Runtime-Guided ECC Protection using Online Estimation of Memory Vulnerability

Luc Jaulmes*, Miquel Moretó*, Mateo Valero*, Mattan Erez† and Marc Casas*
*Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
first.last@bsc.es
†Electrical and Computer Engineering Department
University of Texas at Austin
Austin, USA
mattan.erez@utexas.edu

***Abstract*— Diminishing reliability of semiconductor technologies and decreasing power budgets per component hinder designing next-generation high performance computing (HPC) systems. Both constraints strongly impact memory subsystems, as DRAM main memory accounts for up to 30 to 50 percent of a node's overall power consumption, and is the subsystem that is most subject to faults. Improving reliability requires stronger error correcting codes (ECCs), which incur additional power and storage costs. It is critical to develop strategies to uphold memory reliability while minimising these costs, with the goal of improving the power efficiency of computing machines.**

We introduce a methodology to dynamically estimate the vulnerability of data, and adjust ECC protection accordingly. Our methodology relies on information readily available to runtime systems in task-based dataflow p rogramming m odels, a nd t he e xisting Virtualized Error Correcting Code (VECC) schemes to provide adaptable protection. Guiding VECC using vulnerability estimates offers a wide range of reliability-redundancy trade-offs, as reliable as using expensive offline profiling f or g uidance a nd u p t o t o 2 5% safer than VECC without guidance. Runtime-guided VECC is more efficient than a stronger uniform ECC, reducing DIMM lifetime failure from 1.84% down to 1.26% while increasing DRAM energy consumption by only $1.03\times$**.**

***Index Terms*—Vulnerability, Runtime Systems, Error Correcting Codes, DRAM**

I. Introduction

Errors become more common as silicon technologies shrink and become more vulnerable. Manufacturing variability and circuit ageing cause intermittent or permanent faults [4, 5], while effects such as radiation or small voltage/frequency margins cause transient faults [31]. Growing memory sizes in supercomputers and data centers further exacerbate the effects of errors in memory.

The standard way of tolerating all these forms of errors is to apply an error correcting code (ECC). However, ECC comes with overheads in terms of storage space and power consumption. This increased power cost constitutes an important constraint in areas from high performance computing (HPC) [13] to mobile devices [21], making it undesirable to uniformly increase ECC strength as an answer to higher fault rates. It is preferable to protect data selectively by applying strong ECC to high-risk memory regions only, while cheaper protection can be used on lower risk regions. For this to be possible, we need to develop methodologies to automatically and dynamically quantify the vulnerability of the different portions of data stored in memory before a program executes. Prior work on measuring memory vulnerability relies on offline application profiling [10, 11, 15, 36], but online methods are needed for widespread deployment.

At the same time, the growing complexity of HPC machines increasingly impedes their programmability, which has motivated the emergence of novel task-based dataflow programming models [3, 6, 9, 18, 25]. These models facilitate the programmer's work by providing abstractions (tasks and data dependencies) and supporting software (the runtime system). With an initial focus on scheduling work transparently, runtime systems are used for many dynamic optimisations, such as prefetching or partitioning caches [17, 26].

Our key idea is to exploit the opportunity presented by runtime systems to estimate memory vulnerability online, and to allow adapting ECC protection dynamically using these estimates. We evaluate the vulnerability of the memory pages allocated by each parallel workload using the programming model's dataflow dependencies, which are expressed in the source code and exposed to the runtime system. This allows the runtime system to predict the future data accesses to each memory location, which are a key factor in the probability of encountering an error at this location. For example, data that is overwritten by subsequent stores or that is never consumed by an application has no impact on program state. The ability to detect this kind of behaviour online is then exploited to target the most vulnerable memory pages for increased ECC protection using the VECC scheme [35], with the aim of reducing the failure rate for any storage overhead setting.

The main contributions of this paper are:

- A methodology to estimate the vulnerability of memory at the start of a program's execution, that relies only on information available to the runtime system.
- Selecting memory pages for additional protection based on their estimated vulnerability, to minimise DRAM failures for any given redundancy constraint. This strategy can be used online and performs as well as expensive offline simulator analysis, with at most a 1.92% relative difference in failure rates.
- An in-depth evaluation of the performance, power, and reliability impact of selective memory protection, for various protected page selection strategies, highlighting the power efficiency of VECC strategies, which achieve 31% failure reductions at a 1.04× energy cost.

The paper is organised as follows: Section II presents the background on memory vulnerability and adjustable ECC schemes. We present in Section III an overview of the runtime-guided ECC proposal, detailing in Section IV the online vulnerability model, and in Section V, the adaptable ECC. In Section VI we describe the methodology and our various frameworks, in Section VII we present the evaluation, and in Section VIII our conclusions.

II. Background

Previous work introduced techniques and metrics to protect and evaluate the vulnerability of data stored in memory, as well as programming models relying on runtime systems. We give an overview of their most relevant details.

A. Memory Vulnerability

The architectural vulnerability factor (AVF) [23] is commonly used to estimate the priority of protecting different bits in the state of a computer system. While it seems a natural fit for guiding ECC protection, it is in fact not well suited for memory. This metric is defined for every state bit as the fraction of cycles in which the bit *matters*, i.e. the cycles at which changing the bit to a different value would result in a different outcome – either crashing the system or causing incorrect results to be computed.

However, due to ECC, the outcome of a program execution is never affected by a single bit in memory. Several erroneous bits are required to cause an error, and computing the joint AVF of several bits requires simulating complex models [33]. Furthermore, uncorrected errors in the same ECC word as accessed bits may cause crashes – even when the bits accessed are not erroneous themselves. Therefore, the most appropriate metric to quantify the vulnerability of an ECC word stored in memory is the vulnerability factor induced only by the memory accesses to this word [11, 15]. While other proposals aim to build on or replace single-bit AVF with metrics based on memory access counts [10, 36], all these existing approaches rely on offline profiling or simulating.

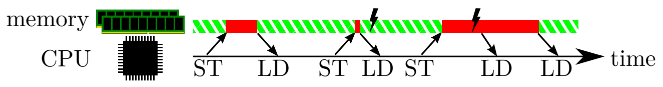


Fig. 1. Vulnerability timeline for a memory location.

Stores (ST) to a memory location overwrite the data it contains, while loads (LD) retrieve it. Time is safe before a store (in hatched green), and vulnerable before a load (in solid red): the first fault (shown as a black lightning bolt) has no effect, while the second affects the next 2 loads.

This paper focuses on the vulnerability in memory induced by memory accesses [11, 15], to which we refer simply as *vulnerability* in the rest of this paper. For every ECC word in memory, cycles are categorised as either safe or vulnerable, depending on the next memory access to this word: cycles preceding a load are vulnerable, and cycles preceding a store are safe. Figure 1 illustrates these categories by depicting vulnerable cycles in solid red, and safe cycles in hatched green. Formally, vulnerability is $V = \frac{\text{vulnerable cycles}}{(\text{safe cycles}+\text{vulnerable cycles})}$. Visually, the vulnerability of a memory location is the fraction of the full timeline that is coloured in solid red in Figure 1.

Vulnerability can be defined for any data granularity that is a multiple of an ECC codeword however, and we will also consider the *row-level vulnerability*, which is defined for a DRAM row. In this case, a cycle is classified as vulnerable if any codeword in the row is next accessed by a load. In other words, a cycle is safe for row-level vulnerability only if every codeword stored in a DRAM row will be overwritten (or remain unused).

B. Variable Strength ECC Schemes

Differences in memory vulnerability allow adjusting ECC protection accordingly, given an ECC scheme with different protection levels. However, industry standard ECC schemes provide uniform protection only, due to their layout in DRAM.

DRAM modules are organised in ranks, which consist of sets of chips accessed simultaneously, each contributing a number of bits to the rank's output [12]. To tolerate errors in main memory, vendors provide DRAM modules with additional storage, by adding more chips to a rank. This widened rank typically accesses 72 bits simultaneously instead of 64, and the additional storage can be used to store the ECC's check bits. State of the art protection in supercomputers is ChipKill [8], which tolerates up to a full chip failure. A stronger alternative, also uniform, is Double ChipKill, which can tolerate 2 full chip failures, at the price of a much wider rank obtained by operating 2 channels in lockstep [34].

Variable strength ECC schemes have been proposed for DRAM. They allow increasing error protection for some parts of memory without uniformly increasing the ECC cost, however always without dynamic tuning strategies. Virtualized ECC (VECC) [35] stores a second level of ECC protection in addressable memory, while using a first tier

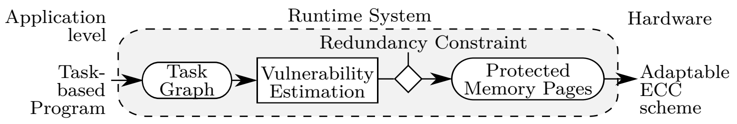


Fig. 2. Vulnerability Modelling and Protection Selection

ECC code stored at the rank-level, identically to uniform ECC schemes. In this configuration, VECC's first tier is used to detect errors, and both tiers of redundancy are used to correct detected errors. Similarly to VECC, Odd-ECC [16] offers two levels of protection, however with a fixed pre-defined arrangement in memory of 256KB pages.

Variable strength ECC schemes have also been devised for caches [2, 27] and NAND memories [37]. Those ECC schemes could be guided by our memory vulnerability estimation methodology, and replace VECC, provided they are extended for DRAM. Furthermore, most of these proposals focus on adjusting the ECC protection to hardware variability, while our work adapts to memory usage.

C. Task-based Programming Models

Several new programming models have been proposed to reduce the complexity of programming current and future HPC infrastructures, such as task-based dataflow programming models [3, 9, 18, 25]. In these models, the programmer subdivides a program's workload in *tasks*, and expresses the flow of data between these tasks. The runtime system then orchestrates the parallel execution of the program on the available hardware resources such as CPU cores, GPUs, etc., while honouring the constraints expressed by the flow of data.

In the *OpenMP Tasks* programming model, the flow of data between tasks is expressed by `pragma` annotations that declare which data each task is going to access [25]. These annotations also specify whether tasks access data dependencies as input, output, or both. For example the annotation `pragma omp depend(in: a, out: b, inout: c)` declares that the corresponding task will consume data `a`, produce data `b`, and both consume and produce data `c`. This information is necessary for the runtime system to correctly compute the ordering of tasks: two tasks accessing the same data as read-only may do so simultaneously, whereas if one task produces and another consumes the same data, their ordering must be enforced. In this way, the runtime system builds the *task dependency graph*, where each node of the graph is a task, and each edge represents a dataflow dependency.

III. Overview of Runtime-Guided ECC

To enable guiding adaptable ECC protection dynamically, we must be able to estimate vulnerability and use this information for ECC decisions as soon as possible during the execution of a program. We illustrate the steps of this process and how they interact in Figure 2.

From the vulnerability estimation, detailed in Section IV, we select memory regions for stronger ECC protection at the granularity of a memory page. Intuitively, the more memory is protected with stronger ECC, the higher the overhead. This overhead comes in terms of memory storage, LLC occupancy, and DRAM write requests, that are dedicated to the supplementary ECC symbols, as detailed in Section V. Thus, we select the amount of memory that is protected as the parameter for the redundancy-reliability trade-off. Given the fraction of protected memory, we select for protection the most vulnerable memory pages according to our model predictions. By prioritising these pages, the runtime system optimises the reliability of the program's execution under redundancy constraint. This means that the correct ordering of vulnerability model values within a benchmark is of particular importance to guide ECC protection.

Our vulnerability model is designed for online use, by keeping the model as simple and lightweight as possible and by relying only on data available at the runtime. This is in contrast to the evaluation of the model and of its impact, presented in Section VII, that relies on complex simulator infrastructures and takes into account all the effects ignored by the model.

Specifically, the simplicity of the model relies on the entire modelling being done with a single traversal of the task dependency graph, and representing the vulnerability for each memory region using only 2 floating-point values. Therefore, vulnerability can be estimated by the runtime system as soon as the task graph is known. In fact, the runtime system may even update the vulnerability model incrementally with each new task creation. Our methodology also enables taking different protection decisions for different time intervals of the program execution time. In practice however, we take protection decisions once only for the whole execution time of each program, and before its execution, to limit the scope of this proposal to modelling vulnerability and evaluating the quality of guiding ECC using this model.

IV. Modelling Vulnerability from the Task Dependency Graph

The information required to compute the vulnerability is the timing of memory accesses for each location in memory. Figure 3 presents an overview of the vulnerability model, and what data is required for each of its parts.

A. Information Available to the Runtime System

As soon as a program starts executing and creates the tasks it intends to run, the runtime system has access to the task dependency graph, represented in Figure 3. Tasks are numbered 1 to 5 and dependencies *a* to *d*, and are respectively depicted as nodes and edges in the task graph. With a simple graph traversal and knowledge of the task execution times, the runtime can predict the scheduling of tasks on the available computing units (3 in this example). This scheduling is represented on the right-hand side of Figure 3, and it allows the runtime system to predict

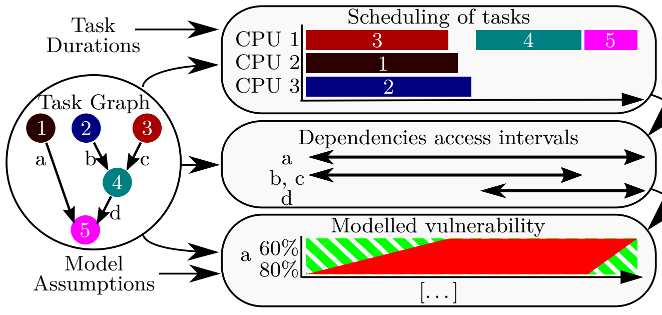


Fig. 3. Vulnerability Modelling
The different steps in vulnerability modelling, displayed in the boxes on the right, only require the information shown on the left side.

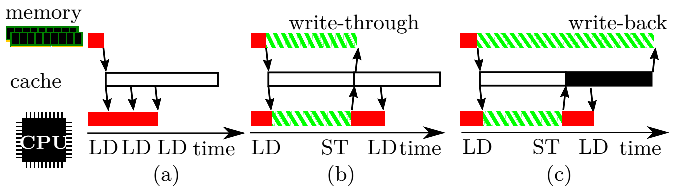


Fig. 4. Difference between vulnerability at memory level, and the value computed from memory accesses
At memory and CPU level we represent in hatched green the safe time, and in solid red the vulnerable time. The boxes at cache level represent the time that the considered data resides in cache: in white, clean, and in black, modified. In the first scenario, the data is not modified, while in the second it is directly written through to memory. In the last scenario, the modified data needs to be written back to memory. All scenarios show either more vulnerable time at CPU level, or more safe time at memory level.

the time at which dependencies will be accessed. This is illustrated by the arrows beneath the scheduling representation, which show for each dependency the interval of time during which they may be accessed.

While many benchmarks have rather uniform task execution times, this is not true in the general case. The execution time of tasks is, however, rather uniform across instances of the same task type, i.e. tasks that execute the same code but with different inputs and outputs. Therefore, *we model the task durations with a single value per task type*, that can be obtained by the runtime system as the duration of the first instance of that type.

B. Additional Model Assumptions

The task graph analysis only provides us with intervals of time during which each dependency may be written and/or read. Further information that is not readily available to the runtime system is required to predict memory access patterns exactly. In particular, the memory access patterns of each task and how they are affected by caching. In the interest of maintaining the model lightweight enough to be used online, we replace expensive profiling by a number of assumptions. Our evaluation shows that these assumptions are sufficient to guarantee a satisfying accuracy with respect to the ordering of memory pages per vulnerability within each benchmark, as presented in detail in Section VII.

Together with the predicted scheduling, knowing how data is accessed within each task would give the runtime system the full picture of the program's memory access patterns. Instead, we model all tasks in the same way: *we suppose each task accesses all of its data linearly over its execution.* That is, if an array `a` of 100 elements is accessed as input by a task, the task will start by accessing `a[0]`, access `a[50]` roughly half way through its execution time, and end by accessing `a[99]`. With this assumption, we can predict from the task scheduling exactly how much time elapses between two accesses to any memory location.

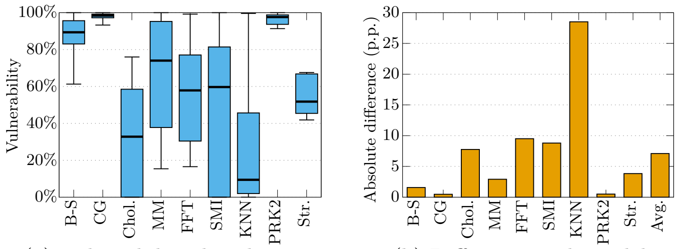
The effects of caching on how main memory perceives a program's memory access patterns have been modelled precisely by Yu *et al.* [36]. Their model exploits data like working set size or cache capacity, as well as data that is not available to the runtime system, such as the precise memory access pattern information (element size, access stride, etc.). Additionally, this model is expensive to compute, and caching effects can only significantly affect vulnerability for data structures that are smaller than the caches. Instead, we conservatively *estimate vulnerability from the times at which memory accesses are issued*, setting aside caching effects for future work.

We illustrate in Figure 4 the fact that vulnerability, when computed from the CPU perspective ignoring any caching effects, is an upper bound of the vulnerability in memory. This figure displays the vulnerability at the memory and CPU levels and the time data spends in cache, based on loads and stores. Whether fetched lines are not modified, modified with immediate or with delayed writing to memory, the data is always rated vulnerable for a longer period than it is vulnerable in memory. The difference between vulnerable time measured at CPU and memory levels is at most the duration for which the data resides in the cache hierarchy.

Owing to the simplicity of our model, the vulnerability of each data dependency can be entirely represented *using only two vulnerability values.* All vulnerability values in a data dependency can be interpolated linearly from the first and last values in this range. This is illustrated at the bottom of Figure 3, where the vulnerable time of the array `a` is depicted by the red trapezoidal area. In the model, this information is represented by the two values 80% and 60%. The first (respectively last) address in `a` is accessed at the start (resp. end) of tasks 3 and 5. Vulnerable time for this location is displayed as the fraction of red at the bottom (resp. top) of the graph, and represents 80% (resp. 60%) of the total time.

C. Vulnerability Distribution

The vulnerability predictions from our model can be compared against exact vulnerability values, measured using the multicore simulator setup described in Section VI. While more details are presented in Section VII-A, we already show in Figure 5 (a) the distributions of mea-



(a) Vulnerability distribution (b) Difference with model

Fig. 5. Vulnerability distribution and average model error

sured vulnerability across the different benchmarks' memory footprints. For each memory page, we compute the average vulnerability value over the whole execution time, and report the distribution of these values per benchmark as box plots. The first and last quartiles of vulnerability values are displayed as whiskers, the two middle quartiles with a box, and the median is represented by a thick line. Figure 5 (a) reveals a wide variety of vulnerability distributions across the various benchmarks: some with high averages such as CG, some with lower averages such as KNN, some with very spread-out values such as FFT, or on the contrary grouped together such as PRK2.

In Figure 5 (b), we plot the average error of our modelling methodology. Each value is computed by averaging the absolute differences between predicted and measured vulnerability values, for each memory page in each benchmark. The differences are reported in percentage points (p.p.), which represent the difference between the two vulnerability values as percentages.[1] The graph reveals that all benchmarks except KNN have an average error of less than 10 p.p., and the average error is of 6.91 p.p., meaning that our model is very accurate. In the case of KNN the error is close to 27 p.p., because one of our simplifying assumptions is not met: this benchmark's tasks access all of their input data immediately and repeatedly over their execution time. While this causes a relatively high average error, it does not perturb the relative ordering of vulnerability per memory page, which means that the model's choice of which pages to protect with stronger ECC is correct, as detailed in Section VII.

We do not observe significant accumulation of estimation errors over program execution time in any benchmark used, however we expect such effects might occur for very deep task graphs. This can in turn be counteracted easily thanks to the light weight of the model, simply by repeating the analysis periodically and limiting it to the upcoming section of the task graph.

V. VECC as a Dynamically Adaptable ECC

Using our modelled vulnerability, we can now select memory pages for increased protection at execution time. This requires a vulnerability threshold, and a two-tier ECC scheme that allows protection to be adjusted. Given a desired fraction of memory pages to protect, the threshold is simply the corresponding quantile in the distribution of predicted vulnerability values.

For the two-tier ECC scheme, we use VECC [35], which is the most appropriate of the several existing two-tier ECC schemes for dynamically adjusting protection at runtime. Its two-level organisation with a ChipKill baseline allows protecting with at least this state-of-the-art technique the whole memory of an application, without additional overhead. The second tier of redundancy is stored in addressable memory, and can be applied only to those parts of memory deemed most vulnerable.

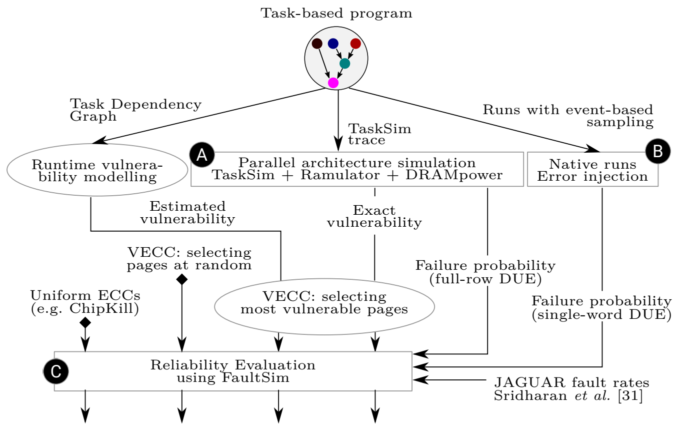
Furthermore, protection is decided at the granularity of a memory page, rather than fixed 256KB blocks as in Odd-ECC [16]. VECC's organisation allows fully flexible mappings per memory page at very low overhead, by proposing a translation from physical addresses to second tier ECC addresses, with an ECC address translation cache that is maintained with each core's TLB. Because VECC changes the way ECC information is stored and accessed, it requires specific hardware support beyond its translation cache – for example, ECC codecs that can decode larger words. Since this support is not currently available, our performance evaluation is limited to simulator infrastructures.

Finally, VECC relies on a Reed-Solomon code [28], as multiple ChipKill implementations do [1]. This means that the protection for a memory page can be changed without modifying the baseline redundancy: both codewords (with and without second-tier) are in fact the same larger codeword, shortened to a different number of symbols. Therefore, downgrading the ECC protection is done without overhead, by simply updating the page-table entry and freeing the tier 2 storage. Upgrading the protection of a memory page requires streaming the page's contents to compute the tier 2 symbols, storing them in newly allocated addressable memory, and updating the page-table entry.

As VECC's first redundancy tier is used for error detection, the second tier only needs to be fetched on memory loads in the rare cases that the first level decoding finds errors. On stores however, the second tier redundancy needs to be updated. Since this second tier is stored in addressable memory, Yoon *et al.* propose caching it in the LLC to limit the increase in ECC update traffic [35]. Furthermore, metadata in cache lines can be used to indicate missing second-tier redundancy, to avoid fetching tier 2 data in cache. In practice, this means that whenever data that has second tier protection reaches the LLC in a modified state, the LLC performs the following operations:

1) Check whether the second-tier ECC corresponding to this data is present in the cache.
2) If not, allocate a new cache line for the second-tier ECC, initially marked entirely as invalid.

[1]For example, a prediction of 55% vulnerability for a memory page measured at 50% is an error of 5 p.p., but a 10% relative error.



Task-based program
Task Dependency Graph
TaskSim trace
Runs with event-based sampling
Runtime vulnerability modelling
A
Parallel architecture simulation TaskSim + Ramulator + DRAMpower
Native runs Error injection
B
Estimated vulnerability
Exact vulnerability
VECC: selecting pages at random
Failure probability (full-row DUE)
Failure probability (single-word DUE)
Uniform ECCs (e.g. ChipKill)
VECC: selecting most vulnerable pages
C
Reliability Evaluation using FaultSim
JAGUAR fault rates Sridharan et al. [31]

Fig. 6. Experimental frameworks

This Figure displays the flow of data between the proposed methodologies (shown in ovals) and experimental frameworks (in rectangles) to evaluate the reliability and cost of guiding VECC at Runtime.

TABLE I
Benchmarks used for evaluation

	Benchmark Description	**Input Size**
B-S	Black-Scholes Option Pricing (Partial differential equation)	8M options
CG	Conjugate Gradient (Sparse linear algebra)	3D Poisson equation, 2M×2M elts.
Chol.	Cholesky factorisation (Dense linear algebra)	8K×8K SPD matrix
MM	Double-precision general Matrix Multiply (Dense linear algebra)	4K×4K matrices
FFT	Stockham Fast Fourier Transform (Spectral method)	16M elements, 1 dimension
SMI	Invert Symmetric Matrix (Dense linear algebra)	8K×8K SPD matrix
KNN	K-Nearest Neighbours (Machine learning)	1M points training, 1k testing sets
PRK2	Parallel Research Kernels stencil [32] (Stencil operation)	16K×16K grid, 10 iterations
Str.	Stream Triad [19] (Memory bandwidth benchmark)	192MB: 8M `doubles` arrays

3) Update the second-tier ECC in cache, and mark this data in the cache line as valid and dirty.

VI. Methodology

A number of different frameworks are needed to evaluate all of the aspects of this dynamic ECC proposal. These frameworks are summed up in Figure 6. For all evaluations, we use the 9 parallel benchmarks listed in Table I.

We measure the precision of the proposed vulnerability model and the performance impact of adjusting ECC at runtime, with Framework Ⓐ : the parallel simulator, which will be described in Section VI-A. We also assess the effects on reliability of our dynamic ECC strategy, and this evaluation is subdivided in two parts. First, we evaluate the experimental risk of failure due to errors in each benchmark's memory, for which we use Framework Ⓑ , presented in Section VI-B. These probabilities of failures can then be used as inputs in a monte-carlo simulator that performs lifetime simulations of entire DIMMs in Framework Ⓒ , detailed in Section VI-C.

We explore the various available trade-offs by considering all 9 deciles of vulnerability values. For every run, we select a protection level, which is the number of most vulnerable deciles of memory pages to protect. This protection level then impacts how errors encountered will be corrected by the ECC and how the program accesses data, specifically with respect to updating with every write to memory pages that are selected for stronger protection.

A. Framework Ⓐ : Architectural Simulator

We use a cycle-based simulator, TaskSim [29, 30], to simulate the performance of benchmarks with various VECC protection levels and to compute the exact memory vulnerability ratings. TaskSim is a task-trace based infrastructure that relies on task-based programming models to generate traces of each task, recording the basic blocks and memory addresses accessed. These are then replayed in a multicore architecture simulator with a simple core model and a full cache hierarchy, using a real runtime system to schedule the tasks onto the simulated cores. The memory controller is simulated using Ramulator [14], and the DRAM power consumption is computed from the DRAM command traces that Ramulator generates using DRAMPower [7].

To compute the vulnerability at the memory level, we record all loads and stores at the memory access granularity and the time at which they reach main memory. From this data, we compute the vulnerability for every 64B word in memory, and report the average values per memory page. We also extend TaskSim to protect selected memory pages with VECC which consists simply of allocating memory for second tier symbols and keeping these second-tier VECC symbols' up to date at the LLC level, as explained in Section V.

We trace applications on an Intel x86_64 Xeon Platinum 8160 and simulate a multicore architecture whose configuration mirrors the Xeon Platinum 8160 characteristics with 8 cores and a proportionally scaled LLC. The cores run one thread each, at 2.1GHz, and each with a reorder buffer of 224 entries. The memory hierarchy parameters are summarised in Table II. All cache levels have 64B lines, are write-back and write-allocate, and are non inclusive. Memory pages are 16KB in size. Ramulator and DRAMPower simulate DDR4 DRAM memory with ranks constituted of 8Gb `x4` chips clocked at 3200MHz, with the parameters reported for MICRON's DDR4 SDRAM *MT40A2G4*`xx`*-062E* [22]. For the ChipKill baseline and for VECC, we use 2 channels composed of one 18-chip rank each, thus 72 bits wide. Double ChipKill uses 2 channels in lockstep to provide a channel with a 144 bits wide data path. We simulate this in Ramulator by using 1 single-rank channel of 36 chips. Both configurations use 36 chips and 32GB of addressable memory.

TABLE II
TaskSim cache parameters

MSHR stands for miss status handling register.

cache	shared	assoc.	size	latency	MSHRs
L1D	private	8-way	32kB	4 cycles	16
L2	private	16-way	1MB	13 cycles	16
L3	shared	16-way	11MB	68 cycles	32

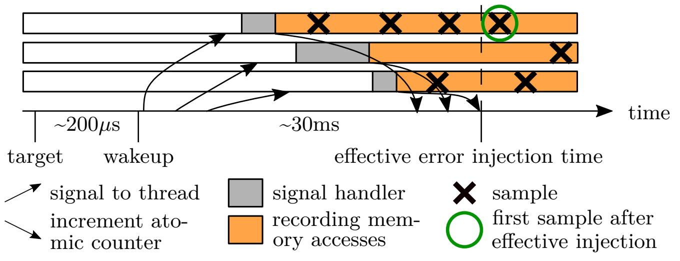

time
~200μs
~30ms
target
wakeup
effective error injection time
signal to thread
increment atomic counter
signal handler
recording memory accesses
sample
first sample after effective injection

Fig. 7. Error injection framework

The effective error injection time is the moment when all threads have started recording samples. The first subsequent sample determines the error injection outcome.

B. Framework Ⓑ : Error Injection Methodology

The objective of our error injection methodology is to evaluate how the runtime-guided VECC decisions impact the lifetime reliability of real systems. To this end, we want to concretely implement the modalities of failures due to errors in memory. In real systems, data is fetched from memory with the redundant information that, together, constitute a codeword. Based on the number and location faults in the codeword, the ECC decoding may succeed, and return the correct data, or fail.

Our error injection experiments quantify the risk of application failure, in the case of an ECC decoding failure. That is, we want to measure as precisely as possible the risk of consuming an uncorrected error (UE). We therefore perform error injections in native runs on the same real system as the one used for tracing and simulating in Section VI-A. Each experiment consists of selecting at random a memory address in the program's memory footprint and a time in the program's execution, which are the location and moment where the UE appears. We then measure how this data location is accessed after this point in time. If the data is not accessed or it is overwritten, we consider the UE as tolerated, whereas if the erroneous data is consumed by the application, the error is considered to cause a failure. Considering the next access to data is the only correct way to measure the worst-case risk of consuming an UE. Indeed, any data may have to be fetched from memory even if it is expected to be cached at that moment, due to, e.g., an unexpected eviction.

The mechanism of these error injections relies on precise event-based sampling (PEBS), and is depicted in Figure 7. Using operating system mechanisms such as signal handlers, we wait for the selected random amount of time and then record the memory accesses to the targeted memory address. After the end of the program's execution, we parse the list of recorded memory accesses and the time at which they occurred according to the high-resolution hardware clock. As illustrated in Figure 7, we consider the first sample after the effective error injection time, and define whether the error was consumed or tolerated based on this memory access.

We have performed over 30000 error injections per benchmark, in order to obtain for each of the 10 runtime-decided vulnerability categories the probability of consuming an UE with small confidence intervals (less than ±2.5% with 99% confidence).

C. Framework Ⓒ : Monte-Carlo Fault Simulator

The third and last framework is used to compute the lifetime reliability of DIMMs. Computing this lifetime reliability analytically would be extremely complex, as the distribution of faults is long-tailed, in particular a Pareto distribution [20], and failures are never caused by a single fault, but by a number of faults intersecting in a single codeword. Therefore, we rely instead on FaultSim [24], which assesses reliability via Monte-Carlo simulations.

For each ECC and workload configuration, we run 1M of FaultSim simulations, each corresponding to the lifetime of a single DIMM. The DIMMs have the same characteristics as used in Ramulator in Section VI-A. In each simulation, FaultSim injects the various types of possible hardware faults randomly, with exponential rates taken from Sridharan *et al.* [31] scaled by 10×. For every fault injection, the simulator then considers whether its physical location affects the same codewords as other faults, and for every intersection of faults the simulator determines if the decoding succeeds. The simulator continues simulating until it encounters an uncorrected error or reaches the end of the DIMM lifetime, which we have set to 7 years.

We first extend FaultSim to take into account application-level error tolerance. Given a physical location of an uncorrected error, we want to determine whether the program can tolerate this error. For errors that have a size of less than a single ECC word, we use the probabilities of consuming uncorrected errors obtained from the error injection experiments presented in Section VI-B. However, larger errors may appear in DRAM: a full DRAM row, column, bank, or rank, may fail at once. In the case of fault intersections that span full DRAM columns, banks, or ranks, we always consider the application-level tolerance to fail. Indeed, data in a single DRAM row may correspond to a small number of DRAM pages (16KB in the system we simulate), however separate rows have a very low likelihood of containing related data. This means that if P is the probability of tolerating a word failure (respectively a row failure), the probability of tolerating a full columns (resp. bank) failure is P^{rows}, thus negligible with a high number of rows (128K per bank in our system).

The last error granularity to consider is a DRAM row error, which corresponds to a memory page with our 16KB pages. The probability that a full memory page

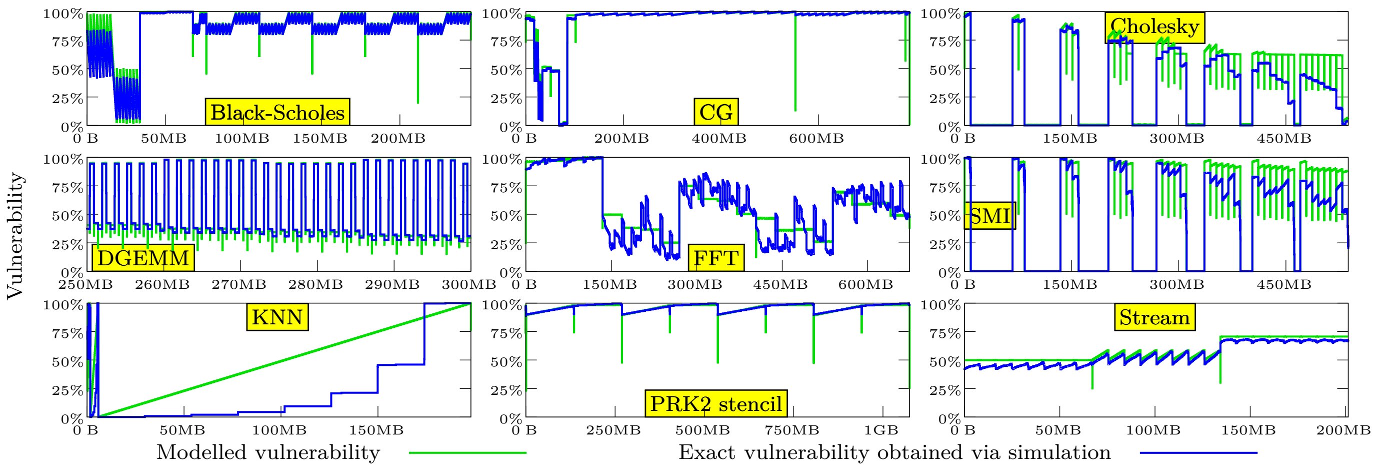


Fig. 8. Distribution per memory page of modelled vulnerability, and exact vulnerability obtained by simulation
Each graph displays the vulnerabilities over the memory footprint of a benchmark, with pages sorted by their virtual addresses.

error can be tolerated by a program is not necessarily 0. Furthermore, it can not be easily determined from the ECC word failure probability, as words in the same page can not be considered to be always accessed independently (nor always accessed together). Therefore, we use the vulnerability computed at a DRAM row granularity as the probability of consuming a row error. That is, we compute for every DRAM row, using the simulator framework Ⓐ described in Section VI-A, the average fraction of time that *any value in the row has a load as its next access*. This is different from (and always higher than) the average vulnerability of the ECC words in the row.

Finally, we extend FaultSim to implement VECC, which has the same ECC layout and software-level error tolerance as ChipKill, except for errors that affect exactly 2 chips in the same ECC codeword. These errors can not be corrected by ChipKill, but can be corrected using the second tier ECC, if present. In this case, we draw at random whether the virtual page mapped at the error's physical location is selected for 2-tier protection, based on the selected fraction of protected memory. If so, we consider the error corrected. Otherwise, we apply the probability of tolerating this error at the software level. For non-random selections of 2-tier protected memory pages, we now have to consider the conditional probability of consuming an error *knowing that this memory page is not selected.*

Thus combining together TaskSim, FaultSim, and error injections in native runs, we are able to provide a reliability evaluation of guiding VECC protection at runtime.

VII. Evaluation

A. Vulnerability Model Accuracy

The distribution of vulnerability across benchmarks has been presented at a high level in Section IV-C. Here, we take a more detailed look at the vulnerability values of the memory pages in each benchmark, and the differences between these values and our model's previsions. These are presented in Figure 8, with pages ordered by their virtual addresses, thus in an order that maintains data structures contiguity and allows discussing application-level behaviour. We present the full memory footprint for all benchmarks, except DGEMM where we only show the 250MB-300MB range of its 415MB memory footprint. DGEMM has a high variability in vulnerability values of address ranges that are next to each other, and the full range is thus too noisy to display in full.

Vulnerability values exhibit spatial locality, with neighbouring addresses grouped in ranges of similar vulnerability values. Such ranges have either the same vulnerability value, such as the 8 blocks of about 48MB in KNN, or neighbouring values that monotonically increase over the block, such as the 24 blocks that can be distinguished in Stream's vulnerability profile. This first observation validates our choice to store the vulnerability of a dependency with only 2 values. Both constant and monotonous ranges of values can indeed be precisely represented by the first and last value in the range, interpolating linearly the values in between. This locality is due to the fact that neighbouring addresses correspond to contiguous memory pages in the application virtual address space, and thus pages in the same data structure.

For example, the matrix in CG corresponds to close to 90% of the memory footprint, and occupies the highest addresses. In Stream, we have 3 arrays that are each subdivided into 8 dependencies. Both Cholesky and SMI handle non-sparse symmetric positive definite matrices, split into 64 blocks of equal size. As can be seen in both profiles, 36 of these blocks have non-zero vulnerabilities, while 28 are always zero, which is due to the fact that the matrices are symmetric. Both algorithms only access the blocks on and above the matrix diagonal, as the blocks below the diagonal are simply a transpose of the blocks above the diagonal. Only FFT displays a very noisy vulnerability profile in most of its memory footprint. This stems from its memory access pattern, which accesses arrays with a different stride at every iteration.

The comparison of these memory page vulnerability values with the model predictions highlights that the model has high accuracy. In particular, the most visible modelling errors on Figure 8 are on the boundaries of application-level structures. This is the case for all the erroneous predictions in DGEMM and PRK2 Stencil. These errors are in fact only artefacts, due to application-level data structures not being aligned on memory page boundaries. On the 5 most accurately modelled benchmarks, Black-Scholes, CG, DGEMM, PRK2 Stencil, and Stream, 98% of memory pages have errors of less than 7.31 p.p., and 90% of pages have errors less than 3.34 p.p. This means that our very simple model is able to determine how memory is used for a huge majority of the memory pages in these benchmarks. Significant mispredictions happen for KNN, where the assumption that tasks access their dependencies linearly over their execution time is not met, causing a step-like vulnerability profile. However, the ordering of all 12079 memory pages in KNN is exactly the same between vulnerability predictions and exactly measured vulnerability values. Therefore, our model prioritises exactly the same pages for protection than the simulation.

Further mispredictions happen for FFT, Cholesky and SMI, as 25% of memory pages have an error of at least 15 p.p. in each of these benchmarks. For Cholesky and SMI, these imprecisions are mostly due to less structured parallelism, and thus more variability in task execution times, which does not significantly alter the ordering of memory pages. The memory page selections made using the model and exact vulnerability differ by at most 2.77% for Cholesky, and 3.71% for SMI. For FFT, the predictions exhibit coarser spatial locality than the exact vulnerability values. This is again due to FFT's strided memory access pattern, causing a very noisy vulnerability profile that the model can not predict correctly. The ordering of memory pages is correctly predicted outside of the 120MB most vulnerable FFT data, which represents 18% of the memory footprint. Within these most vulnerable memory regions, the model wrongly prioritises up to 22% of the pages for protection, whereas this difference is at most 0.62% when protecting more than 18% of memory with tier 2 ECC. However, memory pages that are wrongly prioritised only affect reliability if they replace pages with significantly higher vulnerability. Therefore, the ordering errors in most vulnerable FFT pages have little impact, as all those pages have a similar (over 91%) vulnerability.

B. Reliability Effect of Runtime-Guided VECC

Figure 9 presents our reliability evaluation as the probability of having a DIMM fail in its lifetime, for all benchmarks and all considered protection strategies. Horizontal lines represent the uniform ECCs: ChipKill and the stronger Double ChipKill. The remaining lines represent VECC, each with a different strategy for selecting memory pages for tier 2 protection, where the x axis represents the fraction of 2-tier protected memory pages. The *Oracle* strategy selects memory pages to protect offline using the vulnerability numbers from the tracing and architectural simulating infrastructure, while the *Random* strategy selects pages at random. The *Runtime-Guided* strategy, detailed in Section V, makes its protection decisions based on the vulnerability model evaluated in Section VII-A.

The random strategy's reliability gains exhibit a linear decrease in DIMM failure probability as more of the application's footprint is protected. With 0% of the memory protected, the failure probability of VECC is the same as ChipKill, and with close to 100% it is approaching but a little higher than Double ChipKill. This is due to guiding VECC at the page level, as protecting all of memory with VECC uniformly would have a similar failure rate to Double ChipKill. However, due to the large number of independent memory pages mapped in a single column (128K in our system), the chances of tolerating full column or bank errors affecting 2 chips remains very small, even when we approach 100% coverage. For example, when VECC covers 99.9% of memory pages, a single column error has a $0.999^{131072} \approx 10^{-57}$ chance of affecting only protected pages. Therefore, our methodology never covers these full-column 2-chip errors, as explained in Section VI-C. These rare events cause the difference between guided VECC and Double ChipKill.

The *Oracle* and *Runtime-Guided* VECC strategies both have lower failure probabilities than the *Random* strategy for most benchmarks, and similar probabilities otherwise. Furthermore, the *Runtime-Guided* tracks the *Oracle* reliability results very closely. The point with most difference between both strategies is for FFT with 10% coverage, where the *Oracle* strategy achieves a failure rate of 1.69% against a 1.72% for the *Runtime-Guided* strategy. This 0.03 p.p. gap represents a 1.92% difference. The *Oracle* outperforms the *Random* strategy by selecting which memory pages to protect in priority using the vulnerability metric designed for ECC memory. Our runtime-guided strategy is able to perform nearly the same optimal choices, thanks to the precision of its vulnerability model.

In SMI and Cholesky, the *Oracle* and *Runtime-Guided* strategies both identify that the last pages to protect are those with the unused matrix blocks, thereby reaching the optimal reliability level with only 56% of memory covered. Furthermore, the performance of both non-random VECC strategies is correlated with the variety in vulnerability values as shown on the profiles in Figure 8. KNN, DGEMM, Cholesky, and SMI all exhibit a wide range of vulnerability values, and have the greatest decreases in failure rates with respect to the *Random* strategy, ranging from 0.19 p.p. to 0.29 p.p. This represents a 15% to 25% reduction in failure rate. For Black-Scholes, CG, Stream, and FFT the highest failure rate reductions range from 5% to 7%, while for PRK2, the benchmark with the most grouped vulnerability values, the benefit of a runtime-

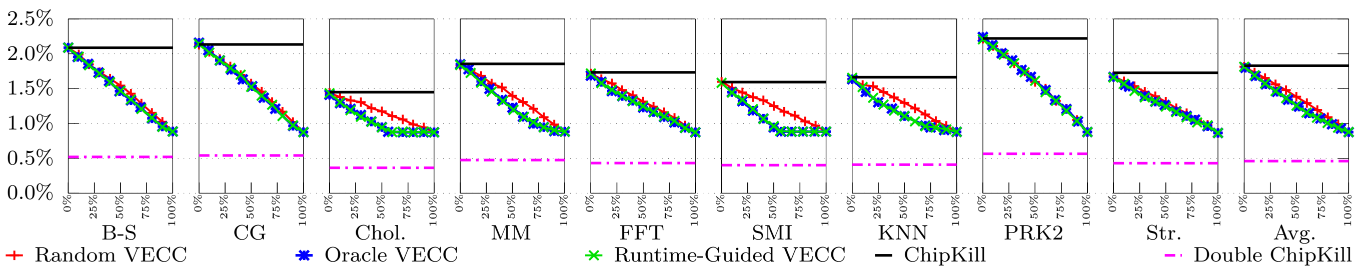


Fig. 9. Probability of DIMM failure after 7 years of use given various reliability strategies

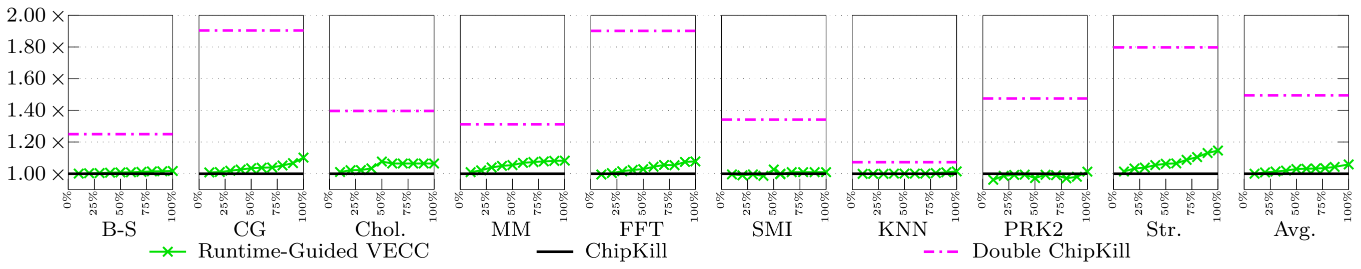


Fig. 10. Energy consumption with respect to ChipKill

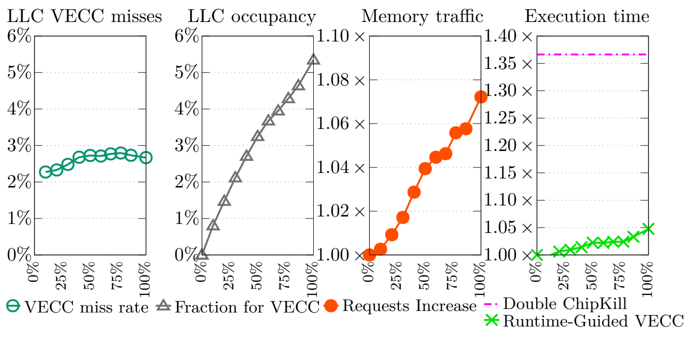


Fig. 11. Average performance impact of VECC

guided strategy is at most a 3% lower failure rate than the random strategy.

The main takeaways are that differences in vulnerability between different parts of memory allow interesting redundancy-reliability trade-offs to be exploited, and that our runtime technique identifies these trade-offs as well as expensive architectural simulation analysis.

C. *Overheads of Runtime-Guided VECC*

The energy cost of each ECC scheme with respect to ChipKill is presented in Figure 10. The average cost of applying VECC progresses linearly with the fraction of memory protected, and remains below 1.06× that of ChipKill. The stronger uniform Double ChipKill ECC is much more costly, consuming 1.49× the energy of ChipKill. This means that upgrading from ChipKill to Double ChipKill requires at least 4 times more additional energy than upgrading to VECC. This is because Double ChipKill always activates twice more chips on every access than ChipKill, which makes every memory access more expensive. Secondly, due to its different organisation, there is also much less memory parallelism available. This has an especially high impact on the execution time of memory-bound benchmarks. For CG, FFT, and Stream, Double ChipKill consumes between 1.80× and 1.90× more energy than ChipKill, while VECC stays below 1.15×.

Furthermore, on benchmarks where read-only data is prioritised for protection first, the cost of VECC remains small at first. In CG for example, the matrix is protected first, and the energy cost of VECC is 1.05× that of ChipKill when protecting 80% of the footprint. When the last 20% of memory is protected with tier-2 ECC as well, the energy cost increases faster, up to 1.12×. This is because this portion of CG memory contains the vectors, and the cost of VECC is mainly driven by stores to protected data.

Finally, we present in Figure 11 a number of performance indicators averaged over all the benchmarks. The miss rate of second tier redundancy at the LLC level (*LLC VECC misses*) remains relatively constant on average, between 2% and 3%. The other metrics, which are the fraction of LLC occupied by tier-2 redundancy (*LLC occupancy*), and the increase in memory requests and execution time with respect to ChipKill, increase roughly linearly with the fraction of memory that is protected. The average increase in execution time is much less for VECC than for Double ChipKill, due to channel-level parallelism. Indeed, Double ChipKill operates 2 channels in lockstep, whereas ChipKill and VECC can operate these 2 DRAM channels independently. For CG, Double ChipKill incurs a 1.81× slowdown against a maximum of 1.11× for VECC. Stream exhibits the worst-case cost of VECC, as it writes to all of its memory footprint at every iteration, with up to 1.15× slowdown, versus 1.70× for Double ChipKill, On the other extreme, Black-Scholes only incurs up to

1.005× slowdowns for VECC, against 1.014× with Double ChipKill.

All in all, using VECC is cheaper than Double ChipKill on all cost and performance metrics, while approaching similar reliability. Guiding VECC can be done online by modelling key memory statistics such as vulnerability, thereby identifying the best reliability-cost trade-offs that are achievable with VECC. Runtime-guided VECC makes the most of the runtime-guidance when protecting around 50% of memory, as this configuration has the highest average reliability gap with the randomly guided VECC. This configuration also sees runtime-guided VECC achieve a significant reliability improvement with 31% reduction of failure rate (from 1.84% down to 1.26%), while limiting both the average performance slowdown and energy increase to 1.02× and 1.03× respectively.

VIII. Conclusion

In this paper we present a methodology to estimate memory vulnerability online and to guide a dynamically adaptable ECC scheme, in order to adjust memory protection accordingly. The runtime modelling is very lightweight, and relies entirely on information to which the runtime system already has access. Furthermore, this modelling gives a faithful picture of the vulnerability in memory down to the memory page level with less than 7 percentage points difference on average.

Guiding the dynamically adaptable Virtualized ECC scheme [35] with the runtime's vulnerability estimation shows that it can achieve a variety of high-quality trade-offs between reliability and overhead, as this technique performs as well as an oracle that uses an offline analysis built on tracing and detailed architectural simulation. Our holistic evaluation of the lifetime reliability of DIMMs also shows that this VECC strategy can approach the reliability of the stronger uniform Double ChipKill ECC with much lower performance degradation and energy cost.

Our proposed runtime vulnerability methodology can be extended in various ways, either repurposed (for e.g. guiding data placement or optimising DRAM refreshes), or expanded with a more complex model of memory access patterns. For example, the assumption that all tasks operate linearly on their inputs could be replaced by compiler-detected access patterns, or by using random sampling of memory accesses to accept and reject task behaviour models.

Acknowledgements

This work has been partially supported by the RoMoL ERC Advanced Grant (GA 321253), the European HiPEAC Network of Excellence, and the Spanish Government (contract TIN2015-65316-P). L. Jaulmes has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2013/06982. M. Moretó and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and RYC-2017-23269. This material is based in part upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Numbers DE-SC0014097 and DE-SC0014098.

References

[1] Advanced Micro Devices (AMD), Inc., "BIOS and Kernel Developer's Guide for AMD Family 15h Models 60h-6Fh Processors," Publication # 50742, 2016, Rev. 3.05.

[2] Alaa R. Alameldeen, Ilya Wagner, Zeshan Chishti, Wei Wu, Chris Wilkerson, and Shih-Lien Lu, "Energy-efficient Cache Design Using Variable-strength Error-correcting Codes," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA, 2011, pp. 461–472. doi: 10.1145/2000064.2000118.

[3] Berry Bauer Michael, Sean Treichler, Elliott Slaughter, and Alex Aiken, "Legion," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012, doi: 10.1109/SC.2012.71.

[4] Leonardo Bautista-Gomez, Ferad Zyulkyarov, Osman Unsal, and Simon McIntosh-Smith, "Unprotected Computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, 2016, 55:1–55–11. doi: 10.1109/SC.2016.54.

[5] Sanguhn Cha, Seongil O, Hyunsung Shin, Sangjoon Hwang, Kwangil Park, Seong Jin Jang, Joo Sun Choi, Gyo Young Jin, Young Hoon Son, Hyunyoon Cho, Jung Ho Ahn, and Nam Sung Kim, "Defect Analysis and Cost-Effective Resilience Architecture for Future DRAM Devices," in *IEEE 23rd International Symposium on High Performance Computer Architecture*, HPCA, 2017, pp. 61–72. doi: 10.1109/HPCA.2017.30.

[6] B.L. Chamberlain, D. Callahan, and H.P. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007, doi: 10.1177/1094342007078442.

[7] Karthik Chandrasekar, Benny Akesson, and Kees Goossens, "Improved Power Modeling of DDR SDRAMs," in *14th Euromicro Conference on Digital System Design*, DSD '11, 2011, pp. 99–108, doi: 10.1109/DSD.2011.17.

[8] Timothy J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," IBM Microelectronics Division, white paper, 1997.

[9] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas, "OmpSs," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011, doi: 10.1142/S0129626411000151.

[10] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta, "Reliability-Aware Data Placement for Heterogeneous Memory Architecture," in *IEEE 24th International Symposium on High Performance Computer Architecture*, HPCA, 2018, pp. 583–595. doi: 10.1109/HPCA.2018.00056.

[11] Luc Jaulmes, Miquel Moreto, Mateo Valero, and Marc Casas, "A Vulnerability Factor for ECC-protected Memory," in *25th IEEE International On-Line Testing Symposium*, IOLTS 2019, 2019, p. 6,

[12] JEDEC Solid State Technology Association, "DDR4 SDRAM," JEDEC Standard JESD79-4B, 2017, Revision B.

[13] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar, "Near-threshold voltage (NTV) design — Opportunities and challenges," in *Proceedings of the 49th annual Design Automation Conference*, DAC, 2012, pp. 1149–1154. doi: 10.1145/2228360.2228572.

[14] Yoongu Kim, Weikun Yang, and Onur Mutlu, "Ramulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016, doi: 10.1109/LCA.2015.2414456.

[15] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *44th International Confer-*

ence on Dependable Systems and Networks, DSN, 2014, pp. 467–478. DOI: 10.1109/DSN.2014.50.

[16] Alirad Malek, Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis, "Odd-ECC," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS, 2017, pp. 96–111. DOI: 10.1145/3132402.3132443.

[17] Madhavan Manivannan, Vassilis Papaefstathiou, Miquel Pericàs, and Per Stenström, "RADAR," in *IEEE 22nd International Symposium on High Performance Computer Architecture*, HPCA, 2016, pp. 644–656. DOI: 10.1109/HPCA.2016.7446101.

[18] Timothy G. Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlić, Sanjay Chatterjee, Josh Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, Benoît Meister, Brian Nickerson, Nick Pepperling, Bala Seshasayee, Sagnak Tasirlar, Justin Teller, and Nick Vrvilo, "The Open Community Runtime," in *IEEE High Performance Extreme Computing*, HPEC, 2016, pp. 1–7, DOI: 10.1109/HPEC.2016.7761580.

[19] John D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[20] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers," in *45th International Conference on Dependable Systems and Networks*, DSN 2015, 2015, pp. 415–426. DOI: 10.1109/DSN.2015.57.

[21] Micron Technology, Inc., "ECC Brings Reliability and Power Efficiency to Mobile Devices," white paper, 2017.

[22] Micron Technology, Inc., "8Gb: x4, x8, x16 DDR4 SDRAM," Data Sheet CCMTD-1725822587-9875, 2019, Rev. P 04/19 EN.

[23] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, 2003, pp. 29–. DOI: 10.1109/MICRO.2003.1253181.

[24] Prashant J. Nair, David A. Roberts, and Moinuddin K. Qureshi, "FaultSim," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 4, 44:1–44:24, 2015, DOI: 10.1145/2831234.

[25] OpenMP Architecture Review Board, "OpenMP Application Programming Interface," Specification, 2013, Version 4.0.

[26] Vassilis Papaefstathiou, Manolis G.H. Katevenis, Dimitrios S. Nikolopoulos, and Dionisios Pnevmatikatos, "Prefetching and Cache Management Using Task Lifetimes," in *Proceedings of the 27th International Conference on Supercomputing*, ICS, 2013, pp. 325–334. DOI: 10.1145/2464996.2465443.

[27] S. Paul, F. Cai, X. Zhang, and S. Bhunia, "Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 20–34, 2011, DOI: 10.1109/TC.2010.203.

[28] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960, DOI: 10.1137/0108018.

[29] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero, "On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 36:1–36:20, 2012, DOI: 10.1145/2086696.2086715.

[30] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramirez, and Mateo Valero, "Trace-driven simulation of multithreaded applications," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2011, pp. 87–96. DOI: 10.1109/ISPASS.2011.5762718.

[31] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi, "Memory Errors in Modern Systems," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XX, 2015, pp. 297–310. DOI: 10.1145/2694344.2694348.

[32] R. F. Van der Wijngaart and T. G. Mattson, "The Parallel Research Kernels," in *IEEE High Performance Extreme Computing Conference*, HPEC, 2014, pp. 1–6. DOI: 10.1109/HPEC.2014.7040972.

[33] Mark Wilkening, Vilas Sridharan, Si Li, Fritz Previlon, Sudhanva Gurumurthi, and David R. Kaeli, "Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 47, 2014, pp. 293–305. DOI: 10.1109/MICRO.2014.15.

[34] Thomas Willhal. (2014). "Independent Channel vs. Lockstep Mode – Drive your Memory Faster or Safer," [Online]. Available: https://software.intel.com/en-us/blogs/2014/07/11/independent-channel-vs-lockstep-mode-drive-you-memory-faster-or-safer (visited on Jan. 30, 2020).

[35] Doe Hyun Yoon and Mattan Erez, "Virtualized and Flexible ECC for Main Memory," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, 2010, pp. 397–408. DOI: 10.1145/1736020.1736064.

[36] Li Yu, Dong Li, Sparsh Mittal, and Jeffrey S. Vetter, "Quantitatively Modeling Application Resilience with the Data Vulnerability Factor," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, 2014, pp. 695–706. DOI: 10.1109/SC.2014.62.

[37] Liu Yuan, Huaida Liu, Pingui Jia, and Yiping Yang, "An adaptive ECC scheme for dynamic protection of NAND Flash memories," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP, 2015, pp. 1052–1055. DOI: 10.1109/ICASSP.2015.7178130.