Universit□t
Salzburg

Capriccio:

Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer

SOFTWARE RESEARCH LAB

What is it?

- Lightweight POSIX threading package
- High performance upto 100k threads
- Flexible to address application specific needs
- Compiler assisted performance increase

Why?

- Internet services have increasing scalability demands
- The Hardware is fast enough but the Software is not using it efficiently
- Event based approaches are hard to understand and maintain
- Current threading packages do no scale well
- Threads consume too much memory space (stack)
- One thread per connection model is not efficient with current threads

Features

- User-level implementation with
  - Cooperative scheduling
  - Asynchronous disk I/O
  - Linked stack management for reduced memory footprint
  - Resource-aware scheduling

Events vs Threads

- Events hide the logical control flow
  - May be difficult to understand
  - Programmers need to match related events and correctly save/restore context
  - Application specific optimizations that are not portable
- Threads are simpler to understand
  - Require efficient thread runtimes
  - No “stack ripping”

- C. Lauer in “On the Duality of Operating Systems Structures” states that Events and Threading systems are alike and performance is only related to hardware

Threading models

- User-level
  - Cleaner programming model
  - Decoupling from kernel
  - Portable and Flexible

- Kernel-level
  - True concurency
  - Benefit of multiprocessor architectures
  - Direct access to hardware resources

- Distribution M:N vs 1:1
  - 1:1 – easier and more efficient scheduling, improved security
  - M:N – closer to logical programming model

User-level threading advantages

- Cleaner programming model
- Decoupling of application logic and kernel threading for faster innovation
- User-level scheduling correlated with application logic
- Lightweight for kernel mode switching and kernel space usage
- Reduced overhead for thread synchronization
- Better memory management (fit application needs)
- Most management operations are O(1)
- Sleep time is O(n)

User-level threading disadvantages

- Blocking systems calls must be replaced with non-blocking constructs with equivalent functionality
- Difficult to schedule on multiprocessor systems
- Ineffective with true concurrency support from hw
- Mapping of user-level threads over kernel-level threads leads to decreased performance
- Two schedulers (kernel & user) for the same purpose
- Increased I-Cache and D-Cache footprint

User-level threading remaining issues

- Cooperative threading expected from compiler
- No preemptive scheduling
- Must be kept in sync with kernel and libraries development
- Difficult handling of precompiled libraries or static compiled applications
- Source code must be preprocessed

Approach

- User-level threading model
- Linked stack management
- Resource aware scheduling

2.4GHz Xeon, 1GB RAM, Linux 2.5.70	Capriccio	LinuxThreads	NPTL
Thread creation	21.5	21.5	17.7
Thread context switch	0.24	0.71	0.65
Uncontended mutex lock	0.04	0.14	0.15

Linked Stack Management

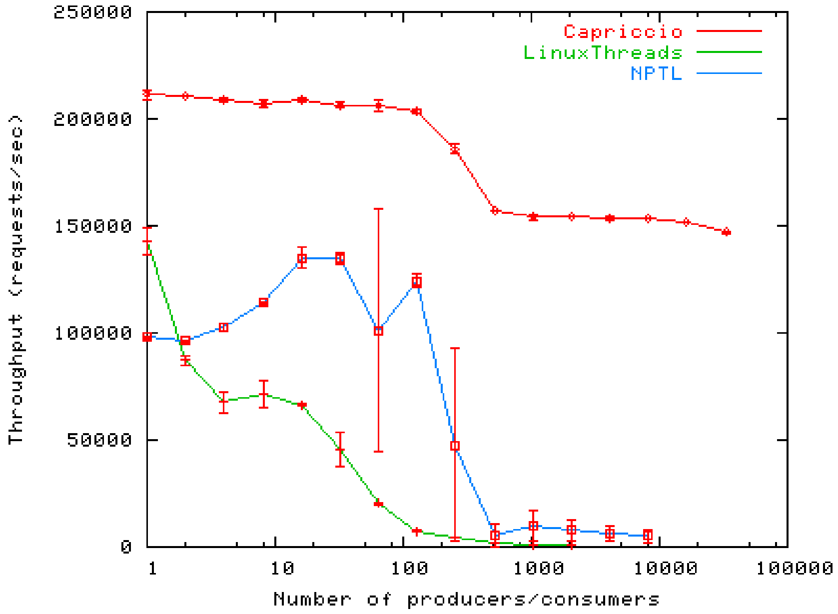
- Avoid large contiguous space allocation that consume virtual memory space
- Better usage of stack space with allocation on demand
- Allocation is done gradually in small linked stack frames
- Compiler analysis for stack frame allocation points
- Checkpoints along “call” graph
- LIFO ordering for transferable stack frames
- No need for Garbage Collector

Linked Stack Management - issues

- Function pointers are difficult to manage
  - Look at type and arguments
  - Annotate external library functions with stack bounds
- Recursion may decrease performance
  - Lightweight checkpoints
  - Application specific local optimizations
- Compiler support required for non contiguous stack
- Space is still wasted in special cases

Scalability test

- Producer-consumer microbenchmark
  - LinuxThreads begin to degrade after 20 threads
  - NPTL works up to 100 threads
  - Capriccio scales to 32K producers and consumers for a total of 64K threads
- Network performance
  - Token passing among pipes that simulates slow client links
  - 10% overhead compared to epoll
  - Faster than LinuxThreads and NPTL with more than 1000 threads
- Disk I/O performance comparable to kernel threads



Resource Aware Scheduling - Purpose

- Monitor
  - Memory and VM
  - Stack usage
  - I/O Socket descriptors for files, network
  - CPU utilization
- Maximize throughput
- Reduce thrashing
- Similar with event-driven but transparent to programmer

Resource Aware Scheduling - HowTo

- Use Blocking Graph based on “call path” (arcs)
- Detect areas where threads block (nodes)
- Dynamically learn behavior of the application
- Measure performance of each path with cycle counters

Resource Aware Scheduling - HowTo

- Dynamically maintain optimal resource utilization
  - increase priority of threads that release that resource
  - decrease priority of threads that request that resource
  - Use application specific metrics for optimum resource utilization level
- Yield profiling
  - User-level threads are problematic if a thread fails to yield
  - Easy to detect - running times are orders of magnitude larger
  - Yield profiling identifies places where programs fail to yield sufficiently often

Resource Aware Scheduling - Performance

- Micro Benchmark for 1MB stack buffer
  - Touch all pages of the buffer randomly
  - Up to 1000 threads with continuous stack
  - Up to 100k threads with linked stacks
- Reduced VM size

Future Work

- Multi processor scheduling
- Profiling tools
- Integration with latest development of the Linux kernel