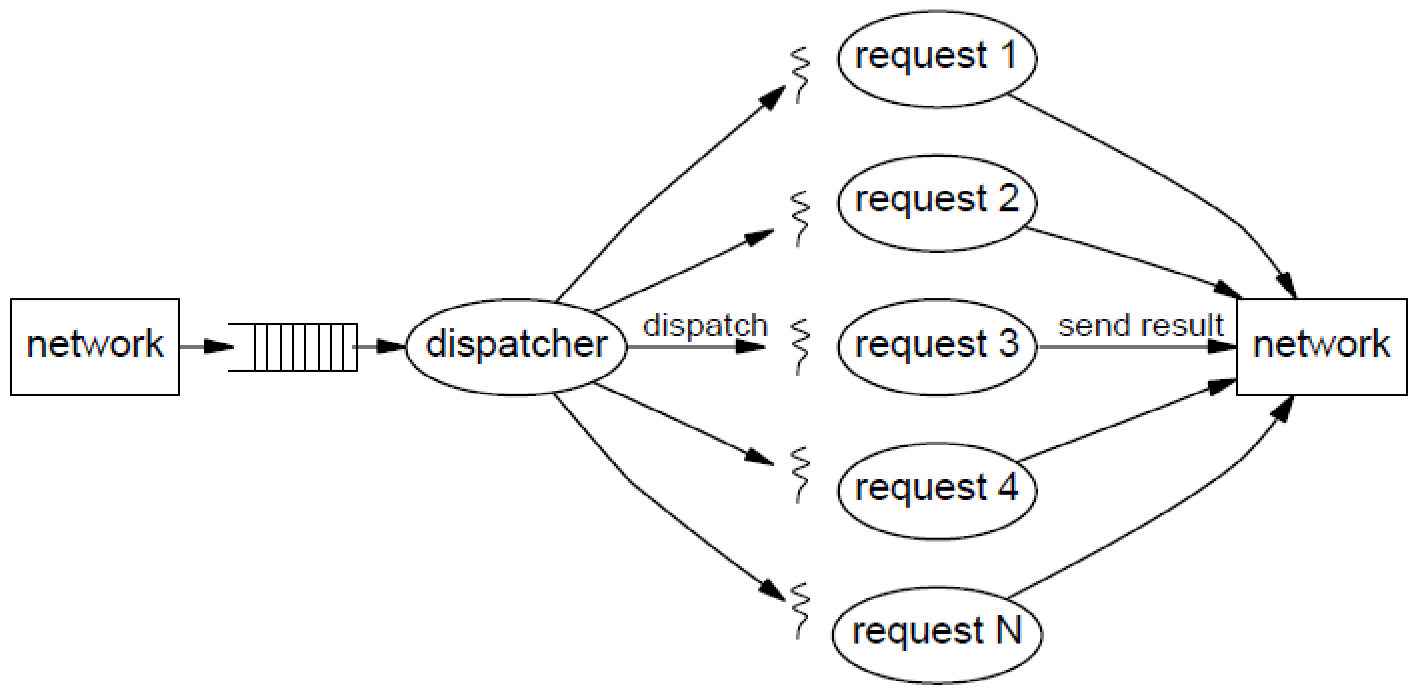
Capriccio : Scalable Threads for Internet Services

- Ron von Behren &et al
- University of California , Berkeley.

Presented By:
Rajesh Subbiah

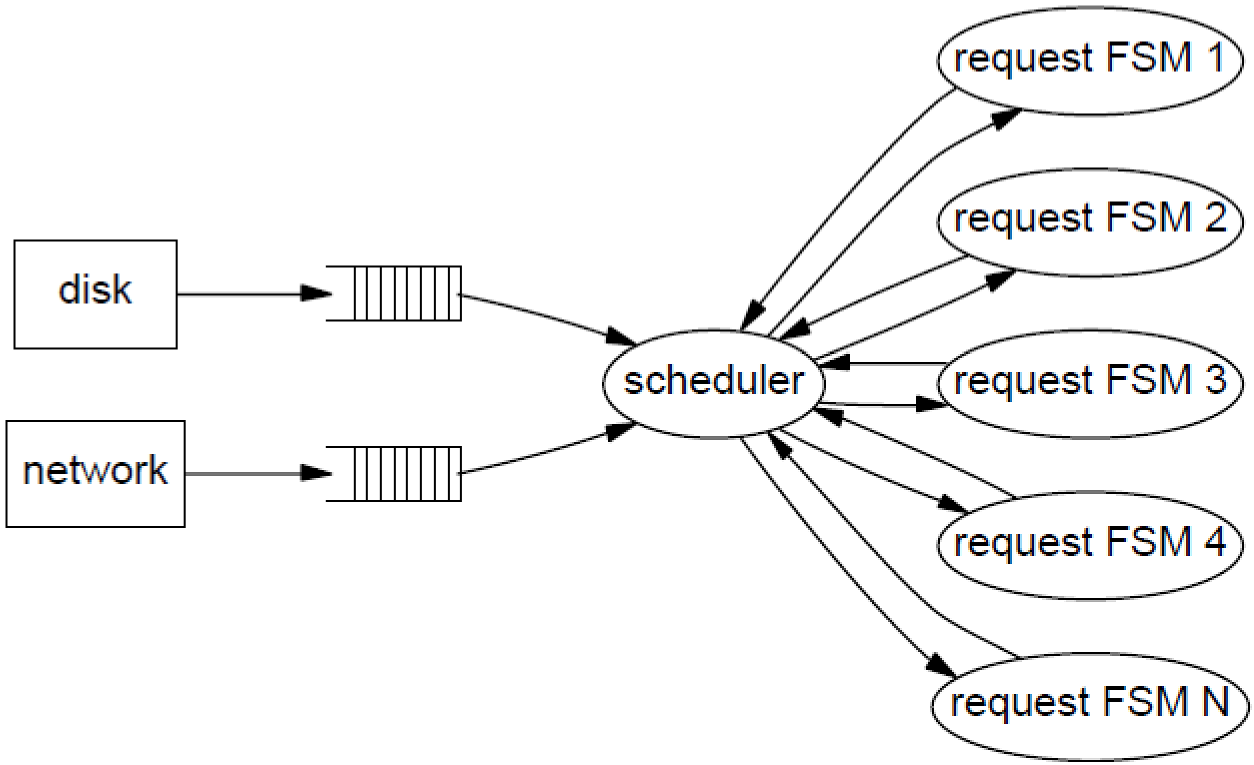
Background



Each incoming request is dispatched to a separate thread

Note: Diagram taken from SEDA paper, SOSP 2001

Background



The main thread processes incoming events & executes the finite state machines

Note: Diagram taken from SEDA paper, SOSP 2001

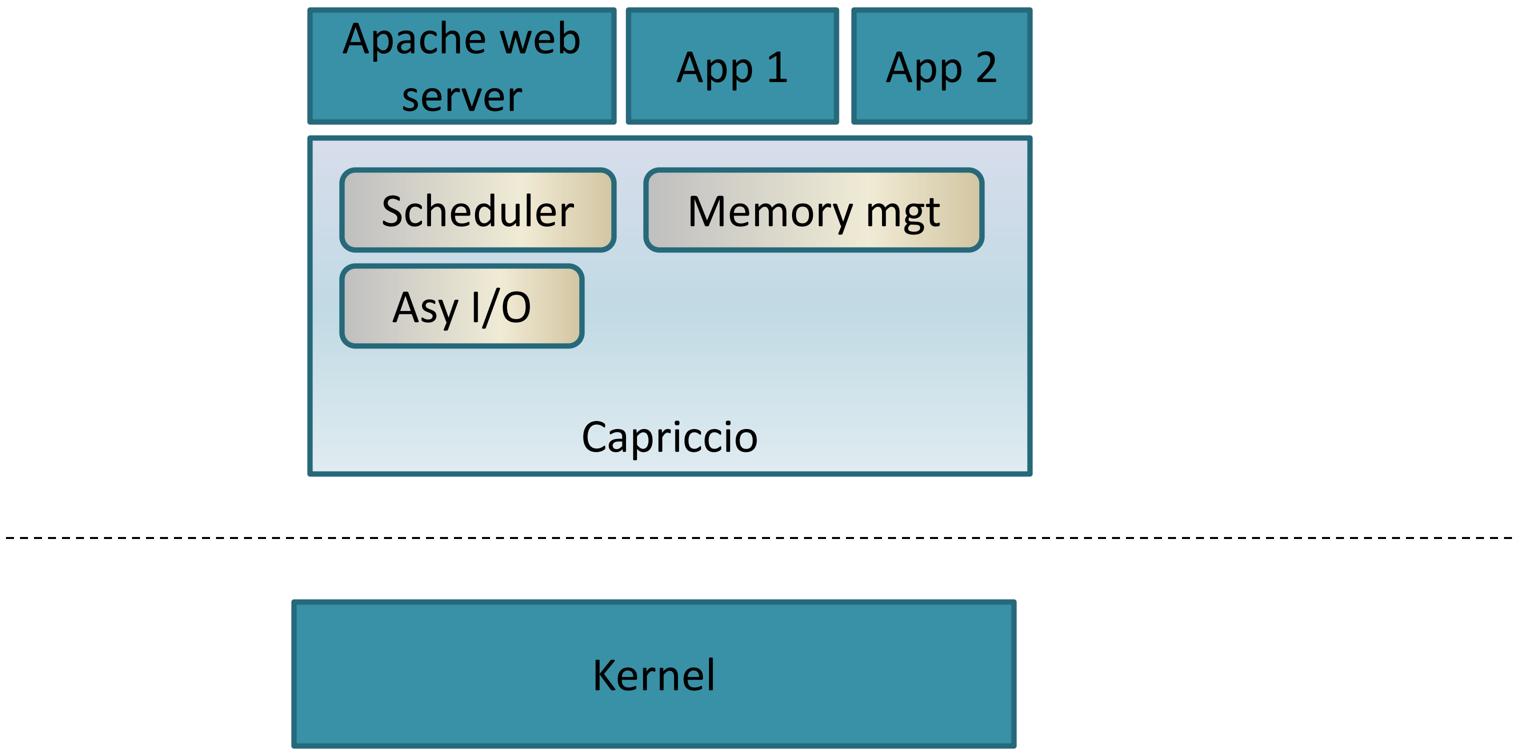
Motivation

- Internet services have increasing scalability demands
  - Need simplified & user friendly programming model
- Available design approaches
  - Event model vs. Thread model
- Problems
  - Event model
    - Hides the control flow
    - Difficult to debug
  - Thread model
    - Consume too much stack space
    - No scalability
    - No resource aware scheduling

Capriccio: Design Objectives

- Use existing threads APIs
- Improve scalability
  - One thread – one connection for Internet servers
- Do efficient memory management
- Perform resource aware scheduling

Capriccio Thread Package: Architecture



Capriccio Thread Package Advantages

- Flexible to address application specific needs
  - Creates one level of indirection between application & the kernel
  - Easily scales up to 100k threads
- Efficient memory management
  - Using compiler analysis
  - By Implementing linked stack
- Efficient resource aware scheduling
  - By generating blocking graphs

Capriccio: Implementation

- Context switches
  - Uses Toering's coroutine library
  - Threads voluntarily yield
- I/O
  - Uses latest Linux asynchronous I/O mechanisms
    - epoll and AIO
  - Increases over head
- Scheduling
  - Resource based scheduling
- Synchronization
  - Takes advantage of co-operative scheduling
  - Uses simple check like boolean locked/unlocked flag
- Efficiency
  - All O(1) expect for sleep queue

Comparison Of Different Thread Packages

	Capriccio	Capriccio_notrace	Linux Thread	NPTL
Thread creation	21.5	21.5	37.9	17.7
Thread context switch	0.56	0.24	0.71	0.65
Uncontended mutex lock	0.04	0.04	0.14	0.15

Latencies (in micro seconds) of thread primitives for different thread packages

- 2X 2.4 GHz Xeon processors, 1 GB of memory.
- 2X 10k RPM SCSI Ultra II hard drives
- 3 Gigabit Ethernet interfaces.
- Operating System: Linux 2.5.70 (epoll supported)

Capriccio: Memory Management

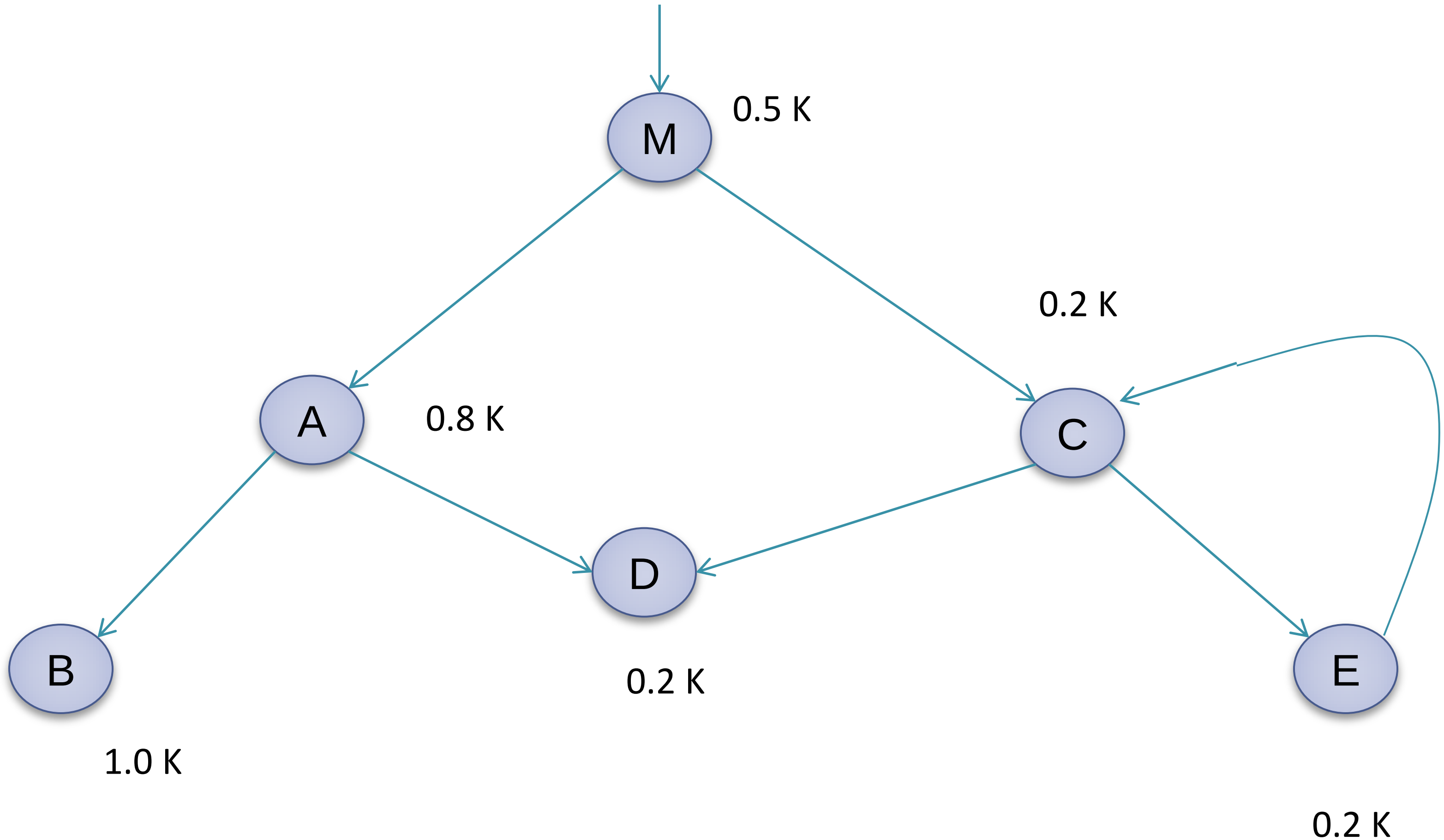
- Does a complier analysis
  - Generates weighted call graph
- Linked stack management
  - Use dynamic allocation policy.
  - Allocate memory chunks on demand
  - Problems ?

Example

```
main ()
{<data type declaration>
function_A(<paramlist>);
function_C(<paramlist>);
}

function_A(<paramlist>)
{<data type declaration>
 function_B(<paramlist>;
function_D(<paramlist>);
}

function_B(<paramlist>)
{<data type declaration>
}

function_D(<paramlist>)
{<data type declaration>
}
```

```
function_C(<paramlist>)
{<data type declaration>
 function_E(<paramlist>;
 function_D(<paramlist>;

}

function_E(<paramlist>)
{<data type declaration>
 function_C(<paramlist>
}
```

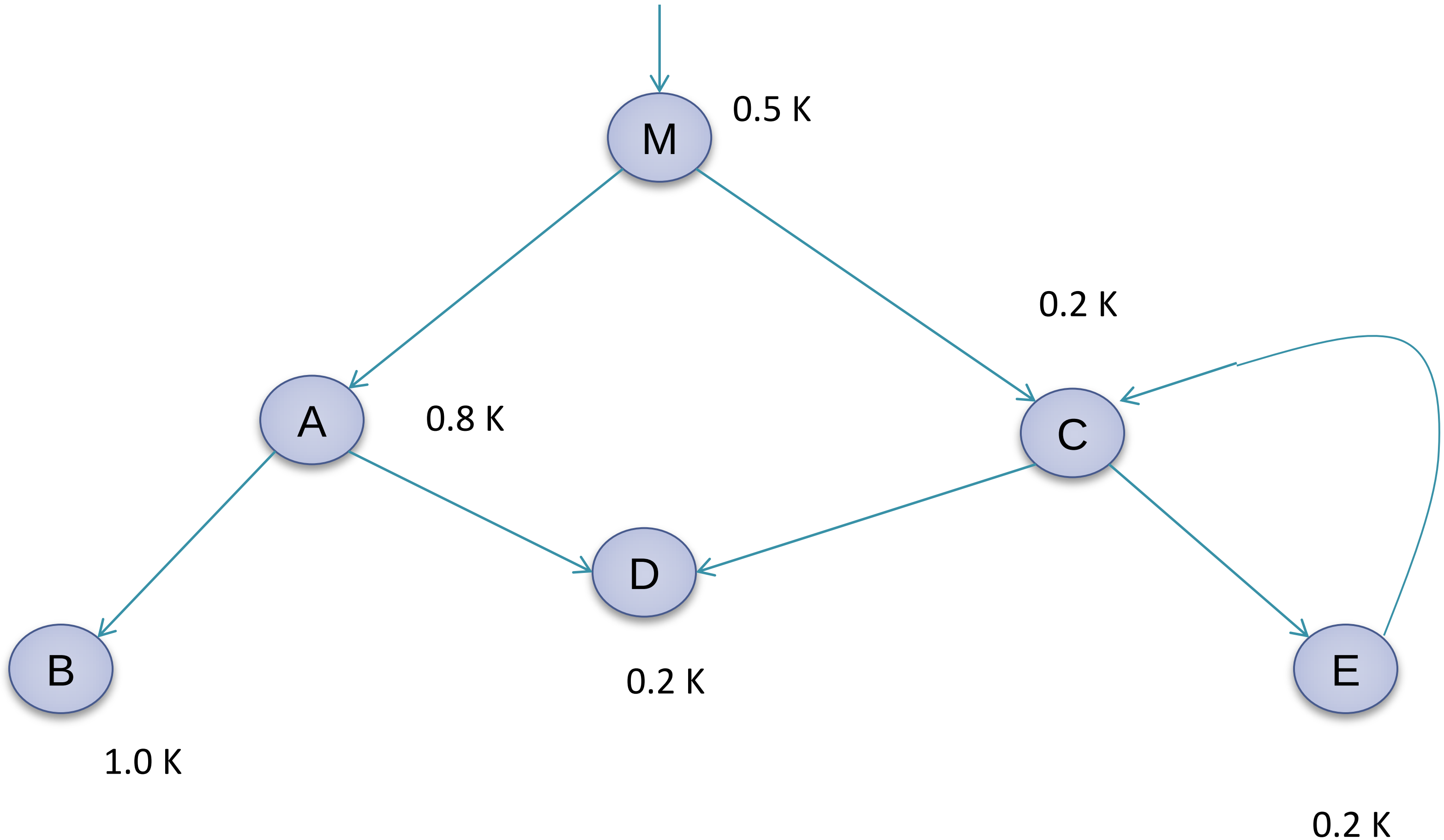
Weighted Call Graph



Weighted Call Graph

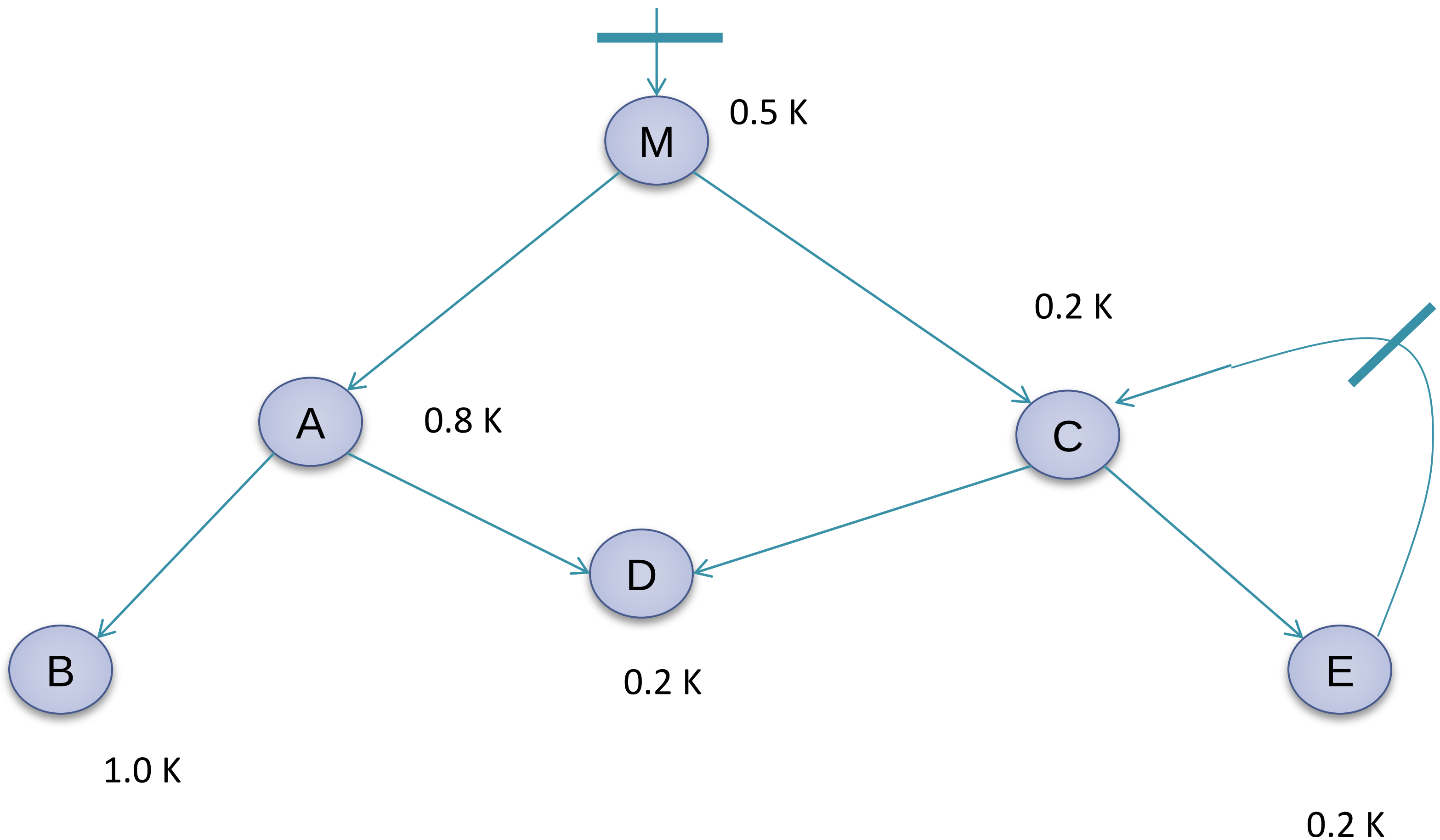
- Each function is represented as a node
  - Weighted by the max stack size it need for execution
- Each edge represents a direct function call
- Checkpoints
  - Inserted at call sites at compile time.
  - Checks whether there is enough stack size left for reaching next checkpoint.
  - If there is no enough stack space ; it allocates a stack chunk.
  - Problem ?
  - Where we should insert checkpoints ?

Weighted Call Graph



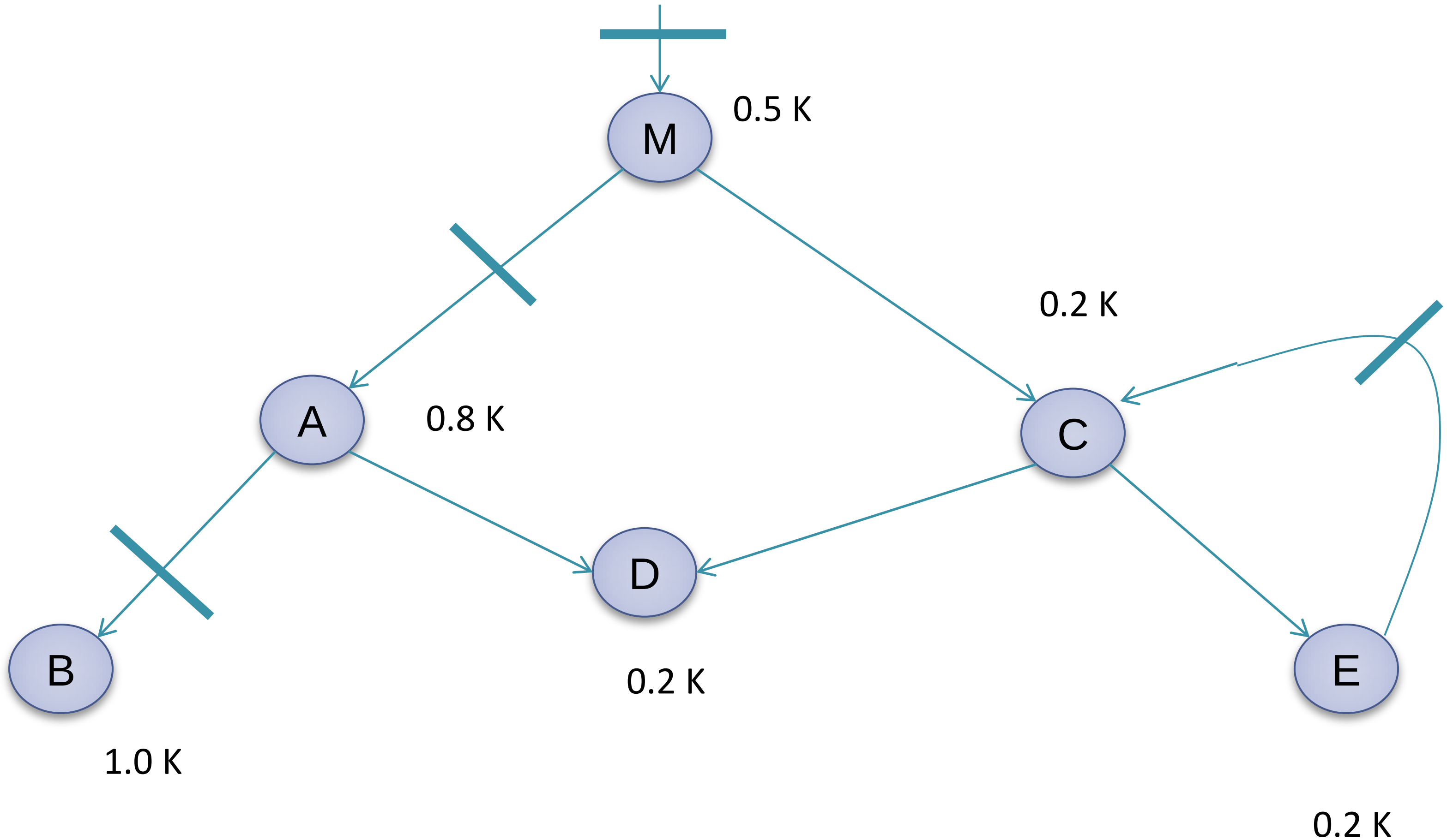
Insert one check point in every cycle back edge

Weighted Call Graph

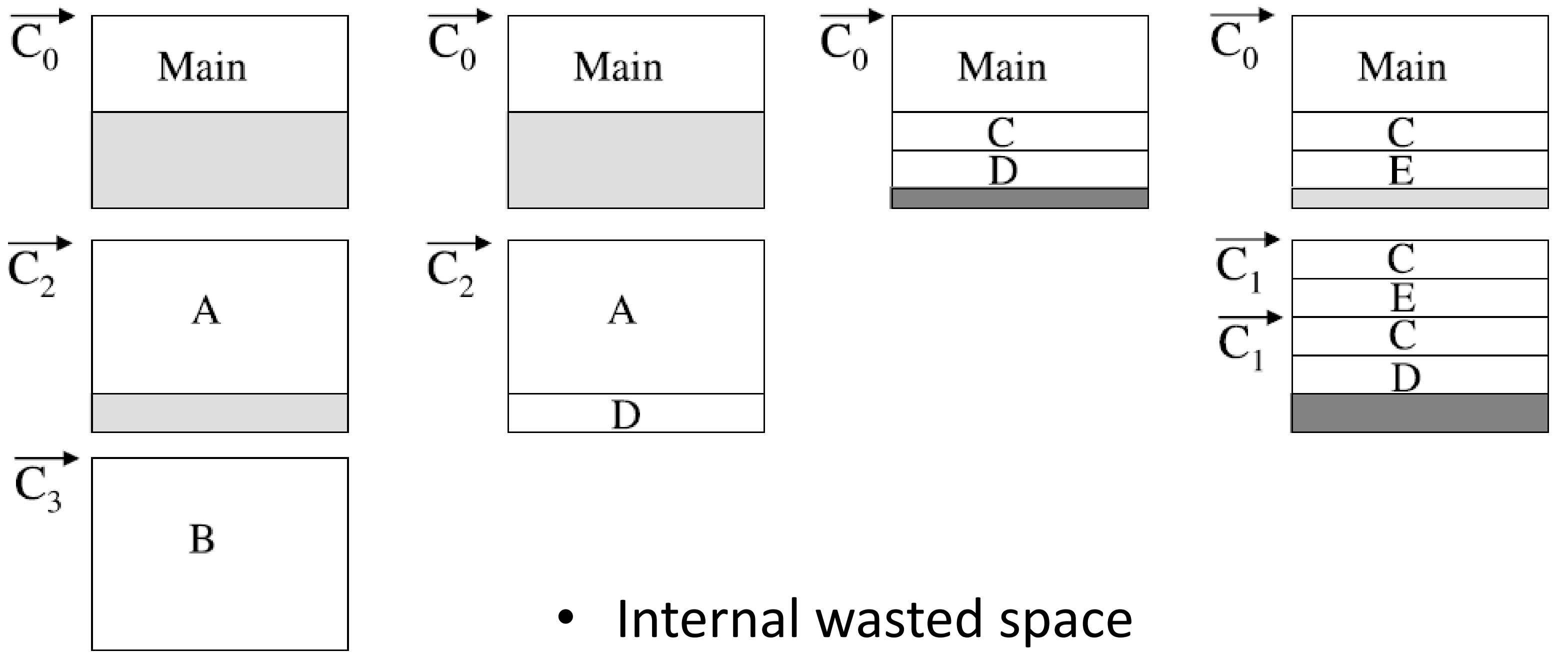


- Use Bottom up approach & MaxPath = 1.0 K
- Check longest path from node to checkpoint, if MaxPath limit is exceeded, add checkpoint

Weighted Call Graph



Memory Allocation - Runtime

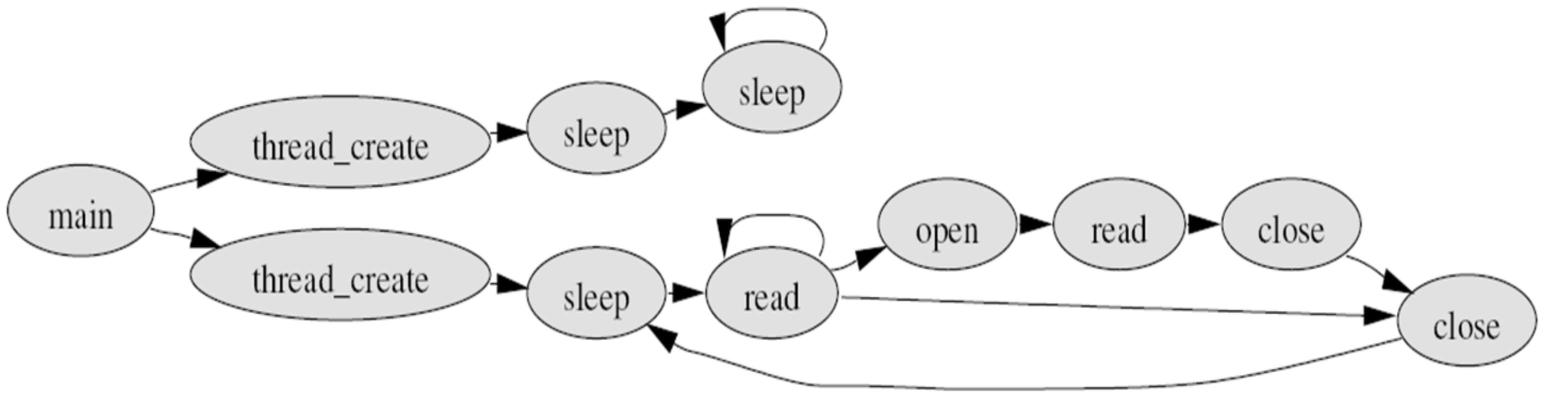


- Internal wasted space
  - MaxPath
- External wasted space
  - MinChunk

Resource Aware Scheduling

- Application is viewed as a sequence of stages separated by blocking points
- Uses blocking graph
  - It is generated at run-time.
  - Each node is location in program that is blocked
  - Node is composed of call chain used to reach blocking point
  - Resource usage are annotated.
    - Resource usage is monitored & scheduling is done based on the resource usage patterns.

Blocking Graph

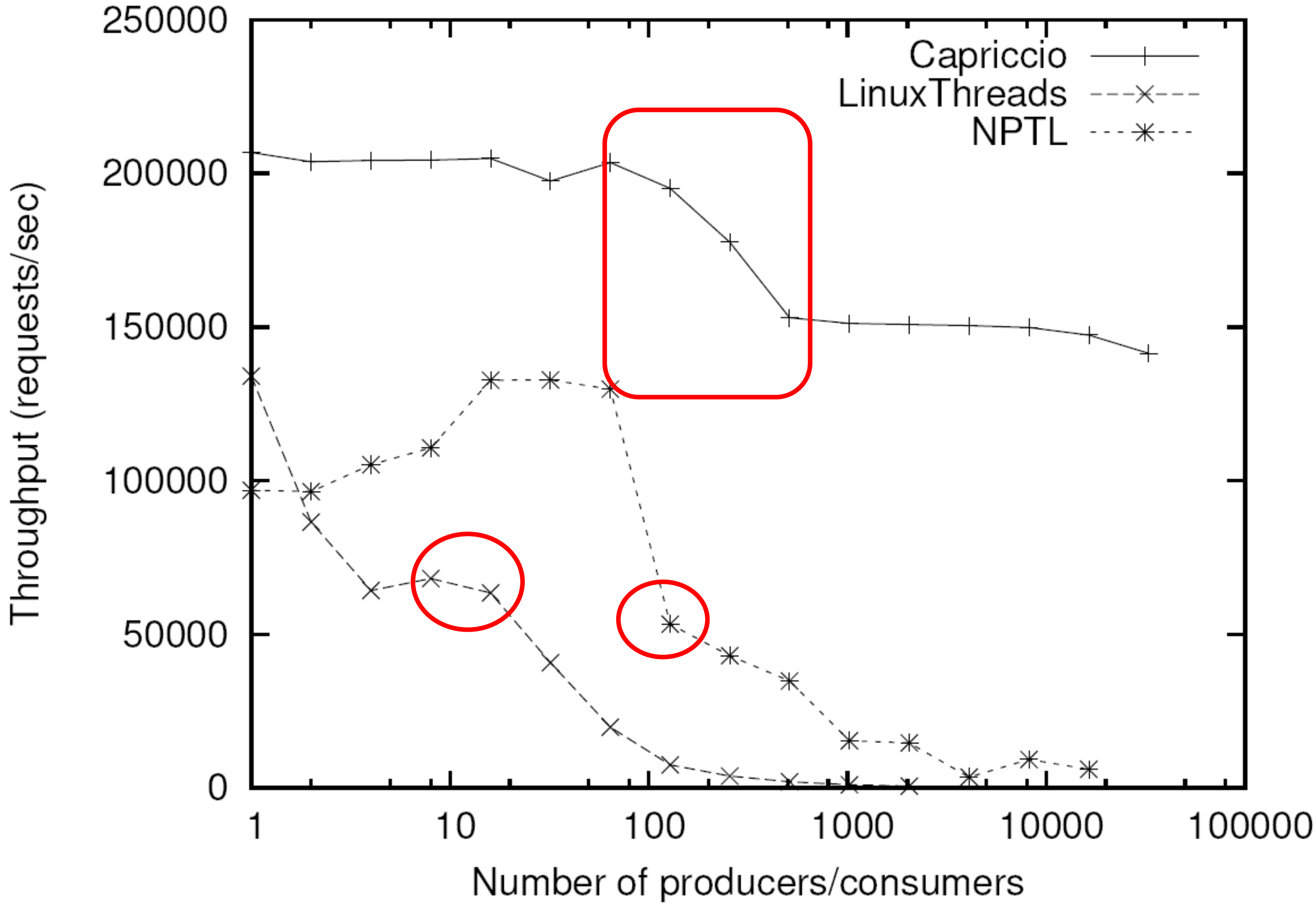


Pitfalls

- Resource’s maximum capacity is difficult to determine.
- It is difficult to detect thrashing
  - Involves system overhead.
- Non yielding threads lead to unfairness and starvation

Experiments & Results

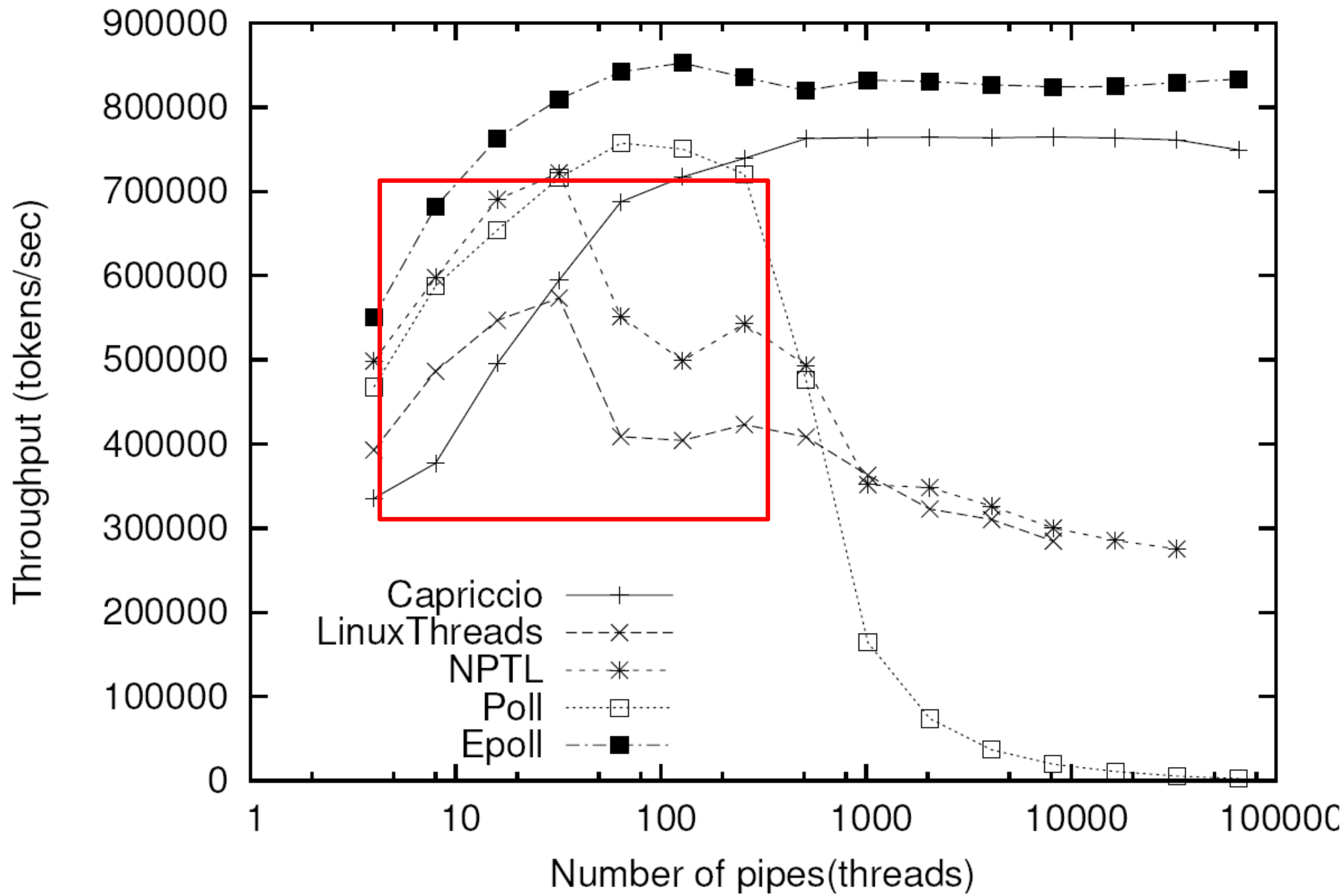
- Thread Scalability
  - Producer & Consumer
- I/O Performance test
- Web Server tests
  - 4*500 MHz Pentium server with 2GB memory
  - Linux 2.4.20
    - No use of epoll or Linux AIO



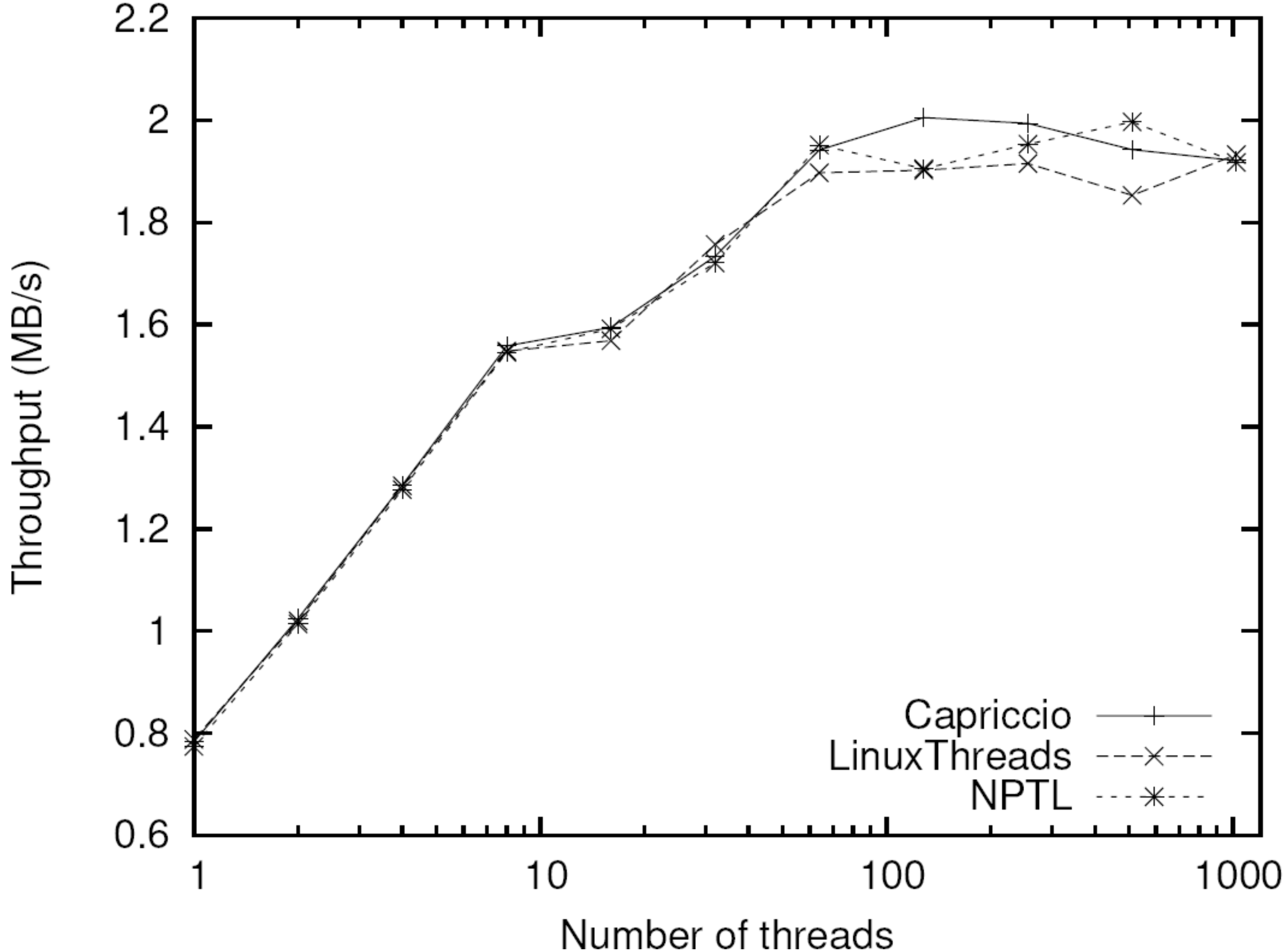
Drop between 100 and 1000 due to cache footprint

I/O Performance

- Concurrently passing 12 byte token to fixed number of pipes
- Disk head scheduling
  - A number of threads perform random 4 KB reads from a 1 GB file
- Disk I/O through buffer cache
  - 200 threads reading with a fixed miss rate



When concurrency is low, performance also decreases

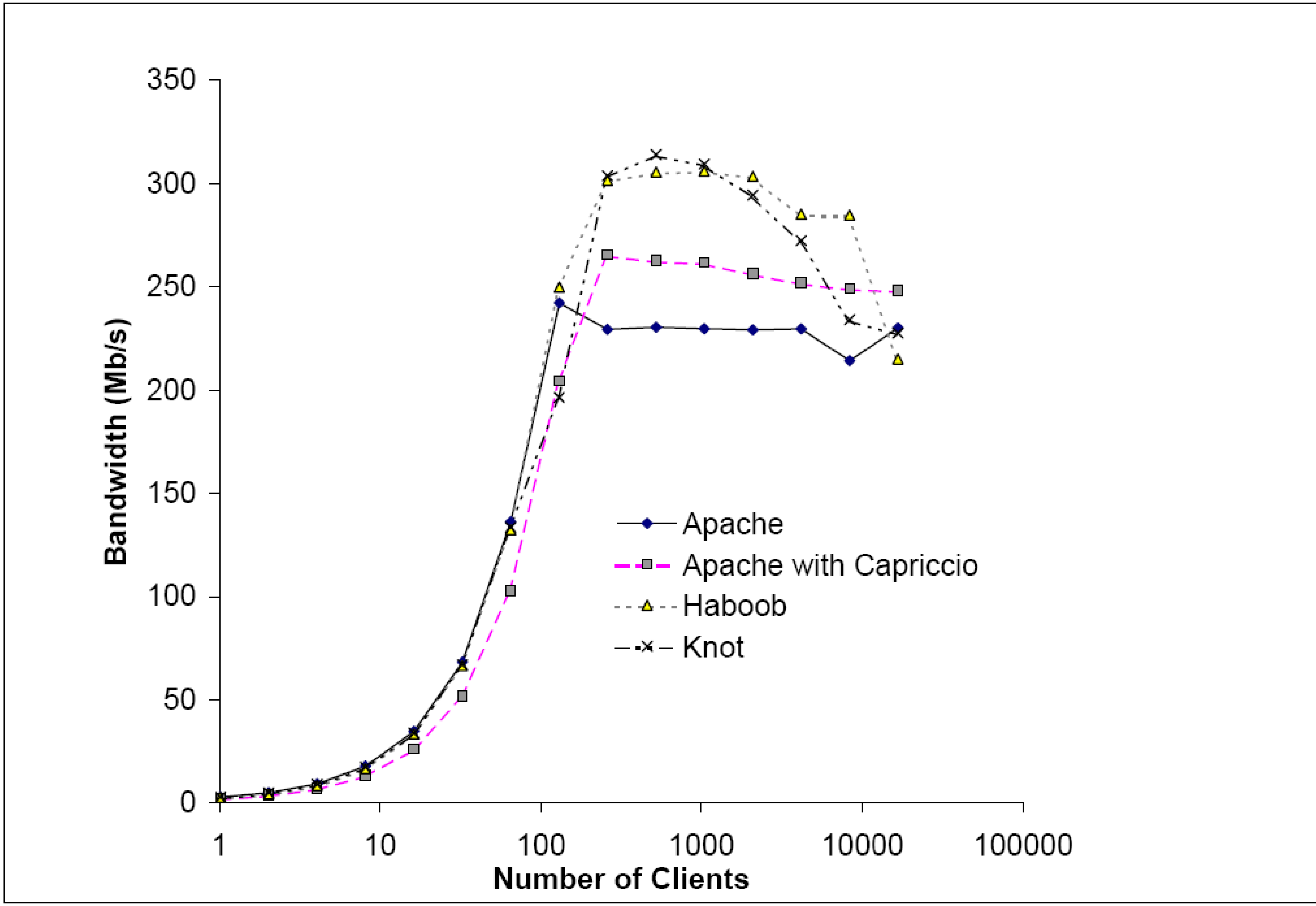


Benefits of disk head scheduling

Web Server Performance Test Results

- Apache web server performance improved by 15%
- Knot’s performance matched the performance of event-based Haboob webserver

Web Server Performance Test Results



Conclusion

- Capriccio illustrates that using user-level threads we can get
  - High scalability
  - Efficient memory/stack management
  - Resource based scheduling
- Drawbacks
  - Lack of multi-cpu support

Future Work

- Extending Capriccio to multi processor environment.
- Producing profiling tools to tune stack parameters according to the application needs

Critique

- Capriccio thread library improves the scalability , memory management & thread scheduling
  - The techniques used by Capriccio are novel

- Presently there is no support for Capriccio thread library