2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

Error Correlation Prediction in Lockstep Processors for Safety-critical Systems

Emre Ozer, Balaji Venu, Xabier Iturbe, Shidhartha Das, Spyros Lyberis, John Biggs, Peter Harrod and John Penton
Arm Ltd.
110 Fulbourn Rd, Cambridge, CB1 9NJ, United Kingdom
Email: {firstname.lastname}@arm.com

***Abstract*—This paper presents a new phenomenon called *error correlation prediction* for lockstep processors. Lockstep processors run the same copy of a program, and their outputs are compared at every cycle to detect divergence, and have been popular in safety-critical systems. When the lockstep error checker detects an error, it alerts the safety-critical system by putting the lockstep processor in a safe state in order to prevent hazards. This is done by running the online diagnostics to identify the cause of the error because the lockstep processor has no knowledge of whether the error is caused by a transient or permanent fault. The online diagnostics can be avoided if the error is caused by a transient fault, and the lockstep processor can recover from it. If, however, it is caused by a permanent fault, having prior knowledge about error's likely location(s) within the CPU speeds up the diagnostics process. We discover that the error's type and likely location(s) inside CPUs from which the fault may have originated can be predicted by analyzing the output signals of the CPU(s) when the error is detected. We design a simple static predictor exploiting this phenomenon and show that system availability can be increased by 42-64% with an overhead of less than 2% in silicon area and power.**

***Index Terms*—Lockstepping, redundant execution, fault tolerance and functional safety**

I. Introduction

Lockstepping is an error detection technique that executes copies of a program on redundant hardware and compares the outputs of the redundant hardware at every cycle to detect a divergence [1]. A lockstep error checker detects the divergence, and signals the error to a system controller. Lockstepping can be implemented at various levels depending on the *sphere of replication* [2], which is defined as a logical zone containing redundant hardware components. Inputs to the sphere are replicated to feed into redundant hardware, and outputs of redundant hardware are compared at the sphere boundary. Thus, hardware components inside the sphere are protected by the lockstep error checker.

There are three types of lockstepping in computer systems: system, sub-system and CPU level lockstepping as shown in **Figure 1**. The sphere of replication in system-level lockstepping in **(a)** contains redundant cores and main memories. The inputs are replicated at the I/O boundary, and the outputs from replicated main memories are compared by the checker. The sub-system level lockstepping **(b)** has a sphere of replication containing replicated cores but share the main memory and I/O. The inputs are replicated at the main memory and I/O boundary, and the outputs from the replicated caches are compared by the checker. The CPU-level lockstepping **(c)** has a sphere of replication containing only redundant CPUs, i.e., caches, including L1, are not in the sphere. In all three types, the checkers can catch the divergence and detect the error but cannot identify the error type, e.g., soft or hard, and nor its origin, e.g., where in the CPU the fault actually occurred. We define the lockstep soft error as the error caused by a transient fault while the lockstep hard error is defined as the error caused by a permanent fault. Transient faults are temporal events and are not sticky while permanent faults are sticky and occur repeatedly at the same location [3].

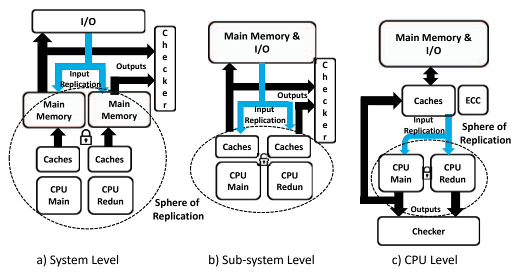


Fig. 1. Three different ways of implementing lockstepping - a) System level, b) Sub-system level and c) CPU level

Lockstepping has been adopted by industry for its favorable properties such as simple implementation, a high error coverage and software transparency in spite of its high cost overhead. System and sub-system level lockstepping are very popular in high availability servers [4] [5] where they can afford to use redundant microprocessors and main memories.

CPU-level lockstepping becomes common in safety-critical systems, more specifically in the automotive world, [6] [7] [8] [9] because it can achieve the most stringent safety level, and is less costly to implement than system or sub-system level lockstepping. Safety-critical systems must provide functional safety to deal with systematic (i.e. design and manufacturing)[1] and random faults (i.e. transient and permanent).

Functional safety is an important concept in safety-critical systems, and is defined as the absence of unacceptable risk due to hazards caused by malfunctioning electronic systems [10]. The end goal of functional safety is to prevent death or injury to the people due to a failure in electronic systems

[1]Faults that can only be prevented by applying process or design measures

(e.g., automobile, train, aircraft). A safety integrity level is assigned to an electronic system to measure the level of risk for a safety measure used by the system. In automotive functional safety, there are four levels of automotive safety integrity levels (ASILs), ASIL-A being the least and ASIL-D being the most stringent. For example, chassis, powertrain, power steering and anti-lock braking system (ABS) need to provide ASIL-C or D capability while body systems often provide ASIL-A or B. CPU-level lockstepping is mainly used in ASIL-D-capable electronics control units (ECUs).

Any safety mechanism in a safety-critical system must guarantee that the system reaches a safe state to prevent hazards upon detecting an error. Typical hazards are caused by errors in the ECUs of the ABS and power steering that may lead to fatal crashes. The system has to reach a safe state to prevent fatal crashes and therefore errors must be detected and handled. Functional safety standards (e.g. ISO26262 [10]) provide guidelines in system design to handle random errors (both soft and hard) in order to meet certain safety integrity levels regardless of how rare these errors are. When a fault occurs and manifests as an error, the safety mechanism must detect it[2] and signal it to the system. The time interval from fault occurrence to error detection is called *error detection time*. Once the error is detected, the safety mechanism must ensure that the system reaches a safe state before any hazard occurs. The interval between the error detection point to the safe state is called *error reaction time* that is an important metric in safety-critical systems.

The safe state concept and its critical timings are shown in **Figure 2**. A safe state must be reached within the hard deadline of the system. Missing a hard deadline can be fatal. Thus, error reaction time is statically provisioned for the worst-case error handling scenario, and becomes a critical parameter affecting the hard deadline, and must not be violated. However, any reduction in the provisioned error reaction time at run time is safe, and increases the availability of the system.

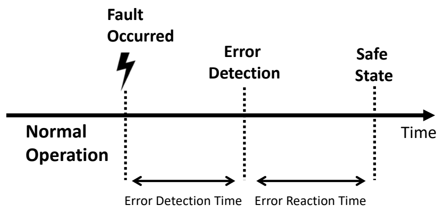


Fig. 2. Safe state concept in safety-critical systems

In this paper, we focus on CPU-level lockstepping as a safety mechanism aiming at safety-critical systems with most stringent safety integrity levels in which missing a hard deadline may lead to a fatal crash. When a divergence in lockstepped CPUs is detected, the lockstep error checker has to alert the system controller to put the system into a safe state. If the system controller knows that the error is soft or hard, it can take an appropriate and swift action. If the error is hard, for example, it can turn on an emergency indicator on the dashboard of the car. If the error is soft, the system controller can reset the lockstep processor and restart the real-time task. Thus, error handling measures are different for hard and soft errors. However, the system controller is only aware of the lockstep error, and does not know the type of the error.

[2]We try to avoid using error and fault interchangeably in the paper. We consider an error being manifestation of a fault, so not every fault is an error.

There are two alternative scenarios that can be taken by the system controller when a lockstep error is detected:

Best-case scenario: The error is assumed to be soft and the system tries to recover from it by resetting the system and restarting the real-time task. This is because it assumes that soft errors are more common than hard errors, and tries to recover from the error quickly. What happens if another error occurs, say a million cycles, after the reset & restart? The system does not know whether the second error is independent (i.e. soft) or correlated to the previous error (i.e. potentially hard). The system can keep "reset & restart" the CPUs by assuming a soft error until a certain number of errors is observed after which the online diagnostics can initiate. This approach is considered to bring non-determinism to the safety-critical system, and therefore more appropriate for low safety integrity levels (e.g. ASIL-A).

Worst-case scenario: The error is assumed to be hard even though hard errors are rare, and the system runs the online diagnostics to find any hard error. If the online diagnostics does not find an error, then the system controller decides that the error is a soft error, and resets & restarts the real-time task. Safety-critical systems equipped with CPU-level lockstepping target most stringent safety integrity levels (e.g. ASIL-D), and therefore require deterministic behaviour in the presence of errors. The execution time of the online diagnostics is deterministic, which makes the hard deadline of the system deterministic, and thereby makes the worst-case scenario approach preferable to the best-case scenario.

In this paper, we use the worst-case scenario approach to handle lockstep errors. Thus, when the system controller runs the online diagnostics to find a hard error, it does so without any prior knowledge where to start the diagnostics within the CPU because the origin of the fault is unknown. If the likely location(s) within the CPU from which the fault may have originated are known, then the diagnostics process can be performed more efficiently. Similarly, the system controller can take the appropriate action if it has some hints about the error type. We discover that the error type and likely location(s) inside the CPUs can be predictable in lockstep processors by capturing and analyzing the outputs of the CPUs when the error is detected. We call this phenomenon *lockstep error correlation prediction*. The main motivation of lockstep error correlation prediction is to reduce error reaction time, which is a statically provisioned for the worst-case scenario, at run time in order to increase system availability.

We build a simple static predictor to demonstrate the phenomenon. Lockstep error correlation prediction reduces the error reaction time, and therefore increasing system availability by 42-65% depending on the granularity of the CPU logical organization. The overhead of the predictor is less than 2% in area and power with respect to a dual-CPU lockstep processor. To the best of our knowledge, this is the first paper that discovers and studies the error correlation prediction phenomenon in lockstep processors.

The paper is organized as follows: *Section II* gives a background to CPU-level lockstepping in safety-critical systems. *Section III* describes the error correlation prediction phenomenon and predictor design. *Section IV* provides the methodology for fault injection study and measurements, and *Section V* presents the experimental results. *Section VI* describes the related work, and finally, *Section VII* discusses various points related to the error correlation prediction, and *Section VIII* concludes the paper.

II. Background on CPU-Level Lockstepping

CPU-level lockstepping can be implemented in dual-modular redundancy (DMR) or multiple-modular redundancy (MMR) configurations (e.g. triple-modular redundancy (TMR)) [11]. CPUs share the caches that are protected by some form of ECC mechanism, so the caches are kept out of the sphere of replication.

The real challenge of implementing lockstepping is that main and redundant CPUs must produce exactly the same outputs in every cycle when there is no error [1]. This implies that CPUs must be initialized to an identical internal state on reset. A flip-flop initialized to a different value in the CPUs on reset may lead to a divergence in the normal operation. Thus, lockstepping requires a careful design to keep the equivalence of main and redundant CPU internal states on reset.

1) Lockstep Error Detection and Reaction: The lockstep error checker reads the output ports of main and redundant CPUs at every cycle, and looks for a divergence. The checker in DMR does not know which of the two CPUs caused the error when it detects the divergence, and therefore flags the error to the system controller and stops both CPUs. On the other hand, the checker in MMR identifies the erring CPU through the majority voter, and flags the error along with the erring CPU ID to the system controller.

In both cases, the checkers in both DMR and MMR configurations do not know whether the error is soft or hard. If it is a soft error, the system controller can recover from the error. On the other hand, the system controller cannot recover from a hard error, so it must alert the system and user (e.g., driver) with the hardware failure.

The system controller initiates the built-in-self-test (BIST), which is a diagnostics process, to identify the cause of the error. The BIST checks both CPUs in DMR since the erring CPU is unknown while only the erring CPU is checked in MMR mode. There are two alternative BIST techniques to identify the cause of an error in CPUs: 1) Logic BIST (LBIST) or 2) Software BIST (SBIST). The system controller will use either LBIST or SBIST to identify the cause of the error detected by the lockstep error checker depending on which BIST is implemented in the system.

LBIST [12] [13] is an online error detection technique that uses dedicated test hardware using scan chains to detect hard errors in logic. LBIST uses a pseudo-random generator to generate test patterns to several internal scan chains in the CPU, and the responses from the scan chains are compacted to create a signature. A hard error is detected by comparing the signature to the expected signature for the pattern.

SBIST [14] [15] is a software-based online error detection technique that uses special software test libraries (STLs) written in the instruction sets of the CPU to generate test patterns, and runs them to test the CPUs in order to detect stuck-at-faults. Normally, an STL is created for each unit within the CPU to ease development time. The advantage of SBIST is that it is a relatively low-cost solution as it does not require extra test hardware like in LBIST.

2) Lockstep Error Handling: If BIST verifies that the error is indeed a hard error, the system controller in DMR stops both CPUs and alerts the rest of the system with an unrecoverable fatal error. Although the system controller in MMR has the option of disabling the erring CPU and continuing with the other correct CPUs, the erring CPU can easily be brought back into the lockstep if the error is known to be caused by a transient event [16] to keep a high system availability.

If the L or S BIST does not find any hard error (assuming 100% error coverage), this implies that the error is actually soft, which can be recovered by the system controller. The system controller in DMR resets both CPUs to restart the current task or starts a forward recovery process in MMR by saving the correct architectural state to the memory through majority voting. After the architectural state is saved, all CPUs are reset and the architectural state is restored from the memory to bring all CPUs back in lockstep.

III. Lockstep Error Correlation Prediction

If the system controller uses LBIST to identify the cause of the detected error, the LBIST is initiated to generate random patterns into multiple scan chains in the CPU(s). If it uses SBIST, the SBIST is initiated to run the STL of each CPU unit in a pre-determined (or random) order to find any hard error. If a hard error is found for a particular pattern in LBIST or in a particular CPU unit in SBIST, the diagnostics stops and the system controller is alerted. Otherwise, the BIST continues. The main issue is that error reaction time depends on the choice of random patterns in LBIST or the default order of CPU units by the SBIST. If likely location(s) from which the error may have originated is known, then error reaction time can be reduced by optimizing the BIST process.

Our hypothesis is that it can be possible to predict error's type and likely locations within the CPUs by only capturing and analyzing the output port signals of the CPUs at the time of error detection. We call this phenomenon *lockstep error correlation prediction*. If different fault types, e.g. transient and permanent, manifest differently at the CPU outputs and if faults occurring in a particular CPU unit have unique and distinguishable signatures at the CPU output ports, then it can be possible to predict error's type and likely location(s).

The key benefit of such a predictor is to reduce error reaction time by performing the BIST more efficiently, and therefore to increase the availability of the safety-critical system. For example, the LBIST can constrain the test search space to the scan chains relevant to the predicted CPU units while SBIST starts testing the units in the predicted order rather than doing it in a pre-determined or random order. Although our technique is applicable to both BIST processes, our focus in this paper will be on an SBIST-based diagnostics.

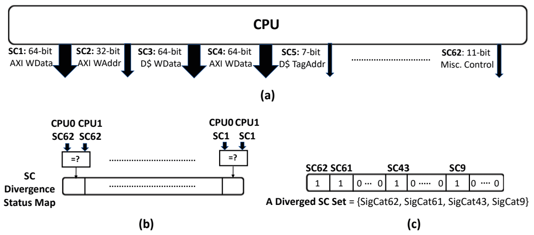


Fig. 3. (a) Output port signals coming out of a CPU and its signal categories (SCs); (b) SC divergence status map; (c) the concept of a diverged SC set.

A. *Location Prediction*

We define some terminology before describing the location prediction concept. A *signal category or SC* refers to a group of related signals, e.g. 32-bit Dcache address bits refer to an Dcache address SC. A *diverged SC* means that the outputs of the CPUs disagree in that SC when the error is detected. A *diverged SC set* refers to a set of such disagreeing SCs when the error is detected.

Figure 3 shows the types of output port signals and how they are categorized, the formation of a divergence map and an example of a diverged SC. The *Arm Cortex-R5* CPU used in the experiments can be categorized into 62 different SCs as shown in (a). A signal divergence status map is created for each SC in (b). The signals from the same SC in two or more redundant CPUs are compared and reduced to a 1-bit map denoting whether the SC diverged or not. A fault can manifest as an error in many SCs at the CPU output ports. Thus, a diverged SC set consists of SCs with their associated bits set in the divergence status map as shown in (c). There are about 1200 distinct diverged SC sets observed for all injected soft and hard faults.

To demonstrate the lockstep error correlation prediction phenomenon, a histogram data of hard error signatures is collected for a dual-CPU lockstep system using the fault injection and evaluation methodology that will be described in *Section IV*. **Figure 4** shows the probability distributions of three CPU units over all possible *diverged signal category sets* observed across all injected hard faults. Each graph in **Figure 4** shows the hard error probability distribution of a CPU unit. For example, the diverged SCs are identified when a hard fault is injected in the *Bus Interface Unit* and manifests in the output ports. The diverged SCs form a set, and the histogram count of the set is incremented. This is repeated for all injected hard faults in the *Bus Interface Unit*. The shape of the probability distribution of each CPU unit defines its signature. If the shape of one CPU unit probability distribution differs from that of another unit's probability distribution, then it is very likely to make a good prediction as to the origin of the error.

To quantify the similarity of two probability distributions, the *Bhattacharyya Coefficient* (BC) [17] is calculated for each CPU unit but we only show the three CPU units with minimum, median and maximum BC values in the figure. The BC is a statistical metric often used to quantify the similarity between two histograms and empirical distributions. It generates a value between 0 and 1, 0 being not similar at all while 1 being perfectly similar. For each CPU unit, we calculate the BC across other CPU units and show their average in each graph. For example, the *Bus Interface Unit* has a BC of 0.37 on average across all other units, which implies that its error manifestation signature is not similar to other units. Overall, the average BC for all CPU units (including the ones not shown in the figure) is around 0.39, which supports our hypothesis that there is an acceptable correlation between the lockstepped CPU outputs at the time of an error and the location of the error in the CPU.

Figure 5 shows the probability distributions of the same three CPU units for soft errors. Similar to hard errors, there is a good correlation between the CPU outputs and the location of the error in the CPU with slightly better BC than hard errors. The average BC for all CPU units (including the ones not shown in the figure) is around 0.32.

B. *Type Prediction*

Similar to the error location prediction, the type of a detected error (i.e., soft or hard) can also be predicted from the output signal port divergence information. A soft error is a transient event and its effect on the sequential element (i.e., flip-flop) will disappear in the next cycle. On the other hand, a hard error is a stuck-at fault, and its effect on the sequential element is permanent and does not disappear in time.

Because error detection in lockstep systems is not immediate, and can take a longer time until its effect propagates to the CPU output ports, a permanent fault on a sequential element may spread to more CPU output ports than a transient fault. In fact, our experimental results show that 54% more diverged SC sets are observed at the time of divergence caused by hard errors than soft errors when faults are injected to the same sequential elements. Thus, we can expect the CPU output port signatures for a lockstep soft error to noticeably differ from the output port signatures for a lockstep hard error.

The probability distributions **Figure 4** and **Figure 5** show evidence that the soft and hard error probability distributions at the same CPU unit can be dissimilar. For example, the BC value between the hard and soft error probability distributions of *Instruction Memory Control Unit* is 0.3[3], which implies that they are not very similar and the error type can be predicted. On the other hand, the BC value between the hard and soft error probability distributions of *Data Processing Unit* is 0.95[3], which implies that they are very similar distributions and it may not be possible to differentiate between a soft and hard error. The average BC value between the hard and soft error probability distributions at the same CPU unit is 0.6[3]. In spite of exhibiting not very strong dissimilarity, it encourages the predictability of the error type.

[3]Not shown in figures

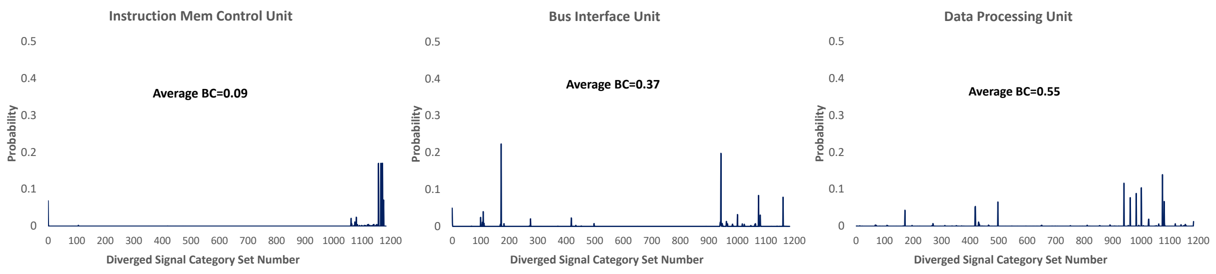


Fig. 4. Hard error distributions for three CPU units over all possible of diverged SC sets across all injected hard faults. The three CPU units are from left to right with the minimum, median and maximum Bhattacharyya coefficients (BCs).

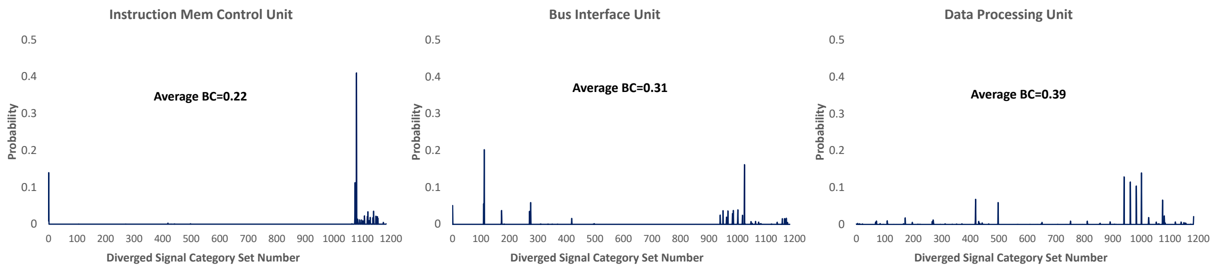


Fig. 5. Soft error distributions for three CPU units over all possible of diverged SC sets across all injected soft faults. The three CPU units are from left to right with the minimum, median and maximum BCs.

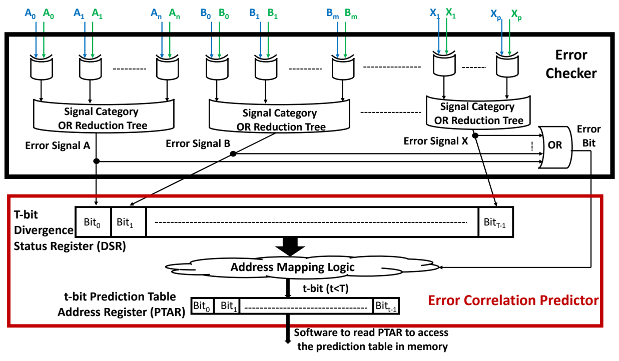

Error Checker
Signal Category OR Reduction Tree
Error Signal A
Error Signal B
Error Signal X
OR
Error Bit
T-bit Divergence Status Register (DSR)
Address Mapping Logic
t-bit (t<T)
t-bit Prediction Table Address Register (PTAR)
Error Correlation Predictor
Software to read PTAR to access the prediction table in memory

Fig. 6. Error correlation prediction logic shown inside the red box is tightly coupled to the error checker shown in the black box.

C. Predictor Design

Figure 6 shows the error correlation prediction logic for a dual-CPU lockstep processor. The predictor is tightly coupled to the error checker that compares each signal coming from the two lockstepped CPUs. The blue signals come from one CPU while the green signals come from the other. The SC OR reduction trees that are used to calculate the final error signal in the error checker generate the diverged SC map information in the error correlation predictor. Essentially, the signal ports that belong to the same SC are ORed using an OR reduction tree, and the 1-bit output of each SC reduction OR tree is also written to a T-bit register called the *Divergence Status Register (DSR)* that has a bit per SC. Initially, the DSR is reset to zero, and the associated DSR bit is set if any bit in a SC differs.

For 62 distinct SCs, the DSR needs to be a 62-bit register. The data in the DSR represents a set of diverged SCs when the lockstep error is detected. The DSR could have been used to access a predictor table to predict the location and type of the error. However, a 62-bit address is not necessary to access the table because, as noted in the previous section, we observe only about 1200 distinct diverged SC sets across all error datasets. This implies that an 11-bit address is sufficient to represent these sets, and is therefore used to access the prediction table. Thus, the 62-bit DSR is mapped into an 11-bit register called the *Prediction Table Address Register (PTAR)* through an address mapping logic. The 11-bit PTAR is used to access the prediction table with 1200 entries each of which keeps the predicted CPU location(s) and error type information associated with a distinct SC. The table also has an extra entry to which all unobserved SCs, i.e., not in the 1200 category sets, are mapped. When this entry is accessed, the error type is always taken to be a hard error, and the default order of CPU units is retrieved instead of some predicted order.

The prediction table can be kept in an on-chip memory (e.g., cache or scratchpad) or in an off-chip memory (e.g., DRAM), which is already protected by ECC, rather than in a separate hardware table. This is because the error correlation prediction is static in nature (i.e., the prediction table contents do not change during the lifetime of the CPUs). Soft or hard errors are rare events, and therefore predicting them statically makes the prediction hardware simpler. When the checker detects an error, the CPUs are interrupted, and the lockstep error handler software is invoked to read the predictor table address stored in the 11-bit PTAR[4]. The handler reads the predicted CPU location(s) and error type information pointed by the 11-bit PTAR, and takes the appropriate actions based on the prediction information.

[4]Similar to an exception handler accessing the exception vector table

Although the predictor is demonstrated for a dual-CPU lockstep processor in **Figure 6**, the same predictor hardware can also be used in systems supporting more than two lockstep CPUs. The predictor hardware complexity does not change with the increasing number of CPUs. This is because the predictor will take the outputs of the SC OR reduction trees as inputs. Thus, the prediction logic scales well with the increasing number of lockstep CPUs.

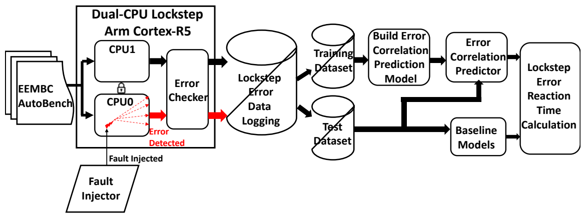

Dual-CPU Lockstep Arm Cortex-R5
CPU1
CPU0
Error Checker
EEMBC AutoBench
Error Detected
Fault Injected
Fault Injector
Lockstep Error Data Logging
Training Dataset
Test Dataset
Build Error Correlation Prediction Model
Error Correlation Predictor
Baseline Models
Lockstep Error Reaction Time Calculation

Fig. 7. Evaluation methodology and framework

IV. Evaluation Framework

We develop an evaluation framework for injecting faults, detecting errors, logging data and building baseline and prediction models in order to demonstrate the error correlation prediction phenomenon. **Figure 7** shows the stages of the framework such as the fault injection tool to inject faults into a CPU while running applications, the checker to detect errors that manifest at the outputs of CPUs, error data logging and model development. Logged error data is split into training and test datasets using random sampling and 5-fold cross validation. Training datasets are used for developing error correlation predictor, and test datasets are used to evaluate the performance of the predictor and baseline models.

We use a dual-CPU lockstep *Arm Cortex-R5* processor [18], which is already used in safety-critical systems. A real system must be used to evaluate the error behaviour in lockstep CPUs rather than a high-level microarchitectural simulator and fault injection tool. As explained earlier, the microarchitectural state (i.e., every flop) of lockstepped CPUs must be the same state after reset so that they can run in lockstep without any divergence in normal operation. Also, faults must be injected to every flip-flop in the CPU in order to characterize the lockstep error behaviour and build robust prediction models.

A. Fault Injection and Detection Framework

We simulate the dual-CPU lockstep *Cortex-R5* netlist using the Synopsys VCS tool [19] by running the *EEMBC AutoBench* benchmark suite [20]. The benchmarks are compiled and the binaries are used to initialize the memory in the simulated CPU design. Then, the VCS simulator is invoked to simulate the design. *AutoBench* is an industrial standard benchmark suite used to measure the performance of microprocessors and microcontrollers in automotive applications. It consists of benchmark kernels that are used in automotive applications such as tooth-to-spark (locating the engine's cog when the spark is ignited), road speed calculation, engine knock detection, vehicle stability control and occupant safety systems. Each kernel consists of an outer loop that runs continuously. For example, in the tooth-to-spark kernel, the CPU controls fuel injection and ignition in the ECU of the engine combustion in real-time. The operating conditions from the ECU are sent to the CPU as inputs, and then the CPU adjusts the output values for fuel injector duration and ignition timing. The entire process is repeated on each loop iteration, i.e., reading inputs and calculating new outputs.

We develop an in-house tool to inject random faults and to evaluate the performance of the baseline and proposed techniques. The fault injection methodology is so rigorous that we inject a total of 10 million soft and hard faults in all flip-flops in *Cortex-R5* CPU across all benchmarks, ensuring that every flip-flop experience many soft and hard faults. Both hard and soft faults are random in nature, and a single random event causes a soft or hard fault in a flip-flop. We inject random faults in one CPU during the simulation, and rely on the lockstep error checker logic to detect the divergence. The *Cortex-R5* CPU is organized into seven coarse-granular units as shown in **Figure 8**.

Each benchmark simulation time is divided into 64 equally sized intervals. When a benchmark runs, only a single random fault is injected in a selected flip-flop in one of these intervals, and the benchmark runs to completion. A random soft fault is simulated by inverting the value stored in a flip-flop for a simulation clock cycle, while a hard fault is simulated by keeping a stuck-at value on the flip-flop until the end of simulation (i.e., covering both stuck-at 0 and 1 faults). This becomes one fault injection experiment. The same benchmark runs and a fault is injected into the same flip-flop but at a different interval, and this becomes the second fault injection experiment. This goes on until faults are injected in every flip-flop in the CPU unit. The same process is repeated for all CPU units across all benchmarks.

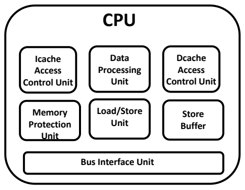


Fig. 8. Cortex-R5 CPU consists of seven logical units

TABLE I
Some important statistics from fault injection experiments

Statistic	**[Min, Mean, Max]**
Soft Error Manifestation Rate	[0.2%, 5%, 27%]
Hard Error Manifestation Rate	[3%, 40%, 88%]
Soft Error Manifestation Time	[2, 700, 80k] in cyc
Hard Error Manifestation Time	[2, 1800, 130k] in cyc

For each fault injection experiment that leads to an error, an error log is generated capturing the most relevant information such as fault injection location and cycle time, error manifestation time etc. When a fault manifests as an error, the error checker detects the divergence and captures around 2500 signals from each CPU, and logs them (i.e., lockstep error data logging in **Figure 7**). 2 million manifested error data points that are logged are then split into training and test dataset bins.

It took us 2 weeks on a cluster of servers to complete the fault injection experiments. The main reason behind the long runtime of experiments is that we had to repeat the same experiments for each fault type (i.e., soft fault, stuck-at-0 and stuck-at-1 hard faults) for every flip-flop in a product-class CPU running every benchmark.

B. Lockstep Error Detection/Manifestation

Table I shows some important statistics from the fault injection and detection experiments. Not every injected fault manifests as an error at the outputs of the CPUs as some faults are masked out. The first two rows show the minimum, mean and maximum lockstep error detection or manifestation rate of all CPU units for soft and hard. An error manifestation rate of a unit is calculated as the number of manifested errors detected by the lockstep checker divided by the total number of injected faults in that unit. The mean soft rate manifestation rate is very small around 5% while it is 40% for hard errors. Overall, about 20% or 2 Million of all injected faults manifest as errors at the output of the CPU ports. The last two rows of the table shows the minimum, mean and maximum lockstep error detection or manifestation times of all CPU units for soft and hard errors. The lockstep error manifestation time is the time interval from the fault occurrence cycle to the lockstep error detection cycle. This is equivalent to *error detection time* in **Figure 2**. Overall, the average error manifestation time for all soft and hard errors over all units is 1300 cycles.

C. Lockstep Error Reaction Time

Lockstep error reaction time or LERT is defined as the time interval from the lockstep error detection to reaching a safe state, and is equivalent to *error reaction time* in **Figure 2**. Reduction in LERT is a measure of increase in system availability. The LERT is calculated for the baseline and prediction models using the test dataset.

1) Baseline Models: The flowchart of the baseline models to handle errors in lockstep processors is shown in **Figure 9a**. When the error is detected, the BIST is initiated in both CPUs to identify whether it is a hard or soft error since the cause of the error and the erring CPU are unknown. The lockstep processor used in the experiments equips a SBIST mechanism, and there is a software test library (STL) for each CPU unit. The execution time of each STL is not uniform, and depends on the complexity of the unit. The SBIST runs the STLs in a fixed or random order. If a hard error is found in a CPU unit, the SBIST stops, and the system reports a failure. If there is no hard error found after running the STL of every CPU unit[5], the system decides that it is a lockstep soft error, resets the CPUs and restarts the application.

The baseline model has three variants (two static and one dynamic): a) The first static baseline model keeps the CPU units in ascending order of their STL latencies. This is because the STL latencies are non-uniform (see **Table II**), and therefore the faulty CPU unit can be found much faster if the CPU units with shorter STL latencies are tested first. b) The second static model keeps the units in descending order of their error manifestation rates. The rationale behind this baseline model is that the units with high manifestation rates should be tested by the SBIST first before looking at the others because they are more likely to expose faults to the lockstep error checker. c) The third model is a dynamic one and orders the STLs of the CPU units in a pseudo-random manner, that is a new random order of the CPU units for BIST is generated for each detected error.

The LERT for the baseline models is calculated as the number of cycles to run the STLs of CPU units until a hard error found or run-to-completion (i.e., no hard error found) followed by the restart penalty due to soft error handling. The restart penalty is determined by the delay in resetting the CPUs and restarting the outer loop of the benchmark.

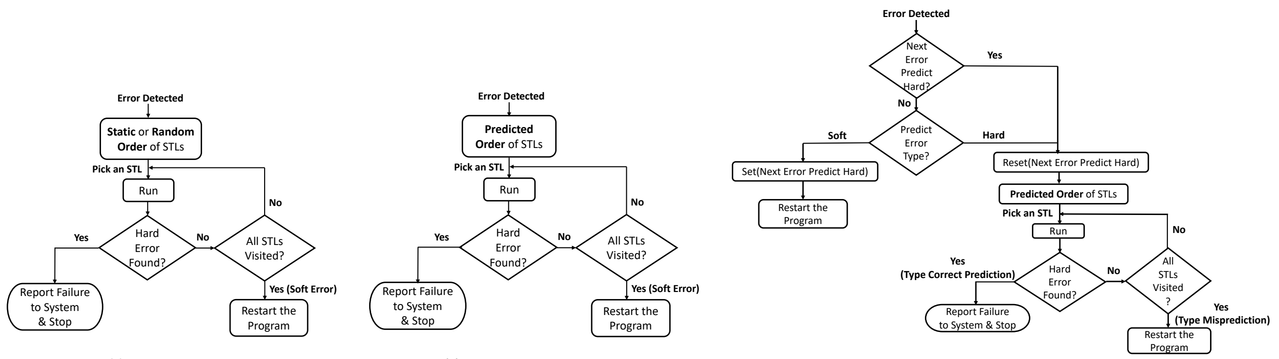
2) Prediction Model Development: The training dataset is used to build the error correlation prediction model, and then the parameters extracted from the model are used to build the error correlation predictor.

First, the prediction model identifies all diverged SC sets in both datasets, which was found to be 1200. Then, using only training datasets, a discrete probability distribution of CPU units for each diverged SC set is calculated. The probability value of each CPU unit becomes its probability score in that set. When a fault injected in a CPU unit leads to a divergence in a SC set, the histogram count of the unit is incremented for that set. The probability score of a CPU unit in a set is determined by the histogram count of the CPU unit divided by the total histogram count of the set. In a similar manner, a probability score is assigned to each error type in a set, and is calculated as the total histogram count of an error type in the set divided by the total histogram count of the set. We calculate the probability scores of all CPU units and error types for each of the 1200 diverged SC sets, and are ranked from high to low as shown in **Figure 10a**).

Once the training stage is complete, the prediction table can be populated with the predicted order of CPU units and error type prediction information as shown in **Figure 10b**). Each table entry has two portions: 1) error location prediction and 2) error type prediction information for a diverged SC set, so the table size is determined by the number of the diverged SC sets. The table is accessed by an 11-bit Prediction Table Address Register (PTAR) (shown in **Figure 6**). The CPU units in descending order of probability scores populate the location prediction portion of each entry, which varies from 3 bits (i.e., 1 unit) to 21 bits (i.e. all 7 units) depending on how many CPU units are predicted. The error type prediction portion of the entry is populated with one bit (i.e., 0 for soft and 1 for hard error). If the hard error probability score is higher than the soft one for a diverged SC set, the error type prediction bit is set to 1, otherwise it is reset.

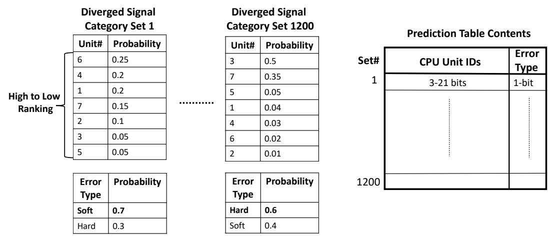
3) Prediction Models: We develop two static prediction models as shown in **Figure 9b and c**. The prediction models

[5]Herein, we assume the error coverage of STLs is 100%.



(a) Baseline Models **(b)** Location Only Prediction Model **(c)** Location and Type Prediction Model

Fig. 9. Comparison of the baseline models (a), the location-only prediction model (b), and the location and type combined prediction model (c)


High to Low Ranking
....

Diverged Signal Category Set 1

Unit#	Probability
6	0.25
4	0.2
1	0.2
7	0.15
2	0.1
3	0.05
5	0.05

Error Type	Probability
Soft	**0.7**
Hard	0.3

Diverged Signal Category Set 1200

Unit#	Probability
3	0.5
7	0.35
5	0.05
1	0.04
4	0.03
6	0.02
2	0.01

Error Type	Probability
Hard	**0.6**
Soft	0.4

Prediction Table Contents

Set#	CPU Unit IDs	Error Type
1	3-21 bits	1-bit
	⋮	⋮
1200		

(a) **(b)**

Fig. 10. (a) Location and type probability scores of each diverged SC set during the training stage; (b) The prediction table contents are populated with the CPU unit IDs and 1-bit error type information after training.

are static in the sense that the contents of the predictor table do not change during the lifetime of the lockstep CPUs. A static predictor is sufficient because errors are rare events unlike other predicted events (e.g., branch, cache miss, data value).

Figure 9b shows the location-only prediction model where only the order of CPU units is predicted, i.e., no error type prediction. The flow is exactly the same as the baseline models except for the order of CPU units. Instead of a static or random order, the SBIST starts the diagnosis from the most likely CPU unit from which the fault may have originated to the least likely one to reduce LERT.

Figure 9c shows the combined location and error type prediction model. The combined location and error type prediction model adds the error type prediction on top of the location-only prediction model to further reduce LERT. The extra reduction in LERT occurs when soft errors are correctly predicted in which case the SBIST process can be avoided.

If the error type is predicted to be a hard error, the SBIST process is started. A misprediction occurs if the SBIST does not find any hard error, and therefore the error must have been soft. In this case, a soft error handling process is started by resetting the CPUs and restart the application. On the other hand, if the error is predicted to be soft, the soft error handling process resets the CPUs and restarts the application. In this case, neither correct prediction nor misprediction can be verified accurately. This is because the error would disappear after the reset & restart if it was indeed a soft error. However, if another error occurs close proximity in time[6], we cannot accurately deduce that the previous error was mispredicted and should have been a hard error because the new error may even lead to a signal signature different from the signal signature of the previous error. Our strategy is very straightforward for this case. If another error occurs after a previous error that is predicted to be soft, it will always be taken as a hard error ignoring the error type prediction. Thus, this triggers the SBIST to run the STLs in the predicted order until the faulty CPU unit is found. The LERT of the combined prediction model in the presence of mispredictions is never greater than the LERT of the baseline model in **Figure 9a**, and therefore, safety is never comprised.

Three components factor into the LERT in the prediction models: 1) The number of cycles to access the prediction table, 2) The number of cycles to run the predicted order of STLs until a hard error found or run-to-completion (i.e., no hard error found), and 3) The number of cycles spent in restarting the benchmark in the case of soft errors.

TABLE II
VARIOUS LATENCIES USED IN THE MODELS

Name	**Latency in cycles**
Prediction Table Access Time	2 (on-chip memory) 100 (off-chip memory)
STL Latency Range	[Min, Mean, Max] = [25k, 170k, 700k]
Restart Latency Range	[Min, Mean, Max] = [2k, 10k, 36k]

V. Experimental Results

The baseline and error correlation prediction models are evaluated using the test dataset to quantify the LERTs and compared. We present an average LERT per error as a performance metric. The following abbreviations of the models are used in the experimental results: 1) Baseline random order of STLs or *base-random*, 2) Baseline static order of STLs sorted in ascending order of latencies or *base-ascending*, 3) Baseline static order is the ordering of the units based on their error manifestation rates or *base-manifest*, 4) Location-only prediction model or *pred-location-only*, and 5) Combined location and error type prediction model or *pred-comb*.

[6]e.g. hundreds, thousands or even millions of cycles after the restart, depending on the application

The latencies used in all models are shown in **Table II**. The two prediction models predict the order of all 7 CPU units and keeps the prediction table on-chip with a 2-cycle access time. The last two rows in the table show the STL and Restart latencies that come from measurements but present only their range in terms of min, mean and max. An STL latency of a CPU unit in *Cortex-R5* denotes the time to execute its test library written in machine instructions, and Restart latencies are the actual execution times of the *EEMBC AutoBench*.

A. Performance Comparison

Figure 11 compares the performance of all models. The best baseline model seems to be *base-ascending* followed by *base-manifest*. Randomizing the order of CPU units at every detected error does not provide much benefit as *base-random* gives the worst performance. It is not surprising that *base-ascending* gives a decent performance because the STL latencies of the CPU units are non-uniform, testing them in ascending order diagnoses the CPU units with short latencies first before testing the long latency ones. The first red number over each bar in the figure shows the average number of tested CPU units until the faulty one is found. Although this number for *base-random* is smaller than *base-manifest*, *base-random* tests the units with long latency STLs before finding the faulty unit because of its random nature. The exact average LERT in terms of cycles is also shown over each bar in parentheses. *base-random* spends twice the time of *base-manifest* on average per error to find the faulty unit even though *base-manifest* finds it by testing more units.

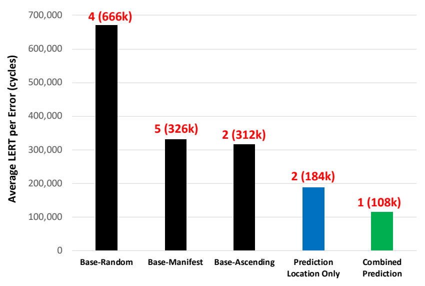


Fig. 11. Performance comparison of all models for 7 CPU units - The numbers in red over the bars represent the average number of tested units before finding the faulty one and the exact average LERT in terms of cycles in parentheses.

pred-location-only finds the faulty unit by testing fewer units and spending much less time than *base-manifest*, and therefore reduces the LERT by 43%. Although *pred-location-only* finds the faulty unit by testing the same number of units as *base-ascending*, it spends much less time to find it. Thus, it reduces the LERT by 40% relative to *base-ascending*.

TABLE III
ERROR TYPE PREDICTION ACCURACY FOR *pred-comb*

Error Type	**Prediction Accuracy**
Soft	86%
Hard	49%
Overall	**67%**

On top of the location predictor, *pred-comb* also uses the error type prediction to avoid unnecessary trigger of the SBIST process by predicting soft errors. **Table III** shows the error type predictor accuracies of *pred-comb*. While 86% of soft errors can be predicted correctly, this is only 49% for hard errors. Overall, the error type prediction accuracy is 67%. We observe that the error type predictor reduces the unnecessary SBIST invocations by 43% because of correctly predicted soft errors. Thus, this reduces the average number of tested CPU units per error down to 1 unit. Overall, *pred-comb* provides speedups of 65%, 64% and 39% relative to *base-manifest*, *base-ascending* and *pred-location-only*, respectively.

Another interesting observation is that the average LERT per error is in the range of 100k-to-670k cycles depending on the model. This is 80-to-515 times greater than the average lockstep error detection time per error, which was found to be 1300 cycles in *Section IV-B*. This shows that the error reaction time is the most critical parameter in a safety-critical system.

B. Analysis of Keeping Prediction Table On/Off-chip

The prediction models in **Figure 11** keep the prediction table in an on-chip memory with a 2-cycle access latency. The size of the prediction table is about 3.2KB to hold 1201 entries each of which has 22 bits (i.e. 21-bit for locations and 1-bit for error type). Although small, 3.2KB still steals away the precious on-chip memory area from the running applications.

Because errors are rare events and predictions are static, the prediction table can be kept off-chip in DRAM, and is only accessed when an error is detected. Thus, we also compare the average LERT per error of *pred-location-only* and *pred-comb* to measure their sensitivities of keeping the prediction table in an on-chip or off-chip memory. The access time to the prediction table in an off-chip memory is 100 cycles as given in **Table II**. We observe almost no performance overhead of keeping the table off-chip. The overhead is only 0.05% relative to the model keeping the table on chip for both *pred-location-only* and *pred-comb*. From this on, we will keep the prediction table in an off-chip memory.

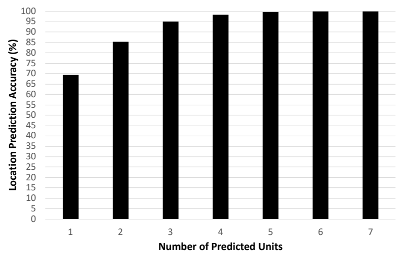


Fig. 12. Location prediction accuracy for *pred-comb* with the varying predicted number of units

C. Predicting Fewer CPU Units

The average number of tested CPU units for the prediction models in **Figure 11** varies from 1 to 2, i.e., the SBIST finds the faulty unit after testing 1-2 units on average. This implies that the predictor may not need to predict the order of all units in order to get a good prediction accuracy.

Figure 12 and **Figure 13** show the location prediction accuracy and average LERT per error trends with the varying

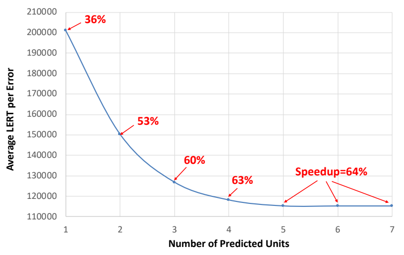


Fig. 13. Average LERT per error for *pred-comb* with the varying predicted number of units

predicted number of units for *pred-comb*. When *pred-comb* predicts fewer number of units, say 3 units, the top 3 units with highest probability scores are stored in the prediction table. If the SBIST cannot find the faulty unit among the 3 predicted units, the remaining units will be tested in a random order of STL latencies similar to *base-random*. We select the random ordering of the remaining units instead of *base-ascending* or *base-manifest* so as not to give unfair advantage to the predictors with a low number of predicted units.

The first bar in **Figure 12** denotes *pred-comb* predicting a single unit, and the second bar is *pred-comb* predicting two units and so on. The last bar is *pred-comb* predicting the order of all units. The location prediction accuracy is defined as the probability of finding the faulty unit in the list of predicted units. The accuracy of predicting 1 unit is about 70%, and it goes up to 85% with 2 units, and then up to 95% for 3 units. After 3 units, the accuracies reach 99%. The average LERT per error in **Figure 13** tracks the location prediction accuracies. It also shows the speedup in percentages relative to *base-ascending*. The sweet spot is to predict 3 to 4 units with 60% to 63% speedups after which predicting more units does not benefit any further. These results support our claim that the predictor does not need to predict the order of all units, which also reduces the prediction table size, e.g., 1.5-2KB storage space is needed to store the 3-to-4 predicted units (i.e., 9-12 bit) and 1-bit error type per entry.

D. *Finer Granularity Analysis in CPU Logical Organization*

We further divide the CPU into finer granular blocks to analyze the behaviour of the models when the number of the CPU logical units increases. We pick the *Data Processing Unit* (DPU), which is the most complex out of all units, and break it down into smaller 7 more units to have a CPU with a total of 13 different units. Also, we break down the DPU STL into its 7 constituents. We repeat the same methodology described in **Figure 7** to split training and test datasets for 13 CPU units.

Figure 14 shows the performance comparison of all models for a finer granular configuration of 13 units. We observe that *base-ascending*, *base-manifest* and the two prediction models improve in terms of average LERT in spite of increased number of units. The biggest drop in average LERT occurs in *base-ascending* by 62%. Although it tests 7 units on average to find the faulty one, the average time to find it is significantly reduced. This is because the breakdown of the DPU creates units with shorter STL latencies that are tested earlier by

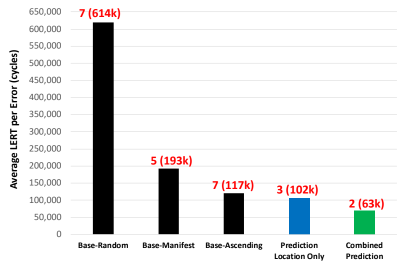


Fig. 14. Performance comparison of all models for 13 CPU units - The numbers in red represent the average number of tested units before finding the faulty one and the exact average LERT in terms of cycles in parentheses.

base-ascending than *base-manifest*. The performance of the prediction models also improves by 40-45% compared to the coarse granular CPU configuration. *pred-comb* provides speedups of 64%, 42% and 34% relative to *base-manifest*, *base-ascending* and *pred-location-only*, respectively. Overall, finer granularity in CPU logical organization improves the LERT both in baseline and prediction models.

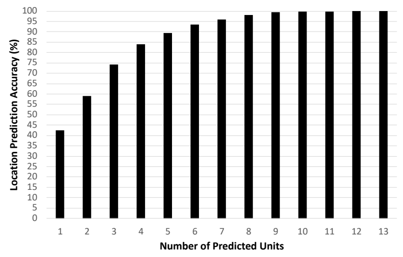


Fig. 15. Location prediction accuracy for *pred-comb* with the varying predicted number of units for finer granular configuration

Similar to the coarse granular configuration, the finer granular configuration may not need to predict the order of all units. It may be sufficient to predict fewer units and the remaining units can be tested in random order if the prediction misses. **Figure 15** and **Figure 16** show the location prediction accuracy and average LERT per error trends with the varying predicted number of units in finer granular configuration for *pred-comb*. **Figure 15** shows that the accuracy of predicting 1 unit drops to 42%, and it does not go up to 95% until 7 units. After 8 units, the accuracies do not change. **Figure 16** shows the average LERT per error with the speedup in percentages relative to *base-ascending* shown on top. The sweet spot is 7 to 8 unit prediction with 36% and 39% speedups after which predicting more units does not benefit any further. Finally, 4-5KB storage space is needed to store the 7-to-8 predicted units (i.e., 28-32 bit) and 1-bit error type per entry.

E. *Area and Power Overhead*

We quantify the area and power overhead of the predictor hardware. The extra hardware needed is the error correlation prediction logic shown in **Figure 6**, which consists of the 68-bit DSR, the address mapping logic and the 11-bit PTAR. We build a Verilog model of the error correlation prediction logic and synthesize it with *Synopsys Design Compiler* [21]. The synthesized design is implemented at the chip-level in 32nm commercial libraries by *Synopsys IC Compiler* [22] to quantify

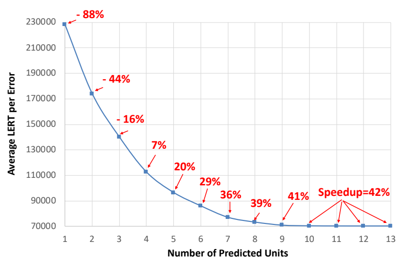


Fig. 16. Average LERT per error for *pred-comb* with the varying predicted number of units in finer granular configuration

its silicon area. We also use *Synopsys PrimeTime PX* [23] to calculate the worst-case total (dynamic + static) power.

TABLE IV
AREA AND POWER OVERHEAD RESULTS

Relative to	**Area Overhead**	**Power Overhead**
Dual-CPU *Cortex-R5* Lockstep	0.6%	1.8%
A Single *Cortex-R5* CPU	1.4%	4.2%

Table IV shows the area and power overheads of the predictor relative to a dual-CPU *Cortex-R5* lockstep processor (i.e., two locksteped CPUs + an error checker) and a single *Cortex-R5* CPU that are also implemented using the same implementation flow and technology libraries. The area and power overheads of the predictor are very low, 0.6% in area and 2% in power, relative to the dual-CPU *Cortex-R5* lockstep processor. The overheads are still low relative to the single *Cortex-R5* CPU, i.e., 1.4% in area and 4.2% in power.

VI. Related Work

Lockstepping gained popularity in 90s and early 2000s because of its excellent fault coverage property and simplicity in its implementation among high availability servers. A good example of system-level lockstepping is the *Stratus FtServer* [4], which is a high-availability server system solution that provides two replicated CPUs and main memories. Similar to *FtServer*, *HP NonStop* [5] is a high-availability server architecture that uses a sub-system level lockstepping with replicated microprocessor chips and off-chip caches. *IBM G5* microprocessor [24] is designed for fault-tolerant servers, and uses a lockstepped pipeline in which fetch and execution units are replicated and lockstepped. *Maxwell Space Computer (SCS750)* [25] is a fault-tolerant computer designed for space, and contains three replicated PowerPC microprocessor chips.

[26] presents a survey of online error detection and recovery in multi-core processors covering redundant execution, built-in-self-test (BIST), dynamic verification and anomaly detection. There is also a plethora of transient and permanent fault characterization studies on architectural and program behaviour of microprocessors [27] [28] [29] [30] and [31].

Redundant execution techniques such as Redundant Multi-threading (RMT) [32] [2] [33] [34] [35] [36] [37], software based redundant techniques [38] [39] and dynamic verification [40] [41] were proposed in the literature but they were not popular in safety-critical systems because they did not provide the maximum error coverage and real-time error checking.

Lockstepping became a popular technique in 2010s in the domain of safety-critical automotive electronic control units (ECUs) to provide the most stringent functional safety, e.g. *TI* safety-critical *Hercules* [6] chip uses an *Arm Cortex-R5* [18] dual-CPU lockstep architecture. Similarly, *Infineon* uses a dual CPU lockstep architecture in its *Aurix* safety-critical automotive SoC [7]. *NXP* [8] also offers a dual-core lock step feature in its *Qorivva MPC56XX* and *MPC57XX* families. The SPC56X series of microcontrollers from *STMicroelectronics* [9] comes with a dual-core lockstep configuration option. [42] exposes extra CPU internal state to the checker to detect faults earlier in the dual-core lockstep safety-critical systems.

Our work is different from anomaly/fault symptom detection/prediction studies within the processor [43] [44] [45] [46]. These techniques monitor certain events inside the processor such as data values, exceptions, cache misses, page faults, fatal traps etc. to detect and/or predict anomalies. On the other hand, our work is a post-error detection technique to reason about the attributes (i.e., type and correlated location) of an already detected error in a predictive manner.

VII. Discussion

Lockstep error correlation prediction can be performed dynamically where the prediction table entries can be updated with the error prediction history similar to the branch prediction. This implies that the prediction table must be kept in hardware, and store error history. However, errors are not frequent events like branches, so the accumulation of error history will take a longer time compared to the branch history, and may not be any more beneficial than static prediction.

The predictor hardware, though implemented in a dual-CPU lockstep processor, can scale to multi-CPU lockstep processors with no additional hardware cost because increasing number of lockstepped CPUs increases the hardware complexity of the lockstep error checker but not the predictor. Also, we have evaluated lockstep error correlation prediction in a specific CPU architecture (i.e., Arm) and CPU (i.e., Cortex-R5). However, the concept does not rely on the specifics of the ISA or microarchitecture, and is applicable to lockstep processors based on different architectures and their implementations.

VIII. Conclusion

This paper has presented a new phenomenon called error correlation prediction for lockstep processors in safety-critical systems, and proposed a simple predictor to exploit it to increase system availability. A prediction is made by analyzing the output signals of the CPUs when an error is detected by the lockstep error checker. Lockstep error correlation prediction helps the safety-critical system to identify the cause of the error, and puts it into a safe state faster. We have developed an evaluation framework to demonstrate the feasibility of the error correlation prediction using a dual-CPU lockstep *Arm Cortex-R5*. We have also built a simple static predictor and shown that system availability can be increased by 42-65%. The cost of the predictor is very low, less than 2% in area and power with respect to a dual-CPU lockstep processor.

References

[1] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 2008.

[2] S. Reinhardt and S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-2000)*, 2000.

[3] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," in *IEEE Micro*, 2003.

[4] Stratus, "FtServer Architecture," http://www1.stratus.com/kr/solutions/platforms/ftserver/ftserver-architecture/.

[5] A. Wood, R. Jardine, and W. Bartlett, "Data Integrity in HP NonStop Servers," in *2nd IEEE Workshop on Silicon Errors in Logic and Systems Effects (SELSE)*, 2006.

[6] Texas Instruments, "Hercules TMS570 Microcontrollers," http://www.ti.com/microcontrollers/hercules-safety-mcus/overview.html, 2014.

[7] Infineon, "AURIX 32-bit Microcontrollers for Automotive and Industrial Applications," in *www.infineon.com/aurix*, 2018.

[8] NXP, "Safety Manual for Qorivva MPC5643L," MPC5643LSM Rev. 2, 2013.

[9] STMicroelectronics, "SPC56EL70L3: 32-bit Power Architecture Microcontroller for Automotive SIL3/ASILD Chassis and Safety Applications," in *http://www.st.com/en/automotive-microcontrollers/spc56el70l3.html*, 2015.

[10] ISO, "ISO 26262-9:2011 Preview Road Vehicles Functional Safety," https://www.iso.org/standard/51365.html, 2011.

[11] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-Tolerant Platforms for Automotive Safety-Critical Applications," in *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES'03*, 2003.

[12] G.Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for Large Industrial Designs: Real Issues and Case Studies," in *IEEE International Test Conference (ITC)*, 1999.

[13] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra Low-Cost Defect Protection for Microprocessor Pipelines," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[14] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[15] M. Psarakis, D. Gizopoulos, E.Sanchez, and M. S. Reorda, "Microprocessor Software-Based Self-Testing," in *IEEE Design and Test of Computers*, 2010.

[16] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lockstep (TCLS) Arm Cortex-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *Proc. of the IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2016.

[17] F. Aherne, N. Thacker, and P. Rockett, "The Bhattacharyya Metric as an Absolute Similarity Measure for Frequency Coded Data," in *Kybernetika*, 1997.

[18] Arm, "Cortex-R5 and Cortex-R5F Technical Reference Manual," in *www.infocenter.arm.com*, 2011.

[19] Synopsys, "VCS," https://www.synopsys.com/verification/simulation/vcs.html.

[20] EEMBC, "EEMBC AutoBench," https://www.eembc.org/benchmark/automotive_sl.php.

[21] Synopsys, "Design Compiler," https://www.synopsys.com/implementation-and-signoff.html.

[22] ——, "IC Compiler," https://www.synopsys.com/implementation-and-signoff/physical-implementation/ic-compiler.html.

[23] ——, "PrimeTime PX," https://www.synopsys.com/support/training/signoff/primetimepx-fcd.html.

[24] L. Spainhower and T. A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," in *IBM Journal of Research and Development*, 1999.

[25] L. Longden, C. Thibodeau, R. Hillman, P. Layton, and M. Dowd, "Designing a Single Board Computer for Space using the Most Advanced Processor and Mitigation Technologies," in *White Paper - Maxwell Technologies*, 2002.

[26] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for Online Error Detection and Recovery in Multicore Processors," in *DATE'11*, 2011.

[27] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-performance Microprocessor," in *International Symposium on Microarchitecture (MICRO)*, 2003.

[28] G.Weining, Z.Kalbarczyk, R.K.Iyer, and Z.Yang, "Characterization of Linux Kernel Behavior under Errors," in *IEEE Transactions on Dependable and Secure Computing*, 2003.

[29] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the Effects of Transient Faults on a High-performance Processor Pipeline," in *International Conference on Dependable Systems and Networks*, 2004.

[30] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, "On the Correlation between Controller Faults and Instruction-level Errors in Modern Microprocessors," in *International Test Conference (ITC)*, 2008.

[31] X. Iturbe, B. Venu, and E. Ozer, "Soft Error Vulnerability Assessment of the Real-time Safety-related ARM Cortex-R5 CPU," in *IEEE Defect and Fault Tolerance in VLSI and Nanotechnology Symposium (DFT)*, 2016.

[32] E. Rotenberg, "R-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, 1999.

[33] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," in *Proc. of 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[34] M.Gomaa, C.Scarbrough, T.N.Vijaykumar, and I.Pomeranz, "Transient-fault Recovery for Chip Multiprocessors," in *Proc. of 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[35] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-effective Multicore Redundancy," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[36] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J.E.Smith, "Configurable Isolation: Building High Availability Systems with Commodity Multi-core Processors," in *Proc. of 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[37] C. LaFrieda, E. Ipek, J. F. Martinez, and R.Manohar, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor," in *International Conference on Dependable Systems and Networks (DSN)*, 2007.

[38] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Superscalar Processors," in *IEEE Trans. on Reliability*, 2002.

[39] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Intl. Symp. on Code Generation and Optimization (CGO)*, 2005.

[40] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1999.

[41] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *Proceedings of the 40th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2007.

[42] C. Hernandez and J. Abella, "Timely Error Detection for Effective Recovery in Light-Lockstep Automotive Systems," in *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 2015.

[43] N. J. Wang and S. J. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," in *2006*, IEEE Trans. on Dependable and Secure Computing.

[44] S. Narayanasamy, A. K. Coskun, and B. Calder, "Transient Fault Prediction Based on Anomalies in Processor Events," in *DATE'07*, 2007.

[45] P.Racunas, K.Constantinides, S.Manne, and S.S.Mukherjee, "Perturbation-based Fault Screening," in *Proc. of 3rd International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[46] M.-L. Li, P. Ramachandran, Swarup, K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.