is paper describes objective technical results and analysis. Any subjective views or opinions that might be expres
the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Governm

SAND2015-10033C

Getting the right answer despite incorrect hardware[1]

Mark Hoemmen
mhoemme@sandia.gov

Sandia National Laboratories[2]

02 Dec 2014

[1] Approved for public release; SAND # 2014-XXXXX C

ories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of San
well International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under cont
Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Thanks!

- To coauthors and collaborators:
  - Dr. James Elliott (North Carolina State University)
  - Prof. Dr. Frank Mueller (his advisor)
  - Drs. Mike Heroux & Kurt Ferreira (Sandia)
  - Prof. Dr. Patrick Bridges (UNM)
- Funding: US Department of Energy Office of Science

Outline

Getting the right answer despite incorrect hardware

- Incorrect hardware. . .
  - . . . may expose incorrect arithmetic or storage
  - (to reduce power or improve performance)
  - This may cause silently wrong results, or . . .
  - . . . make algorithms take a long time to fail
  - Software bugs may have the same effect!
- Getting the right answer
  - Our focus: iterative linear solvers ($Ax = b$)
  - Techniques apply to other numerical algorithms
  - Possible generalization to discrete problems
- Through two, mostly algorithmic techniques:
  - Skeptical programming
  - Selective reliability

Why we care about silently wrong results

Simulations for high-consequence decisions

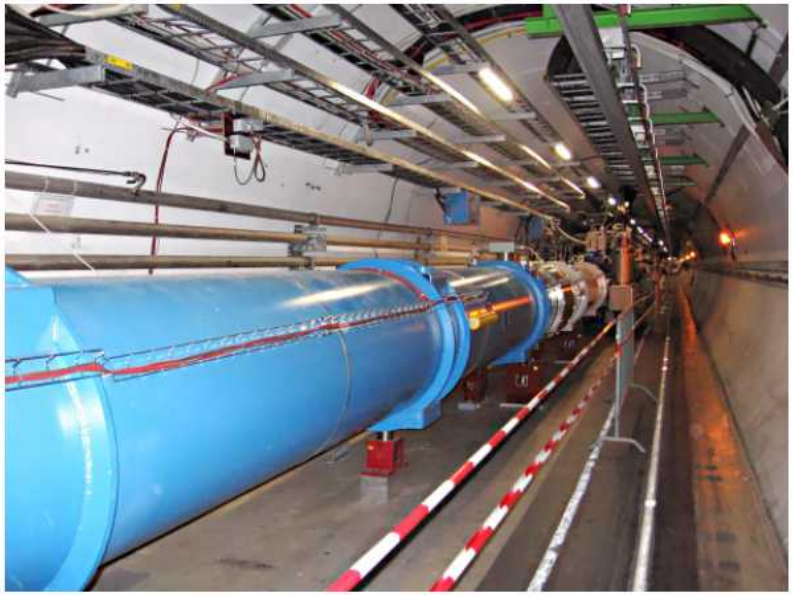
- Decisions like
  - Existential threats
  - Weather disasters
  - Civil engineering
- High fidelity & tight time constraints
- Hard or impossible to validate experimentally

Hurricane Sandy damage (Image credit: US Air Force). Evacuation may kill more than it saves if the weather forecast is incorrect.

High-consequence science & data analysis

Higgs boson: Search for rare event in lots of data

- 1 out of 10^{10} LHC collisions may produce a Higgs boson
- 3×10^{14} collisions analyzed
- 2.5×10^{15} bytes of data / year in 2012
- Expensive; pressure for fast results, but false positive is embarrassing



Large Hadron Collider at CERN.

Definitions: What's a fault?

Fault or failure?

- Fault[3] affects *inside* behavior; failure propagates *outside*
- Definition of "inside" depends on my responsibilities
- Example: bit flip in a defective memory cell
  - DRAM hardware designer:
    - "Fault" if ECC corrects it
    - "Failure" if ECC can't correct it
  - Algorithm developer:
    - "Fault" *only* if it is read
    - "Failure" if code returns wrong result
- This matters because an algorithm might succeed (not fail), despite failure of hardware to correct a fault.

[3] A possible formal definition: B. Parhami, "Defect, fault, error, . . . , or failure?", IEEE Trans. Reliability, 1997.

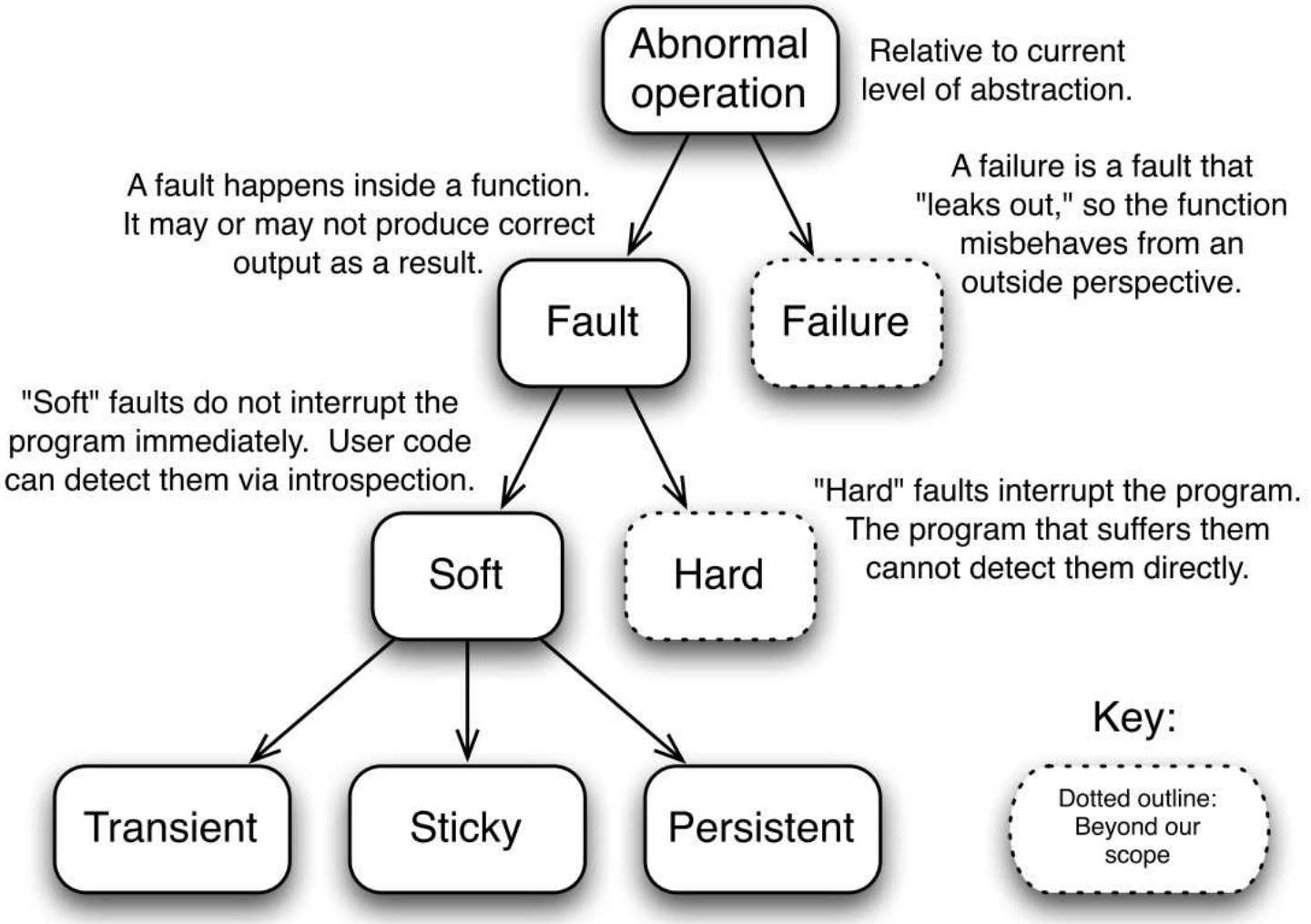
Soft or hard?

- **Hard** breaks "it," **Soft** lets "it" keep going
- What is "it"? Depends on your perspective!
- Hardware designer: "it" = **hardware**
  - "Hard" breaks the hardware permanently
  - "Soft" goes away, perhaps after reset
- Algorithm / application: "it" = **program**
  - "Hard" terminates the program ("crash")
  - "Soft" lets the program keep going
- "Keep going" says nothing about correctness

Transient, "sticky," or persistent?

- Persistently incorrect means broken hardware
  - e.g., Pentium FDIV bug work-around (2× slower)
- Sticky: stays incorrect for a while
  - Goes away after time or an action (e.g., refresh)
  - e.g., uncorrected DRAM / cache bit flips
  - If the original problem got corrupted:
    - Solver will solve $\tilde{A}x = b$, not $Ax = b$
    - Residual $\|b - \tilde{A}\tilde{x}\|$ small, but wrong $\tilde{x}$!
- Transient: happens once, goes away
  - Many existing "resilient" algorithms require transience
  - Transient fault becomes persistent if it corrupts an input
- Can make sticky faults "look transient" by:
  - Recomputing corrupted data (e.g., the matrix)
  - Restoring from a reliable backup

Fault terminology summary



Correct hardware costs performance or power

Correct computer hardware may go away

- The privilege to think of a computer as a *reliable*, *digital* machine for which
  - Arithmetic results are always correct
  - Data in storage always read correctly
- Future hardware may be *silently incorrect*
- Why? *Power & performance*
  - Smaller denser transistors
    - 14nm (Intel KNL) $\approx$ 30 Si atoms (lattice distance)
    - How do we make reliable atom-sized devices?
  - More parallel components (more things to fail)
  - Consumer applications drive hardware development
  - Extreme-scale parallelism is a niche market
  - Improving hardware reliability costs performance or power

Two strategies for handling faults in hardware

Redundancy to detect & correct faults

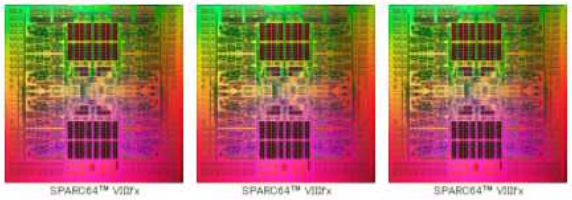


Image credits: Fujitsu, Wikipedia.

Robustness: Absorb fault-causing events

Real hardware combines both, with one of two goals:

- Lower performance, more robust
  - Exotic manufacturing (behind industry performance)
  - Hazardous environments (like outer space)
- Higher performance, higher power
  - Individual components may be less reliable
  - Choice of high-performance computing

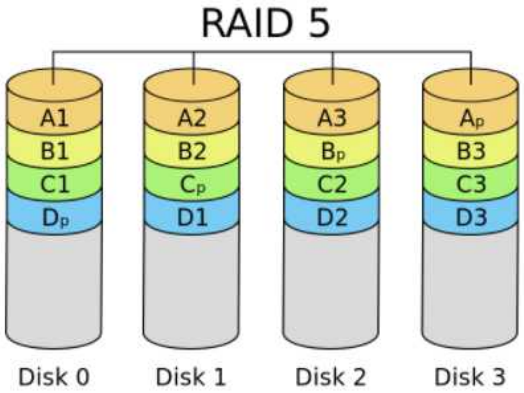
Historical note: Von Neumann and redundancy

- "Probabilistic logics and the synthesis of reliable organisms from unreliable components"
  - *Automata Studies*, 1956
- Sections 7–10
  - Even if each logic gate incorrect with probability ϵ,
  - Can use redundancy to build *arbitrarily* reliable computer
- Practical examples
  - "...a computing machine with 2500 vacuum tubes..."
  - "...a plausible quantitative picture for the functioning of the human nervous system"

John von Neumann (Image credit: Los Alamos Nat. Lab).

Redundancy costs power

- Redundant storage & computation (checksums)
  - ECC (error-correcting) memory
  - RAID (Redundant Arrays of Inexpensive Disks)
  - Software checksums (Algorithm-Based Fault Tolerance)
- Redundant hardware & communication (voting)
  - MPI process replication (in software)
  - Redundant arithmetic units (in hardware)
  - Run it several times ("time replication")



RAID 5 (Image credit: Wikipedia).

Power: hard constraint, decided initially

- Can I power & cool it?
- Often decided first
- For big computers: geography, facilities
- For mobile devices: shape, weight, anatomy
- 2x time / space replication means 2x energy / power

"[S]caling today's systems to an exaflop level would consume more than a gigawatt of power, roughly the output of the Hoover Dam" [1,2]

1. "The opportunities and challenges of exascale computing," Fall 2010, US DOE Office of Science.
2. Image credit: US Bureau of Reclamation.

How much do we need to worry?

- Less reliable hardware may come
- But might not come *soon*
- Vendors work very hard not to sell broken hardware
  - Bad marketing
  - FDIV: $475M recall
  - Legal liability?
- As transistors get *very* small, cosmic rays have *less* effect
- More uncertainty
  - End of CMOS & beyond
  - Vendor trade secrets

Is the sky falling?

How can algorithm developers cope?

- Given uncertainty, seek least invasive solution
  - Avoid asymptotically slower algorithms
  - Use familiar algorithmic ideas & pieces
  - Prefer changes that are always helpful
- Avoid committing to invasive programming models
  - Assume minimum help from system & hardware
  - But influence systems as much as possible
- Our two strategies
  - Skeptical programming
  - Selective reliability

Skeptical programming

Computer generated results need to be treated with some enlightened skepticism. No system or microprocessor can be expected to produce results which are absolutely reliable.

– Prof. Thomas R. Nicely (discovered Pentium FDIV bug)

- Model faults as incorrect intermediate results
- Distrust & check intermediate results, cheaply
- Detect "bad" faults & roll back
- "Converge through" all other faults if possible

Fault model: Incorrect intermediate results

- Any computational kernel could return the wrong answer
- *Not* random bit flips!
  - Binary number representation is not uniform
    - "Expected value" of a bit flip: huge / infinite
    - 52 / 64 bits of `double`: change $\leq 1/2$
  - Bit flips just change values; why flip bits?
  - We lack a good model for hardware faults
- Assumes transience
  - Read-only state external to function not corrupted
  - Can simulate transience by recomputing or rereading

Distrust & check intermediate results

- Numerical algorithms have natural nested structure
  - Iterative eigenvalue solve with shift and invert
    - Large sparse linear solve inside
    - LAPACK & BLAS calls inside that
  - Time-dependent partial differential equation
    - Nonlinear equation at each time step
    - Linear equation at each nonlinear iteration
- Why do you trust software written by humans?
- Can check solution to any equation $f(x) = 0$
- But don't take too long to check!

Selective reliability

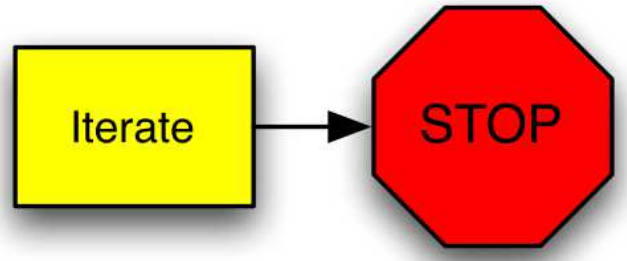
Reliability model lets us prove algorithms correct

- Reliability model: when & where faults may occur
- If faults can happen anywhere or anytime:
  - Can't prove algorithms correct
  - Can't even do time replication (esp. if faults "sticky")
- Analogy: rounding error in floating-point arithmetic
  - IEEE 754 & 854 provide invariants (e.g., exact rounding)
  - Invariants let me bound my algorithm's error
  - Language lets me control error (by declaring precision)
- Programmers cannot provide invariants themselves!
  - Just like IEEE 754, the hardware & system must help

"Floating-point code is just like any other code: it helps to have provable facts on which to depend" (David Goldberg).

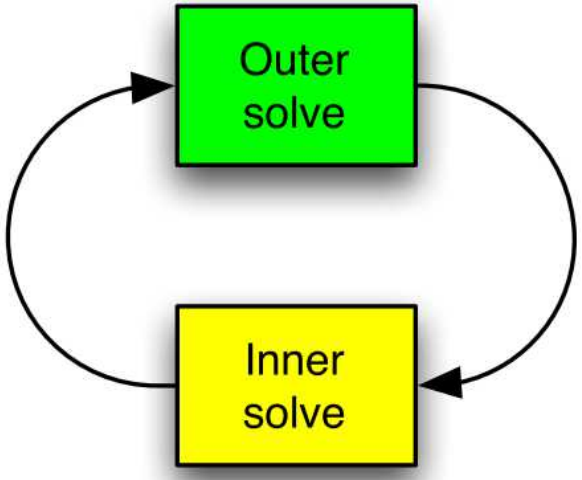
Reliability models

Current model: Fail-stop



- System tries to detect all soft faults
- Turn all detected soft faults into hard faults
- Detected local faults become global
- Checkpoint / restart is the only recovery model

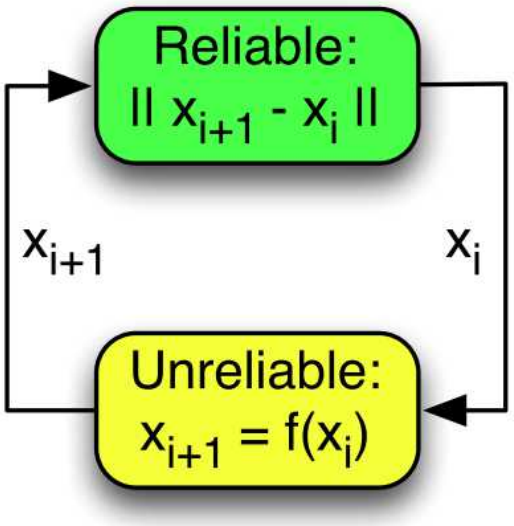
Better model: Sandbox



- Isolate unreliable data & computation in a box
- Reliable code invokes box
- Local faults stay local
- App gets flexibility to define recovery model

Example: Sandbox model for fixed-point iteration

- Solve $f(x) = x$: iterate $x_{i+1} = f(x_i)$ until $\|x_{i+1} - x_i\| < \epsilon$
- If faults can happen anywhere, can't trust convergence test
- Protect convergence test & put f evaluation in sandbox
  - If convergence test passes, we know x_{i+1} is correct
  - If enough iterations pass without a fault, we will converge
  - If f time-consuming, then most time in unreliable mode
- This generalizes to nearly any iterative algorithm!



Improvements on sandbox model

- Detection: report faults to application
  - Detection often cheaper than correction
  - Hardware (e.g., DRAM) can already do this
  - Operating system doesn't expose this (yet)
- Transience: make faults not "stick"
  - Sticky faults more harmful to algorithms
  - Example: $\|b - Ax_k\|_2$ when A is wrong
  - Refresh or recompute unreliable data periodically
  - Algorithm must expose what data need refreshing
- None necessary; all useful
  - Detection allows approximate correction (see later).

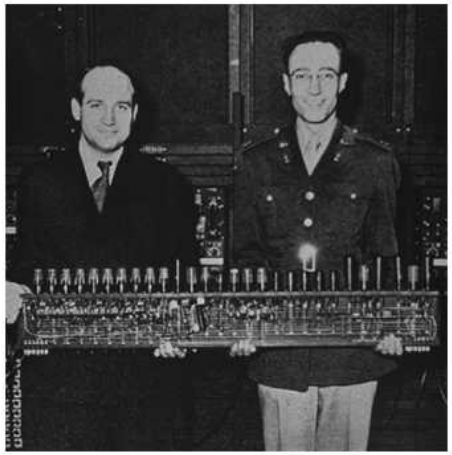
Algorithms

Rounding error analysis gives us hope

Example: Gaussian elimination

- Pessimistic worst-case theory (Hotelling 1943)
- Yet tolerates rounding errors in both theory and practice
- They succeeded for rounding error; hardware faults?

James H. Wilkinson



John von Neumann & Herman Goldstine

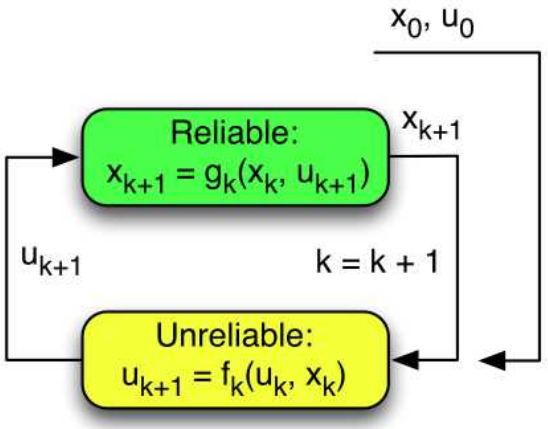
Alan Turing

Desired properties of a fault-tolerant iteration

- Exploit reliability model for predictable behavior
- Converge eventually, or detect & indicate failure
  - Not true of iterative refinement!
- Convergence degrades gradually as fault rate increases
  - Easy to trade between reliability and extra work
- Require as little reliable computation as possible
- Avoid global communication / synchronization
  - If faults are local, recover locally
  - Recover locally first, even if approximate
  - Defer exact, possibly global recovery

How can we guarantee convergence?

A more generic iteration for solving some equation $g(x) = x$:



- x_k is current approximate solution; u_k is auxiliary
- Initial guesses x_0, u_0
- for k in $0, 1, 2, \ldots$:
  - $u_{k+1} = f_k(u_k, x_k)$
  - $x_{k+1} = g_k(x_k, u_{k+1})$

- f_k may experience faults; g_k may not
- g_k must converge eventually, for any sequence of f_k, or clearly detect if it can't converge
- Monotonic convergence would help gradual degradation

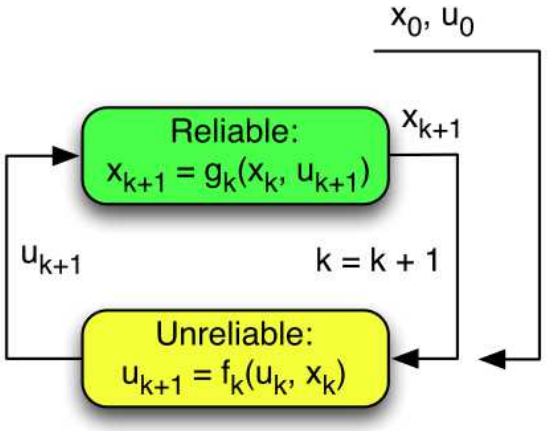
Brief review of iterative linear solvers

- Solve linear system $Ax = b$
  - Gaussian elimination not feasible / too expensive
  - Can compute matrix-vector products $v = Au$
  - Have M ("preconditioner") with $AM \approx I$

1. Pick initial guess x_0, compute $r_0 = b - Ax_0$
2. x_k is x_0 plus linear combination of $(AM)^k r_0$
3. Compute search directions via matrix-vector products
4. Repeat until norm of $r_k = b - Ax_k$ is small

How does this map to solving $Ax = b$?

Imagine generic iteration as an iterative solver for $Ax = b$:



- for k in $0, 1, 2, \ldots$:
  - $u_{k+1} = f_k(u_k, x_k)$
  - $x_{k+1} = g_k(x_k, u_{k+1})$

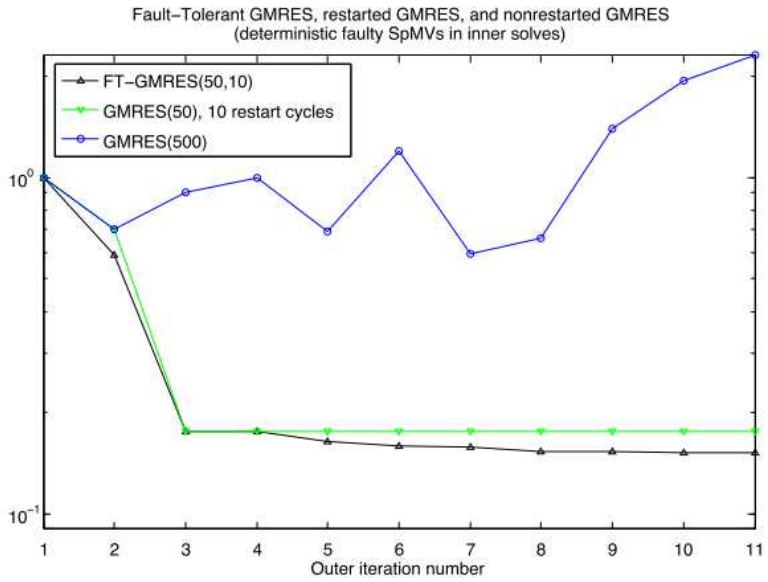
- f_k: apply the preconditioner
  - May be different each k, perhaps because of faults
- $\Rightarrow$ Need a "flexible" method
- Monotonic convergence $\Rightarrow$ (Flexible) GMRES

Fault-Tolerant GMRES

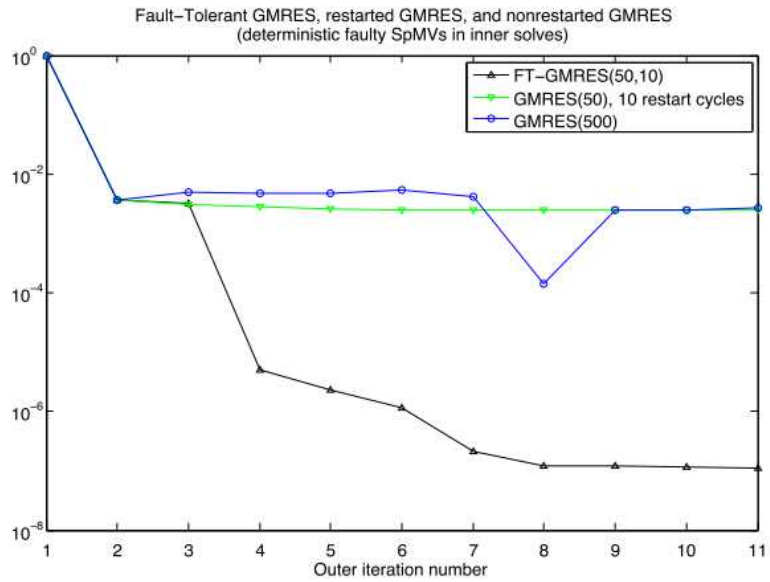
- "Outer solver" g_k is Flexible GMRES, run reliably
  - Converges eventually (or tells you otherwise)
    - Detect failure cheaply and locally
    - ($O(k^2)$ arithmetic operations at iteration k)
    - Same failure condition as without faults
  - Flexible: Allows arbitrary preconditioner changes
    - Any bit in floating-point word may flip
    - Fault = "changing" preconditioner
- "Inner solver" f_k: your current solver & preconditioner
  - Should take most of the time
  - Preconditions the outer solver
- Can reuse existing software stack for inner solver

FT-GMRES can run through faults

- FT-GMRES can run through faults and still converge.
- Standard GMRES, with or without restarting, cannot.



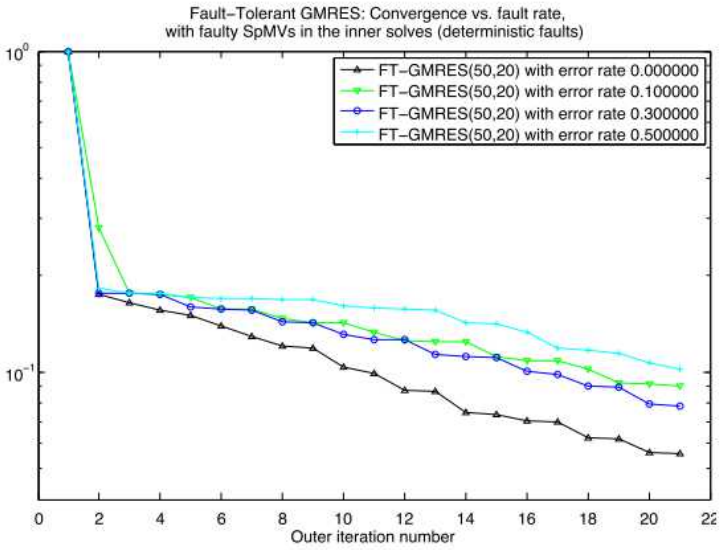
FT-GMRES vs. GMRES on Ill_Stokes (an ill-conditioned discretization of a Stokes PDE).



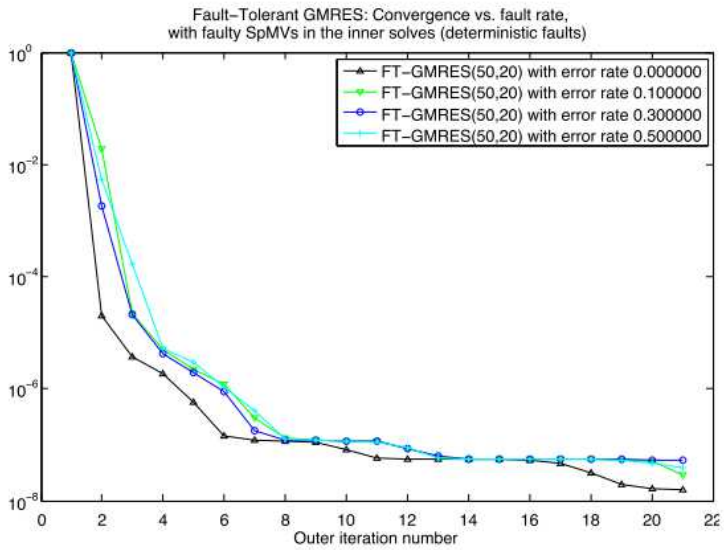
FT-GMRES vs. GMRES on mult_dcop_03 (a Xyce circuit simulation problem).

Observed gradual degradation of convergence

- Empirical observation: FT-GMRES convergence slows gradually as fault rate increases.
- FT-GMRES only; vary fault rate in inner solves' $A \cdot x$.



Ill_Stokes problem.



mult_dcop_03 problem.

Future work

Future work

- Do we even need to worry about linear solves?
  - Linear $\rightarrow$ nonlinear $\rightarrow$ ODE $\rightarrow$ optimization $\rightarrow$ UQ
  - Some levels robust to incorrect results at lower levels
  - Do we expect a fault in several of the linear solves?
- Comparing cost of different resilience approaches
  - Would it have been cheaper just to start over?
  - Should I let the level above (or below) me handle faults?
  - Checkpointing every 5 CG iterations is ridiculous
- What about discrete problems?
  - Easy checks (e.g., did sorting work?) help if faults rare
  - Relax into continuous optimization problem?
  - Learn to tolerate approximate results?

Conclusions

- Algorithms can prepare for incorrect hardware right now
- This also protects them against incorrect software
- Promising correctness requires a fault model
- Extreme-scale computing cannot succeed without codesign
- Lots of problems left to solve
  - Algorithms: nonlinear, preconditioners, ...
  - Programming models and system support
- Algorithm developers must talk to systems researchers

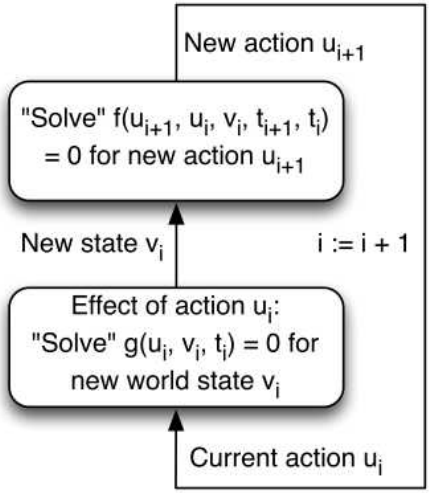
Thanks for listening!

Bill Gates at COMDEX on 20 April 1998.

Extra slides

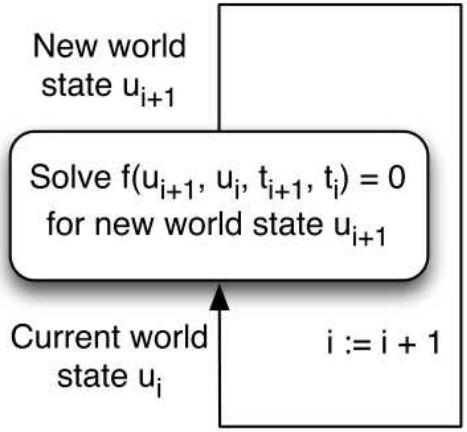
Real-time control system vs. PDE discretization

Real-time control system



The world always computes physical laws correctly. Negative feedback may reduce effect of $f(\dots) = 0$ errors.

Time-dependent PDE discretization



No physical feedback. $f(\dots) = 0$ solve must be correct or mistakes will accumulate.

Detect bad errors and roll back

- We define "bad" errors by their cost
  - Cause solver to fail completely
  - Increase iteration count by too much
- We may not be able to catch all bad errors
- Roll back: Prefer transactional semantics
  - Don't need fancy transactional memory
  - Write code so it has no side effects if it fails
  - C++ calls this the *strong exception guarantee*
  - "[N]ever let go of a piece of information before we can store its replacement" (Stroustrup)