Reliability-Aware Adaptations for Shared Last-Level Caches in Multi-Cores

FLORIAN KRIEBEL, Chair for Embedded Systems, Karlsruhe Institute of Technology
SEMEEN REHMAN, Chair for Processor Design, CFAED, TU Dresden
ARUN SUBRAMANIYAN, SEGNON JEAN BRUNO AHANDAGBE,
MUHAMMAD SHAFIQUE, and JÖRG HENKEL, Chair for Embedded Systems, Karlsruhe Institute of Technology

On account of their large footprint, on-chip last-level caches in multi-core systems are one of the most vulnerable components to soft errors. However, vulnerability to soft errors highly depends on the configuration and parameters of the last-level cache, especially when executing different applications concurrently. In this article we propose a novel reliability-aware reconfigurable last-level cache architecture (R^2Cache) and cache vulnerability model for multi-cores. R^2Cache supports various reliability-wise efficient cache configurations (i.e., cache parameter selection and cache partitioning) for different concurrently executing applications. The proposed vulnerability model takes into account the vulnerability of both the *data* and *tag* arrays as well as the active cache area for applications in different execution phases. To enable runtime adaptations, we 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 runtime reliability manager reconfigures the cache such that, for the next execution epoch, the total vulnerability for all concurrently executing applications is minimized under user-provided tolerable performance/energy overheads. 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 vulnerability-minimizing and reconfigurable caches, the proposed architecture provides 35.27% and 23.42% vulnerability savings, respectively, when averaged across numerous experiments, while reducing the vulnerability by more than 65% and 60%, respectively, for selected applications and application phases.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; **Multicore architectures;**

Additional Key Words and Phrases: Reliability, soft errors, vulnerability, cache, multi-cores, modeling, optimization, performance, energy

ACM Reference Format:
Florian Kriebel, Semeen Rehman, Arun Subramaniyan, Segnon Jean Bruno Ahandagbe, Muhammad Shafique, and Jörg Henkel. 2016. Reliability-aware adaptations for shared last-level caches in multi-cores. ACM Trans. Embed. Comput. Syst. 15, 4, Article 67 (August 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2961059

Authors' addresses: F. Kriebel, A. Subramaniyan, S. J. B. Ahandagbe, M. Shafique (corresponding author), and J. Henkel, Chair for Embedded Systems (CES), ITEC, Karlsruhe Institute of Technology (KIT), Building 07.21, Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany; emails: florian.kriebel@kit.edu, sarun2006k1in@gmail.com, segnon.ahandagbe@student.kit.edu, muhammad.shafique@kit.edu, henkel@kit.edu; S. Rehman, Chair for Processor Design, CFAED, Technische Universität Dresden, Georg-Schumann-Str. 7A, 01062 Dresden, Germany; email: semeen.rehman@tu-dresden.de.

DOI: http://dx.doi.org/10.1145/2961059

1. INTRODUCTION AND RELATED WORK

Due to small feature sizes, faster clocks and low-operating-voltage advanced multi-cores are highly susceptible to soft errors [Baumann 2005]. Soft errors are transient faults induced by energetic particles from cosmic rays or packaging material. These soft errors manifest themselves as spurious bit-flips in the underlying hardware and can lead to application program failures and silent data corruptions (SDCs). In modern multi-cores, 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 [Zhang et al. 2003]. These errors can propagate to the Central Processing Unit (CPU) and other pipeline components (register file, issue queue, etc.), leading to SDCs (i.e., incorrect output) or even application/system failures. As a result, a large body of research has investigated techniques to ensure data integrity in caches [Li et al. 2004; Zhang 2005]. In the following, we discuss cache trends and the associated reliability-related issues, followed by the state of the art.

1.1. Cache Trends and the Associated Reliability Issues

Figure 1(a) shows the trend of increasing size for last-level caches with every technology generation in almost all modern architectures. Last-level cache sizes vary across a broad range based on the application domain (e.g., 512kB for a Pentium II desktop to 45MB for a Xeon E7 server class processor and even 128MB for a Power 8 processor). With such a large footprint, last-level caches occupy more than half of the die area in modern high-end processors (e.g., IBM Power 8 [IBM 2015] and Intel Itanium 2 [Intel 2015]). A consequence of increasing size is the corresponding increase in the cost of uniform SEC-DED ECC protection, as shown in Figure 1(b). Several recent works have projected the increase in multi-bit error trends for the last-level cache [Wilkening et al. 2014], necessitating the need for multi-bit error correction. However, the overheads of multi-bit error correction are even larger [Kim et al. 2007]. *Therefore, there exists a need to develop novel soft error mitigation techniques for both ECC-protected caches as well as unprotected caches in order to alleviate the large protection overheads.*

While a majority of the cache is used for storing data, there is an associated non-negligible tag overhead, especially for large-sized caches. For example, considering an 8MB, 8-way set associative cache with 64B line size, the tag overhead is 192kB. This is about $3x$ the area occupied by typical L1 caches in modern processors (64kB). Therefore, not only the cache data but also the tag bits are highly vulnerable to soft errors on account of their large footprint. Different applications show different cache locality characteristics and do not access all cache lines uniformly. Furthermore, even within the same application, the cache locality varies in different phases of execution. This variability in cache line utilization is also important to be considered in the vulnerability model, as we will demonstrate in Section 4.

1.2. State of the Art and Its Limitations

For designing cost-constrained reliable caches, two key challenges are (1) soft error vulnerability analysis and (2) soft error mitigation mechanisms.

Soft Error 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 soft error vulnerability for caches were developed in Zhang et al. [2003], Mukherjee et al. [2003], Wang et al. [2009], and Haghdoost et al. [2010]. Zhang et al. [2003] proposed the *Cache Vulnerability Factor (CVF)* metric, which is based on the concept of the Architectural Vulnerability Factor (AVF) [Mukherjee et al. 2003]. CVF is defined as the probability that a soft error in the cache propagates to the CPU or lower levels of the memory hierarchy. It characterizes the lifetime of a cache line into vulnerable and non-vulnerable phases. Recently, Wang et al. [2009]

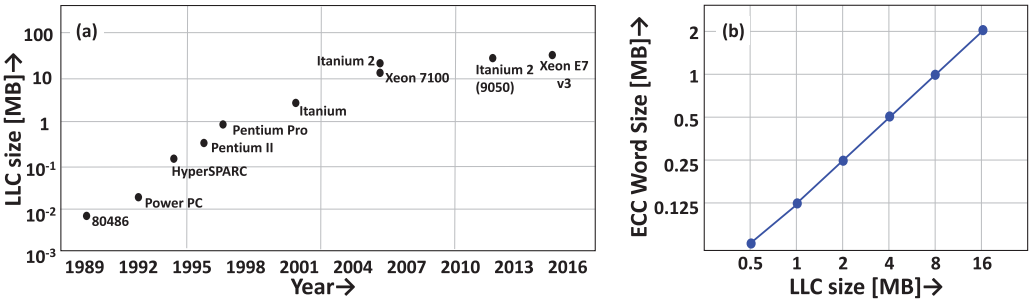

Fig. 1. (a) Cache size trends. (b) ECC size trends.

and Haghdoost et al. [2010] 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 mainly aim at reducing the *Temporal Vulnerability*, that is, 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*, that is, 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 Zou and Pasricha [2014], a runtime adaptation technique is proposed for NoCs that is based on a lightweight vulnerability prediction mechanism. However, this work does not target cache vulnerability estimation and optimization.

Soft Error Mitigation: With regard to soft error mitigation, *Write-Through (WTC)* and *Early Write-Back (EWB)* caches have been explored in the literature [Li et al. 2004] 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 Jeyapaul and Shrivastava [2013] 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 runtime estimation due to their intensive computations.

Therefore, there is a need for an accurate cache vulnerability modeling, considering both data and tag vulnerability, and a light-weight online estimation technique that accounts for both spatial and temporal vulnerabilities of concurrently executing applications under changing cache configurations.

Dynamic Cache Reconfiguration: Our analysis in Section 2 illustrates that dynamically reconfiguring the cache can impact the vulnerability of concurrently-executing applications. The state of the art in reconfigurable caches have primarily explored configurations w.r.t. performance and energy consumption [Zhang et al. 2003, 2004]. These approaches provide basic infrastructure as discussed in Section 3. Recently, adapting partitions for last-level caches has also gained interest where, instead of reconfiguring the whole cache, private and shared partitions are used [Sundararajan et al. 2013a; Wang et al. 2011].

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

1.3. Novel Contributions

In this article, we propose a *novel Reliability-Aware Reconfigurable Cache Architecture (R^2Cache)*[1] (see Section 3) that dynamically reconfigures different last-level cache parameters (like cache partition sizing, line sizing, and associativity adaptation) at runtime for different applications depending on their access patterns. It optimizes the vulnerabilities (considering both data and tag vulnerability) in a two-step process: (1) *Inter-Application Reconfiguration*, that is, adaptation across different cache partitions of different applications, and (2) *Intra-Application Reconfiguration*, that is, adaptations between different execution 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 Cache Vulnerability Predictor that estimates the data and tag vulnerability of an application (phase) considering cache utilization and performance metrics, like cache miss rate in terms of Misses-Per-Kilo-Instructions (MPKI), Instructions-Per-Cycle (IPC), and so on, at runtime (see Section 4.6). Towards this end, we also propose a *novel cache vulnerability quantification model* as a joint function of spatial and temporal vulnerabilities (see Section 4.3).

(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 5). The goal is to minimize the total vulnerability for all concurrently executing applications under user-provided tolerable performance/energy overheads.

Configuration Space: We also present the reliability-performance-energy configuration space for different applications and identify Pareto-frontiers to enable selection of cache configurations that trade off between different design objectives.

Cache Vulnerability Analysis: To devise an efficient configuration management technique, it is important to understand (1) the temporal and spatial vulnerability variations for different applications and their different execution phases, (2) the tag vulnerability analysis, and (3) the influence of different cache configurations on the overall cache vulnerabilities due to different cache operations (like read, eviction, etc.). Towards this end, this article presents a *comprehensive analysis of cache vulnerability for different applications, different execution phases, and different cache configurations* in Sections 2 and 5.1. Our analysis illustrates that adapting different parameters of the last-level cache can significantly impact the vulnerability for different applications and even during different execution phases. This analysis is leveraged for designing and managing the R^2Cache architecture.

Applicability in ECC-Protected Caches: Reliability-aware cache reconfiguration can also be applied in conjunction with error correcting codes (ECCs) like SEC-DED [Chen and Hsiao 1984] (Single Error Correcting-Double Error Detecting) to improve reliability in multi-bit error scenarios or in cases where only some of the

[1]Compared to Kriebel et al. [2015], we enhanced the contributions by additionally considering the cache tags, extending the vulnerability model as well as the vulnerability estimation. Moreover, energy is taken into account as an additional constraint when identifying Pareto-optimal cache configurations and in the vulnerability-driven configuration management.

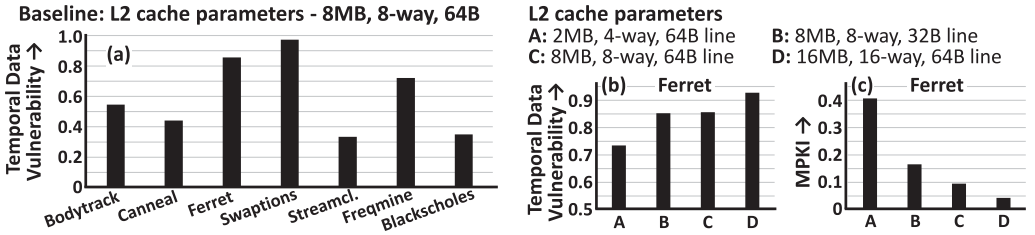

Fig. 2. (a) Varying vulnerabilities of different applications from PARSEC [Bienia et al. 2008] for the baseline case. ((b) and (c)) Changing vulnerabilities and cache misses for *Ferret* application under different configurations.

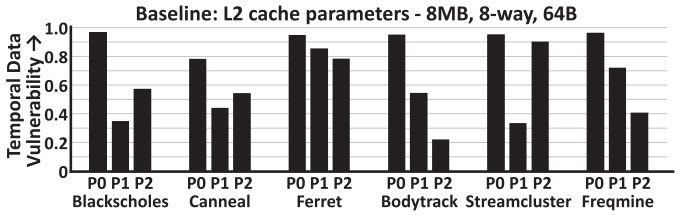

Fig. 3. Data vulnerability for different application phases.

cache partitions are ECC-protected due to area constraints. In Section 7.7 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. MOTIVATIONAL ANALYSIS OF CACHE VULNERABILITY

For the motivational analysis, we used different applications from the PARSEC benchmark suite. In the following, we perform the following different analyses that provide important insights for design and optimization: (1) varying data vulnerabilities for different applications and impact of cache misses, (2) varying data vulnerability across different execution phases of an application, (3) active cache area for different applications that determine the spatial vulnerability, (4) active cache area for different execution phases, and (5) tag vulnerability analysis for different applications and understanding the impact of changing cache configurations.

2.1. Varying Vulnerabilities for Different Applications and Impact of Cache Misses

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 2(a). Moreover, our analysis in Figures 2(b) and (c) for the *Ferret* application shows that different configurations provide a tradeoff between vulnerability and performance (in terms of cache misses given as MPKI).

Observation-1: Dynamic parameter adaptation for last-level caches can be leveraged to decrease the soft error vulnerability for concurrently executing applications under user-provided tolerable performance/energy overheads.

2.2. Varying Data Vulnerability Across Phases

Figure 3 shows the variation in temporal vulnerability of last-level cache data for three execution phases (P0, P1, and P2) of PARSEC applications. All applications show significant variation in *temporal vulnerability* across phases due to change in cache usage patterns and locality of accesses. The high vulnerability in phase P0 can be attributed to the large number of dirty cache lines used during initialization that are

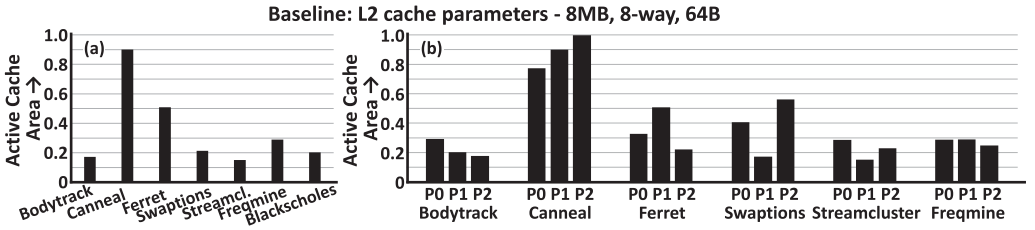

Fig. 4. Active cache area: (a) average value for different applications and (b) average value for different phases.

written back at the end of the phase. This makes it highly likely for corrupt data to propagate to memory in case of errors. *Blackscholes* and *Streamcluster* show less vulnerability in phase P1 because they tend to cluster their read accesses to certain cache regions, resulting in frequent evictions of clean cache lines and allocation of new lines. *Ferret* on the other hand, spreads its accesses across the cache, resulting in fewer evictions and large occupancy time for cache lines increasing its vulnerability.

Observation-2: The above discussion illustrates that intra-application cache reconfiguration may be leveraged to minimize vulnerability in different execution phases.

2.3. Active Cache Area for Different Applications

As discussed before, applications do not access the entire cache during different execution phases. To capture this effect, we define *spatial vulnerability* as the ratio of the unique cache lines referenced by the application in a phase to the total number of cache lines. This metric has also been found to be more effective for measuring cache utilization than miss-rate-based estimations [Srikantaiah et al. 2011]. Figure 4(a) shows the active cache area used by applications, averaged across different execution phases. In Figure 4, it can be seen that both *Canneal* and *Ferret* use more than 50% of the cache on average. *Blackscholes*, *Streamcluster*, and *Swaptions* confine their accesses to small regions of the cache, leading to a low spatial vulnerability to soft errors.

Observation-3: Besides the temporal vulnerability (i.e., cache vulnerability due to different time-wise separated cache access events), the notion of spatial vulnerability (i.e., vulnerability due to active cache regions) also needs to be taken into account.

2.4. Active Cache Area for Different Phases

The active cache area for applications, and hence the spatial vulnerability, also changes in different execution phases. Figure 4(b) shows the variation in percentage of active cache area for different application phases. *Canneal* shows high spatial vulnerability in all execution phases. *Canneal* is a memory-intensive application and makes use of all available cache space.

Observation-4: Consideration of spatial vulnerability at the granularity of phases can be leveraged during intra-application cache reconfiguration to minimize the total vulnerability in different execution phases.

2.5. Tag Vulnerability Analysis for Different Applications and Different Cache Configurations

Figure 5(a) shows the variations in tag vulnerability across applications. We consider up to four spatial multi-bit errors while estimating tag vulnerability (see model details and estimation in Section 4). It can be seen that the tag vulnerability is considerably lower than the data vulnerability for all applications. This can be attributed to the following: (1) errors in tag bits are not overwritten like data bits and (2) tag addresses are spread over a wide range, and only those incoming addresses that are within

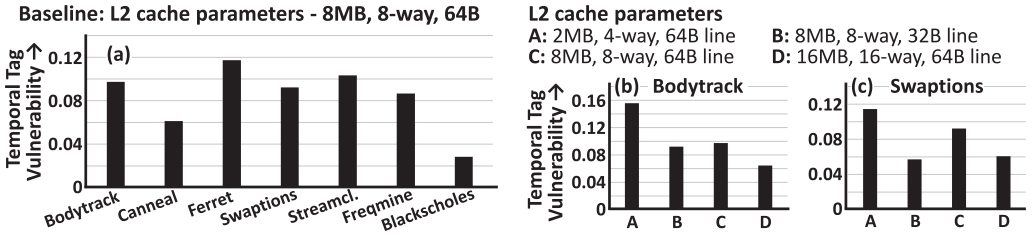

Fig. 5. Tag vulnerability analysis: (a) for different applications; ((b) and (c)) for different configurations.

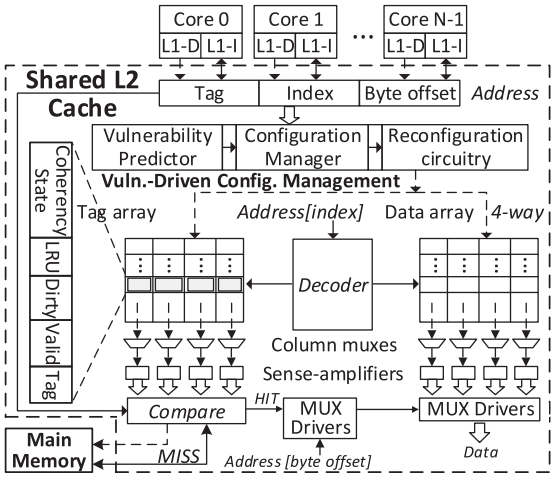

Fig. 6. Architectural overview of R^{2}Cache.

a Hamming distance of 4 (false hits/false misses) with respect to cache tags for clean cache lines are vulnerable. A detailed analysis of the different components contributing to tag vulnerability is presented in Section 4.4.

Figures 5(b) and (c) show the effect of cache parameter adaptation on tag vulnerability for *Bodytrack* and *Swaptions*. For both applications, smaller cache partitions (e.g., configuration *A*) have larger tag vulnerability, 15.5% and 11.4%, respectively. The smaller cache partition has a larger miss latency. As a result, dirty cache lines are resident in the cache for a longer duration before eviction and their tags show greater susceptibility to false misses due to bit errors.

Observation-5: Smaller cache partitions have larger tag vulnerability. Tags for the dirty cache lines with long temporal vulnerability show high susceptibility.

3. R^{2}CACHE: RELIABILITY-AWARE RECONFIGURABLE CACHE ARCHITECTURE

Figure 6 illustrates the overview of our R^{2}Cache architecture. The novel components for online vulnerability prediction and vulnerability-optimizing configuration management are shown along with the architectural support for cache parameter reconfiguration. The reconfiguration decision is orthogonal to the architectural support for reconfiguring cache parameters, thus making it possible for us to implement our own runtime configuration management technique on top of existing reconfigurable cache architectures like those in Zhang et al. [2004] and Wang et al. [2011]. Without the loss of generality, in this article, we adopt the following three architectural features for reconfiguration as the basic infrastructure support from Zhang et al. [2004] and Wang

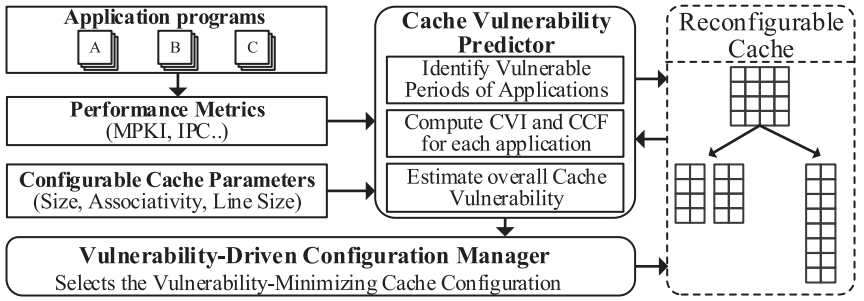

Fig. 7. Overview of R^{2}Cache.

et al. [2011]: *(1) Way-Concatenation* is the method to reduce the associativity of the cache without changing the effective cache size; *(2) Way-Shutdown* switches off certain ways of the cache set, thus resulting in a virtually smaller cache from the application point of view; and *(3) Cache Line Resizing* methods start with a given baseline size and add/remove cache lines based on the requirement. Existing approaches like those in Zhang et al. [2004] and Wang et al. [2011] 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.

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 runtime 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. An overview of R^{2}Cache is shown in Figure 7. 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 7 are explained in detail in the subsequent sections.

4. CACHE VULNERABILITY MODELING AND PREDICTION

4.1. 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. The *soft error rate*[2] of an SRAM-based cache ($Cache_{SER}$) can be defined as the product of the *error rate*[3] and the *cache's vulnerability index* (*CCF*, see Equation (4)), which is an indicator of the likelihood that such an error can corrupt the program state. The top part of Figure 7 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, for instance, masking of errors by invalid cache lines, combinational logic, and so on. For a given technology implementation (e.g.,

[2]Measured in FIT, such that 1 FIT = 1 error in 1 billion hours.
[3]Dependent on the circuit design and particle flux rate.

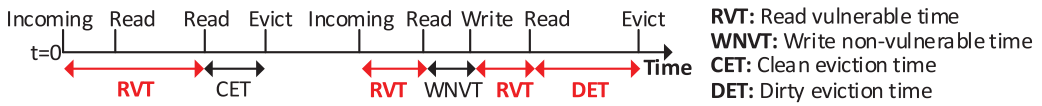

Fig. 8. Vulnerability components in cache accesses.

45nm), circuit design, operating voltage, and particle flux rate, we determine the SRAM soft error rate (fr) as in Dixit and Wood [2011] and Mukherjee et al. [2005]. 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 [Dixit and Wood 2011]. 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 [Rehman et al. 2011] 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:

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

4.2. Cache Vulnerability Components

Figure 8 illustrates the following different 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 overwrite 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. It 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.

4.3. Our Cache Data 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 parameters: cache partition size (CS), cache associativity (CA), and line size (LS). An application is given as a set of nodes $P = \{P_1, P_2, \ldots, P_N\}$ representing different phases during its execution. We consider a phase to be an interval of 100 million instructions, as an extensive characterization of applications in related work [Sundararajan et al. 2013b; Sherwood et al. 2003] has identified that a sampling interval of 10–100 million instructions can be taken to be representative of the phase behavior. Phase classification is based on clustering basic blocks considering their execution behavior and their frequency of occurrence. 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 $CVI\left(P_j, C_i\right)$ (Equation (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 ($VulBits_l$, needed for architecturally correct execution) of a cache line l with total number of bits $Bits_l$.

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

Note that for a cache line l, $VulPeriod_l(P_j, C_i)$ is the sum of RVT and DET (see Figure 8).

Cache Criticality Factor $CCF(P_j, C_i)$ (Equation (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,

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

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

As widely adopted in the cache literature [Srikantaiah et al. 2011], 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.

4.4. Our Cache Tag Vulnerability Model

In order to estimate the tag vulnerability, it is important to consider the tag access patterns. The tag vulnerability estimation differs from data vulnerability estimation in the following ways: First, errors in the tag bits are not overwritten by data writes. Second, errors in tag bits may result in false hits or false misses. While false hits can load (write) data from (to) incorrect cache lines, the effect of false misses depends on whether the cache line is clean or dirty. For a clean cache line, there is only a performance loss associated with loading correct data from the main memory. For a dirty cache line, tag bit errors can lead to writebacks to incorrect memory locations during eviction. Furthermore, errors in tag bits have a greater potential to cause application failures since they can cause applications to access either unmapped memory locations or violate their allocated address space.

We consider up to 4-bit spatial multi-bit errors with the $Nx1$ pattern that has been shown to be the most dominant in recent work [Wilkening et al. 2014]. For estimating tag vulnerability, we consider both single and multi-bit errors and add the resulting contributions from all error types. We use *Hamming Distance analysis* to estimate the number of bits by which the incoming address differs from that of the tag. False hits occur when the incoming address differs from the tag by N bits, and these N bits are flipped by a soft error. Therefore, for a clean cache line, only those N bits of the tag are considered to be vulnerable. In contrast, for a dirty cache line, all the tag bits are vulnerable since false hits can cause reading of faulty data or writebacks to incorrect locations and false misses can cause writeback of incorrect data.

Table I summarizes the vulnerability components in tag accesses. An interval is the time between consecutive references to the same cache line.

Table I. Cache Tag Vulnerability Components

Cache Event	Cache Line State	Vulnerable Bits of Tag	Impact
False Hit	Clean	From Hamming Distance	Interval is vulnerable
	Dirty	From Hamming Distance	Interval is vulnerable
False Miss	Clean	None	Interval is non-vulnerable. Performance loss due to re-fetch
	Dirty	All	Interval is vulnerable
Eviction	Clean	None	Interval is non-vulnerable, possible performance loss
	Dirty	All	Interval is vulnerable

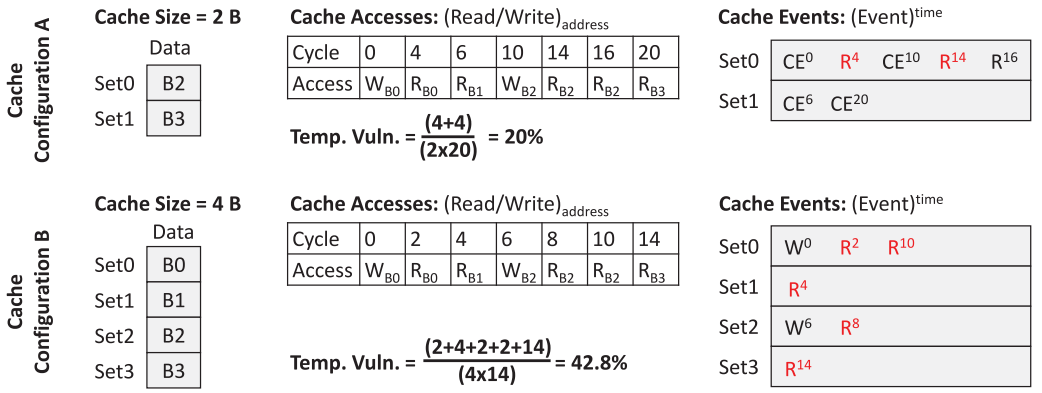

Fig. 9. An illustrative example showing the effect of reconfiguring cache size on the temporal vulnerability.

The cache criticality factor of the tag can be defined as:

$$CCF_{tag}(P_j, C_i) = \frac{\sum_{\forall t \in T} VulBits_t \times VulPeriod_t(P_j, C_i)}{\sum_{\forall t \in T} Bits_t \times ExecTime(P_j)}, \tag{5}$$

where T is the set of all cache tags.

The overall cache vulnerability can then be defined as:

$$CCF_{total}(P_j, C_i) = CCF_{data}(P_j, C_i) + CCF_{tag}(P_j, C_i). \tag{6}$$

4.5. An Illustrative Example

Figure 9 presents an example scenario to illustrate how the vulnerability of the cache varies with changing cache parameters like size. The following notations are used for the cache access events: R as read hit, W as write hit, CE as clean line eviction, DE as dirty line eviction with the superscript indicating the cycle the access was made, and the subscript indicating the block that was accessed. For example, W_{B2} indicates that a write access was made to block $B2$, and CE^{10} indicates a clean eviction event at cycle 10. The block ID indicates the set to which it will be mapped (e.g., B0, B2, B4, ..., map to set 0 while B1, B3, B5, ..., map to set 1). For illustrative purposes, we assume a cache hit latency of two cycles and a miss latency of four cycles and that all cache blocks are initially clean. We assume the same access sequence of reads and writes to blocks in both cases and note the corresponding variation in execution cycles due to different miss behavior. The blocks initially present in the warmed state of the cache are shown in the "Data" columns. The cache blocks marked in red are vulnerable for the duration indicated, while those marked in black are not vulnerable.

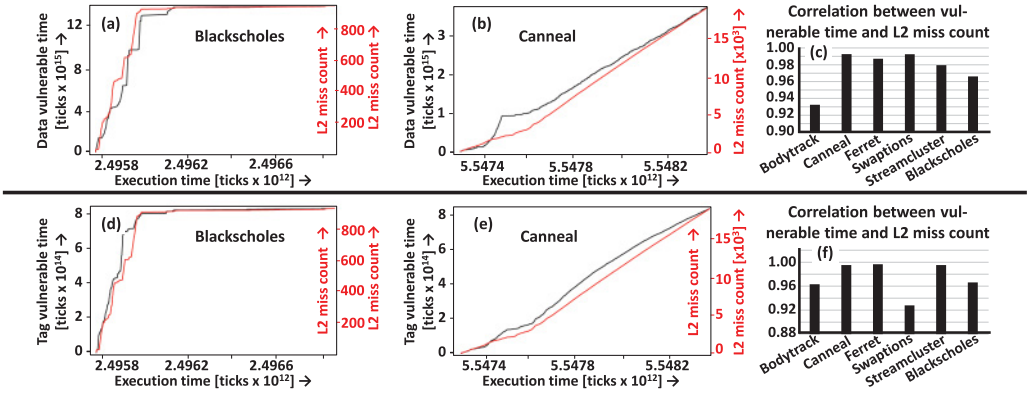

Fig. 10. Analyzing the correlation of cache misses with data vulnerability ((a) and (b)) and tag vulnerability ((d) and (e)). Predictor correlation for data (c) and tag (f).

For the purpose of illustration, in Figure 9, two direct mapped cache configurations are considered, one with two single-byte entries and the other with four single-byte entries. In configuration A, the first write access leads to a clean eviction, while for configuration B, the cache block is overwritten. The subsequent read access leads to a four-cycle vulnerable time for configuration A and a two-cycle vulnerable time for configuration B. This is due to the miss latency seen in configuration A. By performing a similar analysis for the other accesses, it can be seen that the last read access in configuration A does not incur any vulnerable time, while the block in Set 3 of configuration B is vulnerable for 14 cycles. Overall, configuration A shows a temporal vulnerability of 20% while for configuration B it is 42.8%. *This example illustrates that cache size adaptation can significantly influence the temporal vulnerability.*

4.6. Estimation of Cache Vulnerability

Our vulnerability predictor exploits the correlation existing between structure occupancies and cache vulnerability. Considering Equation (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.

The vulnerable bits $VulBits_l$ can be estimated offline using Monte Carlo fault injection campaigns (as discussed in Section 4.1) 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. A similar approach is used for estimating the parameters for tag vulnerability.

Our analyses in Figure 10 show 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 for both data and tag parts. Considering a total of N_O observations, the vulnerable period ($VulPeriod_i$) in the ith observation can be expressed as:

$$VulPeriod_i = \alpha(L2MissCount) + error_i, \tag{7}$$

Table II. Prediction Accuracy for Data and Tag

Application	Linear model used (x10^{12})		Avg. Pred. Accuracy [%]	
	Data	Tag	Data	Tag
Bodytrack	35.89*L2MissCount + 238.3	213.3*L2MissCount − 4221	82.31	90.53
Canneal	0.2396*L2MissCount + 355.5	4.729*L2MissCount + 4173	96.52	94.08
Ferret	1.569*L2MissCount + 948.3	0.539*L2MissCount − 2930	89.16	98.75
Swaptions	103.5*L2MissCount + 7635	12.62*L2MissCount + 6553	92.77	93.77
Streamcluster	0.7385*L2MissCount + 119.4	1.553*L2MissCount + 361.8	83.25	85.27
Blackscholes	11.89*L2MissCount − 69.67	88.91*L2MissCount + 7685	87.69	96.13

where $error_i$ is the prediction error of the linear model in the ith observation. To estimate α, we minimize the sum of squares of errors in different observations:

$$Vul.Period_{phase} = f(L2MissCount_{phase}) \times error_{prev\_phase} \tag{8}$$

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

4.7. 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 II summarizes the linear models used for data and tag vulnerability prediction and the accuracy of our predictor for different applications, averaged over different inputs.

Our approach is orthogonal to existing application phase detection and classification efforts. The greater the accuracy in phase classification, the better 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 (see Equation (8)). *Frequent reconfigurations are avoided by applying a tolerable overhead.*

As can be seen from Table II, the predictor accuracy for data vulnerability varies from 82% to 97% while that for tag vulnerability varies from 85% to 98% for the chosen applications. Both the data and tag vulnerability prediction accuracies are lower for *Streamcluster*. For this application, we found that the number of clean evictions in phase 2 are significantly larger ($>1000x$) than the number of dirty evictions as a result of which the vulnerability predicted using the miss counts from phase 1 results in over-estimation. The prediction error reduces in subsequent phases as the error from previous phases are taken into account. However, it must be noted that the vulnerable period during an applications 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, and so on [Duan et al. 2009]. 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 that show high reuse (N_{AL}). Since N_{AL} is difficult to estimate using well-known performance metrics like IPC, miss rates, and 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

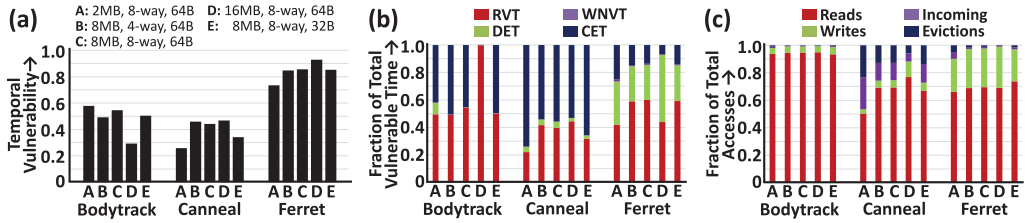

Fig. 11. (a) Vulnerability for three applications; (b) vulnerability components; (c) cache access events.

active cache footprint vectors as proposed in Srikantaiah et al. [2011]. 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 7.6.

5. RELIABILITY-AWARE DYNAMIC CACHE RECONFIGURATION

Before proceeding to the configuration manager of R^{2}Cache, we study the impact of different cache configurations on the cache vulnerability in detail.

5.1. Reliability Analysis of Different Cache Configurations

Figure 11 illustrates the analysis of temporal cache vulnerability, different vulnerability components, and their relationship to different cache events in different configurations for three representative benchmark applications. *Ferret* is highly vulnerable in almost all configurations because of large *DET*. In case of *Canneal*, a large number of cache blocks are evicted and replaced during the execution. 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 an *RVT* value 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 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 an 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 an increase in *DET*.

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 11).

ALGORITHM 1: Run-Time Configuration Management

Input: Library of cache configurations $C = \{C_1, C_2, \ldots, C_K\}$, Set of concurrently executing application threads $A = \{A_1, A_2, \ldots, A_M\}$, Set $P(i) = \{P_1, P_2, \ldots, P_N\}$ representing the runtime execution phases of application thread A_i, tolerable performance overhead τ and tolerable energy overhead λ.

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: **end for**
5: $p_{loss} \leftarrow 0$
6: **for** $a \in A$ **do**
7: $Vul_{min} \leftarrow Vul(C_{baseline})$
8: **for** $c \in C$ **do** \\find optimum cache configuration
9: $p_{loss} \leftarrow p_{loss} + \sum_{p \in P(i)} \eta(p, a, c)$
10: $e_{loss} \leftarrow e_{loss} + \sum_{p \in P(i)} \xi(p, a, c)$
11: $Vul(c) \leftarrow \sum_{p \in P(i)} CCF(p, c)$
12: **if** $Vul(c) \leq Vul_{min}$ && $p_{loss} < \tau$ && $e_{loss} < \lambda$ **then** \\satisfy performance and energy constraints
13: $C_{opt} \leftarrow c$
14: $Vul_{min}(a) \leftarrow Vul(C_{opt})$
15: **end if**
16: **end for**
17: **if** $C_{opt} == C_{valid}$ **then**
18: $CG(a) \leftarrow C_{opt}$
19: **end if**

Step 2: Intra-Application Cache Reconfiguration

20: **for** $p \in P(a)$ **do**
21: **for** $c \in C$ **do**
22: $Vul(c) \leftarrow CCF(p, c)$
23: $p_{loss} \leftarrow p_{loss} - \eta(p, a, CG(a)) + \eta(p, a, c)$
24: $e_{loss} \leftarrow e_{loss} - \xi(p, a, CG(a)) + \xi(p, a, c)$
25: **if** $Vul(c) \leq Vul_{min}(a)$ && $p_{loss} < \tau$ && $e_{loss} < \lambda$ **then**
26: $C_{opt}(p) \leftarrow c$
27: $Vul_{min}(p) \leftarrow Vul(C_{opt}(p))$
28: **end if**
29: **end for**
30: **if** $C_{opt}(p) == C_{valid}$ **then**
31: $FG(a, p) \leftarrow C_{opt}(p)$
32: **end if**
33: **end for**
34: **end for**

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.

5.2. Vulnerability-Driven Configuration Manager

Our runtime configuration manager (Algorithm 1) chooses reliability-wise beneficial cache configurations that minimize the total cache vulnerability for a set of concurrently executing application threads in the following two key steps (see Figure 12).

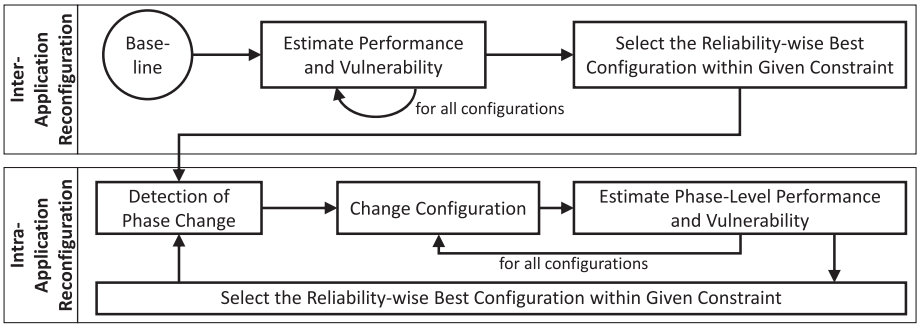

Fig. 12. Overview of Algorithm 1.

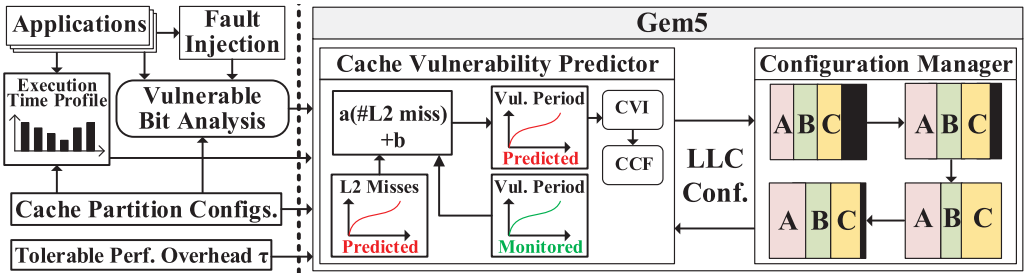

Fig. 13. Experimental setup for R^{2}Cache.

(1) 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 1–19). (2) 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 20–34). 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 100–4,000 cycles based on the number of cache configurations explored [Rawlins and Gordon-Ross 2013]. 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 1–4). Afterwards, it computes the performance loss ($\eta(p, a, c)$) and energy loss ($\xi(p, a, c)$) incurred for different applications when using different cache configurations (Lines 9 and 10) and the vulnerability of each application thread for each cache configuration (Line 11). Under user-provided tolerable performance and energy overhead constraints, the cache configuration with the minimum vulnerability is selected (Lines 12–15). An additional check is performed to ensure that the chosen cache configuration is valid (Line 17). 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 and energy overheads are not exceeded (Lines 20–34).

6. EXPERIMENTAL SETUP

Figure 13 shows the details of our experimental setup. We extended the cycle-accurate *Gem5* [Binkert et al. 2011] full system simulator with the ability to (1) monitor

Table III. Settings for the Experimental Setup

Core parameters

Core type	Alpha 21264 [Kessler et al. 1998]
OS	Linux 2.6
Number of cores	4
Core frequency	2 GHz
No. of Data TLB entries	64
No. of Instr. TLB entries	48

Cache parameters

Coherence	MOESI Snooping based	
Replacement	LRU based policy	
Private L1 cache	Data Cache	32kB, 2-way, 64B
	Instruction Cache	32kB, 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 IV. Application Mixes: (x,y,z) Refers to the Number of Thread Instances of Each Benchmark

Name	Type	Benchmarks	Name	Type	Benchmarks
Mix 1	(2,2)	Blackscholes, Swaptions	Mix 6	(2,2)	x264, Streamcluster
Mix 2	(8,4)	Canneal, Swaptions	Mix 7	(4,4)	Bodytrack, Streamcluster
Mix 3	(2,2)	LU_NC, Swaptions	Mix 8	(4,4)	Canneal, Blackscholes
Mix 4	(8,4)	Canneal, Bodytrack	Mix 9	(2,2)	Swaptions, Radix
Mix 5	(1,1,2)	Blacksc., Bodytrack, Canneal	Mix 10	(1,2,1)	Lame, Basicmath, Patricia

the vulnerable periods during an application's lifetime for both cache data and tag, (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 Section 4. Table III summarizes our experimental setup.

Following the norms of cache and architecture communities, we fast-forwarded the initialization and thread creation phases and report statistics only for the parallel and thread synchronization phases of the execution of each application, since the cache vulnerability while booting the operating system is same for all applications. Based on the offline characterization of applications into phases, the phase boundaries (measured in committed instructions) are used to guide phase detection at runtime. In the experiments, we use performance counters to detect phase changes. In order to make the phase classification independent of hardware statistics and inputs, we rely on offline profiling of the application while defining phases as adopted in Carlson et al. [2014] and Sherwood et al. [2003]. 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 [Bienia et al. 2008] with *simsmall* inputs for our evaluation. Besides mixes of PARSEC applications, we also defined several application mixes with *SPLASH2* (*LU_Non-contiguous*, *Radix*) [Woo et al. 1995] and *MiBench* (*Lame*, *Basicmath* and *Patricia*) [Guthaus et al. 2001] benchmark suites. These application mixes (Table IV) 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 runtime scenarios. For this, we examined the sharing behavior between threads of the same application while running in isolation.

Canneal, *Ferret*, *x264*, and *Bodytrack* show high degrees of sharing between concurrently executing threads. They were grouped together with applications showing comparatively lesser degree of sharing like *Blackscholes*, *Swaptions*, and *Streamcluster* (see Mix 2, 6, 7, 8 in Table IV). Mix 4, 5 comprises applications that compete for shared resources (i.e., *Canneal* and *Bodytrack*). Mix 1, 9, 10 consists of single-threaded applications (i.e., *Lame* and *Patricia*) and multi-threaded applications that show little data sharing between threads (i.e., *Blackscholes* and *Swaptions*).

6.1. Comparison Partners

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

(1) Vulnerability-Reducing Non-Reconfigurable Caches based on smart EWB [Jeyapaul and Shrivastava 2013] that reduce the temporal vulnerability of the cache by virtue of reducing *DET*, but its configuration cannot be adapted at runtime. 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) [Qureshi and Patt 2006], which partitions the shared cache between multiple applications of a workload, so 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.

(4) Vulnerability Reducing Non-Reconfigurable Caches Based on Clean Cache Invalidation (CCI) [Wang et al. 2009], which reduces the temporal vulnerability of the cache by virtue of reducing *RVT*. This scheme invalidates clean cache lines at regular intervals and reduces *RVT* but incurs performance loss due to increased cache misses. We consider an interval for $4K$ cycles for comparison similar to Wang et al. [2009] that has been shown to provide the best tradeoff between reliability achieved and performance degradation incurred.

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. We use *CACTI 6.5* [Muralimanohar et al. 2008] to obtain the power consumed by the last-level cache while executing different PARSEC applications. This includes both the static power (gate leakage and sub-threshold leakage) as well as dynamic power. When our runtime reliability manager chooses smaller-sized cache partitions the remainder of the cache can be switched-off to save power. Similarly to Yoon and Erez [2009], we use an additional 0.4nJ dynamic energy per cache access for SEC-DED protection in a 45nm process. The energy increase during reconfiguration to write-back dirty cache lines is taken into account in the results. 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.

In the following we present results for the vulnerability in order to stay independent from parameters like flux rate, technology size, and so on, and as it is the parameter that can be controlled for the optimization in this work.

7. RESULTS AND DISCUSSION

7.1. Vulnerability Compared to State-of-the-Art for Different Applications

This section analyzes the overall vulnerability savings obtained by our approach under different user-provided performance constraints considering both data and tag vulnerability. Figure 14 shows the cache vulnerability reduction results for our R^{2}Cache for different performance constraints (2%, 5%, 10%, 15%, >15%) compared to the state of the art described in Section 6.1.

The results are presented for individual multi-threaded applications of the PARSEC benchmark suite. The performance-reliability tradeoff can be observed in Figure 14, where increasingly reliable cache configurations are identified by our manager when larger performance overheads are allowed. Notably, *Canneal* can be executed on a number of reliably efficient cache configurations with vulnerability ranging from 53.5% to 18.4%, each coming at a different performance cost. Also, the vulnerability of the best configuration chosen by R^{2}Cache is very close to the vulnerability provided by

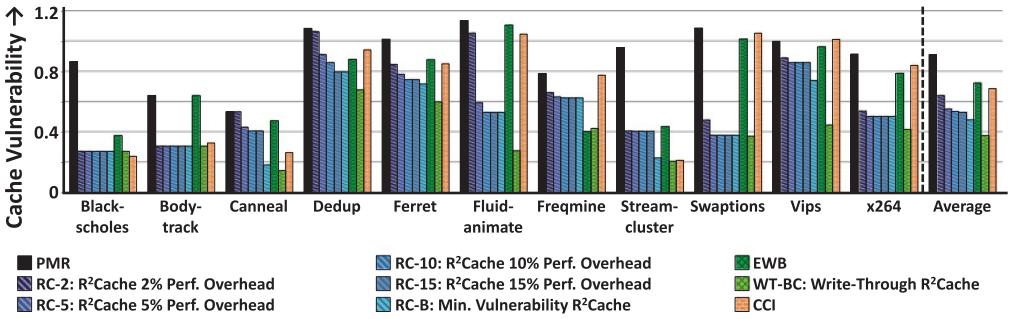

Fig. 14. Comparing vulnerability reductions of R^{2}Cache to state-of-the-art caches for different applications.

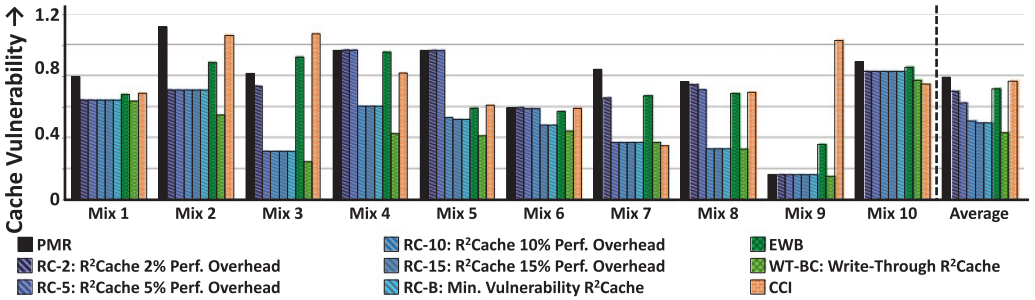

Fig. 15. Comparing vulnerability reductions of R^{2}Cache to state-of-the-art caches for different mixes.

WT-BC (14.7%). 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 >60% vulnerability savings when compared to EWB, PMR, and CCI, 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 runtime. These write hits increase $WNVT$ and decrease DET, reducing temporal vulnerability for the cache. The vulnerability savings achieved by R^{2}Cache are at par or even better than CCI for applications with large RVT like *Blackscholes*, *Bodytrack*, and *Streamcluster*.

Figure 14 shows that, for most applications (*Blackscholes*, *Bodytrack*, *Streamcluster*, etc.), the vulnerability of the best R^{2}Cache partition (i.e., RC-B) is close to WT-BC. *Freqmine*, however, has a large number of dirty cache blocks, even in the best partition selected by our technique, leading to limited gains when compared to EWB.

7.2. Vulnerability Compared to State-of-the-Art for Different Mixes

Figure 15 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 4, with a 1MB partition size for *Canneal* providing 37.4%, 36.5% and 26.2% vulnerability savings when compared to PMR, EWB, and CCI, respectively. Mix 4 shows high MPKI (0.55) in this configuration, and many frequent evictions reduce the vulnerable DET period. Mix 5 also shows a similar behavior. DET is the largest contributor to the vulnerability of Mix 9. As

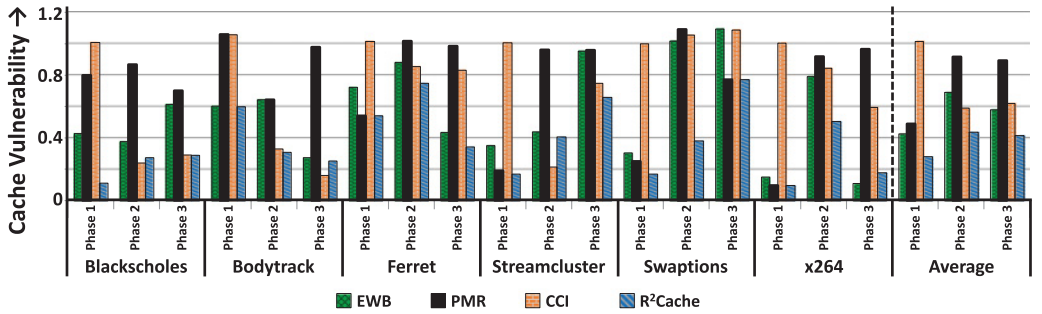

Fig. 16. Detailed phase-level vulnerability comparison for different applications (tolerable overhead = 10%).

a result, CCI, which targets only clean cache lines, performs poorly. However, our manager selects a large L2 partition with low tag vulnerability and is able to provide greater reliability benefits. EWB performs slightly better for Mix 10 mainly because it reduces dirty eviction time. On account of the large DET obtained even in the best configuration provided by our runtime manager, the reliability improvements provided are limited.

7.3. Detailed Phase-Level Vulnerability Comparison for Different Applications

Figure 16 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 [Sembrant et al. 2012], 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. *x264*, which shows high MPKI, benefits from using smaller size partitions in Phase 2, as frequent evictions lead to a decrease in DET. However, these evictions come at a performance cost as they affect cache locality. Under 10% performance overhead constraint R^{2}Cache shows vulnerability reductions of 36.17%, 45.02%, and 40.18% in Phase 2 when compared to EWB, PMR, and CCI, respectively. However, larger cache partitions show lesser temporal vulnerability for applications like *Blackscholes*, whose access pattern shows a larger number 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. Both CCI and R^{2}Cache show good reliability improvements for this application in Phase 2 and Phase 3, which are dominated by clean cache lines. CCI performs poorly in Phase 1 for almost all applications because of a large percentage of idle dirty cache lines after the initialization phase.

7.4. Detailed Phase-Level Vulnerability Comparison for Different Mixes

Figure 17 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, PMR, and CCI caches show high vulnerability. The large sized cache partition (16MB) chosen by the manager for Phase 2 of Mix 3 provides 65.89%, 61.57%, and 70.72% vulnerability savings when compared to EWB, PMR, and CCI, respectively. The baseline cache used for vulnerability optimizations by EWB and PMR has both high data and tag vulnerability as a result of which the improvements provided by these techniques are lesser when compared to R^{2}Cache. Vulnerability

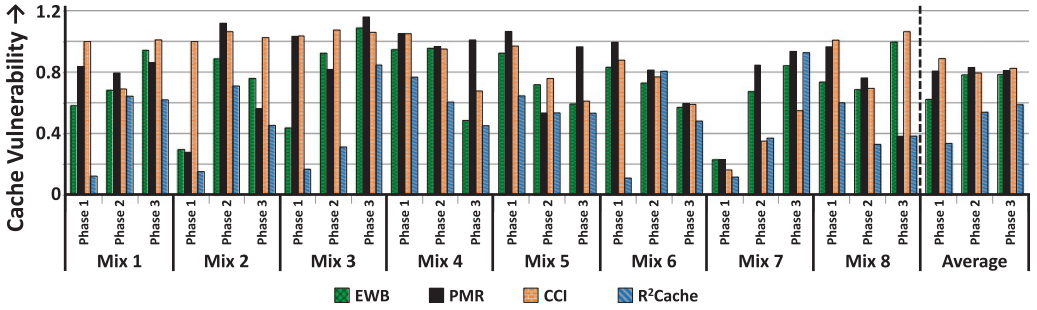

Fig. 17. Detailed phase-level vulnerability comparison for different application mixes.

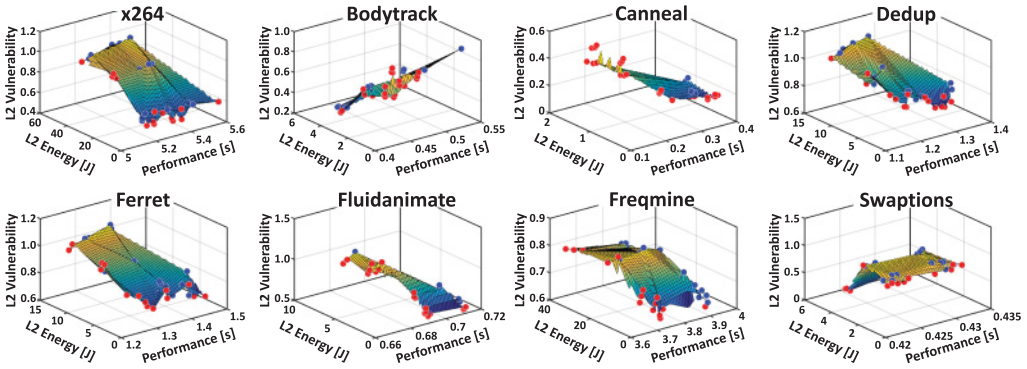

Fig. 18. Reliability-performance-energy configuration design space for different applications.

improvements are not obtained for R^{2}Cache when compared to EWB and CCI in Phase 2 of Mix 6 and Phase 3 of Mix 7 because of the limitation that our reconfiguration cannot completely eliminate the read vulnerable time and dirty eviction time even in the best chosen configuration. However, the vulnerability savings provided by R^{2}Cache are comparable to EWB and CCI even in these cases.

Summary of Vulnerability Reduction Results: Overall, the proposed R^{2}Cache outperforms state-of-the-art vulnerability reducing non-reconfigurable caches [Jeyapaul and Shrivastava 2013; Wang et al. 2009] and performance maximizing reconfigurable cache techniques [Qureshi and Patt 2006]. On average, R^{2}Cache provides 35.27%, 23.42%, and 15.48% vulnerability savings when compared to PMR, EWB, and CCI, 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 on the cache access patterns of different applications. The vulnerability savings achieved by R^{2}Cache are 14.63% away from that obtained by WT-BC.

7.5. Reliability-Performance-Energy Configuration Design Space

Figure 18 shows the last-level cache configuration space for different applications and their corresponding vulnerability, performance, and energy. The Pareto-optimal cache configurations that trade off between vulnerability and performance/energy are

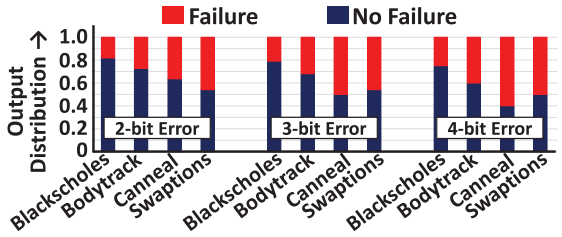

Fig. 19. Evaluation of error distribution.

highlighted in red. For *Canneal*, *Dedup*, *Ferret*, and *Fluidanimate*, it is observed that almost all cache configurations lie on the Pareto-frontier that trades off between vulnerability and performance. For these applications, smaller cache partitions show the greatest reliability. However, these partitions also experience increased number of cache evictions, resulting in a reliability-performance tradeoff. *Bodytrack* and *Swaptions*, on the other hand, show distinct Pareto-frontiers that trade off between vulnerability and energy. Both of these applications show reliability benefits with large cache partitions, as discussed in Section 7.1. However, large cache partitions also consume more energy, hence the reliability-energy tradeoff.

It is important to note that all applications show last-level cache configurations with wide ranges of vulnerability, performance, and energy. This rich design space enables R^2Cache to achieve significant vulnerability savings when compared to other non-reconfigurable caches under different user provided constraints.

7.6. 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 19 shows the results of our Monte Carlo fault injection campaigns on the last-level cache, using the multi-bit error distribution model from Dixit and Wood [2011]. Assuming that the Baseline cache (see Section 3) 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 19, *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* and *Bodytrack* show good error masking properties. 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.

7.7. 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 [Alameldeen et al. 2011; Qureshi and Chishti 2013] are employed to the last-level cache is shown in Figure 20(a). *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.

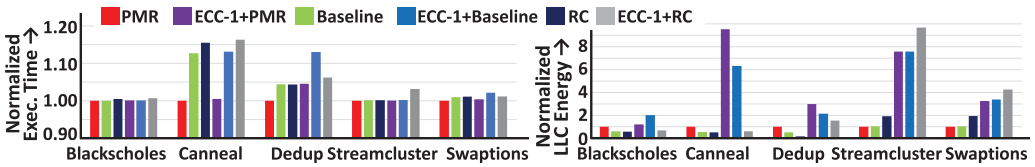

Fig. 20. (a) Normalized execution time and (b) normalized last-level cache energy for ECC-based design.

We consider a pipelined SEC-DED implementation, similar to that in Alameldeen et al. [2011], that incurs an additional latency of 1 cycle per access. In Figure 20(a), we show only R^2Cache 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 7.1.

In the R^2Cache 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.

In Figure 20(a), ECC-1+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^2Cache, respectively. From Figure 20(a) it can be seen that the performance of R^2Cache 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*.

7.8. Last-Level Cache Energy Consumption

In Figure 20(b), the last-level cache energy consumption of our technique (both with and without error protection) is shown, normalized with respect to PMR. Our runtime 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. 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. From Figure 20(b), it can be observed that the savings are highly dependent on the partition sizes chosen for individual applications.

7.9. Hardware Overhead

As a part of the reconfiguration control, we require a 6-bit configuration register for 6 L2 partition sizes, 5 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). The additional control logic and predictor consume negligible energy.

8. CONCLUSION

We presented a novel reliability-aware reconfigurable cache architecture R^2Cache that adapts the cache parameters for concurrently executing multi-threaded workloads at runtime in order to minimize their vulnerabilities to soft errors. To enable an efficient

design, we developed a cache vulnerability model that considers both the data and tag arrays. The model jointly accounts for the spatial and temporal vulnerabilities of the cache when executing different applications. We also performed a detailed analysis of the cache vulnerability for different cache configurations. Using the vulnerabilities estimated at runtime for different applications, an appropriate configuration is selected for different applications and different execution phases within an application. To enable design-time customization, cache configurations that lie on the Pareto-frontier (trade-off between reliability and performance/energy) are identified. On average, R^2Cache provides up to 35.27%, 23.42%, and 15.48% vulnerability savings when compared to the 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-/energy-constrained reliability mitigation in large-sized shared last-level caches.

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) as part of the priority program "Dependable Embedded Systems" (SPP 1500 - spp1500.itec.kit.edu) [Henkel et al. 2011].

REFERENCES

A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu. 2011. Energy-efficient cache design using variable-strength error-correcting codes. In *International Symposium on Computer Architecture (ISCA)*. 461–472.

R. C. Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Trans. Device Mater. Reliab.* 5, 3 (2005), 305–316.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 72–81.

N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Arch. News* 39, 2 (2011), 1–7.

T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout. 2014. BarrierPoint: Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2–12.

C.-L. Chen and M. Y. (Ben) Hsiao. 1984. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM J. Res. Dev.* 28, 2 (1984), 124–134.

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

L. Duan, B. Li, and L. Peng. 2009. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *International Conference on High-Performance Computer Architecture (HPCA)*. 129–140.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. *IEEE International Workshop on Workload Characterization (IISWC)* (2001), 3–14.

A. Haghdoost, H. Asadi, and A. Baniasadi. 2010. System-level vulnerability estimation for data caches. In *IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*. 157–164.

J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Härtig, L. Hedrich, A. Herkersdorf, R. Kapitza, D. Lohmann, P. Marwedel, M. Platzner, W. Rosenstiel, U. Schlichtmann, O. Spinczyk, M. Tahoori, J. Teich, N. Wehn, and H.-J. Wunderlich. 2011. Design and architectures for dependable embedded systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 365–374.

IBM. 2015. Power Servers. http://www-03.ibm.com/systems/power/hardware/. (2015).

Intel. 2015. Itanium Processor. http://ark.intel.com/products/family/451/Intel-Itanium-Processor. (2015).

R. Jeyapaul and A. Shrivastava. 2013. Enabling energy efficient reliability in embedded systems through smart cache cleaning. *ACM Trans. Des. Automat. Electron. Syst.* 18, 4 (2013), 53.

R. E. Kessler, E. J. McLellan, and D. A. Webb. 1998. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design (ICCD)*. 90–95.

J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. C. Hoe. 2007. Multi-bit error tolerant caches using two-dimensional error coding. In *International Symposium on Microarchitecture (MICRO)*. 197–209.

F. Kriebel, A. Subramaniyan, S. Rehman, S. J. B. Ahandagbe, M. Shafique, and J. Henkel. 2015. R^2Cache: Reliability-aware reconfigurable last-level cache architecture for multi-cores. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10.

L. Li, V. Degalahal, N. Vijaykrishnan, M. T. Kandemir, and M. J. Irwin. 2004. Soft error and energy consumption interactions: A data cache perspective. In *International Symposium on Low Power Electronics and Design (ISLPED)*. 132–137.

S. S. Mukherjee, J. S. Emer, and S. K. Reinhardt. 2005. The soft error problem: An architectural perspective. In *International Conference on High-Performance Computer Architecture (HPCA)*. 243–247.

S. S. Mukherjee, C. T. Weaver, J. S. Emer, S. K. Reinhardt, and T. M. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *International Symposium on Microarchitecture (MICRO)*. 29–42.

N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. 2008. Architecting efficient interconnects for large caches with CACTI 6.0. *IEEE Micro* 28, 1 (2008), 69–79.

M. K. Qureshi and Z. Chishti. 2013. Operating SECDED-based caches at ultra-low voltage with FLAIR. In *International Conference on Dependable Systems and Networks (DSN)*. 1–11.

M. K. Qureshi and Y. N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *International Symposium on Microarchitecture (MICRO)*. 423–432.

M. Rawlins and A. Gordon-Ross. 2013. A cache tuning heuristic for multicore architectures. *IEEE Trans. Comput.* 62, 8 (2013), 1570–1583.

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

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

T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. 2003. Discovering and exploiting program phases. *IEEE Micro* 23, 6 (2003), 84–93.

S. Srikantaiah, E. Kultursay, T. Zhang, M. T. Kandemir, M. J. Irwin, and Y. Xie. 2011. MorphCache: A reconfigurable adaptive multi-level cache hierarchy. In *International Conference on High-Performance Computer Architecture (HPCA)*. 231–242.

K. T. Sundararajan, T. M. Jones, and N. P. Topham. 2013a. RECAP: Region-aware cache partitioning. In *International Conference on Computer Design (ICCD)*. 294–301.

K. T. Sundararajan, T. M. Jones, and N. P. Topham. 2013b. The smart cache: An energy-efficient cache architecture through dynamic adaptation. *Int. J. Parallel Program.* 41, 2 (2013), 305–330.

S. Wang, J. S. Hu, and S. G. Ziavras. 2009. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Transactions on Computers (TC)* 58, 9 (2009), 1171–1184.

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

M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli. 2014. Calculating architectural vulnerability factors for spatial multi-bit transient faults. In *International Symposium on Microarchitecture (MICRO)*. 293–305.

S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*. 24–36.

D. H. Yoon and M. Erez. 2009. Memory mapped ECC: Low-cost error protection for last level caches. In *International Symposium on Computer Architecture (ISCA)*. 116–127.

C. Zhang, F. Vahid, and R. L. Lysecky. 2004. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 3, 2 (2004), 407–425.

C. Zhang, F. Vahid, and W. A. Najjar. 2003. A highly-configurable cache architecture for embedded systems. In *International Symposium on Computer Architecture (ISCA)*. 136–146.

W. Zhang. 2005. Computing cache vulnerability to transient errors and its implication. In *International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*. 427–435.

W. Zhang, S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam. 2003. ICR: In-cache replication for enhancing data cache reliability. In *International Conference on Dependable Systems and Networks (DSN)*. 291–300.

Y. Zou and S. Pasricha. 2014. 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)*. 4:1–4:10.

Received January 2016; revised May 2016; accepted May 2016