Challenges in Fault-Tolerance for Peta-ExaScale systems and research opportunities

Franck Cappello
INRIA & UIUC
fci@lri.fr

Failure rate in large systems

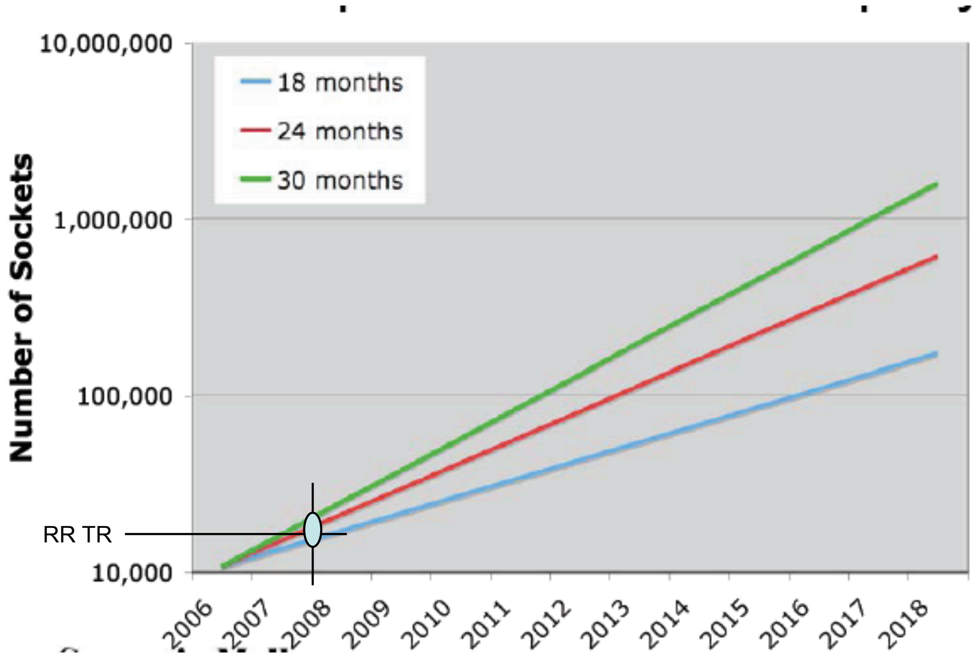
In Top500 machine performance X2 per year

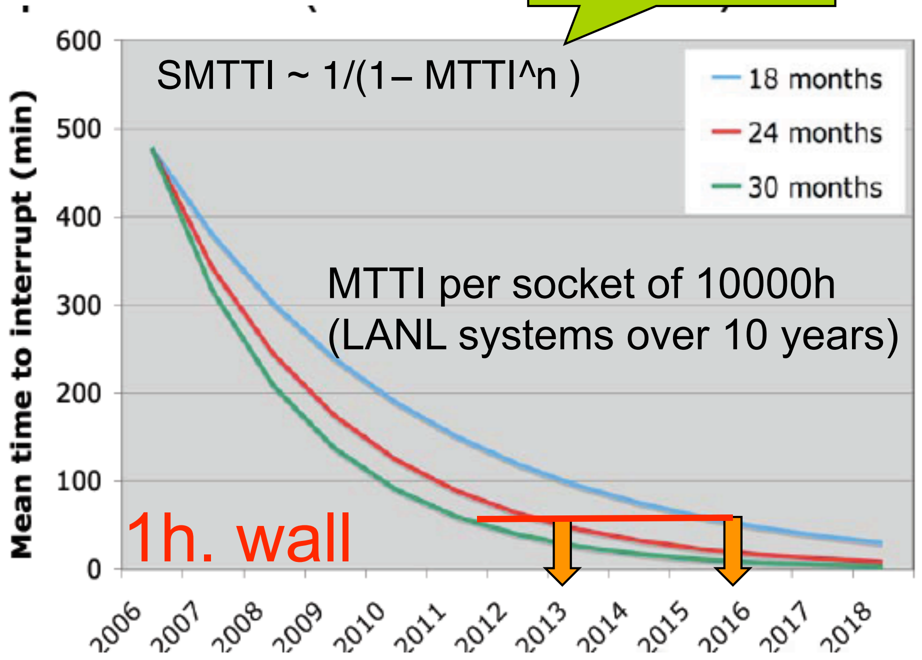
→ more than Moore's law and the increase of #cores in CPUs

→The number of sockets in these systems is increasing.

→No evolution of MTTI per Socket over the past 10 years

Figures from Garth Gibson





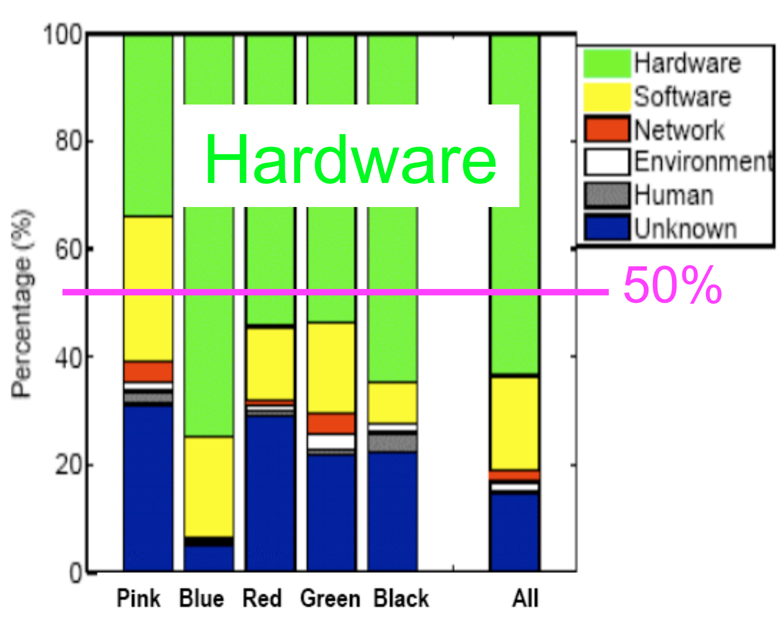
SMTTI may reach 1h. as soon as in 2013-2016 (before Exascale?)

Another projection from CHARNG-DA LU gives similar results

MTTI of 10000h is considering all kinds of faults (software, hardware, human, etc.)

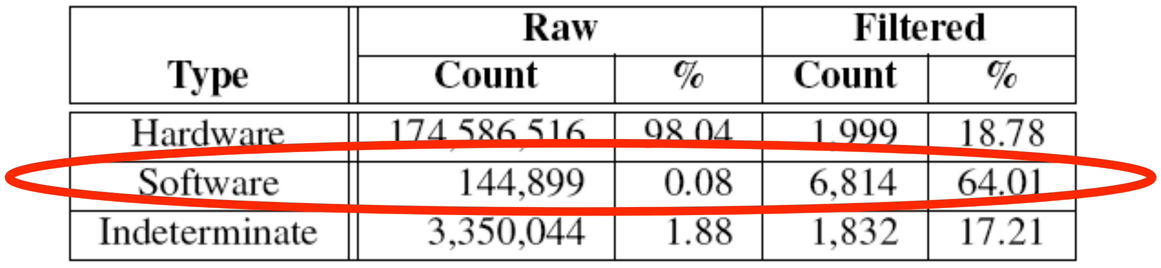
Sources of failures

- Analysis of error and failure logs
- In 2005 (Ph. D. of CHARNG-DA LU) : "Software halts account for the most number of outages (59-84 percent), and take the shortest time to repair (0.6-1.5 hours). Hardware problems, albeit rarer, need 6.3-100.7 hours to solve."
- In 2007 (Garth Gibson, ICPP Keynote):



Relative frequency of root cause by system type.

- In 2008 (Oliner and J. Stearley, DSN Conf.):



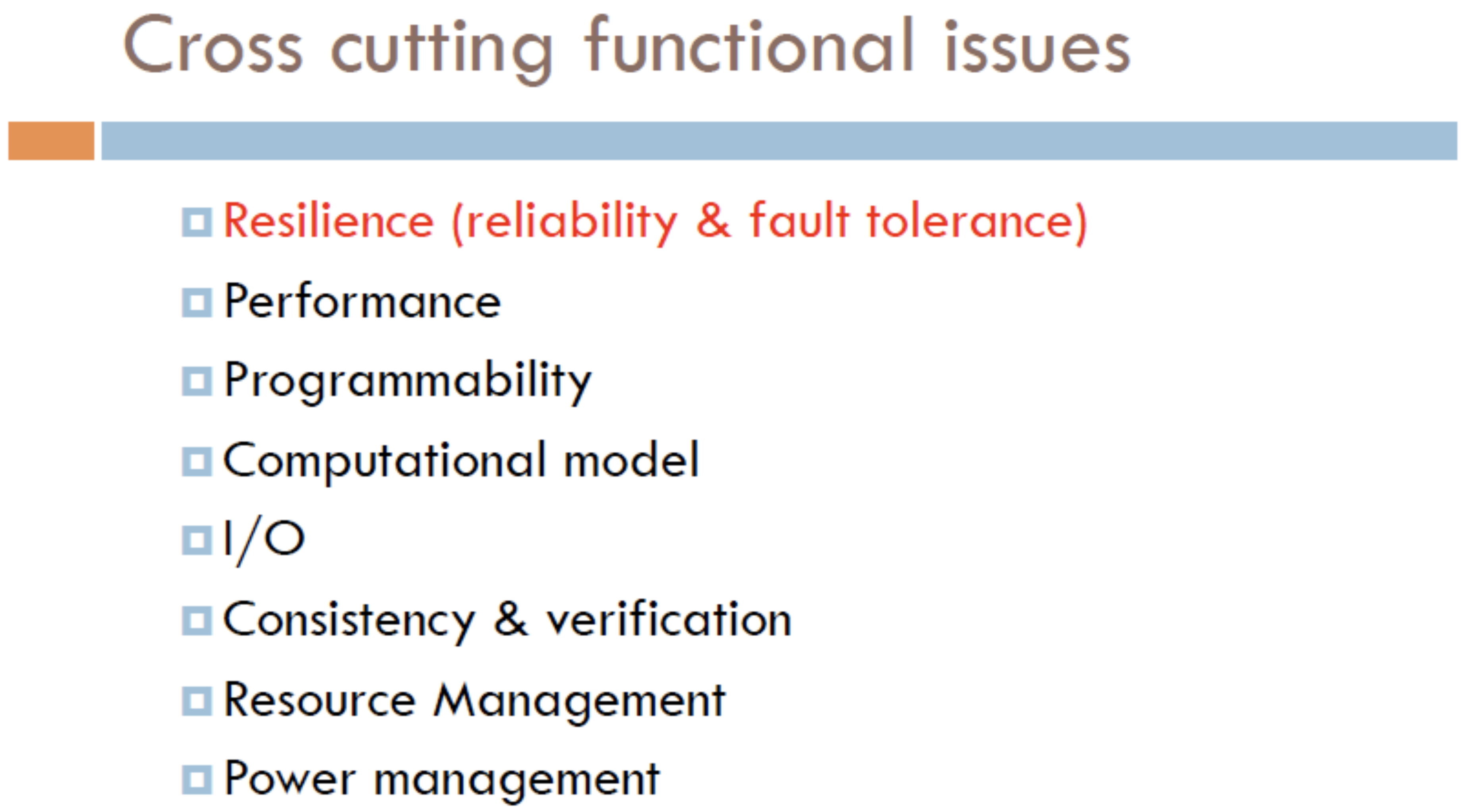
Type	Raw		Filtered	
	Count	%	Count	%
Hardware	174,586,516	98.04	1,999	18.78
Software	144,899	0.08	6,814	64.01
Indeterminate	3,350,044	1.88	1,832	17.21

Software errors: Applications, OS bug (kernel panic), communication libs, File system error and other.

Hardware errors, Disks, processors, memory, network

Conclusion: Both Hardware and Software failures have to be considered

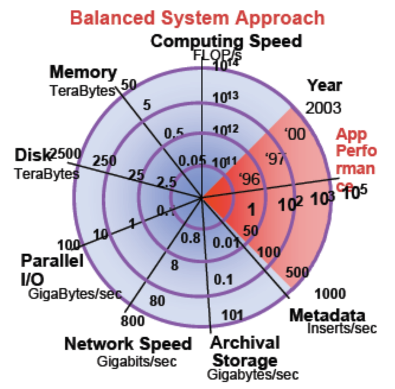
International Exascale Software Project (IESP)



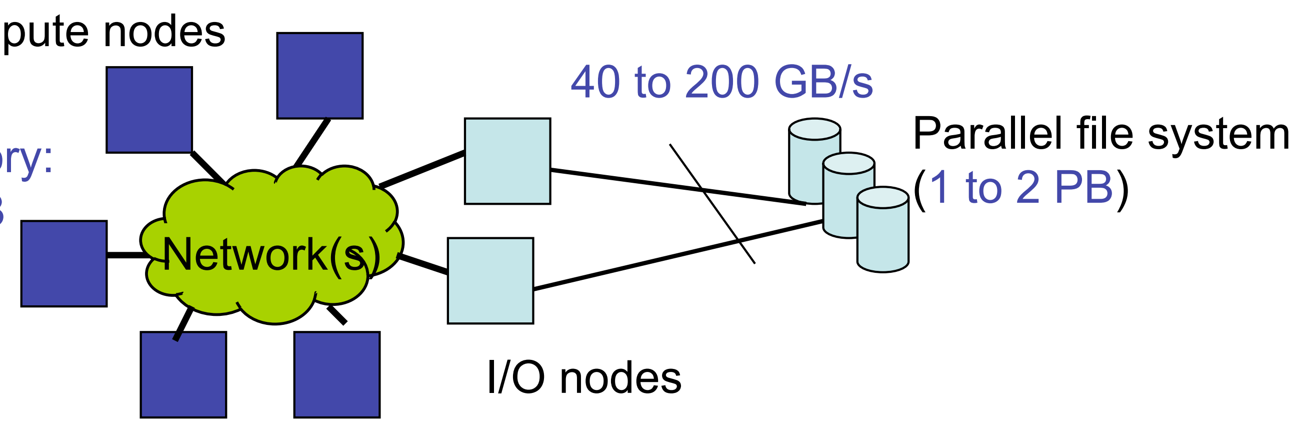
Cross cutting functional issues

- Resilience (reliability & fault tolerance)
- Performance
- Programmability
- Computational model
- I/O
- Consistency & verification
- Resource Management
- Power management

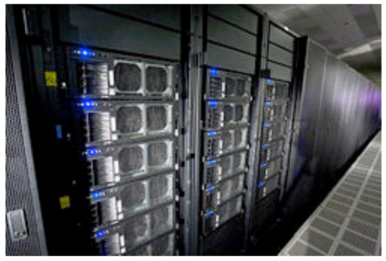
Classic approach for FT: Checkpoint-Restart



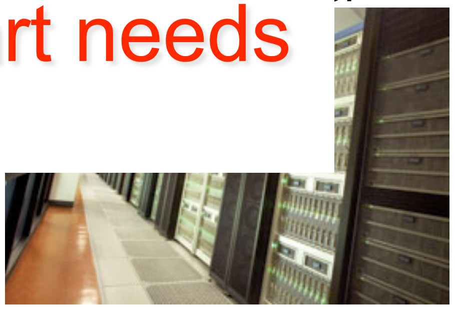
Typical “Balanced Architecture” for PetaScale Computers



RoadRunner



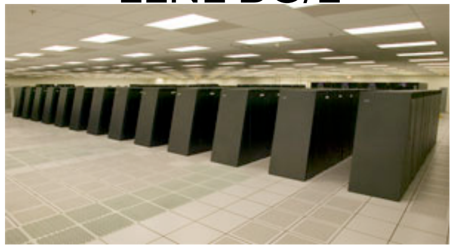
TACC Ranger



Without optimization, Checkpoint-Restart needs about 1h! (~30 minutes each)

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

LLNL BG/L



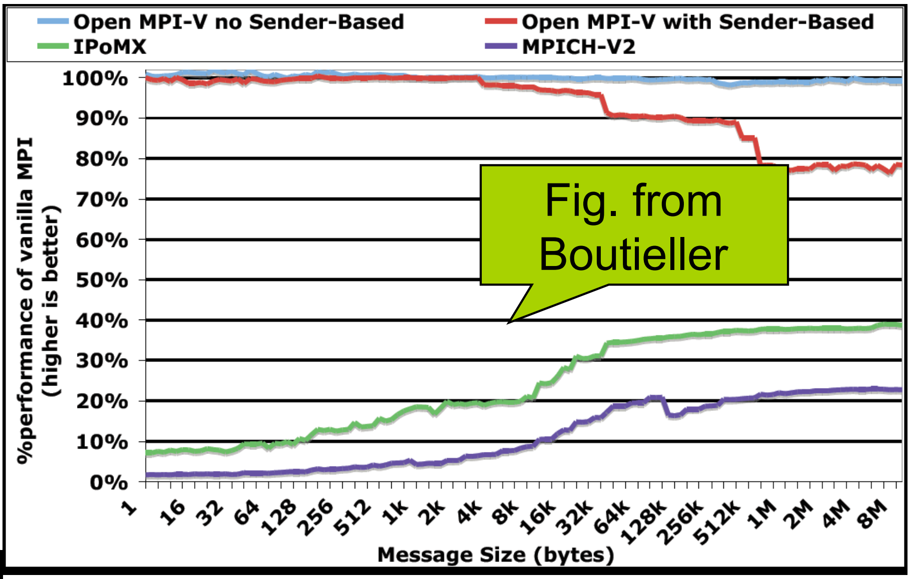
Roll-Back Recovery Protocols?

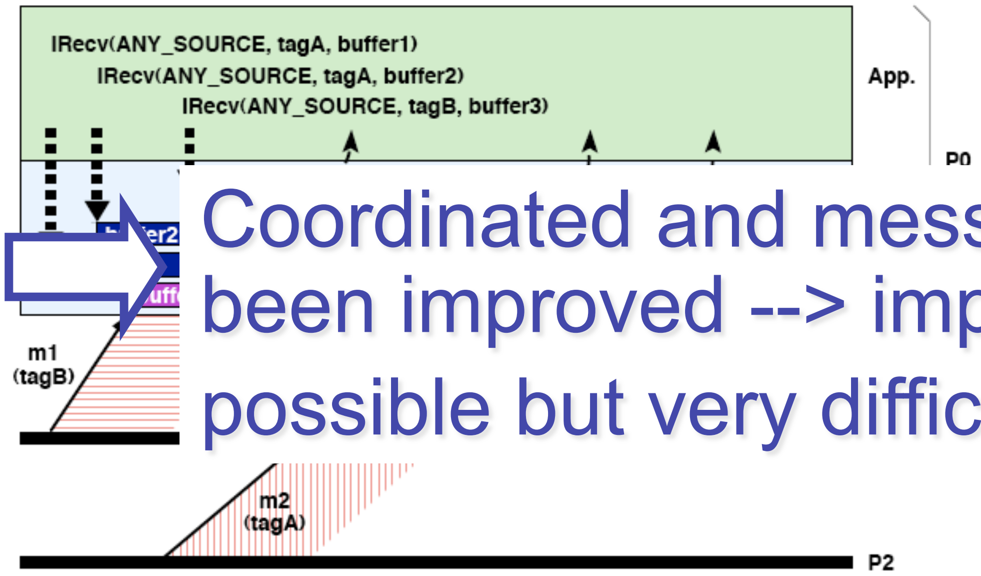
•Classic approach (MPICH-V) implements Message Logging at the device level: all messages are copied

•High speed MPI implementations use Zero Copy and decompose Recv in: a) Matching, b) Delivery

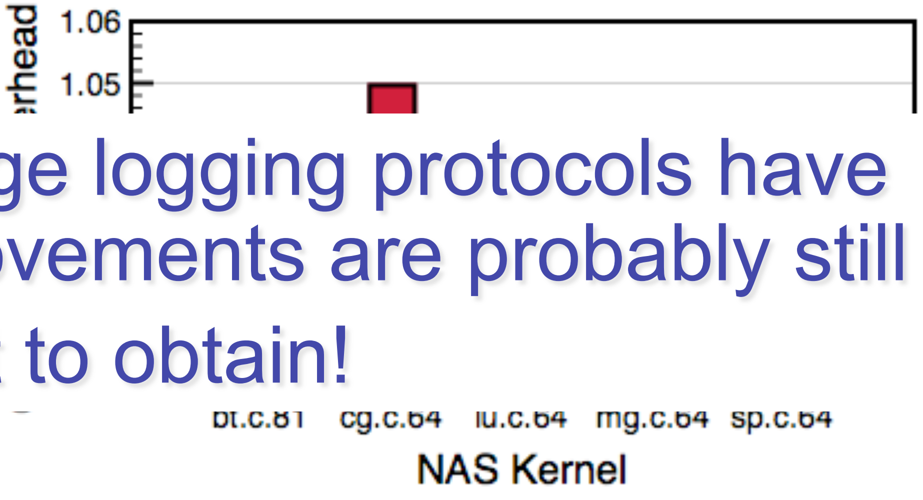
•OpenMPI-V implements Mes. Log. within MPI: different event types are managed differently, distinction between determ. and non determ. events, optimized mem. copy

Bandwidth of OpenMPI-V compared to others




IRecv(ANY_SOURCE, tagA, buffer1)
IRecv(ANY_SOURCE, tagA, buffer2)
IRecv(ANY_SOURCE, tagB, buffer3)
App.
P0
m1 (tagB)
m2 (tagA)
P2

OpenMPI-V Overhead on NAS (Myri 10g)



Coordinated and message logging protocols have been improved --> improvements are probably still possible but very difficult to obtain!

Reducing Checkpoint size 1/2

- Incremental Checkpointing:

A runtime monitor detects memory regions that have not been modified between two adjacent CKPT. and omit them from the subsequent CKPT.

OS Incremental Checkpointing uses the memory management subsystem to decide which data change between consecutive checkpoints

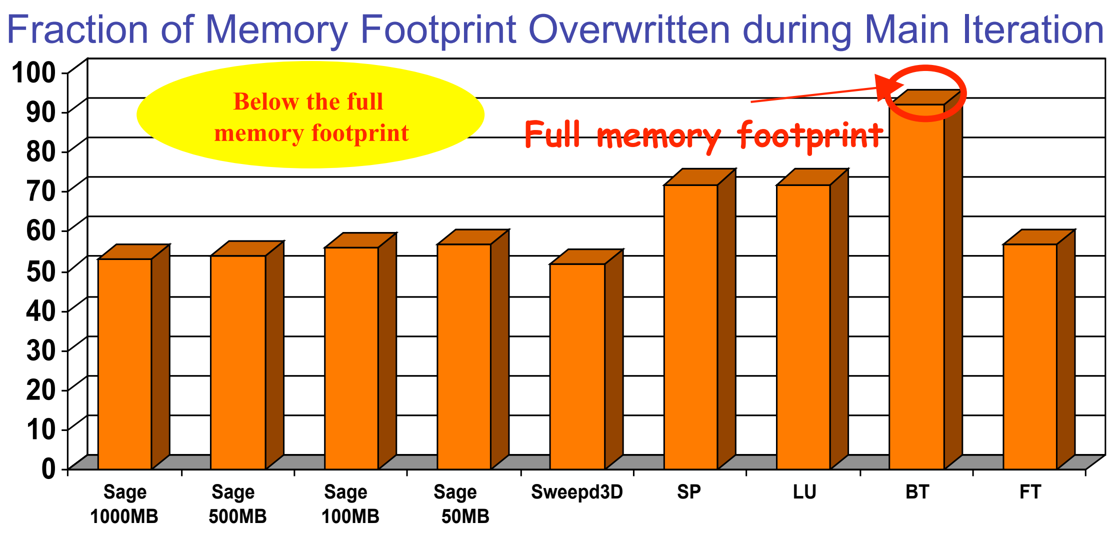


Fig. from J.-C. Sancho

- Application Level Checkpointing

"Programmers know what data to save and when to save the state of the execution".
Programmer adds dedicated code in the application to save the state of the execution.

Few results available:

Bronevetsky 2008: MDCASK code of the ASCI Blue Purple Benchmark

→Hand written Checkpointer eliminates 77% of the application state

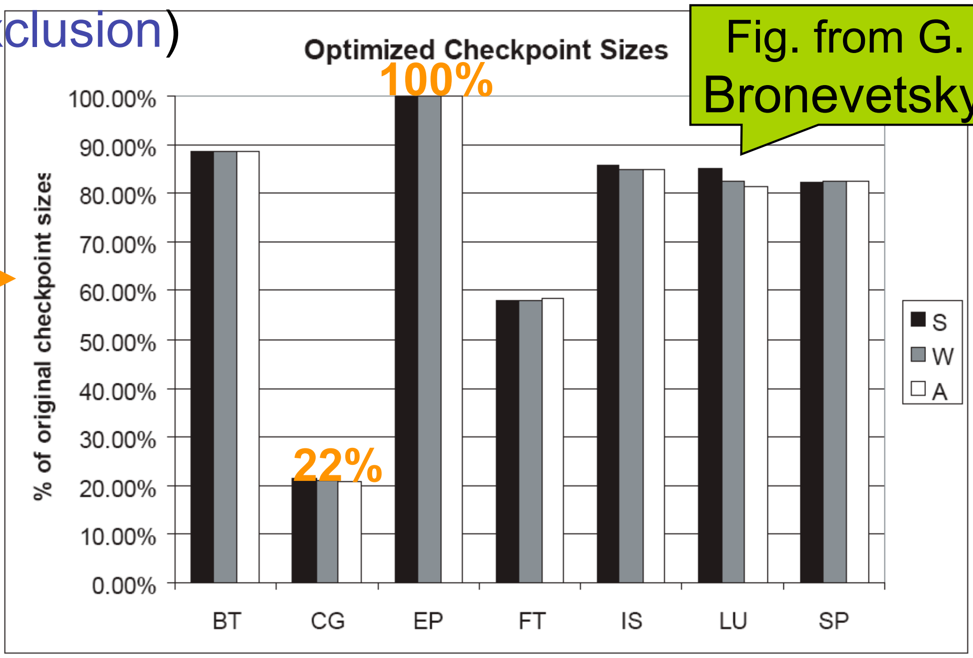
Limitation: impossible to optimize checkpoint interval (interval should be well chosen to avoid large increase of the exec time --> cooperative checkpointing)

Reducing Checkpoint size 2/2

Compiler assisted application level checkpoint

•From Plank (compiler assisted memory exclusion)
•User annotate codes for checkpoint
•The compiler detects dead data (not modified between 2 CKPT) and omit them from the second checkpoint.
•Latest result (Static Analysis 1D arrays) excludes live arrays with dead data:
--> 45% reduction in CKPT size for mdcask, one of the ASCI Purple benchmarks

Fig. from G. Bronevetsky



•Inspector Executor (trace based) checkpoint (INRIA study)
Ex: DGETRF (max gain 20% over IC)
Need more evaluation

Memory addresses

Reducing checkpoint size is still a very difficult problems).

Alternative Approaches

System side:

- Diskless checkpointing
- Proactive actions (proactive migration),
- Replication (mask the effect of failure),

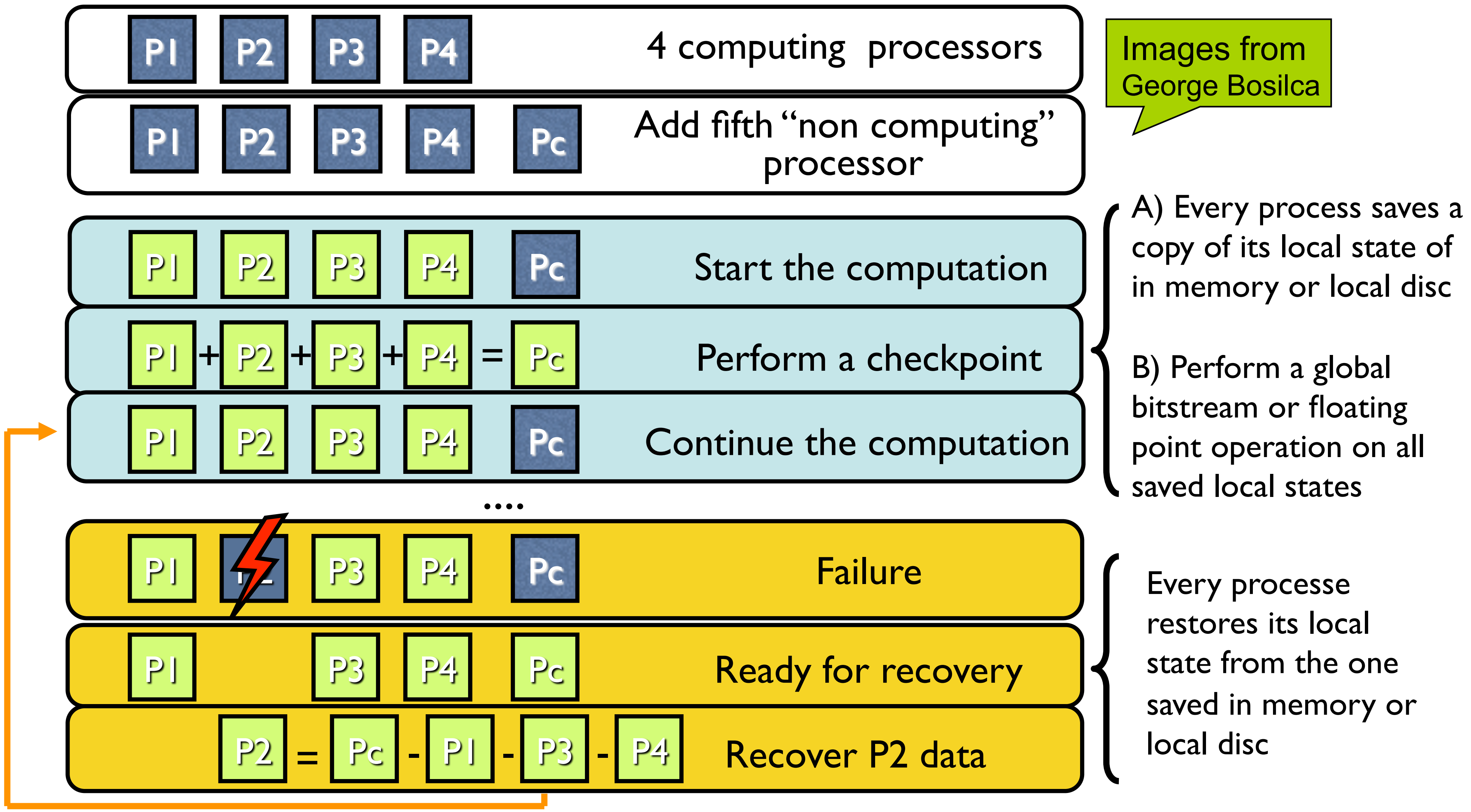
From applications&algorithms side: “Failures Aware Design”:

- Algorithmic Based Fault tolerance (Compute with redundant data),
- Naturally Fault Tolerant Algorithms (Algorithms resilient to failures).

Diskless Checkpointing 1/2

Principle: Compute a checksum of the processes' memory and store it on spare processors

Advantage: does not require ckpt on stable storage.



P1 P2 P3 P4 — 4 computing processors
P1 P2 P3 P4 Pc — Add fifth "non computing" processor
P1 P2 P3 P4 Pc — Start the computation
P1 + P2 + P3 + P4 = Pc — Perform a checkpoint
P1 P2 P3 P4 Pc — Continue the computation
....
P1 P2 P3 P4 Pc — Failure
P1 P3 P4 Pc — Ready for recovery
P2 = Pc - P1 - P3 - P4 — Recover P2 data
Images from George Bosilca

A) Every process saves a copy of its local state of in memory or local disc

B) Perform a global bitstream or floating point operation on all saved local states

Every processe restores its local state from the one saved in memory or local disc

Diskless Checkpointing 2/2

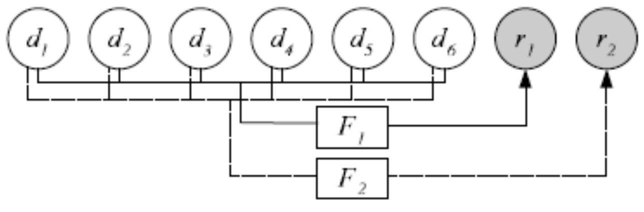
Images from CHARNG-DA LU

•Could be done at application and system levels

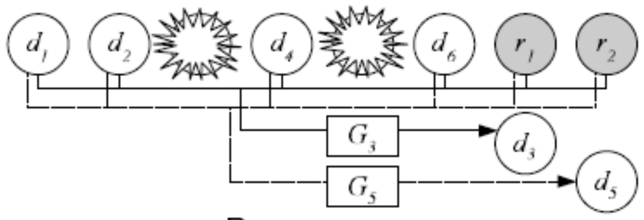
•Process data could be considered (and encoded) either as bit-streams or as floating point numbers.
Computing the checksum from bit-streams uses operations such as parity. Computing checksum from floating point numbers uses operations such as addition

•Can survive multiple failures of arbitrary patterns
Reed Solomon for bit-streams and weighted checksum for floating point numbers (sensitive to round-off errors).

•Work with with incremental ckpt.



Encoding



Recovery

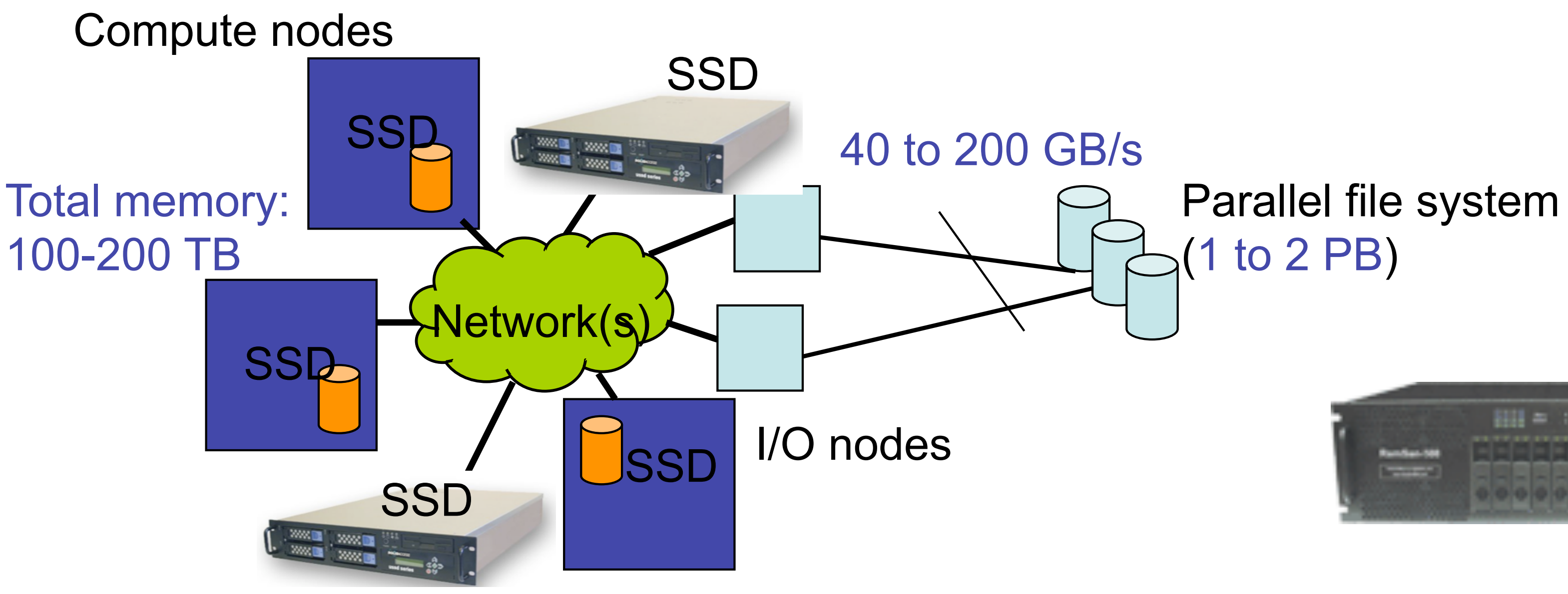
BUT

•Need spare nodes and **double the memory occupation** (to survive failures during ckpt.) --> increases the overall cost and #failures

•**Need coordinated checkpointing or message logging protocol**

•Need very fast encoding & reduction operations

Store ckpt. on SSD (Flash mem.)



Use SSD (Flash mem.) in nodes or attached to network

Compute the checksum (or more complex encoding) of the memory (1 node by one or by partitions)

Distribute the result on SSD device clusters (from 1 to the whole system).

Downside:

→Increases the cost of the machine (100 TB or flash memory)

→Increases the # of components in the system

→ increase power consumption

Proactive Operations ex: Proactive Migration

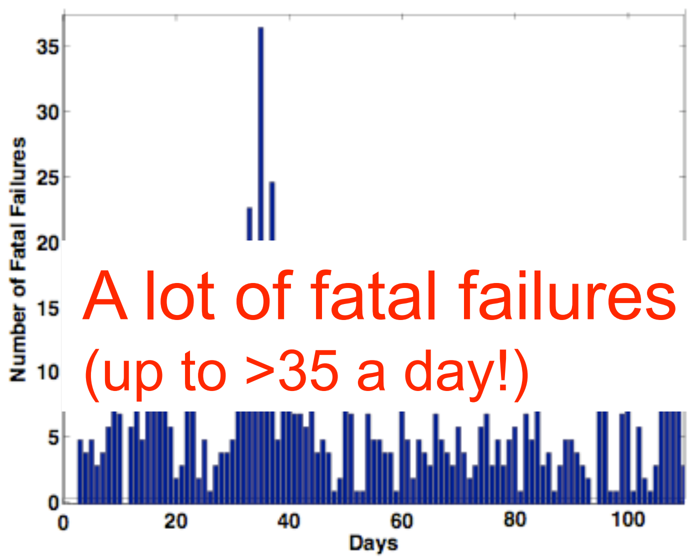
•Principle: predict failures and trigger preventive actions when a node is suspected

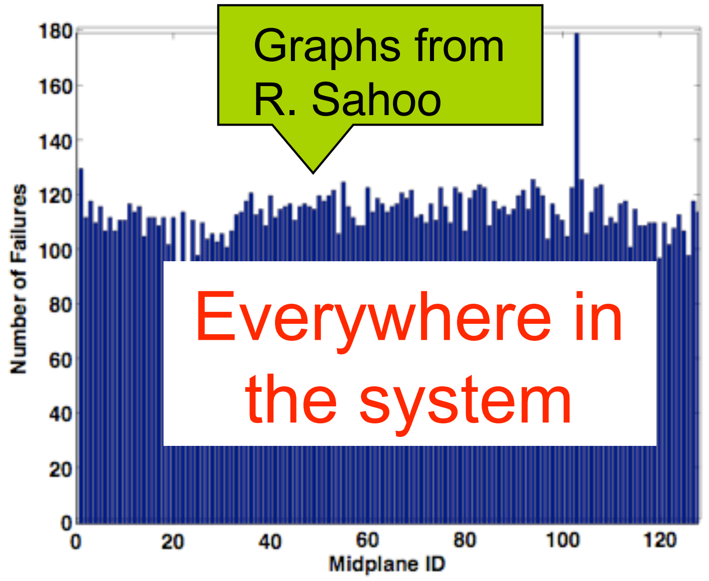
•Many researches on proactive operations assume failures could predicted.

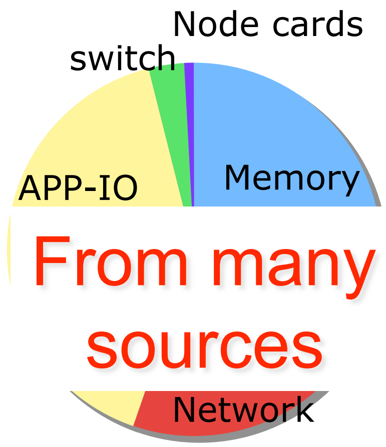
Only few papers are based on actual data.

•Most of researches refer 2 papers published in 2003 and 2005 on a 350 CPUs cluster and and BG/L prototype (100 days, 128K CPUs)

BG/L prototype

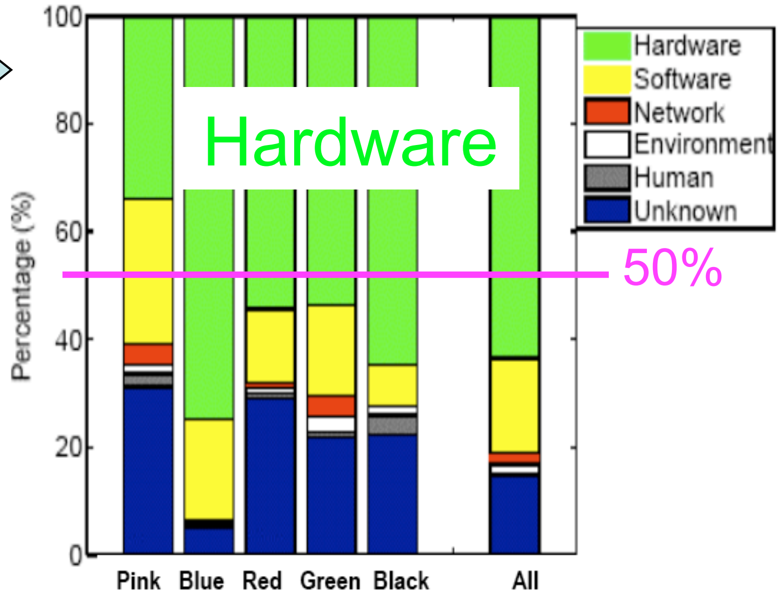






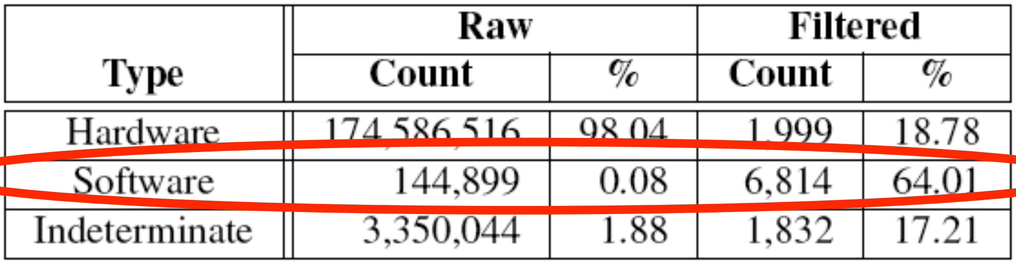
What are the main reasons of failures?

- In 2005 (Ph. D. of CHARNG-DA LU) : “Software halts account for the most number of outages (59-84 percent), and take the shortest time to repair (0.6-1.5 hours). Hardware problems, albeit rarer, need 6.3-100.7 hours to solve.”
- In 2007 (Garth Gibson, ICPP Keynote):



Relative frequency of root cause by system type.

- In 2008 (Oliner and J. Stearley, DSN Conf.):



Type	Raw Count	Raw %	Filtered Count	Filtered %
Hardware	174,586,516	98.04	1,999	18.78
Software	144,899	0.08	6,814	64.01
Indeterminate	3,350,044	1.88	1,832	17.21

Conclusion: Both Hardware and Software failures have to be considered

What proactive operations could you do for Software errors?

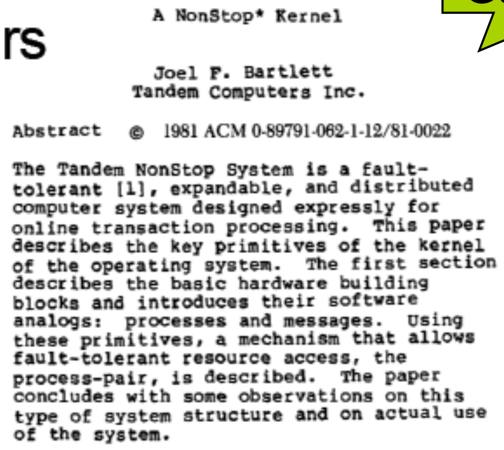
What is the preferred FT approach in other Large Scale Distributed Systems?

	Failures Conside red as:	Cost of nodes	Application Processes coupling	System Designed For FT	Impact of Failures	*Preferred* FT approach
Parallel computers	Excep-tions	Very High	Very Tight	No	Application Stop	Rollback-Recovery
Clouds						
Grids (like EGEE)						
Desktop Grids						
P2P for media files						
Sensor Networks						

Does Replication make sens in parallel computers?

Slide from Garth Gibson

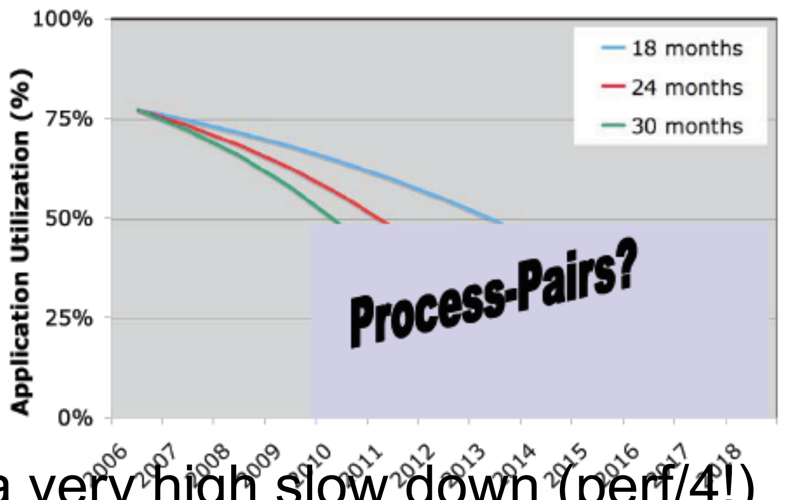
- Classic reliable computing: process-pairs
  - Distributed, parallel simulation as transaction (message) processing
  - Automation possible w/ hypervisors
- Deliver all incoming messages to both
- Match outgoing messages from both
- 50% hardware overhead + slowdown of pair synch
- No stopping to checkpoint
  - Less pressure on storage bandwidth except for visualization checkpoints


A NonStop* Kernel

Joel F. Bartlett
Tandem Computers Inc.

Abstract © 1981 ACM 0-89791-062-1-12/81-0022

The Tandem NonStop System is a fault-tolerant [1], expandable, and distributed computer system designed expressly for online transaction processing. This paper describes the key primitives of the kernel of the operating system. The first section describes the basic hardware building blocks and introduces their software analogs: processes and messages. Using these primitives, a mechanism that allows fault-tolerant resource access, the process-pair, is described. The paper concludes with some observations on this type of system structure and on actual use of the system.



→Some results in P2PMPI suggest a very high slow down (perf/4!)
→Need investigation on the processes slowdown with high speed networks
→Currently too expensive (double the Hardware & power consumption)
•Design new parallel architectures with very cheap and low power nodes

"Algorithmic Based Fault Tolerance"

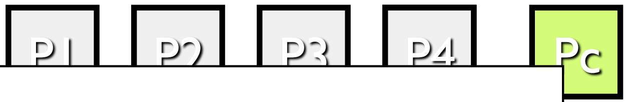
In 1984, Huang and Abraham, proposed the ABFT to detect and correct errors in some matrix operations on systolic arrays.

ABFT encodes data & redesign algo. to operate on encoded data. Failure are detected and corrected off-line (after execution).

From G. Bosilca

ABFT variation for on-line recovery (runtime detects failures + robust to failures):

•Similar to Diskless ckpt., an extra processor is added



But:

1) Does not work for all kind of algorithms
2) Is essentially an "off-line" approach
3) The needed "On-line" version works only for few algorithms

$Xc = X$

$Xf = [$

Xc

Yc

Zc

• Oper

instea

ns:

• Com

checkpointing, the memory AND CPU of Pc take part of the computation):

• No global operation for Checksum!

• No local checkpoint!

= Cf

LU Decomposition: $C = L * U \rightarrow Cf = Lc * Ur$

Addition: $A + B = C \rightarrow Af + Bf = Cf$

Scalar Multiplication: $c * Af = (c * A)f$

Transpose: $AfT = (AT)f$

Cholesky factorization & QR factorization

“Naturally fault tolerant algorithm”

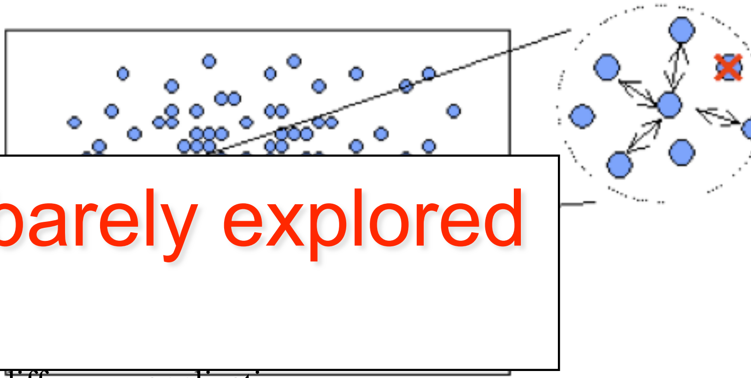
Natural fault tolerance is the ability to tolerate failures through the mathematical properties of the algorithm itself, without requiring notification or recovery.

The algorithm includes natural compensation for the lost information.

Figure from A. Geist

For example, an iterative algorithm may require more iterations to converge, but it still converges despite lost information

Assumes that a maximum of 0.1% of tasks may fail



Ex1 :
(asyn

Very interesting approach but barely explored
Need more research!

finite difference application

Ex2: Global MAX (used in iterative methods to determine convergence)

```
procedure main:
  compute local max
  multicast local max to all neighbors
  do forever
    receive max from any neighbor
    if local max less than neighbor max
      set local max to neighbor max
      multicast local max to all neighbors
    end if
  end do
end
```

This algorithm share some features with SelfStabilization algorithms: detection of termination is very hard!
$\rightarrow$it provides the max « eventually »…
BUT, it does not tolerate Byzantine faults (SelfStabilization does for transient failures + acyclic topology)

Resilience

Definition (workshop Fault Tolerance for Extreme-Scale Computing):

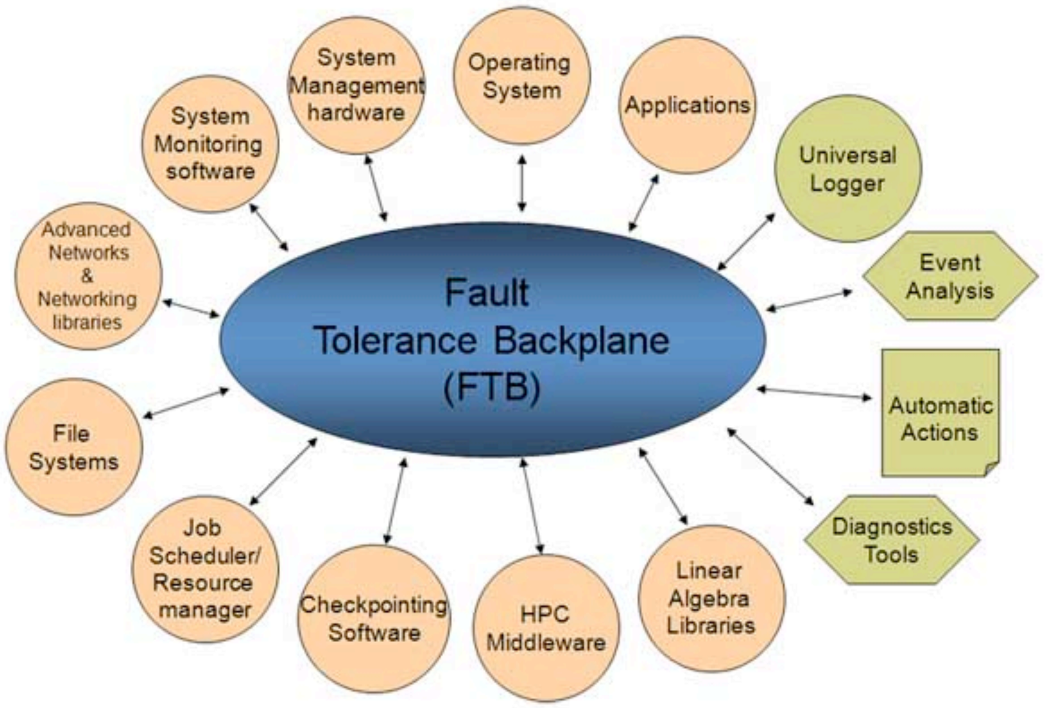
"A more integrated approach in which the system works with applications to keep them running in spite of component failure."

The application and the system (or runtime) need to communicate

-Reactive: FT-MPI (Thomas)

-Proactive actions: reorganize the application and the runtime before faults happen

Problem: to avoid interferences and ensure consistent behavior, resilience need most of the software layers to exchange information and decisions: Fault Tolerant Backplane (CiFTS)
--> huge development effort!



Wrapping-up

Fault tolerance is becoming a major issue

The community “believes” that the current Ckpt-Rst will not work for Exascale machine and probably even before.

Many alternatives but not really convincing yet:

•Reduce the cost of Checkpointing (checkpoint size & time)

•Better understand the usefulness of Proactive actions

•Design less expensive replication approaches (new hardware?)

•Investigate Diskless Checkpointing in combination of SSD device

•Understand the applicability of ABFT and Naturally Fault Tolerant Algorithm (fault oblivious algorithms)

•Resilience needs more research

One of the main problem is that it difficult to get fault details on very recent machines and to anticipate what kind of faults are likely to happen at Exascale.

AND… I did not discuss about “Silent errors”

Questions?