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

25 Jan 2011 linux.conf.au Multicore and Parallel Computing MiniConf

Verifying Parallel Software:

Can Theory Meet Practice?

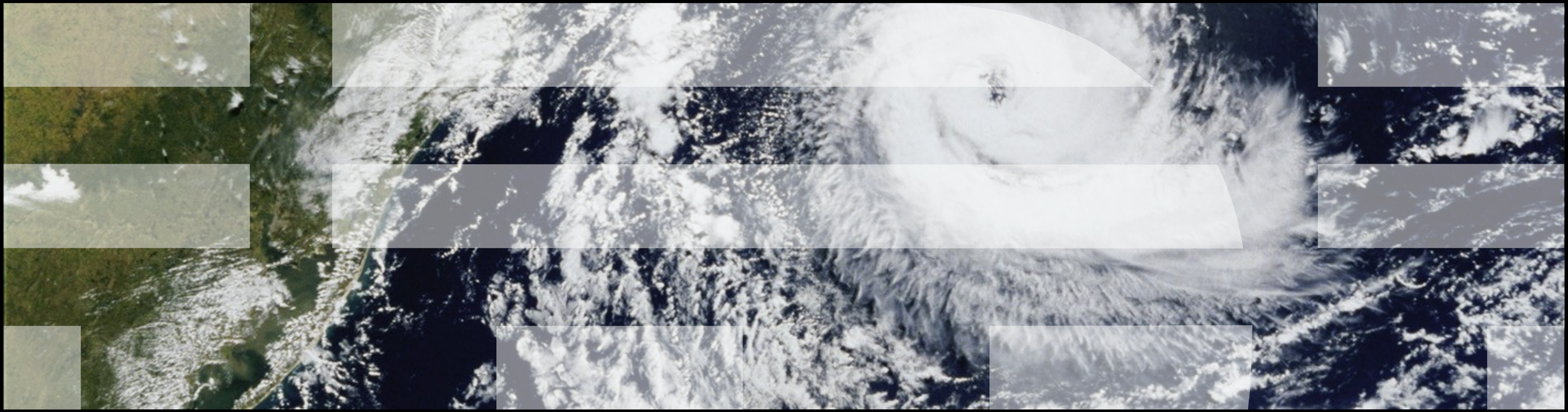


Table of contents

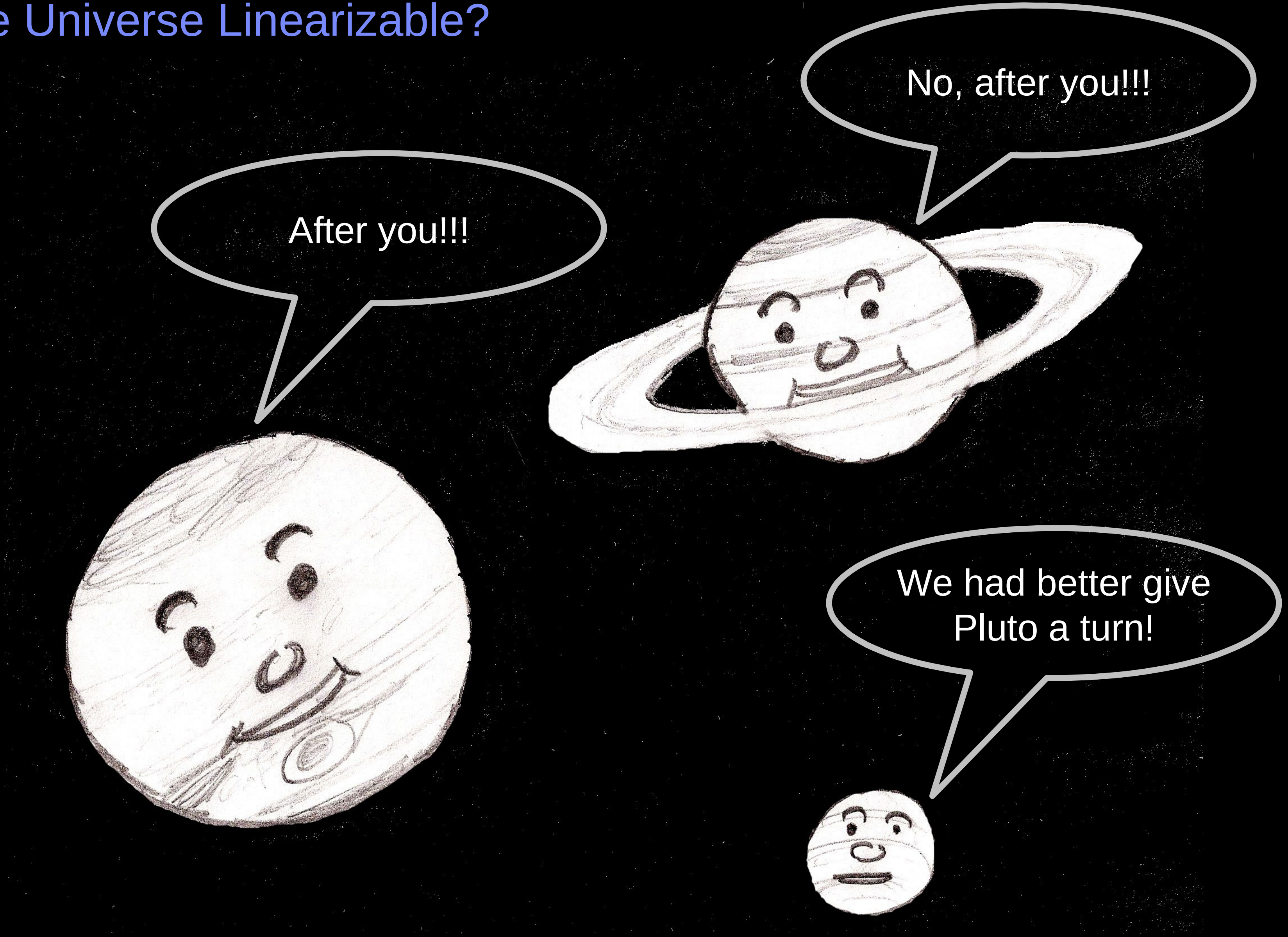
Linearizability: A Critique

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."
- There are therefore performance, scalability, and energy-efficiency benefits to abandoning linearizability

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
    - Great tool, but not a replacement for screwdriver or wrench
- 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
  - 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 (Attiya et al. 2011)
  - 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?

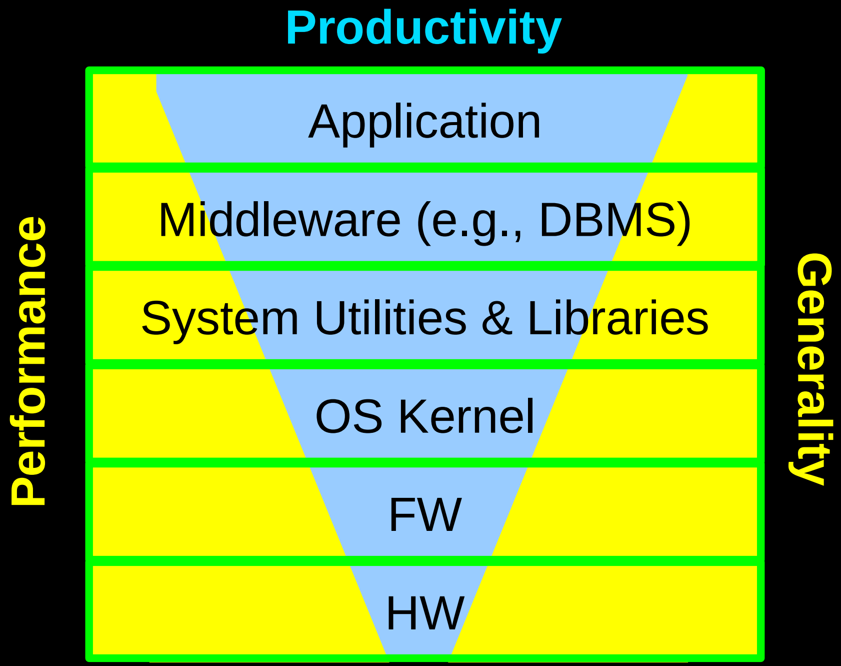
- Applications requiring extreme real-time response
- Non-strongly non-commutative algorithms with extreme performance and scalability requirements (Attiya et al. 2011)
  - Operating system kernels, server applications
- Statistics gathering

- 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

But Just How Application-Level Opportunity Is There?
Lots of Them!!!



There is great variety at the application level

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 burdens 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 is simple, but can greatly reduce synchronization overhead
- Never be afraid to exploit important special cases:
  - Read-only and read-most situations, partitionable common-case execution, privatizable data, …
- 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!!!

Legal Statement

QUESTIONS?