# Avoiding Core's DUE & SDC via Acoustic Wave Detectors and Tailored Error Containment and Recovery

Gaurang Upasani[†] Xavier Vera[♭] Antonio González[†♭]
[†] Dept. d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona
[♭] Intel Barcelona Research Center, Intel Labs, Barcelona
gaurang@ac.upc.edu {xavier.vera,antonio.gonzalez}@intel.com

## Abstract

*The trend of downsizing transistors and operating voltage scaling has made the processor chip more sensitive against radiation phenomena making soft errors an important challenge. New reliability techniques for handling soft errors in the logic and memories that allow meeting the desired failures-in-time (FIT) target are key to keep harnessing the benefits of Moore's law. The failure to scale the soft error rate caused by particle strikes, may soon limit the total number of cores that one may have running at the same time.*

*This paper proposes a light-weight and scalable architecture to eliminate silent data corruption errors (SDC) and detected unrecoverable errors (DUE) of a core. The architecture uses acoustic wave detectors for error detection. We propose to recover by confining the errors in the cache hierarchy, allowing us to deal with the relatively long detection latencies. Our results show that the proposed mechanism protects the whole core (logic, latches and memory arrays) incurring performance overhead as low as 0.60%.*

## 1. Introduction

The large amount of transistors have fueled the growth of big, in-memory data applications that were not feasible several years ago. Moreover, because of the lower voltages and the shrinking feature size, the vulnerability of the current and future processors towards transient errors caused by particle strikes is expected to increase rapidly [6].

With increased core counts per chip and larger memory arrays, the total failure-in-time (FIT) per chip (or package) increases. Decrease in the supply voltage to increase number of cores results in even higher FIT rate [33]. Most of the large memory blocks are already protected with error correcting codes against both hard and soft errors. But logic elements and small arrays, which are the main contributors to the majority of the soft error FIT budget [49], are largely unprotected due to the huge cost of using hardened latches or codes. Hardened latches and parity/error correction codes have 20-30% overhead in terms of extra logic dedicated for error detection and recovery [15, 44]. Hence, meeting the desired FIT budget for current and future multicore systems is a major challenge.

Architecturally, soft error detection and correction mechanisms create two categories of errors: silent data corruption (SDC) and detected unrecoverable errors (DUE). Chip designers have fixed SDC- and DUE-FIT budgets for different application segments similar to power or performance budgets. To fulfill the SDC budget, designers may deploy simple error detection schemes (i.e., parity). However adding parity converts the SDC-FIT into DUE-FIT, adding pressure to the DUE budget. To reduce the DUE, once the error has been detected the system should be able to restore the normal state of operation. For instance, error correction codes (i.e., single error correction) are used to provide recovery in memory.

Instead of relying on redundancy to detect errors (e.g., re-execution techniques or information redundancy), a new direction that is growing in interest among researchers is to detect the actual particle strike rather than its consequence [16, 23, 36, 47].

This paper proposes a light-weight architectural framework that can completely eliminate SDC- & DUE-FIT of a core. The architecture uses acoustic wave detectors for dynamic particle strike detection. Moreover, the architecture does not allow errors to escape to user (i.e., updating main memory or i/o devices) before detection, eliminating SDC. Eliminating DUE of core is more involved and our proposal relies on a novel and cantilever-specific checkpointing for recovery.

Another benefit of employing acoustic wave detectors is that since detectors trigger timely, latent particle strikes do not accumulate and hence reduces the odds of temporal multiple bit errors.

In summary, the principal contributions of this paper are:

- We propose an architectural framework to completely eliminate the SDC- and DUE-FIT related to soft errors in a core. It relies on acoustic wave detectors as a *unified error detection* mechanism to detect errors in both memory and current unprotected logic components in a processor core.
- The Proposed recovery solution is novel and specific to acoustic wave detectors. It relies on a light-weight checkpointing mechanism. It requires just one counter for entire cache to validate checkpoints. We discuss different design parameters and evaluate cost of checkpointing & recovery.
- The Proposed architecture, besides providing a highly reliable core, is able to recover a significant part of the overheads associated with current reliability techniques by potentially eliminating error codes and radiation hardened latches for soft errors. It also significantly reduces the de-

sign complexity compared to other mainstream reliability solutions.

- We evaluate the impact of error detection latency on the cost and complexity of the required recovery technique. We present different trade-offs related with complexity of detectors deployment, detection latency and complexity of recovery mechanism.

The rest of the paper is structured as follows: Section 2 reviews the physics behind the soft errors and how we can detect particle strikes. Section 3 gives an overview of our proposed architecture while explaining the relevance of detection latency for better error containment and recovery. Section 4 discusses the implementation of the architecture for a baseline core. Section 5 presents a discussion about how the proposed architecture scales to multicore architectures. We carry out a detailed evaluation of our proposed architecture in terms of coverage and overheads in Section 6. Section 7 discusses related work. Finally, a summary of our main conclusions is presented in Section 8.

## 2. Background: Physics of Particle Strikes and their Detection

Alpha and neutron particles can cause soft errors in semiconductor devices. Upon a collision of a particle with a silicon nucleus, the ionization process creates a large number of electron-hole pairs, which subsequently produce phonons and photons.

*Generation of phonons and photons indicate that a particle strike results into a shockwave of sound, a flash of light or a small amount of heat for a very small period of time. Therefore, we may try to detect particle strikes by detecting the sound, light or heat.*

Phonons and photons gradually result into cascade of carriers resulting into either drift current, diffuse current or a voltage glitch. *Therefore, we can detect particle strikes also by detecting current or voltage glitches.*

Several detector-based techniques have been studied in the past. These detectors work by detecting particle strikes via detection of voltage or current glitches [16, 23, 36] or via detection of the sound [47]. Comparing the techniques based upon the area and power overheads, false alarms, type of particles it can detect, fault coverage in terms of what kind of structures it can protect (memory or logic blocks), and the intrusiveness of the design, we decided to use cantilever like structures [18, 19] as an acoustic wave detector to detect particle strikes through the sound they generate as proposed in [47].

### 2.1. Acoustic Wave Detectors: Detection of Particle Strikes

A typical acoustic wave detector is a cantilever beam like structure. The particle strike is detected by the change in the capacitance of the gap between the cantilever and the ground pad of the detector structure. The cantilevers occupy an area of one square micron [20], which is roughly the area of one memory bit (i.e., a typical 6T SRAM cell). It has a detection range of 5mm. It takes 500ns (i.e., 1000 cycles at 2 GHz) to detect a strike that is 5mm away from the detector [47]. It is worth mentioning that detectors do not detect the exact location of strikes.

Acoustic detectors adopt silicon based fabrication that is similar to IC fabrication technology. Placement of detectors on active silicon can be done without much complications, making it feasible for detectors to be integrated with the rest of the circuitry [19]. Cantilever based detectors have been developed and used extensively to study bio-interactions at atomic level [26].

**Detecting the right particle:** Recent studies [20, 47] show that the cantilever structure can be calibrated in such a way that it only detects particle strikes that are capable of generating single even transient (SET) in logic or a single event upset (SEU) in memory. The same detectors can be used for memories as well as logic components [6, 12]. Current studies show that only the particles that have energies larger than 10MeV when they hit can cause soft-errors [6, 21].

**Impact of wrong calibration:** Failing to properly calibrate the detectors would result into false positives (i.e., detectors' trigger for the particles that do not carry enough charge to create a soft error). Considering that the flux of energetic particles at sea level is approximately 25 neutrons/($cm^2$-hr), *an improbable scenario of detectors triggering for every harmless particle strike would imply detecting 1 false positive every 1.3 minutes for a modern general-purpose multi-core processor.*

## 3. "SDC & DUE 0" Architecture

The main objective of the proposed architecture is to achieve 0 SDC- and DUE-FIT per core. SDC occurs when errors escape and become visible to the user. DUE occurs when error is detected but we do not have any error recovery mechanism. Error correction is handled by either moving the system to a state that does not contain the error (e.g., using checkpointing) or by an on-the-fly error correction method. Next, we will see how error detection latency plays an important role in deciding the overall cost of SDC and DUE handling solutions.

### 3.1. Effect of Detection Latency on SDC & DUE

Acoustic wave detectors detect all soft errors due to alpha and neutron strikes. However, not only the detection of the error but how soon the error is detected is also very important. Detection latency defines the degree of error containment. Depending on the detection latency, errors can be contained at various granularities in a processor (i.e., within pipeline or caches etc.). Efficient error containment is essential for avoiding SDC (e.g., error is visible to the user before its detection) and it also has an impact on the recovery process.

| Detection Mechanism | | Post Consumption Detection Latency | Containment Cost | Size of Checkpoint | Protected Structure | Detection Coverage | Area Overhead | Performance Overhead |
|---|---|---|---|---|---|---|---|---|
| **Redundant execution** | Lockstep [45, 46], DMR [1], DCC [27] | Cycle-by-cycle detection | Low | Small | Core | 100% | 100% [1, 45, 46], 1% [27] | $>1.5-2\times$ |
| | RMT [17], CRT [34], AR-SMT [43] | Hundreds of cycles & unbounded | High | Large | Core | $\sim$100%$^{\dagger}$ | Low | $>2\times$ |
| **Instruction duplication** | EDDI [37], SWIFT [42], CRAFT [41] | Low & unbounded | Low | Small | Core & Main memory [37] | <100%$^{*}$ [42] | Low | >150% [37] |
| **Symptom checks** | Error Codes [33], Hardened latches$^{\dagger\dagger}$ [44] | 0 cycles & bounded | Low | – | Main memory [33], Logic [44] | 100% [33] | 12.5% [33], $\sim$55% [44] | Low |
| | BIST [10], Bulletproof [11] | Periodic & bounded | High | Large | Core | Only harderrors 90% [11] | 5%-6% | 5%-25%$^{\star}$ |
| | SWAT [28], Shoestring [14], Restore [50], Perturbation [38] | Millions of cycles & unbounded | High | Large | Core | <100%$^{**}$ [14] | Low | 5-16% |
| **Monitoring invariants** | DIVA [3], Argus [31] | Low & bounded | Low | Small | Core Backend [3], Core [31] | 100% [3], <100% [31] | 6% [3], 17% [31] | 5-15% |
| **Sensor based** | Acoustic detectors [47] | 100 cycles$^{\mp}$ (configurable) & bounded | Low | – | Cache | 100% | <1% | Low |
| | (V-I) detectors [16, 23, 36] | 3-4 cycles & bounded | Low | – | Main memory & Logic | – | High | Low |
| **Proposed Architecture** | | 30-100 cycles$^{\mp}$ (configurable) & bounded | Low | Small | Core | 100% detection & recovery | <1% | <0.60% |

**Table 1: Comparison of different error detection schemes ($^{\dagger}$ vulnerability holes in LSQ logic (i.e., MOB logic), $^{*}$ cannot detect errors in stores, $^{\dagger\dagger}$ does not detect but prevents error, $^{\star}$ only for simple in-order cores, $^{**}$ cannot detect if fault does not manifest a symptom, $^{\mp}$ latency from actual strike instance)**

Table 1 reviews the detection latencies for different error detection techniques once the error is consumed. Bounded latency means the error is detected within a fixed number of cycles, that is known a-priori or can be set by the designer (e.g., periodic BIST). Longer detection latency enforces the error containment to be done at a higher degree of abstraction in a processor, and results in more complex hardware and/or software checkpointing/recovery mechanisms. Excessively long detection latencies may not be even recoverable. Long detection latencies can also prevent the fault diagnosis due to weak correlation between the fault and its symptoms or due to the limited on-chip tracing storage (i.e., log sizes [9]).

Error detection mechanisms with low detection latency provide the best tradeoff. Therefore, to achieve SDC-& DUE 0 at a minimum cost we next explore error containment and recovery for minimum latency (i.e., containment before the error updates architectural state).

### 3.2. Achieving SDC-& DUE 0 per Core

In order to achieve 0 SDC, we can equip a processor core with acoustic wave detectors so it detects all particle strikes that may cause an error. To have DUE 0 per core, the architecture must be able to recover from all the errors and restore correct processor state; this includes architectural register file, register allocation table (RAT), program counter (PC) etc.

Previous work [47] proposed using acoustic wave detectors in combination with error correction codes to detect and locate errors in memories. In this section, we extend it and assess its detection latency and capabilities to address challenges in achieving SDC-& DUE 0 for an entire core.

**Achieving SDC 0 per core:** The first option that we explore is protecting the core for the minimum error detection latency. It requires that the error is captured before the wrong value is committed.

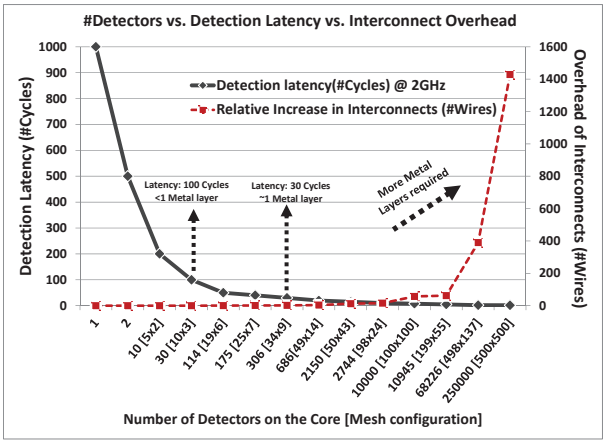



**Figure 1: Number of detectors vs. detection latency at 2GHz**

Given the dimensions of current core designs, a single detector would suffice to detect all the particle strikes. As mentioned in Section 2.1, using just 1 detector implies a worst-case detection latency of 1000 cycles at 2 GHz, which may give time for the erroneous instructions to commit before being detected.

In order to reduce the detection latency, more detectors can be deployed. Multiple detectors can be deployed in different topologies, e.g., a mesh formation. Figure 1 shows the detection latency and complexity for various mesh configurations. Complexity in terms of increased number of wires is calculated. It is clear that the detection latency varies exponentially with number of detectors. Also complexity increases with number of detectors.

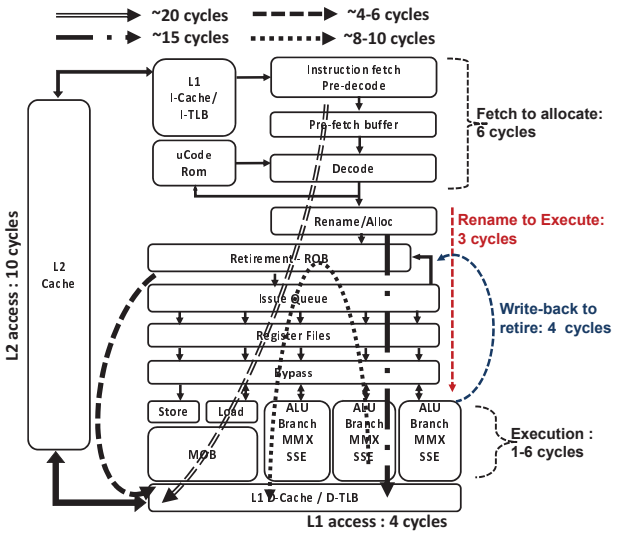



**Figure 2: Pipeline of a state of the art processor and the latency of stages**

According to Figure 1, we will need need more than 68,000 detectors to guarantee that no instruction will be committed before it is checked for errors (error detection latency of 1 cycle).

**Achieving DUE 0 per core:** With 68K detectors we contain the errors before they are committed. If the strike happened in speculative state, a nuke and retry would suffice to recover. However, if the strike was in the architectural state, recovery would be somewhat more involved. One option is to use error correcting codes. However, majority of the structures that hold the architectural state (i.e., architectural register file) do not have error correcting codes. Therefore, we opt to take checkpoints periodically (that include shadow copies of the architectural state).

In a nutshell, for having an SDC- & DUE 0 core we will need 68K detectors. This implies an area overhead equivalent to having 68K bits of SRAM (∼7KB cache). Moreover, as shown in Figure 1 the interconnects to the microcontroller from 68K detectors increases the complexity and require more than 5 metal layers which pose significant challenges in place and route [4, 35].

Next, we will explore an optimized architecture that reduces the number of detectors without compromising the reliability coverage.

### 3.3. Divide and Conquer for SDC- & DUE 0

We made the observation that errors in different stages of pipeline take different time until they propagate outside the containment area (i.e., before they commit). To reduce the number of detectors for containment before an erroneous instruction is committed, we study the pipeline structures and analyze the time each instruction spends in traversing through the pipeline. We collect the latency requirements for all the structures to provide coverage to all instructions. This gives us an insight of the required detection latency for each structure in the core.

| Pipeline stage | #Detectors |
|---|---|
| Fetch + Decode (including I-Cache, D-Cache, TLBs) | 1787 |
| Rename + Schedule | 170 |
| Execute | 461 |
| Writeback + Commit | 139 |
| **Total** | **2561** |

**Table 2: Required number of detectors for containment in core**

Figure 2 shows the pipeline of our base core running at 2GHz. It shows the latency for the different stages of the pipeline up to the commit stage. We identified four different paths with different latency: (i) fetch/decode to commit takes 20 cycles, (ii) rename/scheduler to commit takes 15 cycles, (iii) execute to commit takes 8-10 cycles, and (iv) write-back/retire to commit takes 4-6 cycles.

From this initial observation, we identified that some data-flow paths are more critical (i.e., writeback to commit) and will need stricter detection latency requirements for error containment. So, instead of protecting all the functional units in the pipeline for a common detection latency we propose to put detectors for individual functional units. By protecting each functional unit for its *allowable* detection latency we can reduce the number of detectors and still achieve 0 SDC. In order to have 0 DUE we keep low cost shadow copies of architecture state as described in Section 3.2.

Up to this point, we contain errors before they commit and as shown in Table 2, it requires 2.5K detectors for functional blocks in the pipeline for their *allowable* detection latency requirements.

**Overhead:** 2.5K detectors incur an area overhead equivalent to 2.5K bit SRAM. Complexity for accommodating 2500 (low latency) interconnects occupy $\sim 4$ metal layers causing an unacceptable area overhead as shown in Figure 1. Moreover, control circuit for handling 2500 logic inputs is complex and requires a tree of logic-OR gates (∼22K extra CMOS cells).

### 3.4. Containment in Core: Recap

We realized that achieving DUE 0 by recovering within the core demands 68K detectors. To reduce the area overhead, we explore a modification that protects each pipeline stage based on its *allowable* detection latency. By relaxing error detection latency requirement, the number of detectors for efficient error containment goes down to 2.5K. However, the resulting design is complex and the area overhead of 2.5K interconnects is still unacceptable.

Hence, we propose to extend the error containment area beyond the commit stage to the cache hierarchy.

### 3.5. Proposed Architecture

Including caches in the error containment boundary implies that we can further relax the detection latency requirement

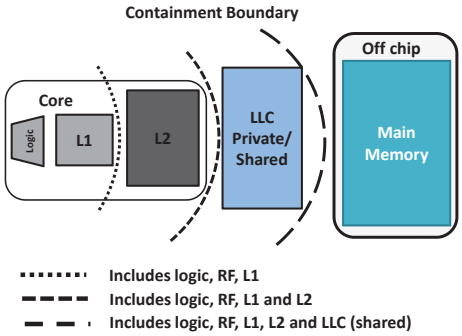



**Figure 3: Error Containment Architecture**

which in turn reduces the required number of detectors. According to Figure 1 relaxing detection latency constraint by 10× (i.e., from 10 cycles to 100 cycles) reduces the number of required detectors by 90× and this reflects in 100× decrease in complexity and interconnects overhead. We believe that a good trade-off between detection latency, area overhead and complexity lies within 30-300 detectors, which means 30-100 cycles latency at 2GHz.

Our proposed architecture to provide DUE 0 core consists of the following steps:

**Error detection:** We use acoustic wave detectors to detect particle strikes in the core. We opt for a simple configuration with a number of detectors in the range of 30-300, which provides a detection latency in the range of 30-100 cycles running at 2GHz.

**Data error containment:** We choose our containment area to be the cache hierarchy. Figure 3 shows the different error containment boundaries for an architecture with a single core and three levels of cache. Notice that the boundary of the containment area can be configured to be at any cache level. We assume that the caches themselves are protected by some mechanism. A datum will be correct once it has spent more time than the worst-case error detection latency in the cache (this way, we guarantee that the datum was produced correctly). In order to guarantee containment, we do not allow any data to go out of containment region before making sure that data is error free.

**Data checkpointing:** Containment boundary helps to decide the checkpoint boundary. By definition, containment boundary lies within the checkpoint boundary. Otherwise, there is a possibility of corrupting the checkpoints. Every *conceptual* checkpoint will consist of the architectural state (e.g., register file, program counter, etc) and the memory data. Process of checkpointing would include saving register values and flushing cache block values within the checkpoint boundaries that have been modified since the last checkpoint.

**Data recovery:** Upon an error, data recovery consists of invalidating all temporal data within the checkpoint boundary, and resume the execution from the latest checkpoint. Notice that this checkpoint will consume the data from outside the checkpoint area (and therefore, the containment area), that is guaranteed to be correct.

Next, we will discuss implementation aspects of proposed architecture.

## 4. Implementation

Without loss of generality, we will use as a running example a system comprising a single core and two levels of cache, with LLC as the boundary of the containment and checkpoint area. For instance, a system with three levels of cache, and L3 as the boundary would be implemented exactly the same way, with L3 acting as our described LLC, and L1 & L2 collectively acting as our L1 cache. For the rest of the text, we assume that worst-case detection latency for the acoustic wave detectors is *ErrorDetectionLatency*.

### 4.1. Error Containment Mechanism

The purpose of the containment mechanism is to make sure that only error free data goes beyond the containment area. In our implementation, where we use acoustic wave detectors as error detection mechanism, only data that has spent more than *ErrorDetectionLatency* cycles within the containment boundary has been produced in the right way.

We propose to add one counter for entire cache within the containment area. The counter monitors the modified data in the cache and keeps track of the correctness by counting *ErrorDetectionLatency* cycles.

Initially, the counter is set to force unknown state (i.e., counter = "X") as there is no modified data in the cache. We reset the counter (i.e., counter = "0") once any cache line in the given cache is modified following a *write* operation. Until counter finishes counting *ErrorDetectionLatency* cycles, the cache is in quarantine as we are not sure if it contains erroneous data or correct data. Once counter reaches *ErrorDetectionLatency* cycles the cache is said to be verified. *Read* operations do not affect the state of counter.

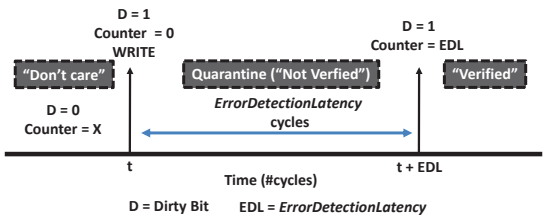



**Figure 4: Timeline of the events in cache**

**Example:** Figure 4 shows the basic events for a cache line in the cache. Before the *write* operation the line is clean (i.e., *dirty bit*, D = "0") and counter = "X". Following a *write* operation at time *t*, D = "1" and counter is reset (i.e., counter = "0"). After *ErrorDetectionLatency* cycles the entire cache is verified. Now, we will discuss how the normal cache operation is carried out in the proposed architecture. For that purpose we will be using Figure 5, which shows different events that may happen to cache lines within the cache of containment area.

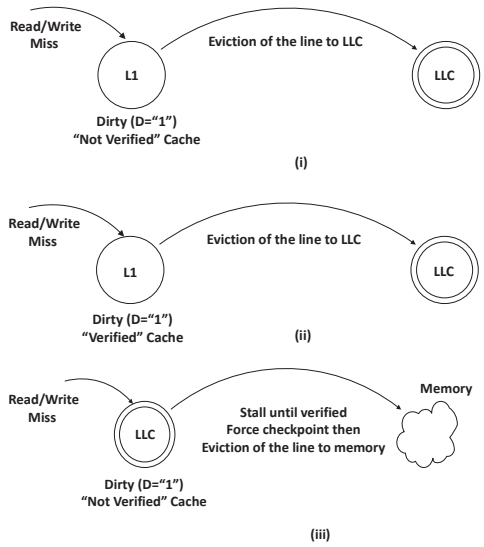



**Figure 5: Containment for evictions caused by read and write operations**

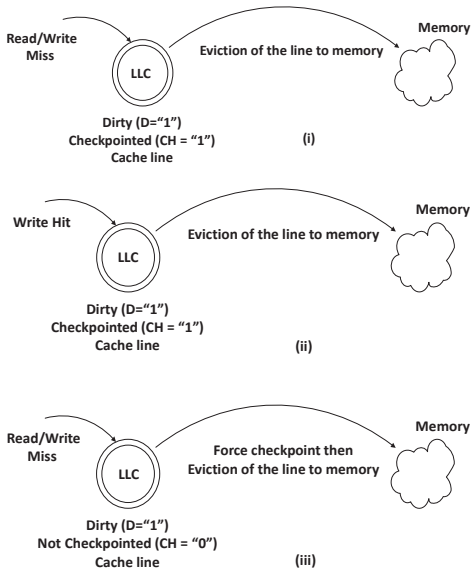



**Figure 6: Checkpointing for evictions caused by read and write operations**

**4.1.1. Dealing with Verified Cache.** Figure 5(ii) shows the case of evictions of dirty cache lines in a verified cache, which are allowed to make forward progress and leave the containment boundary. Later, they can be part of the new checkpoint.

**4.1.2. Dealing with Not-Verified Cache.** Not-verified cache is in quarantine. *Read* operations from the core do not alter the state of counter, since potentially erroneous data will not leave the containment area.

**Evictions from L1 cache:** Figure 5(i) shows the actions to be taken upon an eviction of a dirty cache line when the L1 cache is not verified. First, we will evict the cache line to LLC. The counter could be inherited or pessimistically reset at LLC. Alternatively, we could stall until the L1 cache is verified before evicting modified cache line to LLC. In this work, we do not stall the processor as it can have negative impact on the performance.

**Evictions from LLC cache:** Evictions of dirty cache lines from LLC (i.e., containment boundary) when LLC is not verified are not allowed as is the case of Figure 5(iii). In such event we will stall until LLC is verified.

In Section 6 we analyze all the cases discussed above for their impact on performance and observe the tradeoff between error containment area and cost of containment using real life workloads.

### 4.2. Creating Checkpoints

The checkpointing process should include:

- Copying the architectural state.
- Saving the program counter.
- Waiting for all caches to be verified.
- Writing back all dirty data in lower (verified) caches to main memory.

For checkpointing the architectural state we suggest to use shadow structures as proposed in [13]. The copy of the program counter is stored in a special register. All these structures are assumed to have error recovery capabilities (e.g., ECC).

We anticipate that writing all dirty data present in all caches to memory may be expensive. Similar to previous works [24, 30], we adopt an *incremental checkpointing* where only dirty lines from the caches closest to the core (L1 in our running example) are written back to the cache in the boundary of the checkpoint area (LLC in our running example). Dirty lines in the LLC are now part of the checkpoint. In this configuration, the *data part* of the checkpoint will be split between the LLC and main memory.

In order to implement such optimization, we add a *checkpoint bit (CH)* in every cache line of the cache in the checkpoint boundary (i.e., every cache line of LLC). Initially the *checkpoint bit* is set to "0", which means that the line is not part of the checkpoint.

**Periodicity:** In this proposal we take periodic incremental checkpoints. The frequency of checkpoints and its implications are further discussed in Section 6. Next, we discuss how we handle the events to cache lines of cache in the checkpoint boundary (LLC in our running example).

Figure 6(i) shows the case of a dirty cache line in LLC that is part of the checkpoint. In that case we allow any eviction since the cache lines are already part of the checkpoint and do not affect the recovery of the correct architectural state. Moreover, write hits to a cache line that is part of the checkpoint will result into an eviction as the cache line cannot be modified without having a safe copy in main memory. Therefore, we evict the cache line to memory, reset the *checkpoint bit* and then serve the write as shown in Figure 6(ii). Finally, in

the case of Figure 6(iii) an eviction of a dirty line that is not part of a checkpoint in LLC will force a checkpoint before being evicted to memory.

**Waiting for verified data:** It is important to note that in order to take a checkpoint, we need to stall until the caches (L1 and LLC) are verified. Once they are verified, we can start writing back all the cache lines to the checkpoint boundary.

**4.2.1. Validating the Checkpoint.** Checkpointing process is not free from suffering particle strikes. Therefore, we need to pay careful attention to guarantee that the checkpoint is valid.

To avoid the creation of corrupted checkpoints, we also add one global counter *CheckpointValid* to LLC (i.e., cache in the checkpoint boundary). As soon as the checkpoint process is finished the *checkpoint bit* is set. At the same time the counter *CheckpointValid* is set to $ErrorDetectionLatency$, and we let it decrement. After $ErrorDetectionLatency$ cycles, when *CheckpointValid* reaches 0, it asserts the *valid* signal indicating a valid checkpoint as no error was detected.

*CheckpointValid* counter guarantees the correctness of the checkpoint in the LLC. However, in order to be able to recover we must keep two copies of the state (one for the *yet-to-be-valid* checkpoint, and the other of previous *valid* checkpoint) of RAT, RF and PC. If an error was detected before the *CheckpointValid* reaches 0, we would just rollback to last *valid* checkpoint, ignoring the *checkpoint bit* of all cache lines in LLC.

### 4.3. Recovering from Error

Upon a particle strike, one of the detectors would trigger detecting the error. Recovering from an error requires a few steps:

1. Once we know the checkpoint is valid (*CheckpointValid* = "0") the recovery may begin. If not, we have to discard current checkpoint as explained earlier, and apply the recovery algorithm.
2. Restore architectural state from shadow copy.
3. Invalidate all the dirty lines and set the counter of L1 cache to force unknown state (i.e. counter = "X").
4. Invalidate all the dirty lines of LLC that are not part of the checkpoint.
5. Set the counter of LLC to force unknown state (i.e. counter = "X").

### 4.4. Intrusiveness of Design

The proposed architecture is extremely simple. It achieves SDC- & DUE 0 core using just one counter for caches within the containment area (i.e., L1 and LLC). It also requires one *checkpoint bit* for every cache line in the cache that is the checkpoint boundary (i.e., LLC). To validate the checkpoint we have one global counter *CheckpointValid* for LLC.

Regarding the checkpoint itself, we maintain 2 of the most recent copies of RAT, RF and PC, encoded using ECC. Having a shadow register file for checkpointing register files and keeping the log of register alias table incurs little area and power overhead [13]. Besides their impact on performance is minimal as retrieving and saving the data can be done simultaneously and in 1 cycle.

Also during the recovery process, invalidation of cache lines and clearing the checkpoint bits and counters can be done in one cycle as proposed in [8].

## 5. Scalability of Solution

In this section, we discuss the scalability of the proposed architecture in multicore systems and describe the interaction with the processor during normal operation. We also define the most important challenges for achieving high levels of error protection and error containment.

### 5.1. Shared Memory Architecture

In a shared memory architecture, the LLC is physically distributed in multiple banks but logically unified among all cores. As data are shared among different cores, the allocated blocks and all cache accesses are controlled via a coherency protocol. For our baseline core, we have chosen a MOESI protocol [22].

**5.1.1. MOESI Protocol for Error Containment.** The MOESI protocol allows several copies of cache lines across multiple processors to be different from the copy in shared LLC. Owned *(O)* cache lines are responsible to share data among the requesting processors. Owned state also writes back the data in the case of replacement. All other cache line copies remains in Shared *(S)* state. Moreover, cache lines in Modified *(M)* and Owned *(O)* states hold dirty data. The most important issue in a shared memory architecture is that a dirty block can be directly read by another processor without writing back to the shared memory.

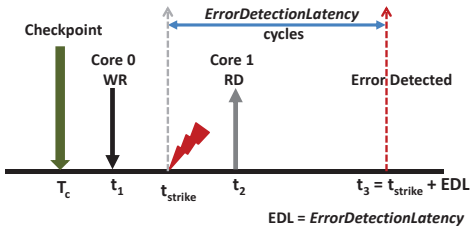



**Figure 7: Shared memory accesses for DUE 0 architecture**

Let us show the potential issue through an example. We consider 2 cores with shared memory. Figure 7 shows a scenario in which "core 0" has taken a checkpoint at time $T_c$. At instance $t_1$ "core 0" writes in cache. At time $t_2$ "core 1" requests a read from the cache in "core 0" following a cache miss in local cache. Now, if there is an error at time $t_{strike}$ in "core 0", detectors from "core 0" trigger after $ErrorDetectionLatency$ cycles at time $t_3$. Now, as soon as "core 0" recovers using the local checkpoint taken at time $T_c$, "core 1" will have invalid data. To avoid such cases we propose to stall all the read requests coming from other cores

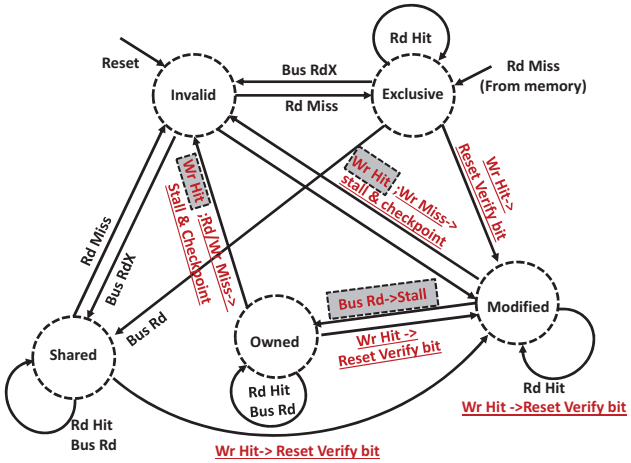



**Figure 8: MOESI protocol: Transitions are shown in the *trigger->action* format. Underlined transition triggers and actions are the same as uniprocessor architecture. The transition triggers in gray boxes are extensions for multicore shared memory architecture. "Wr" stands for write and "Rd" stands for read operation. "Stall" -> *ErrorDetectionLatency* cycles.**

and once the cache is verified, it can service read requests from other cores.

**5.1.2. MOESI Protocol for Checkpointing.** We adopt an incremental checkpointing, similar to the case of uniprocessor architecture explained in Section 4.

Compared to uniprocessor system, shared memory introduces a new situation that needs to be handled to properly create checkpoints: when one processor invalidates dirty data from another processor that is not part of the checkpoint. If it turns out that the requestor processor suffers an error, and triggers a recovery, it has to trigger another recovery in the owner processor in such a way that invalidated data can be recovered. In order to deal with this case, we employ previously proposed solutions that keep track of the sharing history [25, 39]. We summarize the adapted MOESI protocol in Figure 8.

**5.1.3. Recovering from Error.** If a core triggers an error, data recovery takes place in the same way as described for uniprocessor. The only caveat is that we will have to check the sharing history in order to initiate the recovery process in other processors cores [25].

### 5.2. Dealing with Interrupts and Exceptions

In this section we will discuss how we can handle I/O requests and exceptions.

**5.2.1. Handling Interrupts.** Interrupts are asynchronous events coming from the core and external devices (i.e., disk controller). Interrupts are crucial, as the requestor is outside the error containment area.

Similar to [40], we allow only error free stores to propagate to memory. We propose to buffer the requests in local memory, protected with ECC for *ErrorDetectionLatency* cycles. This assures the correctness of each outgoing store and all its preceding instructions. The size of the buffer should be large enough to hold the I/O requests for *ErrorDetectionLatency* cycles. Also, in order to facilitate successful recovery, as we allow all the error free stores to commit to memory after the last checkpoint, we must keep the load values issued so far in the buffer. Upon recovery we replay the loads so all the committed stores are correctly reproduced.

We propose to have one buffer for each I/O device to facilitate successful recovery, with an expected interrupt response time penalty of 30 to 1000 ns, which is acceptable for typical asynchronous interrupts.

**5.2.2. Dealing with Exceptions.** Exceptions are synchronous events such as a "div 0" instruction or a page fault on instruction fetch. When the exception occurs, the corresponding entry of ROB is marked. Since in modern processors exceptions are rare events [22], we propose to delay the exception service by *ErrorDetectionLatency* cycles until all potential errors have been detected. In case of no error detection, we assume the exception to be genuine and invoke the respective handler and handle it precisely.

**5.2.3. Context switching and Multi-programming.** In order to handle context-switching, we allow the preempted thread to swap out and we propose to stall for *ErrorDetectionLatency* to make sure the preempted thread is error-free. After the context switch we take a checkpoint of the incoming thread. This is to make sure that in an event of error due to particle strike the thread can recover its state from the instance after the context switch.

| Parameter | Value |
|---|---|
| Number of Processors | 1-16 |
| Instr Window / ROB | 16/48 entries |
| Frequency | 2GHz |
| L1 I/D Cache per Core | 16 KB, 4-way, 64B |
| LLC Cache per bank | 256 KB, 4-way, 64B (distributed 1-16 banks) |
| L1 access Latency | 2 cycles |
| LLC access Latency | 6 cycles |

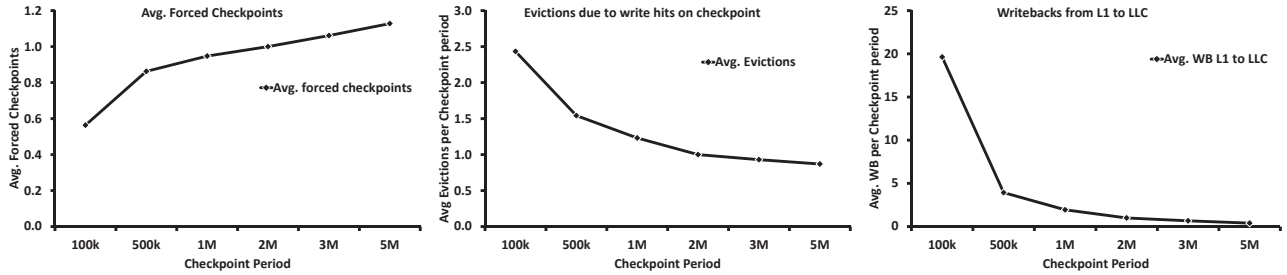
**Table 3: Configuration Parameters**

## 6. Evaluation

In this section, we analyze how error detection latency impacts the choice of error containment boundary. Next, we study the trade-off between checkpoint period and checkpoint boundary. Finally, we evaluate the performance impact of the selected configuration for uniprocessor and multicore system with data sharing and non-data sharing applications.

### 6.1. Experimental Setup

To evaluate the proposed architecture, we use a full-system execution-driven simulator extended with OPAL and GEMS tool-set [29]. We modified the memory hierarchy model to adapt it to the proposed architecture. Table 3 enlists the important configuration parameters.




(a) Forced checkpoints (b) Eviction due to write hits (c) Writeback from L1 to LLC

**Figure 9: Checkpoint events in LLC checkpoint boundary**

We simulate two different configurations as follows:

**6.1.1. Single core system.** All caches are private to the processor. All the LLC misses are served by the main memory. We evaluate the performance of single core system using SPEC CPU2006 benchmark set with the reference input set.

**6.1.2. Multicore system.** We implement a 16-core multicore system. We perform our analysis based on the following categories of applications, with a trace run of 20M cycles.

- **Non-data sharing applications:** To obtain various trade-off details for Non-data sharing applications we replicate the same application for all 16 cores (i.e., all 16 cores running the same application independently). We evaluate the performance of this configuration using SPEC CPU2006 benchmark set with reference input set.
- **Shared Memory Applications:** For our 16 core system, we use the SPEC OMP2001 benchmark set with an appropriate input set in order to evaluate the trade-offs.

### 6.2. Error Detection Latency vs Containment Area

We first analyze the trade-off between the error detection latency and the size of the error containment area. As we mentioned in Section 4.1, non-verified data is not allowed to leave the error containment and we need to stall until data is guaranteed to be correct, which degrades performance. We evaluate the range of detection latency 30 to 100 cycles as proposed in Section 3.5.

| Detection latency | Total #Stalls | Avg. Wait cycles |
|---|---|---|
| 10 cycles | 6111 | 3.45 |
| 30 cycles | 15729 | 25.99 |
| 100 cycles | 38049 | 40.67 |
| 1000 cycles | 55164 | 108.2 |

**Table 4: Containment cost (i.e., #Stalls and wait cycles for each stall) for containment boundary limited to L1**

Table 4 shows result for having one counter for entire L1 cache. It shows total number of evictions that create stalls when L1 is not verified. With increase in detection latency the average wait cycles also increase, which in turn increase the cost of containment. For a detection latency of 100 cycles the total stalls (i.e., over a period of 20M cycles) are more than 35K, i.e., having one stall every 1K cycles. It also shows the average number of cycles that we need to stall for the non-verified L1 cache to be verified.

Overall, we observe that for an *ErrorDetectionLatency* of 100 cycles, we experience a 7% slowdown only due to containment in L1. Even for 30 cycles, slowdown is 2%.

For the sake of comparison, we also experimented with more expensive solutions: (i) having one counter for each line, and (ii) one counter per set. Compared to having a counter per line, we observe an increase in total stalls by 5% for one counter per set, and 21% to one counter for the whole cache. Unfortunately, the slowdown due to containment is still high when having a counter per line, with 5.4% slowdown for 100 cycles of *ErrorDetectionLatency*, and 1.6% for 30 cycles.

When moving containment boundary to LLC, we observed only a handful of stalls. Therefore, we conclude that the best option is to have LLC as containment boundary, with an error detection latency of 100 cycles (which requires 30 detectors) and a slowdown of 0.01%.

### 6.3. Checkpoint Length vs Checkpoint Area

Now, we observe the tradeoff between the checkpoint length and the cost of the checkpointing. LLC is the checkpoint boundary. In our adopted architecture as described in Section 4.2, we have identified the major factors that affect the performance as follows:

1. Wait cycles to guarantee that caches in containment boundary are verified.
2. The write-back of dirty cache lines to the checkpoint boundary upon checkpoint creation.
3. Forced checkpoint events due to evictions of dirty lines that are not part of checkpoint.
4. Evictions to memory due to write hits on dirty and checkpointed lines.

Notice that factors 3-4 are runtime factors, and will largely depend on the footprint of the application (and therefore, the size of the selected checkpoint boundary and the checkpoint period). On the other hand, factors 1-2 are the overhead that is paid at checkpoint creation.

Figure 9(a) shows the number of forced checkpoints, per checkpoint length, for different checkpoint periods. As one

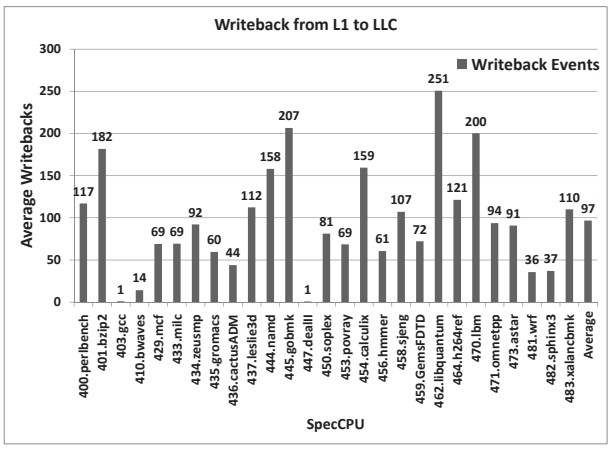



**Figure 10: Average number of dirty lines to be written back from L1 to LLC**

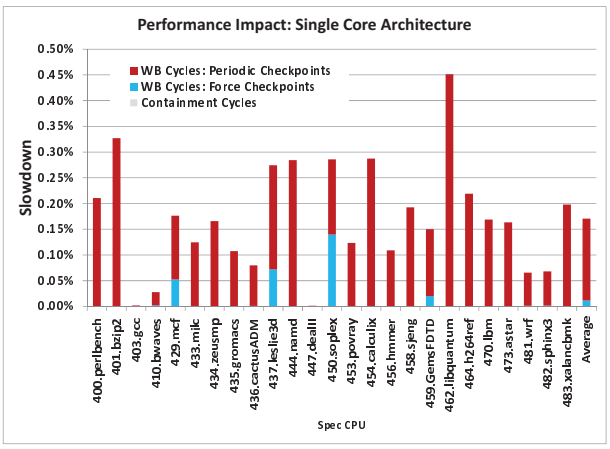



**Figure 12: Impact of containment and checkpointing LLC cache in unicore architecture**

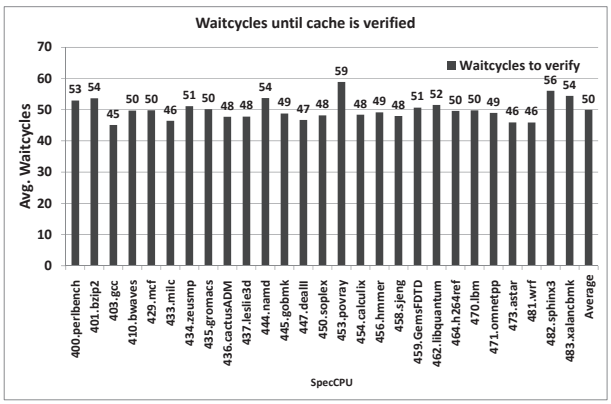



**Figure 11: Average number of wait-cycles until LLC is verified**

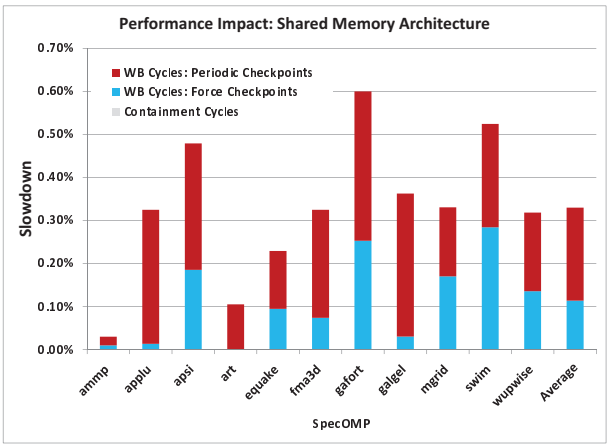



**Figure 13: Slowdown due to containment and checkpointing LLC cache in multi-core system for shared memory applications**

can see, the number of extra checkpoints is negligible, and therefore we can opt for long checkpoints in the order of millions of cycles. Figure 9(b) shows the number of extra evictions to main memory caused by write hits on checkpointed lines. Numbers are relative to the length of the checkpoint period. Regarding the cost of creating a checkpoint, we show in Figure 9(c) the extra write-backs we have to perform when taking a checkpoint. As shown in the figure, increasing the checkpoint period from 100K cycles to 2M cycles brings down the write-back traffic by more than 10×, and after that benefits flatten. Therefore, we opt for a checkpoint length of 2M cycles.

We detail the results for a 2M checkpoint period for our workloads in Figure 10. The results indicate that, for every 2M cycles we have to write-back 97 dirty cache lines from verified L1 cache to LLC.

Finally, we assess how much time we need to wait until we can create the checkpoint. Figure 11 shows the average wait cycles for the LLC to be verified before taking a checkpoint for a checkpoint period of 2M cycles. For a detection latency of 100 cycles, every 2M cycles we have to wait 50 cycles to take a checkpoint in LLC.

Next, we will see how the performance is impacted in the proposed architecture.

### 6.4. Uniprocessor Performance

Figure 12 evaluates the proposed single core architecture in terms of performance vs. cost of containment and recovery. The experimentation shows that the average performance slowdown is 0.17%. We notice that the average performance degradation due to containment is almost 0, since there are no eviction of dirty lines from non-verified LLC cache. The performance degradation due to checkpointing is 0.15%.

### 6.5. Performance of Multicore for Non-data Sharing Applications

We observe similar results for 16 core system for non-data sharing workloads. Figure is omitted due to lack of space. Notice, that we depict the results for the slowest core of the 16 running. The total degradation in performance is 0.19%.

### 6.6. Multicore Shared Memory Performance

Figure 13 shows the impact on performance for 16 cores shared memory architecture. Again, we collect data for the slowest core to reach the 20M cycles. As one can see, the average slowdown is 0.3%. Again even in the case of shared memory we do not have any dirty evictions from LLC before LLC is verified. Hence, the slowdown due to containment is zero. In shared memory architecture we have more cache lines evicting after the LLC is verified. This results into increased forced checkpoints.

### 6.7. Summary and Hardware Cost

The proposed architecture is extremely light-weight. It uses 30 detectors (i.e., area is 30 SRAM memory cells) for error detection, for a worst-case detection latency of 100 cycles. The area overhead of 30 interconnects is also reasonably small. Controller circuit to signal error detection is extremely simple and requires 6 3-input and 2 2-input logic-OR gates.

We implement the containment boundary at the LLC. Containment architecture consists of L1 and LLC with one counter each, to count 100 cycles. Additionally, we will need one counter for LLC to check the checkpoint validity. All 3 counters are 7-bit, non-repeating word counters.

Checkpointing requires a physical *Checkpoint bit* for every cache line in LLC. We propose to use a checkpoint length of 2M cycles, which guarantees a good trade-off between checkpoint overhead and recovery time. For recovery of architecture state it requires 2 shadow copies of the architectural state (register files, RAT and PC). We also use a trivial control circuit for clearing the *Checkpoint bit* and counters in one cycle.

## 7. Related Work

Several popular error detection mechanisms have been compared and summarized in Table 1. The most effective method of dealing with soft errors in memory components is to use codes like parity, or ECC [33]. Execution redundancy is a widely used technique to detect errors in the logic, either using the multithreading capabilities [34, 43] or hardware redundancy [45, 46]. DIVA [3] uses a simple in-order core as a checker for an out-of-order core. Triple redundancy systems are used in commercial processors (i.e., HP NonStop architecture [7]) and "Pair & spare" systems [5] and can achieve 0 DUE without roll-back. The work of [48] shows how to handle the DUE problem in L1 caches.

Current work focuses more on detection latency and its impact on containment and cost of recovery unlike [47]. CARER [24], Cherry(-MP) [25, 30] & others [2, 11, 25, 27, 39, 51] are popular for speculation recovery. The proposed solution is extremely simple and uses only one counter per cache (i.e., one counter for L1 and one for LLC). CARER [24], Cherry(-MP) [25, 30] and others do not benefit from this optimization. Moreover, CARER-like implementation (i.e., containment in L1) causes 7% slowdown as shown in Section 6.2. In contrast to solutions similar to [11], the proposed architecture protects the entire CMP system.

## 8. Conclusions

This paper presents a novel architecture to eliminate particle strike induced SDC & DUE FIT. The architecture uses acoustic wave detectors to detect errors. The proposed architecture minimizes hardware overhead and performance cost. We showed that even in the worst case scenario, the proposed architecture detects only 1 false positive every 1.3 minutes for current general purpose multi-core systems. Next, we proposed an error containment mechanism within the cache hierarchy to manage the detection latency. Finally, we achieve DUE-FIT 0 by enabling a low cost checkpointing mechanism.

Our proposed architecture eliminates particle strike induced SDC & DUE FIT, for systems ranging from one core to 16-core with shared memory with the worst-case performance overhead is 0.60% for shared memory systems.

## 9. Acknowledgements

We would like to thank Nicholas Axelos and Javier Carretero for their valuable comments and discussions in shaping up the idea. We would like to thank Ehsan Ardestani for his inputs towards improving the quality of the paper.

This work has been partially supported by the Spanish Ministry of Education and Science under grant TIN2010-18368, the TRAMS project of the FP7 program of the European Commission under agreement 248789, the Generalitat of Catalunya under grants 2009SGR1250 and FI-DGR-2010.

## References

[1] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P Jouppi, and James E Smith. Configurable isolation: building high availability systems with commodity multi-core processors. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[2] Rana E Ahmed, Robert C Frazier, and Peter N Marinos. Cache-aided rollback error recovery (CARER) algorithm for shared-memory multiprocessor systems. In *Digest of Papers, 20th International Symposium Fault-Tolerant Computing (FTCS)*, IEEE, 1990.

[3] Todd M Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proceedings of International Symposium on Microarchitecture (MICRO)*, 1999.

[4] Rajeev Balasubramonian, Naveen Muralimanohar, Karthik Ramani, and Venkatanand Venkatachalapathy. Microarchitectural wire management for performance and power in partitioned architectures. In *11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 28–39, IEEE, 2005.

[5] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing,*, 1(1):87–96, 2004.

[6] Robert Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, 2001.

[7] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. Nonstop® advanced architecture. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 12–21, IEEE, 2005.

[8] Edson Borin, Youfeng Wu, Mauricio Breternitz, and Cheng Wang. Lar-cc: Large atomic regions with conditional commits. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 54–63, 2011.

[9] Javier Carretero, Xavier Vera, Jaume Abella, Tanausu Ramirez, Matteo Monchiero, and Antonio Gonzalez. Hardware/software-based diagnosis of load-store queues using expandable activity logs. In *Proceedings of 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 321–331, 2011.

[10] Smitha Shyam, Kypros Constantinides, Sujay Phadke, Valeria Bertacco, and Todd 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 (ASPLOS)*, 2006.

[11] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Bulletproof: A defect-tolerant cmp switch architecture. In *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 5–16, 2006.

[12] Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *Proceedings of the International Reliability Physics Symposium (IRPS)*, 2011.

[13] Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. Early register deallocation mechanisms using checkpointed register files. *IEEE Transactions on Computers*, vol 55(9), pages 1153–1166, 2006.

[14] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *ACM SIGARCH Computer Architecture News*, vol 38, pages 385–396, 2010.

[15] Jiri Gaisler. A portable and fault-tolerant microprocessor based on the sparc v8 architecture. In *Proceedings of International Conference on Dependable Systems and Networks*, DSN 2002.

[16] Balkaran Gill, Michael Nicolaidis, Francis Wolff, Chris Papachristou, and Steven Garverick. An efficient bics design for seus detection and correction in semiconductor memories. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 592–597, IEEE Computer Society, 2005.

[17] Mohamed Gomaa, Chad Scarbrough, TN Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of 30th Annual International Symposium on Computer Architecture*, pages 98–109, IEEE, 2003.

[18] M.D. Hammig. The design and construction of a mechanical radiation detector. In *Proceedings of IEEE Nuclear Science Symposium*, pages 803–805, Michigan Univ., Ann Arbor, MI, 1998. IEEE.

[19] M.D. Hammig. Nuclear radiation detection via the detection of pliable microstructures. In *Proceedings of Nuclear Instruments and Methods in Physics Research*, pages 278–281, Los Alamitos, CA, USA, 1999. Elsevier Science.

[20] Eric Hannah. Cosmic ray detectors for integrated circuit chips. United States Patent Number 7309866B2, December 2007.

[21] Tino Heijmen. Radiation-induced soft errors in digital circuits–a literature survey. 2002.

[22] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Elsevier Science, 2007.

[23] Zheng Feng Huang and Mao Xiang Yi. Biss: A built-in seu sensor for soft error mitigation. *Applied Mechanics and Materials*, vol 130, pages 4228–4231, 2012.

[24] Douglas B Hunt and Peter N Marinos. A general purpose cache-aided rollback error recovery (CARER) technique. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems*, pages 170–175, 1987.

[25] Meyrem Kyrman, Nevin Kyrman, and Jose F Martynez. Cherry-mp: Correctly integrating checkpointed early resource recycling in chip multiprocessors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 245–256, 2005.

[26] Sandia National Laboratories. Microsensors and sensor microsystems. `http://www.sandia.gov/mstc/MsensorSensorMsystems/technical-information/SH-SAW-biosensors.html`

[27] Christopher LaFrieda, Engin Ipek, Jose F Martinez, and Rajit Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 317–326, 2007.

[28] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. *ACM Sigplan Notices*, 43(3):265–276, 2008.

[29] Milo MK Martin, Daniel J Sorin, Bradford M Beckmann, Michael R Marty, Min Xu, Alaa R Alameldeen, Kevin E Moore, Mark D Hill, and David A Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[30] José F Martínez, Jose Renau, Michael C Huang, and Milos Prvulovic. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings. 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2002.

[31] Albert Meixner, Michael E Bauer, and Daniel J Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[32] Subhashish Mitra, Norbert Seifert, and Pia Sanda. Soft errors: Trends, system effects, and protection techniques. IOLTS-Tutorial Slides, December 2007.

[33] Shubhendu S Mukherjee. *Architecture Design for Soft Errors.* 1st edition, 2009.

[34] Shubhendu S Mukherjee, Michael Kontz, and Steven K Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2002.

[35] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro*, 28(1):69–79, 2008.

[36] Ashay Narsale and Michael C Huang. *Variation-tolerant hierarchical voltage monitoring circuit for soft error detection*. IEEE, 2009.

[37] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[38] Paul Racunas, Kypros Constantinides, Srilatha Manne, and Shubhendu S Mukherjee. Perturbation-based fault screening. In *IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA 2007.

[39] M Wasiur Rashid and Michael C Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 393–404. IEEE, 2008.

[40] Steven K Reinhardt, Shubhendu S Mukherjee, Joel S Emer, et al. Periodic checkpointing in a redundantly multi-threaded architecture, December 11 2007. US Patent 7,308,607.

[41] George A Reis, Jonathan Chang, Neil Vachharajani, Shubhendu S Mukherjee, R Rangan, and DI August. Design and evaluation of hybrid fault-detection systems. In *Proceedings of 32nd International Symposium on Computer Architecture*, pages 148–159, 2005.

[42] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 243–254, IEEE Computer Society, 2005.

[43] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of International Symposium on Fault-Tolerant Computing (FTC)*, page 84, 1999.

[44] Timothy J Slegel, Robert M Averill III, Mark A Check, Bruce C Giamei, Barry W Krumm, Christopher A Krygowski, Wen H Li, John S Liptay, John D MacDougall, Thomas J McPherson, et al. Ibm's s/390 g5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[45] Lisa Spainhower and Thomas A Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, 1999.

[46] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: improving both performance and fault tolerance. In *Proceedings of the 33th International Symposium on Microarchitecture (MICRO)*, 2000.

[47] Gaurang Upasani, Xavier Vera, and Antonio González. Setting an error detection infrastructure with low cost acoustic wave detectors. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012.

[48] Gaurang Upasani, Xavier Vera, and Antonio González. Reducing due-fit of caches by exploiting acoustic wave detectors for error recovery. In *Proceedings of the 19th International On-Line Testing Symposium (IOLTS)*, pages 85–91, 2013.

[49] Xavier Vera, Jaume Abella, Javier Carretero, and Antonio González. Selective replication: A lightweight technique for soft errors. *ACM Transactions on Computer Systems (TOCS)*, 27:8:1–8:30, January 2010.

[50] Nicholas J Wang and Sanjay J Patel. Restore: Symptom-based soft error detection in microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3):188–201, 2006.

[51] K-L Wu, W. Kent Fuchs, and Janak H. Patel. Error recovery in shared memory multiprocessors using private caches. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):231–240, 1990.