R^{2}Cache: Reliability-Aware Reconfigurable Last-Level Cache Architecture for Multi-Cores

Florian Kriebel, Arun Subramaniyan, Semeen Rehman,
Segnon Jean Bruno Ahandagbe, Muhammad Shafique, Jörg Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT)
Corresponding authors: {florian.kriebel,muhammad.shafique}@kit.edu

ABSTRACT

On-chip last-level caches in multicore systems are one of the most vulnerable components to soft errors. However, vulnerability to soft errors highly depends upon the parameters and configuration of the last-level cache, especially when executing different applications. Therefore, in a reconfigurable cache architecture, the cache parameters can be adapted at run-time to improve its reliability against soft errors.

In this paper we propose a novel reliability-aware reconfigurable last-level cache architecture (R^{2}Cache) for multicore systems. It provides reliability-wise efficient cache configurations (i.e. cache parameter selection and cache partitioning) for different concurrently executing applications under user-provided tolerable performance overheads. To enable run-time adaptations, we also introduce a lightweight online vulnerability predictor that exploits the knowledge of performance metrics like number of L2 misses to accurately estimate the cache vulnerability to soft errors. Based on the predicted vulnerabilities of different concurrently executing applications in the current execution epoch, our run-time reliability manager reconfigures the cache such that, for the next execution epoch, the total vulnerability for all concurrently executing applications is minimized. In scenarios where single-bit error correction for cache lines may be afforded, vulnerability-aware reconfigurations can be leveraged to increase the reliability of the last-level cache against multi-bit errors. Compared to state-of-the-art, the proposed architecture provides 24% vulnerability savings when averaged across numerous experiments, while reducing the vulnerability by more than 60% for selected applications and application phases.

1. INTRODUCTION AND RELATED WORK

Due to small feature sizes, low operating voltages and increased clock speeds, microprocessors in the nano-era are highly susceptible to soft errors [1, 2]. Soft errors are transient errors[1] that manifest themselves as spurious bit-flips in the underlying hardware and can cause application program failures. In modern microprocessors, on-chip caches (especially the last-level) occupy a majority of the die area, making them one of the most vulnerable components to soft errors [3]. These errors can propagate to the CPU and other pipeline components (register file, issue queue, etc.) leading to incorrect output or even system failures. As a result, a large body of research has investigated techniques to ensure data integrity in caches [4, 5]. For designing cost-constrained reliable caches, two key challenges are (1) soft error vulnerability analysis and (2) soft error mitigation/reliability mechanisms.

With regard to vulnerability analysis, since not all errors affect the program output (e.g., read followed by a subsequent write to the same data), analytical models to accurately estimate cache vulnerability to soft errors were developed [3, 6, 7, 8]. Zhang et al. [3] extended the concept of Architectural Vulnerability Factor (AVF) [6] to derive the Cache Vulnerability Factor (CVF) metric, defined as the probability that a soft error in the cache propagates to the CPU or lower levels of the memory hierarchy and characterized the lifetime of a cache line into vulnerable and non-vulnerable phases. Recently, the works in [7, 8] performed a fine-grained cache vulnerability analysis to study the influence of read frequency on the accuracy of their vulnerability estimation. These efforts helped reducing the large overhead incurred by over-estimating the required cache protection. However, these techniques concentrate only on reducing the *Temporal Vulnerability*, i.e., the fraction of the execution time (vulnerable period) during which a fault in a cache line is propagated either to the CPU or lower levels of the memory hierarchy. These techniques do not (precisely) account for the *Spatial Vulnerability*, i.e., the probability of error due to active areas of the cache (regions exhibiting high line reuse) and vulnerable bits of each cache line used by different applications. In [9], a runtime adaptation technique is proposed for NoCs which is based on a lightweight vulnerability prediction mechanism. However, this work does not target cache vulnerability estimation and optimization.

With regard to soft error mitigation, Write-Through (WTC) and Early Write-Back (EWB) caches have been explored in literature [4], that significantly reduce the error probability for dirty cache blocks by writing copies to the memory system as soon as data in the cache lines is updated. But they result in increased memory traffic and subsequently higher performance overhead. Early Write-Back strategies write cache lines to memory at periodic intervals trading off vulnerability for performance. Efforts in [10] extend this approach by customizing the writeback intervals to the data access patterns of applications and achieve a better performance-reliability tradeoff. However, these techniques cannot capture the effects of changing the cache parameters (e.g., partition size, line size and associativity) on the cache vulnerability for different applications that provides an increased potential for vulnerability reduction at low-moderate performance overheads. Moreover, these techniques primarily target *offline* vulnerability analysis and cannot be deployed for run-time estimation due to their intensive computations. *Therefore, there is a need for an accurate cache vulnerability modeling and a lightweight online estimation technique that accounts for both spatial and temporal vulnerabilities of concurrently executing applications*

[1]induced by high-energy neutrons from cosmic rays or low-energy alpha particles from packaging material.

under changing cache configurations.

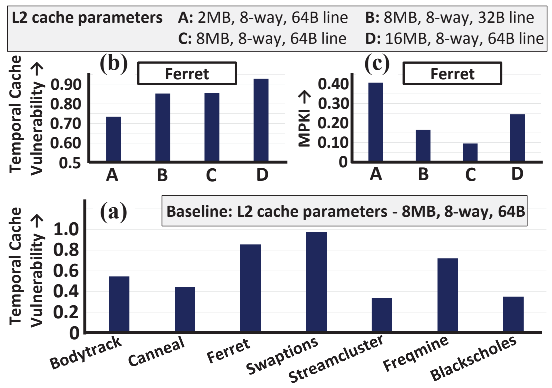

Figure 1: (a) Varying vulnerabilities of different applications from the PARSEC [11] benchmark suite for the baseline case. (b-c) Changing vulnerabilities and cache misses for *Ferret* application under different cache configurations.

Cache Vulnerability and Dynamic Cache Reconfiguration: Different applications differ in their access patterns and occupancy of the last-level on-chip cache, which leads to varying vulnerabilities of these applications for a given cache configuration; see Figure 1(a). Moreover, our analysis in Figure 1(b-c) for the *Ferret* application, shows that different configurations provide a tradeoff between vulnerability and performance (in terms of cache misses given as MPKI, i.e., Misses-Per-Kilo-Instructions). This motivates the need to dynamically reconfigure the cache parameters to provide reliability-wise beneficial configurations for a set of different concurrently executing applications under user-provided tolerable performance overhead. State-of-the-art in reconfigurable caches have primarily explored configurations w.r.t. performance and energy consumption [12, 13]. These approaches provide basic infrastructure, for instance, altering cache size through reconfiguring cache ways, way-concatenation and way-shutdown to power-off unused ways, and adaptation of other parameters like associativity and line size [14, 12] based on the cache miss rate and branch prediction rate. Recently, adapting partitions for last-level caches has also gained interest where instead of reconfiguring the whole cache, private and shared partitions are used [15, 16].

The broad open question that is unaddressed in literature is: *if and how to dynamically reconfigure caches (considering both cache partitioning and parameter selection) to reduce the total vulnerability for a set of concurrently executing applications with distinct access patterns leading to varying spatial and temporal vulnerabilities. We refer to this as reliability-aware dynamic cache reconfiguration.*

1.1 Novel Contributions

In this paper, we propose a *novel Reliability-Aware Reconfigurable Cache Architecture (R^2Cache)* (see Section 2) that dynamically reconfigures different last-level cache parameters (like cache partition sizing, line sizing, and associativity adaptation) for different applications depending upon their access patterns. It optimizes the vulnerabilities in a two step process: (1) *Inter-Application Reconfiguration*, i.e., adaptation across different cache partitions of different applications and (2) *Intra-Application Reconfiguration*, i.e., adaptations between different phases of an application that exhibit diverse cache access behavior due to their varying data and control flow properties. To enable this, *R^2Cache* employs the following two novel components.

(1) A Light-Weight Online Vulnerability Predictor that estimates the cache vulnerability of an application (phase) considering cache utilization and performance metrics (like cache miss rate in terms of MPKI, IPC: Instructions-Per-Cycle, etc.) at run time (see Section 3.4). Towards this end, we also propose a *novel cache vulnerability quantification model* as a joint function of spatial and temporal vulnerabilities (see Section 3.3).

(2) A Vulnerability-Driven Configuration Manager that leverages the predicted vulnerabilities to select a reliability-wise beneficial last-level cache configuration during the Inter- and Intra-Application reconfiguration process (see Section 4). To devise an efficient configuration management technique, it is important to understand the influence of different cache configurations on the overall cache vulnerabilities due to different cache operations (like read, eviction, etc.). Towards this end, we have also performed a *comprehensive cache vulnerability analysis* in this paper (see Section 4.1).

Applicability in ECC-protected Caches: Reliability-aware cache reconfiguration can also be applied in conjunction with error correcting codes (ECCs) like SEC-DED [17] (Single Error Correcting-Double Error Detecting) to improve reliability in multi-bit error scenarios or in cases where only some of the cache partitions are ECC-protected due to area constraints. In Section 6.6 we will show that single-bit error correction can be applied to R^2Cache to provide further reliability benefits at a small performance overhead (less than 10%).

2. R^2CACHE: RELIABILITY-AWARE RECONFIGURABLE CACHE ARCHITECTURE

Figure 2 illustrates the architecture overview of our R^2Cache highlighting the novel components for online vulnerability prediction and vulnerability-optimizing configuration management along with the architectural support for cache parameter reconfiguration. The decision of reconfiguration is orthogonal to the architectural support for reconfiguring cache parameters, thus making it possible for us to implement our own run-time configuration management technique on top of existing reconfigurable cache architectures like [13, 16].

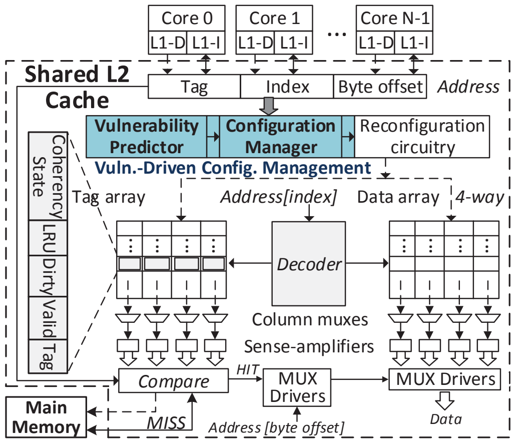

Figure 2: Reliability-Aware Reconfigurable Cache Architecture

Without the loss of generality, in this paper, we adopt the following three architectural features for reconfiguration as the basic infrastructure support from [13, 16]:

1. *Way-Concatenation* is the method to reduce the associativity of the cache without changing the effective cache size.
2. *Way-Shutdown* powers-off certain ways of the cache set, thus resulting in a virtually smaller cache from the application point of view.
3. *Cache Line Resizing* methods start with a given baseline size and add/remove cache lines based on the requirement.

Existing approaches like [13, 16] have shown that the reconfiguration control does not lie on the critical path of execution and incurs minimal performance overhead (e.g., using reconfiguration registers to alter the cache configuration). These configuration registers will be set by our Vulnerability-Driven Configuration Manager that performs inter- and intra application reconfigurations. During *inter-application reconfiguration*, it chooses a reliability-wise beneficial cache configuration (i.e. with the lowest vulnerability) for each application considering its average-case properties (in terms of performance and cache access behavior). Afterwards, it performs *intra-application reconfiguration* to tune the cache parameters in order to minimize vulnerabilities in different execution phases of an application.

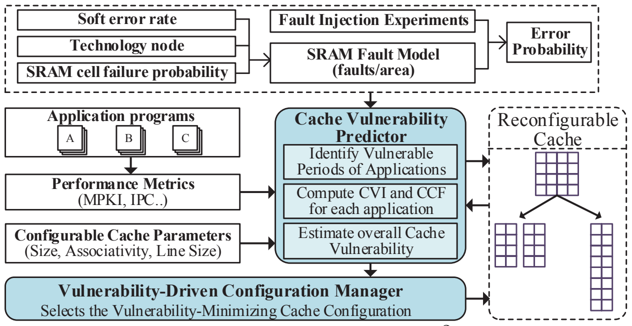
Soft error rate
Technology node
SRAM cell failure probability
Fault Injection Experiments
SRAM Fault Model (faults/area)
Error Probability
Application programs
A
B
C
Performance Metrics (MPKI, IPC..)
Configurable Cache Parameters (Size, Associativity, Line Size)
Cache Vulnerability Predictor
Identify Vulnerable Periods of Applications
Compute CVI and CCF for each application
Estimate overall Cache Vulnerability
Reconfigurable Cache
Vulnerability-Driven Configuration Manager
Selects the Vulnerability-Minimizing Cache Configuration

Figure 3: Operational Flow of R^{2}Cache

To determine a reliability-wise beneficial configuration, there is a need for a light-weight vulnerability estimation technique that estimates the vulnerability of future execution epochs. To reduce the run-time overhead, our vulnerability predictor leverages the cache access and execution behavior (e.g., MPKI, IPC) of the executing applications in the current execution epoch. Our vulnerability predictor employs linear regression to estimate the cache vulnerability using the correlation existing between vulnerability and L2 miss count, as we discuss in the following section. The detailed component flow and system overview along with input/output is shown in Figure 3. Based on an online analysis of the expected applications running on the system, several performance metrics are analyzed for different cache configurations. This information is used in combination with the error probability obtained from an SRAM Fault Model and results from Fault Injection Experiments to estimate the vulnerability of an application phase employing a Cache Vulnerability Predictor. This finally provides the input for the selection of a vulnerability-minimizing cache configuration. The individual components of Figure 3 are explained in detail in the subsequent sections.

3. CACHE VULNERABILITY MODELING AND PREDICTION

3.1 Fault Modeling and Injection

As widely adopted in literature, our fault model for the cache is based on single and multiple bit flips in different cache lines. We define the soft error rate of an SRAM based cache ($Cache_{SER}$, measured in FIT, 1 FIT = 1 error in 1 billion hours) as the product of the error rate (dependent on the circuit design, raw error rate obtained from particle flux rate and the technology implementation) and the cache's vulnerability index (CCF, see Eq. (4)), which is an indicator of the likelihood that such an error can corrupt the program state. The top part of Figure 3 outlines the steps involved in the calculation of the probability of error at a given fault rate fr ($P_{error}(fr)$) after considering different masking effects (e.g., masking of errors by invalid cache lines, combinational logic, etc). For a given technology implementation (e.g., 45 nm), circuit design, operating voltage and particle flux rate, we determine the SRAM soft error rate (fr) as in [18, 19]. Taking p_{flip} to be the probability that a particle strike leads to a change in the logic state of a cache bit, the SRAM fault rate (in terms of #faults/area) is derived. For that, it needs to be considered that at smaller technology nodes the amount of multi-bit upsets (MBUs) from a single radiation event increases [18]. Considering the above fault model, we perform a series of fault injection experiments to include the effects of error masking in our analysis. The number of faults injected in a component is proportional to the area occupied by that component [20], to account for its spatial vulnerability (N_{FI}). With each application exhibiting diverse error masking capabilities, we count the number of application failure scenarios ($N_{failure}$). Using this information, $P_{error}(fr)$ is calculated as:

$$P_{error}(fr) = p_{flip} \times N_{failure}(fr)/N_{FI}(fr) \qquad (1)$$

3.2 Cache Vulnerability Components

Figure 4 illustrates different vulnerability components in cache accesses as explained below.

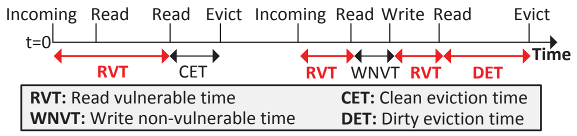

Figure 4: Vulnerability Components in Cache Accesses

Read Vulnerable Time (RVT) denotes the time interval between an incoming/write/read and a subsequent read to a cache block, during which the transient errors are highly likely to corrupt the program state.
Write Non-Vulnerable Time (WNVT): Writes to a cache block after a read overwrites erroneous data caused by bit flips. We perform vulnerability analysis at the fine-granularity of a byte and hence the time interval between a read and a write or subsequent writes can be treated to be non-vulnerable.
Clean Eviction Time (CET) denotes the time interval between the last read and replacement of a cache block and is non-vulnerable because bit flips in clean blocks do not propagate to the memory as there exists an updated copy in the lower levels of the memory hierarchy.
Dirty Eviction Time (DET) denotes the time interval between the last access to a dirty cache block and subsequent replacement because bit flips in the updated blocks that have a *stale* (i.e., not updated) copy in the lower levels of the memory hierarchy (write-back cache) may affect the correct program state.

3.3 Our Cache Vulnerability Model

The set of cache configurations is given as $C = \{C_1, C_2, \ldots, C_K\}$. Without loss of generality, we define $C_i = \{CS_i, CA_i, LS_i\}$ as a point in the configuration space with three configurable parameters: cache partition size (CS), cache associativity (CA) and line size (LS). An application is given as a set of nodes $P = \{P_1, P_2, ...P_N\}$ representing different phases during its execution. As adopted by the architecture community [21], we consider a phase to be an interval of 100 million instructions. Phase classification is based on clustering basic blocks considering their execution behavior and their frequency of occurrence. In order to quantitatively estimate the vulnerability of the cache in configuration C_i, running phase P_j of an application, we define two metrics:

Cache Vulnerability Index $CVI\,(P_j, C_i)$ (Eq. (2)) of an application in phase P_j with cache configuration C_i is defined as the product of total vulnerable cache bits and vulnerable periods of the relevant cache lines normalized to the total bits used by the application and its execution time. CVI captures the *temporal vulnerability* by monitoring the cache's vulnerable periods ($VulPeriod_l(P_j, C_i)$) during an application's execution and *spatial vulnerability* by using the vulnerable bits ($VulBits_l$, i.e., the bits needed for architecturally correct execution) of a cache line l

with total number of bits $Bits_l$.

$$CVI(P_j, C_i) = \frac{\sum_{\forall l \in L} VulBits_l \times VulPeriod_l(P_j, C_i)}{\sum_{\forall l \in L} Bits_l \times ExecTime(P_j)} \quad (2)$$

Note, for a cache line l, $VulPeriod_l(P_j, C_i)$ is the sum of RVT and DET (see Figure 4).

Cache Criticality Factor $CCF(P_j, C_i)$ (Eq. (4)) of an application in phase P_j with cache configuration C_i is defined as the product of $CVI(P_j, C_i)$ and the fraction of the total number of cache lines N_L used by phase P_j of the application weighted with the error probability $P_{error}(fr)$ in SRAM cells.

$$cA_{(P_j,C_i)} = N_{AL}(P_j) \times LA/N_L \quad (3)$$

$$CCF(P_j, C_i) = CVI(P_j, C_i) \times cA_{(P_j,C_i)} \times P_{error}(fr) \quad (4)$$

As widely adopted in the cache literature [22], N_{AL} denotes the number of actively reused cache lines. $cA_{(P_j,C_i)}$ is the cache area used actively by the application in phase P_j using configuration C_i. LA is the area occupied by a cache line. CCF is a joint function of both the *spatial* and *temporal* vulnerability of the cache. CCF uses CVI as well as N_{AL} and $P_{error}(fr)$ (i.e., *hardware-dependent spatial vulnerability to soft errors*) to accurately estimate the overall vulnerability of the cache while executing different phases of an application.

3.4 Estimation of Cache Vulnerability

Our vulnerability predictor exploits the correlation existing between structure occupancies and cache vulnerability. Considering Eq. (2) for CVI, the parameters to be estimated are the number of vulnerable bits ($VulBits_l$) in the cache line l, the time interval for which the cache line is vulnerable during application execution ($VulPeriod_l(P_j, C_i)$), and the execution time of an application phase ($ExecTime(P_j)$). Our predictor relies on a combination of both *offline analysis* and *online monitoring* of certain performance metrics to make a prediction for CCF, as discussed below.

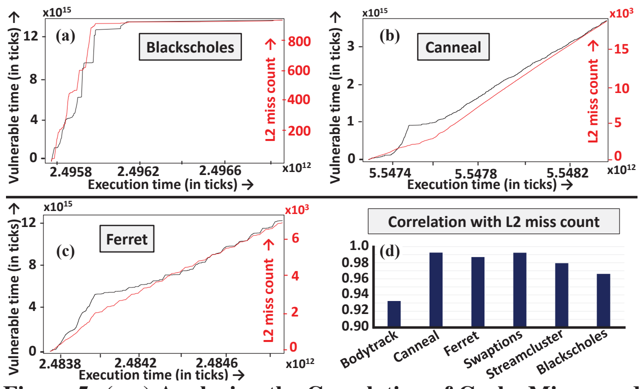

Figure 5: (a-c) Analyzing the Correlation of Cache Misses and Vulnerability for Three Applications. (d) Predictor Correlation for Different Applications.

The vulnerable bits $VulBits_l$ can be estimated offline using Monte Carlo fault injection campaigns (as discussed in Section 3.1) or using static program analysis. We consider $ExecTime(P_j)$ to be the average execution time of the phase, obtained, for instance, using a history of h previous iterations (can also be obtained using an offline execution analysis). To estimate $VulPeriod_l(P_j, C_i)$, we examined various processor performance metrics to study their correlation with the vulnerable period. Our analysis in Figure 5 shows that the number of L2 misses shows a high degree of correlation with the vulnerable period, thus online monitored cache misses can be used for the online prediction of temporal vulnerability. Considering a total of N_O observations the vulnerable period ($VulPeriod_i$) in the i-th observation can be expressed as:

Table 1: Prediction accuracy

Application (phase 2)	Linear model used (x10^{12})	Average prediction accuracy (%)
Bodytrack	35.89*L2MissCount + 238.3	82.31
Canneal	0.2396*L2MissCount + 355.5	96.52
Ferret	1.569*L2MissCount + 948.3	89.16
Swaptions	103.5*L2MissCount + 7635	92.77
Streamcluster	0.7385*L2MissCount + 119.4	83.25
Blackscholes	11.89*L2MissCount - 69.67	87.69

$$VulPeriod_i = \alpha(L2MissCount) + error_i \quad (5)$$

where $error_i$ is the prediction error of the linear model in the i-th observation. In order to estimate the parameter α, we minimize the sum of squares of errors in different observations:

$$Vul.Period_{phase} = f(L2MissCount_{phase}) \times error_{prev\_phase} \quad (6)$$

Since the number of L2 misses is dependent on the input chosen for an application, we consider the average over different inputs while predicting the vulnerable period of the phase ($VulPeriod_i$).

Predictor Accuracy Results: To evaluate the accuracy of our vulnerability predictor, we simulated the phase of each application (phase 1). During this period we monitor the vulnerable periods and the corresponding L2 miss counts and construct a linear model as described above. Using the linear model obtained in phase 1, we predicted the temporal vulnerability of the same application for the next phase (phase 2). Table 1 summarizes the linear models and the average accuracy of our predictor for different applications, averaged over different inputs.

Our approach is orthogonal to existing application phase detection and classification efforts. Greater the accuracy in phase classification, better is the accuracy of the vulnerability predictor. However even if a phase does not provide enough computation for a reasonable estimation of vulnerability due to workload variations, this would be compensated in subsequent phases by considering the error of previous phases (shown in Eq. (6)). Frequent reconfigurations are avoided by applying a tolerable performance overhead.

As can be seen from Table 1, the predictor accuracy varies from 82 to 97%. This is attributed to the fact that the vulnerable period during an application's execution cannot be exactly derived using a linear model of the miss rate alone, but may have a linear/non-linear dependence with other performance metrics like IPC, branch prediction rate, etc. [23]. This increases the complexity of the predictor making it less attractive to be used online. We consider this tradeoff while designing our predictor.

With CVI evaluated as described above, another parameter that needs to be estimated is the number of cache lines which show high reuse (N_{AL}). Since N_{AL} is difficult to estimate using well-known performance metrics like IPC, miss rates, branch misprediction rates, we use the average value obtained during offline characterization of the application to estimate CCF. A potential solution could be to compute N_{AL} using active cache footprint vectors as proposed in [22]. Other parameters necessary to estimate CCF like the Line area LA and the number of cache lines N_L are known for a given microarchitecture implementation, while the error probability at a given fault rate $P_{error}(fr)$ is computed as discussed in Section 6.5.

4. RELIABILITY-AWARE DYNAMIC CACHE RECONFIGURATION

Before proceeding to the details of the configuration manager of R^2*Cache*, we present a comprehensive analysis to study the impact of different cache configurations on the cache vulnerability.

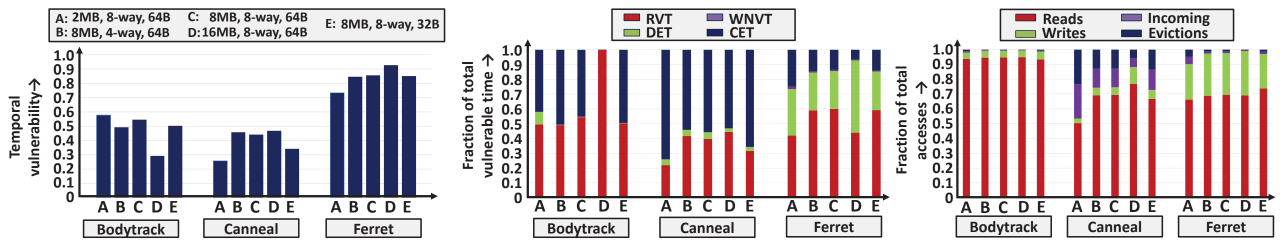

Figure 6: (a) Cache Vulnerabilities for Three Applications; (b) Different Vulnerability Components; (c) Cache Access Events.

4.1 Reliability Analysis of Different Cache Configurations

Figure 6 illustrates the analysis of temporal cache vulnerability, different vulnerability components, and their relationship to different cache events in different configurations for three representative benchmarks. Amongst the chosen applications, *Ferret* is highly vulnerable in almost all configurations because of large DET. A large number of cache blocks are evicted and replaced during the execution of *Canneal*. However, most of the evictions are that of clean blocks, reducing its temporal vulnerability ($40\%-50\%$). *Bodytrack* shows a large number of read accesses with few evictions, a major portion of them being clean. However, a large number of reads gives a RVT similar to *Ferret*, increasing its temporal vulnerability to 54% in the baseline (configuration C). The observations described above led us to explore the possibility of reconfiguring the cache to increase its reliability (i.e. reduce its vulnerability) while running different applications.

Configuring L2 Partition Size: On decreasing L2 partition size (e.g., when moving from configuration B, C or D to configuration A), *Ferret* and *Canneal* show a marked decrease in temporal vulnerability to 25.8% and 73.5%, respectively. The temporal vulnerability of *Bodytrack* increases slightly. *Ferret* also shows high reuse of cache lines with close to 30% of the lines showing active reuse. In configuration A, *Ferret* shows increased number of clean evictions (capacity misses) as compared to configuration B or C where most of the read accesses hit. *Canneal* shows high MPKI (1.475) in configuration A, but a large fraction of the misses are to clean blocks, increasing the CET and reducing the temporal vulnerability. *Bodytrack* experiences the least spatial vulnerability in configuration A (low reuse of cache lines). These idle lines contribute to increase in DET.

When increasing the $L2$ partition size (e.g., when moving from configuration B or C to configuration D), *Bodytrack* shows a significant decrease in temporal vulnerability ($54.57\% - 29.28\%$) by virtue of making efficient use of the available cache space. A large fraction of its reads hit in the cache, leading to an increase in RVT as compared to the other components, but their absolute magnitudes are still much smaller when compared to configurations A, B or C (since the accesses get satisfied in lesser time) leading to an overall decrease in the vulnerable period; see Figure 6(b). *Ferret* on the other hand, shows least reuse of cache blocks in configuration D. Idle cache lines contribute to increase in DET for *Ferret* and subsequently greater temporal vulnerability ($85.6\% - 92.8\%$) when compared to the baseline. Cache partition sizes greater than $8MB$, fail to provide significant reliability benefits for *Canneal* (high MPKI), because the reduction in DET is offset by a corresponding increase in RVT as can be seen in Figure 6(b).

Configuring Cache Associativity: Increasing the associativity of the $L2$ partition from 4 to 8 (i.e., when moving from configuration B to configuration C) results in a slight increase in the temporal vulnerability for *Bodytrack* ($49.2\% - 54.6\%$). For this benchmark the increased number of read hits increases RVT. Also, by virtue of having very few writes in its access pattern, it gets limited benefits in terms of $WNVT$. Reduction in the number of conflict misses greatly increases the reuse of blocks in *Ferret*, causing an increase in its spatial vulnerability ($32.51\% - 42.18\%$). *Canneal* also shows high reuse of cache lines, resulting in reduced DET (see Figure 6).

Configuring Line Size: Decreasing the line size (e.g., when moving from configuration C to E) causes a reduction in the vulnerability of *Bodytrack*, *Canneal* and *Ferret*. A smaller line size results in an increased number of evictions. In particular, *Canneal* shows a higher number of clean evictions increasing CET.

Algorithm 1 Run-time Configuration Management

```
Input: Library of cache configurations $C = \{C_1, C_2, ..., C_K\}$, Set of
    concurrently executing application threads $A = \{A_1, A_2, ..., A_M\}$,
    Set $P(i) = \{P_1, P_2, ..., P_N\}$ representing the run-time execution
    phases of application thread $A_i$, tolerable performance overhead $\tau$.
Output: Run-time coarse-grained mapping function ($CG : A \rightarrow C$), fine-
    grained mapping function ($FG : P(i) \rightarrow C$)
    Step 1: Inter-Application Cache Reconfiguration
 1: for $a \in A$ do \\Initialization
 2:   $CG(a) \leftarrow C_{baseline}$
 3:   $Vul(C_{baseline}) \leftarrow \sum_{p \in P(i)} CCF(p, C_{baseline})$
 4: $p_{loss} \leftarrow 0$
 5: for $a \in A$ do
 6:   $Vul_{min} \leftarrow Vul(C_{baseline})$
 7:   for $c \in C$ do \\find optimum cache configuration
 8:     $p_{loss} \leftarrow p_{loss} + \sum_{p \in P(i)} \eta(p, a, c)$
 9:     $Vul(c) \leftarrow \sum_{p \in P(i)} CCF(p, c)$
10:     if $Vul(c) \leq Vul_{min}$ && $p_{loss} < \tau$ then \\satisfy performance
    constraints
11:       $C_{opt} \leftarrow c$
12:       $Vul_{min}(a) \leftarrow Vul(C_{opt})$
13:   if $C_{opt} == C_{valid}$ then
14:     $CG(a) \leftarrow C_{opt}$
    Step 2: Intra-Application Cache Reconfiguration
15:   for $p \in P(a)$ do
16:     for $c \in C$ do
17:       $Vul(c) \leftarrow CCF(p, c)$
18:       $p_{loss} \leftarrow p_{loss} - \eta(p, a, CG(a)) + \eta(p, a, c)$
19:       if $Vul(c) \leq Vul_{min}(a)$ && $p_{loss} < \tau$ then
20:         $C_{opt}(p) \leftarrow c$
21:         $Vul_{min}(p) \leftarrow Vul(C_{opt}(p))$
22:     if $C_{opt}(p) == C_{valid}$ then
23:       $FG(a, p) \leftarrow C_{opt}(p)$
```

4.2 Vulnerability-Driven Configuration Manager

Our run-time configuration manager (Algorithm 1) chooses reliability-wise beneficial cache configurations that minimizes the total cache vulnerability for a set of concurrently executing application threads in the following two key steps. In the first step, each application thread is assigned a single cache configuration of minimum vulnerability, with no reconfiguration between different execution phases (i.e., inter-application reconfiguration) (Lines 1-14). The second step fine-tunes the cache configuration to meet the reliability requirements of the application in different phases (i.e., intra-application reconfiguration) (Lines 15-23). Each time a phase change is detected, Algorithm 1 is invoked to identify a minimum vulnerability cache configuration for the subsequent application phase. The reconfiguration overhead is 100-4000 cycles based on the number of cache configurations explored [24]. This overhead is negligible compared to the phase duration.

In the initialization phase, each application thread is initialized to run on the baseline cache configuration and the total vulnerability of the baseline configuration across all phases of the thread is estimated (Lines 1-3). Afterwards, it computes the performance loss incurred ($\eta(p, a, c)$) for different applications when run using different cache configurations (Line 8) and the vulnerability of each application thread for each cache configuration (Line 9). Under a user-provided tolerable performance overhead constraint, the cache configuration with the minimum vulnerability is selected (Lines 10-12). An additional check is performed to ensure that the chosen cache configuration is valid (Line 13). This check ensures that all applications receive at least their minimum required cache space according to their working set requirements. Afterwards, the configuration manager proceeds to determine the least vulnerable cache configuration for each phase of the application, provided the performance overhead is not exceeded (Lines 15-23).

5. EXPERIMENTAL SETUP

Figure 7 shows the detailed flow of our experimental evaluations. We extended the cycle-accurate Gem5 [25] full system simulator with the ability to (1) monitor the vulnerable periods during an application's lifetime, (2) identify the cache lines exhibiting high reuse, (3) support for cache reconfiguration and (4) fault injection. The inputs to the simulator include the parameters explained in Sections 3.1 and 3.4. Table 2 summarizes our experimental setup.

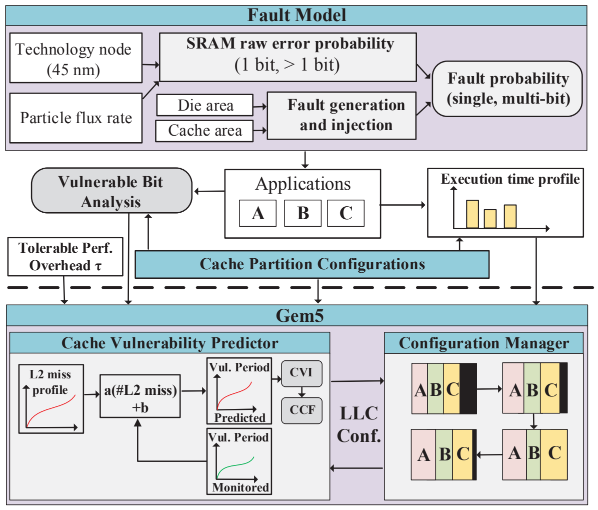

Figure 7: Experimental tool flow for R^{2}Cache

Following the norms of cache and architecture communities, we fast-forwarded the initialization and thread creation phases and collected statistics only for the parallel and thread synchronization phases of the execution of each application. We also profiled the number of $L2$ cache misses and used the collected traces to predict the vulnerability of the cache. We chose PARSEC multi-threaded benchmarks [11] with *simsmall* inputs for our evaluation and defined several application mixes (Table 3) to realize varying mixed workloads, where different applications share the last-level cache and have their own partitions. Different applications exhibit varying cache access behavior resulting in diversity of run-time scenarios. For this, we examined the sharing behavior between threads of the same application while running in isolation. *Canneal* and *Ferret* show high degrees of sharing between concurrently executing threads. They were grouped together with applications showing a comparatively lesser degree of sharing like *Blackscholes*, *Swaptions* and *Streamcluster* (see Mix $1, 2, 6, 9$ in Table 3). Mix $4, 10$ comprise of applications that compete for shared resources (i.e., *Canneal* and *Ferret*).

Table 2: Settings for the Experimental Setup

Core parameters		
Core type and OS		Alpha 21264 [26], Linux 2.6
Number of cores		4
Core frequency		2 GHz
Number of Data TLB entries		64
Number of Instruction TLB entries		48
Cache parameters		
Coherence	MOESI Snooping based	
Replacement policy	LRU based	
Private L1 cache	Data Cache	64kB, 2-way, 64B
	Instruction Cache	64kB, 2-way, 64B
	Hit/Response latency	2/2 cycles
	Number of MSHR's	4
Shared L2 cache	Cache	8MB, 8-way, 64B
	Hit/Response latency	20/20 cycles
	Number of MSHR's	20

Table 3: Application mixes from PARSEC [11]: (x,y,z) refers to the number of thread instances of each benchmark

Name	Type	Benchmarks
Mix 1	(4,4)	Canneal, Blackscholes
Mix 2	(8,4)	Canneal, Swaptions
Mix 3	(4,4)	Bodytrack, Streamcluster
Mix 4	(4,4)	Bodytrack, Ferret
Mix 5	(8,8)	Streamcluster, Bodytrack
Mix 6	(2,2)	x264, Streamcluster
Mix 7	(4,8)	Streamcluster, Bodytrack
Mix 8	(1,1,2)	Blackscholes, Bodytrack, Canneal
Mix 9	(8,4)	Canneal, Bodytrack
Mix 10	(8,8)	Ferret, Canneal

5.1 Comparison Partners

We compare the proposed R^{2}Cache architecture with the following state-of-the-art techniques:

(1) Vulnerability-Reducing Non-Reconfigurable Caches based on smart early write-back (EWB) [10] that reduce the temporal vulnerability of the cache by virtue of reducing DET, but its configuration cannot be adapted at run-time. We assume our baseline cache to be of write-through type only for the purpose of comparison of vulnerability savings with these techniques.

(2) Performance-Maximizing Reconfigurable Cache Architecture (PMR) [27], that partitions the shared cache between multiple applications of a workload, so that the overall performance is improved.

(3) Write-through Cache with Minimum Vulnerability - Best Case (WT-BC): For the sake of completeness, we also present a comparison when the best chosen configuration is realized in a write-through cache. It corresponds to the ideal case best achievable vulnerability without a soft error protection mechanism since it has no dirty eviction time. This serves as the vulnerability for the ideal case which our approach strives to meet.

Execution time is used as the performance metric in our evaluations. While evaluating R^{2}Cache, we report all performance overheads with respect to the execution time using PMR. Note, a last-level write-through cache is prohibitively expensive both in terms of performance and network bandwidth and is not used in practice. It is used only to demonstrate what our R^{2}Cache can achieve compared to the best case vulnerability.

6. RESULTS AND DISCUSSION

6.1 Vulnerability compared to State-of-the-Art for Different Applications

This section details the vulnerability savings obtained by our approach under different user-provided performance constraints.

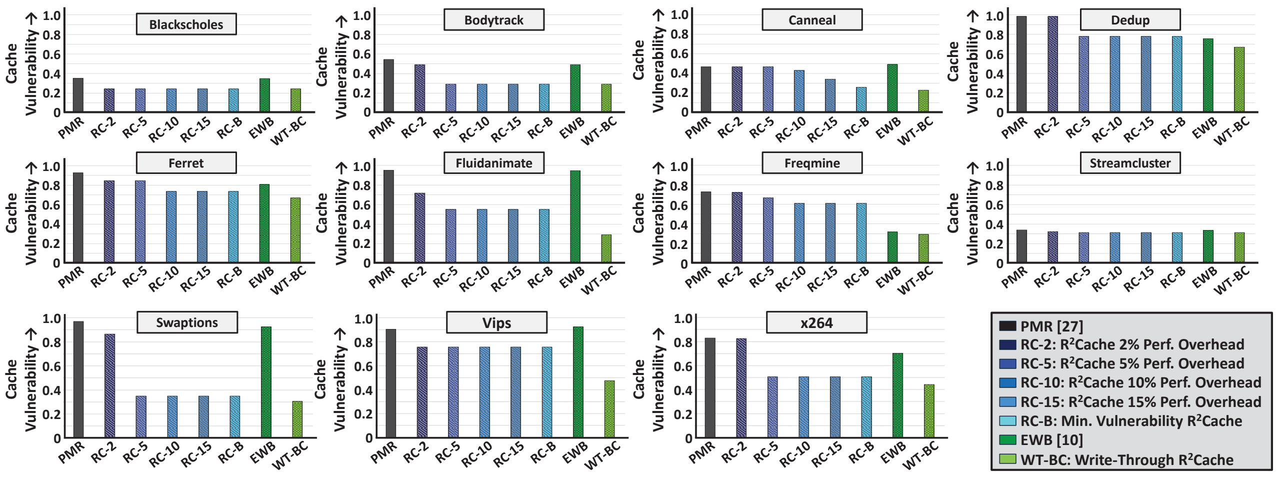
Figure 8: Comparing cache vulnerability reductions of R^{2}Cache to different state-of-the-art caches for different applications

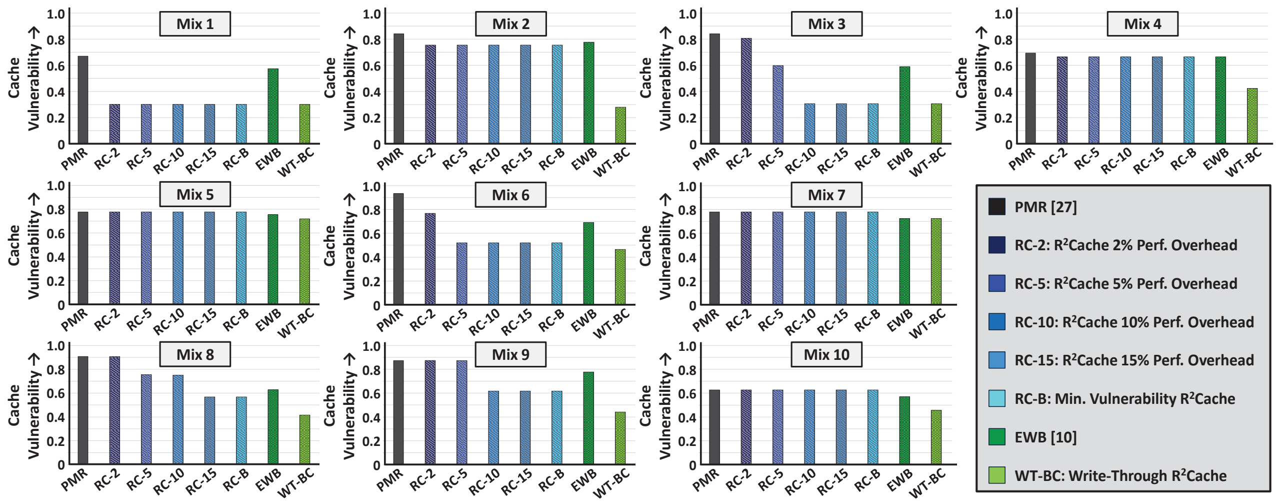
Figure 9: Comparing cache vulnerability reductions of R^{2}Cache to different state-of-the-art caches for different application mixes

Figure 8 shows the cache vulnerability reduction results for our R^{2}Cache for different performance constraints (2%, 5%, 10%, 15%, >15%) compared to state-of-the-art described in Section 5.1. The results are presented for individual multi-threaded applications of the PARSEC benchmark suite. The performance-reliability trade-off can be observed in Figure 8 where increasingly reliable cache configurations are identified by our manager when larger performance overheads are allowed. Notably, *Canneal* (a memory intensive benchmark) can be executed on a number of reliably efficient cache configurations with vulnerability ranging from 46.8% to 25%, each coming at a different performance cost. Also, the vulnerability of the best configuration chosen by R^{2}Cache is very close to the ideal case vulnerability, i.e., 22.95% provided by WT-BC. Also, many applications (*Bodytrack*, *x264* etc.) show large improvements in vulnerability at low performance overheads (2%-5%). A slight increase in performance overhead can be related to a corresponding increase in the number of cache misses and possibly frequent cache line evictions. These events contribute to the decrease in RVT and DET, respectively. The multiplicative effect of these changes greatly reduces cache vulnerability. In particular, *Swaptions* shows 57.56% and 62.13% vulnerability savings when compared to EWB and PMR, respectively, under 5% overhead. These can be attributed to the increase in the number of write hits when a large size partition (e.g., 16MB) was selected at run-time. These write hits increase $WNVT$ and decrease DET, resulting in reduced temporal vulnerability for the cache. *Bodytrack* also shows close to 25% vulnerability reductions when compared to other approaches. Here again, the run-time manager chooses a large partition that reduces capacity misses. With greater number of reads and writes hits in the cache (mostly in quick succession), the contribution of DET to the overall vulnerable period is reduced. $WNVT$ also increases for the same reason. Similarly, for *Fluidanimate*, greater than 40% reduction in cache vulnerability is observed under 5% performance overhead compared to both EWB and PMR. This is because of an increase in the number of clean block evictions when a small size partition was chosen at run-time.

Another observation is that the cache vulnerability reduction saturates beyond a given performance constraint (variable across different applications) because of a limited number of partition configurations and vulnerability optimization potential. Figure 8 shows that, for most applications (*Blackscholes*, *Streamcluster*, *Ferret* etc.), the vulnerability of the best R^{2}Cache partition (i.e., RC-B) is close to the ideal WT-BC case. *Freqmine* and *Vips*, however have a large number of dirty cache blocks even in the best partition selected by our technique leading to limited gains when compared to the EWB [10].

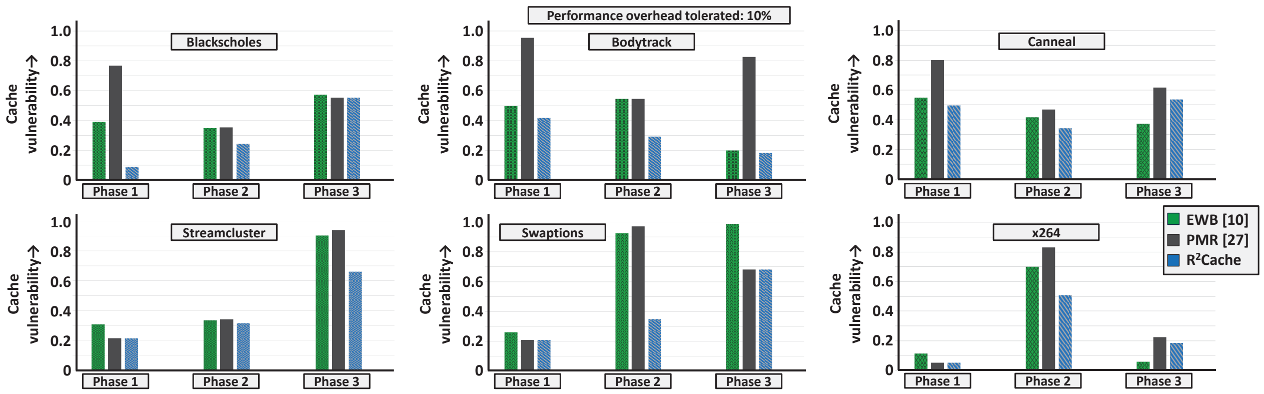

Figure 10: Detailed phase-level vulnerability comparison for different applications

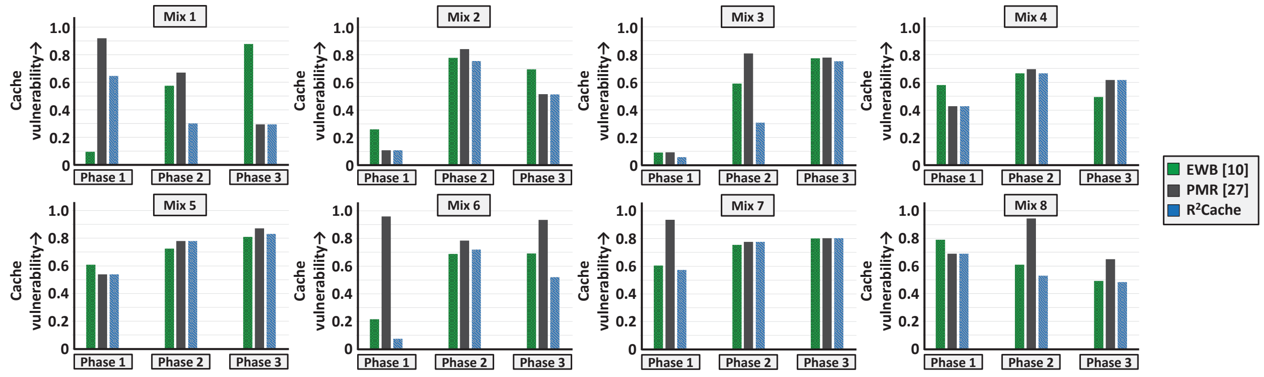

Figure 11: Detailed phase-level vulnerability comparison for different application mixes

6.2 Vulnerability compared to State-of-the-Art for Different Mixes

Figure 9 compares the cache vulnerability of different approaches for different application mixes. Our R^{2}Cache demonstrates vulnerability reductions for almost all mixes. For instance, R^{2}Cache performs well for Mix 9, with a 2MB partition size for *Canneal* providing 15.9% and 25.6% vulnerability savings when compared to EWB and PMR. Mix 9 shows high MPKI (0.55) in this configuration and the greater number of frequent evictions brings about a reduction in the vulnerable DET period. Mix 6 also shows a similar behavior. However, EWB [10] performs slightly better for Mix 7 and 10 mainly because it reduces dirty eviction time. On account of the large DET obtained even in the best configuration provided by our run-time manager, the reliability improvements provided by our Vulnerability-Driven Cache Reconfiguration Manager are limited.

6.3 Detailed Phase-level Vulnerability Comparison for Different Applications

Figure 10 illustrates that the vulnerability of the cache partitions also shows significant variation between phases of individual applications. Since the number of distinct phases during the execution of PARSEC benchmarks is small [28], we show the results for three prominent phases, after ensuring that the cache has sufficiently warmed up and enough statistics have been collected for a reasonable estimation of vulnerability. R^{2}Cache utilizes intra-application reconfiguration to exploit this phase behavior of applications and obtain maximum reliability benefits. *Canneal*, which shows high MPKI, benefits from using smaller size partitions in Phase 2, as frequent evictions lead to decrease in DET. However these evictions come at a performance cost as they affect cache locality. *x264* also shows similar characteristics with a vulnerability reduction of 39% compared to PMR and 28% compared to EWB in Phase 2. However, larger cache partitions show lesser temporal vulnerability for applications like *Blackscholes* whose access pattern shows a larger of reads in quick successions. In a smaller cache partition, frequent reads also lead to frequent eviction of clean cache blocks. These cannot significantly increase CET as compared to a larger cache partition where evictions are less frequent.

6.4 Detailed Phase-level Vulnerability Comparison for Different Mixes

Figure 11 shows the cache vulnerability of R^{2}Cache in different phases of execution of application mixes. With application mixes exhibiting different cache vulnerabilities across phases, significant vulnerability reductions are obtained for Mix 3, especially in Phase 2 for which the EWB and PMR caches show high vulnerability. The large-sized cache partition (16MB) chosen by the manager for Phase 2 of Mix 3 provides 28.31% and 50.03% vulnerability savings when compared to EWB and PMR, respectively. This can be attributed to reduced DET. Significant vulnerability improvements are not obtained for Phase 3 of Mix 4 and Mix 5 and Phase 1 of Mix 1 when compared to EWB because of the limitation that our reconfiguration cannot completely eliminate the dirty eviction time due to the presence of a large number of idle dirty blocks especially during the beginning and end of an application's execution.

Summary of Vulnerability Reduction Results: Overall, the proposed R^{2}Cache outperforms state-of-the-art vulnerability reducing non-reconfigurable caches [10] and performance maximizing reconfigurable cache techniques [27]. On average, R^{2}Cache provides 12.6% and 24.3% vulnerability savings when compared to EWB and PMR, respectively, averaged across individual applications as well as application mixes under 10% performance overhead. *Note that these average savings are already significant because we have done full-system cycle accurate simulations under mixed* *workloads* that create load on the cache and averaged over numerous cases. When looking into details, for some application phases, R^{2}Cache provides vulnerability savings of >60%. However, the net vulnerability savings are highly dependent upon the cache access patterns of different applications. The vulnerability savings achieved by R^{2}Cache are only 12.2% away from that obtained by the ideal-case WT-BC.

6.5 Evaluation of Error Distribution for Different Applications

Here we include the results of our fault injection experiments and $P_{error}(fr)$ profile for different applications. Figure 12 shows the results of our Monte Carlo fault injection campaigns on the last-level cache, using the multi-bit error distribution model from [18]. Assuming that the Baseline cache (see Section 2) is protected against single bit errors using SEC-DED, we present the results only for the multi-bit error scenarios. During each program run, we inject a single fault at random bit positions of cache, ensuring that the faults occur in the cache region used actively by the application. To compute $P_{error}(fr)$ we consider the fraction of simulation runs that lead to application failures (program crashes or timeout). As can be seen from Figure 12, *Swaptions* shows high failure rates among the applications investigated. *Swaptions* used in financial analysis spends a significant fraction of its execution computing random numbers, critical to its execution. Faulty numbers often lead to timeout for this application. *Blackscholes* shows good error masking properties, since its input set uses plenty of redundant values. The human body tracking algorithm used in *Bodytrack* also has large code sections with inherent redundancy. Although *Canneal* uses incremental simulated annealing to minimize chip routing cost, it is unable to tolerate large multi-bit errors, especially when fault values are loaded into the instruction cache.

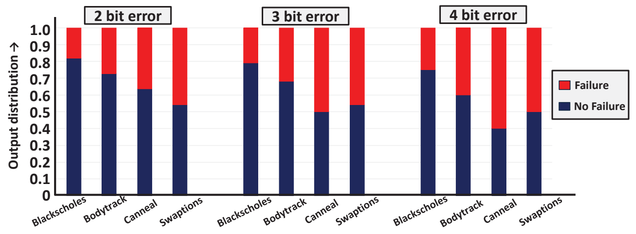

Figure 12: Evaluation of error distribution for representative applications

6.6 Performance Overhead Analysis

We analyzed the performance overhead of our approach when compared to PMR and the Baseline cache configurations. In addition, a comparative evaluation of the performance overhead of different approaches when soft-error protection schemes like SEC-DED [29, 30] are employed to the last-level cache is shown in Figure 13. *It should be noted that our approach is orthogonal to ECC-based correction* and that the ECC-based caches combined with reconfiguration can perform better. For the case of multi-bit errors our approach can serve to reduce cache vulnerability to soft errors where ECC fails.

We consider a pipelined SEC-DED implementation, similar to that in [29] that incurs an additional latency of 1 cycle per access. In Figure 13 we show only R^{2}Cache configurations that can be attained within a performance overhead of 20%. This is reasonable since vulnerability reduction saturates at higher performance overheads for the chosen applications as discussed in Section 6.1.

In the R^{2}Cache architecture there is a performance degradation because of the time taken (1) to flush dirty cache lines to main memory, (2) to identify reliability-wise efficient cache partitions and (3) to set appropriate bits in the configuration register needed for reconfiguration. In addition, another overhead is due to execution of the application in a performance-wise sub-optimal configuration. The reconfiguration overhead of our approach is 100-4000 cycles (as in [24]). This overhead is negligible compared to the execution time of an application phase. Even if it is not the case, the user-tolerable performance overhead constraint in our run-time manager considers this overhead to limit the number of reconfigurations that can take place.

In Figure 13, ECC1+PMR refers to the performance-maximizing reconfigurable cache architecture with SEC-DED protection and ECC-1+Baseline and ECC-1+RC refer to the SEC-DED protected baseline cache and R^{2}Cache, respectively. From Figure 13 it can be seen that the performance of R^{2}Cache is comparable to ECC-1+PMR and even better than ECC-1+Baseline for most applications. With 15-20% tolerable overhead, the cache vulnerability to multi-bit errors can be reduced for even applications with high MPKI like *Canneal*.

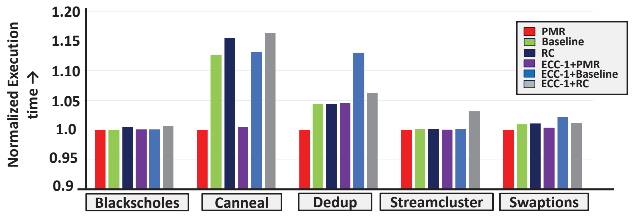

Figure 13: Normalized execution time for representative applications

6.7 Last-Level Cache Energy Consumption

In this section, the last-level cache energy consumption of our technique (both with and without error protection) is shown. We use McPAT [31] to obtain the power consumed by the last-level cache while executing representative PARSEC applications. This includes both the static power (gate leakage and sub-threshold leakage) as well as dynamic power. Our run-time reliability manager chooses smaller-sized cache partitions for *Blackscholes*, *Dedup* and *Canneal*. The remainder of the cache can be switched-off to save power leading to energy savings as shown in Figure 14. The results are normalized with respect to PMR. Larger cache partitions are chosen for *Streamcluster* and *Swaptions* as they are reliability-wise more efficient. However, this leads to an increase in power consumption.

Similar to [32], we use an additional 0.4nJ dynamic energy per cache access for SEC-DED protection in a 45nm process. From Figure 14 it can be observed that the savings are highly dependent on the partition sizes chosen for individual applications.

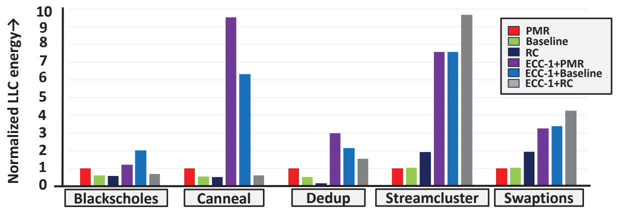

Figure 14: Normalized last-level cache energy for representative applications

6.8 Hardware Overhead

The additional hardware overhead involved in the proposed technique is briefly discussed here. As a part of the reconfiguration control, we require a 6 bit configuration register for 5 L2 partition sizes, 4 L2 associativities and 2 L2 line sizes. In order to evaluate the area overhead required by the predictor, we implemented the predictor in VHDL using a RAM with 32 entries for the prediction coefficients for each application, a 4-stage multiplier and an adder. In a Virtex-4-vlx160 FF1148 FPGA it requires 526 LUTs and 264 slices corresponding to 16,122 gate equivalents. The prediction time of 6 cycles is negligible compared to the execution time of an application phase. The SEC-DED implementation requires an additional 12.5% area overhead (8 bits for every 64 bit cache word).

7. CONCLUSION

We presented a novel reliability-aware reconfigurable cache architecture R^2Cache that adapts the cache parameters for concurrently executing multi-threaded workloads at run-time in order to minimize their vulnerabilities to soft errors. To enable an efficient design, we developed a cache vulnerability model that jointly accounts for spatial and temporal vulnerabilities and performed cache vulnerability analysis for different configurations. At run time, vulnerabilities are estimated for different applications and an appropriate configuration is selected for different applications and different execution phases within an application. On average, R^2Cache provides up to 15.6% and 28.6% vulnerability savings when compared to state-of-the-art, averaged across individual applications as well as application mixes. Caches protected with ECC can use our approach to achieve a reduction in vulnerability towards multi-bit errors. R^2Cache enables performance constrained reliability mitigation in large-sized shared last-level caches. Our approach can also be beneficial for soft real-time systems, where task start and preemption times are unknown. This would require keeping track of the phase in which each application was preempted and additional configuration decisions when a new task enters the system or when a task completes.

8. ACKNOWLEDGMENT

This work was supported in parts by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu). We would like to thank Anu Gupta, Nitin Chaturvedi and Sujay Narumanchi, EEE, BITS-Pilani for their insights and discussions.

References

[1] R. C. Baumann. “Radiation-induced soft errors in advanced semiconductor technologies”. In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316.

[2] J. Henkel et al. “Reliable On-Chip Systems in the Nano-Era: Lessons Learnt and Future Trends”. In: *Design Automation Conference (DAC)*. 2013.

[3] W. Zhang, S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam. “ICR: In-Cache Replication for Enhancing Data Cache Reliability”. In: *International Conference on Dependable Systems and Networks (DSN)*. 2003, pp. 291–300.

[4] L. Li et al. “Soft error and energy consumption interactions: a data cache perspective”. In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2004, pp. 132–137.

[5] W. Zhang. “Computing Cache Vulnerability to Transient Errors and Its Implication”. In: *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*. 2005, pp. 427–435.

[6] S. S. Mukherjee et al. “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor”. In: *International Symposium on Microarchitecture (MICRO)*. 2003, pp. 29–42.

[7] S. Wang, J. S. Hu, and S. G. Ziavras. “On the Characterization and Optimization of On-Chip Cache Reliability against Soft Errors”. In: *IEEE Trans. Computers* 58.9 (2009), pp. 1171–1184.

[8] A. Haghdoost, H. Asadi, and A. Baniasadi. “System-Level Vulnerability Estimation for Data Caches”. In: *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. 2010, pp. 157–164.

[9] Y. Zou and S. Pasricha. “HEFT: A hybrid system-level framework for enabling energy-efficient fault-tolerance in NoC based MPSoCs”. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2014, 4:1–4:10.

[10] R. Jeyapaul and A. Shrivastava. “Enabling energy efficient reliability in embedded systems through smart cache cleaning”. In: *ACM Trans. Design Autom. Electr. Syst.* 18.4 (2013), p. 53.

[11] C. Bienia et al. “The PARSEC benchmark suite: characterization and architectural implications”. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 72–81.

[12] C. Zhang, F. Vahid, and W. A. Najjar. “A Highly-Configurable Cache Architecture for Embedded Systems”. In: *International Symposium on Computer Architecture (ISCA)*. 2003, pp. 136–146.

[13] C. Zhang, F. Vahid, and R. L. Lysecky. “A self-tuning cache architecture for embedded systems”. In: *ACM Trans. Embedded Comput. Syst.* 3.2 (2004), pp. 407–425.

[14] D. H. Albonesi. “Selective Cache Ways: On-Demand Cache Resource Allocation”. In: *J. Instruction-Level Parallelism* 2 (2000).

[15] K. T. Sundararajan, T. M. Jones, and N. P. Topham. “RECAP: Region-Aware Cache Partitioning”. In: *IEEE International Conference on Computer Design (ICCD)*. 2013, pp. 294–301.

[16] W. Wang, P. Mishra, and S. Ranka. “Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems”. In: *Design Automation Conference (DAC)*. 2011, pp. 948–953.

[17] C. L. Chen and M. Y. B. Hsiao. “Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review”. In: *IBM Journal of Research and Development* 28.2 (1984), pp. 124–134.

[18] A. Dixit and A. Wood. “The impact of new technology on soft error rates”. In: *Reliability Physics Symposium (IRPS), IEEE International*. 2011, 5B.4.1–5B.4.7.

[19] S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt. “The Soft Error Problem: An Architectural Perspective”. In: *International Conference on High-Performance Computer Architecture (HPCA-11)*. 2005, pp. 243–247.

[20] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. “Reliable software for unreliable hardware: embedded code generation aiming at reliability”. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2011, pp. 237–246.

[21] T. Sherwood et al. “Discovering and Exploiting Program Phases”. In: *IEEE Micro* 23.6 (2003), pp. 84–93.

[22] S. Srikantaiah et al. “MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2011, pp. 231–242.

[23] L. Duan, B. Li, and L. Peng. “Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics”. In: *International Conference on High-Performance Computer Architecture (HPCA-15)*. 2009, pp. 129–140.

[24] M. Rawlins and A. Gordon-Ross. “A Cache Tuning Heuristic for Multicore Architectures”. In: *IEEE Trans. Computers* 62.8 (2013), pp. 1570–1583.

[25] N. Binkert et al. “The gem5 simulator”. In: *SIGARCH Comput. Archit. News* (Aug. 2011), pp. 1–7.

[26] E. J. McLellan and D. A. Webb. “The Alpha 21264 Microprocessor Architecture”. In: *International Conference on Computer Design (ICCD)*. 1998.

[27] M. K. Qureshi and Y. N. Patt. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches”. In: *International Symposium on Microarchitecture (MICRO)*. 2006, pp. 423–432.

[28] A. Sembrant, D. Black-Schaffer, and E. Hagersten. “Phase behavior in serial and parallel applications”. In: *International Symposium on Workload Characterization (IISWC)*. 2012, pp. 47–58.

[29] A. R. Alameldeen et al. “Energy-efficient cache design using variable-strength error-correcting codes”. In: *International Symposium on Computer Architecture (ISCA)*. 2011, pp. 461–472.

[30] M. K. Qureshi and Z. Chishti. “Operating SECDED-based caches at ultra-low voltage with FLAIR.” In: *DSN*. 2013, pp. 1–11.

[31] S. Li et al. “McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *International Symposium on Microarchitecture (MICRO)*. 2009, pp. 469–480.

[32] D. H. Yoon and M. Erez. “Memory mapped ECC: low-cost error protection for last level caches”. In: *International Symposium on Computer Architecture (ISCA)*. 2009, pp. 116–127.