Capriccio: Scalable Threads for Internet Services

Rob von Behren, Jeremy Condit, Feng Zhou, George Necula and Eric Brewer
University of California at Berkeley

Presenter: Cong Lin
Outline

Part I  Motivation & Goals

Part II  Capriccio: Approaches
  - Thread Package
  - Linked Stacks
  - Resource-aware Scheduling

Part III  Experiments & Results

Part IV  Conclusion & Future work
Part I  Motivation

Increasing **scalability** demands for Internet services
The situation is…

Hardware: “I’m ready”

Software: “…Wait for me”

Current implementations are event based
Event Based Systems - Drawbacks

- Events systems hide the control flow
  - Difficult to understand and debug
  - Programmers need to match related events
Goals: Capriccio

- Support for existing thread API
- **Scalability** to hundreds of thousands of threads
- Flexibility to address **application-specific** needs
Part II  Capriccio: Approach

- Thread package
  - User-Level thread & Cooperative scheduling
- Linked stacks
  - Address the problem of stack allocation for large numbers of threads
  - Combination of compile-time and run-time analysis
- Resource-aware scheduler
Thread package: User Level Thread

- Advantages

- Performance
  - Ease thread synchronization overhead
  - No kernel crossing for preemptive threading
  - More efficient memory management at user level

- Flexibility
  - Decoupling user and kernel threads allows faster innovation
  - Can use new kernel thread features without changing application code
  - Scheduler tailored for applications
Thread package:  
User Level Thread

- Disadvantages
  - Additional Overhead
    - Replacing blocking calls with non-blocking calls
  - Multiple CPU synchronization
It is Worthy.
Thread package: User Level Thread

- **Implementation**

- **Context Switches**
  - Fast context switches provided by Edgar Toernig’s coroutine library

- **I/O**
  - Intercepts blocking I/O calls & uses epoll() for asynchronous I/O

- **Scheduling**
  - Very much like an event-driven application
  - Events are hidden from programmers

- **Synchronization**
  - Supports cooperative threading on single-CPU machines
Question

It seems that cooperative scheduling is a must for highly concurrent servers. Does this carry to multiprocessors? (i.e. does cooperative scheduling maintain its advantages on a multiprocessor?)
Linked Stack

- The problem: fixed stacks
  - Overflow vs. wasted space
  - Limits thread numbers
- The solution: dynamic allocation
  - Allocate space as needed
  - Compiler aid
    - Add runtime checkpoints on Call Graph
    - Guarantee enough space until next check
Linked Stack

- Parameters
  - \textit{MaxPath(Bound)}

- Steps
  - Break cycles
  - Trace back

\[ \text{MaxPath} = 8 \]
Linked Stack

- Parameters
  - $MaxPath$

- Steps
  - Break cycles
  - Trace back

$MaxPath = 8$
Linked Stack

- **Parameters**
  - $MaxPath$

- **Steps**
  - Break cycles
  - Trace back

$MaxPath = 8$
Linked Stack

- Parameters
  - MaxPath
- Steps
  - Break cycles
  - Trace back

\(\text{MaxPath} = 8\)
Linked Stack

- Parameters
  - MaxPath

- Steps
  - Break cycles
  - Trace back

MaxPath = 8
Linked Stack

- Parameters
  - \( \text{MaxPath} \)

- Steps
  - Break cycles
  - Trace back

\[ \text{MaxPath} = 8 \]
Questions

Each execution of checkpoint needs a kernel thread, since it needs a new allocation of memory, and in the large recursive program, it may occur thousands of times. How about the time assumption and the newly created kernel threads? Is it efficient for all the conditions?
Resource-aware Scheduling

- Similar to event-based:
  - Whether a process is close to completion
  - Whether a system is overloaded

- Blocking Graph
Resource-aware Scheduling
The Blocking Graph

- Thread-based
- View applications as sequence of stages, separated by blocking calls
Resource-aware Scheduling

- Track resources used along BG edges
  - Mark memory, file descriptors, CPU
  - Predict future from the past
- Algorithm
  - Increase use when underutilized
  - Decrease use near saturation

- Advantages
  - Workload-sensitive admission control
  - Operate before thrashing
Questions

Given that resource status is hard to detect inside a process and other processes in the operating system may influence the resource status,

- Is it so important to design a resource aware scheduling mechanism in a user level thread library?
- How to estimate the maximum capacity of a particular resource?
### Part III Experiments & Results

#### Thread Primitive - Latency

<table>
<thead>
<tr>
<th>Thread Primitive</th>
<th>Capriccio</th>
<th>LinuxThreads</th>
<th>NPTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread creation</td>
<td>21.5</td>
<td>37.9</td>
<td>17.7</td>
</tr>
<tr>
<td>Thread context switch</td>
<td>0.24</td>
<td>0.71</td>
<td>0.65</td>
</tr>
<tr>
<td>Uncontended mutex lock</td>
<td>0.04</td>
<td>0.14</td>
<td>0.15</td>
</tr>
</tbody>
</table>
Results: Thread Scalability
Results:

- I/O performance
- Web server performance
- Runtime overhead
Capriccio simplifies high concurrency
- Scalable & high performance
- Control over concurrency model
  - Stack safety
  - Resource-aware scheduling
Future work

- Threading
  - Multi-CPU support
- Compile-time techniques
  - Parameters tuning
- Scheduling
  - More sophisticated prediction
Orz

THANKS...
Questions

In the Resource Aware Scheduling, it is mentioned that it has workload sensitive form of admission control. How does it actually measure whether a given thread is near to complete?

Besides I/O, there is another user level operation which can affect the kernel thread into block, that is page fault. Then, how to deal with the page fault problem in Capriccio?
The "linked stacks" mechanism isn't specific to Capriccio or User space threading. Has this mechanism been investigated in combination with normal kernel threads, as an attempt to increase scalability by reducing address space pressure?

Do you think the Berkley team downplays the difficulty of programming in parallel? They mention starvation a few times, and in 6 say that "any apparent advantages of events [over user-level threads] are simply artifacts of poor thread implementations". The paper focuses on performance and efficiency and ease of debugging and leaves out the fact that people are bad at writing parallel programs free of deadlock and starvation.