Processor-Level Techniques

♦ Duplicating processors is often prohibitive
♦ Simpler techniques with smaller overheads exist:
  ♦ 1. Re-executing the program (or part of it - requiring checkpointing) - time redundancy
    * Up to 50% performance loss
  ♦ 2. Instruction Retry
    * Requires fault detection circuitry
    * Effective for transient faults
    * Low performance overhead - transparent to program
  ♦ 3. Monitoring by a simpler "watchdog" processor
  ♦ 4. "Checker" processor
  ♦ 5. Redundant Multithreading
Checkpointing

♦ When execution time is very long - probability of failure (soft errors) and cost of failure become significant
♦ A checkpoint is a snapshot of entire process state
  * all information needed to restart the process from that point
♦ Checkpoint saved on stable storage - most commonly used - Disks: can hold data even if power is interrupted
♦ Checkpoints can be very large - 10s to 100s of Mbytes
♦ Uniform checkpointing:


♦ Process Migration - from one processor to another to recover from permanent or transient faults

Watchdog Processor

♦ Performs concurrent system-level error detection
  * Monitoring bus connecting main processor and memory
  * Targets control-flow checking: correct program blocks executed in the right order
  * Can detect hardware/software faults causing erroneous instructions to be executed or wrong execution paths
  * Watchdog needs the program's control-flow information

![Watchdog Processor Diagram]
Watchdog – Control-Flow Information

♦ Derived from Control-Flow Graph (CFG) of program
♦ Assign signatures to CFG nodes & store in watchdog
♦ Run-time signatures computed & compared to stored
  * Signatures assigned (e.g., successive integers) or calculated
  * Assigned can detect invalid execution paths (e.g., \{V1,V4\})
  * Can not detect errors in instructions

Assigned signatures:

```
accept sig(V1);
either
  accept sig(V2);
either
    accept sig(V3);
or
    accept sig(V4);
accept sig(V5);
or
accept sig(V5);
```

Calculated Signatures

♦ Calculate signature from instructions in block, e.g., add them (mod 2) or checksum code
♦ Calculating run-time signatures & comparing to stored ones can detect erroneous instructions
♦ Most data errors will not be detected
♦ Improvement: include "assertions" (acceptance tests) to be checked by watchdog
  * Need forwarding of variables to watchdog and ability to verify assertions
  * Alternative: use parity to check data
  * Current superscalers are difficult to monitor - accessibility and speculative fetching

```
accept & check sig(V1);
either
accept & check sig(V2);
either
  accept & check sig(V3);
or
  accept & check sig(V4);
accept & check sig(V5);
or
accept & check sig(V5);
```
**DIVA checker**

*Dynamic Implementation Verification Architectures*

- **CT** - Commit stage
- **WT** - Watchdog Timer: timeout for control faults (restart)
- All registers and memories protected using ECC
- A checker for each core - 5% area overhead

**DIVA Benefits**

- Simpler design verification prior to production
- Replaces erroneous result generated by the core
  - Checker designed with larger transistors and timing margins
  - Checker has operands and result - no dependencies
  - Main benefit: No need to re-execute or rollback
- Allows more aggressive design rules
  - Smaller transistors
  - Tighter timing and voltage margins (soft errors corrected)
Sentry Core

- Use idling cores (average core utilization is ~30%)
- Incorporate a small core called Sentry Core (SC)
- SC will verify operation of other cores - not just one
- A single idle core is sufficient

Sentry Core - Principles of operation

- When a core idles SC initiates testing
- SC replicates a functional thread on one core on the idle core
- Cores generate two signatures
  * Program counter values of branch instructions to track control flow
  * Data value and data address of store instructions
- SC checks signatures after n committed branches
- Difference indicates presence of fault

1. Detect idle core
2. Suspend target core
3. Initialize MBSR on cores
4. Duplicate thread on idle core
5. Resume thread execution on cores
6. Halt cores periodically and collect and compare signature
Sentry Core (SC) Capabilities

- To enable online test the SC has added functionality
  - Detect idle cores
  - Duplicate threads
  - Collect signatures (using a MISR - Multiple Input Shift Register)
  - Halt cores
  - Resume cores

- Hardware overhead
  - Overhead estimated to be 2.75%
  - More cores served, less is the overhead
  - Limit to how many cores may be served

Fault coverage

<table>
<thead>
<tr>
<th></th>
<th>Control</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>80</td>
<td>85</td>
</tr>
<tr>
<td>80</td>
<td>85</td>
<td>90</td>
</tr>
<tr>
<td>85</td>
<td>90</td>
<td>95</td>
</tr>
<tr>
<td>90</td>
<td>95</td>
<td>100</td>
</tr>
</tbody>
</table>

Average control = 82%
Average data = 92%

- Workloads chosen are functional threads and not test programs
- Faults injected into ROB, INT/FP ISQ, LSQ, Register file, execution units etc.
- Missed faults due to faulty hardware not used and signature aliasing
**Redundant Multithreading**

- Use Simultaneous Multi-Threading (SMT) for fault detection
- Create two independent & identical threads for each application thread
  - Comparing their results will detect transient faults
  - To reduce (performance) penalty 2nd thread trails the 1st
  - Pass information from leading thread to trailing, e.g., results of conditional branches

- Sphere of replication:
  - Units not included in sphere can use error-detection codes, e.g., cache units

**Comparing to checker nodes**

- SRT (simultaneous redundant threads) is unable to detect permanent faults
- SRT requires less hardware changes
- SRT can be used as SMT if reliability is less important than performance
CRT - Chip-level Redundancy Threading

- CRT still allows the trailing thread to avoid cache misses and branch miss-predictions
- CRT detects permanent faults

Byzantine Failures

- Devices can exhibit malicious faults and produce arbitrary outputs - Byzantine faults
  - No problem in M-of-N systems - voter masks out erroneous outputs
  - Distributed system without a centralizing entity may have problems
- Example: A sensor providing temperature to two processors, using point-to-point links
  - Sensor has suffered a Byzantine failure and tells processor P1 that the temperature is 25 degrees and P2 that it is 45
  - P1 and P2 can exchange messages and know that something is wrong, but not who is faulty - P1, P2, or the sensor
**Byzantine Generals (or Interactive Consistency) Problem – General Model**

- A single entity (such as a sensor or processor) sends the value of some variable to a set of receivers
- The receivers can communicate among themselves to exchange information about the value they received
- A functional unit is truthful in all its messages
- A faulty unit may send out contradictory messages
- Time-out mechanism detects absence of a message
- We are looking for an algorithm which will satisfy
  - **IC1**: All non-faulty units must arrive at an agreement of the value that was transmitted by the original source
  - **IC2**: If the original source is non-faulty, the value they agree on must be the value that was sent out by the original source

**Byzantine Generals (or Interactive Consistency) Algorithm**

- Original (and simplest) algorithm is recursive
- \(N\) units (one source and \(N-1\) receivers): Up to \(m\) faulty
- Interactive consistency obtained only if \(N \geq 3m+1\)
- The algorithm \(Byz(N,m)\) has three steps:
  - **Step 1**: *Original source sends information to each of \(N-1\) receivers*
  - **Step 2**: *If \(m > 0\), each of the \(N-1\) receivers runs \(Byz(N-1,m-1)\) to send the value it received to the other \(N-2\) receivers*
    * If a unit does not get a message from another unit (after some time-out), it enters the default value into its records
    * If \(m = 0\) this step is bypassed
  - **Step 3**: *Each receiver now has a vector of \(m\) values and uses the majority, or the default if no majority exists*
    * If \(m = 0\), it uses value received from original source*
Interactive Consistency Algorithm - Notation

- **A** and **B** are units - **A.B(n)** means **A** sent **B** the message **n**
- **U** is a string of units **A_1, A_2,..., A_m** - **U.B(n)** means **B** received **n** from **A_m** who claims to have received it from **A_{m-1}** and so on
- **ϕ** - A message that is not sent, e.g., **A.B(ϕ)** means the message that **A** was supposed to send **B** was never sent
- **A.B.C(n)** - **B** told **C** that the value it received from **A** was **n**
- **A.B.C.D(n)** - **D** received **n** from **C** who claims to have received it from **B** who, in turn, claims to have received it from **A**
- **Black.White.Green(341)** means **Green** received the message **341** from **White** who claims to have received it from **Black**

- **Example: m=0** - Degenerate case: no fault tolerance
  * Step 2 bypassed - Interactive Consistency Vector (ICV) consists of a single value that has been received from original source

- **Example: m=1**
  - Must have at least **3m+1=4** units: sensor **S** + 3 receivers, **R_1, R_2, R_3**
  - Sensor is faulty
    * sends out inconsistent messages to receivers: **S.R_1(1), S.R_2(1), S.R_3(0)**.
    * All receivers are functional
    * default is **1**
  - **Step 2**: **R_1, R_2, R_3** each acts as source and runs **Byz(3,0)**
    * following messages are sent:
      * **S.R_1.R_2(1)** **S.R_1.R_3(1)**
      * **S.R_2.R_1(1)** **S.R_2.R_3(1)**
      * **S.R_3.R_1(0)** **S.R_3.R_2(0)**
  - Element **j** of **ICV** at receiver **R_i** is equal to
    * Report of **R_j** as determined by **R_i** if **i≠j**
    * Value received from original source if **i=j**
  - At the end of this step, **ICVs** are (**1,1,0**) at every receiver
  - Majority vote yields **1** - the value used by each of them
Example: N=7, m=2 (R1 and R6 faulty)

- Messages sent in first round by S are consistent
  - S.R1(1), S.R2(1), S.R3(1), S.R4(1), S.R5(1), S.R6(1)
- Each receiver executes Byz(6,1) in step 2 of Byz(7,2)
- R1 is faulty and can send out any message (or nothing) - it sends in step 1 of Byz(6,1):
- In step 2 of Byz(6,1) each of R2-R6 uses Byz(5,0) to disseminate the message received from R1 - the messages are:
- R6 being maliciously faulty is free to send out anything it likes

Example – Cont’d (R1 and R6 faulty)

- ICVs at each receiver in connection with the S.R1(1) are:
  - ICV_{S.R1}(R2) = (1,2,3,4,1)
  - ICV_{S.R1}(R3) = (1,2,3,4,8)
  - ICV_{S.R1}(R4) = (1,2,3,4,0)
  - ICV_{S.R1}(R5) = (1,2,3,4,0)
  - ICV_{S.R1}(R6) is irrelevant, since R6 is faulty
- R5 received nothing from R6, its value is recorded as the default, say 0
- R2, R3, R4, R5 find no majority in ICV and assume the default value (0) for S.R1
- Similarly, agreement can be reached on the message that S sent to each of the other receivers
- This completes the generation of the ICVs connected with the original Byz(7,2) algorithm
Message Authentication

♦ The Byzantine Generals problem is hard because faulty processors could lie about messages received
♦ Simplify by providing a way to authenticate messages
  * Each processor can append to its messages (included forwarded ones) an unforgeable signature
  * The recipient can check the authenticity of signatures
♦ A message forwarded through A and B, can be checked to see whether the appended signatures of A and B are valid
♦ All processors have a time out mechanism against faulty processors that remain silent

Algorithm with Message Authentication

♦ Algorithm \textbf{AByz(N,m)} (for maintaining interactive consistency):
♦ Step A1: Original source signs its message \(\psi\) and sends out to each processor
♦ Step A2: Each proc. \(i\) that receives a signed message \(\psi:A\) (\(A\) - set of signatures appended to \(\psi\)), checks \# of signatures in \(A\)
♦ If \(\#<m+1\) - sends out \(\psi:A\cup\{i\}\) (adds its own signature) to each processor not in \(A\) & adds \(\psi\) to list of received messages
♦ Step A3: After seeing signatures of all processors (or time-out), apply some decision to select from messages received
Outline of Proof for $N>m+2$

♦ **Case 1**: Original source functional - a signed message $\mu$ transmitted to every processor

♦ Nobody can forge the source's signature, no processor will accept message other than $\mu$ in Step A2

♦ **Case 2**: Original source faulty - different messages may be sent out to different processors, each with a correct signature

♦ List of received messages (minus signatures) is the same at each processor

♦ Proof by contradiction