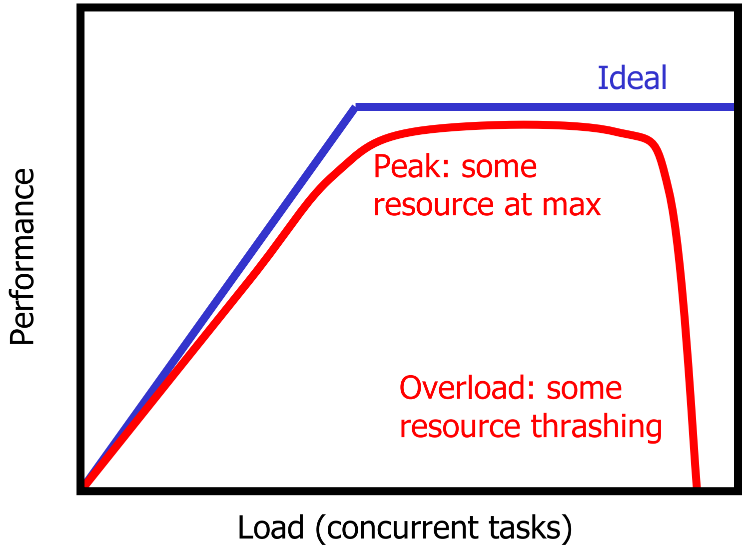
Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou,
George Necula and Eric Brewer
University of California at Berkeley
{jrvb, jcondit, zf, necula, brewer}@cs.berkeley.edu
http://capriccio.cs.berkeley.edu

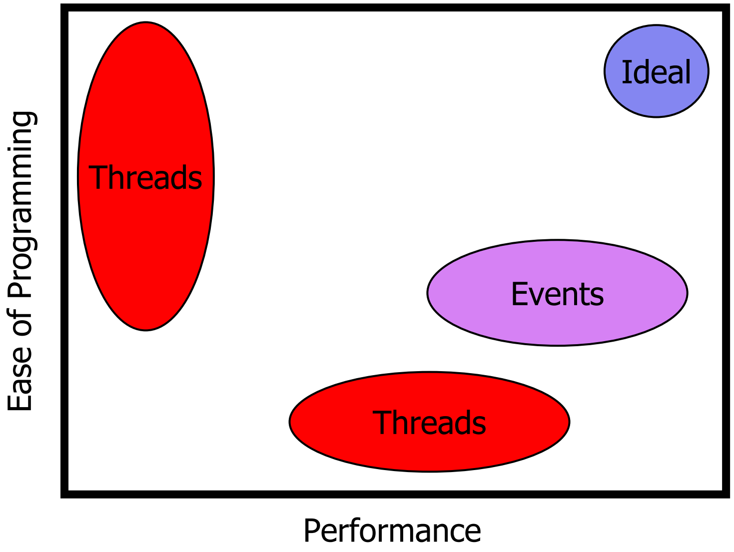
The Stage

- 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

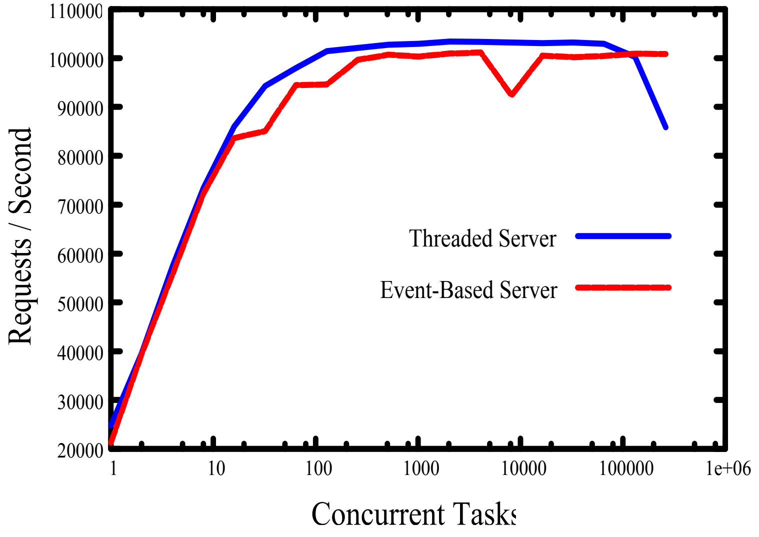
- Goals
    - Simple programming model
    - Good tools & infrastructure
        - Languages, compilers, debuggers, etc.
    - Good performance
- Claims
    - Threads are preferable to events
    - User-Level threads are key

“But Events *Are* Better!”

- Recent arguments for events
  - Lower runtime overhead
  - Better live state management
  - Inexpensive synchronization
  - More flexible control flow
  - Better scheduling and locality
- All true but…
  - Lauer & Needham duality argument
  - Criticisms of *specific* threads packages
  - No *inherent* problem with threads!
    - Thread implementations can be improved

Threading Criticism: Runtime Overhead

- *Criticism: Threads don't perform well for high concurrency*
- Response
  - Avoid *O(n)* operations
  - Minimize context switch overhead
- Simple scalability test
  - Slightly modified GNU Pth
  - Thread-per-task vs. single thread
  - Same performance!

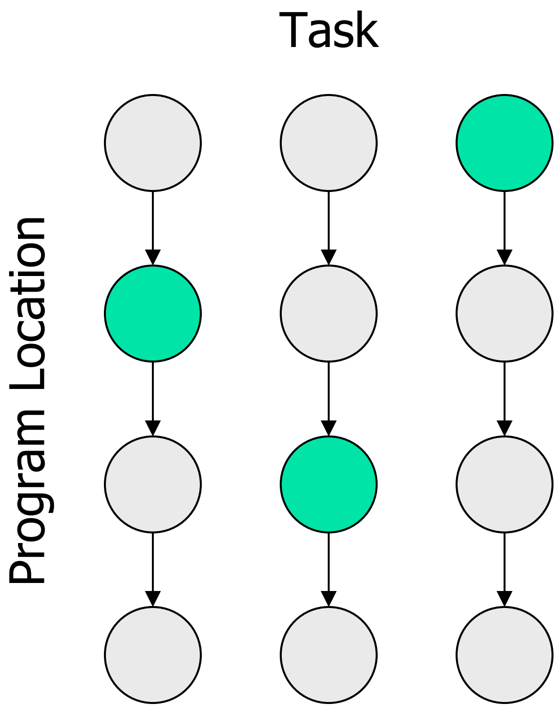


Threading Criticism: Synchronization

- *Criticism: Thread synchronization is heavyweight*
- Response
  - Cooperative multitasking works for threads, too!
  - Also presents same problems
    - Starvation & fairness
    - Multiprocessors
    - Unexpected blocking (page faults, etc.)
  - Both regimes need help
    - Compiler / language support for concurrency
    - Better OS primitives

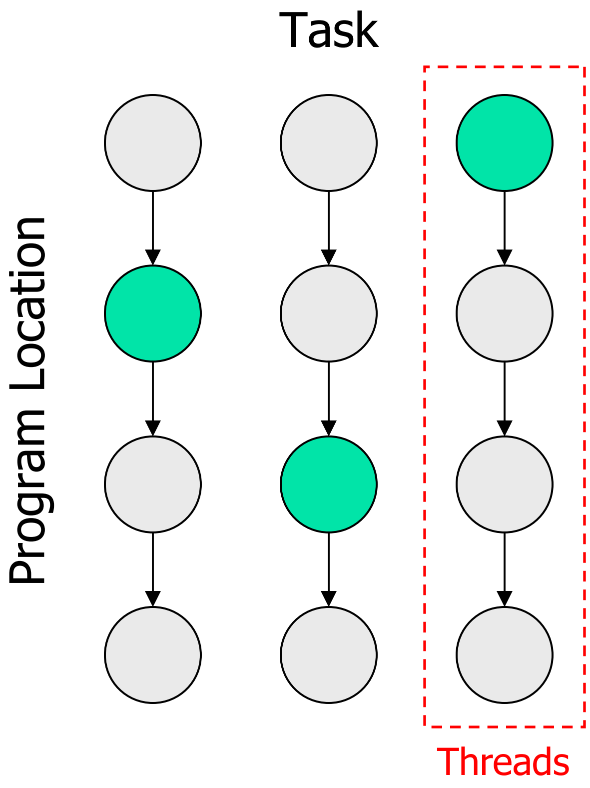
Threading Criticism: Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



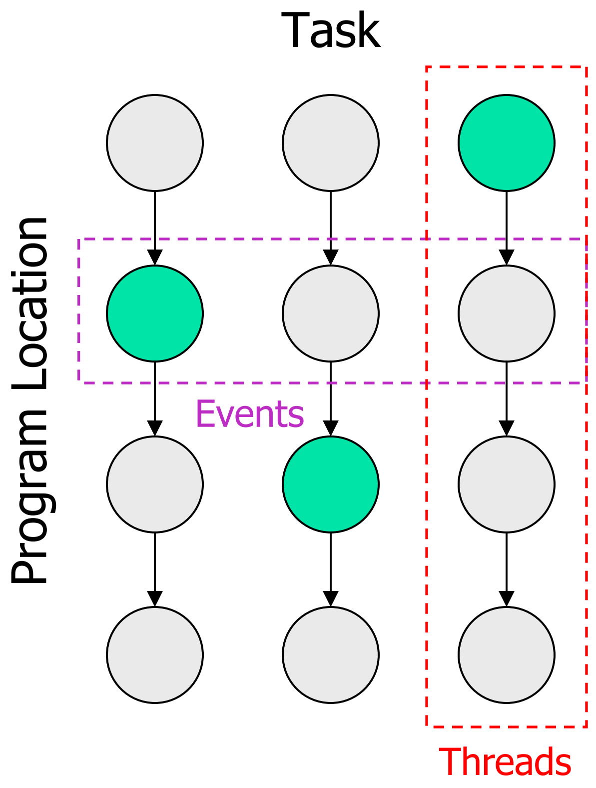
Threading Criticism: Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



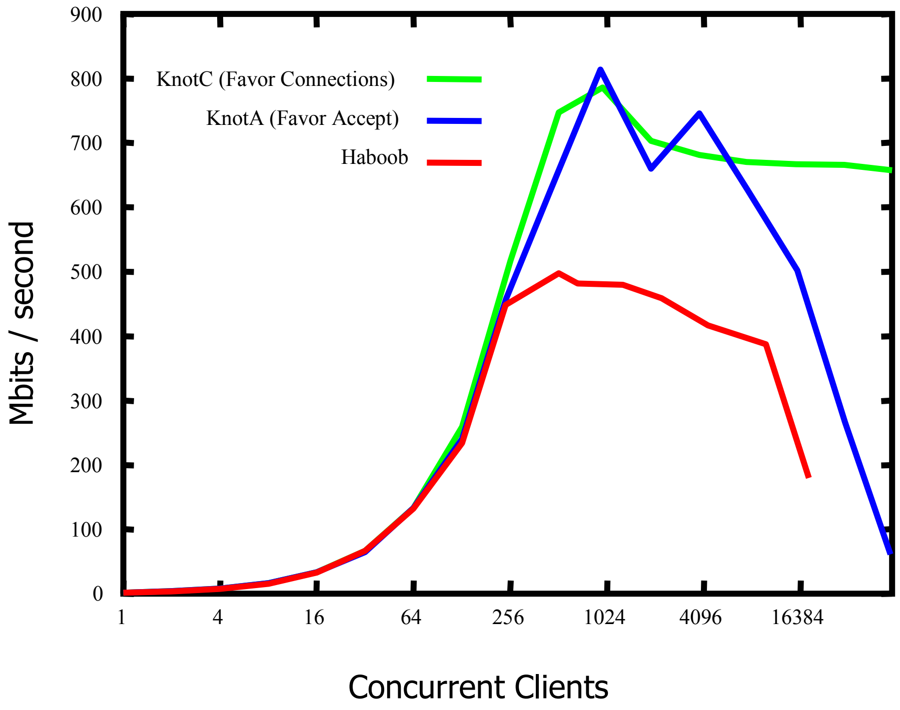
Threading Criticism: Scheduling

- *Criticism: Thread schedulers are too generic*
  - Can't use application-specific information
- Response
  - 2D scheduling: task & program location
    - Threads schedule based on task only
    - Events schedule by location (e.g. SEDA)
      - Allows batching
      - Allows prediction for SRCT
  - Threads can use 2D, too!
    - Runtime system tracks current location
    - Call graph allows prediction



The Proof's in the Pudding

- User-level threads package
  - Subset of pthreads
  - Intercept blocking system calls
  - No *O(n)* operations
  - Support > 100K threads
  - 5000 lines of C code
- Simple web server: Knot
  - 700 lines of C code
- Similar performance
  - Linear increase, then steady
  - Drop-off due to `poll()` overhead



Arguments For Threads

- More natural programming model
  - Control flow is more apparent
  - Exception handling is easier
  - State management is automatic
- Better fit with current tools & hardware
  - Better existing infrastructure

Arguments for Threads: Control Flow

- Events obscure control flow
  - For programmers *and* tools

Threads

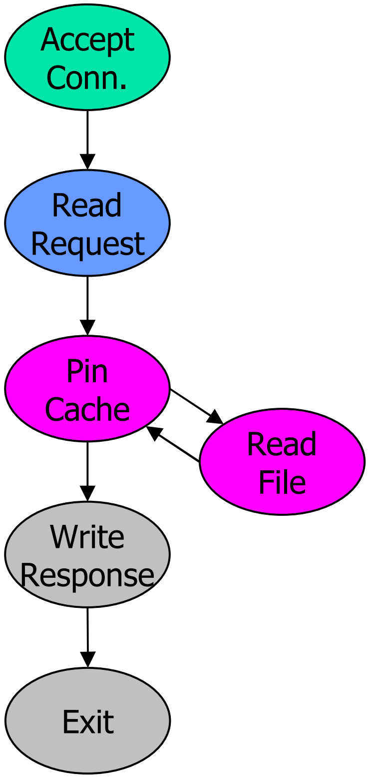
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; CacheHandler.enqueue(s);
}
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
. . .
ExitHandler(struct session *s) {
    …;  unpin(&s);  free_session(s);  }
```

Web Server



Arguments for Threads: Control Flow

- Events obscure control flow
  - For programmers *and* tools

Threads

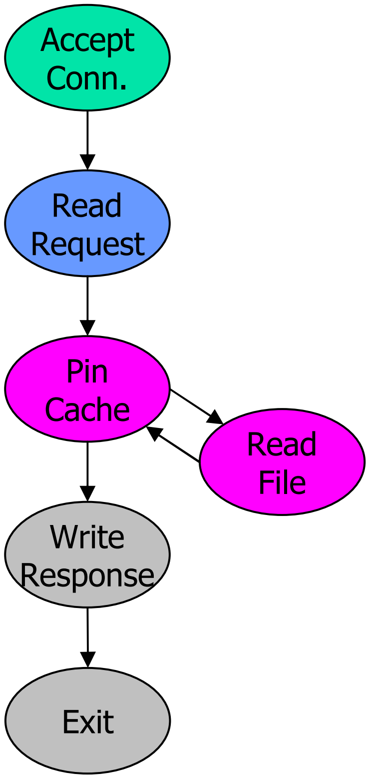
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; CacheHandler.enqueue(s);
}
. . .
ExitHandler(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



Arguments for Threads: Exceptions

- Exceptions complicate control flow
  - Harder to understand program flow
  - Cause bugs in cleanup code

Threads

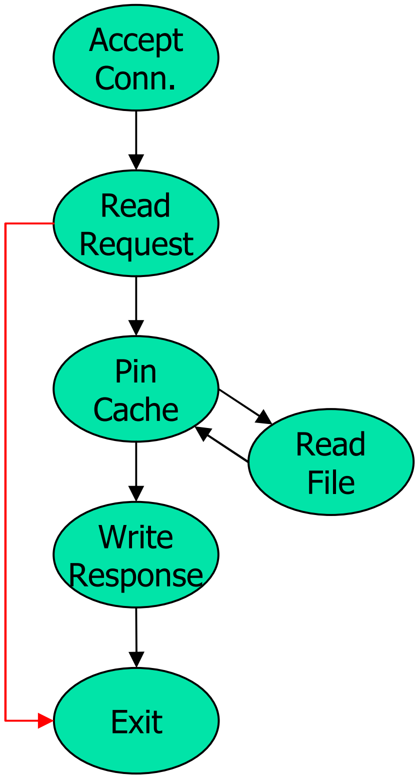
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandler(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



Arguments for Threads: State Management

- Events require manual state management
- Hard to know when to free
    - Use GC or risk bugs

Threads

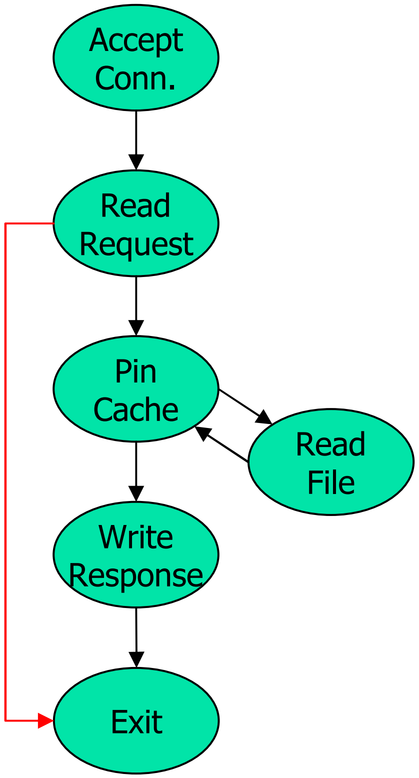
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    …; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandler(struct session *s) {
    …;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



Arguments for Threads: Existing Infrastructure

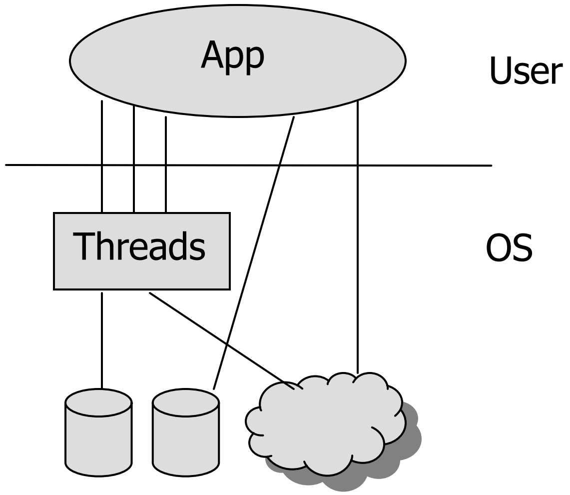
- Lots of infrastructure for threads
    - Debuggers
    - Languages & compilers
- Consequences
    - More amenable to analysis
    - Less effort to get working systems

Building Better Threads

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

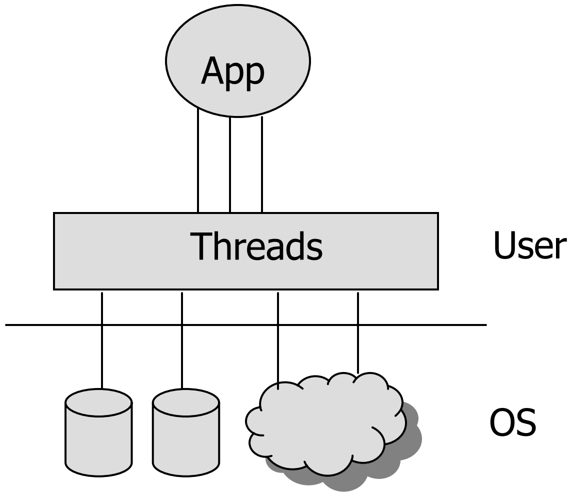
The Case for User-Level Threads

- 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

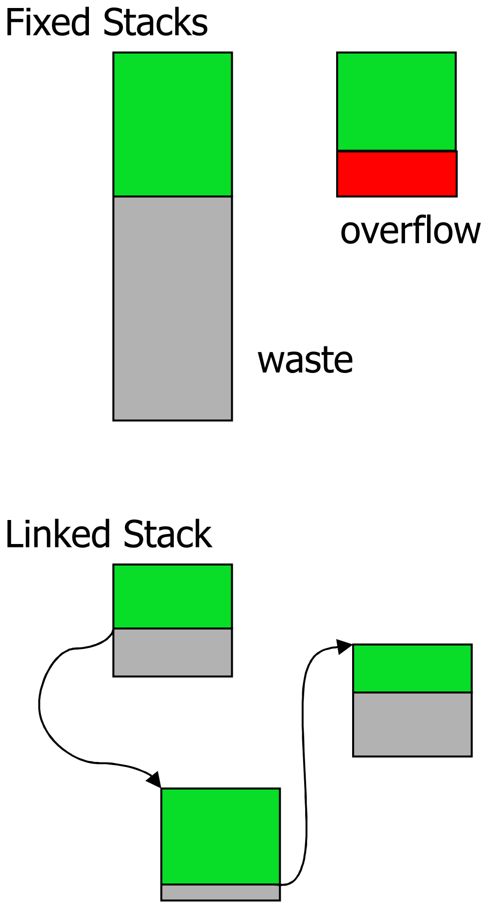


Capriccio Internals

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

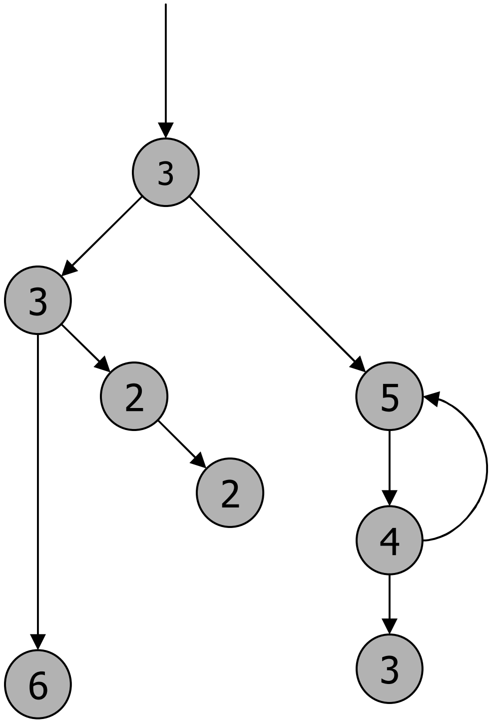
Safety: Linked Stacks

- 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

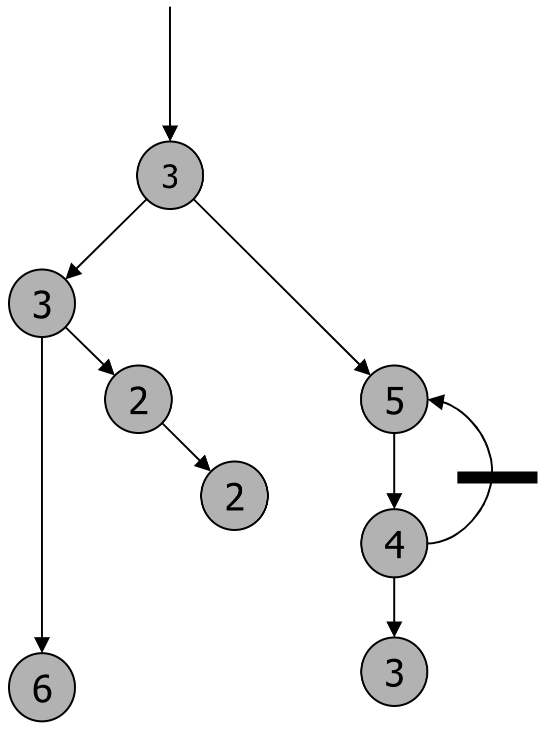
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

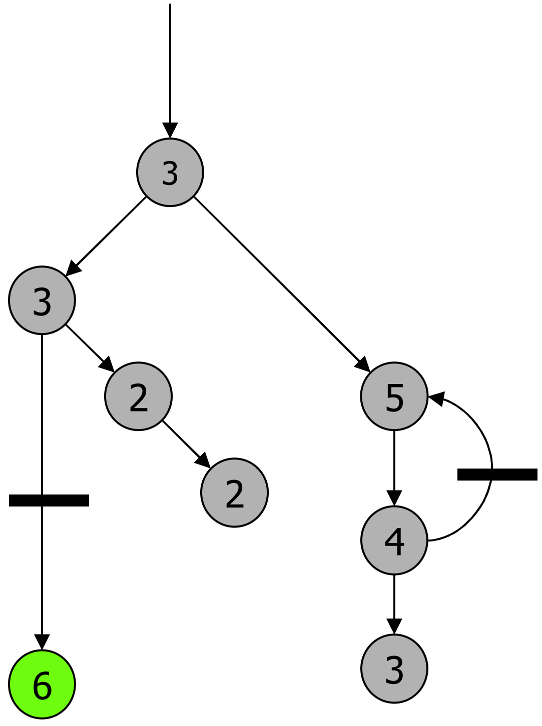
- Parameters
    - *MaxPath*
    - *MinChunk*
- Steps
    - Break cycles
    - Trace back
- Special Cases
    - Function pointers
    - External calls
    - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

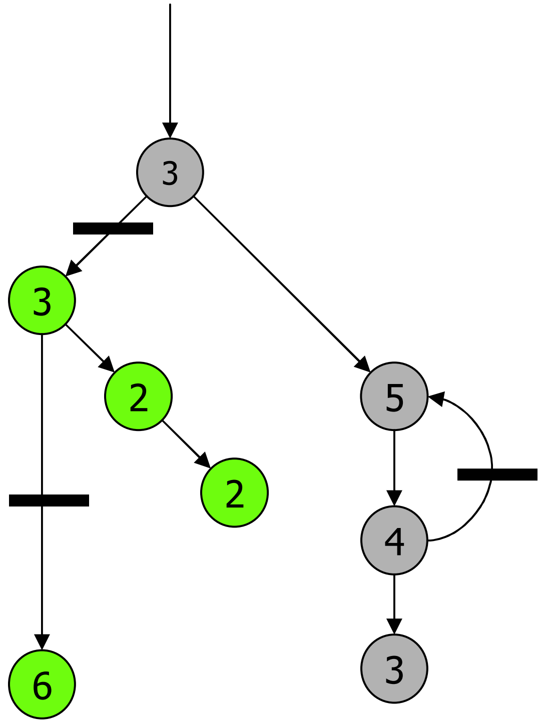
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

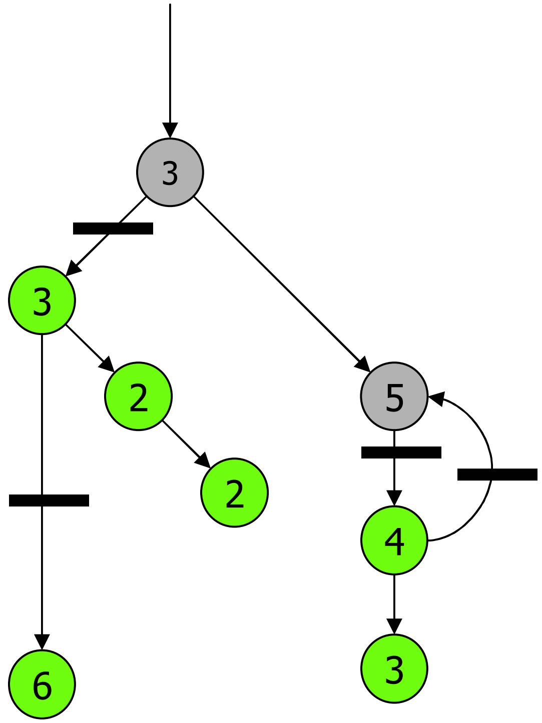
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

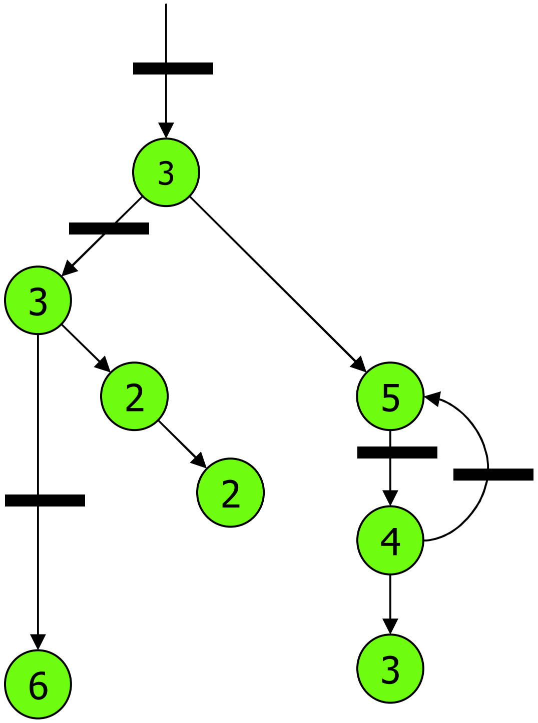
- Parameters
  - *MaxPath*
  - *MinChunk*
- Steps
  - Break cycles
  - Trace back
- Special Cases
  - Function pointers
  - External calls
  - Use large stack



MaxPath = 8

Linked Stacks: Algorithm

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

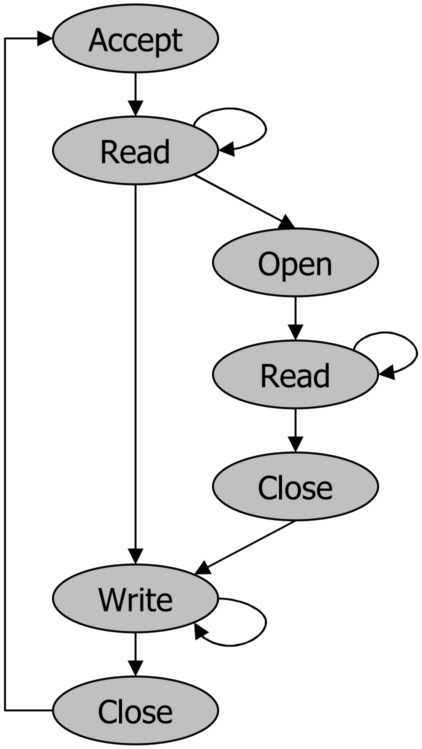


MaxPath = 8

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

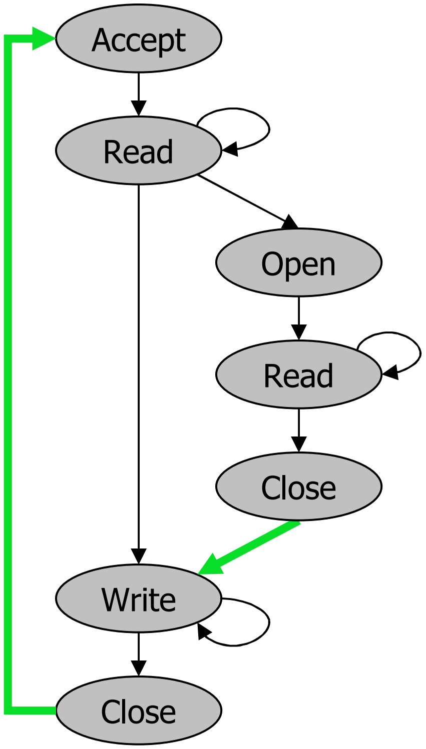
Web Server



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

Web Server



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**

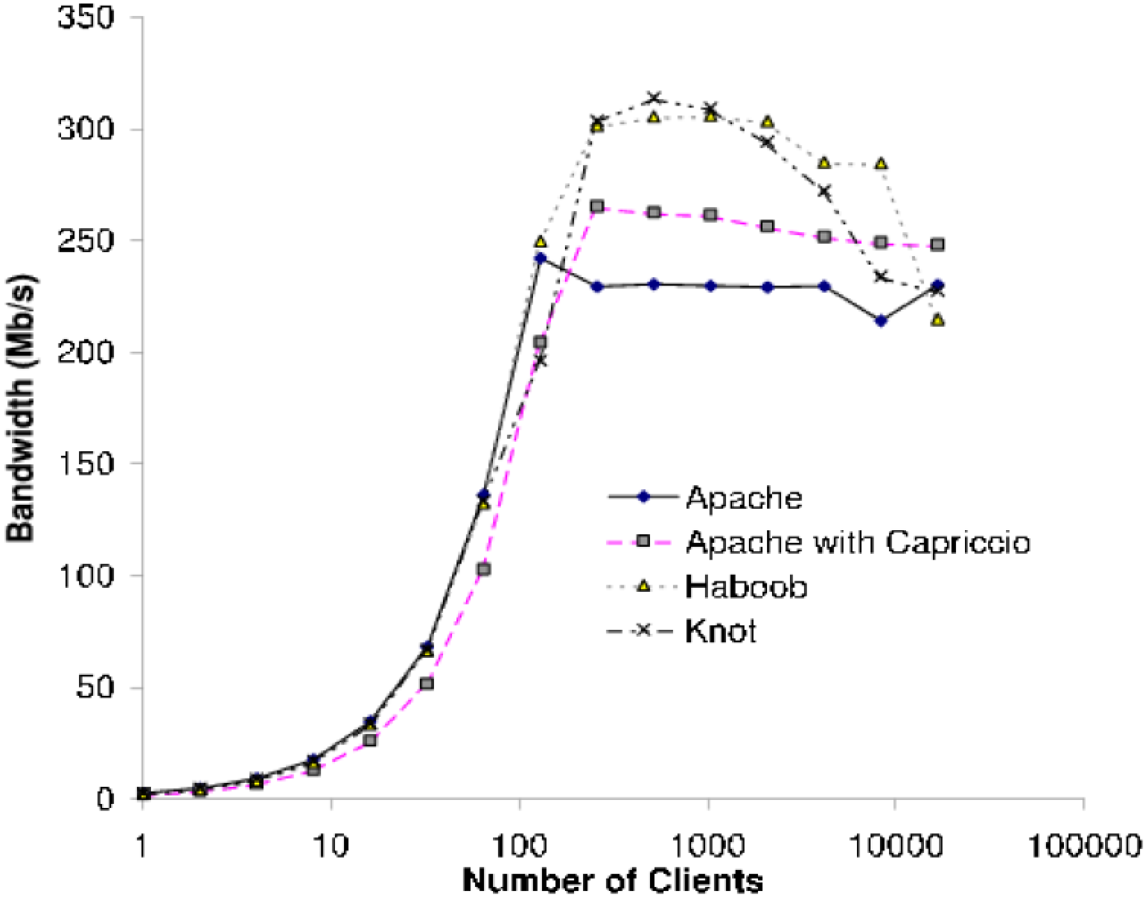
Time of thread operations (microseconds)

- 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

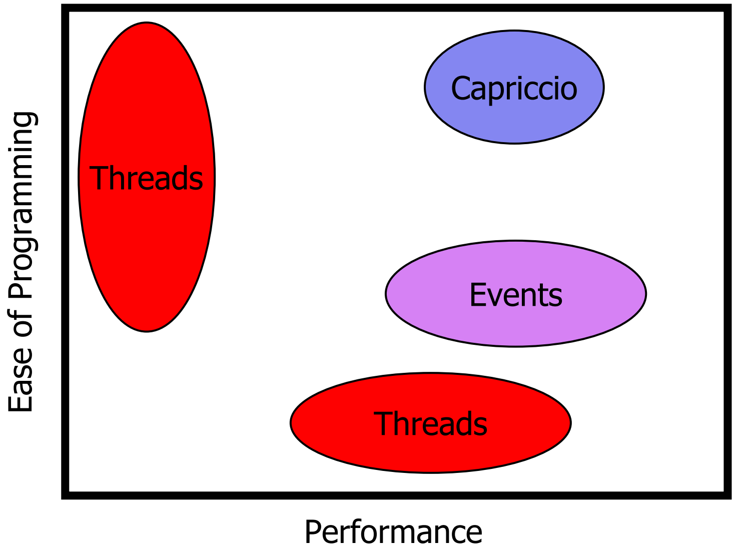


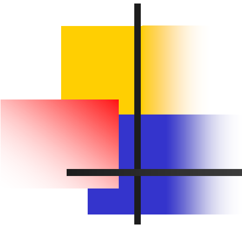
The Future:
Compiler-Runtime Integration

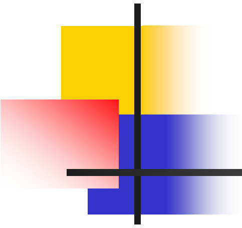
- Insight
  - Automate things event programmers do by hand
  - Additional analysis for other things
- Specific targets
  - Live state management
  - Synchronization
  - Static blocking graph
- Improve performance *and* decrease complexity

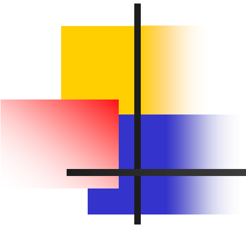
Conclusions

- Threads > Events
    - Equivalent performance
    - Reduced complexity
- Capriccio *simplifies* concurrency
    - Scalable & high performance
    - *Control* over concurrency model
        - Stack safety
        - Resource-aware scheduling
        - Enables compiler support, invariants
- Themes
    - User-level threads are key
    - Compiler-runtime integration very promising

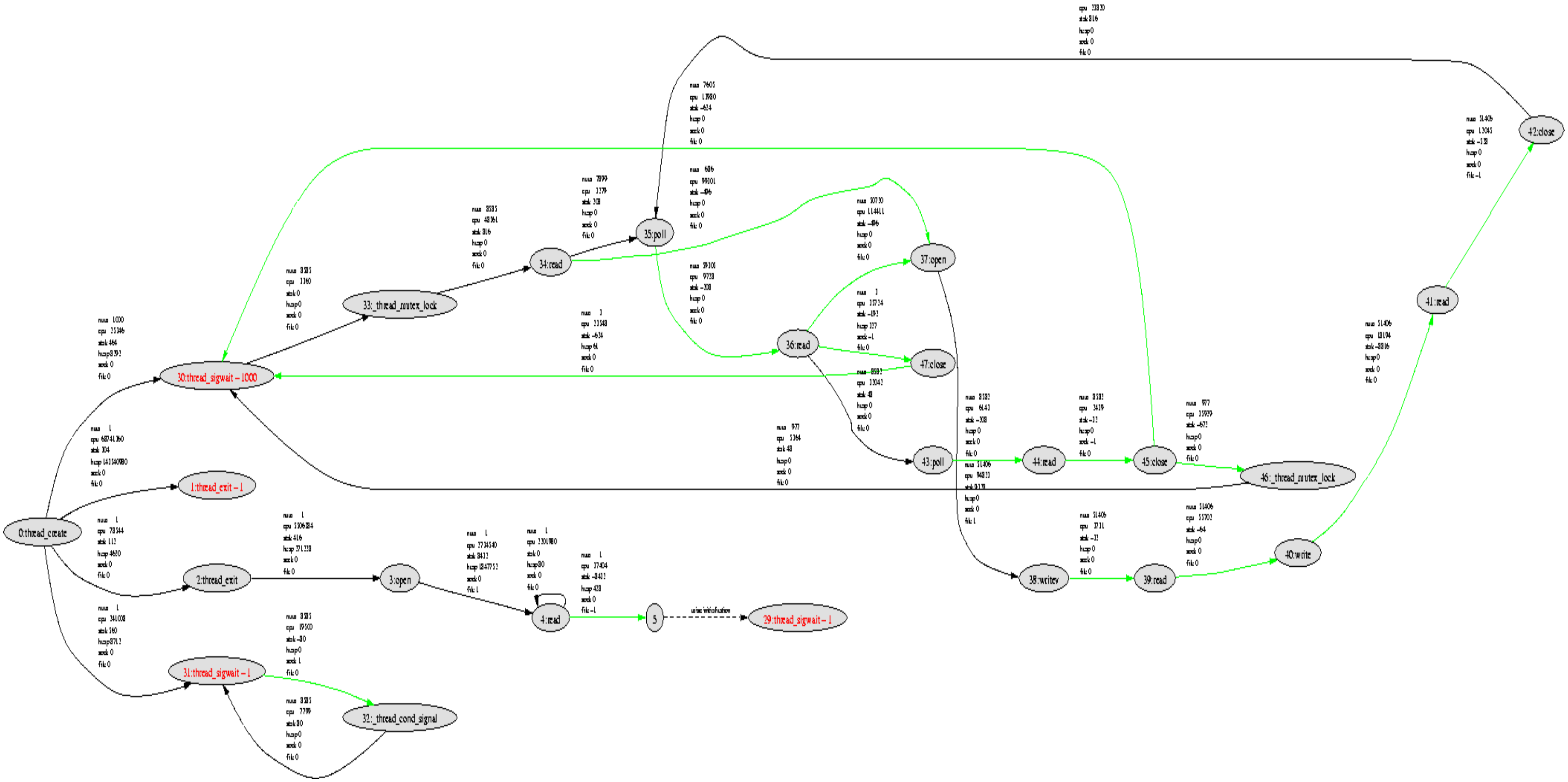


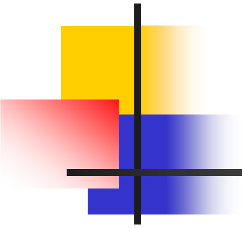




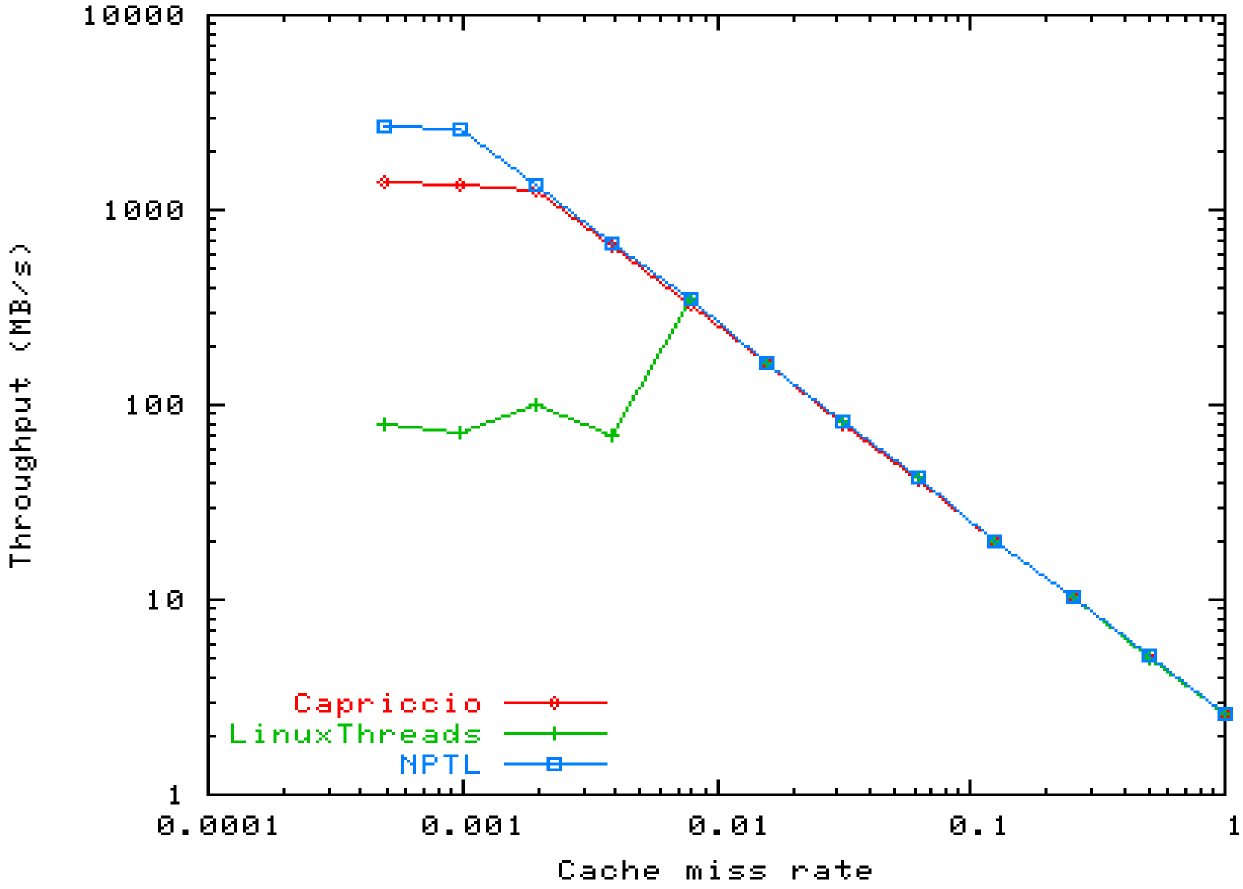


Apache Blocking Graph

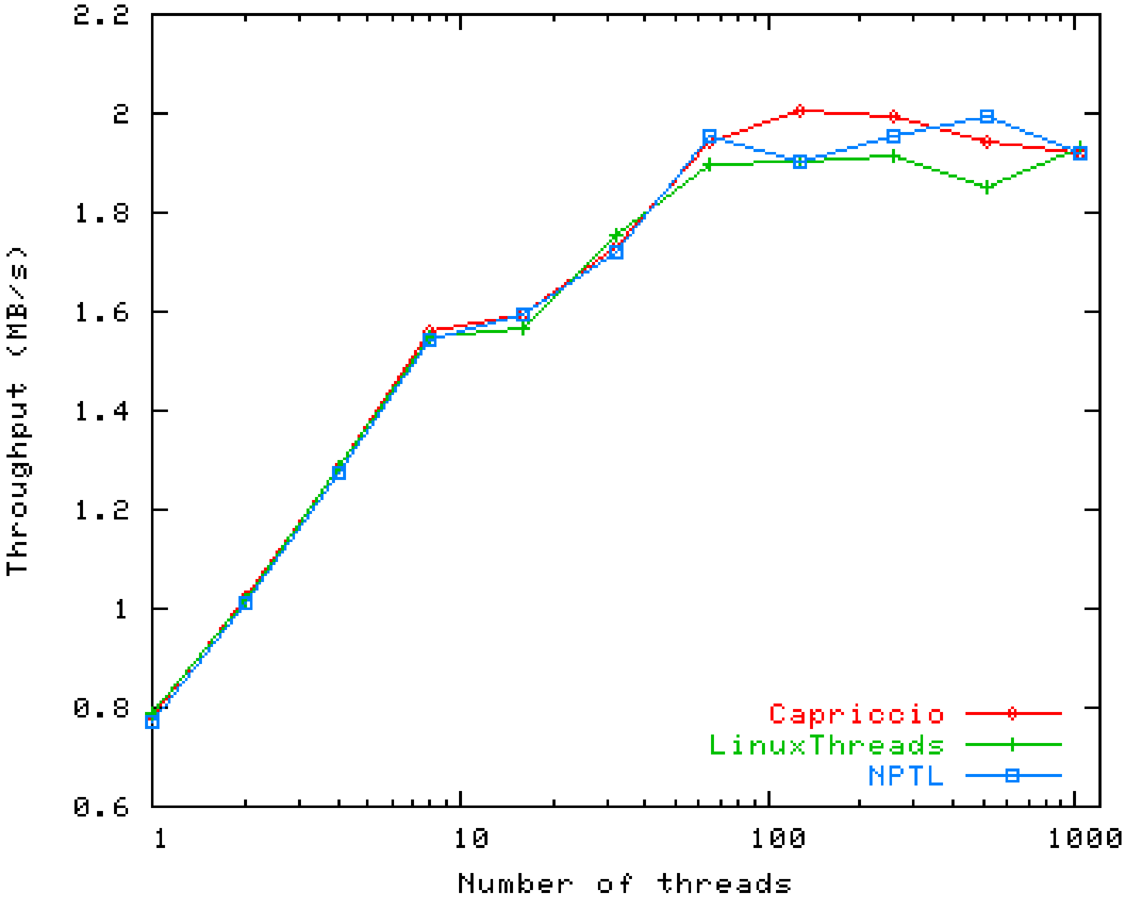




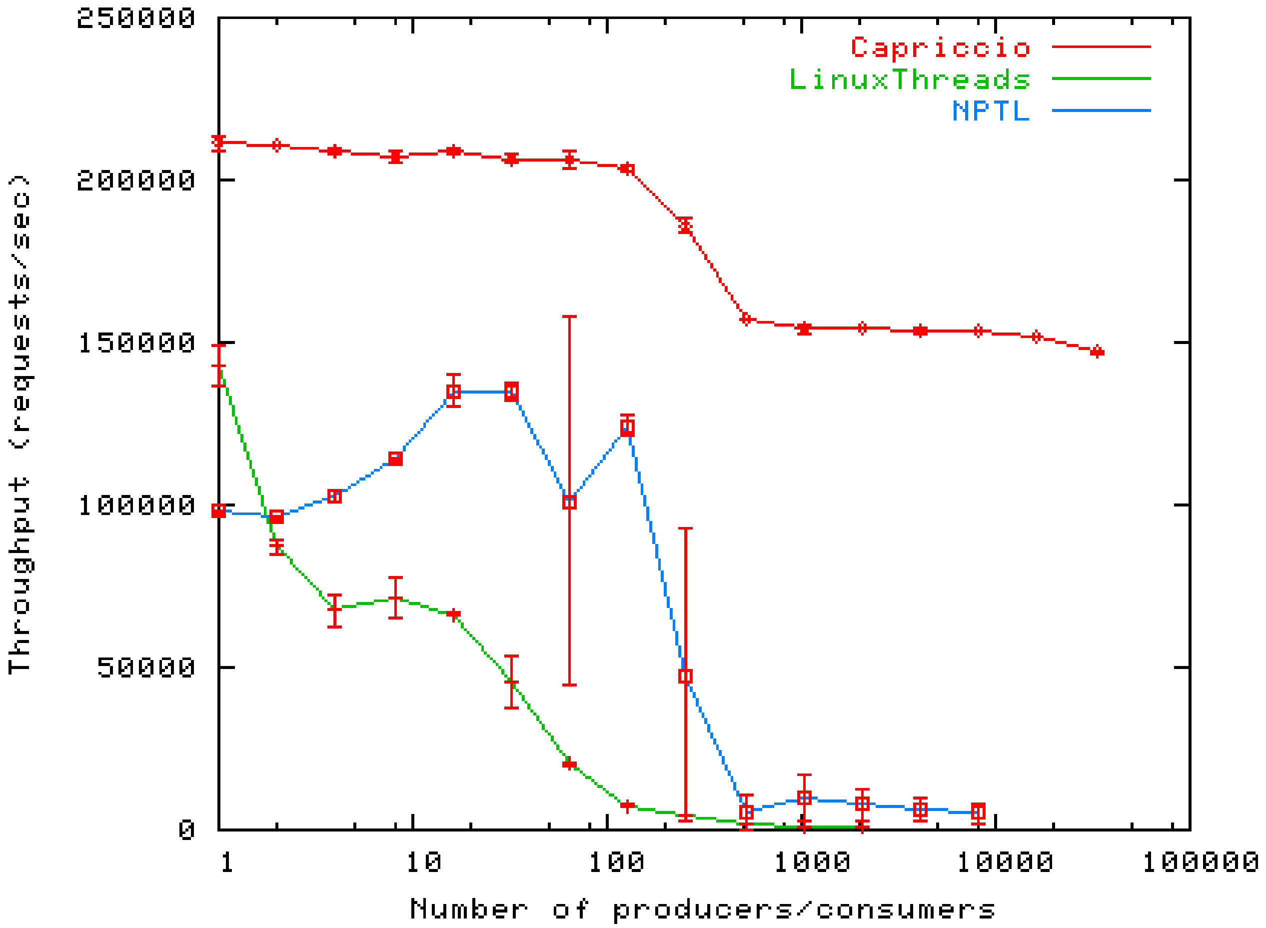
Microbenchmark: Buffer Cache



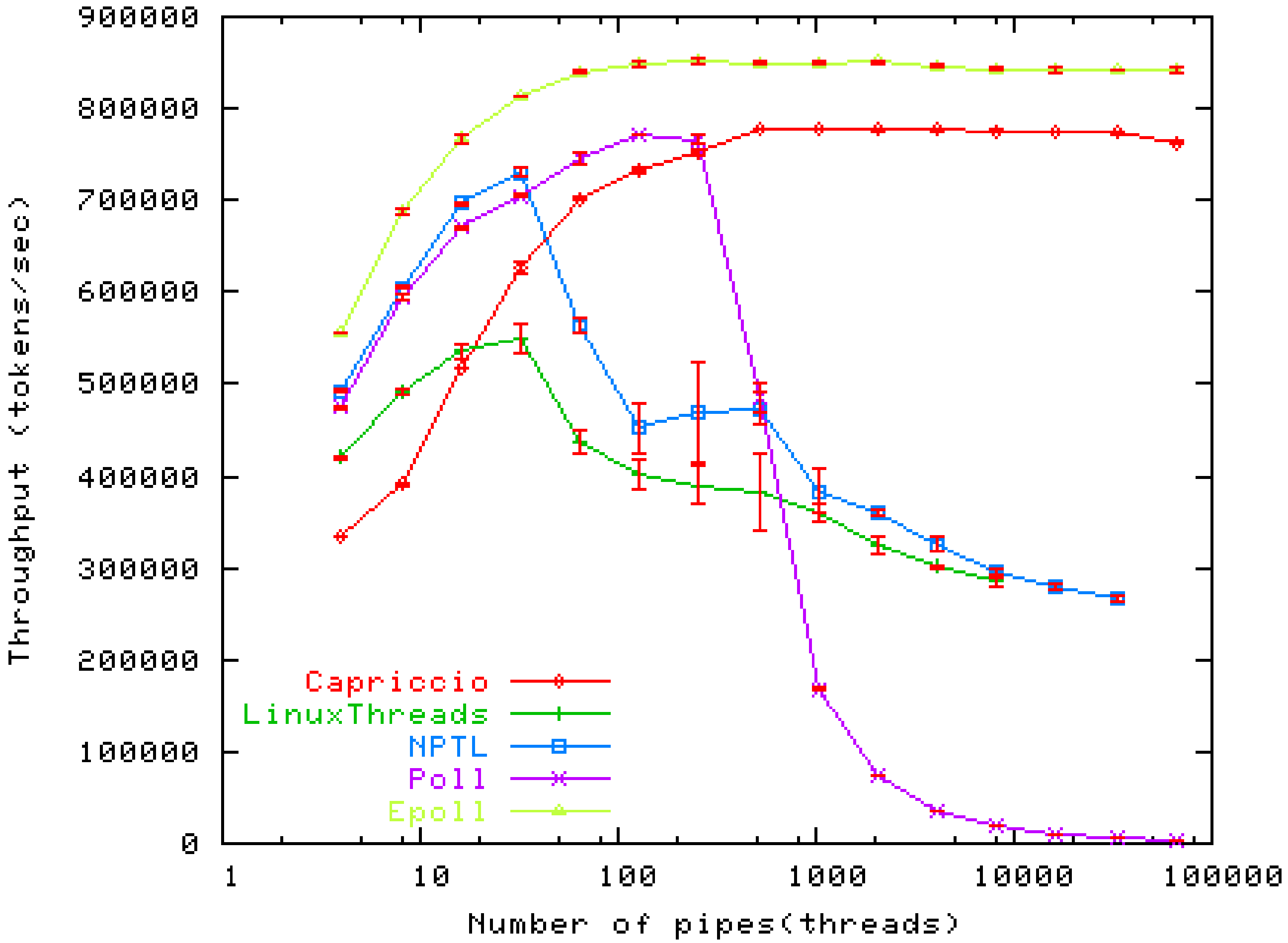
Microbenchmark: Disk I/O



Microbenchmark: Producer / Consumer



Microbenchmark: Pipe Test

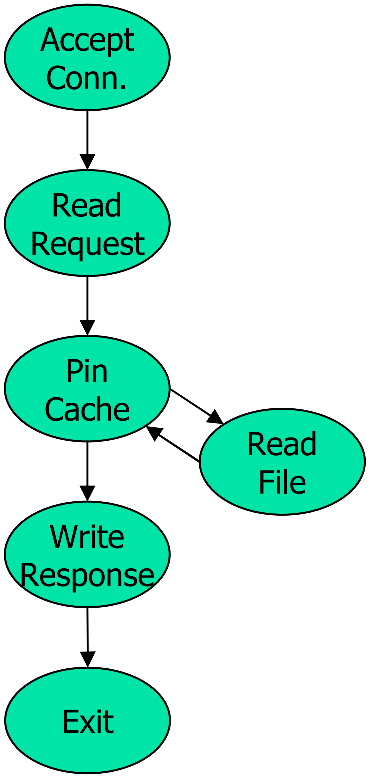


Threads v.s. Events: The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

Threads	*Events*
Monitors	Event handler & queue
Exported functions	Events accepted
Call/return and fork/join	Send message / await reply
Wait on condition variable	Wait for new messages

Web Server

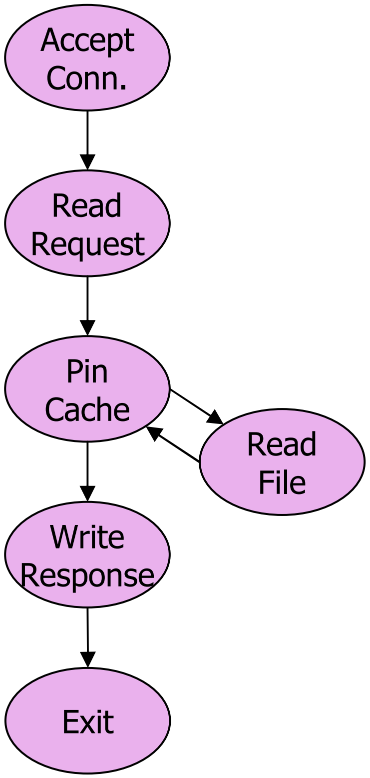


Threads v.s. Events: The Duality Argument

- General assumption: follow “good practices”
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

Threads	*Events*
Monitors	Event handler & queue
Exported functions	Events accepted
Call/return and fork/join	Send message / await reply
Wait on condition variable	Wait for new messages

Web Server

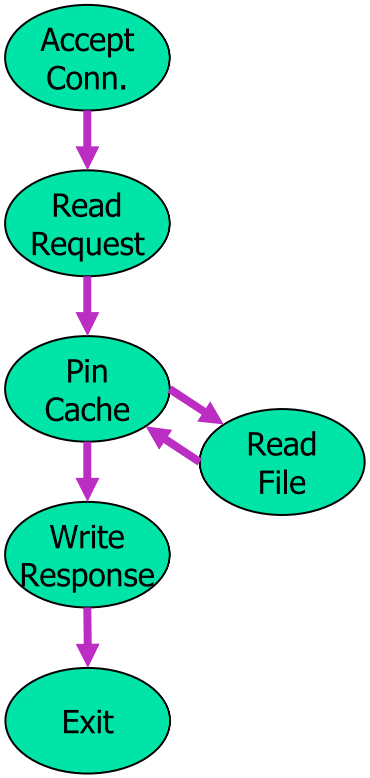


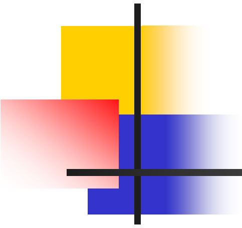
Threads v.s. Events: The Duality Argument

- General assumption: follow "good practices"
- Observations
  - Major concepts are analogous
  - Program structure is similar
  - Performance should be similar
    - Given good implementations!

Threads	*Events*
Monitors	Event handler & queue
Exported functions	Events accepted
Call/return and fork/join	Send message / await reply
Wait on condition variable	Wait for new messages

Web Server



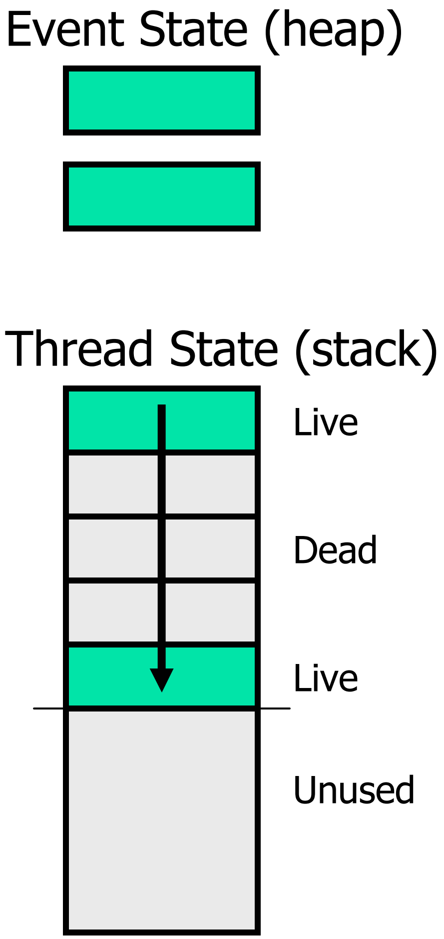


Threads v.s. Events: Can Threads Outperform Events?

- Function pointers & dynamic dispatch
  - Limit compiler optimizations
  - Hurt branch prediction & I-cache locality
- More context switches with events?
  - Example: Haboob does 6x more than Knot
  - Natural result of queues
- More investigation needed!

Threading Criticism: Live State Management

- *Criticism: Stacks are bad for live state*
- Response
  - Fix with compiler help
  - Stack overflow vs. wasted space
    - Dynamically link stack frames
  - Retain dead state
    - Static lifetime analysis
    - Plan arrangement of stack
    - Put some data on heap
    - Pop stack before tail calls
  - Encourage inefficiency
    - Warn about inefficiency



Threading Criticism: Control Flow

- *Criticism: Threads have restricted control flow*
- Response
  - Programmers use simple patterns
    - Call / return
    - Parallel calls
    - Pipelines
  - Complicated patterns are unnatural
    - Hard to understand
    - Likely to cause bugs