System software support of hardware efficiency

by Igor Schagaev
Plan for today - I

Theories in brief:
- FT: GAFT, Processes
- Redundancy
- Recoverability
- Reconfigurability

System Structure Change
- GAFT, Supportive processes
- Redundancy handling
- Reconfigurability support
- Static&Dynamic Control of RP

Properties...
- Reliability...
- Fault tolerance...
- Performance...
- Maintainability...
- Adaptability...
- Scalability...
- Life circle costs (manufacturing, run-time, utilization)
- Energy efficiency
- Ease of use (learning, application, maintenance)
Plan for today - I I

System Software for FT: Language
  Data structures
  Control operators
  Semaphores
  Recovery point formation

System Software for FT: Run-time system
  Health monitoring - tests and checks
  Recovery point support
  Recovery point handling (organization, HW use)
  Recovery point search
  GAFT support: reconfiguration control - a syndrome

System Software & Hardware for future: PRESSA
R&D principles for computer systems

Redundancy Theory

Reliability Theory
Control Theory
Computer Science
Economics
Management

Hardware
Active Zone:
- Arithmetic Unit
- Logical Unit
Interface zone
- Bus, Configurator
  - Internal
  - External
Passive zone
- System memory
- User memory

System Software
Semantic
- GLL
Structure
- Language
- Concurrency
Runtime
- HW state handling
- SW state handling
- Recovery points

Simplicity
Redundancy
Reconfigurability
Scalability
Reliability
Reliability vs. performance in computer systems

- Reliability of the system is limited
- Performance of the System is limited
- System Complexity is Constant
- Number of elements in the System is $n$
Model of fault tolerance: introduction of GAFT

The system model, fault model and fault tolerance model are mutually dependent as it is shown at the bottom of Figure 3.5. Note that in the here presented approach, the development and manufacturing cost of a solution is not considered.

Fault is a description of all possible faults a system must tolerate. In binary logic a typical permanent fault manifests as "stuck at zero" or "stuck at one". Behavioral faults such as Byzantine faults (malfunctions) and hidden faults (so-called latent faults) that exist in the hardware over a long period of time do not ease the life of a system engineer of fault tolerant systems: all described faults should be tolerated within a limited and specified period of time. This period actually determines the availability of the system. Fault types differ by their impact, as well as the way they are handled. Thus, the fault model has its own hierarchy, including single-bit, element, behavioral and subsystem faults. One has to accept that the fault type is varying and some action hierarchy to tolerate them is also required. All faults types should be tolerated, as there are no such systems called half- or semi-fault-tolerant.

The so-called fault encapsulation approach to fault handling can help: due to de-liberate design solutions it is possible to ensure that severe faults in the system will manifest themselves as simpler to handle faults from the system's point of view; therefore making the fault handling practically possible to implement. This approach will be further developed and applied here.

RT FT system applications assume long operational life; however, fault-handling schemes are needed much more often towards the end of the device life cycle.

Generalized Algorithm of Fault Tolerance (GAFT)

- Detecting faults
- Identifying faults
- Identifying faulty component
- Hardware reconfiguration to achieve a fault-free state
- Recovery from correct state(s) for both: the system and user software

FT property will be achieved if...
Model of fault tolerance GAFT and HW & SSW

Hardware and System Software for Fault Tolerance

GAFT in HW and SSW

IF error is detected THEN

Determine the fault type;
IF the fault is permanent THEN
Locate the faulty component;
Reconfigure the HW by excluding faulty unit;
END;
ELSE
Locate faulty program states and find the correct ones to continue;
Recover the system from preliminary stored correct SW states;
END;
END;

New property achieved
GAFT implementations using redundancy

### Step Description

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Name</th>
<th>Redundancy type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PERIODICALLY DO</td>
<td></td>
<td>Create recovery point END</td>
</tr>
<tr>
<td>A</td>
<td>IF error is detected THEN</td>
<td>HW(I), HW(S), HW(T)</td>
<td>The processor itself has measures to detect faults during execution and can abort and restart the currently running instruction</td>
</tr>
<tr>
<td>B</td>
<td>Determine the fault type</td>
<td>SW(I), SW(S), SW(T)</td>
<td>The memory chips are triplicated and a voter compares the output of the three memory chips. If a deviation is detected, the majority voting is used to identify the faulty chip and the faulty value is rewritten. Read after write ensures the proper storing of the data.</td>
</tr>
<tr>
<td>C</td>
<td>IF fault is permanent THEN</td>
<td></td>
<td>The same program is run twice with the same input data set. The output of both programs is then compared. Prior to running the program, the input data is validated to conform to a certain pattern and range.</td>
</tr>
<tr>
<td>D</td>
<td>Locate Faulty Element</td>
<td></td>
<td>The data stored to the external storage device is protected by a CRC-32. This allows the identification of incorrect data but no recovery.</td>
</tr>
<tr>
<td>E</td>
<td>Reconfigure Hardware</td>
<td></td>
<td>Storage devices such as flash cards or hard disks are duplicated. Note that this feature does only provide fault detection but not recovery.</td>
</tr>
<tr>
<td>F</td>
<td>IF hardware has been reconfigured OR software is affected</td>
<td></td>
<td>As an ultimate resort, a watchdog is used to restart parts of, or the whole system. Hardware based watchdogs can typically on restart the whole system at once.</td>
</tr>
<tr>
<td>G</td>
<td>Locate faulty software states</td>
<td></td>
<td>This way we think..</td>
</tr>
<tr>
<td>H</td>
<td>Recover software</td>
<td></td>
<td>This way we do..</td>
</tr>
<tr>
<td>I</td>
<td>IF hardware has been reconfigured THEN</td>
<td></td>
<td>This way we think..</td>
</tr>
<tr>
<td>J</td>
<td>Reconfigure software</td>
<td></td>
<td>This way we do..</td>
</tr>
<tr>
<td>K</td>
<td>CONTINUE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GAFT might be implemented (and fault tolerance achieved) using redundancies of:

- **Information (I)**
- **Time (T)**
- **Structure (S)**

Implemented in and by:

- **Hardware (HW)** and/or
- **Software (SW)**
GAFT vs. ontologies... personal comment

Now in BOLD: Classification of system redundancy in terms of Information (I), Time (T), Structure (S) can be used to implement Fault Tolerance (GAFT).

Fault tolerance, as a process, has to be implemented through hardware and system software combination. Both: concept and implementation form a theory that allows to analyze, control and predict behavior of the system with new properties.

An example of GAFT extension: “Method and apparatus of system safety” (patent http://it-acs.co.uk/files/GB2448351B.pdf)

In contrast to popular wave of various kind of ontologies that execute descriptive function of knowledge - GAFT and rigorous classification of redundancy analyze & predict system behavior.
### GAFT & redundancy theory vs. ontologies

<table>
<thead>
<tr>
<th></th>
<th>Definitive function (DF)</th>
<th>Characteristic Function (CF)</th>
<th>Predictive function (PF)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GAFT</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Ontologies</strong></td>
<td>+</td>
<td>?+</td>
<td></td>
</tr>
</tbody>
</table>

*Power of any theory is in predictions and our ability to use them*
GAFT impact on performance and reliability

Performance & Reliability of our systems should be within required zone for the whole operation cycle...

Reliability is achievable with system software support
4.4 GAFT implementation: performance, reliability, coverage

As already mentioned above and shown in Figure 4.1 and Table 4.1, the three connected processes checking and testing, preparation for recovery and recovery might be implemented by and within SSW and HW at the design phase and run time phase of the whole system life cycle. Obviously, different implementations of the three processes differ in terms of fault coverage, achieved reliability, availability and cost. Different GAFT algorithm implementations vary in terms of used redundancy types and therefore also the time to complete GAFT.

The run-time phase of the system life cycle in terms of time redundancy fault tolerance can be considered at different levels of granularity which are related to the scope of the program being executed. We differentiate the following 5 levels: instruction, procedure, module, task and system (not shown) as presented in Figure 4.3.

The instruction level scheme assumes that when a fault appears, its influence is eliminated within the instruction execution, using hardware redundancy for fault detection, fault location and fault recovery. Only hardware based redundancy (HW) can detect faults to a powerful storage subsystem that can detect and recover from faults. The duplicated storage is used to detect faults, whereas the SW based CRC-32 is used to identify the correct data which is then used to correct the faulty instance.

**NB:** if probability of recovery ≠ 1 the system is **NOT** fault tolerant !!!

![Diagram showing system recovery times according to the used scheme](image)
Language reflection of FT systems

Real Time

- **HW** (Timers, RISC structure of processor etc)
- **Language** (Limitation of the language constructions that complicate RT capability)
- **OS** (Management of timers and task scheduling with RT constraints)
- **AP** (Application specific schemes of RT)

Fault tolerance

- **HW** (Majority schemes, Hamming codes)
- **Language** (check points, recovery points, at language level)
- **OS** (management of check points, recovery points, synchronization, HW reconfiguration)
- **AP** specific realization of possible hardware deficiency solutions

New Features

Approaches
Next we present some comments on Real Time and Fault Tolerance features and the supportive means by the operating system. Assume that a program requires RT access to program data.

To guarantee the required real time constraints it is important to exclude file structures, as these do in general not allow direct and equal (in time) access to the data. Instead, simpler data structures with guaranteed by design equality to access each data element or record should be introduced.

Clearly, RT as a new feature requires modification of almost all elements in Figure 6.1. RT and FT are synthetic, not elementary features of computer systems, and possible implementations can be located at different layers in system hierarchy.

Some well-known solutions exist for achieving RT in a computing system, such as limiting the use of data constructs, deliberate introduction of time-limit program control structures, exclusion of complex instructions from the processor architecture, limitation of pipelining, strong extension of timer schemes, etc.

The ellipses in the middle of Figure 6.1 represent possible implementations of new features.

As a general rule, the top down principle should be applied, i.e. every new feature should be implemented at the top level of the system hierarchy if possible. This rule does also imply that implementations of new features like S1, which is implemented on the application level, are excluded from this research. Instead, we concentrate on new features implemented at the top layers of the presented hierarchy, assuming that the resulting dynamic system behavior will be under full control.

Let’s take language features as an example. A programming language can be described by means of control structures, presentation of data types and the realization of sequential and conditional expressions. For example, typical data structures are: arrays, strings, files, records (Figure 6.2).

Figure 6.2: Modification of data structure limited by application domain


File structure must be modified, Control operators - upgraded
Level \( i \) is what has been changed in state of hardware.

All: control, data and conditions involved must be preserved (to be able to recover...)
Control using nWhile

So far there is no clear separation of the actions to react on exceptions at the language level for operators of repetitions.

This is because awaiting of an event might be perfectly valid action or ...

endless wasting due to hardware fault that has happened.

New nwhile* loop can be useful

Takaoka* suggested new control operator, actually without thinking about embedded system issues...

This operator was called nwhile, and looks like below:

```
nwhile B do S
```

where B is condition to enter the loop and S is body of the loop.

Introducing precondition P and post condition Q for this structure Takaoka suggested to use several assignment statements S_1, S_2, S_3,... S_N in S which affect the condition B and therefore P_1, P_2, P_3, P_4,... P_N that held immediately before S_1, S_2, S_3,... S_N under precondition P.

Control using $\text{nWhile}$

Then we have an inference rule:

$$\prod_{i=1}^{n} \{P_i \land B\} S_i \{Q_i\}, \prod_{i=1}^{n} (\neg B \land Q_i \Rightarrow Q), P \land \neg B \Rightarrow Q$$

$$\{P\} \text{nwhile} B \text{do} S \{Q\}$$

What it gives us? Actually, a lot.

Writing a program we use loop operators as usual, but during compilation System Configurator should introduce other $S_2$-$S_n$ conditions for loop exit that might be connected with computer state changes including hardware faults, timer run out or other interruptions, including interaction with other processes.

Then in case of hangs of the loop due to problem within hardware and/or arrival of another signal we are able to break loop execution and make it visible...

hardware state change is reflected immediately within program control construction and... we are not using brutal force of waiting or waste of vast amount of another redundancy... still being uncertain...
More control: Fault Tolerant Semaphores

Concurrency and parallelism confusion;
What we start in parallel eventually will end up with concurrency... and (possibly) vice versa...

Resolving confusion - Graph Logic Model (GLM)

XOR and AND will resolve confusion. Known solutions semaphores, monitors are all down to XOR.... But we should be using both operators...

Still... Why Fault Tolerance is required?
...Waiting mad driver to release train handle might be costly... (Jethro Tull, Locomotive breath)

Time redundancy - waiting for few ms for processes duration within dozens of ns? ... It is NOT the solution.
Information and structure redundancies should be used instead...

GLM - is structural redundancy...
Information?

a : $XOR_{(\alpha b, \gamma d)}, \text{AND}_{+}(\beta b, \delta c)$

...old Charlie stole the handle and the train it won't stop going no way to slow down...
Mad driver, also known as “dining philosopher” should be trained to be polite...

We need to teach our dining philosophers to be polite, and ... "die like a man", especially when they are in critical section.

What does it mean? The one, being in critical section, if sick or dies must:

- Return all spaghetti, forks - resources he uses
- Inform the rest by "I am dying" message...

Then “the rest” (Runtime system in our case...) has to:

- Reduce number of voters for further voting in sessions of concurrency resolution
- Mark a messenger as suspected (not excluded, or dead) and place in a special pool
- Schedule a “reincarnation procedure”

(...power of malfunction might be big, duration long (200ms+), but “treatable”...)

This scheme has been called FT semaphores...
FT Semaphores hardware support: T-logic scheme

for one computer

for multiprocessor system

- HW element “suspected” should “switch itself” - (left RAM above);

- System should be able to return it in action after full-size check, if it was recovered.
A testing phase is required initially at boot up time to guarantee the correctness of the hardware; periodic test is required before and after the execution of a program. The applied tests might vary in depth (coverage), type of faults and the set of the tested hardware.
GAFT: System checking by system software - II

Testing at the level of tasks...

Three tasks are running, each with its own test (the green boxes) at the end of the task execution.

**NB1** If condition of hardware component $U_i$ has implicit dependencies on component $U_j$, test of $U_j$ must be executed first.

**NB2** It is wise to wait task completion and then run test of hardware instead of stop, unload and reload task after test.
GAFT: System checking by system software - II

Testing at the level of tasks...

An algorithm for scheduling and imbedding tests of hardware used by each task should suit various number of tasks and time constrains for group of task completion...

The test of task $i$ is performed asynchronously, if it is possible to schedule it in the timeframe $t_i$ to $d_i$ as long as all other tasks can still meet their deadlines and only one test is executed simultaneously. Otherwise, execute the test synchronously. More see pp 68 -79 (http://www.it-acs.co.uk/book.html)
GAFT: Recovery by System Software

Recovery points (RP)

RP formation

- RP formation rules

RP monitoring/handling

- RP structure support
- Search of correct RPs

Language support

Run-time system support

Time

- Timer of and for:
  - Process
  - Module
  - Task
  - System RQ
  - Runtime control

Structure

- Plain: back-up for
  - Task
  - Program
  - Full size back-up
- Structure-wise:
  - Hierarchical (static)
  - Save as you go (dynamic)

Information

- Condition of:
  - a System:
  - a Task (dynamic priority)
- Run-time control
- Check sums of RPs

“Fake” formation

- Check-sums
- Indexing
- RP handling

Hardware support

- Linear
- Binary
- Modified linear
- Power of search
- Search control
RP formation: structure-wise scheme

Recovery points (RP)

Hierarchy of program

Save as you go

N Wirth’s structural programming can be exploited:

1) Structural features and limitation of visibility for lower layer variables reduce a volume of recovery points;
2) Only variables that are accessible at the level are required to save at recovery point;

Along the tree, selected path:

1) Select subset of visible variables you use
2) Create a “Key”, i.e. collection of variables to a given leaf

K = <V_{01}, ..., V_{0k} > <V_{11}, ..., V_{1p} > <V_{21}, ..., V_{2q} > <V_{31}, ..., V_{3k} > <V_{41}, ..., V_{4m} >
Recovery Points Support Summary

- The original program code should be re-processed, introducing a generation of recovery point at the beginning of each hierarchy level;

- During compilation a data structure with a list of accessible variables should be formed for each level of a program nesting;

- The program begins an execution of each successive level by calling run-time system indicating the level number, the number of accessible modules and list of variables;

- Run-time system has to monitor keys along executing a program and generate checksums along execution of recovery point;

- Run-time system can “simulate” recovery point formation when it is required;

- Execution of “generate recovery point” action consists of recording of variables accordingly key generated scheme along the execution a program: Save as you go

**NB.** Efficiency of “static and dynamic economy” methods of recovery point formation was proven to be at the order of magnitude better than other known schemes of recovery point schemes.
Searching of correct state of a program: MLR

...Faults might stay latent in the system for a long time until they trigger an error, i.e. it implies that the last recovery points have been damaged by the latent fault...

Modified Linear Recovery Algorithm (MLR)
During search MLR creates Check Sums (CS) and compares them with previously created CSs of RPs - sequence of matches and mismatches. In the given example, the detection of \( \varepsilon_1 \) is now eliminated. In other words, several successively occurring faults do not affect the correctness of the MLR algorithm as long as the fault manifestation do not overlap in time, i.e. at least one recovery point exists between the fault manifestations.

If no recovery point was generated between the two fault manifestations, recovery is also successful, but the two faults are no longer distinguishable. From the system point of view, it just recovered from one fault.

The detection of any corruptions due to the fault.

As \( \varepsilon_2 \) occurs after the elimination of \( \varepsilon_1 \), again by comparing the respective CSs match and mismatch sequences, we can identify the different stages of the MLR process. As long as the fault manifestation do not overlap in time, i.e. at least one recovery point exists between the fault manifestations, the detection of \( \varepsilon_2 \) can happen anytime, even after the detection of \( \varepsilon_1 \). In the given example, the detection of \( \varepsilon_2 \) is already completely eliminated, the second match-mismatch sequence does no longer exist and therefore the recovery process. As a result, the program execution is then resumed, as the fault is now eliminated.

To eliminate the detected fault \( \varepsilon_2 \), the MLR process can thus successfully recover from any corruptions due to the fault.
Recovery support: Hardware

Hardware checking schemes must be involved in generation of a RP and CS for each program segment.

...The RP and associated CS should be generated concurrently with program execution.

For performance gain the RP storage and checksum generator should be directly accessed by processor (to speed up RP generation and recovery).

When the error is detected, by direct intervention from Run-time system an access to RP sequence is enabled and searching of correct RP initiated.

...The checksum and recovery point device has two operating modes: the *standard mode* which creates RPs and CSs and *recovery mode* which generates only the checksums.

...The operating mode is configurable by the system software (Run-time system).
Reconfigurability: a syndrome for FT

First version of syndrome concept: witnessed by PhD students V Castano and A Petukhov.

File name: FT resolved, Sept 2010
Reconfigurability: a syndrome

From a system software point of view the syndrome is represented as a set of special hardware registers. Syndrome Registers indicates the current hardware state (current configuration, detected faults, power)...

Fault detection schemes signal to syndrome causing hardware interrupts and initiation of GAFT by run-time system. Run-time system, when necessary, executes reconfiguration of hardware.

Run-time system new functions of control are:

a) reconfiguration for reliability, performance or power-saving
b) control of graceful degradation
Reconfigurability: use of syndrome for memory

From defined by hardware design, system configurations set memory configurations:

<table>
<thead>
<tr>
<th>Mode Number</th>
<th>Number of used banks</th>
<th>Redundancy Mode</th>
<th>Number of used memory modules</th>
<th>Usable Size in Mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Triplicated + 1 Spare</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Triplicated</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Triplicated + 1 Linear</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Duplicated + 2 Spare</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Duplicated + 1 Spare</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Duplicated</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Duplicated + 2 Linear</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>Duplicated + 1 Linear</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Linear</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>Linear</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>Linear</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>Linear</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Areas of processor, interfacing zone, passive zone in terms of configurations can be defined together with their degradation sequences. Configurations and their changes supported by run-time system, in principle, enabling sequential degradation “up to the last soldier”, when single element of each section left, but system will remains operable.
After boot up, all devices are either in state OFF or in one of the blue operation modes. As the BIST automatically configures the most reliable possible memory configuration, the initial states of all devices must be acquired by reading the syndrome. Here is a short list of all possible states and a short description:

- **OFF**: The device is currently not in use, powered off and isolated for fault containment;
- **Stand-by**: The device is powered on but not yet in use, i.e. in case of memory not yet assigned to a memory bank. In case of reconfiguration, all transitions go through this state.
- **Active**: The device is in use in a non-redundant mode. In case of memory, the memory module is assigned to a bank in linear non-redundant mode;
- **Duplicated**: The device acts in duplicated mode;
- **Triplicated**: The device acts in triplicated mode;
- **Suspected**: As soon as a fault in the hardware is detected, the state of the affected hardware component is set to suspected and the testing procedures are initiated to diagnose the fault. If a device is often in this state, this could be a hint that the device might fail in the near future. For reliability purposes it might therefore be sensible to replace the component with a spare one;
- **Faulty**: Dependent on the analysis outcome, the state is then set either to Faulty if a permanent fault was diagnosed or back to the previous state if it was only a malfunction. A device in the state Faulty is powered off.

**Figure 7.19**: HW state diagram

The runtime system has to detect hardware fault, define fault type, handle hardware reconfiguration, control hardware degradation.
Reconfigurability, FT, is it important? PRESSA

Redundancy and reconfigurability of a system can be exploited differently...

...gaining in:
- Performance (P),
- Reliability (R),
- ...or saving Energy (E).

Trading of P,R,E - in next generation of stand alone, connected and distributed systems is one of the biggest challenge...
Future: PRESSA concept

Performance  Reliability  Energy

Smart  System  Architecture

       http://www.researchgate.net/profile/Igor_Schagaev/
Thanks for...

- Discussions, joint efforts: T Kaegi, S Monkman, B Kirk
- Discussions on redundancy: J C Laprie (late 80’s)
- Discussions on reliability vs. FT: S Birolini (2005-10)
- Discussions on GLM: Felix Friedrich
- Pictures: S Monkman, V Castano
- Publication support: Chong Chen (Chinese version)
- Publication support: Yurzin Bogdanov (Russian version)
Materials used...

Book: T Kaegi, I Schagaev System Software Support of Hardware Efficiency, ITACS Ltd 2013, UK
http://www.it-acs.co.uk/book.html

Papers:
http://www.researchgate.net/publication/220908686_Era_Evolving_Reconfigurable_Architecture
http://www.researchgate.net/publication/253645566_Control_Operators_vs_Graph_Logic_Model
http://www.researchgate.net/publication/252627186_Redundancy_Reconfigurability_Recoverability
http://www.researchgate.net/publication/244477145_Method_and_apparatus_for_active_system_safety
http://www.researchgate.net/publication/252628898_Operating_system_for_fault_tolerant_SIMD
http://www.researchgate.net/publication/252629068_Using_Software_Recovery_For_Determine_The_Type_of_Hardware_Faults
http://www.researchgate.net/publication/252628869_Recovery_Points_And_Hardware_Reliability_Indices
http://www.researchgate.net/publication/253584715_Using_Data_Redundancy_For_Program_Rollback
http://www.researchgate.net/publication/228728452_Hardware_Testing_on_the_Level_of_Tasks