

Scalable Systems Software and Failure

Ron Minnich

Sandia National Labs (SAND 2008-2005C)



Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company,
for the United States Department of Energy's National Nuclear Security Administration
under contract DE-AC04-94AL85000.

What's with the boat?

- It was actually a good ship
- But it was not treated well (cf. Pellegrino)
- Compounding of bad decisions based on unrealistic expectations
- Expectations outran reality
 - “God Himself could not sink this ship” -- Marketing

This talk was hard

- I'm supposed to be doing something about failure and systems software at different levels
- What's failure mean?
- User: “app did not work”
- What can cause such failure?
- Can failures *always* be mitigated by systems software?

Simple (recent) example

- Edit file on file server
- Spawn job on 1024 nodes
- Jobs see old, not new file: “it's an OS bug”
- What *really* happened?
- $T_{\text{Propagate}} \gg T_{\text{spawn}}$
- All systems work to plan
- Result is failure

Failure from bottom to top

- Hardware
- OS
- OS plugins (e.g. Parallel File Systems)
- Libraries
- Compilers
- Apps

Hardware

- Stuff you can figure out
 - temps, volts, etc.
 - except the numbers are not always right
- Stuff you can't figure out
 - Bad pass and bias resistor on SPD bus
 - voids in voltage regulator module solder
 - outsize PCI card that damages PCI socket
 - PCI chipsets that work *almost* all the time

Everyone always mentions the easy stuff

- Tend to ignore the hard stuff
 - which makes sense: it's hard
- Everyone can point to the “if a fan fails” or “if I get hot” example
- How well can they do?
- Take one real case: 30-second CPU heatup
- Sample whole cluster at 15 second intervals

Getting that data

- Want temps, fans, volts, ... the usual stuff
- And to manage all that, we use ...
- The best 8-bit micro money can buy!
- And tie it to ...
- The worst protocol ever designed -- IPMI
- And it can't keep up *or* give useful answers
- Gee, if it is so important, can't we do better?

Doing better?

- Other tools are not much better, e.g. Perl scripts
- To get statistics these tools:
 - open/read/close proc files and run programs
- They're slow and resource-hungry
- Hence run at 10 min intervals
- So we fail: can't catch 30-second heatup
- Note: can't even catch it on XT3 RAS network!

Did we fail in reality?

- No: we used the supermon kernel module
- And supermon daemons
- No visible impact with 15-second sample interval
- We could catch the overheat case and take systems offline
- Failure management is possible with right tools

If we want to get serious about hardware failure

- We have to get serious about hardware monitoring
- Which means a kernel driver
 - It has to be in-band since so many on-CPU registers can only be read in-band
- U. Oregon showed we can do in-band high-rate monitoring with no application degradation

But how much trouble does hardware really cause?

- I'm not convinced, in a well-designed cluster, that it should dominate
- Real-world experience at LANL says you can build lights-out clusters that fail once a year or less
- So it must be the OS, right?

OS failure

- Has not been a huge issue for me anyway
- Clusters reboot for building shutdown, or maintenance, or ...
- but they don't seem to reboot because of Linux
- Which makes a kind of sense
- IBM runs 6000 Linux instances at a time on Z-series

OS modules

- A different story
- The really big problems I'm used to are Parallel File Systems
- Most seem to be finicky
- The Google ones seem to be very solid
- What's different?

What's different:

Failure model (i.e expectations)

- One failure model: spend your way out of trouble
- Google model: failure is a constant; accept and deal with it
- Result is a far higher degree of reliability than we are used to
- Expectations drive architecture

What else is different

- Many DOE parallel file systems -- kernel modules -- developed *external to the kernel*
- I.e. not available at kernel.org
- The “external module” development model is known to produce more failure-prone code
- If it's a kernel module it belongs in mainline kernel, not built/developed externally
- Why I'm doing my work with PNFS

OS vs. OS module failure

- The OS probably won't fail -- it's pretty hardened
- External modules -- e.g. our parallel file systems -- will have higher failure rate
- If they fail, app will probably go with it
- Since the user expectation is that these subsystems never fail
- And OS has same expectations

Libraries and Compilers

- The main problems I'm aware of are bugs etc.
- How often have you heard “use x.y.z version” or “don't compile *that* app *with* this compiler”
- Most recent good find: CLD flag and Linux
 - “Can't build Linux with newer gcc”
- If compiler builds a bad kernel, you're dead -- sooner or later

MPI libraries

- MPI works well for 2^{16} CPUs
- MPI users have certain expectations
 - I.e.: we can keep all nodes running long enough to load/run from/create a checkpoint
- Can we support these expectations with 2^{26} CPUs?

MPI library fault tolerance approach: more code

- Add more and more complexity for failure
 - Is this a downward spiral? probably
- Did anyone ever make something simpler and more reliable by making it complex-er?
- I do not think we can support user's MPI expectations in a 2^{26} CPU world

Applications

- Applications programmers expectations can not be sustained
- We can't keep pretending that a distributed system with 1,000,000 CPUs is a single computer with one disk
- Many causes of failure derive from unrealistic expectations (as from first example)

Resilience in the face of failure

- Failures at all levels will be a continuous event
- Future million-CPU systems can not:
 - be “all up” at one time
 - provide an unchanging subset of CPUs that all stay up for a “load/run/checkpoint” interval
 - provide a constant file system appearance to all CPUs
- Sounds like fun! Can I retire soon?

Can't we just predict all our failures?

- Except for the unpredictable ones, yes
 - And you know they're out there ...
 - Can *you* predict the “tin whiskers”?
- Design for unpredictability
- Build the apps to be *fault-oblivious*, not *fault-tolerant*
- That way we're safe against all failures, not just the ones we think we can predict

Conclusions

- Failure management is driven by expectations
- Unrealistic expectations make for failure-sensitive systems
- Put less effort into eliminating failure
- More effort into fault-oblivious software
- Build systems that tolerate faults and hence are more reliable for applications
 - Simple real-world example: ECC