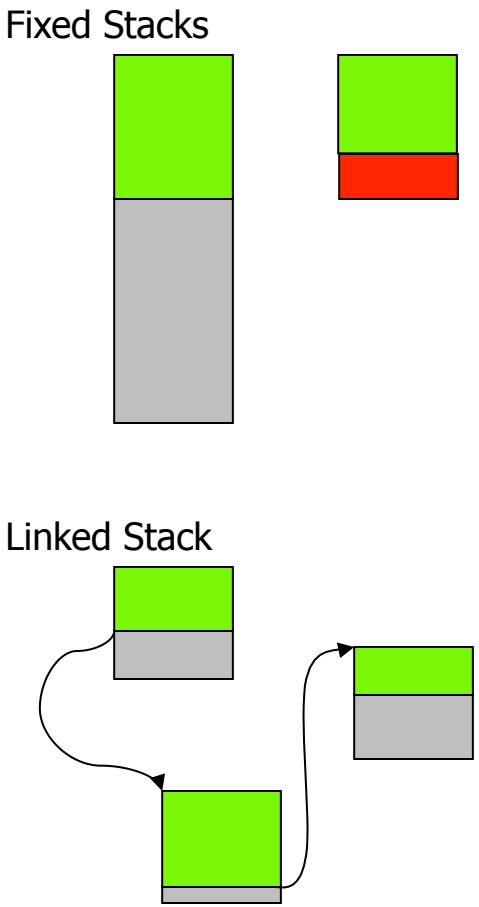


Capriccio Internals

- Cooperative user-level threads
  - Fast context switches
  - Lightweight synchronization
- Kernel Mechanisms
  - Asynchronous I/O (Linux)
- Efficiency
  - Avoid *O(n)* operations
  - Fast, flexible scheduling

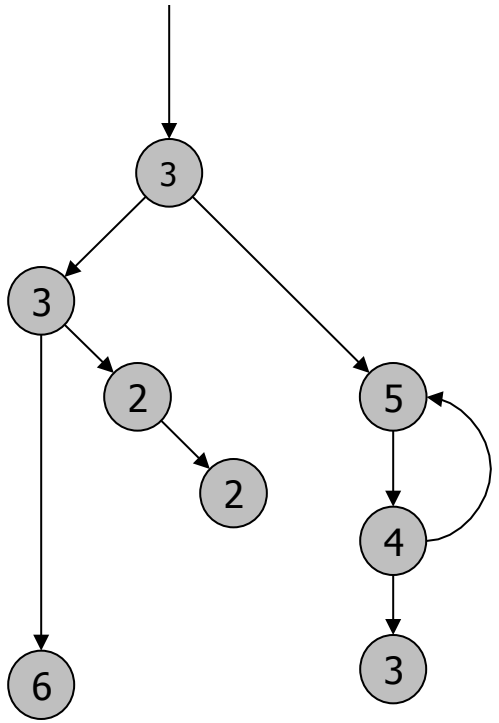
Safety: Linked Stacks

- The problem: fixed stacks
  - Overflow vs. wasted space
  - Limits thread numbers
- The solution: linked stacks
  - Allocate space as needed
  - Compiler analysis
    - Add runtime checkpoints
    - Guarantee enough space until next check



Linked Stacks: Algorithm

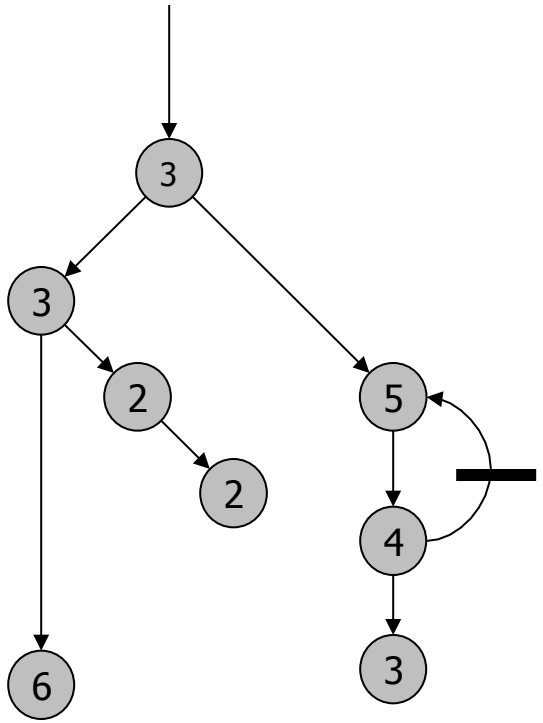
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

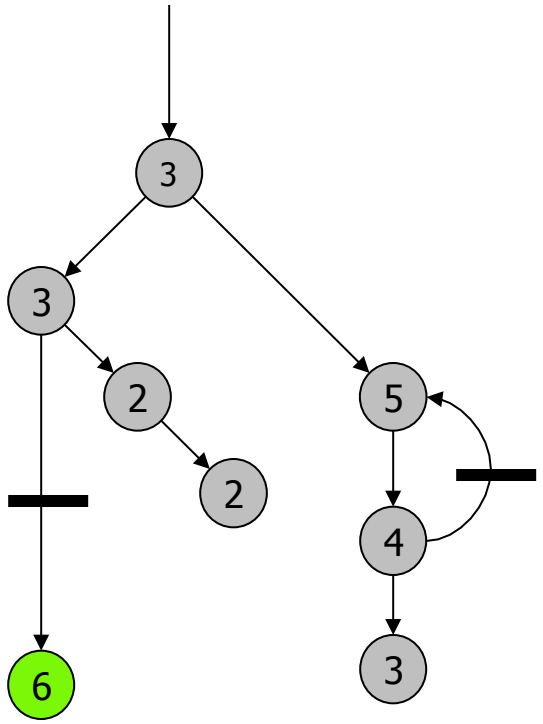
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

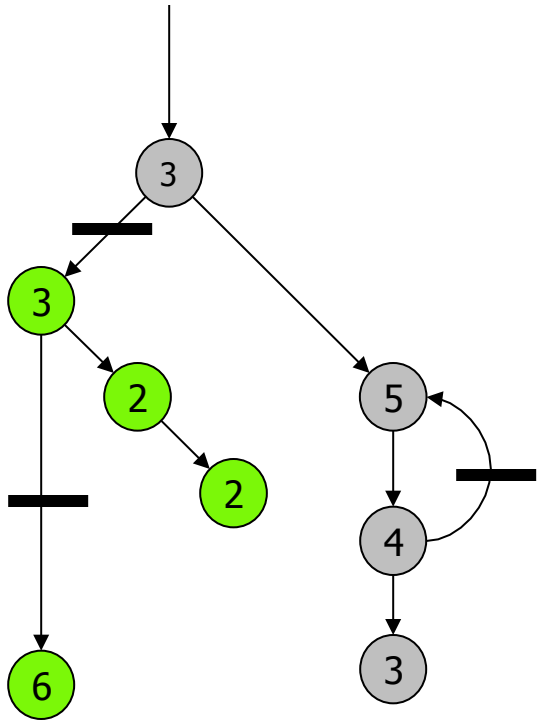
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

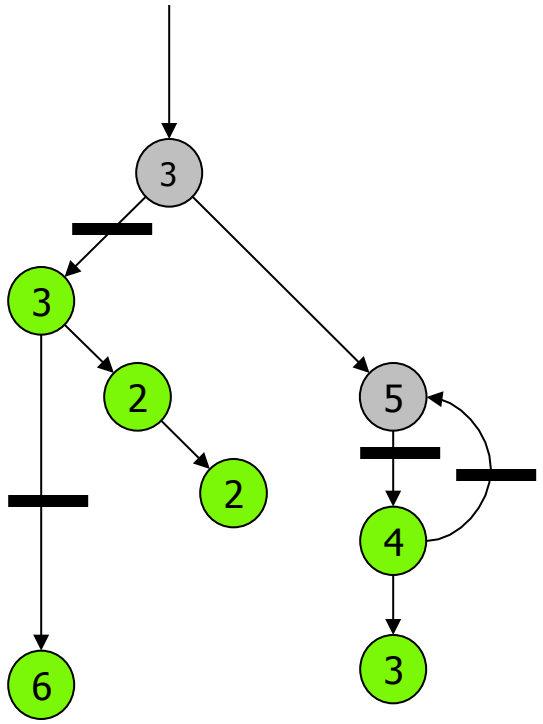
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

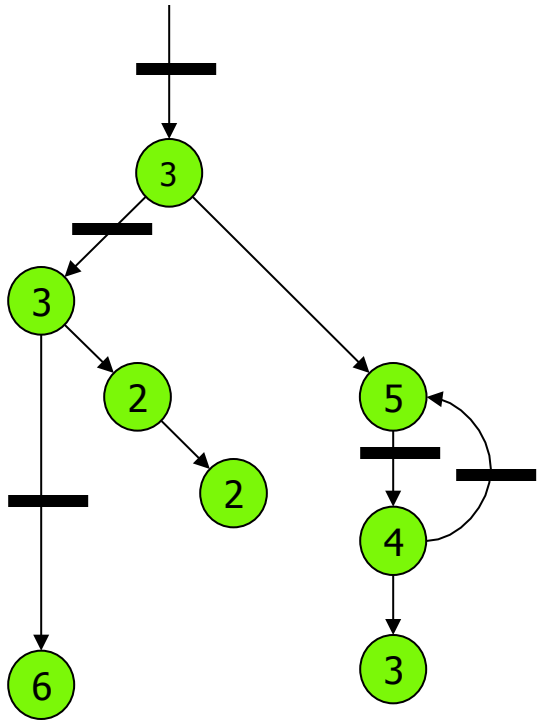
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

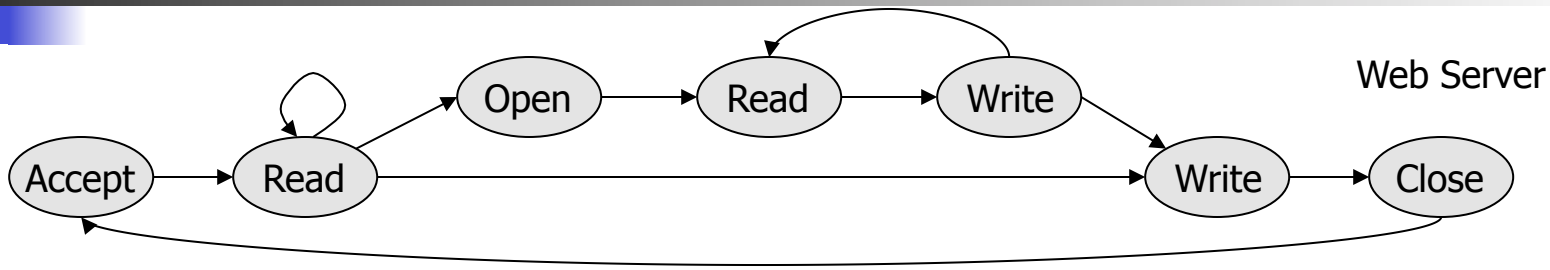
Linked Stacks: Algorithm

- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack

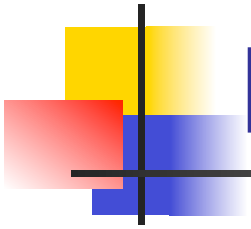


MaxPath = 8

Scheduling: The Blocking Graph

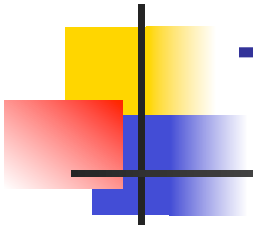


- Lessons from event systems
  - Break app into stages
  - Schedule based on stage priorities
  - Allows SRCT scheduling, finding bottlenecks, etc.
- Capriccio does this for threads
  - Deduce stage with stack traces at blocking points
  - Prioritize based on runtime information



Resource-Aware Scheduling

- Track resources used along BG edges
  - Memory, file descriptors, CPU
  - Predict future from the past
  - Algorithm
    - Increase use when underutilized
    - Decrease use near saturation
- Advantages
  - Operate near the knee w/o thrashing
  - Automatic admission control

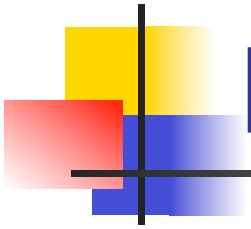


Thread Performance

	Capriccio	Capriccio-notrace	LinuxThreads	NPTL
Thread Creation	**21.5**	**21.5**	**37.5**	**17.7**
Context Switch	**0.56**	**0.24**	**0.71**	**0.65**
Uncontested mutex lock	**0.04**	**0.04**	**0.14**	**0.15**

Time of thread operations (microseconds)

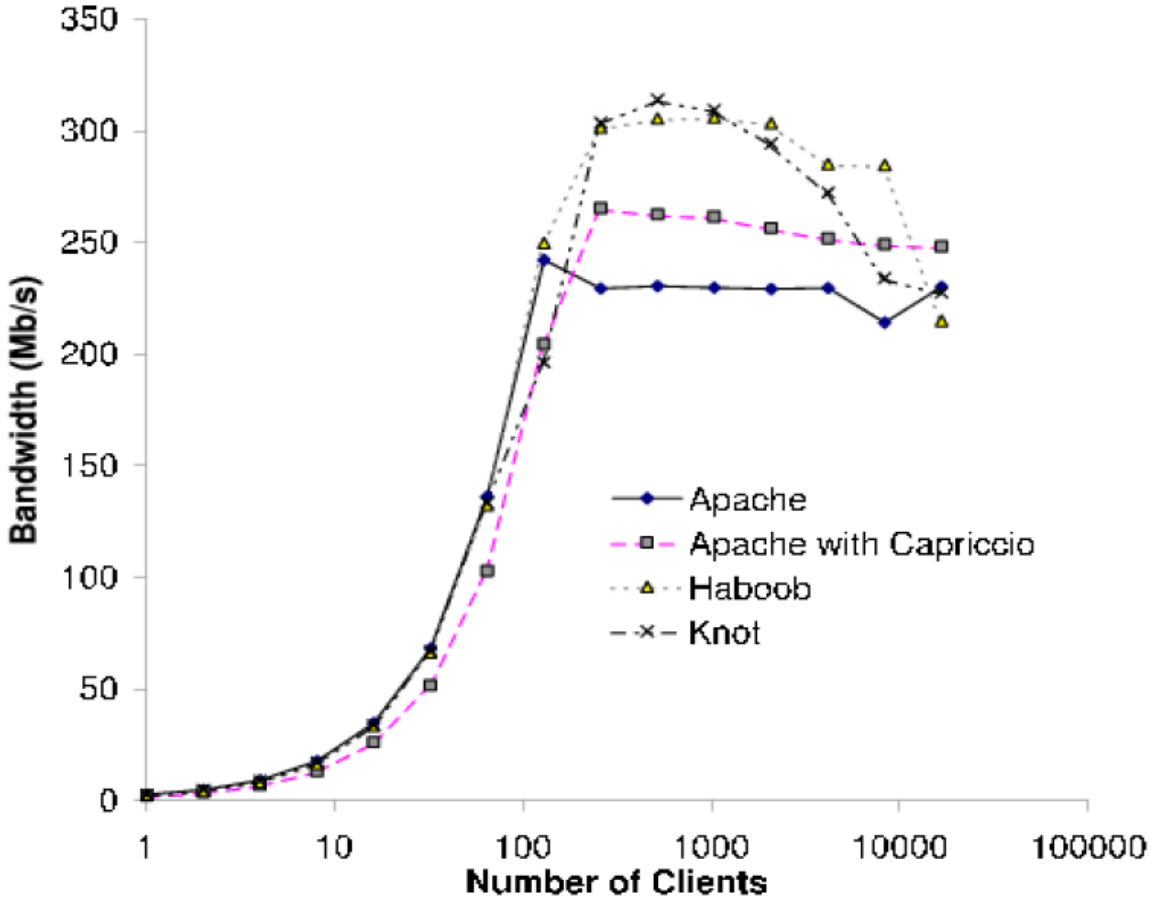
- Slightly slower thread creation
- Faster context switches
  - Even with stack traces!
- Much faster mutexes

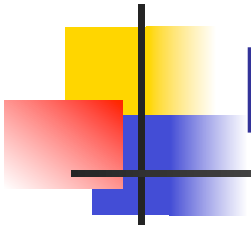


Runtime Overhead

- Tested Apache 2.0.44
- Stack linking
  - 78% slowdown for null call
  - 3-4% overall
- Resource statistics
  - 2% (on all the time)
  - 0.1% (with sampling)
- Stack traces
  - 8% overhead

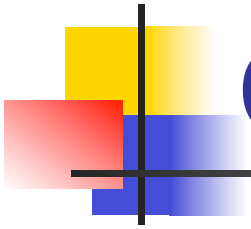
Web Server Performance





Future Work

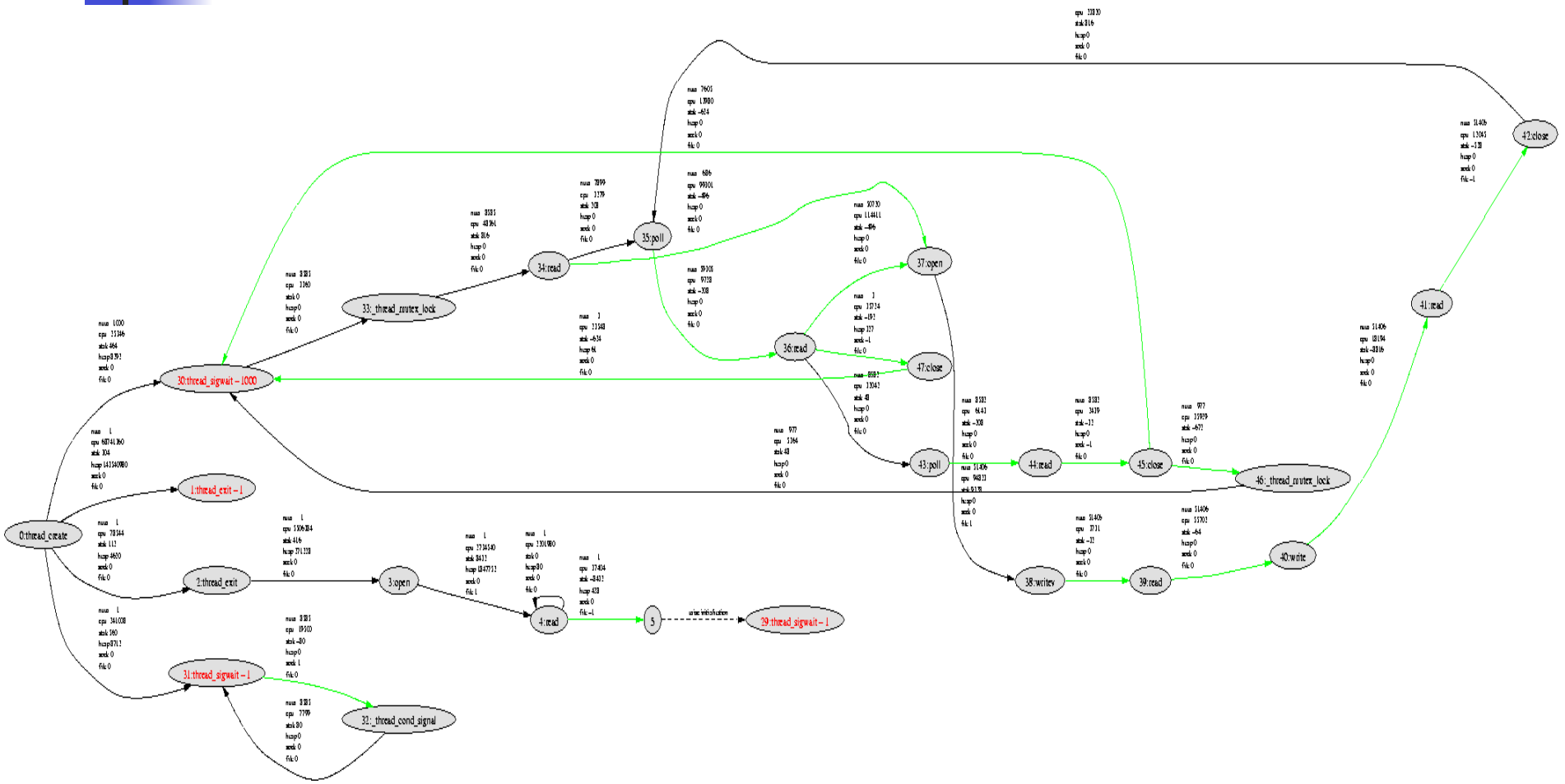
- Threading
  - Multi-CPU support
  - Kernel interface
- (enabled) Compile-time techniques
  - Variations on linked stacks
  - Static blocking graph
  - Atomicity guarantees
- Scheduling
  - More sophisticated prediction



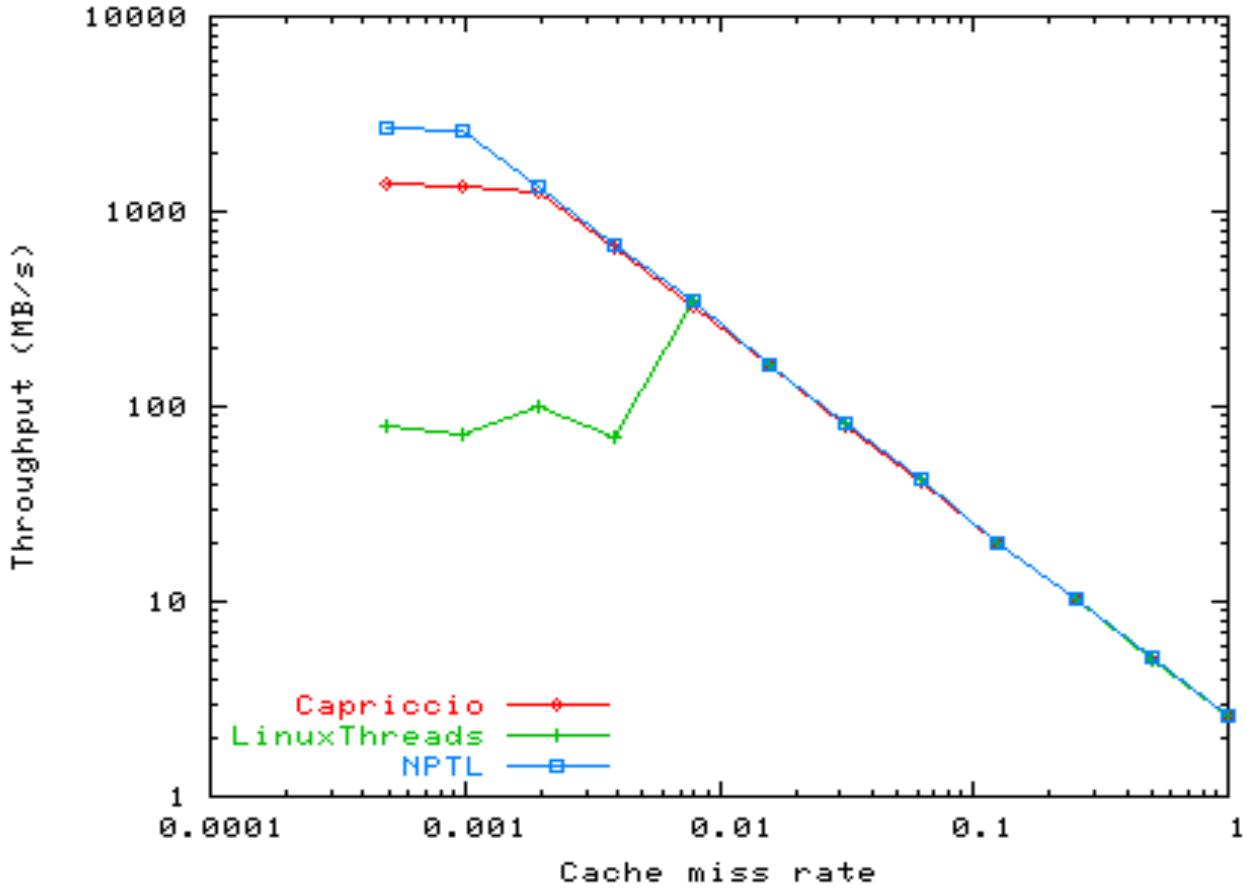
Conclusions

- Capriccio *simplifies* high concurrency
  - Scalable & high performance
  - *Control* over concurrency model
    - Stack safety
    - Resource-aware scheduling
    - Enables compiler support, invariants
- Themes
  - User-level threads are key
  - Compiler techniques very promising

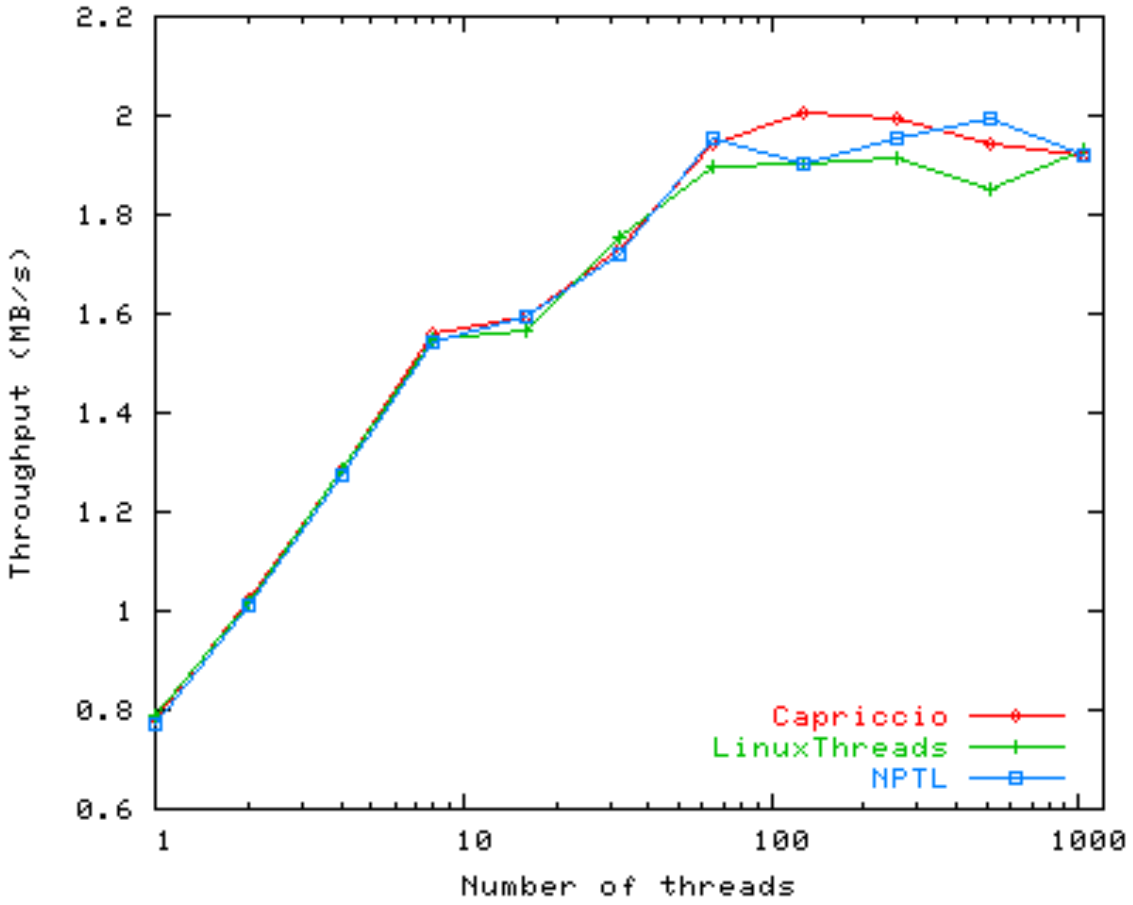
Apache Blocking Graph



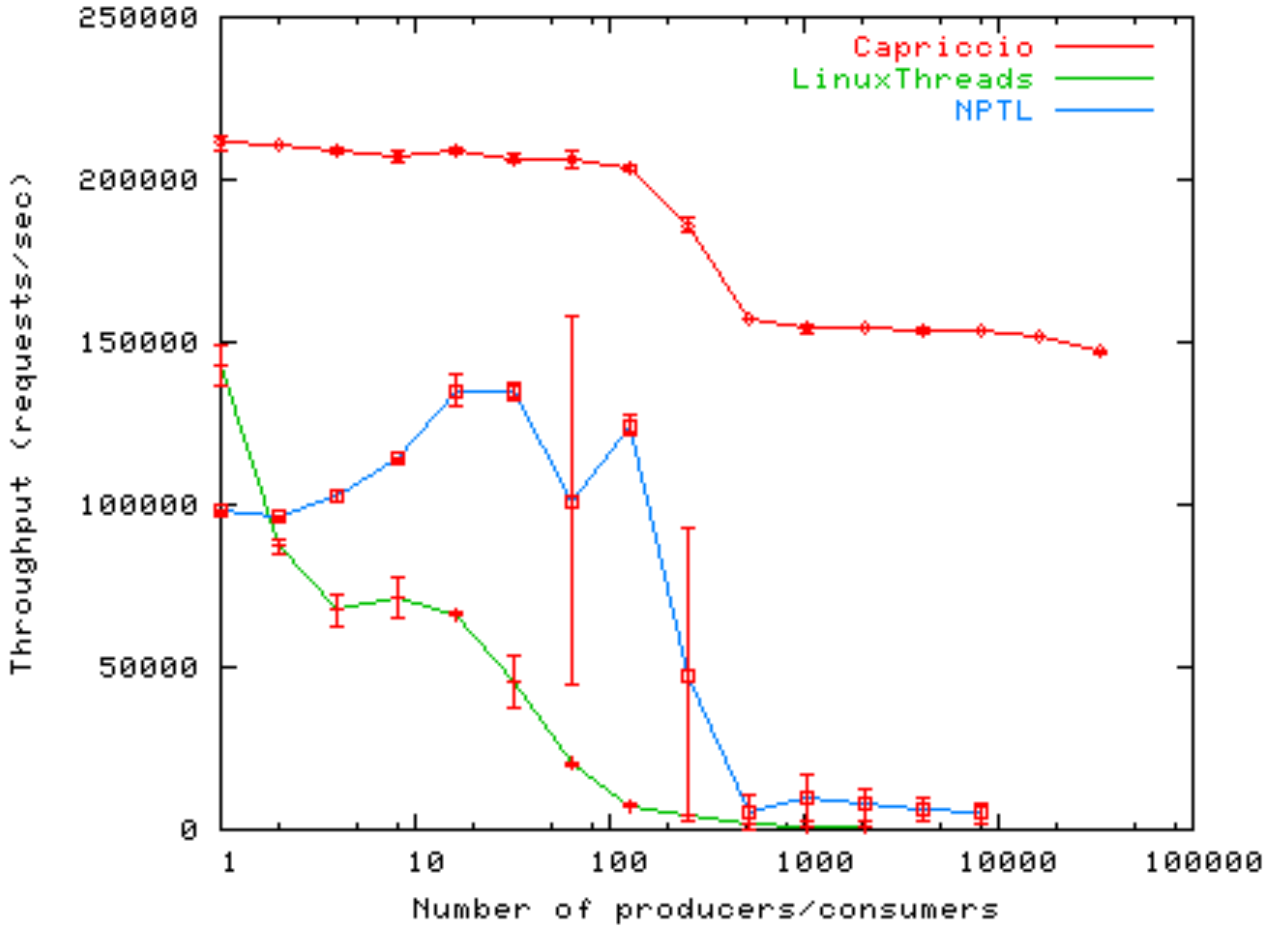
Microbenchmark: Buffer Cache



Microbenchmark: Disk I/O



Microbenchmark: Producer / Consumer



Microbenchmark: pipetest