Thread Analysis and Implementation

CSC 258

Parallel & Distributed Systems

January 31st, 2005

John Heidkamp & Chris Meteyer

Outline

- Thread Management for Shared Memory
  - Concurrency refresher
  - Alternatives to spin-locks
  - Spin-lock management alternatives
- Dynamic Thread Implementations
  - Decoupling
  - Linked stack management
  - Resource-aware scheduling
  - Scalability

A Concurrency Refresher

- Threads – a lightweight process that avoids overhead by sharing address space with other threads
- Contention – too much demand for a hardware resource
- Race Condition – when threads are not synchronized and the behavior of the program depends on the order the thread operations are performed in

A Concurrency Refresher

- Synchronization – making sure actions in different threads happen in the order you want then to
- Spinning (busy-wait) – processor waits for a lock to be released by another processor so it can perform its tasks on the previously locked data

Data Structures

- Program Counter (PC)
- Stack
- Control Block
- Ready Queue

Thread Operations

- Creation – allocate/initialize control block and save PC, allocate stack and copy in thread arguments, place new thread on ready queue
- Startup – remove thread from ready queue and begin execution
- Block – save register values and PC on thread's stack, place thread on condition queue, look for different thread on ready queue to start/resume

Thread Operations

- Signal Blocked Thread – remove from condition queue and add to ready queue
- Resume – remove from ready queue, restore registers, continue executing from saved PC
- Finish – deallocate stack and control block, look for different thread in ready queue to start/resume

7

Thread Package Performance

- Performance is very dependant on the algorithm used for locks and organization of data structures
- Spin-waiting has the clear cost of one or more processors just waiting for the lock. The processor doing useful work is also affected.

8

The Abstract Thread Package

9

Spin-Lock Concerns

- Latency – time which is used to create, start, or finish a thread that could have been spent doing something useful –OR- how much time it takes a for created thread to begin running (no contention)
- Throughput – rate that threads can be performed (created, started, finished) when there is contention

10

Alternatives to Spin-Locks

11

Overview

- Single Lock – all data structures protected by a single lock
- Multiple Locks – each data structure is protected with its own lock
- Local Free List – local copy of free lists – control block and stack – for each processor

12

Overview

- Idle Queue – central queue of idle processors, local copies of free lists
- Local Ready Queue – local copies of ready queue and free list for each processor

13

Single Lock

Central data structures all protected by a single lock.

Advantages

- Only a single lock per thread operation

Problems

- The single lock limits throughput
- Loop checking processors looking for work cause contention for ready queue lock

14

Multiple Locks

Each data structure protected by its own lock.

Advantages

- Smaller critical section for a lock means higher maximum rate of thread operations. Increased throughput

Problems

- More lock accesses are needed to perform a thread operation. Increased latency

15

Local Free List

Avoids locking by maintaining local copy of free lists – control block, and stack – for each processor. All still share a single ready queue.

Advantages

- Fewer lock acquisitions per thread means lower latency
- Better throughput since only the accesses to the ready queue are serialized

16

Local Free List

Problems

- Need to keep free lists balanced

Solutions

Use a global pool for free lists. Each local list has a threshold (T) which, when reached, causes half of the list to be given to the pool.

Extra Space? P = # processors, T = threshold
O(P*T) more than with a central list

Worst Case? Global and allocating processor lists almost empty and local lists are almost full.

17

Local Free List

Solutions (cont)

Utilization of Pool Lock: O(P*(R/T)), R is rate of thread creation for a single processor. Set T = P to keep lock from becoming source of contention.

Control blocks are fairly small, so having one per processor isn't a big deal.

Trading space for time isn't worth it with stacks so, only allow local stack free lists to contain one element. Allocate from pool if a stack needs more.

18

Idle Queue

Exploits parallelism when creating threads by preallocating data structures. Maintains central queue of idle processors and local copies of free lists.

Advantages

- Reduced latency when there are idle processors

Problems

- Increased latency when there aren't idle processors

Doesn't change the basically sequential method of creating new threads

19

Local Ready Queue

Each processor has its own copy of the ready queue and free list.

Advantages

- Enqueueing and dequeueing can happen in parallel

Problems

- Assignment of threads to local ready queues
- Ready threads are rare, so time spent searching for them is time not spent running them

20

Local Ready Queue

Solutions

Lock each ready queue, so an idle processor can scan them all for work, starting with its own. Assign threads to queues randomly.

Should have a one-to-one correspondence between the number of processors and ready queues to maintain locality (decreased cache misses).

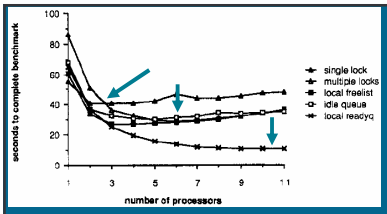
Throughput is higher with more queues, but so is latency, since there is more to search through.

21

Comparison of Operations

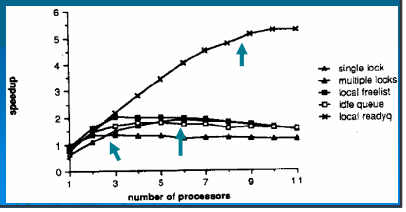
Sequent Symmetry Model A Multiprocessor, 20 Intel 80386 processors

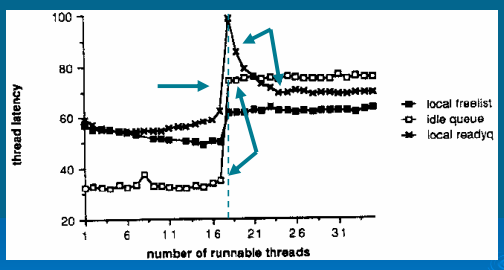
22



Time to create, start, and finish 1,000,000 null threads.

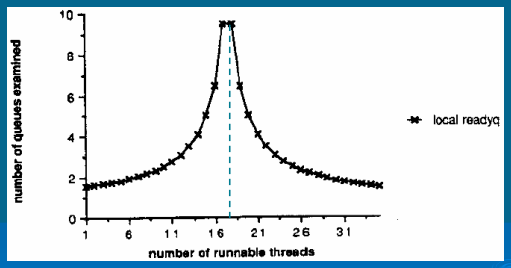
Speedup to create, start, and finish 1,000,000 null threads.





Latency vs. number of runnable threads (each thread doing 300 microseconds of work (18 processors)

24



Queues examined vs. number of runnable threads (18 processors)

25

Spin-Lock Management Alternatives

26

Overview

Hidden cost: processor doing work (holding lock) is slowed down by processors that are spinning due to bus contention.

Here, each processor has its own cache. Data not in cache must come through the bus.

Xchgb – test and set instruction, examines memory location and writes in new value

- Spin on Xchgb – each processor loops on xchgb instruction until success
- Spin on Read – spin reading lock memory location (avoids bus traffic)
- Ethernet-Style Backoff – processor only attempts to acquire lock when it estimates its chance of success is high

27

Spin on Xchgb

Processor loops on xchgb instruction until success.

Advantages

- Simplest method

Problems

- Every xchgb instruction uses resources, even if it doesn't succeed, so lock holder is slowed down
- Freeing the lock (getting permission to update), must contend with xchgb instruction trying to acquire lock

28

Spin on Read

After a lock acquisition failure, processor reads lock memory location until the cache is invalidated when the lock is released.

Advantages

- Spinning is done on the cache, so bus traffic is avoided

Problems

- Messy lock release. All processors will have caches invalidated at the same time and will all try to acquire the lock at the same time. Unsuccessful xchgb operations invalidate caches, so the problem repeats.

29

Ethernet-Style Backoff

Processor only attempts to acquire lock when it thinks its probability of succeeding is high. Estimated chance of success inversely proportional to number of past failures.

Advantages

- Avoid problem with Spin on Read where all processors try to acquire lock at the same time
- Processor with lock is less affected by waiting processors

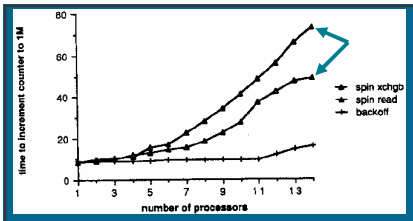
30

Ethernet-Style Backoff

Problems

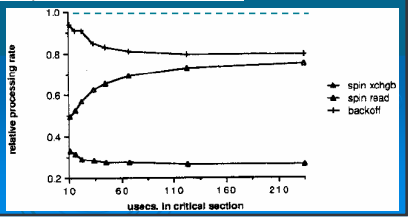
- Protocol is unfair. Newly-arrived processors more likely to acquire lock since they don't have past failures decreasing their estimated success rate.
- Takes longer for spinning processor to acquire a released lock
- Bus can still become saturated when there are large numbers of spinning processors

31



Time to increment shared counter to 1,000,000

Relative processor speed (8 processors vs 1) vs. critical section size



Recap

- What we've looked at:
  - Concurrency
  - Locking (spin-locks)
  - Performance of alternatives to spin locks and alternate methods of spin lock management

33

Implementation Overview

- The Issue:
  - Need for servers to handle 1,000's of simultaneous connections.
  - Still need to maintain:
    - Performance
    - Scalability
    - Efficient Stack Management
    - Portability

34

Two Approaches

- Event Based
  - Advantage:
    - Precise control over state management
  - Disadvantage:
    - Hides control flow from application
    - Requires stack ripping
- Thread Based
  - Same event based benefits
  - Better programming model for Internet services
  - Lightweight

35

Introducing Capriccio

- Support for existing APIs (POSIX standard)
- Scalability to 100,000 threads
- Flexibility to meet application specific needs
- Increased interaction with machine overhead and resource usage

36

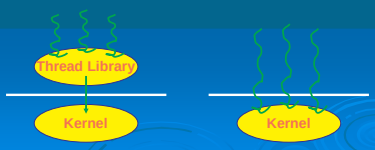
How Does it Work?

- Thread Decoupling
  - Separate user-level from kernel-level threads
  - Allows for more integrated compiler support
- Linked Stacks
  - Dynamic growth to improve scalability
- Resource-Aware Scheduling
  - Schedules based on predicted resource usage

37

User-Level vs Kernel Level Threads

- User-Level
  - Managed by thread package. Kernel is unaware.
- Kernel –Level
  - All management done in kernel



38

Decoupling

- Separate user- from kernel-level threads
  - Faster execution without changing app's code
  - Adaptive to application requirements
  - Lightweight → reduced synchronization costs
- But...
  - Difficult to support multiprocessors
  - Extra translation layer for blocking I/O calls

39

Decoupling (cont'd)

- Capriccio does decouple
- Overall, decoupling is beneficial…
  - Cooperative threading model
  - Bounded worst-case thread operation time

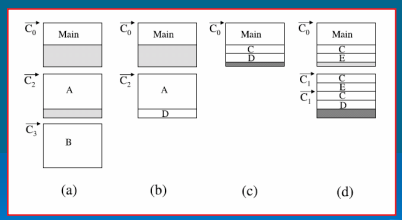
40

Stack Management

- Goal
  - An abstraction that the stack size is unbounded
- Issue
  - Fixed stack space limits number of threads
  - Ex: 2MB/thread = 1GB in just 500 threads!
- Solution
  - Linked Stacks!

41

Linked Stack Operation



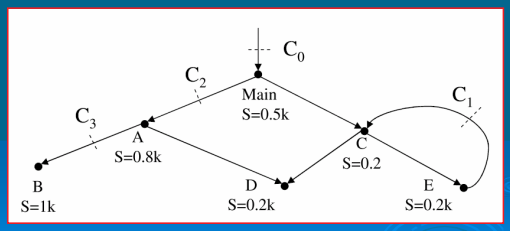
42

How Linked Stacks Work

- Goal
  - Reasonable bound on stack space consumed by each thread
- Weighted Call Graph
  - Generated by compiler
  - Node = function() Edge = call site
  - Path length = stack size of function calls

43

Weighted Call Graph



44

WCG Checkpoints

- Basis for dynamic stack allocation
- Allocation of new *stack chunks*
  - If not enough space to reach another checkpoint or leaf, allocate a new chunk
- Results in non-contiguous stacks
  - Callee requires no code change
  - Debuggers can still follow backtrace
- "Free" list of unallocated stacks

45

Placing Checkpoints

- Option 1: Place at all call sites
  - Too expensive
- Option 2: Cycle-MaxPath approach
  - First make sure at least 1 checkpoint on all cycles
  - Then determine MaxPath, and place a checkpoint on any edge that exceeds MaxPath

46

Issues

- Function Pointers
  - Solution: Categorize by number/type of args
- External Function Calls
  - Solution:
    - Programmer defined trusted stack bounds
    - Allow larger stack chunks (through Capriccio)
- Internal/External Wasted Space
  - Solution: Adjust MaxPath & MinChunk
    - Note this is a performance-space tradeoff

47

Linked Stack Summary

- Makes preallocation of large stacks unnecessary
- Linked stacks shared between threads, therefore improving paging behavior
  - Benchmark Test: Fixed vs Dynamic Stacks
    - Reduced paging costs by 69%
    - Scales linearly up to 100,000 threads

48

Thread Scheduling

- Goal
  - A dynamic scheduling algorithm that is transparent yet application specific
- Resource-Aware Scheduling
  - Promote threads that will free scarce resources
  - Blocking Graph

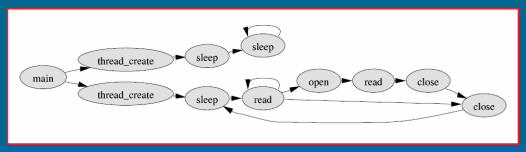
49

How RAS Works

- Threads are scheduled as they traverse the blocking graph
- Strategy
  1. Monitor resource utilization levels
  2. Annotate nodes with next-stage cost of blocked threads
  3. Dynamically prioritize threads

50

Blocking Graph



Assumption:

Resource usage similar for many threads at each *blocking point* (node).

51

RAS Details

- RAS is powerful!
  - Deduce stages automatically
  - Knowledge of resource usage at each stage
  - More adaptive scheduling
- Finer grained dynamic scheduling
  - Greater scalability & performance
  - Keeps programming model simple

52

Blocking Statistics

	Item	Cycles	Enabled
Apps	Apache	32697	n/a
	Knot	6868	n/a
System	stack trace	2447	Always for dynamic BG
	edge statistics	673	During sampling periods

Capriccio — **Application Specific** — **Cycles used to maintain scheduler info**

- Statistics Cycles = <2%
- Stack Trace Cycles = 5.7%
  - Could be improved through greater compiler integration

53

Pitfalls

- Determining max capacity of a resource
- Virtual memory not tracked at nodes
- Thrashing
- Application memory management hides resources allocation from runtime system

54

Capriccio Test Results

- Apache Web Server
  - Apache with Capriccio improved threading performance by 15%
- Resource-Aware Admission Control
  - Producer-Consumer test application
  - Capriccio detects stack overload conditions and limits thread producer

55

Scalability

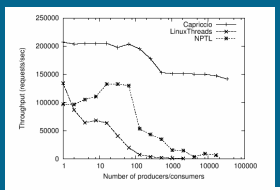


Figure 1: Producer-Consumer - scheduling and synchronization performance.

Results

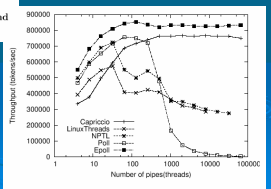


Figure 2: Pipetest - network scalability test.

56

Scalability

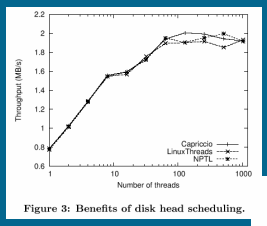


Figure 3: Benefits of disk head scheduling.

Results

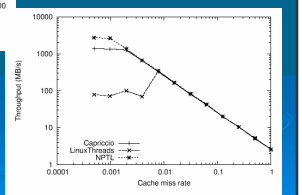


Figure 4: Disk I/O performance with buffer cache.

57

What's Next?

- Multiprocessor support
- Enhanced compiler design
  - Blocking graph generation at runtime
  - Warn programmer of race conditions
- Automatic parameter tuning
  - Dynamically optimize stack parameters
    - Reduce wasted space
    - Maximize performance

58

Implementation Summary

- Thread vs event-driven models
- Decoupling
- Linked Stack Management
- Resource-Aware Scheduling
- Scalability
- Compiler Support

59

References

The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors

Anderson, Lazowska & Levy

Capriccio: Scalable Threads for Internet Services

Von Behren, Condit, Zhou, Necula, Brewer

60