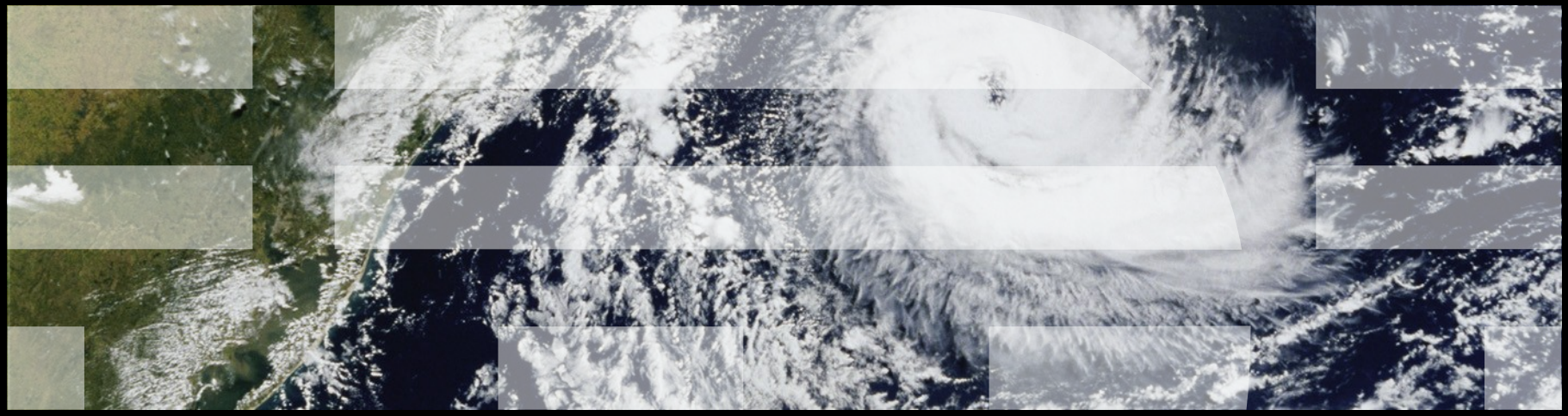
Paul E. McKenney – IBM Distinguished Engineer, Linux Technology Center

29 Jan 2011

# Verifying Parallel Software:

## *Can Theory Meet Practice?*



# Question 4

**In what sense are criteria such as linearizability, commutativity, lock freedom, wait freedom, essential or desirable or useful? Why is linearizability "the right" correctness specification to verify? Or why is it wrong? What are the alternatives?**

# Question 4: Short-Form Answer

**In what sense are criteria such as linearizability, commutativity, lock freedom, wait freedom, essential or desirable or useful? Why is linearizability "the right" correctness specification to verify? Or why is it wrong? What are the alternatives?**

- Linearizability, commutativity, lock freedom, and wait freedom are useful tools in the practicing parallel programmer's toolbox. But if they are the ***only*** tools in the toolbox, then that programmer is in a world of hurt!!!

# Table of contents

# How did I get this way?

- Student programmer, business applications software (dorm room assignment/billing)
- Contract programmer (building control, cardkey access systems, acoustic navigation)
- SRI International (systems administration, packet-radio research, Internet research)
- Sequent Computer Systems: DYNIX 3 and DYNIX/ptx on Balance and Symmetry
  - 1990: 30-way SMP, primarily via locking: parallel memory allocation
    - Workload moving from scientific/technical to RDBMS transactional
  - 1993: Clustering requires scalable distributed lock manager: RCU (AKA “rclock”)
    - Workload almost entirely RDBMS transactional
  - 1996-7: NUMA-Q enables scalable 64-way system: counting requires distribution
    - Workload still almost entirely RBMS transactional
    - In 1999, seven of the ten largest Oracle databases ran on NUMA-Q
  - Note: Symmetry and NUMA-Q are single CPU architecture: x86
- IBM: AIX: 2000: More of the same, except that Power brings weaker memory ordering
- IBM: Linux: 2001: Some changes...

# Starting the Linux journey

- 2001: 2-4 CPU systems (SGI does 100s with special patchset), lots of CPU architectures
  - Locking with much coarse-grained locking
    - Infrastructure, web serving, small DBMS, …
- 2001-present: a simple matter of applying scalability lessons learned in the 1990s?

# The Linux journey

- 2001: 2-4 CPU systems (SGI does 100s with special patchset), lots of CPU architectures
  - Locking with much coarse-grained locking
    - Infrastructure, web serving, small DBMS, …
- 2001-present: a simple matter of applying scalability lessons learned in the 1990s?
  - Not quite!!!
  - Several complicating factors: workloads, real-time response, energy efficiency
- Workloads:
  - Old way: When a certain RDBMS can run, we are done!!!
  - New way: Everything from smartphones to supercomputers. We are *never* done.
- Real-time response:
  - Old way: If 90% of transactions complete in under two seconds, we are good!!!
  - New way: Make that 100% in a few tens of microseconds. But no pressure!!!
- Energy efficiency:
  - Old way: Energy costs are a negligible fraction of total costs
  - New way: Hand-held battery-powered SMP systems. Yes, they are already here.

# Practitioner Role For This Talk

# Education

- 1981: BS Computer Science, Oregon State University
- 1988: MS Computer Science, Oregon State University
- 2004: PhD Computer Science and Engineering, OGI School of Science and Engineering, Oregon Health & Science University

# Education

- 1981: BS Mechanical Engineering, Oregon State University
- 1981: BS Computer Science, Oregon State University
- 1988: MS Computer Science, Oregon State University
- 2004: PhD Computer Science and Engineering, OGI School of Science and Engineering, Oregon Health & Science University

- It is my mechanical engineering background that I bring to bear on Question 4

# Not That Mechanical Engineers are Perfect...

# But They Have Many Centuries Of Experience To Draw From

- Elastic tensile response (stress/strain)
- Plastic tensile response (stress/strain)
- High-temperature deformation (creep)
- Viscoelastic response in polymers
- Defect population in solids
- Griffith crack theory
- Critical crack length
- Phase diagram (transition temperatures)
- 1st, 2nd, and 3rd laws of thermodynamics
- Ideal gas law
- Maxwell relations for simple compressible systems
- Clapeyron equation for change in phase
- Fugacity for non-ideal gasses
- Control-volume analysis
- The Bernoulli equation for fluid flow
- Navier-Stokes equations for viscous fluid flow
- Reynolds-number analysis (laminar vs. turbulent flow)
- Froude-number analysis (ship hydrodynamics)
- Coefficient of drag
- Boundary-layer analysis for fluid flow
- Von Karman momentum integral analysis
- Conductive, convective, and radiative heat transfer
- Free-body diagram (force/acceleration analysis)
- D'Alembert method for analyzing dynamic systems
- Lagrange's method for analyzing dynamic systems
- Root-locus method (stability analysis)
- And I have gone through only four of my old mechanical engineering textbooks!!!

# In Contrast, Question 4 Proposes Only Four Approaches

- Linearizability
- Commutativity
- Lock freedom
- Wait freedom

# In Contrast, Question 4 Proposes Only Four Approaches

- Linearizability
- Commutativity
- Lock freedom
- Wait freedom

- All good, but...
  - I respectfully suggest that it might not hurt to provide a few more

# What Additional Approaches Can There Be?

# Example of Additional Approach: RCU Grace-Period Guarantee

- RCU implementations validated “on bare metal” – not necessarily via linearizability
  - *Uses* of RCU handled much differently! Key point: RCU efficiently removes data races
- Given an RCU read-side critical section:
  - `rcu_read_lock()`; $R_1$; $R_2$; $R_3$; ...; `rcu_read_unlock()`;
- And given an RCU update:
  - $M_1$; $M_2$; $M_3$; ...; `synchronize_rcu()`; $D_1$; $D_2$; $D_3$; ...;
- Then the following must hold:
  - $\forall m,i(M_m \rightarrow R_i) \vee \forall i,n(R_i \rightarrow D_n)$
- The following pair of derived formulas are usually more helpful for validating RCU uses:
  - $\exists i,m(R_i \rightarrow M_m) \Rightarrow \forall j,n(R_j \rightarrow D_n)$
  - $\exists n,i(D_n \rightarrow R_i) \Rightarrow \forall m,j(M_m \rightarrow R_j)$
- Joint work with Mathieu Desnoyers, Alan Stern, Michel Dagenais, and Jonathan Walpole
  - http://www.rdrop.com/users/paulmck/RCU/urcu-supp.2010.12.14a.pdf
  - To appear in IEEE Transactions on Parallel and Distributed Systems

# Relationship of RCU guarantees to validation efforts

Prove based on higher-level guarantees

**Algorithm Using RCU**

**Algorithm Using RCU**

Prove based on RCU guarantee

**RCU Implementation**

Prove based on first principles

# But Isn't This Guarantee Simply Derived From RCU Implementation Proof of Correctness?

## But Isn't This Guarantee Simply Derived From RCU Implementation Proof of Correctness?

# Yes, by Definition

Just like the Bernoulli Equation can be derived from the Navier-Stokes Equations

## But Isn't This Guarantee Simply Derived From RCU Implementation Proof of Correctness?

# Yes, by Definition

Just like the Bernoulli Equation can be derived from the Navier-Stokes Equations

*The Navier-Stokes Equations are quite general*

## But Isn't This Guarantee Simply Derived From RCU Implementation Proof of Correctness?

# Yes, by Definition

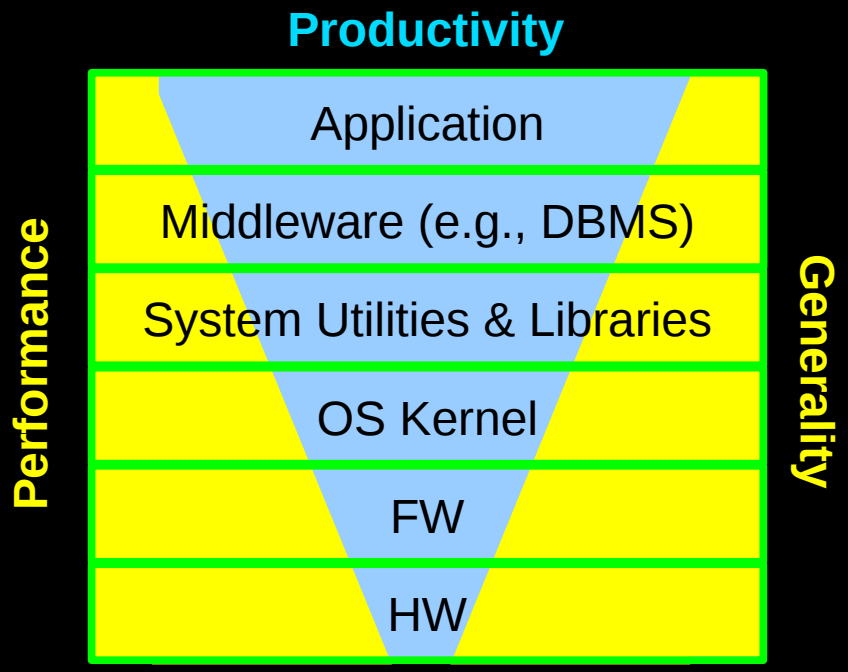
Just like the Bernoulli Equation can be derived from the Navier-Stokes Equations

*The Navier-Stokes Equations are quite general*

*But within their area of applicability, the Bernoulli Equations are much, **much** easier to use!!!*

# But Just How Many More Such Approaches Are There?

# But Just How Many More Such Approaches Are There?
## *Lots of Them!!!*




There is great variety at the application level

# Use of Different Paradigms

- Low-level critical code: Do what is required for performance
  - Combinatorial explosion limits the size of algorithm that can be proven
    - But spinlocks, sequence locks, RCU, etc. can be usually verified reasonably
  - Use higher-level semantics to prove uses of such algorithms
  - Analogy with mechanical engineering: use beam equations, not Schroedinger equations!!!
  - In real life: use rated loads for specific beams
- Interactions with the outside world:
  - You don't know what the ordering is in any case: why fabricate one?
  - “Stop worrying and learn to love non-determinism, non-linearizability, non-strong non-commutativity, conflicting operations, ...”
  - Added benefits: real-time response, keep up with networking HW, ...

# Validating Low-Level Performance/Latency-Critical Code

- Combination of stress testing and mechanical proofs
- Mechanical proofs
  - Few Linux kernel programmers do proofs, but the number is growing
    - Paul E. McKenney, Mathieu Desnoyers, and a few others
  - Use Promela/spin (historical choice, might choose differently today)
  - Explicitly represent non-determinism
    - Model full permutation of possible orderings (below), or...
    - Explicitly model abstract representation of cache and/or store buffer
    - Some too-large models converted to VHDL and subjected to hardware validation

```
do :: 1 -> sum = ctr[0]; i = 1; break
   :: 1 -> sum = ctr[1]; i = 0; break
od;
sum = sum + ctr[i];
```

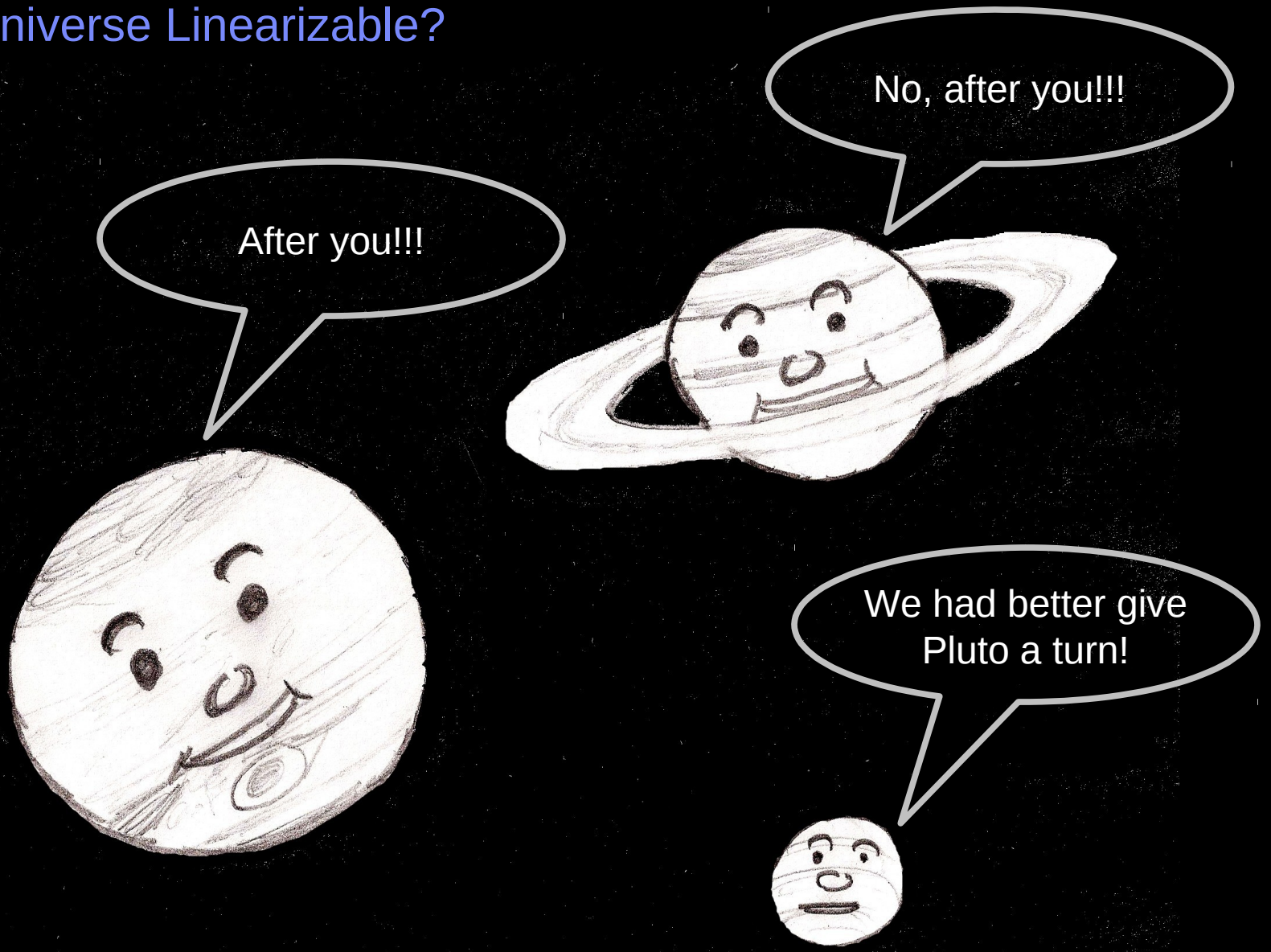
# Linearizability: A Critique

# Linearizability Can Be Expensive

- The added cost of linearizability should not be controversial
- 1996 paper entitled “Linearizable counting networks” by Herlihy, Shavit, and Waarts:
  - “Finally, we prove that these trade-offs are inescapable: an ideal linearizable counting algorithm is impossible. Since ideal non-linearizable counting algorithms exist, these results establish a substantial complexity gap between linearizable and non-linearizable counting.”

# Is the Universe Linearizable?

# Is the Universe Linearizable?




**No, 'fraid not!!!**

# Is Linearizability Useless?

# Is Linearizability Useless?

- Of course not!!!
  - Where it applies, linearizability simplifies analysis and verification
  - Linearizability applies much of the time
  - In the concurrent programmer's toolbox, it is analogous to the hammer
- But linearizability is not always the right tool for the job
  - For small critical code paths, the additional complexity of analysis without linearizability can be very worthwhile
  - For code that interfaces to the outside world, linearizability can be a useless and expensive fiction
    - Network routing tables are the poster child for this case
    - Tracking the order of routing updates is less important than delivering packets
  - For statistics gathering, linearizability can be useless and expensive
- Linearizability is often the right tool for the job, but not always

# Where is Non-Linearizability Most Important?

- Applications requiring extreme real-time response
- Non-strongly non-commutative algorithms with extreme performance and scalability requirements
  - Operating system kernels, server applications
- Statistics gathering

- Yes, you can sometimes transform algorithms to preserve linearizability at the cost of more-complex semantics

# Where is Non-Linearizability Most Important?

- Applications requiring extreme real-time response
- Non-strongly non-commutative algorithms with extreme performance and scalability requirements
  - Operating system kernels, server applications
- Statistics gathering

- Yes, you can sometimes transform algorithms to preserve linearizability at the cost of more-complex semantics
- You can also describe planetary movements using epicycles

# What Is Needed Going Forward?

- A major motivation for linearizability is simplification of proofs
  - But practitioners' proofs are often carried out mechanically
  - Adopting state-space-reduction techniques from hardware validation is likely to be quite fruitful
- Low-level search for parallelism likely to result in low gains
  - More significant gains available at the application level
    - Larger units of work results in lower communication overhead, which in turn results in better performance and scalability
    - See Patterson's “The Trouble with Multicore” in July 2010 IEEE Spectrum
  - Application-level parallelism will require higher-level criteria
    - These will tend to be application specific: specialization has many benefits

# Commutativity: A Critique

# Is Commutativity Useless?

# Is Commutativity Useless?

- Of course not: Commutativity can be quite useful
  - Statistical counters, searches and non-conflicting updates
- But its area of applicability appears to be limited
  - For example. searches do not commute with conflicting updates
  - But there are important use cases that ***don't care***:
    - Network packet routing: by the time the update arrives, packets have already been going the wrong way, perhaps for ***minutes***
    - Security policy updates: in some cases, uncertainty in time of update is OK
    - Detection of new hardware: the timeframe that matters is often human reaction time
  - In many cases, just wait for the period of uncertainty to complete
- And strong non-commutativity seems much more interesting
  - Use cases are non-commutative, but not strongly non-commutative
  - "Laws of Order" by Attiya, Guerraoui, Hendler, Kuznetsov, Michael, and Vechev contains interesting results in this area

# What Is Needed Going Forward?

- Identify non-strongly non-commutative algorithms that can make use of inexpensive operations
- Bite the bullet and relax linearizability requirements where it makes sense to do so
  - If you please that non-linearizability ***never*** makes sense:

# What Is Needed Going Forward?

- Identify non-strongly non-commutative algorithms that can make use of inexpensive operations
- Bite the bullet and relax linearizability requirements ***where it makes sense to do so***
  - If you believe that non-linearizability ***never*** makes sense:
  - Please let me be the first to inform you that the 1980s ended long ago

# Lock Freedom and Wait Freedom: A Critique

# Are Lock Freedom and Wait Freedom Useless?

- Absolutely not!!!
- Lock-free & wait-free algorithms heavily used in practice
  - For example, in real-time systems and performance-critical software
- At least in the special cases where they are simple and fast
  - Simple stacks and queues
  - Statistical counters
  - RCU read-side primitives (and update-side primitives in some cases)
- General lock-free/wait-free constructs fare less well
  - Before you tell me that software transactional memory is a good example of a lock-free/wait-free construct, keep in mind that the semi-reasonably performing STMs use locking
    - And the fastest of these (e.g., swissTM) place significant burden on developers

# What Is Needed Going Forward?

- Greater focus on semi-non-blocking and semi-wait-freedom
  - Example: non-blocking enqueue with blocking dequeue
    - Michael and Scott: “Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors”
  - Example: wait-free RCU readers with blocking RCU updaters
    - http://www.rdrop.com/paulmck/RCU
    - Non-blocking RCU updates are possible in some situations
  - Such algorithms are well suited for situations where real-time response is required only on some code paths
    - For example, real-time threads queuing data for a non-real-time logging thread
- Combining non-blocking and wait-free algorithms with other concurrency-control mechanisms
  - Many software artifacts require a variety of approaches

# Theoretical Correctness Criteria

# What Is Needed Going Forward For Correctness Criteria?

- Rethink the name “correctness criteria”
- We have seen that correct algorithms can be non-linearizable, non-deterministic, non-wait-free, and non-lock-free

- Therefore, shouldn't we say “properties” rather than “correctness criteria”?

# Don't Forget The Simple Stuff!!!

# Understand The Properties of Underlying Software and Hardware

- Would you trust:
  - A bridge designed by someone who didn't understand that concrete, while strong in compression, is weak in tension?
  - A home heating system designed by someone who didn't understand that would houses burn?
  - A home in the rainy Pacific Northwest designed by someone who didn't understand that wood rots in temperate rain forests?
  - A space shuttle designed by someone who didn't understand the low-temperature properties of O-rings?

# Understand The Properties of Underlying Software and Hardware

- Would you trust:
  - A bridge designed by someone who didn't understand that concrete, while strong in compression, is weak in tension?
  - A home heating system designed by someone who didn't understand that would houses burn?
  - A home in the rainy Pacific Northwest designed by someone who didn't understand that wood rots in temperate rain forests?
  - A space shuttle designed by someone who didn't understand the low-temperature properties of O-rings?
- If not, why would you trust an algorithm designed by someone who didn't understand hardware properties?

# Understand The Properties of Underlying Software and Hardware

- Would you trust:
  - A bridge designed by someone who didn't understand that concrete, while strong in compression, is weak in tension?
  - A home heating system designed by someone who didn't understand that would houses burn?
  - A home in the rainy Pacific Northwest designed by someone who didn't understand that wood rots in temperate rain forests?
  - A space shuttle designed by someone who didn't understand the low-temperature properties of O-rings?
- If not, why would you trust an algorithm designed by someone who didn't understand hardware properties?
  - Yes, these properties have changed over time
  - And these changes have dramatically affected algorithm design

# Don't Forget Simple Techniques

- Partitioning is simple, but can be extremely effective
- Batching is simple, but amortizes synchronization overhead
- Sequential execution is simple, and should be used when the resulting performance is sufficient
- Pipelining can provide low synchronization overhead
- Never be afraid to exploit important special cases:
  - Read-only and read-most situations, partitionable common-case execution, privatizable data, …
  - Relativistic programming codifies some of these
    - Joint work: Jonathan Walpole, James Hook, Josh Triplett, Phil Howard, Eric Wheeler
- Finding bottlenecks should be simple, but often isn't

# How Does the Linux Kernel Community Cope?

# Linux Kernel Scalability

- Not perfect by any means, but...
- Boyd-Wickizer et al.: “An Analysis of Linux Scalability to Many Cores”

# Kernel-Community Approaches to Concurrency (Subset 1/2)

- Organizational mechanisms
  - Maintainers and quality assurance: recognition and responsibility
  - Informal apprenticeship/mentoring model
  - Design/code review required for acceptance
  - Aggressive pursuit of modularity and simplicity
- Use sane idioms and abstractions
  - Locking, sequence locking, sleep/wakeup, memory fences, RCU, ...
  - Conventional use of memory-ordering primitives, for example:
    - Susmit's message passing (MP): sync + dependency
    - Susmit's write-to-read causailty (WRC): sync + dependency
  - This avoids Susmit's PPOCA, RSW, RDW, …
    - Hard to even express in core kernel code
  - Needing to know too much about the underlying memory model indicates broken abstraction, broken design, or both

# Kernel-Community Approaches to Concurrency (Subset 2/2)

- Static source-code analysis
  - "checkpatch.pl" to enforce coding standards
  - "sparse" static analyzer to check lock acquire/release mismatches
  - "coccinelle" to automate inspection and generation of bug fixes
- Dynamic analysis
  - "lockdep" deadlock detector (also checks for misuse of RCU)
  - Tracing and performance analysis
  - Assertions
- Aggressive automation
  - "git" source-code control system: from weeks to minutes for rebases and merges
- Testing
  - In-kernel test facilities such as rcutorture
  - Out-of-kernel test suites

# Kernel-Community Approaches to Concurrency

- To err is human, and therefore...
  - –People/organizational mechanisms are at least as important as concurrency technology
  - –Use multiple error-detection mechanisms
  - –For core of RCU, validation starts at the very beginning:
    - Write a design document: safety factors and conservative design
    - Consult with experts, update design as needed
    - Write code in pen on paper: Recopy until last two copies identical
    - Do proofs of correctness for anything non-obvious
    - Do full-up functional and stress testing
    - Document the resulting code (e.g., publish on LWN)
  - –If I do all this, then there are probably only a few bugs left
    - And I detect those at least half the time

# An Important Question

# Given a Randomly Selected Human Being...

- *Any* human being: head of state, rock star, street person, farmer, researcher, student, CEO, diplomat, janitor, plumber, housewife, toddler, juvenile delinquent, bureaucrat, mafia don, warlord, mercenary soldier, terrorist, policeman, lawyer, doctor, kernel hacker, hardware architect, concurrency-theory researcher, application developer, …

- What one change would you make to this person's life?

# How To Help Someone

- I am perhaps overly proud of my contributions to the Linux kernel community
- I have been able to contribute because I have been:
  - a kernel hacker myself for almost 20 years
  - a member of the Linux kernel community for almost 10 years

# How To Help Someone

- I am perhaps overly proud of my contributions to the Linux kernel community
- I have been able to contribute because I have been:
  - a kernel hacker myself for almost 20 years
  - a member of the Linux kernel community for almost 10 years
- To reliably make a positive change to people's lives, you must live among them
- I hope that this workshop helps us get to know one another

# Summary of Recommendations

# Some recommendations from two decades of parallel experience

- Adopt HW-validation state-space-reduction techniques for non-linearizability
- Develop high-level application-specific criteria to validate large applications
- Identify non-strongly non-commutative algorithms that can use inexpensive operations
- Relax linearizability requirements where it makes sense to do so
- More focus on semi-non-blocking and semi-wait-free algorithms
- Combine non-blocking and wait-free algorithms with other mechanisms
- Call a spade a spade: “properties” rather than “correctness criteria”
- Understand the underlying hardware and software
- Don't forget the simple stuff
  - “Embarrassingly parallel” is an embarrassment only to those who fail to exploit it
  - The simpler the theory, the more likely you are to get It right!!!
  - Not everything needs to be parallel

# Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

# QUESTIONS?