**CSED 601 Dependable Computing**

**Assignment 2**

****

**Submitted by- Sajan Maharjan**

**POVIS ID-** [**thesajan@postech.ac.kr**](mailto:thesajan@postech.ac.kr)

**Registration Number- 20182095**

# Inherent Time Redundancy (ITR): Using Program Repetition for Low-Overhead Fault Tolerance (Summary)

As advancement in technology, we are being able to scale more and more transistors into a single microprocessor chip. However, with the increase in the numbers of transistors in the chip, there is also an increase in the chance of fault occurrence- specifically transient faults in micro-architectural operations in the CPU. One of the traditional approaches to achieving fault tolerance is via redundancy whereby programs and instructions are executed in repetition in the time and space domain. However, this approach to fault tolerance is costly in terms of power, area and performance, ultimately counteracting the benefits of scalability of modern processors. The authors of this paper propose a novel approach of fault tolerance based on micro-architectural level program repetition which engages in low-overhead microarchitecture level fault checks. The authors propose a new concept called inherent time redundancy (ITR), which provides the basis for developing low-overhead fault checks to protect the fetch and decode units of a superscalar processor.

It has been found that programs repeatedly execute the same instructions in a close interval of time. Previously, studies on instruction repetition in programs were focused on reusability of dynamic instructions for achieving high performance. Through this paper, the authors aim to exploit repetition of static instructions for low-overhead fault tolerance. The proposed concept ITR- can be used to detect transient faults in the fetch and decode units of a processor pipeline, instead of costly operations such as structural duplication or explicit time redundant execution.

The paper is organized into different sections- background information regarding instruction repetition in programs, design method of inherent time redundancy, fault injection experiments & fault coverage and comparison of power and performance with processors.

## Instruction Repetition in Microprocessors-

In SPEC2K (software provider for aviation industry) programs, instructions are grouped into traces containing a group of 16 instructions or that terminate when encountering a branching instruction. It has been found that a relatively small number of static instructions contribute to a large number of dynamic instructions. For both integer and floating point benchmark, dynamic instructions are repeated after reaching a certain number of static traces. This shows that there is a certain degree of inherent time redundancy in programs. This paper aims to exploit the highly repetitive instruction traces in programs to record and confirm microarchitecture events. Also, given the fact that a small number of static traces contribute to a large number of dynamic instructions- suggests that a small structure is sufficient to record such events. To make use of such repetitions, the authors suggest an addition of a small cache to record microarchitecture events. A hit or miss of the repeated instruction when compared with the instruction stored in ITR cache enables the detection of faults or unavailability of a counterpart to check the correctness of the micro-architectural events or no errors encountered. However, in the case of a missed instance being evicted from the cache before it is accessed, there is a loss in fault detection.

## ITR Design Methods-

ITR is used to support transient fault protection to the fetch and decode units of a processor. The signal generated by the decode unit are continuously combined until the end of a trace to generate a signature. Signatures are then dispatched to the ITR ROB (Reorder Buffer). The ITR ROB is sized to match the number of branches that could exist in the processor. Each ITR ROB entry contains the start PC (program counter) and the signature of the trace. ITR ROB entry also contains control bits (chk, miss, retry) used for checking the status of the trace with the copy in the ITR cache. ITR cache stores signature of previously encountered traces and is indexed with the start PC of a trace. Signature generation could be done in multiple ways, however the authors opted to XOR the signals of a new instructions with the corresponding signals of the previous instruction. Using XOR, it is not possible to determine which instruction in the trace caused the fault, but just detect if a fault has occurred or not. Each instruction trace in the ITR ROB accesses the ITR cache at dispatch. If the trace hits, the signature is read from the ITR cache and checked with the signature of the trace. The chk (check) bit is set regardless of the outcome of the comparison. In case of mismatch, the retry bit of ITR ROB entry is set while in the case of a trace miss, the miss bit of the ITR ROB entry is set. Thus the setting of the retry bit indicates a transient fault occurred in either the new trace or the previous trace stored in the ITR cache. On fault detection, there are two different recovery strategies- a lightweight flush and restart of the processor or a more expensive program restart. To identify which trace instance (the current instruction trace or the previous instruction trace) was faulty, the processor is flushed and restarted from the start PC of the new trace. If the signature mismatch again, we can confirm the previous trace executed with the fault. This means the processor’s architectural state could be corrupted and the program is aborted which requires a program restart. In the case of signature match after restart, it indicates the new trace we had received earlier was faulty and recovery through flushing and restarting the processor was successful. The ITR cache is of definite size and is not able to store all instruction traces. Replacing an existing instruction trace signature in the ITR cache is based on the LRU (Least Recently Used) algorithm. Thus evicting a trace signature before being referenced could lead to loss in fault detection coverage i.e. the number of instructions in which a fault can be detected. Also on the other hand, ITR cache misses decrease the fault recovery coverage i.e. the number of instructions in which a fault can be detected and successfully recovered by flushing and restarting the processor. This is because on a miss, an unchecked trace signature is entered into the cache. If the unchecked trace is faulty, the fault is only detected in the future by the next instance of the trace. The authors also suggest that the recovery coverage can be improved by using a coarse-grain check-pointing scheme. In the case when lightweight processor flush and restart is not possible, recovery can be done by rolling back to the previously taken coarse-grain checkpoint instead of program abort.

Faults may arise in various components of the ITR as well. Faults on signature generation component or faults on start PC or even fault in the ITR cache. The authors advise to avoid such faults by making use of a parity bit. On a signature mismatch, retry is attempted. If the signature mismatches again, then the parity is checked on the trace signature in the cache. A parity error indicates an error in the ITR cache and not the previous instance of the trace. A fault in the PC or the next-PC logic causes incorrect instructions to be fetched. Such situations present a vulnerability of the ITR cache and require other method of protection. An easy method of protection of the PC can be implemented by the addition of a simple commit PC and asserting that a committing instruction’s PC matches the commit PC.

ITR Cache configuration parameters- associativity and cache size, determines the fault detection and fault recovery coverage. For a given associativity, a smaller cache size increase the number of evictions of unreferenced ITR signatures and the number of ITR cache misses. The coverage loss of the instruction trace correlates to the distance between repeating instructions in the trace and the percentage of dynamic instruction contained within static traces. Traces that repeat far apart contribute to a large number of dynamic instructions and hence have a higher loss in fault coverage. Increasing the cache capacity (no. of signatures to hold) can effectively mitigate such losses. Also given the fact that the fault detection and fault recovery coverage is influenced by the number of the instructions in the trace, which is not uniform. Thus, in some cases, higher associativity of cache sometimes happens to show slightly higher loss in fault coverage rather than lower associativity. Such losses in fault coverage can be mitigated by redundantly copying the fetch and decode traces on a miss in the ITR cache. However, doing so would results falling back on conventional time redundancy.

## Fault Injection Experiments-

The authors performed fault injection experiments on a detailed cycle-level simulator that models a microarchitecture. As many as 1000 faults were introduced on the decode signals by randomly flipping a selected bit. Side by side another fault free simulator was run along with the faulty simulator to determine whether or not the architectural state was corrupted. The injected faults based on detection by ITR could be categorized as- detected by ITR, undetected within observation timeframe and undetected for sure. On the basis of ability to corrupt the architectural state, injected faults would also be classified as- corrupting the architectural state (SDC – Silent Data Corruption) or not (masked). Also recoverability of such faults were studied. As high as 95.4% of the faults were detected by the ITR and 32% of the injected faults were detected and recovered by ITR that would have led to a silent data corruption state otherwise. Only a few faults, accounting to 1% of the faults were detected but not recovered through ITR. On average, 4.5% of the faults go undetected by ITR. 2.6% of the faults lead to SDC that were not detected by ITR. Only 0.1% of the faults lead to a deadlock which were not detected by ITR but detected by use of a watchdog timer. These experiments confirm most of the faults injected were detected by ITR and a significant portion of these were recovered.

## Comparison of Area, Power and Performance-

The IBM S/390 G5 processor, I-unit was compared with the ITR architecture. In the IBM S/390 G5 processor, the fetch and decode units, is duplicated and signals from the two units are compared to detect transient faults. The area of the I-unit is 1.5cm X 1.4cm i.e. 2.1cm2 while the area of ITR cache is 1.5cm X 0.2cm i.e. 0.3cm2. Thus, the area of the ITR cache is seven times smaller when compared to the I-unit. The modelling of power consumption is done by measuring the number of accesses to the ITR cache and the instruction cache of the processor. Both cache model models are fed into CACTI to obtain the energy consumption per access. The energy consumption of the ITR cache and the IBM Power4 I-cache were compared. The CACTI numbers were: 0.87nJ per access for the I-cache, 0.58nJ per access for the ITR cache. This data shows that the ITR approach is more energy efficient rather than fetching twice from the instruction cache. Thus ITR cache is more cost-effective than straightforward space redundancy in IBM Power4 mainframe computer. Although, complete structural duplication provides more robust fault tolerance rather than the ITR cache.

## Conclusion-

In this paper, the author introduced a new approach to develop low-overhead fault checks for a processor, based on inherent time redundancy in programs. This approach could be used to provide micro-architectural support to protect the fetch and decode units of the processor. They also presented how faults could be detected and recovered in case of transient faults by injecting faults. Also they compared the effectiveness of the ITR approach with existing mainframe computers in terms to power, area and performance.