Application Specific Fault Tolerance Techniques

Jacob A. Abraham
The University of Texas at Austin
Computer Engineering Research Center
Austin, Texas 78712
jaa@cerc.utexas.edu
www.cerc.utexas.edu/~jaa

ITS Workshop
October 5, 1999

Fault Tolerance Techniques

- Steps in tolerating a fault
  - Detection of an error (due to a fault) at some module output
  - Correction of the error
  - Identification of the faulty module
  - Reconfiguration of the system to bypass the faulty module
- Conventional redundancy techniques include triplication and voting (masks errors) and duplication or self-checking circuitry (detects errors) with check-pointing (for recovery)
- If redundant processors are loosely synchronized, it may be(possible to detect errors due to software or design faults
- Application-specific techniques proposed to reduce the cost of fault tolerance
  - data integrity checking
  - control flow checking
  - executable assertions

Algorithm-Based Fault Tolerance

- Novel system method for achieving fault tolerance
  - encoding data results in low overhead
  - fault tolerance scheme is tailored to the algorithm to be executed
  - initially proposed for highly parallel arrays for signal processing
- Characteristics of algorithm-based fault tolerance
  - encoding of the data used by the algorithm at a high level
  - redesign of the algorithm to operate on the encoded data
  - distribution of the computation steps in the algorithm among computation units
- Technique and extensions have been shown to detect a wide variety of errors

Checksum Encoding for Matrices

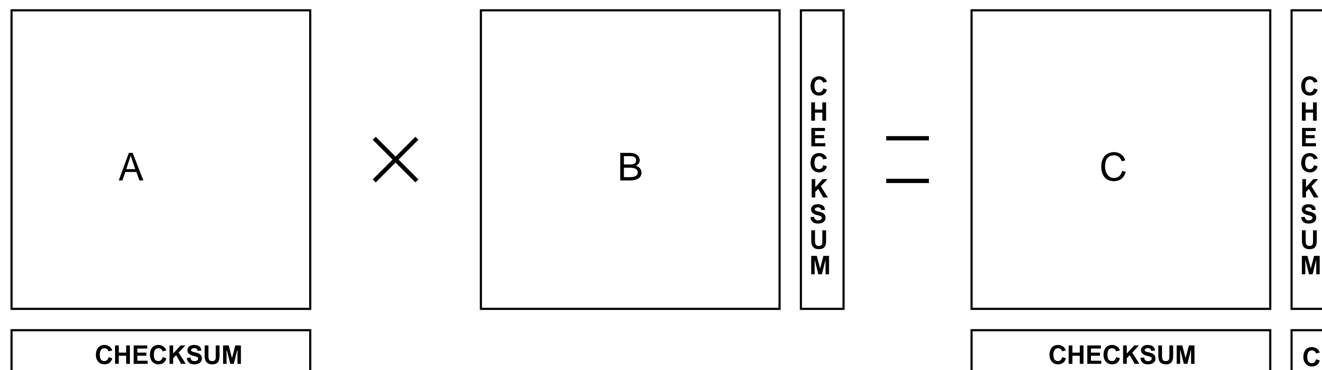
- Let A be an $n \times m$ matrix
  - Define column, row and full checksums,

$$A_c = \begin{bmatrix} A \\ e^T A \end{bmatrix},$$

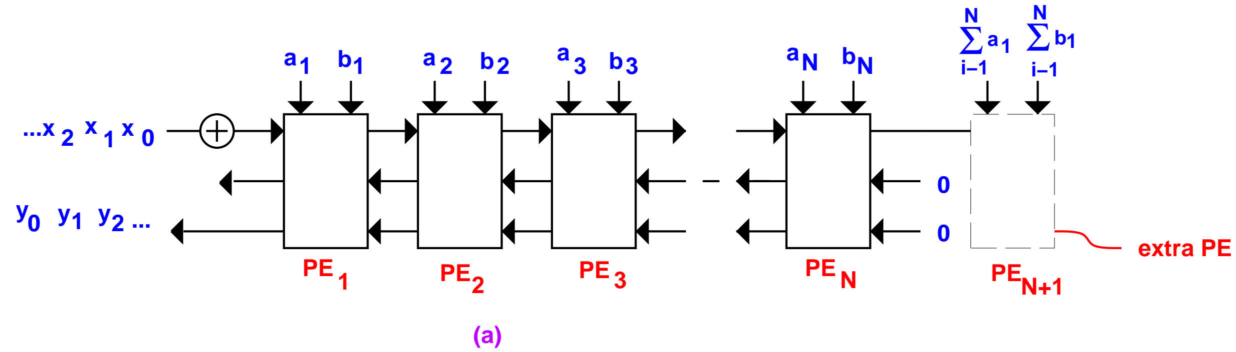
$$A_r = \begin{bmatrix} A & Ae \end{bmatrix},$$

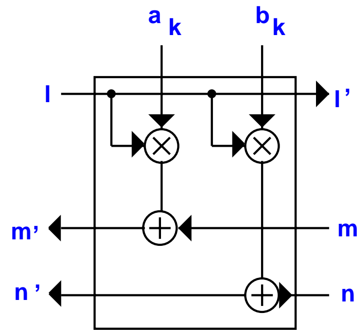
$$A_f = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}, \text{ where } e^T = [111 \ldots 1]$$

- Checksum property preserved by multiplication, LU decomposition, etc.



Error Detection in Linear Systolic Array



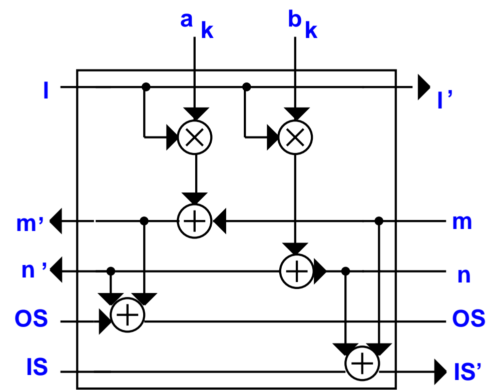


$l' = l$

$m' = m + a_k l$

$n' = n + b_k l$

(b)



$IS' = IS + m + n$

$OS' = OS + m' + n'$

$l' = l$

$m' = m + a_k l$

$n' = n + b_k l$

(c)

(a) A systolic array for the IIR filter

(b) PE used in the original (non–CED) design

(c) PE used in the CED design

Control Flow Checking

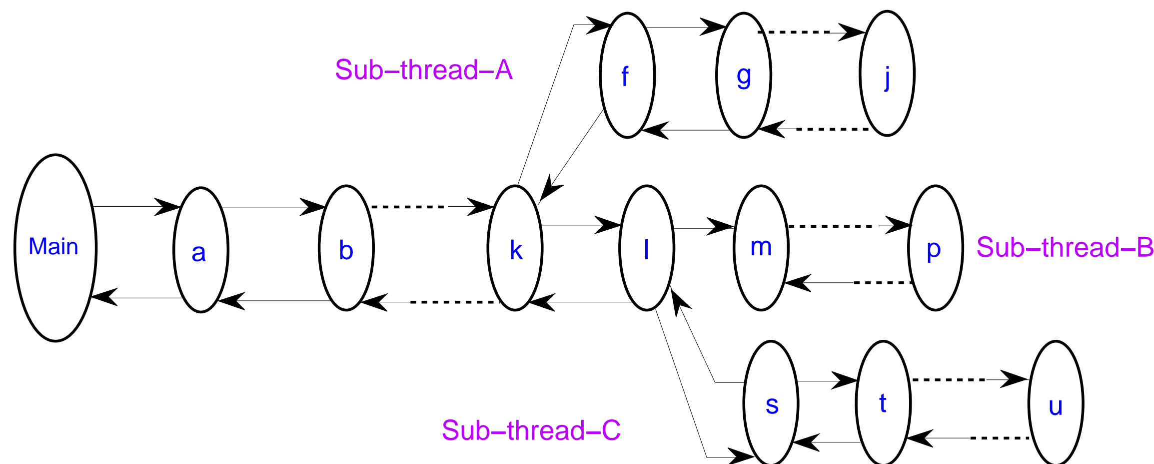
- Detect errors due to failure in the underlying hardware by monitoring the execution sequence of a program and comparing it to allowable sequences defined by the system model
- Program can be divided into loop-free intervals and code inserted for performing concurrent checking
- Embedded signature monitoring – hardware supported monitoring
  - an application program is partitioned, at assembly time, into a sequence of instructions (called segments) with one entry point and one exit point
  - signatures are generated (off-line) as a function of the sequence of the instructions within a segment
  - a monitor (watchdog processor) generates signatures of the code executed by the processor during normal operation
  - the monitor periodically compares the signatures to pre-computed reference signatures
- Checking techniques can be implemented in software
  - need to address problems with overhead and coverage

Control-Flow Checking Using Assertions (CCA)

- Successor to hardware-specific control-flow checking techniques
- Branch-free intervals (BFIs) of high-level language programs identified
  - branch-free interval ID (BID) is unique for each BFI
  - control-flow identifier (CFID) represents permissible control flow and is the same for all BFIs sharing a parent BFI
- Entry and exit points of BFIs fortified through pre-inserted assertions
  - BID set on entry to a BFI and checked on exit
  - on entry to a BFI, the CFID of the next BFI is placed on a queue; upon exit, the CFID is dequeued and verified
- Assertions depend only on syntactic structure of language
- Portable across architectures

Concurrent Object Thread Monitoring - COTM

- High-level checking scheme for object-based systems – execution of the application is viewed as a thread invoking methods contained in the objects traversed



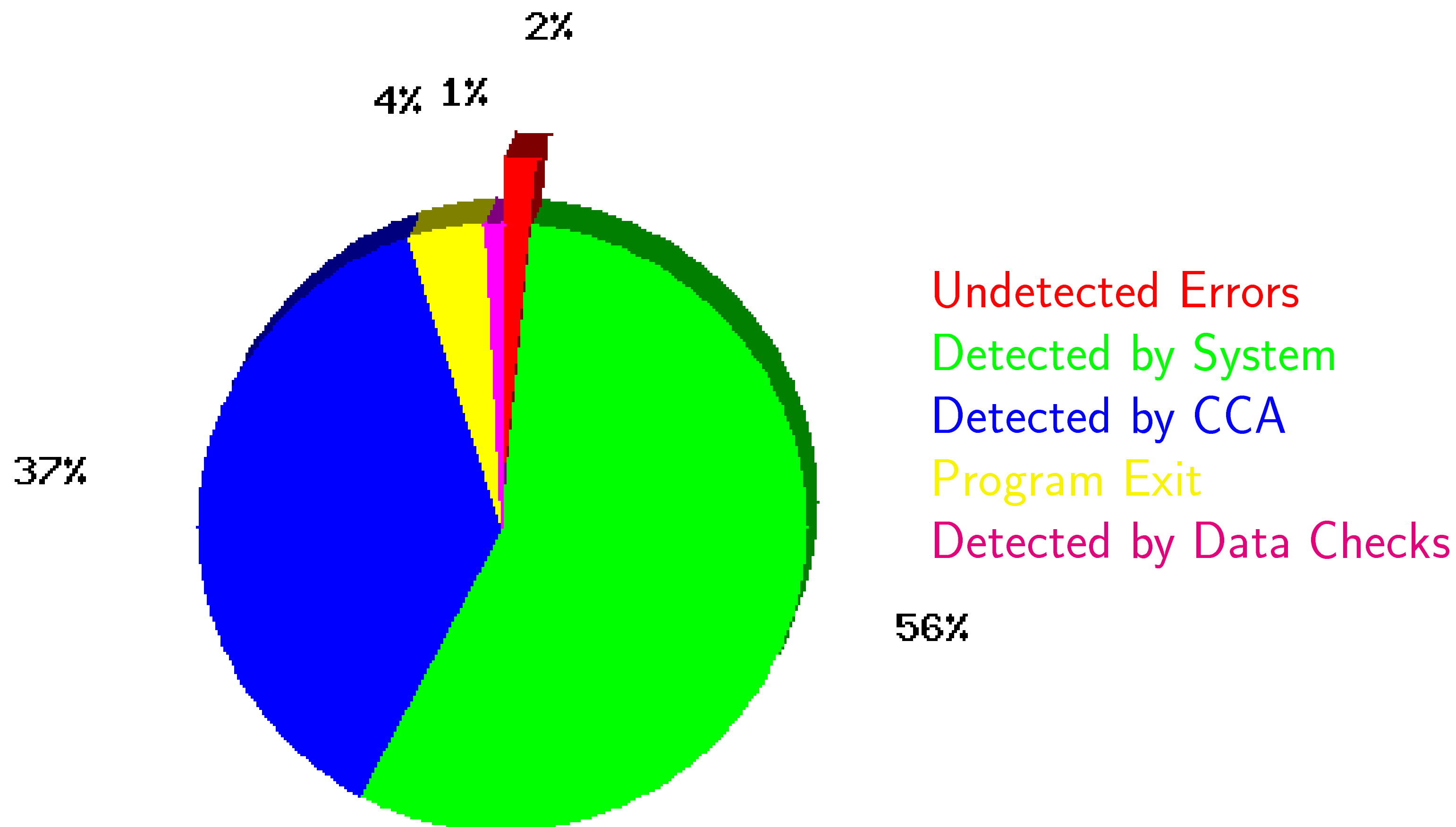
- Use the object-thread model to obtain control flow information and exploit features embedded in the object oriented paradigm for error detection
  - insert segment ids and checking code for control flow checking inside methods of the classes (segment validation)
  - modify the application program to record all segment ids it has traversed during the course of its execution
  - monitor the recorded segments to check control flow

Evaluation of Techniques Using Fault/Error Injection

- Evaluate the behavior and performance of complex systems under faults while executing real applications
  - determine effectiveness (coverage) and performance impacts of error detection mechanisms
  - evaluate overhead of recovery algorithms under permanent and transient errors
  - study effects of faults occurring during the recovery process
- Usefulness has been documented with many techniques of fault and error injection utilizing hardware, software and heavy ion techniques
- FERRARI (Fault and ERRor Automatic Real-time Injector) – a flexible tool for injecting faults which emulates hardware faults using software
  - faults injected in real time while a task is executing
  - ability to inject transient as well as permanent faults
  - control the type of fault to be injected, its location and duration (for example, can synchronize error injection with instructions being executed)

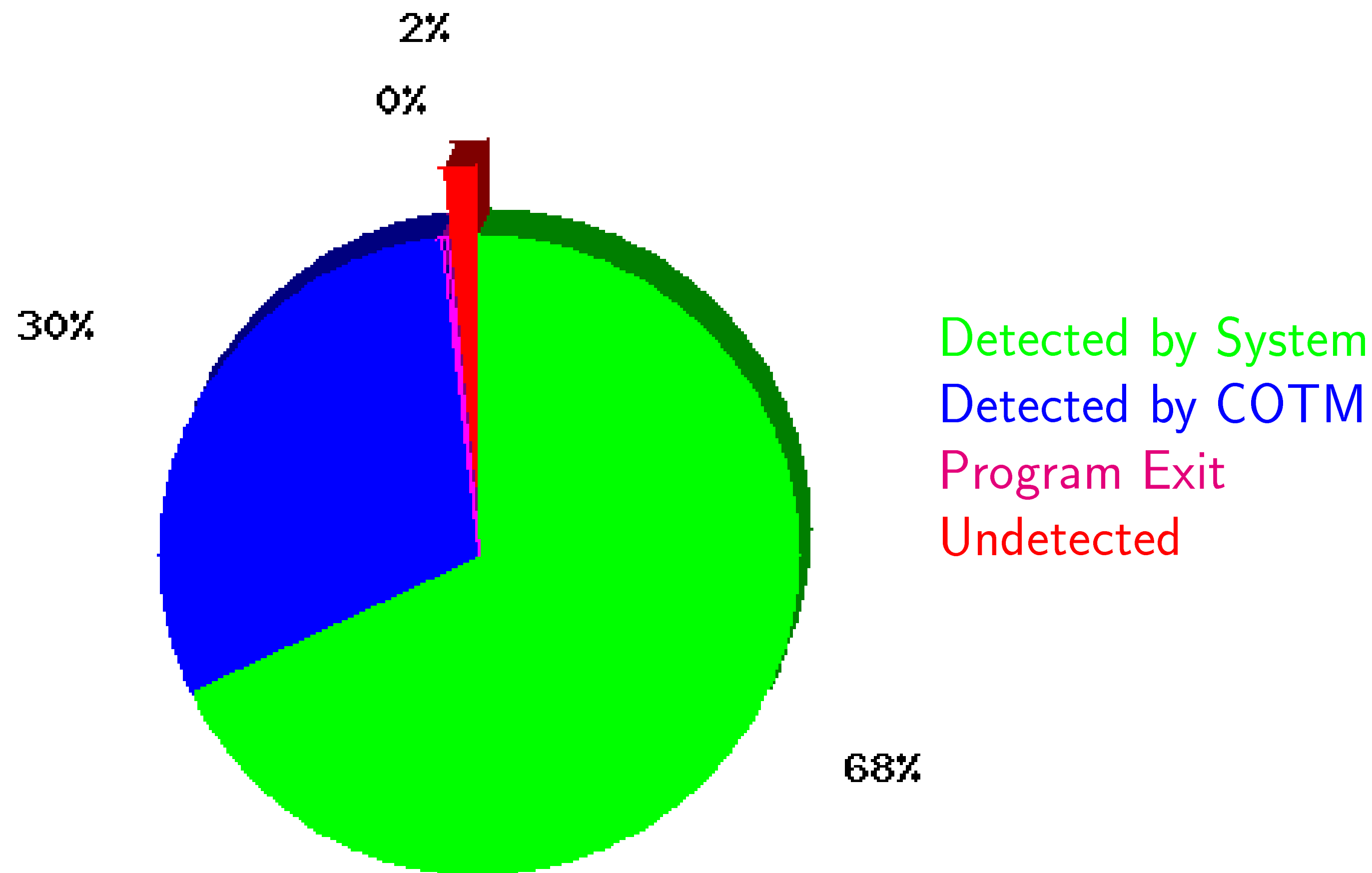
Error Coverage of CCA with Data Checks

- Experiments included data as well as control errors



Error Coverage Using COTM

- Gtroff application program
- Errors in Condition Code Registers



Enhanced Control-flow Checking using Assertions – ECCA

- Preprocessor adds assertions to code to detect control-flow errors
  - divide-by-zero exception signifies a control flow error
- Steps in high-level ECCA
  1. Divide program into blocks, sets of consecutive branch-free intervals
  2. Assign to each block a unique prime (> 2), the Block IDentifier, BID
  3. Place two assertions per block

  SET assertion at the entry point of each block

$$id \leftarrow \frac{BID}{\overline{(id \mod BID)} \cdot (id \mod 2)}$$

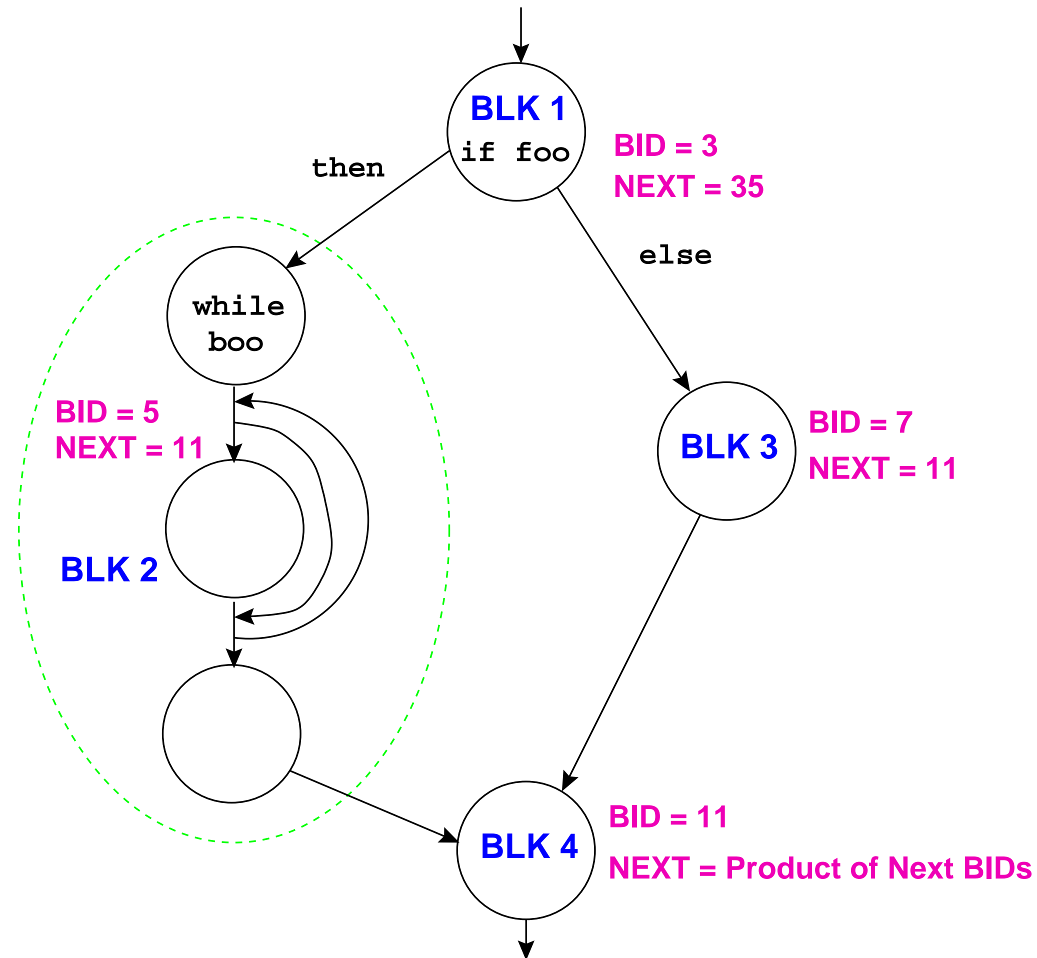
```
id = <BID>/((!(id%<BID>))*(id%2));
```

  TEST assertion at the exit point of each block

$$id \leftarrow NEXT + \overline{\overline{id - BID}}, NEXT = \sum_i BID_i$$

```
id = <NEXT>+!!(id - BID);
```

Example of ECCA Grouping Into Blocks



Intermediate Level ECCA

- Improve efficiency and coverage of ECCA by implementing it at a lower level
- Place additions to the compiler as low as possible without becoming machine dependent
  - add the assertions the intermediate Register Transfer Language used by GNU compilers
- Since branches are reduced to simple conditionals and jumps, the maximum number of locations a branch could target is two
  - lower overhead for assertions

SET assertion: $r_1 \leftarrow (r_1 - BID) * (r_2 - BID), \qquad r_1 \leftarrow \frac{BID+1}{\left(\frac{r_1+1}{r_1*2+1}\right)}$

TEST assertion: $r_2 \leftarrow (r_1 - BID) * NEXT_2, \quad r_1 \leftarrow (r_1 - BID) * NEXT_1$

- Performance overhead ranges from 5% to 55%
  - can reduce overhead significantly by reducing the block size with very little loss of coverage

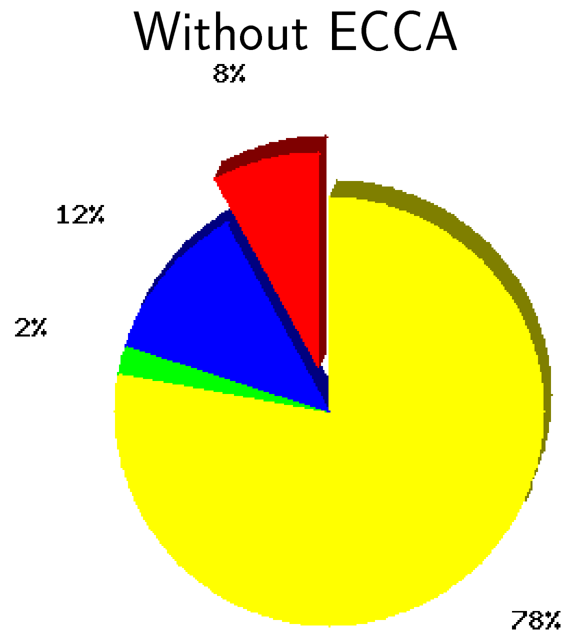
Experimental Evaluation of ECCA Using FERRARI

- Results on *espresso*, one of the SPEC benchmarks
  - operates on Boolean functions
- Effects of both control and data errors studied
  - errors in condition code flags
  - data line error when an operand is loaded
- First run of benchmark without any errors injected, output stored in a reference file
- Multiple error injection runs, with triggering error detection mechanism being recorded
  - if no error detection mechanism is triggered, output compared with the reference, with difference indicating an error escape
- Injection runs continued until results do not vary by more than 0.5% even after increasing the number of runs in the experiment
  - system behavior converged between 10,000 and 20,000 runs

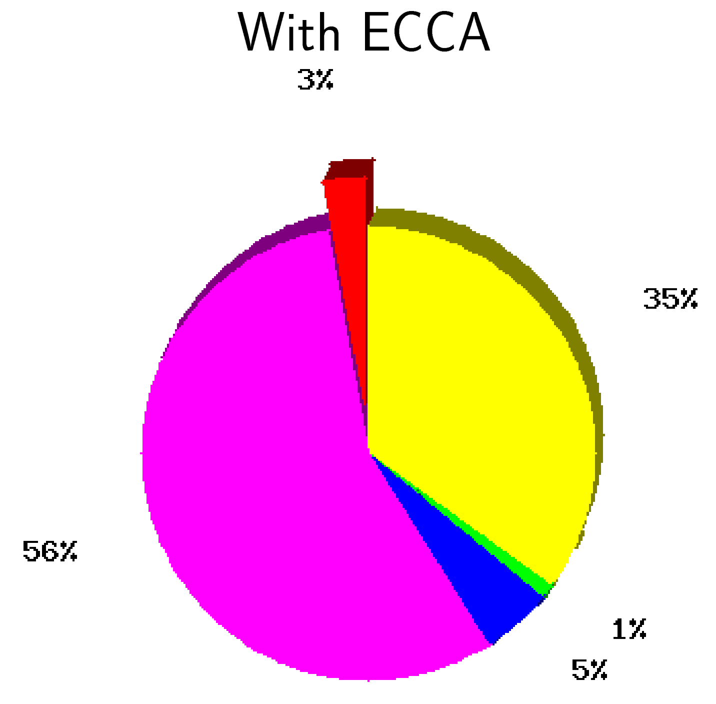
Error Detection Using ECCA

- Data line error when an operand is loaded

Without ECCA



With ECCA



Detected by System
Timeouts
Program Exits
Undetected

Detected by System
Timeouts
Program Exits
Detected by ECCA
Undetected

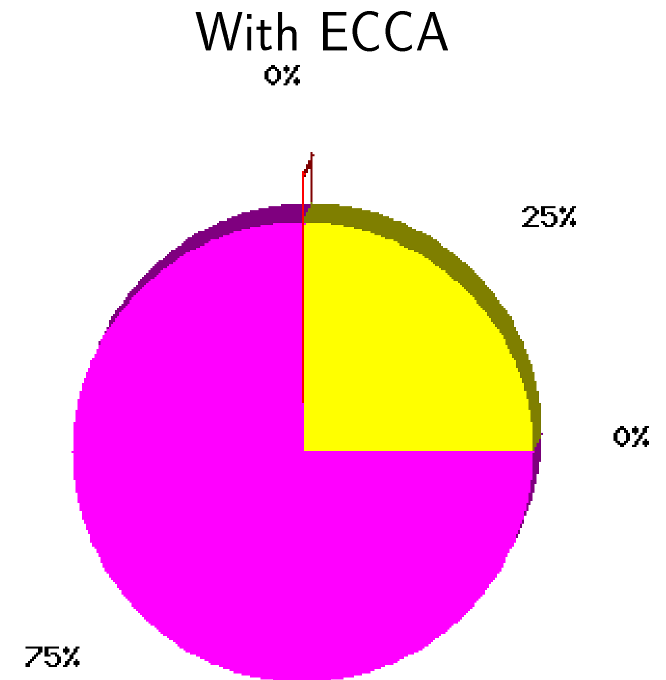
Error Detection Using ECCA

- Errors in condition code flags

Without ECCA

Detected by System
Timeouts
Program Exits
Undetected

With ECCA



Detected by System
Timeouts
Program Exits
Detected by ECCA
Undetected

Dealing with Design Faults

- In order to deal with design faults (which can lead to intrusions), systems can be analyzed carefully by a variety of techniques, including formal methods
- Fault Intolerance
- Formal analysis is limited by the enormous number of possible states
- Abstractions can be used to reduce the state space which needs to be analyzed
  - extracting the control behavior of the system can lead to a simpler finite-state machine
  - results in a conservative approximation
- Safety and liveness properties can be specified as non-deterministic machines
- Model can be checked against the property to find problems

Combining Fault Intolerance and Fault Tolerance

- Improve the level of intrusion tolerance by
  - improving analysis techniques for complex systems
  - adding fault tolerance as an additional layer of protection
- In real systems, even the control approximations may result in enormous state spaces
  - conservative abstractions can lead to false negatives
- Ideally, the analysis should be done on the program code, and not on a model of the system
  - abstractions may mask flaws in the system
  - results are valid only on the model, not on the real system
- Need to understand nature of application-specific checks which will best deal with design faults
- Develop analysis techniques which deal with the real system, and supplement deficiencies of the analysis with fault tolerance mechanisms, such as application-specific checks and control flow checks