Pragmatic Lessons in Resilient System Design

Background:

Full day tutorial at ISCA 05 – Lisa Spainhower & Subhasish Mitra

Later approached by Morgan Claypool publisher to author ‘book’ for Synthesis Lecture Series

Completed detailed outline, course structure but job demands prevented publication

Proposal:

Community members expressed interest in experiential guidelines

Prepare course to meet student needs

Format TBD – lecture, series, tutorial, etc.

Optimize design tradeoffs to achieve acceptable resilience

While frequent crashes, unpredictable results, and long recovery are unacceptable to almost everyone, there are many acceptable levels between those conditions and complete transparency of failure, repair, and upgrade. Acceptable is dependent upon:

- workload/application characteristics
- availability requirements

Define – useful – common terminology and metrics and their potential inadequacies.

Resilient design and implementation are evolutionary and depend upon tradeoffs with other design considerations, including:

- system organization/architecture
- performance considerations
- semiconductor (and other) technology and its reliability
- system packaging
- cost
- Future part – Moving forward

Point our differing requirements (for instance, HPC vs stock exchange vs website).

Resilience must address all system disturbances

System effects of disturbances

Data and computational integrity exposures: manifestations and implications

Sources of downtime (planned and unplanned)

- Relative contributions, perhaps an overview of trends
- Effects of existing ’taken for granted’ resilience (e.g., rate of memory failures today were there no ECC)

Planned downtime

- Relative contributions and overview of trends
- Upgrade paradox (resistance to put on bug fixes and ease-of-use, availability enhancements)
- Future considerations

Unplanned downtime

- Relative contributions and overview of trends.
- Hardware causes: failures and design bugs

Resilience must address all system disturbances(cont)

- Software failures: design
- Operator or other human failures: some surveys claim these are the predominant cause today. Address how this qualitatively changes the nature of addressing the problem.

Causes and behaviors of failures

- Transient and intermittent failures: Examples of software ‘Heisenbug’ and intermittent vs transient hardware failure.

  When are these failures that should be fundamentally fixed via a design change? Some – like array soft errors – are normal events that one expects and lives with.

- Permanent faults: Usually considered hardware only and the focus of traditional FT techniques, a small part of the overall problem yet their impact can be great (give examples).

  Failures can be relative depending on the observed level. Give CPU sparing example (<<MTTR appears as >>MTTF from a higher level. ECC is another example.)

Problems are alleviated via detection, recovery, and correction

For hardware, there are well-known techniques in both practice and theory – not necessarily the same. Some are better for transient failures and some are better for permanent failures

- Theory techniques – nmr, n-version programming that have never widely caught on and why (mention niche usages)
- Theory techniques that have been adopted and continued to benefit from academic contributions such as parity, ECC, CRC and other coding
- Practice techniques – inline checking, sanity checking, hang detection

Emerging techniques –

Software failure detection: similar evaluation

Common techniques for software and hardware

Emerging considerations (per item #1) that are introducing new failure detection concepts

Optimize design tradeoffs to achieve acceptable resilience