Exploiting Task-Based Programming Models for Resilience

Luc Jaulmes

Barcelona, 2019

Advisors: **Marc Casas Guix,**
Miquel Moretó Planas

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Abstract

Hardware errors become more common as silicon technologies shrink and become more vulnerable, especially in memory cells, which are the most exposed to errors. Permanent and intermittent faults are caused by manufacturing variability and circuits ageing. While these can be mitigated once they are identified, their continuous rate of appearance throughout the lifetime of memory devices will always cause random unexpected errors. In addition, transient faults are caused by effects such as radiation or small voltage/frequency margins, and there is no efficient way to shield against these types of events.

Other constraints related to the diminishing sizes of transistors, such as power consumption and memory latency have caused the microprocessor industry to turn to increasingly complex processor architectures. To solve the difficulties arising from programming such architectures, new programming models have emerged that rely on runtime systems. These systems form a new intermediate layer on the hardware-software abstraction stack, that performs tasks such as distributing work across computing resources: processor cores, accelerators such as GPUs and FPGAs, etc. These runtime systems dispose of a lot of information, both from the hardware and the applications, and offer thus many possibilities for optimisations.

This thesis proposes solutions to the increasing fault rates in memory, across multiple resilience disciplines, from algorithm-based fault tolerance to hardware error correcting codes, through OS reliability strategies. These solutions rely for their efficiency on the opportunities presented by runtime systems.

The first contribution of this thesis is an algorithmic-based resilience technique, allowing to tolerate detected errors in memory. This technique allows to recover data that is lost by performing computations that rely on simple redundancy relations identified in the program. The recovery is demonstrated for a family of iterative solvers, the Krylov subspace methods, and evaluated for the conjugate gradient solver. The runtime can transparently overlap the recovery with the

computations of the algorithm, which allows to mask the already low overheads of this technique.

The second part of this thesis proposes a metric to characterise the impact of faults in memory, which outperforms state-of-the-art metrics in precision and assurances on the error rate. This metric reveals a key insight into data that is not relevant to the program, and we propose an OS-level strategy to ignore errors in such data, by delaying the reporting of detected errors. This allows to reduce failure rates of running programs, by ignoring errors that have no impact.

The architectural-level contribution of this thesis is a dynamically adaptable Error Correcting Code (ECC) scheme, that can increase protection of memory regions where the impact of errors is highest. A runtime methodology is presented to estimate the fault rate at runtime using our metric, through performance monitoring tools of current commodity processors. Guiding the dynamic ECC scheme online using the methodology's vulnerability estimates allows to decrease error rates of programs at a fraction of the redundancy cost required for a uniformly stronger ECC. This provides a useful and wide range of trade-offs between redundancy and error rates.

The work presented in this thesis demonstrates that runtime systems allow to make the most of redundancy stored in memory, to help tackle increasing error rates in DRAM. This exploited redundancy can be an inherent part of algorithms that allows to tolerate higher fault rates, or in the form of dead data stored in memory. Redundancy can also be added to a program, in the form of ECC. In all cases, the runtime allows to decrease failure rates efficiently, by diminishing recovery costs, identifying redundant data, or targeting critical data. It is thus a very valuable tool for the future computing systems, as it can perform optimisations across different layers of abstractions.

Resumen

Los errores en memoria se vuelven más comunes a medida que las tecnologías de silicio reducen su tamaño. La variabilidad de fabricación y el envejecimiento de los circuitos causan fallos permanentes e intermitentes. Aunque se pueden mitigar una vez identificados, su continua tasa de aparición siempre causa errores inesperados. Además, la memoria también sufre de fallos transitorios contra los cuales no se puede proteger eficientemente. Estos fallos están causados por efectos como la radiación o los reducidos márgenes de voltaje y frecuencia.

Otras restricciones coetáneas, como el consumo de energía y la latencia de la memoria, obligaron a las arquitecturas de computadores a volverse cada vez más complejas. Para programar tales procesadores, se desarrollaron modelos de programación basados en entornos de ejecución. Estos sistemas forman una nueva abstracción entre hardware y software, realizando tareas como la distribución del trabajo entre recursos informáticos: núcleos de procesadores, aceleradores, etc. Estos entornos de ejecución disponen de mucha información tanto sobre el hardware como sobre las aplicaciones, y ofrecen así muchas posibilidades de optimización.

Esta tesis propone soluciones a los fallos en memoria entre múltiples disciplinas de resiliencia, desde la tolerancia a fallos basada en algoritmos, hasta los códigos de corrección de errores en hardware, incluyendo estrategias de resiliencia del sistema operativo. La eficiencia de estas soluciones depende de las oportunidades que presentan los entornos de ejecución.

La primera contribución de esta tesis es una técnica a nivel algorítmico que permite corregir fallos encontrados mientras el programa su ejecuta. Para corregir fallos se han identificado redundancias simples en los datos del programa para toda una clase de algoritmos, los métodos del subespacio de Krylov (gradiente conjugado, GMRES, etc). La estrategia de recuperación de datos desarrollada permite corregir errores sin tener que reinicializar el algoritmo, y aprovecha el

modelo de programación para superponer las computaciones del algoritmo y de la recuperación de datos.

La segunda parte de esta tesis propone una métrica para caracterizar el impacto de los fallos en la memoria. Esta métrica supera en precisión a las métricas de vanguardia y permite identificar datos que son menos relevantes para el programa. Se propone una estrategia a nivel del sistema operativo retrasando la notificación de los errores detectados, que permite ignorar fallos en estos datos y reducir la tasa de fracaso del programa.

Por último, la contribución a nivel arquitectónico de esta tesis es un esquema de Código de Corrección de Errores (ECC por sus siglas en inglés) adaptable dinámicamente. Este esquema puede aumentar la protección de las regiones de memoria donde el impacto de los errores es mayor. Se presenta una metodología para estimar el riesgo de fallo en tiempo de ejecución utilizando nuestra métrica, a través de las herramientas de monitorización del rendimiento disponibles en los procesadores actuales. El esquema de ECC guiado dinámicamente con estas estimaciones de vulnerabilidad permite disminuir la tasa de fracaso de los programas a una fracción del coste de redundancia requerido para un ECC uniformemente más fuerte.

El trabajo presentado en esta tesis demuestra que los entornos de ejecución permiten aprovechar al máximo la redundancia contenida en la memoria, para contener el aumento de los errores en ella. Esta redundancia explotada puede ser una parte inherente de los algoritmos que permite tolerar más fallos, en forma de datos inutilizados almacenados en la memoria, o agregada a la memoria de un programa en forma de ECC. En todos los casos, el entorno de ejecución permite disminuir los efectos de los fallos de manera eficiente, disminuyendo los costes de recuperación, identificando datos redundantes, o focalizando esfuerzos de protección en los datos críticos.

Contents

Chapter 1

Introduction

Throughout recent decades, computing has become exponentially more powerful, efficient, and present in our daily lives. This is due to the many ways in which the design of microprocessors has followed exponential laws, such as Moore's law. In 1965, Gordon Moore observed that the transistor density in an integrated circuit doubled every year, and he projected that it would continue to do so for the next decade [Moore 1965]. Amazingly, the transistor count per microprocessor has continued following an exponential growth until today, as illustrated in Figure 1.

The reduction in transistor sizes that allowed this exponential growth unfortunately comes with undesirable side effects. One of those issues are spontaneous non-reproducible errors appearing in circuits, which when tested later still operate correctly. These errors can thus not be linked to defects in the hardware. As early as 1975, unexpected triggering of circuits in communication satellites are

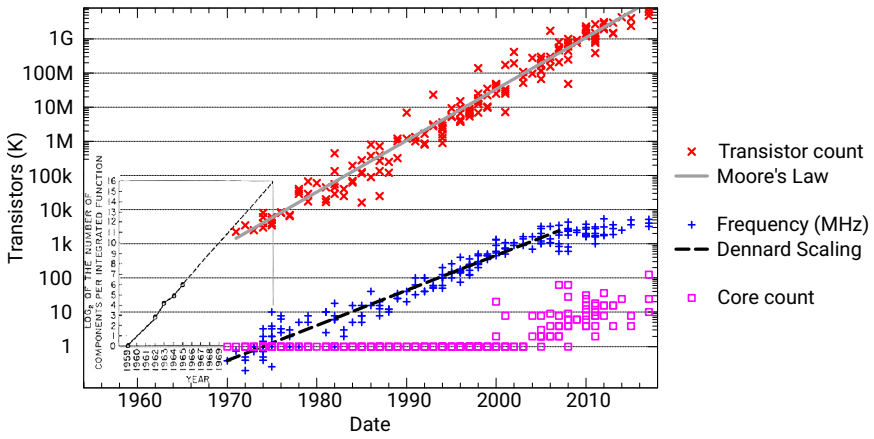


Figure 1.1: Evolution of microprocessors

The number of transistors grows exponentially, as does the frequency until around 2005. The initial extrapolation from Moore [1965] is reproduced in the bottom left corner.

attributed to interferences from cosmic rays [Binder *et al.* 1975]. In 1978, Intel detect and report errors in dynamic memories due to alpha particles [May and Woods 1978], and Ziegler and Landford predict the impact of cosmic rays on a 64KB Dynamic Random Access Memory (DRAM) in 1979 [Ziegler and Lanford 1979]. They estimated a fault rate of 7 errors per 10^{12} hours of DRAM device operation at sea level, and around 1.4 errors per 10^9 hours, at 10 km altitudes. Expressed in the FIT (Failures In Time) unit, which represents 1 error per 10^9 hours, the rates are respectively 0.007 FIT and 1.4 FIT per device.

Such faults, called soft faults or transient faults, are due to particle strikes that drain electrons in their wake and thus create a charge that interferes with the normal operation of a circuit. Soft faults leave the circuit intact, and can not be prevented. They are different from permanent faults or intermittent faults, also called hard faults, which are reproducible as they are due to hardware defects. While DRAM constructors typically do not publish the fault rates that affect their technologies, a 2015 study reveals soft error rates for 2 supercomputers of 75.24 FIT and 36.57 FIT per DRAM device [Sridharan *et al.* 2015]. In other words, these supercomputers experience a soft error every 5h23min and 15h49min respectively.

The strong increase in fault rates due to particle strikes is due to the miniaturisation as well as the reduction of operating voltages. The main parameter governing the appearance of a fault is the critical charge Q_{crit}, which is the minimum charge necessary to cause a circuit to malfunction. This critical charge decreases with the voltage and the size of components, and will thus continue to decrease as feature sizes decrease and power limitations constrain the design of new microprocessors. Furthermore, techniques to reduce leakage power such as increasing threshold voltage [Degalahal *et al.* 2004], or decreasing supply voltage to be close to threshold voltage [Kaul *et al.* 2012], will further deteriorate the fault rates. Current reliability techniques are unable to cope with expected fault rates, as they impose an upper limit on scalability called the Reliability Wall [Yang *et al.* 2012], which bars the way to building exaflop-capable supercomputers.

While all transistors can be subject to soft faults, memory cells are among the most vulnerable hardware components [Mukherjee *et al.* 2003]. They are typically protected by Error Correcting Codes (ECC) [Reed 1954] implemented at the hardware level. The widespread Single-Error Correct Double-Error Detect (SECDED)

ECC can for example detect and correct all single bit flips. It can also detect, but not correct, when two bits are flipped in the same codeword.

Moore's Law has remained true for a long time, partly as a visionary prediction and partly as a self-fulfilling prophecy. However, other similar empirical laws have broken down. Dennard's Law, which described the gains in power and switching frequency obtainable from reducing transistor sizes, lasted for 30 years [Bohr 2007]. The end of this law, commonly called Power Wall, is mainly due to increased current leakage, and has resulted in the stagnation of microprocessor operating frequencies. Converting more transistors into more performance after the Power Wall requires executing work in parallel, rather than executing the same work faster. In consequence, the industry has turned to multi-core processors since around 2005, as can be seen on Figure 1, and further to accelerators such as Graphical Processing Units (GPUs). Similarly, the slower scaling of memory as compared to microprocessors has caused an increase in latency of memory accesses. This increased gap is called the Memory Wall [Wulf and McKee 1995], and has been the cause for increasing cache sizes and complex memory hierarchies – including non-uniform hierarchies, where some cores have faster access to some part of the memory than others.

These evolutions have caused a paradigm shift in programming, from sequential programs being accelerated by the hardware evolutions, to parallel programs needing to spread their workload across the different Processing Elements (PEs, cores and accelerators). The earlier parallel programming models, such as pthreads and MPI [Gropp *et al.* 1994], were presented as libraries. They require all of this work repartition to be performed manually, as their constructs (threads and ranks, respectively) map directly to hardware resources such as cores. Thus, programming a machine becomes increasingly more complex with the number of cores per processor, the number of different core types, and the complexity of the memory hierarchy. The ability to build machines with a peak performance of 10^{18} floating point operations per second (1 exaflop/s), commonly called exascale, is planned to be reached by 2021 using extremely distributed machines [ETP4HPC 2017]. Therefore, programming models have emerged centred around more hardware-agnostic constructs, such as tasks, which are units of work that can then be scheduled on processing units. Such programming models often take the form of language ex-

tensions or entirely new languages [Chamberlain *et al.* 2007; Blumofe *et al.* 1995; Duran *et al.* 2011].

A number of complex problems are created by the scheduling of work across PEs, such as ensuring that the distribution of work is balanced, scheduling tasks in a way that minimises data movement, etc. Furthermore, a program whose parallel execution is tailored for a given architecture will need optimising again for a different architecture, thereby creating a new problem of performance portability. To take these burdens away from the programmer, the hardware-software abstraction stack is extended by these programming models with a new intermediate layer, called the runtime system [Casas *et al.* 2015]. The programmer then only needs to express how the work may be split up in tasks, and the runtime system performs the scheduling of work transparently, according to predefined policies, heuristics or strategies. Runtime systems have been used for many dynamic optimisations, such as accelerating critical tasks, prefetching or partitioning caches, and managing scratchpad memories [Castillo *et al.* 2016; Papaefstathiou *et al.* 2013; Manivannan *et al.* 2016; Alvarez *et al.* 2015]. Indeed, the position of a runtime system, having information both about the software and hardware, provides an incredible opportunity to perform optimizations on modern systems.

1.1 Thesis Objectives and Contributions

The main goal of this thesis is to use the opportunity presented by runtime systems to mitigate the reliability issues raised by increasing fault rates. In particular, the aim is to have a multi-disciplinary approach, developing strategies ranging from software-based fault tolerance to hardware proposals, all enhanced by the intervention of the runtime system, to diminish the overall failure rates due to soft errors in main memory.

1.1.1 Algorithmic-Based Exact Forward Recovery

The first contribution of this thesis is an algorithmic-based strategy to recover from Detected and Uncorrected Errors (DUE), presented in Chapter 4. Such errors are detected by ECC, but can not be corrected transparently. The responsibility of reacting against DUE is handed to the software stack. Some straightforward

approaches, such as crashing the affected process, are ineffective except against very low fault rates.

The proposed strategy consists of identifying algorithmic redundancies in the form of relations between data. The discarded or corrupted data can then be restored by recomputing or inverting the appropriate relations. This has the advantage over the most generic technique, which consists in rolling back the program state to a previously taken checkpoint, that it preserves the progress done by the program. Alternately, application-specific techniques exist, such as restarting a program with its latest values as initial guess. Our exact recovery however allows to maintain the algorithmic properties of the program, contrary to restart recoveries. This forward and exact recovery is thus doubly advantageous, and can be applied to an entire class of algorithms, the Krylov-subspace methods.

The error correction in itself, which is the recomputation of the data lost due to the DUE, can further be masked as well. The contribution proposes to overlap the error correction with the solver's normal workload. This is performed by encapsulating the error correction code inside a task, and leveraging the asynchrony of task-based programming models. The runtime system can then schedule the recovery work during load imbalance of the normal solver, without programmer intervention, which completely masks overheads for most error rates.

1.1.2 Quantifying the Risk of Error in Memory

The second contribution of this thesis tackles the need to quantify the risk of error due to faults in memory and is detailed in Chapter 5. The particularity of memory is that error characterisations can only be done at the scale of the program, as the impact of an error only depends on how data is accessed.

The Architectural Vulnerability Factor (AVF) is a common way of characterising the impact of circuits on a program's outcome. The literature shows many metrics and proxies for the error risk due to a fault in memory, including the AVF with a scope limited to memory. The proposal extends this metric, the *Memory Vulnerability Factor* (MVF), to take into account false errors. These are reported errors which would have no impact on the program if they were ignored. The resulting metric is called *False Error Aware MVF (FEA)*. In practice, MVF represents the probability of accessing data, while FEA represents the probability of consuming data.

FEA, MVF, and other related metrics are measured using a cycle-accurate simulator, and compared against the effects of injecting faults in a program's data, which are measured in native parallel runs. The only metrics that provide upper bounds on the error risk are MVF and FEA, and the latter provides a tighter bound. Based on this finding, the contribution proposes a hardware-enabled OS-level mechanism to ignore false errors in memory. The reduction in fault rate due to false errors is quantified using the previously validated metrics.

1.1.3 Dynamically Adaptable ECC

The third contribution of this thesis, presented in Chapter 6, is a methodology to dynamically identify vulnerable parts of memory during a program's execution. The runtime methodology proposed in this contribution relies on the FEA metric previously established, and can be used for offline analysis of a program's vulnerable data (e.g. to select regions for algorithmic protection) or for other optimizations, such as data placement.

The proposed methodology relies on modern and readily available Performance Monitoring Unit (PMU) capabilities. It consists in sampling load and store instructions to identify streaming memory access patterns, from which the average vulnerability of data can be extrapolated. The proposed methodology is implemented in a real POWER8-based system, demonstrating its feasibility and low overhead.

At the circuit level, error mitigation techniques are applied to the most critical elements, which are identified using architectural vulnerability analysis. Likewise, the data in memory where faults have the strongest impact should be protected in priority to improve reliability at low cost. Therefore, the contribution includes a hardware proposal for a dynamically adaptable ECC, designed to increase ECC protection for the most vulnerable memory regions. The dynamically guided ECC strength using the vulnerability identification allows to find a number of interesting trade-offs between decreasing error probability and increasing the cost of ECC protection.

1.2 Thesis Outline

The contents of this thesis are organised as follows:

Chapter 2 reviews the background in DRAM, and shared memory programming models' runtime systems, exposing their main characteristics, issues, and opportunities. It then reviews the state-of-the-art related to each of the multidisciplinary resilience techniques presented in this thesis.

Chapter 3 introduces the error injection methodology used to perform the experiments described in this thesis, as well as the simulation infrastructure and benchmarks used to evaluate the proposals presented in this work.

Chapter 4 presents the algorithmic recovery technique that allows to recover from DUE, and the novel runtime-enabled overlapping of undisturbed computation from the algorithm with the recovery computations.

Chapter 5 presents the new FEA metric to better characterise the probability of an error in memory causing a program to fail, as well as an OS-level technique to reduce DUE rates from memory leveraging key insight developed from the FEA metric.

Chapter 6 proposes a methodology to estimate FEA online using sampling of memory instructions. The chapter also introduces WITSEC, a hardware proposal for a dynamically adaptable two-level ECC scheme, used to protect in priority data that is identified as requiring more protection, by the implementation of the runtime methodology on a real system.

Finally, chapter 7 concludes this dissertation by stating its main contributions, and providing an overview of possibilities for future research derived from the work presented in this thesis.

Chapter 2
Background

In this chapter, we present the background and give an overview of the state-of-the-art of resilience techniques related to DRAM errors.

2.1 Evolution and Prospectives for DRAM

DRAM is the memory type used in computers as the *main memory*, for its low cost and high capacity. This memory is not persistent, meaning that the values stored in memory are not preserved if the power is switched off. The data also needs to be periodically refreshed while the power remains on.

In DRAM, data in stored in cells that can each hold 1 bit of information, and that consist of 1 transistor and 1 capacitor, as illustrated in Figure 2.1. To read or write to a cell, the corresponding word line (2) needs to be activated, switching on the transistor (3). To read the contents of the cell, the voltage alteration of the bit line (1) due to connecting the capacitor (4) is sensed and amplified. To store a bit in the cell, the bit line (1) is driven to the desired voltage until the capacitor (4) is charged. The capacitors of each DRAM cell leak charge over time, requiring the contents to be periodically refreshed.

DRAM technology sizes shrink exponentially, at a slower pace than transistors [Sunami 2008]. As DRAM technology becomes denser, the cell capacitance and voltage also decrease, to diminish power consumption and allow faster operation. The first DRAM technologies such as Fast-Page Mode DRAM used a 5.0V supply voltage, which has decreased until 1.1V standardised in the Low-Power Double Data Rate 4 (LPDDR4) DRAM [JEDEC 2014], and sub-1.0V designs have already been proposed [Cho *et al.* 2013]. Similarly, the capacitance has reduced from 100fF to about 30fF in the time that the memory cell size has reduced by

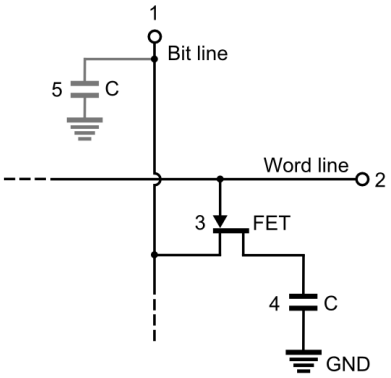


Figure 2.1: DRAM cell

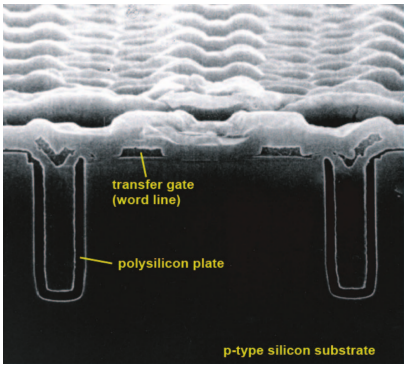


Figure 2.2: DRAM trench capacitor cross-section. Reproduced from Sunami [2008].

10,000×. This much slower decrease is due to the strong impact of cell capacitance on the efficiency of sensing voltage variations on the bit line [Hong 2010].

The architecture of DRAM is organised in memory modules, which are circuit boards holding DRAM chips. Each DRAM chip outputs a number of bits at a time, which is referred to as the width of the chip. Common chip widths are 4, 8 or 16 bits nowadays.

2.1.1 Causes for Faults in DRAM

The soft error probability in a DRAM cell can be decomposed in the probability for a cell to be hit by a particle, and the probability that this event causes the stored bit to change value. The slow decrease in capacitance and voltage has caused the critical charge to decrease by about an order of magnitude since the beginnings of DRAM, as the stored charge is *capacitance* × *voltage*. At the same time, the probability of being hit by an energetic particle has evolved with the cell's surface. This probability decreases as device density increases – however the die size has remained constant [Sunami 2008]. That is, the decrease in memory cell size has been compensated by the increasing number of memory cells, causing the overall Soft Error Rate (SER) of DRAM to remain roughly constant [Baumann 2005]. This trend seems to come to an end. To maintain storage capacitance while decreasing sizes, the DRAM industry has had to create deep capacitors, such as the trench capacitor whose cross-section is displayed in Figure 2.2. Aspect ratios of capacitors for 20nm DRAM sizes are of the order of 70:1, with an oxide layer of 5Å, which already represent significant challenges [Hong 2010]. Thus the capacitance

of DRAM cells is at risk of decreasing again, and the overall SER will increase accordingly.

Another class of errors, due to manufacturing defects and process variability, are known as *hard faults*. These faults are either permanent or intermittent, and affect a number of cells that do not function within the specification. The main mechanism causing individual DRAM cells to fail is through leaking charge. Each cell has a leakage current that defines its retention time. If this retention time is smaller than the period at which DRAM data is refreshed, the data in the cell is lost. A similar problem is variation in the write recovery time [Kang *et al.* 2014], which causes cells to be slower to charge than the timing requirement. Leakage current is known to be proportional to temperature, and also varies over time for some cells. This is known as Variable Retention Time (VRT) [Restle *et al.* 1992], and cells affected by it may transition back and forth between functioning within specification and losing data.

The intuitive response for hard errors is to use sparing techniques. Supplementary rows or columns can be built in DRAM chips, using laser fuses to disable and replace faulty rows and columns [Schuster 1978; Nair *et al.* 2013]. At low error rates, entire rows or columns can be decommissioned to mask a single faulty cell. Redundant Bit Steering (RBS) is a similar technique that uses a spare DRAM chip, to replace a chip with an identified chip or pin error at runtime [HPEC 2013]. Techniques have been proposed to support higher error rates and be more cost-effective. SECRET stores Error Correcting Pointers (ECPs) in DRAM, which store the address of a failing bit and a replacement for this bit [Lin *et al.* 2012]. ArchShield uses a reserved area in DRAM to replicate 8-byte words that contain faulty DRAM cells, and to keep a map of these words [Nair *et al.* 2013]. CiDRA places a small SRAM cache in each DRAM device to replace addresses associated with faulty cells [Son *et al.* 2015]. All of these approaches have the downside of requiring a profiling of which cells in DRAM are malfunctioning.

Another approach, which tackles the retention time problem, is to adjust the refresh rate per DRAM row to the retention time of the leakiest cell in each row. However, this is problematic as profiling needs to be done continuously, since DRAM cells start displaying VRT behaviour throughout the lifetime of the DRAM device. This has been identified by Qureshi *et al.* [2015], who simultaneously proposed AVATAR, a refresh scheme that adjusts dynamically to the appearance of

VRT behaviour of DRAM cells. This is done by upgrading all rows that present an error to the refresh group with highest refresh rate, as soon as the error is encountered. A yearly retention testing can then downgrade DRAM rows to lower refresh rates, if they were misidentified as having low retention times (i.e. suffered from a soft error). Ultimately, unexpected faults can not be prevented, thus redundancy needs to be added to detect and correct such faults.

2.1.2 Error Correcting Codes for DRAM Memories

Reliability and availability are major concerns for servers and supercomputers. As such, machines built for those purposes have long used ECCs to ensure that no errors occur in memory, especially without being detected. Such codes are now available on machines for personal use, all the way to mobile devices [Micron 2017].

Until the early 1990s, parity was the standard for memory protection [IBM 1999]. To tolerate higher fault-rates, SECDED ECC was put in place. SECDED protection can be achieved for a 64 bit word with a systematic code (i.e. where the initial data is unmodified) with a truncated Hamming code and a parity bit, for a cost of 8 redundancy bits [Hsiao 1970]. That is, to store data in memory using this SECDED code, 72 bits of storage are required for every 64 bits of data. This redundancy has been stored in separate DRAM chips on the same DIMM, creating the industry standard 72-bit wide DIMM, composed of 18 4-bit wide chips or 9 8-bit wide chips. Similarly, parity was implemented with a separate 1-bit wide DRAM chip.

With increasing error rates, and chances of having larger granularity errors, ChipKill-level protection was adopted [Dell 1997]. ChipKill-level protection means that the failure of an entire DRAM chip can be tolerated. This can be achieved by interleaving SECDED codes, however this creates an impractical configuration. With 4-bit wide chips, 72 DRAM chips need to be accessed simultaneously, forming a channel of 256 data bits and 32 check bits. This approach is detrimental to system performance and energy efficiency. Another approach is to use a code with symbols that correspond to the output of a single chip, instead of a single bit. Then, a Single-Symbol Correcting (SSC) code provides ChipKill-level protection. This has been proposed with 8-bit symbols on 4-bit wide chips. Double-ChipKill protection, also called Double Device Data Correction (DDDC), is provided with a combination of SSC and RBS [Kim *et al.* 2015].

Facing the increasing fault rates, constructors have now proposed on-die ECC, where both the ECC encoder and decoder and the redundancy bits are located inside each DRAM chip [Kang *et al.* 2014; Oh *et al.* 2014], instead of using a separate chip. This approach increases redundancy by an additional 6.25% or 12.5%, depending on the scheme, and is proposed as a complement to the previously discussed DRAM-level ECCs. Academic proposals have quickly shown however that exposing the full amount of supplementary redundancy is much more efficient than applying independent layers of ECC [Nair *et al.* 2016; Gong *et al.* 2018].

In practice, ECCs correct errors transparently and, on modern x86 and AMD64 architectures, report them in dedicated registers [Intel 2017; AMD 2018]. When an error is detected but can not be corrected, a machine check exception is triggered that the Operating System (OS) has to handle. Most OSs simply kill the program that accesses the memory location containing the DUE. Improvements include supporting errors in pages mapped from a file that are unmodified, and tracking the number of Corrected Errors (CEs), which indicate a higher risk of a future DUE. In both cases, the contents of the offending memory page can be placed at a different physical location in memory (respectively reloaded or copied), and the program can keep running with its virtual address space updated to point to the new physical location. Memory pages can be blacklisted from further use [Kleen 2010]. If an error can not be detected by an ECC, or if it is miscorrected, then Silent Data Corruption (SDC) occurs.

Additionally to ECC, replication techniques can be used at a wider scope to catch SDC or recover from DUE. Replication can consist in storing several copies of the same data at different locations in memory, as fail-over if errors are detected or to detect errors. Intel for example offers *Memory Mirrorring*, which consists in replicating data across two different DRAM DIMMs [Intel 2011].

2.1.3 DRAM Error Rate Studies

Theoretical fault rates and probabilities are known through simulations and laboratory experiments such as soft error measurement under accelerated particle beams [JEDEC 2007]. Real-world fault rates however need to be gathered over large periods of time and statistically significant amounts of hardware. All research groups use slightly different methodologies and terminologies. One key distinction is between the event which is called a fault, and its symptom which is called an

error. A fault is a number of bits changing value unexpectedly, either a single time or repeatedly at the same location. An error is an access to a memory location that returns a value different from the value that was previously stored at that location.

Many studies rely on error logs of real machines. A single fault, soft or hard, can cause a large number of errors, which are then repeatedly reported in error logs. In fact, Sridharan and Liberty [2012] report that the Jaguar supercomputer encounters a fault (either new or re-activating a hard fault) every 5.64 hours on average, while the system reports an error every 10 seconds on average. In a more recent study, Levy *et al.* [2018] report for the Cielo supercomputer that ageing does not seem to impact the DDR3 DRAM fault rate of 3 vendors, over the 5 years lifetime of the machine. They also show that correctable faults are not good predictors for uncorrectable errors, meaning that uncorrectable errors will appear as random events. Finally, Levy *et al.* also note that at least 27% of machine down events (i.e. nodes becoming unavailable for use) are due to memory DUE.

Bautista-Gomez *et al.* [2016] perform a study on a machine without ECC, storing known data and fetching it later to compare it against the expected value. The advantage of this approach is that it is unbiased by multiple reporting of error in logs. They report an error every 10 minutes on average for the machine, and show that while single bit errors remain the majority, multiple bit corruptions are common occurrences, and correlate with other simultaneous single errors in the same machine, as well as with time of day. Bautista-Gomez *et al.* do not distinguish between soft and hard errors, however their analysis of spatial correlation only identifies 4 nodes as having either an overall DIMM failure (numerous repeated errors spread out) or a *weak bit*, i.e. a single bit that intermittently displays the same error – which is typical of a retention error.

2.1.4 Summary

Fault rates are on the rise due to many combined factors, both for soft (thus random) and hard (thus spatially correlated) faults. These rising rates and the current solutions are problematic to successfully reach the next era of computing, exascale [Cappello *et al.* 2009; Cappello *et al.* 2014]. While only ECCs can cope with random faults in DRAM, many techniques are proposed to handle hard faults. These proposals either provision sufficient ECC redundancy to tackle all types of

errors and uniformly ensure low DUE and SDC rates, or mitigate hard faults by targeted techniques, such as faster refresh, sparing, or retirement. In this latter case, which is much less wasteful of resources, hard faults may well only appear once as a random error, before being profiled and mitigated, either by faster refresh or a smart sparing approach. These mitigations techniques make hard faults behave like soft errors, in the sense that they only appear once. Thus, the work in this thesis will focus on random independent errors to either handle when ECC fails, thus encounters DUEs, or to apply ECC in a cost-effective way.

2.2 Programming Models and Runtime Systems

2.2.1 A Brief History of Parallel Programming

Parallel programming begins years before the appearance of the first chip multiprocessor. The first mention of getting "several results at the same time" dates back to 1842 [Menabrea 1842], and supercomputers in the late 80s started taking on the shape of many interconnected processors, each with its own RAM, as described by Hillis [1981]. To program these machines, data needs to be communicated from one node to the next. A standardised interface was proposed to perform these tasks and to abstract away the network's specifics [MPI Forum 1993]. MPI is still the de-facto standard to this day for distributed memory computing in High Performance Computing (HPC).

With the advent of chip multiprocessors in the 2000s, a new model of parallelism became predominant: shared memory. In this model, all the cores of the multiprocessor are organised under a common memory hierarchy, and can share virtual address spaces. This allows the usage of the same memory in multiple cores without requiring explicit messages to send the data from one core to the other. In this paradigm, the programmer does not have to keep track of the location of data and ensure its coherence any more.

The earliest and the simplest shared memory programming models are the thread libraries offered by the OS [Nichols *et al.* 1996; Beveridge and Wiener 1997]. Some programming languages offer thread libraries as well [Oaks and Wong 2009; Stroustrup 2013], to provide a portable interface wrapping the low-level parallelism tools provided by the OS. The constructs in these libraries are software

threads, and synchronisation primitives. Threads map directly to the hardware in the same way that an MPI process maps to a node in distributed memory programming models. Here, a software thread is a stream of instructions, and maps to a core in the context of a multiprocessor. Through the shared memory model and synchronization, threads can communicate. However the repartition of work across the threads needs to be performed explicitly by the programmer. This paradigm for parallel programming is also called the *fork-join* model, as there is an explicit moment when parallelism starts (a fork) and when it ends (a join).

The current de-facto standard for parallel programming on shared memory multiprocessors is a fork-join model, OpenMP [OpenMP Board 2005]. OpenMP is a standard specification that allows to declare parallel constructs in C, C++ and Fortran using simple and portable compiler directives. These directives are supported by the vast majority of modern compilers and OSs for shared memory multiprocessors. In an OpenMP program, a single thread is used in sequential regions of the program, and at the start of the program. Then, parts of the programs that are annotated with directives by the programmer are executed across several threads. The main parallelism constructs are:

- parallel regions, which are executed in every thread,
- parallel sections, which are distributed across threads to be executed,
- parallel loops, which declare that iterations of a loop can be executed independently of one another, and can thus be distributed across threads similarly to sections.

Further key directives exist to perform synchronisation, such as atomic values and critical sections, which ensure that only a single thread at a time respectively modifies a value or executes a portion of the code. This allows to ensure consistency in the shared memory, by avoiding data races.

2.2.2 Task-based Programming Models

Departing from constructs that represent the hardware resources, the notion of task provides a higher level abstraction for parallel programming. A task is a unit of work that can be executed by any core, much like an OpenMP parallel section, or an iteration in an OpenMP parallel loop. These tasks can then be executed

immediately or deferred to be executed later. This subdivision of the work coupled with mapping tasks to computing resources dynamically allows to solve problems such as load balancing and expressing parallelism in recursive algorithms with very little programming effort. Much like the out-of-order execution of instructions in a processor, the decomposition of a parallel program in tasks allows for much more flexibility in the way that the computations are performed. A program with tasks is no longer using a fork-join model, where some parts of the program are parallel and some sequential. In the task-based paradigm, cores are resources much like floating point units, and work is scheduled onto such resources. That is, the execution is always parallel, and the throughput is limited by the amount of work that is available.

Examples of early task-based programming models are Cilk [Blumofe *et al.* 1995] for shared memory programming models, and Charm++ for distributed memory programming model [Kale and Krishnan 1993]. Evolutions include OpenMP 3.0 [OpenMP Board 2008], which includes basic tasking constructs. As opposed to these approaches that extend existing languages with libraries or compiler directives, some programming models are entirely new languages [Chamberlain *et al.* 2007]. There exist many other programming models targeting the exascale era, with a variety of subtleties (global vs. local view of data and control flows, different levels of asynchrony, etc.) [Gropp and Snir 2013]. All these novel programming models, as well as recent evolutions of historic programming models, have in common that they require a runtime system, which is a software library that supports the programming model. These libraries perform all the work that the programmer does not need to explicitly express any more, and in particular the scheduling of tasks onto hardware resources according to load balancing constrains.

However, this asynchronous approach is limited due to the lack of ordering between tasks. Therefore, only tasks that perform the same types of operations or independent operations can be executed in parallel.

2.2.3 Data-flow Task-based Programming Models

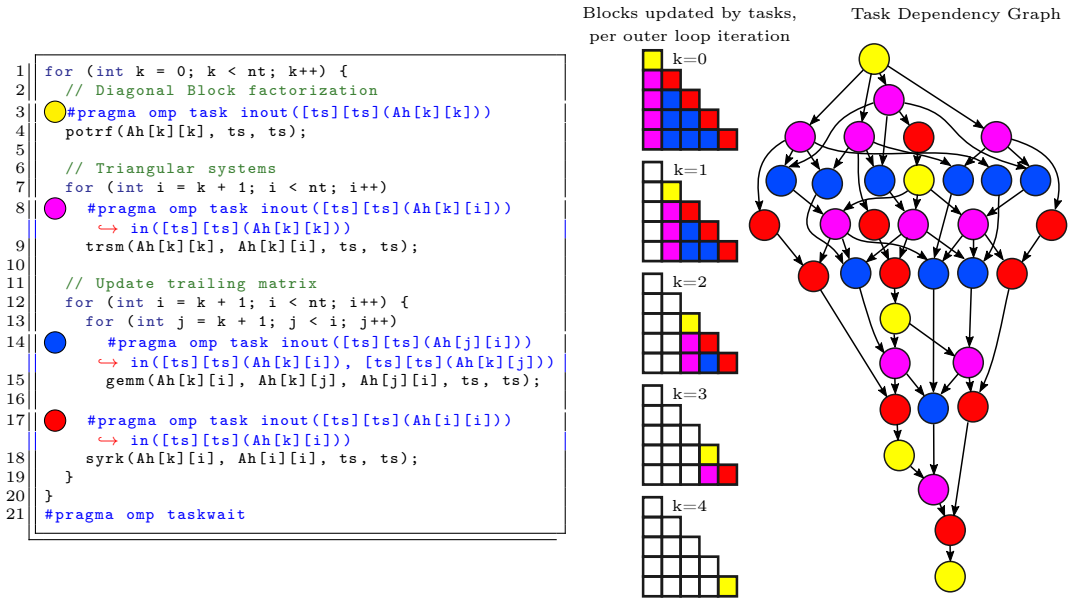
For any program to be fully expressed as tasks, it is necessary to be able to enforce ordering between the computations performed in different tasks. Such scheduling constraints are called *dependencies*. Tasks are created ahead of time, and placed in a Task Dependency Graph (TDG). This direct acyclic graph has tasks as nodes,

and dependencies between tasks as edges. A task without any predecessors in the graph is ready to be executed, and can be scheduled onto one of the hardware resources that the runtime system has at its disposal. When a task is finished, it is removed from the graph, and all its successors now have one less predecessor. The program execution then simply continues until the TDG is empty.

Dependencies can be declared with asynchronous constructs. Modern programming languages such as C++11 [ISO 2011] and ECMAScript 6 [Ecma 2015] include constructs which represent the result of a computation that is not yet performed. Accessing this result, or explicitly waiting for the computation, then interrupts the current task until the result is available. While these techniques allow for some parallelism to be exploited, they are more geared towards asynchrony and interruptibility. A more explicit way of expressing dependencies are *data-flow dependencies*. These are at the heart of OmpSs [Duran *et al.* 2011], an OpenMP forerunner that expands its tasking constructs to explore their capabilities, and have been included in the OpenMP 4.0 standard [OpenMP Board 2013]. Rather than waiting for the result of a task one at a time, this approach consists of declaring at task creation which data a task requires as input, and which data it produces as output. The runtime system then infers the dependencies between tasks from the data that each task accesses.

An example of the syntax of such data-flow is presented in the listing in Figure 2.3, where every kernel function has been annotated to become a task with a `#pragma` directive. The program then instantiates a task that consists in executing a kernel, instead of directly calling the functions. The execution of the parent task then stops when it reaches the `taskwait` directive, similarly to waiting for an asynchronous result, and resumes when all the tasks have finished executing. We can see data dependencies declared with `in(...)` which mean that the tasks will take this data as input, and as `inout(...)`, meaning that the task will use this data both as input and output. From these annotations, the runtime can define the dependencies. For example, all the `trsm` kernels depend on the previous `potrf` kernel due to the `Ah[k][k]` dependency. On the task dependency graph on the right of Figure 2.3, each task instance is a separate circle, with the colour indicating the task type. The dependencies between tasks are represented by arrows, and the dependencies from `potrf` to `trsm` kernels are thus the arrows from yellow to

Figure 2.3: Cholesky source code with OmpSs data-flow tasks annotated.
An example TDG is also displayed for a 5-by-5 blocked matrix, with colour-coded tasks.




```
1  for (int k = 0; k < nt; k++) {
2    // Diagonal Block factorization
3    #pragma omp task inout([ts][ts](Ah[k][k]))
4    potrf(Ah[k][k], ts, ts);
5
6    // Triangular systems
7    for (int i = k + 1; i < nt; i++)
8      #pragma omp task inout([ts][ts](Ah[k][i]))
         ↪ in([ts][ts](Ah[k][k]))
9      trsm(Ah[k][k], Ah[k][i], ts, ts);
10
11   // Update trailing matrix
12   for (int i = k + 1; i < nt; i++) {
13     for (int j = k + 1; j < i; j++)
14       #pragma omp task inout([ts][ts](Ah[j][i]))
           ↪ in([ts][ts](Ah[k][i]), [ts][ts](Ah[k][j]))
15       gemm(Ah[k][i], Ah[k][j], Ah[j][i], ts, ts);
16
17   #pragma omp task inout([ts][ts](Ah[i][i]))
       ↪ in([ts][ts](Ah[k][i]))
18     syrk(Ah[k][i], Ah[i][i], ts, ts);
19   }
20 }
21 #pragma omp taskwait
```

magenta tasks. Dependencies can also be declared as `out(...)` or one of many other dependency types [BSC PM 2018].

2.2.4 Parallel Runtime Systems

The runtime is a software layer that orchestrates the parallel execution of a program. It is thus in charge of managing threads and hardware resources, the pool of tasks, tracking the dependencies, and choosing the scheduling of the tasks. Together with the programming model, this provides the end user with a high-level, clean, and abstract interface for any underlying hardware. Indeed, these roles taken on by the runtime system allow the programming model to be extended for more complicated architectures than symmetric multiprocessors without burdening the programmer. For example, some architectures are heterogeneous, with some cores being faster or more powerful than others. Accelerators provide even more heterogeneous hardware, with for example GPUs or FPGAs, that often have their own dedicated memory. The simplicity of a task-based programming model makes using accelerators very easy. If a program provides implementations of a single task for various different hardware resources, the runtime can then schedule the task either onto a core or an accelerator, and handle the data movement that

this requires transparently [Bueno *et al.* 2011; Planas *et al.* 2013; Bueno *et al.* 2013].

Runtime systems bring many features for tasks, similarly to what modern processors perform for instructions, such as out-of-order execution. Concepts such as pipelining and speculative execution then seem natural to explore for tasks [Valero *et al.* 2014; Brumar *et al.* 2017]. Runtimes can optimise their scheduling work using additional information such as task criticality, for example exploiting heterogeneous hardware to diminish the execution time of a program, by scheduling first tasks that are the most critical [Chronaki *et al.* 2015; Castillo *et al.* 2016; Chronaki *et al.* 2017]. The research in the domain of runtime optimisations is rich and promising, as runtime systems have been used for other purposes that are beyond their initial goal.

For example, runtime systems are used to manage non-computational resources such as I/O and checkpointing [Jia *et al.* 2017], and overlap their use with computations, similarly to what is presented for recoveries in the algorithmic contribution of this thesis, in Section 4 [Jaulmes *et al.* 2015]. The task dependency information of runtime systems can be used as well, and many proposals use it to manage the cache hierarchy: partitioning last-level caches [Pan and Pai 2015], evicting dead blocks [Manivannan *et al.* 2016], selecting insertion policies dynamically [Dimić *et al.* 2017], guiding prefetchers [Papaefstathiou *et al.* 2013], and optimising data movement for producer-consumer patterns [Manivannan *et al.* 2013; Manivannan and Stenström 2014]. The same information can be used to manage scratchpad memories [Bellens *et al.* 2006; Alvarez *et al.* 2015] or heterogeneous memory systems [Liu *et al.* 2017; Alvarez *et al.* 2018]. Caheny *et al.* propose to use the runtime scheduling and data allocation information to reduce cache-coherence traffic [2016; 2018a], and to deactivate coherence for data that is identified by the runtime as not requiring it [2018b]. Conversely, the runtime can also move computation to where the data resides, instead of adjusting the memory hierarchy, e.g. by partitioning task dependency graphs [Sánchez Barrera *et al.* 2018]. The core functions of runtime systems are themselves being accelerated in hardware, such as managing task dependencies and scheduling tasks [Kumar *et al.* 2007; Etsion *et al.* 2010; Tan *et al.* 2016; Tan *et al.* 2017; Castillo *et al.* 2018] and optimising core-to-core queue-based communications [Sanchez *et al.* 2010; Wang *et al.* 2016].

All of these optimisations exploit runtime information to perform optimisations bridging several abstraction levels, and while some can be performed with existing hardware, some require extending the hardware capabilities to be runtime-aware [Casas *et al.* 2015]. Thus, the goal of this thesis is to leverage the capabilities presented by the emergence of runtime systems, to tackle the problems due to DRAM error rates presented in Section 2.1.

2.3 Application-Level Fault Tolerance

In HPC, computations often require many nodes over long periods of time. In that context, the risk of a computation being interrupted due to a failure is high, and restarting the computation has a high cost. For example, a program could run into a software error, some node could malfunction, or the program could be killed after encountering a DUE. A study on the Blue Waters supercomputer [Di Martino *et al.* 2014] shows that a failure happened on average every 4.2 hours, with system-wide outages every 160 hours on average.

2.3.1 Checkpointing and Rolling Back

To guard against such events, techniques such as checkpointing are commonly used. Checkpointing consists in writing the state of a program to persistent storage in such a way that it can be loaded later, and that the computation can be continued from this state. Then, in the case of a failure, only the computations performed since the last checkpoint are lost. Recovering the state from the latest checkpoint is known as a rollback, and is categorised as a backwards recovery as it consists in going back to a previous state. The state-of-the-art libraries providing checkpointing capabilities are SCR [Moody *et al.* 2010] and FTI [Bautista-Gomez *et al.* 2011]. Given the cost (in time) to write and load checkpoints and the Mean Time Between Failures (MTBF), the optimal checkpointing rate can be computed to minimise a job's expected execution time [Bougeret *et al.* 2011]. Recent work has focused on multi-level checkpointing to differently handle errors with different granularity, and models for these checkpointing strategies [Di *et al.* 2017].

In general, it is interesting to write as little data as possible to disk, as this is a slow process. Optimisations include incremental checkpointing, which consists in writing only the data that changed since the last checkpoint [Gioiosa *et al.* 2005].

Another approach is to modify the application to be able to write and restart with only the minimal amount of data that allows to restart the computation. Such an algorithm-specific technique has been presented by Chen [2013] for the family of solvers that we will investigate in Chapter 4, the Krylov subspace methods.

2.3.2 Checkpointless Algebraic Recoveries

Algorithms can be extended for resilience as a means to recover from a failure. Instead of relying on a checkpoint to restore the state, the program identifies that it has encountered an error and lost some data, and recovers from it. The usual model for error detection in HPC is the fail-stop model: in a multiple node execution, one of the processes is interrupted. This can be due to the node crashing or the process being killed, and results in a localised loss of data. This naturally leads to the proposal for a Fault-Tolerant Message Passing Interface (FT-MPI), that hands control back due to a node having stopped, in order to recover data from this part of the program [Fagg *et al.* 2001].

Chen [2011] proposes a partly algebraic recovery method for Krylov solvers with matrix A and right-hand side b, solving for $Ax = b$. Upon a fail-stop error, most vectors are recovered from MPI messages, which are logged for this purpose as a sort of implicit checkpoint, except the iterate x which is seldom passed in MPI messages. The exact arithmetic relation between the residual r and x, thus $r = b - Ax$, is used to recover x without checkpointing it, by factorising the diagonal block of the matrix that was stored on the node encountering the failure.

Langou *et al.* [2007] introduced the Lossy Approach with the block-Jacobi step interpolation. This restart method, designed for an MPI fail-stop error model, is extensively discussed in Section 4.4.3. Agullo *et al.* [2013; 2016b] extended this work by introducing least-squares methods for the interpolation, and further studying strategies to minimise computations and communications in recoveries in the case of multiple simultaneous errors. These trade-offs can naturally be applied to the interpolations we use, if needed. Agullo *et al.* [2016a] have further extended restart recoveries relying on lost data interpolation to eigensolvers, and to hybrid solvers such as MaPHyS [2015].

2.3.3 Detecting Errors

Algorithms can be extended for resilience beyond recovering from identified errors, instead detecting SDC by identifying and checking program invariants. Chen [2013] proposed to use two application specific invariant relations to catch undetected errors in CG and its derived method BiCGStab, which are the (bi-)orthogonality of search directions, and the relation between gradient and iterate. This consists in using inherent redundancy, by identifying a relation ($||b - Ax - r|| = 0$ with our notations) that remains true in exact arithmetic during the program execution, as leveraged in the algorithmic contribution of this thesis. Agullo *et al.* [2018] evaluate this relation as an SDC detector, while precisely bounding this relation's error due to accumulated round-offs in finite arithmetic. Agullo *et al.* compare this "reduction gap" detection to other techniques, including one using a bounding relation between an algorithmic parameter, α, and the eigenvalues of the matrix. Liang *et al.* [2017] similarly detect soft faults online in the fast Fourier transform.

Instead of finding existing invariants in programs, to recover or identify errors, invariants can be added into algorithms to detect errors. Such invariants can be checksum values that are transformed in the same way than the data of the program. For example, Huang and Abraham [1984] propose adding a row that is the sum of all the other rows in a matrix. The result of a matrix multiplication then also contains a last value which is the sum of all the other values, and can be used to verify the validity of the multiplication. Similar approaches exist for other operations such as QR and LU factorizations [Davies and Chen 2013; Heroux *et al.* 2005]. This approach adds little memory space overhead at the price of computational overhead. However, checks on finite precision numbers (as opposed to bitwise checks) are sensible to round-off errors.

Subasi *et al.* [2016] use machine learning, specifically support vector machine supervised learning, to detect SDCs by monitoring state variables that are declared by the application programmer. This technique is only partly application-specific, as it is a runtime library performing the SDC detection. However the programmer does need to specify which variables to use for SDC detection, and more importantly, the SDC detector has to be trained for the detection of errors with each selected program. That is, the programmer does not specify invariants that exist within the program, however the machine learning approach identifies application-

specific redundancy between the state variables of a program. This is what allows it to detect whether a snapshot is potentially suffering from SDC. Other techniques such as AID [Di and Cappello 2016] take advantage of the range of possible outcomes of SDC to only detect errors that have a significant impact, for a large number of HPC applications. Furthermore, generic SDC detector strategies can be combined with techniques such as replication to maximise error detections while allowing more variability in the data, i.e. removing false positives [Berrocal *et al.* 2017].

2.3.4 Application Sensitivity to Faults

Bridges *et al.* [2012a] identify application-level resilience that is not due to inherent redundancy, but stems from the nature of the preconditioning operation. Indeed, a preconditioner is often a heuristic, and in the restarted GMRES solver it only defines a new search direction in which to expand the Krylov subspace. Thus selecting a poor vector does not affect the correctness of the GMRES iterations, and only risks causing the solver to converge slower. Consequently, the authors propose the Fault Tolerant GMRES consisting of GMRES iterations, run safely, enclosing a preconditioner which may run unreliably and return inexact values. This unreliability may be due to storing the data in a less reliable region of memory, for example. The protection proposed for GMRES in Section 4.3.1.2 of this thesis protects all vectors of GMRES, and this work by Bridges *et al.* suggests that we might not need to recover preconditioned vectors exactly, instead replacing lost data with an approximation and protecting exactly only the vectors basis.

Another technique to identify which parts of an application are more resilient is to inject errors, and measure the program's outcome to determine their impact. Elliott *et al.* [2014] evaluate the sensitivity of the GMRES solvers while Bronevetsky and Supinski [2008] use 6 different linear algebra solvers. Both studies demonstrate the necessity of detecting errors with inexpensive application-level checks, in the absence of other techniques. Casas *et al.* [2012] identify pointers as the most sensitive data structures in the algebraic multigrid solver and propose triplicating them as a protection measure.

A number of studies present frameworks for error injections. Li *et al.* [2012] present BIFIT, an error injection tool to target specific locations and times of the program's data and execution time, later used in the evaluation of a statically

adjustable ECC proposal [Li *et al.* 2013], using algorithm-based fault tolerance to detect and correct errors that are not covered by ECC. The tool relies on Pin [Luk *et al.* 2005], an instrumentation framework made available by Intel. Luo *et al.* [2014] inject faults by running the application inside of a debugger. Finally, Gupta *et al.* [2018] use FaultSim, a Monte-Carlo model-based simulator, which allows to measure the MVF exactly – however ignoring the application-specific impact of memory errors. In particular, in Gupta *et al.*'s work, loading an uncorrectable or undetected error is equated with program failure.

2.4 Evaluating Vulnerability of Data in Memory

The idea of identifying more vulnerable parts of a program's data is not limited to algorithmic approaches. Indeed, such a determination would allow to select for example whether to place data in a more or less reliable memory location [Chen *et al.* 2005; Gupta *et al.* 2018]. However, generalising this for any program is not straightforward and requires gathering information through other means than the theoretical analysis of an algorithm. To identify reliable and vulnerable memory segments, e.g. for a programmer writing compile-time annotations, this reliability needs to be quantified. To do this, metrics of vulnerability have been developed.

2.4.1 Metrics for Memory Vulnerability

Program-agnostic approaches have attempted to follow the example of how reliability decisions were guided at the chip level. Mukherjee *et al.* [2003] have devised a metric to model the error rate of any component of a computing system, the AVF. This metric quantifies whether a given bit matters for reliability, using the probability that a fault at this bit may cause an error in the final outcome of a program. Averaging this value per component for various workloads indicates which components require error mitigating techniques. For example, a speculative component such as a branch predictor will never affect reliability, while an instruction decode unit has a major impact on any program.

It is worth noting that the AVF is dependent on the scope in which it is defined [Mukherjee 2008, Chapter 3.2]. That is, if we limit the analysis of the AVF to memory, then the AVF of a bit is the probability that a fault at this bit causes the memory subsystem to return an erroneous value, i.e. a value different

from the one stored at this location. This does not take into account what the program does with the erroneous value, and is thus an upper bound on the full-system AVF. Thus, limiting the AVF analysis to memory classifies a bit that is later loaded as vulnerable (or required for Architecturally Correct Execution (ACE), in Mukherjee *et al.*'s terminology). This loaded bit is vulnerable whatever the outcome of the program after loading this bit with a wrong value: the value may be ignored, have a negligible impact, cause an incorrect result or even crash the program. To underline this key distinction, we call Memory Vulnerability Factor (MVF) the AVF scoped to memory only, thus the probability that a bit causes an incorrect value to be returned from memory. In the remainder of this work, we call AVF the probability that a bit causes an incorrect outcome of the program, thus considering the full system running a program as the default scope for the AVF.

Yu *et al.* [2014] define the *Data Vulnerability Factor (DVF)* per data structure d, defined as the multiplication of the structure's size S_d, the program execution time T, the number of accesses to this structure in memory N_{ha}, and the overall fault rate FIT: $DVF_d = FIT \cdot T \cdot S_d \cdot N_{ha}$. They then use mathematical models to compute the DVF based on memory access patterns. This does obviously not take into account the relative timing of the memory accesses, nor whether faults are consumed or overwritten. Luo *et al.* [2014] use the *safe ratio*, which is the fraction of time that data resides in memory before being overwritten. This is the same as the MVF, except that it chooses to quantify the opposite of vulnerability. The MVF and the safe ratio sr are related by $MVF + sr = 1$. Gupta *et al.* [2018] use two metrics, initially measuring the MVF, defined as "the average duration that data is stored in memory before being loaded". They then use a proxy metric, which is the ratio of stores (ST) to loads (LD), thus ST/LD, for the purpose of their runtime page-placement algorithm.

2.5 Dynamically Adaptable ECC Protection

In order to adapt the memory protection dynamically, it is necessary to measure our vulnerability metric dynamically. In order to do so, PMUs can be used to identify memory access patterns. This is explained in detail in Section 6.4. PMUs have been used for similar analyses, either at runtime or offline.

2.5.1 Sampling to Identify Memory Access Patterns

PMUs allow to gather information about program executions, through the sampling of addresses accessed by memory instructions. Servat *et al.* [2015] and Giménez *et al.* [2014] correlate these accesses with memory regions identified at runtime, to detect memory access patterns and visualise memory bottlenecks. The low-overhead instrumentation techniques allow the authors to gain insights and tune applications offline, but are not used for runtime optimisations.

A dynamic use of the POWER5's PMU capabilities has been presented by Tam *et al.* [2009] to log memory instructions. From this log of addressees gathered dynamically, the applications' miss rate curves can be computed online. This information is then used to partition cache resources among different applications running simultaneously.

2.5.2 Variable Strength ECC schemes

Managing to measure vulnerability at runtime is enabled through PMUs, however adapting ECC strength dynamically is not possible with commodity hardware. At best, two hardware memory resources with different reliability features can be used, and the vulnerability-based decision is then to decide the placement of data on one or the other of these resources. Luo *et al.* [2014] choose to recommend whether to place data in a DIMM without ECC, with parity, or with SECDED ECC, while Gupta *et al.* [2018] choose to place data either in High Bandwidth Memory (HBM) or slower but more reliable DRAM, in a hybrid memory architecture setup.

Academic proposals exist for ECC schemes for DRAM with variable strength, however none use dynamic online guidance. Virtualized ECC [Yoon and Erez 2010] stores a first tier ECC code, mainly to detect errors. Yoon and Erez then store a second tier ECC code, used in the case of uncorrected errors in the first level, in addressable memory. Similarly to Virtualized ECC, Odd-ECC [Malek *et al.* 2017] uses two tiers of error correction. However, Odd-ECC relies on a pre-defined arrangement in memory at a 256KB granularity. A 256KB region selected for higher protection has less memory usable by the program, as part of it is reserved for ECC. This organisation is less flexible than a page-table level mapping as used in Virtualized ECC. Runtime-adaptable ECC schemes have been devised for other

hardware than DRAM, such as caches [Alameldeen *et al.* 2011; Paul *et al.* 2011], and NAND memories [Yuan *et al.* 2015].

Finally, a cache in the memory controller has been proposed to store meta-data of Error Correction Pointers [Lin *et al.* 2012].

Chapter 3
Methodology

In this chapter, the experimental methodology used in this thesis is explained. The first section covers the error injection mechanisms common to all contributions of this thesis, while the second section describes the simulation infrastructure used to measure vulnerability precisely. Finally, we present the benchmarks that are used throughout this work.

3.1 Injecting Errors

All error injections are performed in native parallel runs on the same real system. Errors are injected using a separate thread, at a random time during the Region Of Interest (ROI) of the benchmark. The prologue, which consists of generating the program's input, and the epilogue, which consists of verifying the program's output, are not considered for error injections. The injector thread simply sleeps for the selected amount of time, then injects the error using one of the two possible mechanisms: DUE or bit flip injection.

The type of error that is injected depends on the error model that is used. At the application-level, we assume ECC is in place and simulate what happens whenever it fails, that is when a DUE occurs. At lower levels, when studying the sensitivity of applications to errors in their data and evaluating the impact of ECC, we inject bit flips in a program's data and classify the outcome of its run.

3.1.1 DUE Injection

Currently, OSs retire full memory pages upon a DUE [Kleen 2010]. It is worth noting that this error model is generalisable to more types of DUE, as long as the

extent of the potentially corrupted data can be identified. This is always the case for memory DUE. Since the Linux Kernel 2.6.32, hardware poisoning of a memory page can be triggered through various means, such as the `madvise` system call.

To simulate errors we use the `mprotect` system call available in Linux kernels to change the authorizations of the targeted memory page. This is more practical than triggering a real hardware retiring of a memory page, and behaves identically: the program receives a signal at the time it accesses the memory page. We recover in the same way as we would from a real error: in a signal handler, we request a new memory page at the same virtual address through means of the `mmap` system command. All the recoveries operate exactly in the same way as they would if a real DUE took place. For the solver, there is no difference between real hardware DUE and our error injection mechanism.

3.1.2 Injecting Bit Flips

To inject bit flips, we select a 64 bit word in the targeted application-level data, as this is the granularity at which the most common ECCs, SECDED and ChipKill, operate. Two types of faults can be injected, either a number of bit flips, where random bits in the selected 64 bit word are flipped, or a DUE. For bit flips, we pick at random the selected number of bits in the word, and flip them in the targeted word at the moment of the error injection. A DUE consists in poisoning the data, and always causes an incorrect program outcome if and only if the data is consumed. In practice, it consists of inserting a `NaN` in floating-point data or a very high value for integer data. This is useful both to measure the exact failure rate due to DUE, and to get an upper bound on failure rates due to bit flips and ECC miscorrections.

We verify analytically that the results of consuming a DUE is indeed always an incorrect outcome. In most cases, integer data is used to index other arrays, in which case injecting a high value causes the program to crash from a segmentation fault. Otherwise, we ensure that incorrect integer data causes an incorrect result, such as a non-existing category for a benchmark performing classification, etc. Similarly, we ensure that verifications on floating point numbers fail for unexpected `NaN` values, keeping in mind that comparisons against `NaN` values always return false. For example, verifying that a value `v` is smaller than a threshold ϵ needs to be rewritten from `if(v`$<\epsilon$`)` to `if(!(v`$\geq\epsilon$`))`.

Each experiment consists of a single fault injection. The program runs until it finishes abnormally or until completion, in which case the validity of the solution is checked. In the case of benchmarks that have a built-in check of the solution, we always verify the solution using non-modified input data. Otherwise, we compare against the output of a reference run in which no faults are injected. The experiments' outcomes are classified using the taxonomy described in Section 6.2.1 and Figure 6.2: *ok* and *slow* (successes), and *hang*, *wrong*, and *crash* (failures). An experiment is classified as *hang* after failing to complete within 10 times the reference execution time. Finally, an experiment is classified as *slow* if the program runs for at least 20% more time than the reference run, or performs at least 2 more iterations than the reference for iterative benchmarks.

3.1.3 Assessing the Impact of ECC

In error-injecting experiments with a dynamically adaptive ECC scheme, we check the selected ECC strength at the moment of the fault injection. If the ECC corrects more bit flips than are injected, we count the outcome as *ok*. Otherwise, we use the outcome of the experiment, in which we injected the error.

We combine the results of experiments with different numbers of bit flips, using a relative fault rate of 0.01 between successive numbers of flips [Mitra *et al.* 2014]. That is, we suppose double bit flips are $100\times$ less frequent than single bit flips, and triple bit flips $10,000\times$ less frequent. This allows us to measure *the overall probability of failure in the event of a fault*, for every possible combination of ECC levels, without needing to postulate a fault probability.

3.2 Simulation Infrastructure

To be able to gather precise information about when data reaches or is fetched from memory, we use a cycle-accurate simulator. We extend TaskSim [Rico *et al.* 2011; Rico *et al.* 2012], a task-trace based multicore simulator, to compute the exact memory vulnerability ratings of data. Its infrastructure relies on task-based execution models to generate detailed traces for each task, including the basic blocks that are executed and memory addresses that are accessed. TaskSim's multicore architecture simulator then simulates parallel runs in detail by fetching and simulating all instructions, using a simple core model and a full memory hierarchy.

Table 3.1: TaskSim cache parameters

cache	shared	assoc.	size	latency	MSHRs
L1D	private	8-way	32kB	4 cycles	32
L2	private	8-way	256kB	12 cycles	32
L3	shared	16-way	20MB	28 cycles	128

The simulator also relies on a real runtime system, to schedule the tasks across the simulated hardware. The memory is simulated using Ramulator [Kim *et al.* 2016]. TaskSim has also been extended to allow multi-level simulation encompassing multi-node executions of MPI programs [Grass *et al.* 2016].

To compute the various vulnerability metrics, we capture all loads and stores and the time at which they reach main memory. We then update at each access the necessary counters per memory location: time before stores, time before loads whose contents are directly overwritten, time before remaining loads, and number of loads and stores. From this data, we can compute all the vulnerability metrics that we consider. We only update these counters during the ROI and compute all metrics at a 64 bit granularity, which is the granularity used for SECDED and a subset of the granularity commonly used in ChipKill-level ECC. Finally, we also report for each memory page of a benchmark the average fraction of time it resides in cache.

We trace applications on an Intel x86_64 Xeon E5-2670 and simulate a multi-core architecture whose configuration mirrors the Xeon E5-2670's characteristics. It consists of 8 cores running at a frequency of 2.6GHz, each with a reorder buffer of 192 entries, and one thread per core. The memory hierarchy's parameters are summed up in Table 3.1. All cache levels have 64B lines, write-back and write-allocate policies, are non inclusive, and track outstanding misses in Miss Status Handling Registers (MSHRs). Ramulator simulates 4GB of DDR4 DRAM memory, organised in one rank of 4Gb x8 chips clocked at 2400MHz.

3.3 Benchmarks

For the purpose of the work presented in this thesis, we use 14 parallel benchmarks, which are listed in Table 3.2. All benchmarks are written using the OmpSs programming model, with tasks that use real data-flow dependencies. The source codes of all the benchmarks are available online [Jaulmes 2019].

Table 3.2: Benchmarks used for evaluation

The last column indicates whether the benchmark's output verification is built-in (✓) or done comparing with the output of a reference run (✗).

Name	Benchmark description	Category	Input size	Built-in output verification
Blackscholes	Option pricing [Chasapis *et al.* 2015]	Partial Differential Equation	400M options	✓
Cholesky	Cholesky factorization	Dense linear algebra	8192×8192 matrix	✓
CG	Conjugate Gradient	Sparse linear algebra	16Mi×16Mi matrix	✓
DGEMM	Matrix multiplication	Dense linear algebra	5120×5120 matrix	✓
FFT	Fast Fourier Transform	Spectral method	Stockham, 2Mi points	✗
Gauss-Seidel	Heat diffusion, Gauss-Seidel solver	Structured grid	4500×4500 grid	✗
Jacobi	Heat diffusion, Jacobi solver	Structured grid	4500×4500 grid	✗
KNN	K-nearest neighbours	Machine learning	500K points training, 5k testing sets	✗
K Means	K-means clustering	Machine learning	100K points, 30 dimensions, 8 clusters	✗
N-body	Astrophysical simulation	N-body method	16Ki particles, 100 iterations	✓
PRK2 stencil	Parallel Research Kernels stencil [Wijngaart and Mattson 2014]	Stencil operation	16Ki×16Ki grid, 130 iterations	✓
Red-black	Heat diffusion, red-black solver	Structured grid	4500×4500 grid	✗
SMI	Symmetric matrix inverse	Dense linear algebra	4608×4608 matrix	✓
Stream	Stream Triad [McCalpin 1995]	Memory bandwidth benchmark	192MB	✓

3.3.1 The Conjugate Gradient Benchmark

We present in detail the Conjugate Gradient (CG) benchmark Hestenes and Stiefel [1952] and Saad [2003], as it is the main code used for the evaluation of the algorithmic techniques, and will serve as a motivating example for the other chapters. Shewchuk [1994] also provides an in-depth and accessible introduction to CG. The algorithm's pseudo-code is presented in Listing 3.1.

Listing 3.1: Conjugate Gradient (CG) pseudo code

```
1  ϵ_old ⇐ +∞ ,  p′ ⇐ 0
2  for t in 0..t_max :
3     r ⇐ b − Ax  if  t ≡ 0 (mod 50)  else  r − αq
4     ϵ ⇐ ||r||^2
5     if  ϵ < tol : break
6     β ⇐ ϵ/ϵ_old
7     p ⇐ βp′ + r
8     q ⇐ Ap
9     ||p||^2_A ⇐ ⟨q, p⟩
10    α ⇐ ϵ/||p||^2_A
11    x ⇐ x + αp
12    ϵ_old ⇐ ϵ
13    swap (p ,p′ )
```

CG solves $Ax = b$ for x, where A is a sparse Symmetric Positive Definite (SPD) matrix. b, x, r, p, p', and q are vectors, and ϵ, ϵ_{old}, α and β are scalars. The matrix A is stored in memory in compressed sparse row format. Thus, we refer to it as 3 separate data structures: the rows Ar, columns Ac, and values Av.

CG is implemented using OmpSs, a task-based data-flow programming model, with tasks each generating one block of each of the vectors or sums (in the case of reductions). We use two copies of p and swap their pointers (line 13) to allow delaying tasks that depend on one copy, such as $x \Leftarrow x + \alpha p$. This allows the runtime to ignore the false dependency due to overwriting p, and to overlap x's update with operations that incur load imbalance such as $\langle p, q \rangle$, thus improving scalability. The periodic recomputation of r allows to limit the accumulation of round-off errors in the $r = b - Ax$ relation, and the convergence threshold tol is set to $||b||^2\ 10^{-20}$ to make the stopping criterion $||b - Ax||/||b|| < 10^{-10}$.

Two alternate implementations are also used, one with a block-Jacobi preconditioner (see Section 4.3.2), and an implementation that can scale up to over a thousand of cores. For this latter implementation, we use a hybrid implementation where the node-level parallelism is leveraged using MPI while the intra-node (shared-memory) parallelism remains exploited by the asynchronous task-based data-flow programming model OmpSs. The CG solver only requires the following additions:

- Global MPI reductions after the local reductions,
- Exchanging locally updated parts of p with nodes depending on it.

This new exchange task takes place after updating p and before computing q (between lines 7 and 8 on Listing 3.1), while the MPI reductions occur after the local reductions (norm of r line 4, and A-norm of p line 9). In practice, the MPI calls for those reductions are placed inside the tasks that compute β and α, respectively.

For the single node implementation, we measure solving 9 matrices selected from the University of Florida sparse matrix collection [Davis and Hu 2011]. They are well-conditioned matrices for CG selected among the biggest of each family of SPD matrices. For the MPI + OmpSs implementation, we solve Poisson's equation in 3D using a 27 point stencil discretization, which is also used in the HPCG benchmark [Heroux *et al.* 2013], with a system size of 512^3 unknowns for the multiple-node setup. This matrix is also used when CG is used as part of a larger collection of benchmarks. We then use a single node and a smaller number number of unknowns (256^3), as shown in Table 3.2.

3.3.2 Remaining Benchmarks

For the purpose of the work presented in this thesis, we use parallel benchmarks from various origins, selected because they represent a varied set of application types, and because they can all have the validity of their output verified. All benchmarks are written for a shared-memory environment using the OmpSs programming model, except N-body which can run on several nodes, as using an MPI + OmpSs hybrid programming model. The verification of the outputs is preserved from the original code for all the benchmarks that had built-in verifications, and we list here how each verification works.

Cholesky computes the factorisation of an SPD matrix A of dimension n into an upper triangular matrix L, such that $L^T L = A$. The built-in verification takes this matrix L and checks that $\frac{||L^T L - A||_\infty}{n \ ||A||_\infty} < 60 \ \epsilon_{\text{BLAS}}$.

SMI computes the inverse of an SPD matrix by first factorising the matrix A using Cholesky, then inverting L and finally multiplying the obtained triangular matrix with its transposed, to compute $A^{-1} = (L^{-1})^T (L^{-1})$. The built-in verification checks with the same threshold as Cholesky that $\frac{||Id - AA^{-1}||_1}{n \ ||A||_1 \ ||A^{-1}||_1} < 60 \ \epsilon_{\text{BLAS}}$.

Both **Blackscholes** and **DGEMM** verify that every individual value in the result deviates by less than 10^{-4} from the expected result. In DGEMM, the input matrices are set statically and the expected output matrix is thus known analytically, while in Blackscholes the expected option prices are generated together with the input data for each option.

N-body checks the final positions of each simulated astrophysical body against those of a reference run, and checks that the average relative error per component of the position is at most 8×10^{-6}.

For **K-means**, we compare the resulting metric that the algorithm is minimizing, the within-cluster sum of squares, and check that it is not higher. For **KNN**, we verify that all the points to be classified are attributed to the same class.

Finally, the **Stream** and **PRK2 Stencil** benchmarks can both compute the expected results of their computations, and internally verify that the average relative error is less than 10^{-8}. All the remaining benchmarks similarly output floating point values, and we check that the average relative error compared to a reference run is less than 10^{-8}.

Chapter 4
Algorithmic Recoveries

4.1 Introduction

As memory errors become more frequent, so is the probability of encountering a DUE. When such an error happens, the software stack needs to handle the error. Some straightforward approaches like cancelling the affected process or relocating a faulty memory page to another physical location may be effective against low fault rates, but they are insufficient against the predicted rates that processors will suffer in the future. Also, very aggressive resilience strategies like process triplication are completely impractical unless we face very high fault rates [Ferreira *et al.* 2011]. Therefore, intermediate solutions that recompute an approximation of the lost data [Langou *et al.* 2007] or that save the process state in a checkpoint with a certain frequency have been extensively used [Moody *et al.* 2010; Sorin *et al.* 2002; Chen 2013]. However, most of these solutions involve backward recoveries, discarding useful computations, and thus incur significant slowdowns.

The application itself may be able to handle the error and terminate cleanly [Bland *et al.* 2013] or perform some sort of recovery procedure relying on Algorithmic-Based Fault Tolerance (ABFT), which has been extensively applied to MPI programs [Fagg *et al.* 2001; Langou *et al.* 2007; Casas *et al.* 2012], as well as shared memory programming models [Wong *et al.* 2010; Subasi *et al.* 2015]. While the majority of ABFT techniques are geared towards error detection, such as discussed in Section 2.3.3, algorithmic approaches have also demonstrated to provide more efficient recoveries than backward recovery techniques like checkpointing-rollback. However, most ABFT recovery techniques so far suffer from two main drawbacks: being very application dependent, and still incurring significant overheads. In this chapter we aim to reduce the impact of these two issues which have avoided the

wide-spread usage of algorithmic resilience. The proposed ABFT methods to deal with DUE are based on very simple algebraic relations that do not require any kind of deep understanding of the algorithm and can be almost always derived for iterative methods. When a DUE is signalled, we always discard the whole memory page where affected data resides, as OSs do for bus and memory ECC errors, and recompute the discarded data using the algebraic relations that we derived. The cost of this recovery is proportional to the page size, and we evaluate the proposed ABFT methods for all possible sizes from 4K bytes up to 2M bytes. The overheads related to recovering data are reduced by overlapping them with algorithmic computations. Since the responsibility of such overlapping is left to the runtime system, we do not significantly increase the programming burden.

This chapter proposes an integrated resilience approach, where the error detection is performed by hardware mechanisms that report DUE to the OS, which identifies lost data at a memory page level and triggers a signal caught by the application. We use the OmpSs task-based data-flow programming model [Duran *et al.* 2011], in which serial code is split into several pieces, called tasks, that are dynamically scheduled according to data dependencies explicitly expressed by the programmer. We combine the OmpSs annotations with MPI to scale our implementation up to over thousand cores. We demonstrate the feasibility of our approach by applying it to relevant iterative methods of the Krylov subspace family: CG, Bi-Conjugate Gradient Stabilised (BiCGStab), and Generalised Minimal RESidual (GMRES) [Barrett *et al.* 1994], and implementing it for CG. The main contributions of this chapter are:

- A general resilience solution for DUE based on straightforward algorithmic recoveries. With the lowest error injection rate considered, corresponding to one expected error per baseline execution time, the overhead of this technique is 5.40%, whereas that of the checkpointing-rollback technique is close to 55%.

- An asynchronous and programmer transparent variant of our recovery implementation that reduces the overhead down to 2.24% under the lowest error rate, and that offers a trade-off between low overhead and convergence rate for higher error rates.

- A mathematical proof showing that the Lossy Approach by Langou *et al.* [2007], an interpolation-restart strategy, is the best of all the restart techniques in the literature.
- An exhaustive and comprehensive evaluation, using real world matrices, of our method against a more sophisticated algorithm-specific restart method, derived from the Lossy Approach, and checkpointing-rollback mechanisms. We consider different parallel scenarios from 8 up to 1024 cores and we show that our methods always improve the performance of the above mentioned state-of-the-art methods.
- This chapter further studies the effect of page sizes, from 4KB up to 2MB, on the overheads of the techniques. Our algorithmic methods outperform the state-of-the-art on average up to 512KB page sizes. For bigger sizes, our methods still perform better for the bigger matrices of the test set, and perform similarly to the Lossy Restart method on small matrices where a full vector can fit inside a single memory page.

The rest of this chapter is organised as follows: Section 4.2 explains how to recover from hardware detected memory errors by using inherent redundancy, while Section 4.3 shows how to use this redundancy to make Krylov subspace methods resilient, as well as implementation details of this methodology for CG. Section 4.4 introduces the methods with which we compare our recoveries, and the next sections provide the experimental evaluation in Section 4.5, and the evolution of these results with page sizes in Section 4.6. Section 4.7 provides our concluding remarks.

4.2 Exact Interpolation Recovery

4.2.1 Error Detection and Reporting

Due to the advent of faults, many processors have specific registers dedicated to signalling errors to the OS layer. On modern x86 and AMD64 architectures for example, a memory controller discovering data that is incoherent with the ECC, while accessing or periodically scrubbing it, reports it in a specific register [Intel 2017; AMD 2018]. For memory pages, when the corrected errors exceed a

threshold, the OS transparently relocates the page at another physical location. When a DUE is reported, the OS kills the affected process. This feature is known as memory page retirement on Solaris, and soft or hard page offlining in Linux kernels [Kleen 2010; Tang *et al.* 2006].

In practice, application termination after a page failure is done by a `SIGBUS` signal. This signal can be caught and also specifies the failing memory addresses. By catching the signal and requesting a new hardware page at the same virtual address, the program can continue executing without further errors. Thus, to be resilient against memory DUE, an HPC application simply has to be able to replace lost data.

We classify all data as either static if it does not change during execution time (matrix, preconditioners, right-hand side), or dynamic if it may be modified. Static data is assumed to be saved to a reliable backing store, from which it is reloaded when errors are detected, similarly to other work using memory-page level fault models [Bridges *et al.* 2012b]. While there typically is not enough information available in the solver to recover static data, there may likely be where the matrix was generated or read. Alternately, such data could be protected by software ECC at low cost, by exploiting the fact that this second ECC tier only needs to correct, and not detect errors [Yoon and Erez 2010]. Thus, only dynamic data needs to be made recoverable.

4.2.2 Extracting Redundancies of Linear Solvers

Linear iterative solvers perform operations like matrix-vector multiplications $q = Ap$, linear combinations $u = \alpha v + \beta w$, and combinations of the above, e.g. the very common residual $r = b - Ax$, where A is a matrix while q, p, u, w, r, b and x are vectors and α and β are scalars. In many cases the left and right hand side of these operations coexist during the whole execution of the solver.

In some cases, we know that such a relation between vectors holds true (minus round-off errors) without having to recompute them. For example, if we define $x' = x + \alpha p$ and $r' = r - \alpha q$ with the above notations, then $r' = b - Ax'$. Finally, similar relations can hold true by construction, without ever having been computed. By analysing an iterative solver, we find redundancies expressed in terms of explicit or implicit relations between data.

Table 4.1: Block recoveries for operations $q = Ap$, $v = \alpha v + \beta w$ and $r = b - Ax$

Block relation, recover left side	Inverted relation, recover right side
$q_i = \sum_{j=0}^{n-1} A_{ij} p_j$	$A_{ii} p_i = q_i - \sum_{j \neq i} A_{ij} p_j$
$u_i = \alpha v_i + \beta w_i$	$w_i = (u_i - \alpha v_i)/\beta$
$r_i = b_i - \sum_{j=0}^{n-1} A_{ij} x_j$	$A_{ii} x_i = b_i - r_i - \sum_{j \neq i} A_{ij} x_j$

Trivially, if any vector r, q, u is lost or partially corrupted, it can be recovered by recomputing the relation involving that vector. Given the inverses of A, α, β and other potential operands of a relation, it would also be possible to recover a lost or corrupted p, v, w, t or x. However, solving $Ap = q$ for p or $Ax = b - r$ for x is as expensive as running the whole iterative method. The matrix A can not be inverted either in the general case, due to numerical and computational considerations. However, recoveries based on such redundancy relations are applicable if only a small portion of the data structures involved in a relation is lost. This is the case with our error model since modern hardware is able to report errors at memory page level. In order to operate at such fine grain level, the redundancies must be expressed in terms of relations between small blocks of data.

4.2.3 Block Decomposition

The relations exposed previously, decomposed in n blocks, are listed in Table 4.1. We use the normal block relation to recover the left-hand side of a relation, and the inverse of this relation for the right-hand side. In the event this inverse relation relies on a diagonal block of the matrix, we need to use a solver to recompute the lost data. If we know that a diagonal block is non-singular, e.g. when A is SPD, we solve the inverse block relations with a direct solver. Otherwise we solve this relation in the sense of least squares for the full columns of the matrix corresponding to the lost memory page as input, similarly to what Agullo et al. do for restart methods [Agullo *et al.* 2013].

The formula for x_i's recovery, shown at the bottom of the right hand side of Table 4.1 has been used by Chen [Chen 2011] to recover the iterate, in complement of implicit checkpointing methods. Our approach requires no checkpoints however, as we protect all vectors with interpolation methods – as detailed in Section 4.3.1 for CG, BiCGStab, GMRES, and in Section 4.3.2 for their preconditioned variants. Exploiting these relations for recovery is a novel idea, since all previous work on

making Krylov-subspace solvers fault-tolerant relies on a fail-stop failure model in a distributed memory environment. The required information to use redundancy of linear relations is then not available since corresponding parts of different vectors are lost simultaneously.

Furthermore, the granularity of the blocks of lost data in our recoveries is very different from the one in the context of process failure, which allows different and faster recoveries. Indeed, our block decomposition is dictated by the underlying layers (hardware detection, OS, runtime) that do the DUE reporting. This means the block size that we use as granularity for recovery is a memory page. For a typical off-the-shelf machine, memory pages are 4K bytes, though bigger pages can be used in HPC settings – provided architectural support, such as the 2MB "huge pages" on x86. We evaluate our technique for any power of 2 sized page from 4KB to 2MB, thus for data losses from 512 to 262,144 double precision floating-point values.

Any DUE in our data protected by relations can thus be rectified by applying a small amount of computations, at worst factorising a diagonal block of a matrix if one is used by that relation. This is a forward recovery scheme, since we can continue executing the program with our interpolated replacement data and the data that is not affected by the error. When A's diagonal block is non-singular or a linear relation is used, we can even guarantee the exact same data as was lost for all relations (up to rounding errors), thus guarantee the same convergence rate as when the algorithm is not subject to faults. These recovery operations are usually small compared to the total computations, since matrix dimensions reach up to more than a million rows for real-life problems, as available in the University of Florida sparse matrix collection [Davis and Hu 2011].

4.2.4 Dealing with Multiple Errors

Our approach requires no assumptions on simultaneous errors. Indeed, our techniques can easily handle multiple errors (discovered simultaneously) in most situations. Errors can always be recovered if they affect different instances of blocked linear relations expressed in Table 4.1. However, if simultaneous errors impact a single relation we have two possible scenarios:

1 *Simultaneous errors in a single vector* are not a problem for our recovery strategy. This is trivial for vectors recovered from linear relations, and straightforward for submatrix relations [Langou *et al.* 2007]. For two failed blocks i and j, we can combine both block relations:

$$\begin{pmatrix} A_{ii} & A_{ij} \\ A_{ji} & A_{jj} \end{pmatrix} \begin{pmatrix} x_i \\ x_j \end{pmatrix} = \begin{pmatrix} b_i - r_i - \sum_{k \neq i,j} A_{ik} x_k \\ b_j - r_j - \sum_{k \neq i,j} A_{jk} x_k \end{pmatrix}$$

This relation is extendable to any number of blocks, with an increasing submatrix size to be factorised.

2 *Simultaneous errors on related data*, e.g. both q_i and p_i for a given i in a $q = Ap$ relation. Assuming no other relationship allows to recover these data, we may fall back to a restart method, e.g. the Lossy Restart which is adapted from the Lossy Approach [Langou *et al.* 2007] to fit our error model (see Section 4.4.3). The frequency of occurrence of simultaneous errors is discussed in Section 4.5.3.

In conclusion, our forward interpolation recovery relies on very simple redundancy relations that are easy to identify in any iterative method, and that can efficiently be used at memory page level. This recovery can deal with multiple errors, but that may imply a more expensive computation or, at worst, the usage of a restart method as fallback.

4.3 Applying Recoveries to Iterative Solvers

4.3.1 Making Redundancies Explicit

DUE are reported when a faulty operation is made or when trying to access data that is corrupted. Even for data corruption discovered by the OS while periodically scrubbing memory pages, no error is signalled until that data is accessed, in the hope the page will be freed.

So in order to make an iterative solver resilient with our technique, it is sufficient to find for each operand of each operation done by the solver a relation

- that either allows to recover the operand, and then compute the result of the operation,

Listing 4.1: CG pseudo code with redundancy relations

```
1  ϵ_old ⇐ +∞                                  r = b − Ax
2  r ⇐ b − Ax
3  for t in 0..t_max
4      ϵ ⇐ ||r||²                               r = b − Ax
5      if ϵ < tol : break
6      β ⇐ ϵ/ϵ_old
7      p ⇐ βp + r                               p = A⁻¹q      r = b − Ax
8      q ⇐ Ap                                   see 4.3.1.1
9      α ⇐ ϵ/ < q, p >                          q = Ap        p = A⁻¹q
10     x ⇐ x + αp                               p = A⁻¹q      x = A⁻¹(b − r)
11     r ⇐ r − αq                               q = Ap        r = b − Ax
12     ϵ_old ⇐ ϵ
```

- or that allows to compute the result without this operand – that is, finding an alternate formulation.

Note that the main difference between this work and previous application-level recoveries for the same iterative solvers is the error model: since we do not consider complete failure of a node, we do not incur the loss of *a part of every vector*, which would render the relations we use here inapplicable. Over the next Sections we explain in detail how three commonly used iterative methods, CG, BiCGStab, and GMRES, can be protected using redundancy relations.

4.3.1.1 Conjugate Gradient

The pseudo-code for CG is given in Listing 4.1 [Saad 2003], with the relations used for recovering each accessed data annotated on the right. Relations are written as whole-matrix relations for the sake of readability, but we use the memory page grained system described previously. Whenever possible, the relation that last produced data is used, which is not possible when data is updated in place. By construction, the algorithm conserves the relation $r = b - Ax$, and we can define an alternate way of computing q besides performing Ap from the update formula of $p \Leftarrow \beta p + r$, which is $q \Leftarrow \beta q + Ar$. When computing q, if a page of p is missing, q can be computed using this alternate formulation. When the whole matrix-vector multiplication is done, p can be recovered using $p = A^{-1}q$, in order to continue computations. However, it is impossible to use this relation when updating p. Let

us consider the blocked formulation:

$$p_i = A_{ii}^{-1}\big(q_i - \sum_{j \neq i} A_{ij} p_j\big)$$

At this stage in the update of p, all pages $p_0..p_{i-1}$ are at iteration $t+1$, and all pages $p_{i+1}..p_{n-1}$ are at iteration t.

We have two possibilities for recovery:

1 Postpone the recovery and compute q_i at iteration $t+1$ using $\beta q + Ar$, then factorise A_{ii} to get p.

2 Perform *double buffering*, thus have two copies of a vector, either p or q, and use them alternately from one iteration to the next to remove in place updates.

The first option, though possibly more elegant, would imply taking rather considerable distances from the original algorithm. It is also arguable that the $\beta q + Ar$ operation might need to be protected, since a matrix-vector multiplication is the most computationally intensive and longest operation in a CG iteration. The latter option is beneficial to the performance of the algorithm, regardless of resilience considerations, as explained in Section 3.3.1. We thus opted for double buffering p, and unrolled the loop to use two p vectors alternately, as illustrated in Listing 4.2. Code unchanged by this transformation is skipped. This solution adds redundancy to the method at the cost of some minimal memory overhead.

4.3.1.2 Generalised Minimal RESidual

The code for GMRES is available in Listing 4.3. Each iteration of GMRES consists of running the Arnoldi method - the part creating an orthogonal basis of vectors spanning $(r, Ar, ..., A^{(m-1)}r)$ and an associated upper-Hessenberg matrix H - followed by a QR decomposition of this matrix H through Givens rotations. We may then increment the iterate by the solution y of $min_y||r - Hy||$.

Protecting the biggest part of the data, which is the v^k vectors, is straightforward thanks to the Hessenberg matrix. At any time, we have at step t,

$$l > 0 \text{ and } l < t \Rightarrow v^l = \frac{1}{h_{l,l-1}} \left(A v^{l-1} - \sum_{k=0}^{l-1} h_{k,l-1} v^k \right)$$

Listing 4.2: CG with p double-buffered

```
1  for t in 0..t_max
2      ...
3      p_1 ⇐ βp_2 + r
4      q ⇐ Ap_1
5      α ⇐ ϵ/ < q, p_1 >
6      x ⇐ x + αp_1
7      ...
8      t++
9      p_2 ⇐ βp_1 + r
10     q ⇐ Ap_2
11     α ⇐ ϵ/ < q, p_2 >
12     x ⇐ x + αp_2
13     ...
```

Listing 4.3: GMRES pseudo code

```
1  for t in 0..t_max
2      r ⇐ b − Ax
3      v^0 ⇐ r/||r||_2
4      for l in 0..m − 1
5          w ⇐ Av^l
6          for k in 0..l
7              h_{k,l} ⇐< w, v^k >
8              w ⇐ w − h_{k,l} v^k
9          h_{l+1,l} ⇐ ||w||_2
10         v^{l+1} ⇐ w/h_{l+1,l}
11     solve H = QR
12     y ⇐ R^{−1} Q^T ||r||_2 e_1
13     x ⇐ x + Σ_{l=0}^{m−1} y_l v^l
14     check convergence
```

Thus the redundancy kept in the Hessenberg matrix' elements allows us to recover any Arnoldi vector under our error model.

Note that it is possible (and usual) to build the QR decomposition of the Hessenberg matrix H as the Arnoldi method goes, by computing the Givens rotation that corresponds to each new vector of the Arnoldi method. Q is thus computed as the set of Givens rotations, and $Q^T||r||_2 e_1$ is also updated at every step. Thus we could use the relation $H = QR$ by keeping a copy of H even while building R:

- Givens rotations are easily deducible from H, thus Q and R are recoverable from H

- Givens rotations are easily invertible, since inverting a rotation means rotating by the opposite angle. Thus H is recoverable from Q and R.

Even though space is a limiting factor in GMRES, the H and R matrices are respectively upper Hessenberg and upper triangular of size $m(m+1)$, thus much smaller than the set of Arnoldi vectors of size mn (with $m << n$). Agullo et al. consider H to be stored (and solved) redundantly [Agullo *et al.* 2013], which would then need no further protection. This also indicates that keeping the matrix H has a reasonable cost.

Listing 4.4: BiCGStab pseudo code with redundancies listed

1	$g, r, p \Leftarrow b - Ax$		
2	$\rho \Leftarrow < g, r >$		
3	for t in $0..t_{max}$		
4	$q \Leftarrow Ad$	p double-buffered	
5	$\alpha \Leftarrow \rho / < q, r >$	$q = Ad$	
6	$s \Leftarrow g - \alpha q$	$g = b - Ax$	$q = Ad$
7	$t \Leftarrow As$	$s = g - \alpha q$	
8	$\omega = < t, s > / < t, t >$	$t = As$	$s = A^{-1}t$
9	$x \Leftarrow x + \alpha p + \omega s$	$x = A^{-1}(b - g)$	$p = A^{-1}q$
10	$g \Leftarrow s - \omega t$	$t = As$	$s = A^{-1}t$
11	check convergence		
12	$\rho_{old} \Leftarrow \rho$		
13	$\rho \Leftarrow < g, r >$	$g = b - Ax$	
14	$\beta \Leftarrow \rho / \rho_{old} * \alpha / \omega$		
15	$p \Leftarrow g + \beta(p - \omega q)$	$q = Ad$	$p = A^{-1}q$

4.3.1.3 Bi-Conjugate Gradient Stabilised

BiCGStab is one of the generalizations of CG to matrices that are non-SPD. The pseudo-code for this method and the relations that may be used to make it resilient are presented in Listing 4.4, similarly to what has been done for CG. r is static, along with the usual A and b. BiCGStab exhibits more redundancies than CG, and only an example set of relations that can be used is shown.

With $q = Ap$, $s = g - \alpha q$ and $t = As$, updating g can be rewritten $g \Leftarrow g - \alpha Ap - \omega As$. Thus we have another way of computing g if for example q is faulty, but we also verified that the algorithm still conserves $g = b - Ax$.

Note that other assignments can also be expressed as slightly more complicated updates, we have e.g. $s \Leftarrow s - \omega t - \alpha q$. The reverse also holds true, from the update of x we may get a direct relation such as $x = A^{-1}(b - s + \omega t)$

4.3.2 Preconditioned algorithms

The described recovery techniques can be straightforwardly applied to the same algorithms with a preconditioner. To preserve the generality of our approach, and to avoid preconditioners specifics, we consider a generic preconditioning operation "solve $Mu = v$", M being the preconditioning matrix. To derive protected versions

of the preconditioned algorithms we have to protect all the linear operations involving the preconditioned vectors. Protecting the execution of the preconditioner itself is beyond the scope of this work, but a topic of complementary work, describing for example how to effectively protect multi-grid preconditioning [Casas *et al.* 2012].

To recover part of a preconditioned vector, there is no general way to avoid re-applying the preconditioner. Therefore, the prerequisite for the recovery to be cheap is the ability to perform a partial application of the preconditioner, that is, to apply the preconditioner to a small subset of v such that all lost data in u is recovered. If M is a block-diagonal matrix, solving $Mu = v$ only on the set of blocks that supersedes the lost data achieves this. If M is a fixed point method's matrix, the sparse set of elements in v that contribute to the lost portion of u is sufficient. If M denotes a multigrid method, we consider the nodes of the coarsest grid that participate to producing lost data, then we only need the inputs that contribute to these nodes for recovery. In any case, re-running the preconditioner completely is a viable, though slow, forward recovery for u. Finally, a corrupted v after a "solve $Mu = v$" operation is always recoverable without using the equation $Mu = v$. This is an important point since M is not always explicitly formed.

This can be made explicit by looking at the preconditioned versions of CG, BiCGStab, and GMRES, which are shown in Listings 4.5, 4.6 and 4.7. We can easily observe that in both CG and BiCGStab the preconditioned vectors z, p and s always exist at the same time as their non-preconditioned counterparts, r, p and g, because the latter are still used in the solver. Thus we can always recover the preconditioned vectors as discussed in the previous paragraph. All the relations protecting operations that involve z or r in CG, and p, s, d or g in BiCGStab are detailed next to the code of the preconditioned versions. For preconditioned GMRES, shown in Listing 4.7, the main redundancy relation from its non-preconditioned counterpart linking all the v^k is still valid. The only addition is the need for r to be conserved for the possible recovery of x.

4.3.3 Implementing Recovery with Asynchrony

CG and BiCGStab are harder to protect, as they require both redundancy relations and double buffering approaches to be fully protected, while GMRES just requires redundancies. For this reason, as well as because CG is a very popular method

Listing 4.5: Preconditioned CG

```
1  ϵ_old ⇐ +∞
2  r ⇐ b − Ax
3  for t in 0..t_max
4      solve Mz = r          r = b − Ax
5      ρ ⇐< z, r >           Mz = r
6      β ⇐ ρ/ρ_old
7      p ⇐ βp + z            Mz = r
8      q ⇐ Ap
9      α ⇐ ϵ/ < q, p >
10     x ⇐ x + αp
11     r ⇐ r − αq
12     ρ_old ⇐ ρ
```

Listing 4.6: Preconditioned BiCGStab

```
1  g, r, p ⇐ b − Ax
2  ρ ⇐< g, r >
3  for t in 0..t_max
4      solve Md = p          p double-buffered
5      q ⇐ Ad                Md = p
6      α ⇐ ρ/ < q, r >
7      r ⇐ g − αq            r = g + ωt
8      solve Ms = r
9      t ⇐ As                Ms = r
10     ω =< t, r > / < t, t >
11     x ⇐ x + αd + ωs       d = A^{-1}q, Ms = r
12     g ⇐ r − ωt            r = g − αq
13     ρ_old ⇐ ρ
14     ρ ⇐< g, r >           r = g + ωt
15     β ⇐ ρ/ρ_old ∗ α/ω
16     p ⇐ g + β(p − ωq)     p double-buffered
```

Listing 4.7: Preconditioned GMRES

```
1  for t ln 0..t_max
2      r ⇐ b − Ax
3      solve Mz = r          r = b − Ax
4      v^0 ⇐ z/||z||_2       Mz = r
5      for l in 0..m − 1
6          u ⇐ Av^l
7          solve Mw = u
8          for k in 0..l
9              h_{k,l} ⇐< w, v^k >
10             w ⇐ w − h_{k,l}v^k
11         h_{l+1,l} ⇐ ||w||_2
12         v^{l+1} ⇐ w/h_{l+1,l}
13     solve H = QR
14     y ⇐ R^{-1}Q^T||z||_2 e_1
15     x ⇐ x + Σ_{l=0}^{m-1} y_l v^l   x = A^{-1}(r − b)
```

for solving SPD matrix equations in the HPC context, we select it to test our approach. We implement two versions of CG, one without a preconditioner and a second one using a block-Jacobi preconditioner. Any conclusion obtained from our experiments with CG can be trivially extended to the other two since they constitute a similar and simpler use-cases respectively, and to their preconditioned versions as explained in Section 4.3.2.

We start by presenting the implementation of CG for a shared-memory model, and extend it to a distributed memory systems in Section 4.3.4.

4.3.3.1 Conjugate Gradient's Parallel Decomposition

The pseudo-code for CG is given in Listing 3.1, and its parallelization in tasks is done by strip-mining as shown in Figure 4.1a, with each set of tasks being named after the value or vector it outputs. Dependencies between tasks are generated from annotations to the sequential code, and represented by arrows on this graph. Tasks are then scheduled asynchronously by the runtime according to this data-flow. Some dependencies that do not affect the ordering or scheduling of tasks are not drawn for the sake of clarity.

Sets of tasks depicted in white represent operations that are strip-mined into as many parallel tasks as available threads. Blue tasks (after converging arrows) depend on all the previous tasks (because of a reduction operation) and represent a single task producing a scalar value. They are thus de facto synchronization points. The lattice-like arrows describe the fact that each following task depends on each previous task, as the block-row matrix-vector multiplication takes a whole vector as input for each single block as output.

4.3.3.2 Packing Recovery Tasks out of the Critical Path

We divide those relations in blocks as described in Section 4.2.2 and maintain an atomic bitmask (e.g. an `int`) per block of failure granularity, thus per memory page. Each data vector and task output is represented by a bit in this mask. Thus, if a task T works on a page p of a vector, it can check whether (one of) its inputs(s) is corrupted, and if so skip the computation while marking the bitmask with the bit representing T's output. We similarly skip computations that have inputs whose computation was skipped, due to their inputs being corrupted. This is necessary as to keep track of when errors happen and avoid overwriting data potentially needed

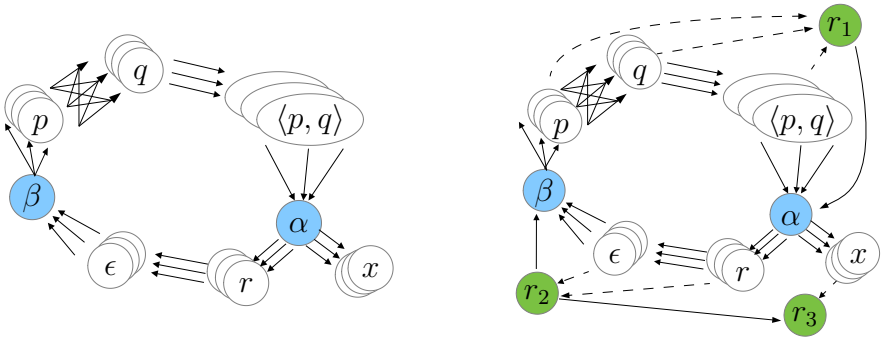
for recovery, and works especially well with linear relations (which are the majority of considered relations). There is a memory overhead directly proportional to the size of the linear system n to store this information.

Skipping computations is critical for reductions, because a floating point accumulation can be irremediably corrupted by adding (or multiplying by) $+/\text{-}inf$ or nan. Through a thread-private `sig_atomic_t` variable, each task is made aware of interruptions, and only contributes a page-level accumulation to the task-level one when no errors were reported. For this page based division to be valid, the reduction needs to be associative, which is always guaranteed since it is already required for the strip-mining into tasks.

While errors are not corrected, the skipping of computations that depend on not-produced data propagates through the different tasks. When reaching a scalar task, skipping dependent computations would mean stop progressing completely. Thus we have to recover errors before the said scalar tasks. The graph in Figure 4.1b shows the modified cycle with the green tasks where recoveries take place: replacing lost data and recomputing skipped computations.

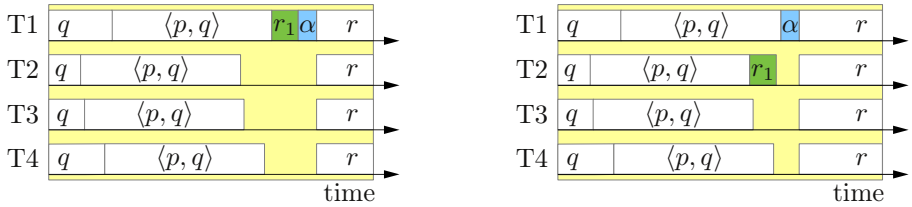
Each recovery task recovers the inputs and outputs of normal tasks which point to it with dashed lines. Recovery tasks are always added to the execution flow of the program and check the global variables for signalled errors. If none occurred, the recovery tasks do nothing.

The more conservative approach allows the recovery tasks to execute in the critical path, that is, waiting for all computations that do not need lost data to



(a) textbook version (b) with recovery tasks

Figure 4.1: Task decomposition of CG. Circles represent tasks producing the data inscribed on them, with white sets of tasks for strip-mined operations and blue for tasks producing a scalar, and full arrows representing data dependencies. Green tasks implement recoveries of the tasks linked to them with dashed arrows.



(a) FEIR, in critical path (b) AFEIR, overlapped

Figure 4.2: Traces illustrating the scheduling of recovery tasks

finish and only then run the recovery, as illustrated in Figure 4.2a. This option makes no compromise on the coverage of faults, since all the tasks (thus potential error discoveries) have finished executing when the recovery starts, as we will see with high error injection rates in Section 4.5.3. We call this technique Forward Exact Interpolation Recovery (FEIR).

Because dashed lines represent communication through atomic global bitmasks rather than dependencies, recovery tasks can execute concurrently to CG tasks, which do not touch the memory pages to be recovered, either skipping them or working on unrelated data. This allows to overlap computations and recoveries, thereby reducing overheads; however errors discovered between recovery tasks and the following scalar task are not recoverable. Thus we execute r_1 and r_2 asynchronously as late as the scheduler allows us to schedule them, which means concurrently with $< p, q >$ and ϵ respectively, and with a lower priority as to start all reduction tasks first, see Figure 4.2b. Faults discovered after the recovery tasks start being executed can not be corrected. This technique is the Asynchronous Forward Exact Interpolation Recovery (AFEIR).

The parallelization strategy for the preconditioned CG is exactly the same as for the non-preconditioned CG. The only added recovery technique is the partial "solve $Mu = v$", as explained in Section 4.3.2, which is easy to perform since block-Jacobi is a blocked preconditioner.

4.3.4 Recovery on Distributed Memory Systems

The recovery methods described so far apply to shared memory models, in a single node or in a partitioned global address space for example. We list here the few modifications needed to extend our resilient methods to distributed memory programming models like MPI, which we will use to evaluate how the recovery

techniques impact the scaling of the application in highly parallel scenarios. We use our hybrid implementation of CG where the node level parallelism is leveraged using MPI and the intra-socket parallelism is exploited by the asynchronous task-based data-flow programming model, OmpSs. The necessary additions, which are two MPI reductions after the local reductions and exchanging parts of p, modify the task graph as follows:

- Global MPI reductions, occur during the α and β tasks (see Figure 4.1).
- A new exchange task to exchange locally updated parts of p with neighbouring nodes depending on it, instead of the lattice-like dependencies between tasks p and q.

For our FEIR and AFEIR methods, we instantiate a second r_1 recovery task to be executed before our new exchange task, to avoid sending potentially failed and non-corrected data. Finally, MPI communications added inside the task recovering the x vector, r_3, request and perform exchanges of parts of x when needed for recovery, since this vector is not exchanged at every iteration.

4.4 Other Recovery Approaches

4.4.1 Trivial Forward Recovery

The trivial forward recovery consists in simply keeping the program running, by allocating new (uninitialised) memory for corrupt or lost data. This is the minimum required, due to the OS retiring affected memory pages, and no other actions are taken. While an error in a part of the data that is not reused later would be masked, we lose all guarantees on convergence.

4.4.2 Rollback Recovery

Checkpointing is applied only to dynamic data in CG, to be fair in our comparison of methods, and to be consistent with the assumptions on which the DUE recovery relies. Each Processing Element (PE) periodically writes to its local disk the values of the iterate and search direction vectors it has at that given moment (x and p), which is the minimum to allow rolling back. The checkpointing rate is computed

for each experiment to minimise execution time, taking into account the time to write and read checkpoints, the MTBE, and no downtime [Bougeret *et al.* 2011].

There is no need to use a parallel file system, since we assume that the program will not crash (as we catch the errors). At rollback, each PE restores the vector portions that were saved to its local disk at the last checkpoint.

Checkpoints and rollbacks are global, that is, they involve all the PEs of a parallel run. In a distributed memory scenario, we perform a global MPI reduction once per iteration to decide whether a rollback is needed. This global communication is executed simultaneously with the α MPI reduction to avoid further synchronization overheads.

4.4.3 Lossy Restart

Langou *et al.* [2007] present a forward recovery method for the fail-stop model of an MPI process, the Lossy Approach, applicable to all Krylov-subspace methods. To compensate for the loss of a part of the iterate x, a step of the block-Jacobi is used, which relies only on static data and the remaining parts of x. This operation is similar to our recovery for the iterate, while discarding the residual in the block relation. After such an interpolation, a restart is necessary since the residual r is outdated and not easily deducible.

We adapt this Lossy Approach into a recovery for our error model, that we name the Lossy Restart:

1 If part of the iterate is lost we use the interpolation from the Lossy Approach. With i the failed block:

$$A_{ii}x_i = b_i - \sum_{j \neq i} A_{ij}x_j$$

2 We restart the method with either the intact or the newly interpolated iterate as initial guess.

Before comparing these methods, let us consider theoretical results presented on this interpolation, by noting x^* the solution of the system $b = Ax^*$, x the iterate, x^I the newly interpolated iterate, $e = x^* - x$ and $e^I = x^* - x^I$ the respective errors, and $r = b - Ax$ and $r^I = b - Ax^I$ the respective residuals. Langou *et al.* show the following:

Theorem 1. *The interpolation is contracting for a constant* $c_i = (1 + ||A_{ii}^{-1}|| \sum_{j \neq i} ||A_{ij}||)^{1/2}$, *thus* $||e^I|| \leq c_i ||e||$.

This result grants the block-Jacobi's fixed point property: if $x = x^*$, then $x^I = x^*$, since $e = 0$. For A SPD, Agullo *et al.* [2013] show the following:

Theorem 2. *With A symmetric positive definite, the interpolation diminishes the A-norm of the error:* $||e^I||_A \leq ||e||_A$.

This is a stronger result, as it additionally guarantees that the A-norm of the error, which decreases along the iterations in CG, keeps decreasing through interpolation recoveries. Thus a CG (or PCG) algorithm made resilient with the Lossy Approach strategy is guaranteed to have a monotonically decreasing error.

From here on, we will restrict ourselves to SPD matrices and show that the block-Jacobi step does not just give better replacement data, but the best possible – in the short run.

Theorem 3. *For A SPD, the interpolation* minimizes *the A-norm of the error* $||e^I||_A$ *over all possible values for* x_i^I

The proof of our theorem relies on the transformation of the error implied by the linear interpolation, $p_i^I : e \to e^I$ being a linear projection, orthogonal for the norm $|| \cdot ||_A$

Proof. By construction, the residual at x^I for the block i is $r_i^I = b_i - \sum_{j=0}^{n-1} A_{ij} x_j^I = 0$. Let us also notice that $r = b - Ax = A(x^* - x) = Ae$ and similarly $r^I = Ae^I$.

Now let us show that the kernel and image of p_i^I are orthogonal for A:

$$\begin{aligned} \forall e \in \Re, < p_i^I(e), e - p_i^I(e) >_A &=< e^I, e - e^I >_A \\ &=< Ae^I, e - e^I > \\ &= \sum_{j=0}^{n-1} < r_j^I, e_j - e_j^I > \end{aligned}$$

This is always zero, since for $j = i$, $r_j^I = 0$ and for $j \neq i$, $x_j^I = x_j$ thus $e_j = e_j^I$. It then comes clearly that:

$$||e||_A = ||p_i^I(e)||_A + ||(Id - p_i^I)(e)||_A$$

where $p_i^I(e)$ depends solely on the e_j (thus x_j) with $j \neq i$. Hence the minimum of this norm for all possible x_i, or e_i, is reached in $p_i^I(e)$. □

We can also deduce this from Theorem 2 and the fact that the unknown part of x is in the kernel of p_i^I. $||e^I||_A \leq ||e||_A$ then holds for any x_i, hence the *min* relation of our theorem.

Restarting the solver, with a good or unmodified initial guess, still harms the superlinear convergence of CG, which relies on the fact that the sequence of iterates x minimizes at each iteration the norm $||x^* - x||_A$ on a sequence of increasing subspaces. However, this disturbance may benefit methods who have a tendency to stagnate (such as GMRES).

All recoveries based on restarting are identical as long as the iterate is untouched, and trade in convergence properties for simplicity of recovery in the same way. It is to be expected that such methods would behave very similarly to the Lossy Restart, though always worse in the short run, hence it is the only restart method against which to compare.

4.5 Evaluation

We run experiments on an Intel® Xeon® E5-2670, with one thread on each of its 8 cores. Our evaluation is done on two versions of CG, a non-preconditioned version to show the hardest case possible, and one using a block-Jacobi preconditioner. Due to the wide variety of preconditioners available for CG, it is impossible for us to evaluate every single one. We list in Section 4.3.2 the desirable properties of preconditioners for an efficient recovery. The block-Jacobi is simple to implement, and trivially applicable to a subset of a vector. We select it also because, if its block size coincides with the memory page size, the factorization of diagonal blocks for the recovery of single errors is already computed. Thus we will use diagonal blocks of 512 by 512 elements, which coincides with 4KB page sizes.

We compare the following methods: our Forward Exact Interpolation Recovery (FEIR) without asynchrony (recovery tasks in the critical path), our Asynchronous FEIR (AFEIR), the Lossy Restart, checkpointing-rollback to local disk, and trivial forward recovery. The optimal checkpointing rate is used whenever errors are injected, and no fallback is used for FEIR or AFEIR: simultaneous errors on related data are simply ignored (see Section 4.2.4).

Table 4.2: Resilience methods' overheads, no errors

	Lossy	trivial	AFEIR	FEIR	ckpt 1K	ckpt 200
overhead	0.00%	0.00%	0.23%	2.73%	17.62%	46.20%

Table 4.3: Increase of time spent per state for FEIR methods

	imbalance	runtime	useful
AFEIR	4.30%	8.11%	1.90%
FEIR	25.06%	7.84%	2.78%

4.5.1 Techniques Overheads

From here on, the "ideal" CG will refer to our version of CG with no resilience mechanisms nor error injections.

We present in Table 4.2 the harmonic means of overheads for all methods in absence of faults, compared to the ideal CG. The Lossy Restart and trivial techniques have no overhead when no errors are injected, since catching the error, replacing memory pages and ordering a restart is done in a signal handler which is never called. To give a sense of checkpointing cost we arbitrarily consider checkpointing periods of 1000 and 200 CG iterations. This is because the optimal checkpointing rate without injecting errors would be to never write checkpoints. Outside of these overhead measurements, we use the optimal checkpointing rate explained in Section 4.4.2. The corresponding overhead raises from 17.62% to 46.20% as the checkpointing frequency increases, which constitutes a significant cost.

The overheads associated to the AFEIR and FEIR techniques are much smaller since they are associated to activities like task creation or scheduling, that are much cheaper than writing data to disk. The asynchronous nature of the AFEIR technique allows to compensate much of the overhead incurred by the FEIR technique. We can see in Table 4.3 a detailed breakdown of what is involved in the overheads of the FEIR and AFEIR methods, expressed as the increase of the proportion of time spent in each state while the solver is running: either idle, thus suffering load imbalance, or performing runtime work, such as creating and scheduling tasks, or finally executing tasks, thus doing computations for the solver. Executing the recovery tasks in the critical path obviously increases the load imbalance.

Most of the runtime overhead of FEIR and AFEIR techniques could be removed if application-level resilience were supported by the runtime, instantiating recovery tasks only when DUE are signalled.

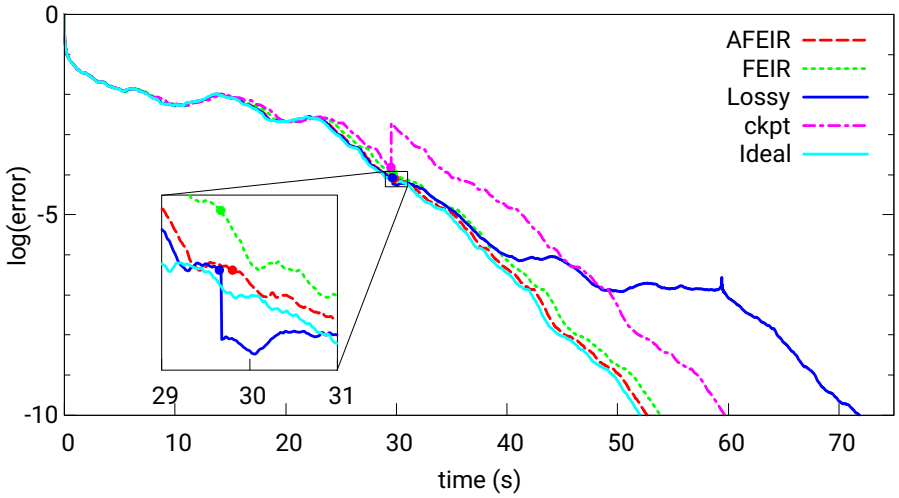


Figure 4.3: CG convergence for different resilience methods with matrix thermal2 and same single error injection in the iterate x around $t = 30$s

4.5.2 Convergence

Figure 4.3 illustrates the convergence of CG for a sample scenario consisting of a single error injection. The x-axis represents the time and the y-axis shows the execution progress in terms of the logarithm of the residual norm defined as $||Ax-b||/||b||$, updated at each iteration. The ideal CG is represented by the cyan line; all the other experiments have a single error injected 30 seconds after the beginning of the execution at a certain memory page that contains a portion of the iterate x. Before the error is injected, each resilience method pays its typical overhead in absence of faults. The purple line corresponding to the checkpointing mechanism is the one with more overhead, which is consistent with the analysis presented in Section 4.5.1, as we use a checkpointing frequency of 1000 iterations. At the time of the error, it already incurs a 9.12% slowdown. Once the error is injected, the checkpointing mechanism rolls back a certain number of iterations and resumes progress from there. The Lossy Restart, represented by the blue line, has an immediate reduction in the error thanks to its block-Jacobi step interpolation, but converges slower afterwards because restarting harms CG's superlinear convergence. The FEIR and AFEIR methods recover the lost data by using an exact interpolation and keep progressing. The overhead paid by the AFEIR technique is significantly smaller than the one paid by FEIR, since asynchrony allows most of the recovery work to be overlapped with other computations.

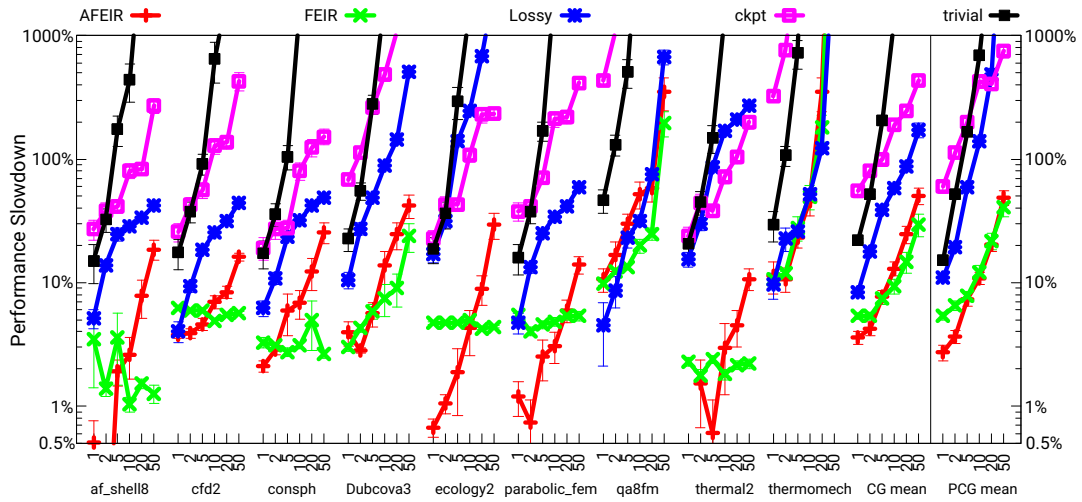


Figure 4.4: Comparison of the execution time for resilience methods and matrices, varying error injection rates

4.5.3 Shared-Memory Performance

Figure 4.4 shows an exhaustive evaluation of the performance slowdown associated to the 5 resilience mechanisms listed in Section 4.5: trivial, checkpointing-rollback, Lossy Restart, FEIR and AFEIR. We consider the same 9 input matrices as for the overhead measures, and 6 error injection scenarios per matrix and method, which means that we provide an evaluation of 270 different experiments. Each experiment has been run over 50 times and Figure 4.4 reports their harmonic mean, and standard deviation as error bars. In each repetition, the errors have been injected randomly at different times and memory pages. On the x-axis of Figure 4.4 we display the name of the considered matrices and, for each matrix, the error injection frequency normalised to the ideal CG's convergence time τ for that matrix. A value n means an error frequency of $\frac{n}{\tau}$, thus an MTBE of $\frac{\tau}{n}$. In other words, n is the expected number of errors injected during the ideal convergence time τ. The convergence times per matrices range from 60ms to 54s, meaning the MTBE ranges from 54s down to 1.33ms. These very high error rates allow to stress test all the recovery methods, while the overheads analysis presented in Section 4.5.1 gives indications on which techniques to prefer when errors are less likely. The y-axis is displayed in logarithmic scale and shows the measured performance slowdown in percentage for each experiment, with respect to the ideal CG. A slowdown close to 0 means the resilient CG converges at a speed close to

that of the ideal one, whereas a bigger slowdown means its convergence is slower. The convergence threshold is 10^{-10}.

We have run the exact same 270 experiments with the block-Jacobi Preconditioned CG (PCG), and report the mean of those results, displayed at the right hand side of Figure 4.4.

The trivial method reacts badly against few errors and its convergence times diverge extremely fast, with overheads over 200% with a normalised error frequency of 5 only. For PCG, the overheads of the trivial recovery reach 50% with a normalised frequency of 2 and become larger than 700% for frequencies of 10 or more. The checkpointing scheme reacts better than the trivial method, with substantial overheads that tend to increase slower, ranging on average from 55% to 433% for CG and from 60% to 752% for PCG. These convergence times are close to the expected values from the checkpointing frequency computation [Bougeret *et al.* 2011]. The Lossy Restart behaves better on average than the trivial method and the checkpointing schemes. Regarding CG, it has an overhead of 8.4% with one expected error per ideal convergence time, reaching up to 87% and 170% against 20 and 50 times higher frequencies respectively, whereas for PCG these overheads are 12.80% and 500% for normalised frequencies of 1 and 20. This better behaviour of the lossy mechanisms with respect to checkpointing and trivial techniques is already reported in the literature [Agullo *et al.* 2013], and the fact that our experimental framework has reproduced known results demonstrates its accuracy and reliability. This is matrix specific however, as non-ABFT methods are clearly outperformed for af_shell8 or cfd2 but have overheads similar to Lossy for thermal2.

The most important fact of our evaluation is that methods FEIR and AFEIR behave much better than the current state-of-the-art resilience techniques for iterative solvers. When applied to CG, FEIR has an overhead of 5.37% and 29.68% under normalised frequencies of 1 and 50, whereas AFEIR has overheads of 3.59% and 50.47% respectively for the same error rates. On PCG, FEIR and AFEIR have an overhead of 5.36% and 2.72% with the smallest frequency, and reach 40.55% and 48.55% with the largest.

While FEIR has a roughly constant overhead on most matrices, the impact of recoveries in the critical path can be seen where execution times are the shortest, such as for Dubcova3 and especially qa8fm and thermomech. As recoveries run

on a per iteration basis, the error rates per iteration determine the chances of encountering errors on related data and during recovery tasks. Under injection frequencies of 20 and 50, both qa8fm and thermomech experience over 0.2 and 0.6 errors per iteration, which significantly pulls up the mean overheads. Such extreme cases, which correspond to MTBEs of 5ms and less, could be dealt with by using more recoveries per iteration, or a fallback method for unrecoverable errors. These matrices also show that Lossy Restart is most efficient on fast converging problems. Finally, these matrices and fault rates are the only ones that display occurrences of multiple simultaneous corrections on related data. This is due both to the high error rates and the small sizes of the matrices, causing the relative size of a page in a vector to be significant.

AFEIR is slower than FEIR for high error injection rates, because errors happening between the end of a recovery task and its following scalar task are unrecoverable. That is the time between the end of $r1$ or $r2$ and the beginning of α and β respectively, as illustrated by Figure 4.2b. With very high error injection rates, the probability of an error happening during these time windows may cause the contribution of a memory page to $\langle$p,q$\rangle$ or ϵ (see Figure 4.1b) to be ignored. Depending on the matrix and the actual data lost, this might have a significant impact, as matrix ecology2's behaviour shows. The FEIR method is not at risk of discovering an error after a recovery task ran, because these tasks start after all computations are done. However, both methods are still vulnerable during the recovery's execution. There is thus a trade-off between the low overheads of AFEIR at frequencies of 10 and less, and a more conservative approach, FEIR, which trades in some convergence speed for safer recoveries and is thus useful at higher error rates. Given an execution trace of the CG solver, the probability of missing an error with AFEIR could be quantified, as it corresponds to the number of pages that are touched during the load imbalance preceding the execution of a recovery task. As the injection rate is normalised to the execution time, this probability is higher with a smaller number of iterations, but also with matrices that suffer from higher load imbalance. The same trade-off applies to the PCG results for low error injection rates. The precomputed factorization of diagonal blocks reduces recovery time, thus a block-Jacobi preconditioner weakens this trade-off for high error injection rates. It is to be expected that when using a preconditioner

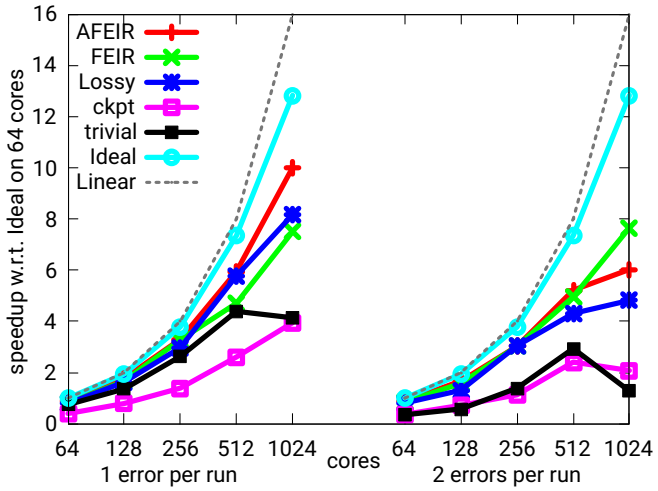


Figure 4.5: Speedup of the MPI+OmpSs resilient CGs

whose partial application is computationally hard (see Section 4.3.2), the average recovery time will increase and this trade-off will become stronger.

4.5.4 Scaling Results

In this section, an evaluation of the scalability of our recovery techniques is performed with a hybrid MPI + OmpSs implementation. Experiments are run in the MareNostrum supercomputer, whose nodes contain two Intel Xeon CPU E5-2670 sockets. Each MPI rank is mapped to one 8-core socket, running 1 OmpSs thread per core. We consider runs on 8, 16, 32, 64 and 128 sockets (64 to 1024 cores), since we need 8 sockets to fit the matrix in memory.

We present in Figure 4.5 a complete evaluation in terms of speedup, injecting one and two errors per run. The speedups are computed taking the execution time of an ideal CG on the smallest possible core count, 64, as a reference. We display data concerning the FEIR, AFEIR, Lossy Restart, checkpointing and trivial techniques, and include the ideal CG's and linear speedups for reference. Our MPI+OmpSs CG implementation achieves a parallel efficiency of 80.17% on 1024 cores in a faultless run, which highlights its quality in terms of parallel performance.

AFEIR and FEIR techniques clearly overcome the trivial and the checkpointing techniques, achieving speedups of 10.01 and 7.50 respectively when 1 error is injected and 6.03 and 7.65 against two errors on 1024 cores. The Lossy Restart

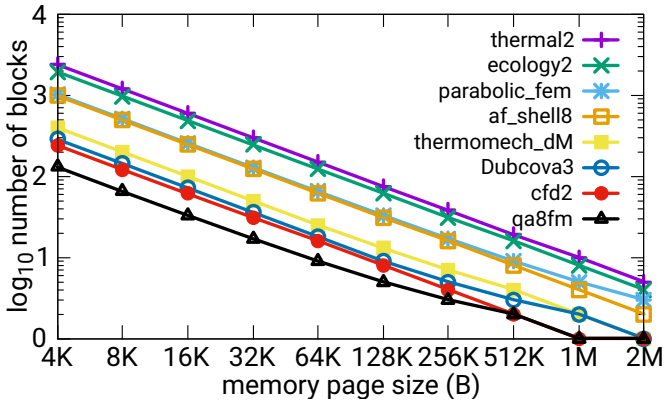


Figure 4.6: Subdivision of CG vectors in blocks, with one block per page

achieves speedups of 8.17 and 4.82 respectively on 1024 cores. It is worth noting that only a few tens of iterations are required to achieve convergence for the 27-point stencil matrix, which causes any overhead to be important compared to the ideal execution time, but also makes this matrix the ideal workload for a restart method. Even in this case, restarting is as costly if not more than our FEIR and AFEIR methods' overheads. Regarding checkpointing, writing vectors to disk already causes the checkpointing to perform significantly worse than our baseline, and when injecting errors its speedups stay below a third of the ideal CG, close to that of the trivial method.

4.6 Analysis of Data Loss Granularity

While the checkpointing scheme recovers full vectors, the algorithmic techniques recover or replace data at the memory page level, which is usually 4KB. However, some systems use different page sizes, and some HPC codes rely on huge pages (2MB) to improve performance, hence we present an in-depth study of the impact of varying page sizes on recoveries and convergence rates.

The impact of using huge pages on the ideal performance (without resilience methods nor error injections) was evaluated as well, however details are left out due to space constraints. Overall, the number of TLB misses decreased drastically with negligible impact on convergence time.

The number of pages, thus blocks of data erased by a single error, per vector (b, x, r, p, q, see Listing 3.1) are displayed in Figure 4.6. At the biggest page sizes, a single page may contain a vector entirely, thus full vectors can be erased

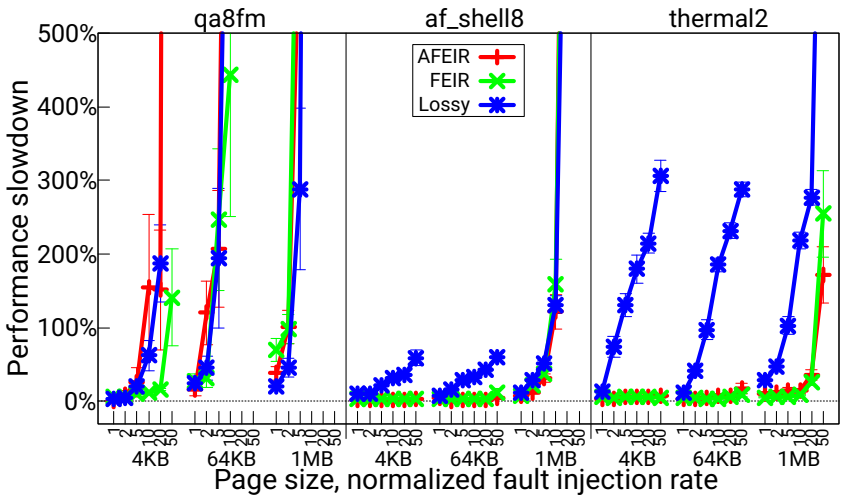


Figure 4.7: Overheads of algorithmic techniques for 3 selected matrices and 3 page sizes, presented linearly

by a single error, as is the case for qa8fm and cfd2 starting at 1MB page sizes, and Dubcova3 and thermomech_dM at a page size of 2MB. In this case, the Lossy recovery consists of solving the problem by factorising directly the whole matrix until the resulting vector is not subject to faults.

4.6.1 Impact of page size depending on matrix size

To show the different impact of page size on different matrices, we display in Figure 4.7 the overheads for 3 different page sizes and 3 matrices: the smallest, qa8fm, the biggest, thermal2, and an intermediate one, af_shell8, which appear respectively at the bottom, middle and top of Figure 4.6. These matrices represent all possible behaviours that we observed. The plotted values are, as in previous Figures, average performance slowdown compared to the "ideal" baseline execution time, and this data is a subset of the data that will be presented in Figure 4.8. Each point corresponds to the harmonic mean of up to 70 runs divided by the ideal time, where each configuration consists of a recovery method, matrix, page size, and fault injection rate.

The overall tendency is the same as previously: FEIR and AFEIR perform significantly better than Lossy in most scenarios, with the notable exception of big page sizes for small matrices. However, looking at differences per matrix sizes, we note that all methods diverge for qa8fm under high fault injections rates, except FEIR with 4KB pages which incurs a 141% slowdown. For af_shell8, until 64KB

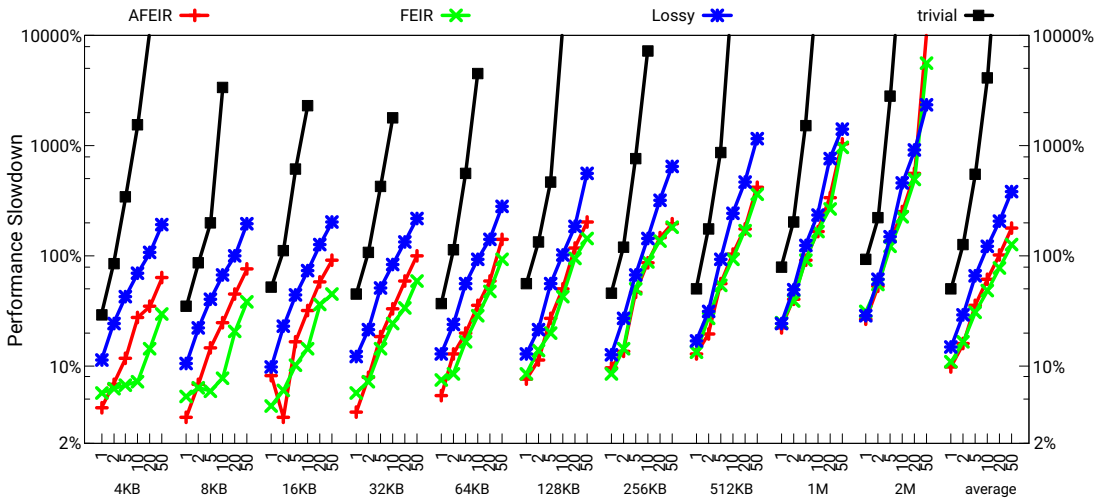


Figure 4.8: Evolution of the overheads of algorithmic techniques for page sizes from 4KB to 2MB, varying error injection rates

pages the FEIR and AFEIR methods stay below 11% while Lossy peaks at 60%, and with 1MB pages the fault rates are similar until a fault injection rate of 5 faults per run, with quick divergence after that. The biggest matrix, thermal2, behaves similarly as FEIR and AFEIR have low slowdowns for page sizes of 4KB and 64KB while Lossy's slowdown seems roughly proportional to the fault injection rate, until 307%. With 1MB pages we see a similar divergence to previous matrices for Lossy, while the increase in slowdowns is slower for FEIR and AFEIR with 254% and 171% respectively.

This Figure allows to illustrate that the bigger the matrix, the better FEIR and AFEIR perform in comparison to Lossy. This is in part due to bigger matrices usually having worse convergence [Hestenes and Stiefel 1952]. Such matrices will incur more slowdown from restarting than from exact recoveries, such as FEIR's and AFEIR's. Coincidentally, dividing a vector into more blocks due to a matrix being bigger also reduces the probability of having two faults affect related data, which adversely impacts FEIR and AFEIR as explained in Section 4.2.4.

4.6.2 Overall Page Size Evaluation

Figure 4.8 presents the average overheads of all algorithmic techniques (trivial, FEIR, AFEIR, Lossy) for CG, with different page sizes. We repeat for all page sizes ranging from 4KB to 2MB the full evaluation done in Figure 4.4 for each

matrix, technique (except checkpointing), and fault injection rate, for a total of 2160 different configurations. We use the same methodology as explained in Section 4.5 and compute the harmonic means of slowdowns (convergence time divided by baseline execution time) for each configuration of up to 70 runs. We then report the harmonic means of all the matrices' slowdowns. That is, every point is the average of all the averaged runs of all the matrices, for that page size and fault injection rate. The first graph "4KB" coincides with the graph titled "CG mean" in Figure 4.4.

Overall, the trade-off between asynchronous and critical-path recoveries discussed in Section 4.5.3 is still applicable up to 64KB page sizes, above which FEIR and AFEIR show similar performances. The high cost of recoveries for 128KB page sizes and above makes the choice of overlapping recoveries largely irrelevant. Both methods outperform the Lossy Restart strategy significantly up to 512KB page sizes, above which all the convergence times are similar, and even to the advantage of Lossy for an error injection rate of 50 with 2MB page sizes.

At the same time, all forward recovery methods (FEIR, AFEIR and Lossy) still outperform by far the trivial method, even though it does not perform any computation. The trivial method often does not converge at all, as is the case with normalised error injection rates over 5 with page sizes of 128KB or more.

With increasing page sizes comes an increasing cost of computations during recoveries, and we can indeed see how increasing recovery costs drive convergence times up for all algorithmic techniques. The minimum slowdowns of AFEIR, FEIR and Lossy are all around 28% for a single expected fault per run at 2MB pages against 4.2%, 5.5% and 11.2% respectively with 4KB pages. This means all methods still outperform checkpointing, whose overhead for a single expected fault is 55% as presented in the previous Section. Maximums values increase, and for 1MB pages the slowdowns of FEIR, AFEIR and Lossy are 1072%, 1146% and 1516.59%. As in Section 4.5.3, these high numbers are mostly driven by the overheads of small matrices.

The different techniques are affected differently by the page size modification. For similar convergence rates, the Lossy recovery has higher rates of converging runs than FEIR and AFEIR. However, these runs are also on average much longer (up to 1000 times), and their duration has much higher variability. This is expected as with both the FEIR and AFEIR techniques several simultaneous faults

may cause a failure to recover, whereas Lossy only relies on static data and thus can always continue solving. The smaller relative differences between the expected convergence times of all methods indicate that with high page sizes, the convergence time is dominated by recoveries.

With increasing page sizes, all interpolation recoveries become less performant, however the exact ones (FEIR, AFEIR) still outperform the Lossy approach, and all these techniques remain better than the checkpointing approach for low fault rates.

4.7 Conclusions

This chapter demonstrates that hardware DUE reporting can be exploited jointly with redundancy relations to protect iterative solvers paying a very low cost. Our two proposed methods, FEIR and AFEIR, overcome the state-of-the-art techniques in terms of overheads. They are moreover based on very simple relations that do not require deep algorithmic understanding, whereas an algorithmic technique like the Lossy Approach [Langou *et al.* 2007] is harder to derive. Varying page sizes up to 2MB has shown that our methods retain lower expected convergence times than the current state-of-the-art, until the edge case where vectors fit in a single memory page. At that moment, all methods become roughly equivalent due to single errors erasing full vectors. In short, our methods' efficiency is increasing with matrix size, relative to state-of-the-art methods. These straightforward low-overhead recoveries open the door to wide-spread use of algorithmic-based techniques to protect iterative methods when DUE detection is available. On the other hand, one could apply this technique even more efficiently if provided with the unaffected data on the memory page. If, instead of allocating a new page on encountering a DUE, the OS allowed to migrate all non-affected value to a new page, then the size of the recovery becomes trivial: a single ECC word, thus 64B at most with current ECCs.

Second, the contribution demonstrates that by overlapping recovery with algorithmic computation, overheads can be drastically reduced. Under high error rates, of roughly more than 0.1 errors per second, the overlapping stops paying off since the chances of getting errors on non-protected computations increase, even though this trade-off is largely matrix specific. The FEIR technique provides

then better performance. In any case, task-based data-flow programming models have interesting properties for resilience, not only because of inherently splitting programs into tasks, but also because overlapping computations and recoveries is done without explicit programmer intervention. Runtime support for application-level resilience could reduce the overheads by injecting recovery tasks only when errors are encountered – this would also increase AFEIR's coverage by executing the recovery later, and still asynchronously.

Our resilience method opens the door to interesting trade-offs when SDC comes into play. Since we cover with very low overhead nearly all memory page failures, an ECC that focuses more on detecting than correcting errors would reduce SDC [Kim *et al.* 2015], while delegating some correction to the application level. This work will hopefully encourage future architectural, OS and runtime features to expose errors at the application level whenever lower level recoveries fail, allowing resilience aware applications to resist significantly higher fault rates than applications oblivious to resilience.

Chapter 5
Vulnerability Analysis

5.1 Introduction

In memory subsystems, all storage bits are uniformly protected with the same level of ECC. This means that bits in memory that will have no impact on the program are protected at the same expense than bits that will have a major impact on the program's outcome. This is because it is impossible to statically assess which storage bits are more likely to affect program reliability. Indeed, as the placement of data in memory can change at every program execution, all bits in DRAM are interchangeable. Thus, to quantify the risk of error in memory, it is necessary to use a metric at the scale of a single program execution, depending on the data stored at a given bit, and on the way the program uses this data.

In this chapter, we examine the *Memory Vulnerability Factor (MVF)*, which is a special case of the AVF [Mukherjee *et al.* 2003], with the scope of analysis limited to memory. It is targeted at approximating the probability of error in a program, and has been used under different names in the literature. We then extend this metric to take into account false errors, which are reported but would otherwise have no impact on the program being run. The MVF metric is dynamic, adapting to program behaviour, yet is not application-specific as it can be used for any program. We measure it using a cycle-accurate simulator, as well as all previous generalisations of AVF to memory [Yu *et al.* 2014; Luo *et al.* 2014; Gupta *et al.* 2018]. We compare these metrics to the probability of an architecturally incorrect program outcome due to a fault in its data, obtained from real-world fault injections. These experiments demonstrate that the false error aware memory vulnerability metric correlates best with the probability of an architecturally incorrect program outcome, and gives a consistent upper bound on this probability.

This chapter is organised as follows: in Section 5.2 we define the vulnerability metrics for memory, and present in Section 5.3 the comparison between sensitivity to faults and the vulnerability ratings. The insight about false errors gained from this new vulnerability metric allows to reduce failure rates by delaying error reporting, which we quantify in Section 5.4. In Section 5.5, we present early results of similarly inspired savings in DRAM refresh energy, before presenting our concluding remarks in Section 5.6.

5.2 Metric Definition

The goal of quantifying memory vulnerability is to assess which bits matter to the correct execution of a program. In order to do this, we first present a classification of the possible outcomes of a program, then introduce the vulnerability metrics from the literature. We then present the *False Error Aware (FEA)* vulnerability metric, by analysing how false errors can in fact be ignored.

5.2.1 Linking Program Outcome and Vulnerability

We categorise the outcome of a program as correct when its execution is indistinguishable from an execution without errors. This includes executions with errors that are corrected, ignored, or benign (thus have no measurable impact on the final state). Incorrect executions may be due to a program that finishes running improperly (e.g. crash), stops making progress, finishes but returns an incorrect result, or finishes and returns a correct result but having performed more work than a non-faulty execution. The goal of assessing vulnerability in memory is to know, dynamically, which bits have a higher likelihood of causing an incorrect outcome. We also call incorrect outcomes *failures*.

When encoding data with ECC, data bits are stored together with redundant bits, as an ECC codeword. This makes computing a per-bit metric such as vulnerability slightly more complex. For example a SECDED code means any single bit error in the codeword is correctable [Hamming 1950]. However, the bits in the codeword still affect the state of the program. To reflect this, we attribute to an ECC codeword stored in memory the average vulnerability of all the bits in the unencoded data. This allows to quantify the importance of the data that is encoded beyond the simple ECC strength considerations. The properties of the

AVF are also maintained: the structure's value is still the average of that of all its bits. Thus, in order to assess the real impact of faults in data stored in memory, we need to measure the program outcomes when injecting faults in the unencoded data bits.

5.2.2 Existing Metrics for Memory Vulnerability

We reuse all the metrics that are presented in Section 2.4.1:

- MVF, which is the AVF scoped to memory only, and coincides with the probability of loading data from memory. *MVF is the fraction of time a memory location contains data that will be loaded*

- DVF [Yu *et al.* 2014], which is the product of the program execution time T, the size S_d and number of accesses $N_{ha,d}$ per memory structure, and the fault rate:

$$DVF_d = FIT \cdot T \cdot S_d \cdot N_{ha,d}$$

  It is worth noting that the fault rate FIT and T do not vary per data structure, thus the differences between DVF among various data structures is only due to their size and number of accesses $S_d \cdot N_{ha,d}$.

- The store-to-load ratio ST/LD [Gupta *et al.* 2018]. The authors state that "most periods of data 'deadness' end in a write, so more writes indicate more dead intervals."

5.2.3 Accounting for False DUE

MVF is overestimating the probability of a bit affecting the program outcome, by counting as vulnerable data that is fetched only to be overwritten. Let us consider a write or a set of writes that spans a full ECC word (thus 8B for SECDED or 16B for ChipKill-level ECC [Dell 1997]). In a cache with a write-allocate policy, if these writes cause a cache miss, data will be fetched from memory. Any miscorrected errors in this ECC word are masked by the new data being written. Similarly, any uncorrectable errors would be masked as well, if they did not trigger an exception. These are called false errors. Contrarily to benign errors, which affect the program in a negligible way, false errors are caused by faults in data that is not consumed

by the program. Thus, in a system without ECC, or with an ECC scheme that has no DUE [Abdoo and Cabello 1996], MVF incorrectly categorises that data as vulnerable, as it is based only on fetching the data.

Thus we can write off false errors, as they should not affect the program state. Accordingly, we differentiate between MVF, whose definition is given in Section 2.4.1 and which limited to what happens in memory, and False Error Aware MVF (FEA). *FEA is the fraction of time a memory location contains data that will be consumed.* That is, we consider a memory location as safe not only when it is next accessed by a store, but also when it is next accessed by a load whose contents will be overwritten without being used. The data is considered vulnerable in memory only before a load whose contents will be used.

5.2.4 Vulnerability under Transient Fault Models

Both the MVF and FEA metrics relate naturally to the probability of consuming a fault, under the common hypotheses for transient fault models.

If we suppose that faults happen as a random memoryless process, we can model them using an exponential model, with λ the average fault arrival rate. This assumption is commonly made [Li *et al.* 2007], and is a particular case of the gamma and Weibull distributions, which are also often observed [Levy *et al.* 2018]. Formally, we define for any location in memory S and U, the sets of safe and unsafe accesses to that location (i.e. respectively stores and loads for MVF), and t_a and f_a, the time and the number of faults consumed by any access a respectively. Since safe accesses overwrite faults, we have $a \in S \Rightarrow f_a = 0$. For unsafe accesses, we consider the period p_a before an access a. This lasts $p_a = t_a - t_{prev(a)}$, where $prev(a) = \max\{b \in S \cup U | t_b < t_a\}$. Thus for unsafe accesses, we have:

$$P(f_a > 0) = 1 - P(f_a = 0) = 1 - e^{-\lambda p_a}$$

Hence the overall probability of consuming a fault is $P(F) = \sum_{u \in U}(1 - e^{-\lambda p_u})$. If we reasonably assume that faults are rare, i.e. that the program execution time T is such that $\lambda T \ll 1$, then we can approximate $P(F) \approx \lambda \sum_{u \in U} p_u$, which is the total time spent before unsafe accesses multiplied by λ.

The vulnerability V is the fraction of time a memory location contains data that is unsafe, thus with the same notations, $V = \frac{1}{T} \sum_{u \in U} p_u$. We then have

$P(F) \approx \lambda TV$, which confirms the intuition of the vulnerability V being proportional to the probability of consuming a fault in memory.

The DVF metric can be formulated as $\lambda T\,(S_d N_{ha,d})$, confirming that only the size and number of accesses per data structure should be used to compare DVF against other vulnerability metrics, as indicated in Section 5.2.2.

5.3 Evaluation

In this section, we compare program's sensitivity to errors against MVF, FEA, and other metrics used by related work. To perform this comparison, we examine outcomes when injecting faults in native runs of 12 parallel benchmarks, and measure vulnerability ratings precisely using a cycle-accurate simulation infrastructure.

The simulator setup is presented in Section 3.2, and the mechanisms to inject errors in native runs and classify an experiment's outcome are detailed in Section 3.1. The benchmarks used are a subset of the benchmarks presented in Section 3.3.

5.3.1 Comparing Metrics and Fault Injections

The results of fault injections in real runs are presented as bars in Figure 5.1, while the various vulnerability ratings obtained from cycle-accurate simulations are presented as lines. The bars represent the probability of an incorrect program outcome, and are obtained from fault injections, with whiskers on top of the bars representing the 95% confidence interval for the failure probability. Benchmarks are sorted in increasing failure probability for DUE. The vulnerability metrics, displayed as lines, are MVF, defined in Section 2.4.1 and similar to the *safe ratio* [Luo *et al.* 2014], FEA, defined in Section 5.2.3, the store-to-load ratio [Gupta *et al.* 2018], and finally DVF [Yu *et al.* 2014] in the separate graph above.

The failure probabilities are varied per benchmark and per data structure within each benchmark. First, failures are always increasing with the number of bits flipped, except for the transition from 2 to 3 bit flips for CG which is roughly constant. Similarly, we should note that the probability of a failure due to a DUE is always bigger than for any number of bit flips, sometimes by several orders of magnitude. This is a known phenomenon, called *derating* [Mukherjee *et al.* 2003]. At every level of the vulnerability analysis, only a portion of the errors

have an impact: only some transistor errors make a gate misbehave, gates affect the circuits they compose only a part of the time, only some circuits have an effect on the operation being performed, ans so on. As such, the vulnerability factor is a derating factor at the level of memory analysis, and the difference between DUE vulnerability and bit flip vulnerability corresponds to the algorithmic derating factor for that bit. Indeed, with the DUE error model, as soon as the bit is consumed we consider it the program execution a failure, while in the bit flip error model we let the program run with its data silently corrupted and measure the outcome.

Some benchmarks are very tolerant to faults, such as KNN, which uses training points which are assigned a class, and classifies a different set of points based on the classes of their nearest neighbours. Indeed, a bit flip in the biggest data structure, which is the set of points used for training, modifies at most one point, which in itself does not significantly alter the outcome of the classifications. Benchmarks such as Cholesky, Jacobi, Gauss-Seidel and Red-black are more resilient to bit flips due to the nature of their data. Modifying a single of many floating point values is likely to only have a small impact on the final outcome. The outcome can however be significantly perturbed if this error is significant for Red-black and Gauss-Seidel, as indicated by the high failure probabilities for DUE. On the more vulnerable side of the spectrum, modifying any data in the precise computations of Blackcholes or the numerically unstable SMI directly impacts the output computations. Similarly, the biggest data structures in CG on are the matrix values, columns and rows. The two latter of those data structures are integer data, and used to index the values, thus even a single bit flip can cause the benchmark to crash. Finally, the impact of the numerical stability of the algorithm shows that the sensitivity to faults is a very application-dependent behaviour, as Cholesky and SMI solve similar problems with comparable input sizes, but have very different sensitivities to faults.

The various vulnerability metrics are presented as lines in Figure 5.1. MVF (unbroken line) and FEA (dashed line) often have the same value, and always provide an upper bound on the failure probability for all error types. FEA gives a much tighter bound for the Jacobi, FFT and Stream benchmarks, and a slightly tighter bound for Blackscholes and CG. This is due to these benchmarks having data structures being overwritten and not updated, thus false errors that can be ignored. Overall, FEA correlates very well with the failure probability for DUE injections, and noticeably overestimates the vulnerability only of KNN and

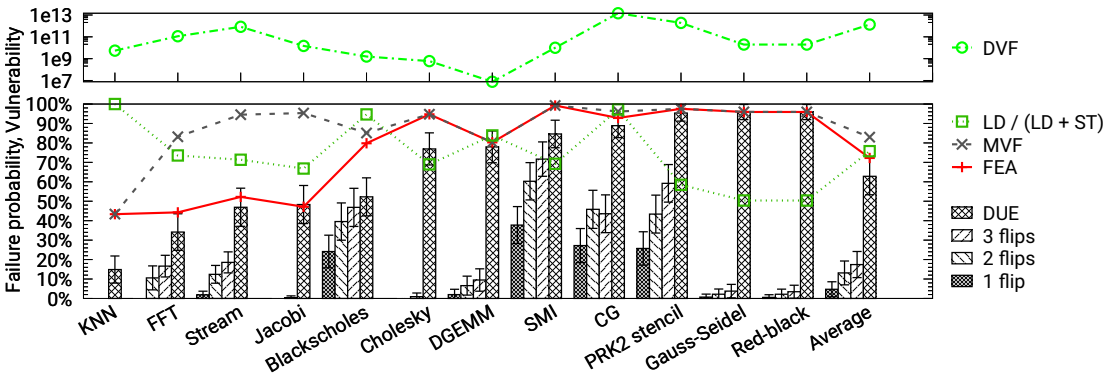


Figure 5.1: Incorrect outcome frequency when injecting faults in real runs (as bars), and simulation-based vulnerability ratings per benchmark (as lines).

Blackscholes. This is due to the fact that these benchmarks have a relatively small memory footprint, thus with an important error masking effect from the cache. The only two other benchmarks where FEA differs from the DUE fault rate are Cholesky and SMI, which are two benchmarks respectively factorising and inverting a symmetric matrix. In these benchmarks, the diagonal blocks are fetched entirely from memory, however only half of this data is used and thus affects the program outcome. The unused data that is fetched is not overwritten however, thus FEA does not identify it as safe. The reason why unimportant data is accessed is unclear, as the accesses are part of Lapack library calls. All remaining benchmarks have an average FEA value very close to the average failure frequency when injecting DUE. As such, it also gives a good proxy and consistent upper bound for the bit flip error injections.

As the store-to-load ratio ST/LD [Gupta *et al.* 2018] is unbounded and inversely correlated to the vulnerability, we normalise it as $LD/(LD+ST)$, and present this value as the dotted line in Figure 5.1. This transformation maintains the relative ordering of the ratings' values (in opposite direction), while bringing them back in the interval $[0, 1]$. We see several problems with this load-to-store ratio: KNN, which is a very safe benchmark, is rated with maximal vulnerability. Furthermore, the load-to-store ratio also under-estimates the vulnerability of several benchmarks, and can thus not be safely used as an upper bound on failure probability. On Cholesky, Gauss-Seidel and Red-black, this metric severely underestimates the probability of failure due to a DUE. Looking at errors caused by bit flips, this metric correctly rates Gauss-Seidel and Red-black lower than FFT and Stream, and those lower than Blackscholes and CG. However as many benchmarks

contradict this correlation. PRK2 stencil is rated as one of the safest benchmarks, even though it is the third most vulnerable for single bit flips and second most vulnerable for triple bit flips, and SMI is rated with an average load-to-store vulnerability while it is the most vulnerable benchmark for bit flips. While this metric might be a useful heuristic for online optimisations, due to the fact it is simple to compute, it is easy to see that it lacks the timing information to satisfyingly inform on vulnerability. One example is the iterate of CG, which is an iterative method that updates this vector at every iteration to get closer to the solution. The load-to-store ratio sees as many loads as stores, and thus rates it with a value of 50%, the lowest data structure of the whole benchmark. However the store immediately follows the load, as the iterate is simply updated, and a whole iteration subsides before the next update. Thus the iterate is in fact one of the most vulnerable data structures of CG, with 70% of failures for double bit flips and 96% for DUEs.

The DVF metric [Yu *et al.* 2014] is not bounded either and its values are high and very spread out, hence we display them on a log scale at the top of Figure 5.1. The main factor impacting the DVF of a data structure is its size, which causes benchmarks such as CG and PRK2 stencil to be rated with high DVF, however Blackscholes has low DVF while being the second most vulnerable benchmark for bit flips. Similarly, DVF correlates poorly with the probability of failures due to DUE, as Red-black, Gauss-Seidel and DGEMM are rated with middle to low DVF values comparing to other benchmark. Finally, the fact that the metric is not bounded makes it harder to use at runtime, as the metric only has meaning when comparing values relative to each other, making it impossible to set thresholds on DVF values for example.

5.3.2 Quantifying the Correlation Between Metric and DUE

We report in Figure 5.2 the Pearson correlation coefficients between all the vulnerability metrics and the failure rate. In systems with any level of ECC, DUE is the more likely type of error to be encountered, thus we only compare here the failure rate due to DUE injections. For each pair of metrics, the correlation is computed using all of the data structure values from every benchmark, normalised for data structure size within each benchmark. Metrics are sorted using the magnitude of correlation with DUE. In addition to the metrics used so far, we display the original ST/LD metric from Gupta et al. [Gupta *et al.* 2018], which allows to verify

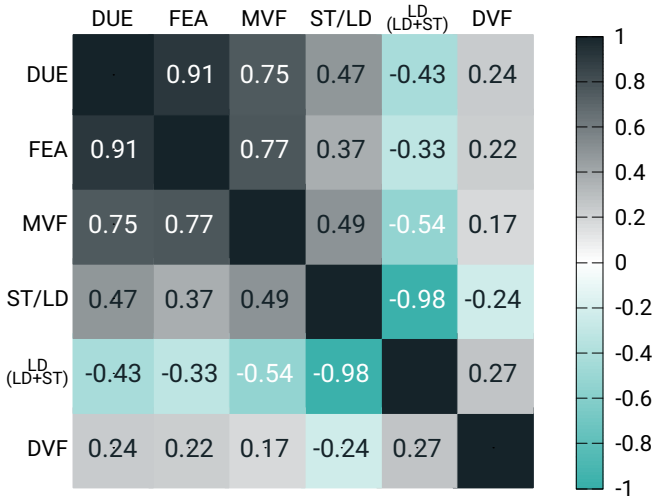


Figure 5.2: Correlations between the failure rate from DUE injections and the various vulnerability metrics.

that the normalised version, $LD/(LD + ST)$, gives the same information. Indeed, both metrics correlate with a factor of -0.98, meaning a very strong negative linear correlation.

Confirming our previous observations, the DVF is the metric correlating the worst with DUE fault rates. Furthermore, it correlates weakly (≤ 0.27) with all other metrics. While the DVF provides some information on vulnerability, it seems to be the less suitable metric. The ST/LD and $LD/(LD + ST)$ metrics correlate with the DUE failure rate with coefficients of 0.47 and -0.43 respectively. These two metrics use the same information, however it seems that the distribution of the non-normalised version is slightly better at informing on vulnerability. Interestingly, Gupta et al. verify the relevance of the ST/LD metric by comparing it against a MVF value obtained from simulation, and report a correlation of -0.32. This is contrary to our findings, even when computing the correlation with MVF per memory page instead of per data structure (respectively 0.37 for ST/LD and -0.40 for $LD/(LD + ST)$). It is worth noting however that in their work, Gupta et al. use benchmarks from different suites, for which they report much lower MVF (ranging from 1.7% to 22.5%) than what we measure for our benchmarks. This difference could be the cause for the discrepancy between the results. However, the opposite signs of correlations for this metric do not bode well for its reliability as an indicator.

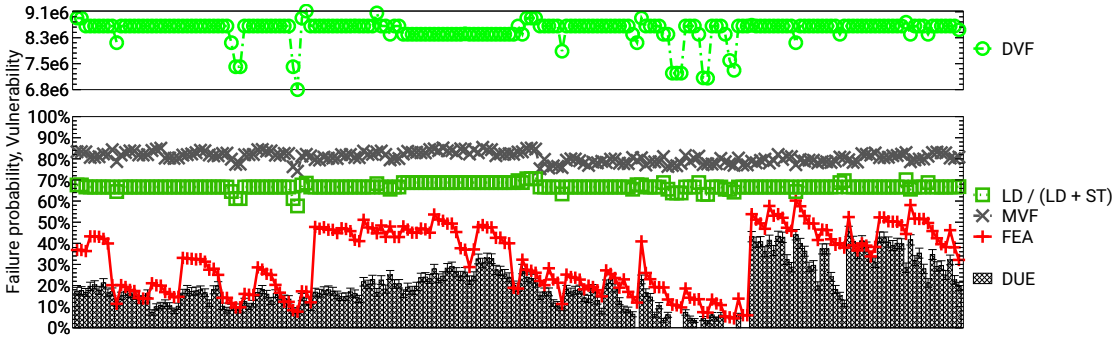


Figure 5.3: Failure frequencies from DUE injections and vulnerability metrics, for 200 random memory pages in FFT.

The MVF metric correlates rather well with FEA (0.77) and with the DUE failure rate (0.75), while FEA correlates really well with DUE (0.91). This confirms that the FEA improves the precision from MVF, and that the timing information which is exploited by these two metrics only is key in providing useful information on memory reliability.

5.3.3 Memory Page Comparison

Finally, to take a more detailed look at the differences between the metrics, we measure the DUE failure rate and report all metrics for 200 randomly selected memory pages from the FFT benchmark, presented in Figure 5.3. This Figure is the analogue of Figure 5.1 presenting averages per memory page instead of per benchmark, sorted on the memory page address. These pages are selected uniformly at random in the two data structures where FEA and MVF differ, which represent 80% of the benchmark's memory footprint. The 95% confidence intervals are presented as whiskers on top of the bars, and are all less than 5 percentage points, while vulnerability metrics are displayed as lines.

This figure reveals that the MVF, DVF and $LD/(LD + ST)$ metrics are roughly constant within a data structure. FEA behaves exactly as MVF on most data structures, which is why we have selected here data structures which have loads to memory whose contents are overwritten without being consumed. Thus FEA is capable of correlating much better with the error injection outcomes overall by using the behaviour of these memory pages, and it is apparent that even within the data structure the FEA metric correlates much better with the error outcome than the other metrics. Furthermore, the measured FEA values per memory page are

almost always above the mean failure frequency, and are always above the lower bound of the confidence interval. Thus we can say with 95% confidence that FEA provides an upper bound on the DUE injection rate. Finally, FEA identifies much more varied behaviour in memory than the other metric, meaning that this metric provides a much wider range of possibilities to exploit for memory vulnerability applications.

5.4 Delaying Error Reporting

The metrics evaluation reveals that there are a number of benchmarks where the impact of false errors significantly causes the failure probability to be overestimated. To ignore these false errors, we propose to delay the reporting of any error until it is actually consumed. This is a common pattern in resilience, used for example to handle DUEs in memory discovered during scrubbing. An application likely to access that erroneous data is not terminated preventively, but only whenever it attempts to access the location of the error [Kleen 2010]. Similarly at the architectural level, delaying a machine check exception due to an incorrect instruction allows to avoid exceptions for instructions that do not affect correctness (no-ops, prefetches, etc.), or whose results are either not committed or ultimately do not affect the program [Weaver *et al.* 2004].

Scrubbing errors and accessing incorrect data with a load are explicitly listed as recoverable uncorrected errors, with respectively optional and required software action [Intel 2017; AMD 2018]. However, there is no mention of how errors detected in DRAM are handled when accessing incorrect data indirectly, such as overwriting it or accessing neighbour data such as another word in the same cache line. The error codes leave the possibility for these errors to be reported as optional-action uncorrected errors until they are overwritten, after which they would appear as requiring no action for recovery.

In the case that deferring error reporting does not exist for errors in fetched DRAM values, we outline two simple mechanisms to support it. As modern microarchitectures already track poisoned data from other causes, this tracking can be reused and possibly expanded to mark lines that contain errors originating in DRAM. For example, marking data in cache as both dirty and poisoned would stop

a load to this cache line to attempt to re-fetch from memory, instead triggering the appropriate machine check exception.

Alternately, on hardware where DUE in memory are reported immediately as uncorrectable and unrecoverable, the OS can ignore false errors by relying on the machine check exception mechanism. If the DUE location is directly accessed, the DUE should only be reported for a load, and ignored for a store. If the DUE location is accessed indirectly however, such as a word in the same cache line than the data that is accessed, the OS can catch the machine check exception and track the next memory access to this address. As soon as the ECC word has been overwritten, the OS can definitively ignore the DUE and stop tracking accesses to this location. On the other hand, if an access loads the ECC word that has not been reported, the OS should propagate the reporting of the DUE.

To track accesses to a location with low overhead, hardware watchpoints are available on commodity hardware and supported by modern OSs [Intel 2017; AMD 2018; IBM 2015b]. These watchpoints raise an exception when a monitored memory location is accessed, and impose no overhead for non-monitored accesses. Krishnan [2009] reports overheads of the order of 5ms to register an address across all processors of a 24-CPU Xeon MP system, around 200µs to unregister a watchpoint, and around 2µs to handle a triggered watchpoint exception, while accessing an address that is not monitored incurs no overhead.

We can quantify the reduction in the DUE rate using the previous metrics. Let FIT_{UF} be the rate of faults causing uncorrectable errors (for example double bit flips for SECDED). In current systems, as soon as the location of an uncorrectable error is accessed, a DUE is triggered. Thus, since the average probability of accessing data is MVF, the DUE rate is $MVF \times FIT_{UF}$. If we delay reporting the error to ignore false errors as proposed, the DUE rate becomes $FEA \times FIT_{UF}$, as errors are only triggered when data is consumed and FEA is the average probability of consuming data.

We present in Figure 5.4 the reduction in DUE rate that can be achieved with the proposed mechanism to delay and possibly ignore false DUE. A number of benchmarks have their average DUE rate reduced by over 45%: Stream with 45.4%, FFT with 48.1%, and Jacobi with 50.5%. FFT (using the Stockham algorithm) and Jacobi are benchmarks that can not perform in-place computations. Instead, these algorithms use additional memory to store results of intermediate

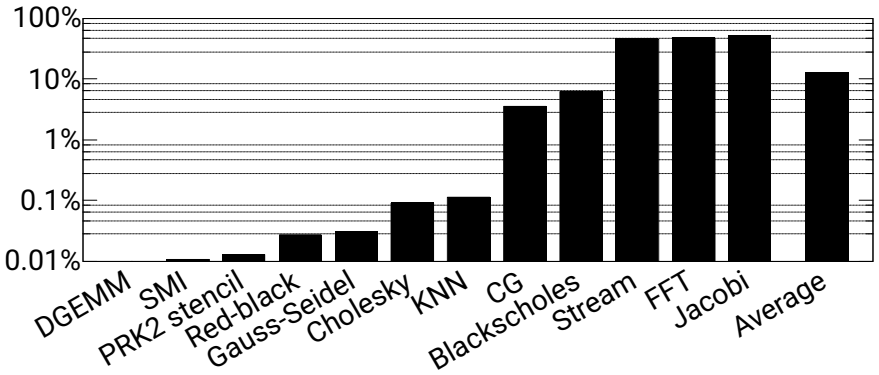


Figure 5.4: Gains in DUE rate when ignoring false errors

computations, which creates memory accesses that overwrite data. Other benchmarks only overwrite a single of their data structures, such as CG and Blackscholes. The remaining benchmarks have no or negligible amounts of data (less than 0.1%) that is accessed without being consumed. Overall, over 12 benchmarks, the average reduction in DUE rate is of 12.75%, for a proposal that imposes no overhead in a scenario without errors, and the low overhead of a hardware watchpoint in the event of a delayed DUE.

5.5 Saving DRAM Refresh Energy

A major obstacle in the future of DRAM is the Refresh Wall [Stuecheli *et al.* 2010; Qureshi *et al.* 2015]. This is due to multiple simultaneous factors: having more data in memory to refresh, at an ever faster pace due to increased leakage power, and the duration of a refresh operation increasing due to write recovery time [Kang *et al.* 2014]. Refresh power already constitutes an important part of the DRAM power, and other sources of power consumption in DRAM are diminishing. For example, optimisations such as low voltage swing terminated logic [JEDEC 2016] and power efficient data encoding [Seol *et al.* 2016] considerably reduce the I/O power consumption of DRAM.

The refresh rate is adjusted so that nearly all cells function without losing data. This means that the refresh rate is provisioned for the most leaky cells acceptable. Thus, a number of refresh schemes propose to adapt to the hardware variability, refreshing more often rows with leakier cells, and refreshing at lesser frequencies

rows that exhibit better retention times. One such proposal is AVATAR [Qureshi *et al.* 2015], which adapts dynamically to cells that start displaying VRT.

A complementary approach is to adapt the refresh rate to software variability. That is, identify portions of memory that are unused, and avoid refreshing those. This is already implemented in hardware for lower-power states. Partial Array Self Refresh (PASR) is a low-power mode where only a subset of the banks of the LPDDR are self-refreshed, and the data in the other banks is lost [Elpida 2005]. For normal DRAM usage, this idea was first proposed abstractly by Ohsawa *et al.* [1998], and embodied in proposals to identify and avoid refreshing allocated-but-uninitialised or freed memory [Isen and John 2009], and memory in the OS's pool of unallocated pages [Baek *et al.* 2014]. However, opportunities to avoid refreshing data that is allocated and used by the program remain.

Indeed, as the difference between MVF and FEA underlines, a number of programs have data stored in memory only to be overwritten. A runtime system can use the semantic knowledge it has of future accesses, through data-flow dependencies of tasks, to know which data to stop refreshing without risking to lose information from the solver.

5.5.1 Overwriting as a Runtime Contract

A task may declare some memory regions as their inputs and outputs. For example, the triad task in Listing 5.1 shows a function creating a set of tasks that compute an array `a` from two arrays `b` and `c`. In order to execute in the correct order, the task declaration (line 4) declares, for each task working on a block of size `bs` of the arrays:

- `in(b[j;bs], c[j;bs])`: the task needs to wait for the blocks of arrays `b` and `c` to be computed,
- `out(a[j;bs])`: the task computes the corresponding block of array `a`.

If a task modifies a value, instead of computing it from scratch, this should be expressed as simultaneously an input and an output dependency. The shorthand for this type of dependency is `inout()` (see also Section 2.2.2). This should be used when the value is updated (e.g. incremented). In particular, if only a subset of the declared range is overwritten and the rest is left untouched by a task, then:

Listing 5.1: Triad task of the stream benchmark

```
1 void triad(unsigned N, double a[N], double b[N], double c[N], double scalar,
      ↪ unsigned bs)
2 {
3   for (unsigned j = 0; j < N; j += bs)
4     #pragma omp task in(b[j;bs], c[j;bs]) out(a[j;bs]) label(triad_kernel)
5     for (unsigned i = j; i < j + bs && i < N; i++)
6       a[i] = b[i] + scalar * c[i];
7 }
```

- either the whole region needs to be declared as input *and* output
- or only the parts of the region that are modified should be marked an output.

This is so that the correct ordering of tasks is enforced, and that the semantic information provided by the tasks is correct.

From this, we can conclude that when entering a task with a memory region declared as output, this region will be fully overwritten. With input dependencies, several tasks can access a single memory region simultaneously or in arbitrary order. Output dependencies however create either read-after-write or write-after-write dependencies, thus a strict ordering in tasks before and after a task with an output dependency. Thus, when the next task accessing a memory region in the task graph has this memory region declared as an output dependency, we know that it is the only task to be the next to access this data. Since we also know that all the data in the memory region will be overwritten, looking ahead in the Task Dependency Graph (TDG) allows to find out which data can be discarded safely by the program.

5.5.2 Prospective Gains from Skipping Refresh

Using the TaskSim infrastructure presented in Section 3.2, we simulate the potential grains from skipping refreshes. Data that can be discarded is identified using the state-of-the-art ESKIMO technique [Isen and John 2009] to identify uninitialised or freed memory, and by looking ahead in the task graph to identify task outputs. The non-refreshed memory locations only start being refreshed again when it is next accessed. We know the next access is always a store, as the data will be overwritten, and verify that this is the case during the simulation.

We measure refreshes that can be skipped, by measuring the amount of contiguous data for which refreshes can be skipped at a DRAM row granularity, and

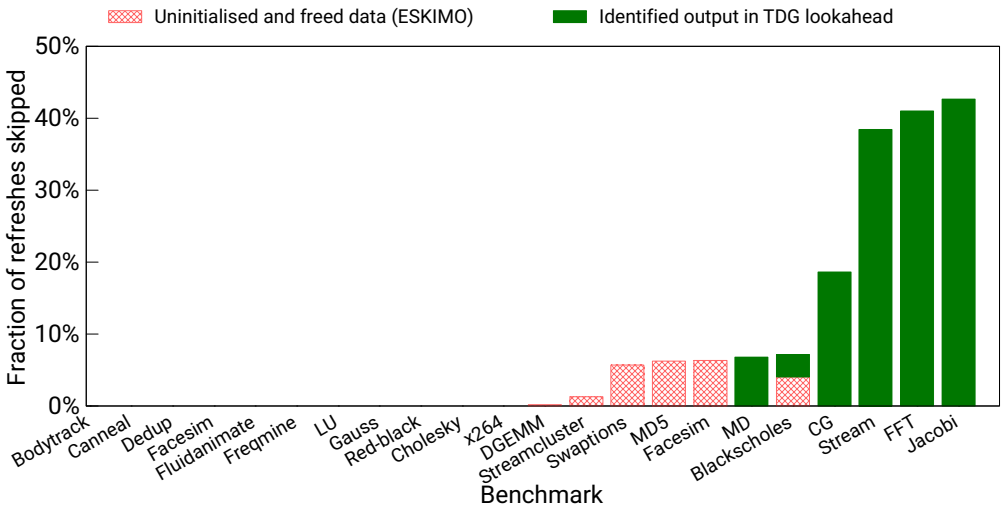


Figure 5.5: Fractions of DRAM refreshes that can be skipped per benchmark, cumulating state-of-the-art (ESKIMO) and proposed TDG lookahead techniques to identify inconsequential data

the duration for which refreshes can be skipped using a 64ms refresh period. In our configuration, a DRAM row represents 1KB. That is, for each region R of size S_R for which we identify that refresh can be skipped at a time T_R, we have $\lfloor \frac{S_R}{1KB} \rfloor$ rows. For each of these rows, the number of saved refreshes is $\left\lfloor \frac{\text{Time(next store to row)} - T_R}{64ms} \right\rfloor$.

However, for fair comparison, we have to compare the number of saved refreshes to the number of refreshes that can be attributed to the benchmark. We compute this using the duration of the program Δ, and the memory footprint M of the program, as measured using the *massif* tool of valgrind [Nethercote *et al.* 2006]. We round these numbers up respectively to the refresh duration of $64ms$ and the size of a DRAM subarray, which is 512KB [Kim *et al.* 2012]. Then, the fraction of refreshes skipped is computed for each benchmark as:

$$\frac{1}{\lceil \frac{\Delta}{64ms} \rceil \lceil \frac{M}{512KB} \rceil} \cdot \sum_{\text{Region } R} \left(\left\lfloor \frac{S_R}{1KB} \right\rfloor \sum_{\text{row } i \in R} \left\lfloor \frac{\text{Time(next store to row } i) - T_R}{64ms} \right\rfloor \right)$$

Results are presented in Figure 5.5, and include results from benchmarks presented in Section 3.3, from the PARSEC benchmark suite parallelised using OmpSs [Chasapis *et al.* 2015], and a Molecular Dynamics (MD) simulation, using the Lennard-Jones potential. Benchmarks are sorted in increasing order for the

fraction of refreshes skipped. It is apparent that a number of benchmarks make good use of their allocated memory, as no potential savings can be found by either technique. Most benchmarks that show a good reduction in DUE rate in Section 5.4, such as Jacobi, FFT, Stream, and CG in a lesser measure, also have a high fraction of refreshes that can be skipped. This shows that overwriting data in memory allows refreshes to be skipped, as long as the duration between the last access and the next store is long enough. ESKIMO manages to pick up a number of regions for which to skip refreshes that are not caught by the task output approach, especially for C++ benchmarks whose memory is not managed manually, such as Facesim, Swaptions, and Streamcluster. It is interesting to see that the two approaches are usually exclusive. Only one benchmark shows potential gains for both approaches, which is Blackscholes.

5.6 Conclusion

A number of metrics aim at quantifying the risk associated with encountering an error in data in memory. Comparing these metrics with the likelihood of incorrect program outcomes due to a fault in memory indicates that the metric we introduce in this chapter, FEA, is the most accurate one and gives a consistent upper bound on failure probability. This can be explained by the fact that it takes into account timing effects, as opposed to DVF or store-to-load ratios, and the fact that it takes into account errors in data that is inconsequential, as opposed to MVF. This work further exploits the insight on inconsequential data by proposing to track DUE until they are consumed or ignored, instead of directly reporting the errors. This technique would significantly decrease error rates due to DUE in memory, by 12.75% on average and as much as 50.5% for the Jacobi solver. We also present a runtime technique to anticipate which data stored in memory is inconsequential, and promising prospective results for DRAM refresh savings. Finally, this work opens the door to runtime-level optimizations that can now accurately model the risk associated with any given data.

Chapter 6

Dynamically Adaptable ECC Protection

6.1 Introduction

ECC, which is necessary to detect and correct random errors in memory, comes with overheads in terms of storage space and thus power consumption. This increased power cost constitutes an important constraint in areas from HPC [Kaul *et al.* 2012] to mobile devices [Micron 2017], making it undesirable to uniformly increase ECC strength as an answer to increasing fault rates. It is then preferable to protect against transient faults by selectively applying strong ECC to high-risk memory regions only, while cheaper protection can be used on lower risk parts. For this to be possible, we develop in this chapter a methodology to automatically and dynamically quantify the vulnerability of the different portions of data stored in memory.

Data accesses to a given memory location 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 the program state, as illustrated in Figure 6.1. Being able to detect this kind of behaviour enables the use of different levels of ECC protection.

The key idea of this chapter is to exploit the opportunity presented by runtime systems to *adapt ECC protection dynamically using online estimates of memory vulnerability*. These estimates are obtained by deploying low-overhead sampling techniques to analyse the memory access patterns of parallel applications on real systems. These online sampling techniques incur very low overheads and are able to precisely estimate the vulnerability of data stored in memory. This information is then exploited to dynamically target the most vulnerable memory pages for

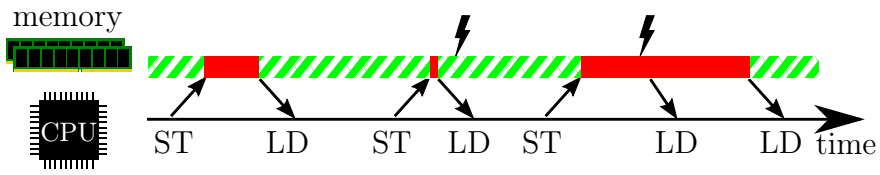


Figure 6.1: Vulnerability timeline for a memory location.
Stores (ST) to a memory location overwrite the data it contains, while loads (LD) retrieve it. Time before a store is safe (in hatched green), but before a load it is vulnerable (in solid red), as the first fault (depicted by a black lightning bolt) has no effect, while the second affects both subsequent loads.

increased ECC protection, with the aim of reducing the failure probability while minimising the memory storage overhead.

The main contributions presented in this chapter are:

- A runtime sampling methodology to estimate the vulnerability of memory regions online. This methodology is independent of hardware, as it relies on runtime-level information, requiring only the ability to sample instructions randomly. It is also independent of software, as it estimates the vulnerability of memory by analysing access patterns, requiring no application-level knowledge.

- A runtime-adjustable ECC scheme, WITSEC (WITSEC Is Targeted Stronger Error Correction), which can increase protection for dynamically selected memory pages, to tolerate one more bit flip than a uniform baseline. WITSEC is entirely implementable in the memory controller.

- An in-depth evaluation of the impact of selectively increasing memory protection based on our vulnerability estimates, implemented for a POWER8 processor, and measured on 11 parallel benchmarks. Guiding WITSEC with our vulnerability estimates allows for a wide range of trade-offs between failure probability and redundancy. This allows to choose just the minimal redundancy for a target probability of error, or to minimise the error probability under some budget for redundant storage, without having to pick only between uniform ECCs.

This chapter is organised as follows: Section 6.2 presents the fault model and defines memory vulnerability. In Section 6.3, we present our sampling-based memory vulnerability estimation methodology, and in Section 6.4 the adaptable ECC,

WITSEC. In Section 6.5 we describe the experimental setup, in Section 6.6 we present the evaluation, and conclude in Section 6.7.

6.2 A Metric for Memory Vulnerability

In order to know which data in memory benefits from stronger ECC protection mechanisms, we first need to look at the mechanisms of failures caused by faults in memory.

6.2.1 Modelling Faults in Memory

This work focuses on transient faults, as they always require some level of ECC to be applied. Alternative and more efficient techniques exist to correct permanent faults, such as sparing components, but there is no efficient way to protect memory against effects such as particle strikes. Thus, all data in memory is always at risk of having some bits flipped. We refer to the number of bits flipped as the multiplicity of the fault: single faults correspond to one bit flipped, double faults to two, and so on.

The effects of overwriting or consuming faults, and the cascade of possible consequences, are represented in Figure 6.2. At the top of this diagram, the access to a memory location, which defines its vulnerability. A fault that is consumed is called an error, which may be transparently corrected by ECC, leading to an *ok* outcome. Alternately, errors can cause DUEs, which raise a machine check exception. Finally, errors may go undetected and cause SDC if there are more bits flipped than the ECC can tolerate.

Finally, the impact on the running program, shown at the bottom of Figure 6.2, can be classified as:

- *ok:* Indistinguishable from an execution without faults.
- *slow:* The program finished with a correct result, but performed more work than a non-faulty execution.
- *hang:* The program stopped making progress.
- *wrong:* The program finished, yielding an incorrect result.

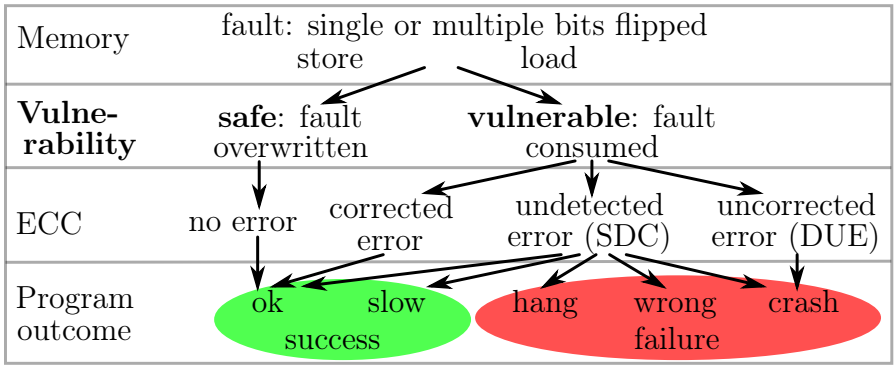


Figure 6.2: Taxonomy of resilience: faults cause errors, which can affect the program outcome based on memory accesses, ECC, and application-dependent behaviour. In bold, the level corresponding to the vulnerability metric.

- *crash:* The program finished running improperly.

We classify the outcomes *ok* and *slow* as successes, and the outcomes *hang*, *wrong*, and *crash* as failures.

This cross-level view from faults to success or failure of a program highlights that whether data is vulnerable does not quantify the effect of an error. This is due to not taking into account application-specific effects such as the propagation or significance of an error. Instead, it captures the effects of memory access patterns and indicates the probability of consuming a fault, thus giving an upper bound on the failure probability. By being independent of application-level information, vulnerability can be used at the runtime level, for any workload.

6.2.2 Memory Vulnerability at the CPU level

The vulnerability metric we use is FEA, introduced in Section 5.2.3, as it is shown in Section 5.3 that this metric is the one that correlates best with failure probabilities, and also displays most disparities between different data structures. This makes it the thus the best metric to identify opportunities for adjusting ECC. We will refer to FEA simply as "vulnerability" in this chapter, which is thus for each memory location to *the fraction of time it contains data that will be consumed.* In Figure 6.1, the vulnerability of a memory location is displayed visually, as the portion of the timeline coloured in solid red. The arrows represent the transfers of data for a particular memory location over time, between memory and the CPU.

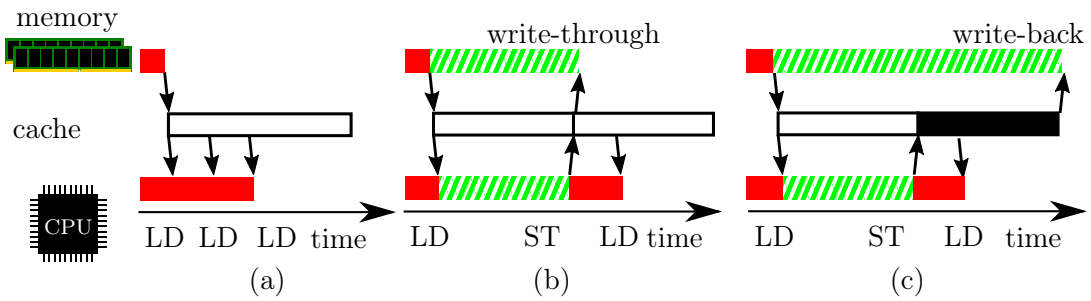


Figure 6.3: Difference between vulnerability at memory level, and the value computed from instructions, which always has higher vulnerability.
At memory and CPU level we represent in hatched green the safe time, and in solid red the vulnerable time. The boxes at cache level represent the time that the 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.

This figure shows how the duration before a memory access is categorised as safe or vulnerable, depending on whether the access is a store or a load.

6.2.3 Difference between CPU and Memory Vulnerability

We call CPU-level vulnerability the vulnerability computed using load and store instructions executed instead of requests reaching memory. The vulnerability of data differs at CPU and memory level, as caches serve part of the requests emitted by the CPU. However, the CPU-level vulnerability can be safely used, as it gives us a conservative estimate of the vulnerability at the memory level.

Indeed, the vulnerability metric at the CPU level is always higher than at the memory level. This is illustrated in Figure 6.3, which displays the vulnerability at memory and CPU levels, and the time data spends in cache, based on loads and stores. A cache line that is not modified, as in Figure 6.3 (a), is rated vulnerable for a longer period than it is vulnerable in memory. Similarly, a modified line with a write-through policy is safe until the last store, and any subsequent accesses make this cache line seem more vulnerable than it is in memory, as depicted in Figure 6.3 (b). Finally, Figure 6.3 (c) illustrates how a write-back policy delays the stores to a cache line until its eviction from last-level cache, which increases the safe duration in memory when compared to the CPU level. 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.

6.3 Dynamic Estimation of Vulnerability

In order to use the vulnerability metric during program execution, we need to estimate the vulnerability in real time. In this section, we present a generic methodology that allows to perform this vulnerability estimation on existing hardware, by relying only on the instrumentation tools built in the hardware, runtime and Operating System (OS). This online vulnerability analysis can be used offline to inform the programmer on which structures to protect with algorithmic techniques, for example. Our evaluation uses the online vulnerability estimation to guide ECC strength dynamically.

We identify *memory regions*, which are contiguous ranges in memory that are allocated or used together, from runtime and OS library calls. The Performance Monitoring Unit's (PMU) random sampling capabilities, which are available on many different architectures [IBM 2015a; Intel 2017], allow us to record load and store addresses and their respective timestamps. We then combine all this information to estimate the vulnerability of each memory region, thus the fraction of time that a memory region contained data that was loaded. While PMUs advertise their sampling capabilities as being "random" sampling of instructions, their documentation does not specify the criteria of randomness and therefore guarantee the fairness of this selection. In Section 6.5.1, we characterise the distribution of obtainable samples on the hardware that we use.

Organisation of Memory Access Sampling. To control the overhead of sampling memory instructions, we organise it in phases. Outside sampling phases the sampling process is completely disabled, and our methodology incurs no overhead. During sampling phases, we only record a fraction of the samples that are generated. There are thus three parameters that control this mechanism: the duration of sampling phases, the delay between phases, and the number of skipped samples for every recorded sample.

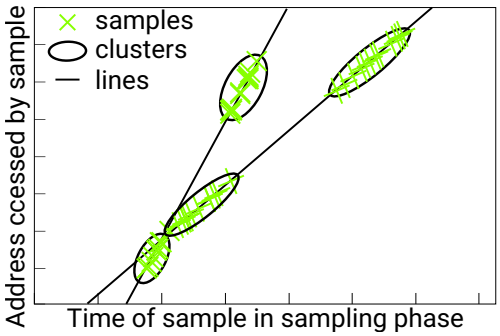


Figure 6.4: Synthetic example of the KHT algorithm, showing samples, 4 clusters and 2 identified lines, for a single sampling phase and single memory region.

6.3.1 Identify Memory Access Patterns

The set of recorded sampled instructions is a subset of the instructions that were executed. To compute the vulnerability of a memory region from the samples, we need to extrapolate memory access patterns from the recorded data. When analysing the recorded memory accesses, we classify them by thread, memory region, and whether they are loads or stores. Plotting samples reveals the streaming memory access patterns which appear as aligned points, in the 2D space with the accessed addresses inside the memory region on the y-axis and time on the x-axis, as represented in Figure 6.4. We identify aligned points, and thus streaming memory patterns, by using a kernel-based Hough transform voting algorithm [Fernandes and Oliveira 2008].

The key idea of the Hough transform to detect aligned points is to consider potential lines with different slopes for each point, and to compute how many points belong to each potential line. This voting scheme's complexity is linear with the number of samples as well as with the number of considered slopes. The kernel-based algorithm is optimised for real-time performance: instead of considering individual points, it operates on clusters of approximately collinear points [Fernandes and Oliveira 2008]. Votes are cast only for slopes close to that of the best-fitting line of each cluster, and according to a model of the uncertainty from associating the best-fitting line to the cluster. We represent 4 clusters in Figure 6.4, and the votes for the lines from all clusters identify 2 lines in total from the recorded samples.

Clusters are identified by recursively subdividing strings of points, until the maximum deviation inside each cluster is sufficiently low. The lower half of Fig-

ure 6.4 shows one cluster subdivided in two at the sample with the maximum deviation. We use the algorithm's implementation available online [Fernandes 2008].

We extrapolate lines identified by this algorithm as access patterns that will stream linearly the whole memory region. This allows to estimate the vulnerability of addresses in memory regions for which no instructions were recorded. To extrapolate, we extend any identified line, using a linear interpolation, until it spans the whole considered memory region. Thus a stream in a given region can be summed up by three parameters: whether it consists of loads or stores, the time at which it accesses the lowest address in the region, and the time at which it accesses the highest. We then combine for each region all the identified streams, from all threads and all types (load or store), and compute the average vulnerability, which is the fraction of the time after any stream and before a stream of loads.

By supposing data access patterns repeat over time, we can use this estimated vulnerability value to predict the future vulnerability of each memory region. This supposition is generally accepted for HPC workloads, which often consist of scientific codes such as iterative solvers, or large-scale simulations of physical phenomena. In fact, it is sufficient that data has the same sort of use over time, and thus approximately the same vulnerability level, even though the access pattern does not repeat. Furthermore, changes in behaviour during an application's execution time can be accounted for by a new sampling phase to adjust the vulnerability rating accordingly.

6.4 WITSEC Adaptable ECC

The memory vulnerability can be used to adjust the ECC strength dynamically. To enable this, we present an ECC scheme that allows extending a uniform N-flips ECC with N+1 correction capability for selected pages at runtime. It is worth mentioning that our vulnerability approach remains a general methodology, which can be used to guide an offline analysis useful for an application programmer, or any other runtime-adaptable ECC scheme on existing platforms.

6.4.1 Different ECC Strengths

We consider three ECC strengths, which are perfect binary codes with even Hamming distances: Single Error Correct (SEC), Double Error Correct (DEC), and Triple Error Correct (TEC). We also consider using no ECC, thus Zero Error Correct (ZEC). While codes with uneven Hamming distances are commonly used, as they allow to detect one more bit flip than they can correct, DUE-less codes are also used [Abdoo and Cabello 1996]. DUEs are reported even for benign and false errors, which have no impact if ignored. Thus, using an ECC with DUE artificially causes much higher failure rates, due to faults in overwritten or otherwise unimportant data. The vulnerability analysis can be done taking DUE into account, however this requires tracking and ignoring these false errors [Jaulmes *et al.* 2019c]. Thus, for the sake of simplicity, we consider codes that can cause only corrected or silent errors.

The most commonly used ECC is a SECDED that encodes 64 bits of data on 72 bits using a Hamming (127, 120) code truncated to (71, 64) with an additional parity bit [Hamming 1950], thus with 8 redundant bits. We use as SEC code the same truncated Hamming (71, 64) code without the parity bit, that is, our SEC code requires 7 bits of redundancy per 64-bit word. To work at the same granularity for all ECC levels, we choose Bose–Chaudhuri–Hocquenghem (BCH) codes [Hocquenghem 1959] for DEC and TEC protection: a BCH (127, 113) code with 14 bits of redundancy and a BCH (127, 106) with 21 redundant bits, respectively truncated to (78, 64) and (85, 64). Thus, every supplementary ECC level requires 7 more bits per 64 bit word.

6.4.2 WITSEC ECC Organization

WITSEC Is Targeted Stronger Error Correction (WITSEC). We choose an approach that uses a uniform ECC correcting N bit flips as a baseline, and extends protection for some memory regions identified as more vulnerable at runtime.

This choice is motivated by the fact that saving and reallocating redundancy is not efficient. Indeed, diminishing the redundancy for some region implies exposing the program to faults with lesser multiplicity, which have a much higher rate. Instead, we only ever increase redundancy using addressable memory as supplementary ECC storage.

Thus, we use as a baseline a uniform ECC with its redundancy bits stored in a widened data path, as is done currently with SECDED. We then extend protection for some memory regions identified as more vulnerable at runtime, by protecting those with an ECC correcting N+1 flips. In order to do so, we reuse the bits from the baseline as well as additional redundancy in decoupled storage. As these additional bits are part of N+1-ECC codewords, they require no ECC encoding of their own, and can be stored using the full baseline's data + ECC width. That is, for a SEC baseline of 64 data bits + 7 redundancy bits, the ECC can use 71 bits to store supplementary ECC bits instead of 64.

Supplementary bits are organised in blocks, one per page with extended protection, addressing bits in-block with the word position in-page. This organisation allows a large amount of flexibility and only requires 448B of storage to provide extended protection for a full 4kB page, as extending protection means storing 7 supplementary bits per word. (7bits/64bit word $\times$ 4kB page = 448B.) For each page, it is sufficient to store its ECC strength, and the address of the decoupled ECC storage. The OS and runtime are in charge of allocating and maintaining the mapping to this decoupled storage. Updating ECC strength, to increase or decrease the protection, requires re-encoding the ECC and can be done whenever data is next updated, or next accessed for data identified as read-only.

6.4.3 WITSEC-aware Memory Controller

Figure 6.4.2 presents a memory controller extended with the structures in grey to support WITSEC. We show extended-protection codewords and requests to data under extended protection in shades of green, and show baseline protection in white. Any additional requests required to fetch supplementary ECC blocks are also shown in grey.

Extended Protection Page to ECC Block Mapping. This structure stores the physical addresses of extended-protection pages and corresponding supplementary ECC blocks. Each entry in this structure stores the physical address of a page, and the physical address of an ECC Block. These addresses are updated by the runtime through a memory-mapped register, with memory it allocated to that effect. Each entry requires $2 \cdot log_2(\text{adressable memory}) - log_2(\text{page size})$ bits, e.g. 62 bits for a 128GB DRAM channel and 4kB pages. For each incoming request, the memory controller looks up the ECC level of the corresponding page, marked

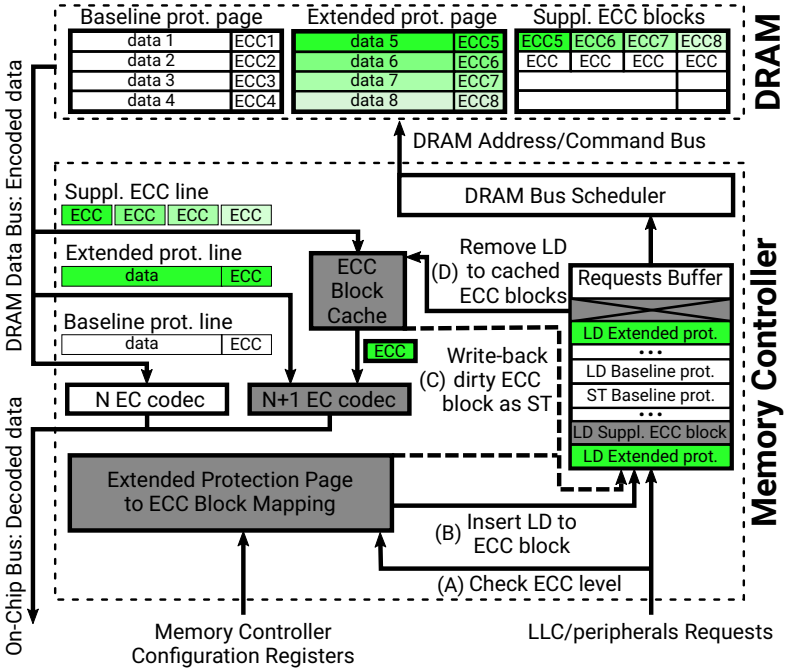


Figure 6.5: DRAM and runtime-aware controller implementing WITSEC

ECC-encoded data is stored in DRAM, with *Baseline protection* codewords numbered 1–4, and *Extended protection* numbered 5–8 and coloured in shades of green, with supplementary bits in decoupled locations in memory. The memory controller respectively passes these codewords to the N-Error Correct (N-EC) and (N+1)-EC encoders and decoders (codecs). In grey, added structures required to support WITSEC. All solid arrows (marked A, B, and D) represent new data and control flow for loads. For stores the path is exactly the same in opposite direction, except supplementary ECC blocks are not loaded but written back, as depicted by the dashed arrow (marked C).

(A) in Figure 6.4.2, and chooses the right codec to use. The memory controller also inserts load requests to corresponding supplementary ECC blocks for extended-protection loads, marked (B) in Figure 6.4.2.

N+1 EC codec. An extended-ECC codec is required to decode the N+1-ECC codewords. Low-power low-latency decoding strategies exist for multi-bit ECC, for example Fougstedt et al. report less than 0.06pJ/bit, and 8.7ns to decode DEC BCH codes with 511 bits block size, synthesised on 28nm [Fougstedt *et al.* 2017], which are small overheads compared to DRAM access delays.

ECC Block Cache. To reduce extra loads and stores to supplementary ECC blocks, we further extend the memory controller with a write-back cache. This cache serves part of the loads to supplementary ECC blocks as illustrated with the arrow (D) in Figure 6.4.2, and coalesces stores. The eviction of blocks simply

consists in inserting a store to the corresponding supplementary ECC block in the requests buffer (marked (C) on Figure 6.4.2). Furthermore, extended ECC blocks can be written to memory using partial (or masked) writes. Such writes have an input mask specifying parts of data that should be ignored in a write command [JEDEC 2013]. Thus, extended ECC blocks can be written to DRAM without needing to first fetch other neighbouring data in memory.

6.4.4 Discussion of Hardware Design Decisions

Due to the decorrelated storage architecture for supplementary ECC redundancy, a single memory access may need to access two separate access location. This in turn may have an impact on performance. However, the alternative way of storing an adjustable amount of redundancy is to store data and ECC next to each other, which has several drawbacks. First, this requires that the data be fully moved in memory to adjust the amount of interleaved redundancy every time that the ECC strength is adjusted. However repeatedly reading writing from a single DRAM row is slow as it incurs delays to switch between reading and writing. Thus, a better implementation of this adjustable would require moving the data from one physical location to another, which can already be implemented using two DRAM DIMMs with different reliability characteristics. Second, this organisation modifies the alignment of data and makes accessing it much more complicated, in particular by having physical pages with various possible sizes, thus potentially different from the virtual pages that translate to them.

For the most efficient implementation of WITSEC, accesses to data and supplementary ECC should be paralleliseable. To minimise the energy the energy spent on those two accesses and the number of DRAM rows thrashed by these paired accesses, the granularity at which this parallelism is offered should be as small as possible. For example, it is preferable to activate two banks rather than two ranks. Hardware constructors have now adopted Sub-Array Level Parallelism [Kang *et al.* 2014], thus whenever possible supplementary ECC blocks should be allocated in sub-arrays inside the same bank as the data they cover. As supplementary ECC bits are distributed in memory in a much denser way than the data they protect (7 bits per ECC word, vs. 64, 71 or 78 for data and baseline ECC), a much larger of number of neighbour accesses will map to a single DRAM row. Thus, the number of row misses will be much smaller.

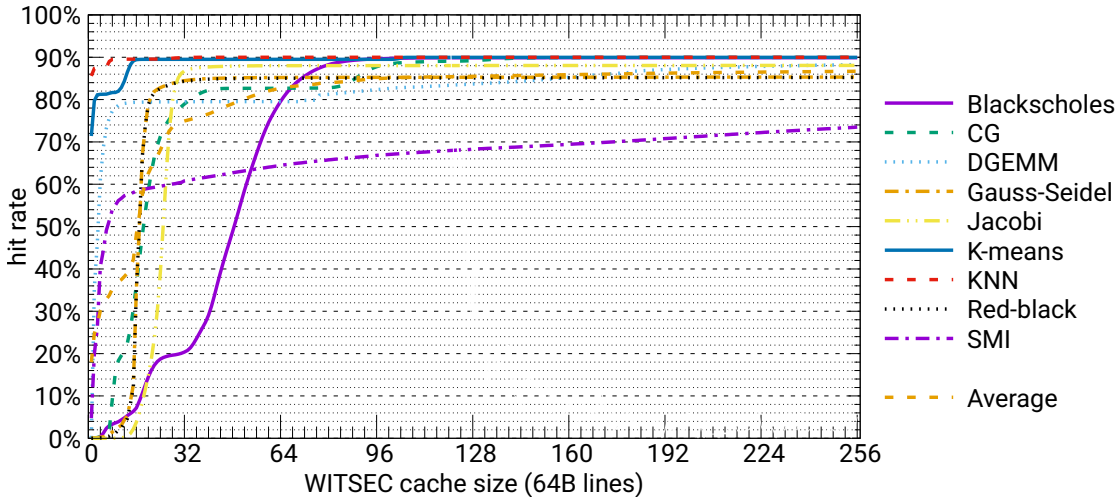


Figure 6.6: Impact of WITSEC cache size on its miss-rate

In particular, all the row hits for DRAM load will be row hits for the associated supplementary ECC bits with very high probability in the case of an open-row policy, while the latency of a row miss will be increased for a closed-row policy, though only by a fraction of this latency due to the Sub-Array or bank parallelism.

The cache size also will have an impact on minimising the additional latency due to supplementary requests. The trade-off will be between the amount of power that maintaining the cache costs and the delay incurred by requesting supplementary ECC blocks. Supposing all of the data structures of a benchmark need to be tracked we are able to measure the worst-case efficiency of the cache. Using the TaskSim setup displayed in Section 3.1, we compute at every memory access the reuse distance From this reuse distance and a given cache size, we can deduce whether the access would have been a hit or a miss in an LRU cache.

We present these results in Figure 6.4.4. The hit rates all tend to an upper bound which correspond to the mandatory misses, without any prefetching or similar optimisations. All benchmarks except Blackscholes and SMI have steep increases in their hit rates for less than 32 cache lines, that is 128B of cache memory. Blackscholes reaches a 90% hit rate in the WITSEC cache for about 80 cache lines, while SMI has a hit rate that remains below 70% until 160 lines but is already above 60% at a size of 24 cache lines.

For all of these benchmarks, the amount of memory for which supplementary redundancy needs to be tracked will be much less than what is measured here. This is because WITSEC aims at targeting only part of the memory footprint for

extended protection, not the entirety of it, thus any realistic scenario will have lower reuse distances and thus higher hit rates. From these results it seems that a cache size containing 64 lines of memory, thus only 512B, should be a good trade-off to maximise the cache's efficiency while keeping it small and fast.

6.4.5 Related Variable Strength ECC schemes

Two main adaptable ECC schemes for DRAM could be guided by our memory vulnerability estimation methodology, and be used as stand-ins for WITSEC, provided they are extended to allow runtime-adaptability.

The main differences with Virtualized ECC [Yoon and Erez 2010] is that it uses a first tier ECC code, mainly to detect errors, and the second tier codes in the case of uncorrected errors in the first level. WITSEC on the opposite always has a single tier of ECC, that can be of one of two different strengths, that is always accessed fully. The cache in the memory controller allows to mitigate the performance impact of accessing the supplementary ECC blocks, and is similar to what Lin *et al.* [2012] propose to store meta-data of ECPs. This means in particular that the additional redundancy required is much smaller for WITSEC than for Virtualized ECC.

Similarly to Virtualized ECC, Odd-ECC [Malek *et al.* 2017] uses two tiers of error correction, with a pre-defined arrangement in memory at a 256KB granularity rather than allowing the fully flexible page-table level mapping used in Virtualized ECC. The drawbacks of this organisation with data and ECC tiers located next to each other are discussed in the Setcion [**ssec:design-impact**].

6.5 Experimental Setup

To evaluate our general methodology on a real system, we choose one of its possible uses: to adapt WITSEC ECC protection dynamically. We implement on a POWER system the necessary tools to recommend which portions of memory to protect in priority. In this section, we present the experimental setup used for the implementation of the dynamic vulnerability estimates using sampling. The error injection methodology is explained in Section 3.1, and the whole evaluation is performed with the benchmarks presented in Section 3.3.

6.5.1 Online Tool Experimental Framework

We evaluate the solution on an IBM POWER8 based system (PowerNV 8335-GTB model) [Sinharoy *et al.* 2015]. The system has 256GB of DDR3 CDIMM memory running at 2.4GHz. The POWER8 processor in this system has 10 cores running at 4.00GHz, each has 64kB L1 data and 32kB L1 instruction caches, a 512kB L2 cache and an 8MB L3 cache. The system runs a Red Hat Enterprise Linux Server 7.3 OS with the kernel version 3.10.0. We compile all the benchmarks with GCC 7.1.0. Our implementation's source code is available online [Jaulmes 2018].

Recording Memory Accesses. We setup the PMU to count *marked* loads and stores, and to trigger an Event-Based Branch (EBB) on counter overflow [IBM 2015b]. *Marked* instructions are instructions that are randomly selected by the hardware and populate Special Purpose Registers (SPRs) with information such as the type of instruction and the virtual address accessed. An EBB is an interrupt in user space, and we use a hand-written assembly routine, summarised in Figure 6.7, to record the content of the relevant SPRs. Once setup, the sampling can be enabled and disabled by flipping bits in SPRs directly, thus without requiring a system call. According to the ISA specification, no EBB can be triggered during another EBB [IBM 2015a].

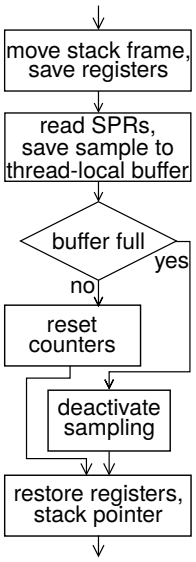


Figure 6.7: EBB handler

The PMU only selects random loads and stores as marked instructions for a short time every 16ms, as shown in Figure 6.8. We call this a *sampling burst*, which is entirely controlled by the hardware, as opposed to a *sampling phase*, which is the time during which we enable sampling, as explained in Section 6.3. This means that memory access patterns lasting less than 16ms will most probably not appear in any recorded samples.

Sampling phases are started and stopped periodically, by setting a timer to deliver a SIGALRM signal. Since the PMU configurations are thread-local, the sampling enabling and disabling instructions need to be run in every thread, which we manage by broadcasting the same signal to the other threads. The thread awoken by the timer then waits for all other threads to toggle their sampling

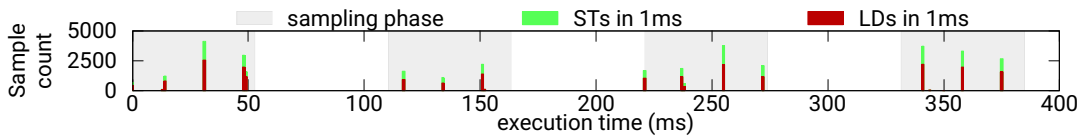


Figure 6.8: Number of recorded samples per millisecond over time, summed over all threads, running using 8 threads

Samples are only recorded during sampling phases, which are set to last a little over 50ms in this example. However, the hardware random sampling mechanism functions in bursts, and only marks a few thousand instructions during less than 1ms and every 16ms on average.

configuration, and also performs the analysis of the samples recorded during a sampling phase at the end of that phase. Vulnerability values are computed per memory region, as described in Section 6.3.1. We discretise the vulnerability rating into 101 categories, of the form $\forall t \in [0..100]\ \frac{t}{100} \leq V < \frac{t+1}{100}$. The last category is thus the single value $V = 100\%$.

6.6 Evaluation

In this section, we first report the overheads of our online vulnerability estimates, and then study the capabilities of our methodology to adjust ECC strength dynamically with WITSEC. Reliability gains and trade-offs against storage overhead can be achieved by extending the ECC protection of the most vulnerable pages, for various parallel benchmarks.

We have validated the values reported by the online vulnerability methodology against accurately simulated values of vulnerability, which are presented in previous work [Jaulmes *et al.* 2019c]. The comparison of per-page averages demonstrates the estimation methodology is accurate, as will be highlighted by the overall evaluation in Section 6.6.3. Furthermore, the vulnerability estimation methodology tends to conservatively over-estimate vulnerability ratings whenever it lacks accuracy.

6.6.1 Overheads

The cost of estimating vulnerability online is two-fold: the cost of recording samples, and that of analysing them, thus estimating the vulnerability. We measure the overhead of sampling at 0.86µs/sample, using CG on a single thread, sampling continuously, and having disabled vulnerability estimation.

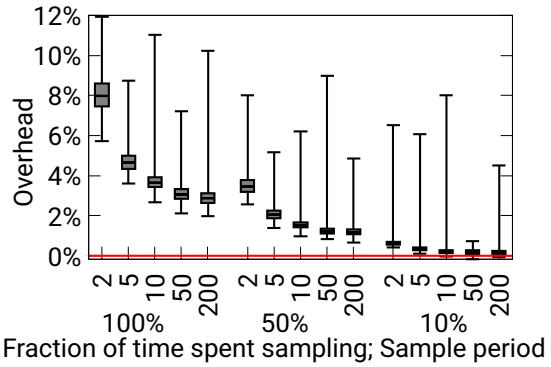


Figure 6.9: Evolution of total overhead, varying the sampling period and phases

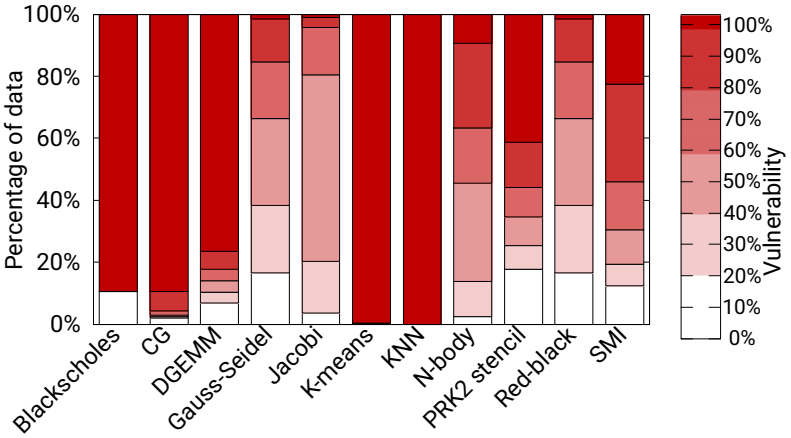


Figure 6.10: Distribution of vulnerability ratings across each benchmark's memory

The total overhead of estimating vulnerability online, sampling and analysis together, is presented in Figure 6.9. Average overheads range from 8.6% down to 3.1% with no delay between sampling phases, with the maximum measured at 11.9%. When maintaining sampling phase durations of 500ms and increasing the delay between sampling phases, overheads decrease close to linearly. Overheads range from 3.8% down to 1.3% when sampling 50% of the time, thus waiting 500ms between phases. When sampling 10% of the time, which corresponds to waiting 4.5s between sampling phases, overheads become very low, with 75% of overhead values below 0.5% as soon as we sample every 5 or less marked instructions. Sampling only 50% of the time does not affect the results' quality by much, however using only 10% hides some accesses to some data structures altogether due to the bursty behaviour of the POWER8 PMU.

For the remainder of our work, we will maximise the precision of our technique by using a sampling period of 2 enabled 50% of the time. It is worth noting that

a sampling period of 2 means recording half of the randomly selected instructions, which is only a small subset of the total instructions in the program. To achieve 50% of time with sampling enabled, we schedule sampling phases lasting 500ms, and the same delay in-between sampling phases. This means the majority of overheads of the chosen configuration range between 3.2% and 3.8% of the execution time, as presented on Figure 6.9.

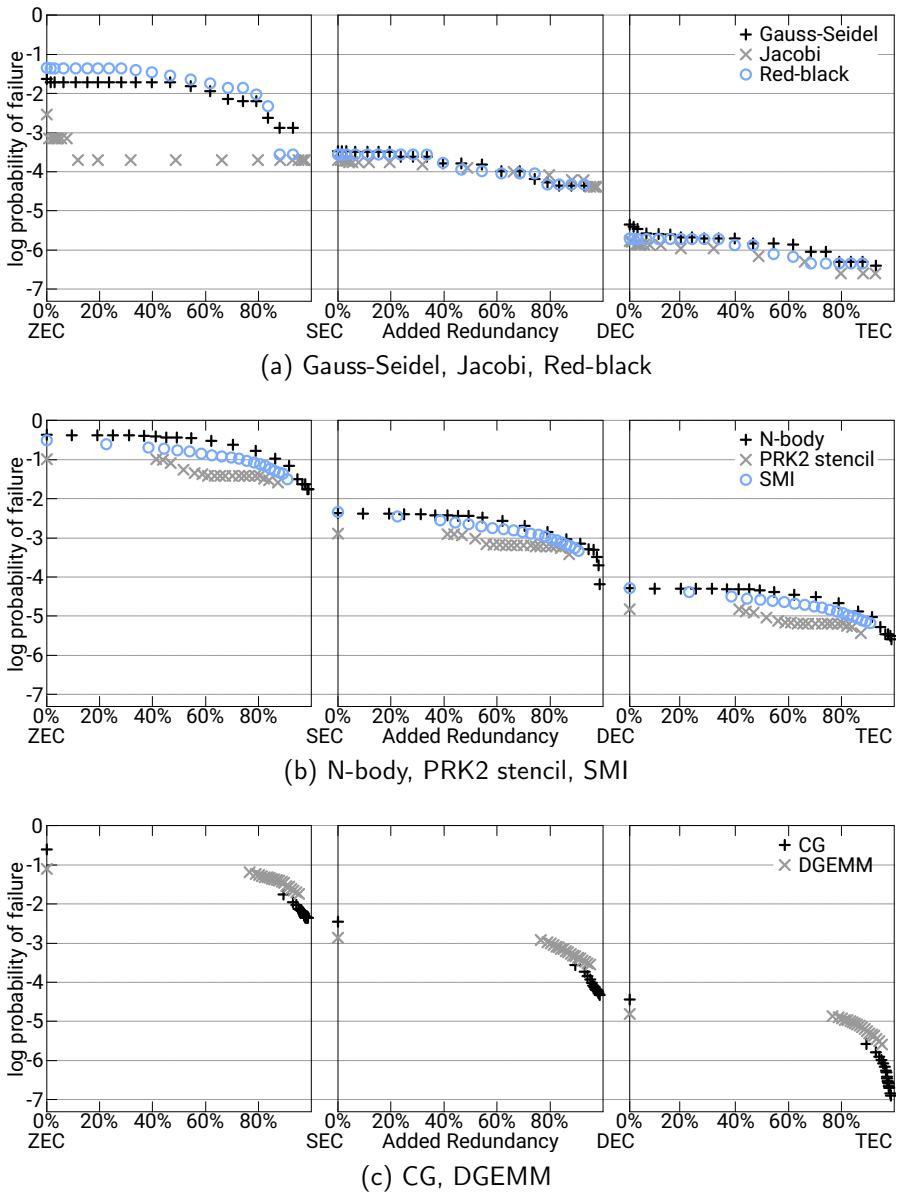
6.6.2 Distribution of Data per Vulnerability Level

In order to measure the average vulnerability ratings from the online implementation of our methodology, we record, for each benchmark, the vulnerability ratings for each identified memory region and at each sampling phase over 100 runs. The breakdown of the average distribution of vulnerability across memory is presented in Figure 6.10. We only represent 6 coarse categories of vulnerability, for the sake of readability. The full results contain 101 categories, as explained in Section 6.4.1. The last category is the single value $V = 100\%$, and the other categories displayed have the same 20% size.

Figure 6.10 reveals that the considered benchmarks represent a varied set of workloads in terms of vulnerability. The machine learning benchmarks, K-means and K-nearest neighbours, use training data sets which are only ever read, and are orders of magnitude bigger than any other data structure, thus showing a 100% vulnerable memory footprint. Some benchmarks are very imbalanced, such as Blackscholes, DGEMM, or CG, which has close to 90% of its data always rated 100% vulnerable, due to the matrix being an overwhelming portion of CG's data and being only ever accessed by loads. SMI and PRK2 stencil are more balanced, though most of their data set is still vulnerable. On the other hand, N-body, Gauss-Seidel and Red-black seem to have different vulnerability ratings distributed rather uniformly across their memory. Finally, the Jacobi has most of its data non-vulnerable, which is due to it having two copies of the problem space, and alternating their uses at each iteration, hence having always half of its data being safe at any given time.

This information can already be useful as-is, for an application programmer to choose which regions to protect with algorithmic techniques, or to perform data placement choices on devices with different reliability characteristics. Our further

evaluation uses this varied vulnerability to choose memory regions for increased ECC protection.



(a) Gauss-Seidel, Jacobi, Red-black

(b) N-body, PRK2 stencil, SMI

(c) CG, DGEMM

Figure 6.11: WITSEC trade-offs between redundancy (i.e. ECC storage) and reliability, in 5% vulnerability increments.

Uniform ECC levels are drawn as vertical lines, and each point is an available reliability-overhead trade-off. For each benchmark, going from one given point to the next one to the right means adding extended protection to data with the next vulnerability level: the left-most points per box are uniform ECCs, with no extended protection. The next ones extend protection for data with $V = 100\%$, the next ones $V \geq 95\%$, then 90%, etc.

6.6.3 Evaluation of Dynamically Guiding WITSEC ECC

In order to drive dynamic ECC decisions, we use the WITSEC scheme, introduced in Section 6.4. To select which pages to protect with stronger ECC, we set a threshold on vulnerability ratings. Pages with vulnerability ratings above this threshold are then selected for extended protection. For example, we may use a baseline of ZEC (thus no ECC) and a threshold of $V \geq 90\%$, above which data will be protected with SEC. For all of the possible ECC protection scenarios that WITSEC can offer, we compute the amount of redundancy needed, and the probability of suffering a failure (*hang*, *wrong* or *crash*). The amount of redundancy is the storage required for the selected ECC configuration.

We evaluate each benchmark for each threshold from 0% to 100% and display our results on Figure 6.11. The figure is organised in three sub-figures, grouping benchmarks together by their vulnerability repartition as identified in Section 6.6.2. On the y axis we depict the logarithm of the failure probability in the event of a fault. On the x axis we display the redundancy required for each configuration, normalised to the amount of redundancy needed to uniformly apply SEC. As protection levels are linear in redundancy (respectively 7, 14 and 21 bits per word for SEC, DEC and TEC, see Section 6.4.1), the amount of redundancy needed for DEC and TEC are respectively exactly twice and three times that of SEC. Each point corresponds to a different threshold, with points on the solid vertical lines on the left-hand side of the boxes being the uniform baselines, where no data has extended protection.

The first thing to note is that our methodology allows for a wide range of trade-offs, offering intermediate levels of protection by extending protection to a part of the memory footprint. Jacobi, Red-black, and Gauss-Seidel in particular offer trade-offs with reductions of failure probability early on. Results for benchmarks with more imbalanced vulnerability distribution require more protection before reducing failure probability significantly, as demonstrated by SMI and N-body, and in a lesser measure PRK2 stencil. Finally, the benchmarks that have an important part of their data rated $V = 100\%$ (see Figure 6.10), such as CG and DGEMM, do not offer many possibilities for redundancy trade-offs.

The available trade-offs for Gauss-Seidel, Jacobi and Red-black are shown on Figure 6.11a. These benchmarks have their vulnerability evenly distributed

amongst data structures, which causes the trade-offs to be rather evenly distributed along the x axis. The main feature of those benchmarks however, is the impact of protecting data rated 100% vulnerable. In these configurations, WITSEC achieves steep reductions in terms of failure probability with very small increases in redundancy. For example, Jacobi gets its failure probability reduced by 4× when protecting its most-vulnerable data with SEC against a ZEC baseline, while Gauss-Seidel achieves a 22% reduction. This behaviour corresponds to protecting the boundary conditions of the heat diffusion domain. In general, injecting errors in this part of the data is significantly more likely to cause a wrong output than in any other part, where the main outcome is a slower convergence. Furthermore, this sensitive and small section of memory is well identified by our methodology, which enables increasing the protection of this region to achieve significant vulnerability improvements at a very small extra cost with respect to configurations applying ECC uniformly.

Looking at benchmarks with more imbalanced vulnerability distribution, such as N-body, SMI and PRK2 stencil, we see on Figure 6.11b that the redundancy levels which correspond to extending protection only for $V = 100\%$ are around 41% for PRK2 stencil, 22% for SMI and 9% for N-body. This still allows for a wide range of trade-offs, adding more redundancy to gain better failure rates. If given a reliability target of a failure probability of 10^{-3} for PRK2 stencil, we can apply a SEC baseline, protecting with DEC data rated with $V \geq 85\%$, which corresponds to 51% of supplementary redundancy and a failure probability of 9.4×10^{-4}. This still corresponds to saving 49% of redundancy in memory when compared to the next uniform ECC achieving a failure rate below 10^{-3}, which is DEC. This demonstrates that only part of all the data requires stronger protection to reach reliability goals that are in-between uniform ECC levels of reliability.

For DGEMM and CG, trade-offs shown on Figure 6.11c are available starting around 80% and 90% supplementary ECC overhead respectively, with reductions in error rates around 14× for CG, and 1.2× for DGEMM. These results are driven by the fact that most of the data is rated with a vulnerability of 100%, which causes a big gap between the baseline configuration and the next available trade-off. Extending protection to less vulnerable data then reduces the failure rate further.

Overall, a wide range of possibilities is opened for trade-offs between error protection and ECC overhead, while always maintaining a low overhead. For any reliability threshold, we can find the best configuration to achieve dynamically that level of protection with the minimum overhead. Conversely, if a power budget limits the amount of redundancy, online vulnerability ratings identify dynamically which data to protect more in order to minimise the failure probability.

6.7 Conclusion

In this chapter we present a methodology to estimate memory vulnerability online and WITSEC, a dynamically adaptable ECC scheme to adjust memory protection accordingly. With low execution time overhead – 3.47% on average – and functioning on real hardware, our methodology allows to explore trade-offs between ECC overhead and protection, by efficiently and precisely targeting the most vulnerable parts of memory for increased protection. Given a reliability target of how unlikely we want our application to fail in the event of a fault, we can protect only memory rated more vulnerable than a threshold, to reach the desired failure probability with minimal overhead – much smaller than that of a stronger ECC applied uniformly. This is possible by applying ECC with a much finer granularity, targeting data that requires it most. Most importantly, our method never decreases ECC protection, which could present a risk in case of mis-estimating vulnerability, instead only increasing ECC protection in a smart way.

This methodology can be used for other purposes, such as guiding data placement or informing a programmer which data structures to protect with algorithmic techniques. The methodology could further be extended in several ways, for example to support DUE or hard errors. Furthermore, exploring more applications would allow to confront our results to more varied and more complicated access patterns, and to applications whose behaviour changes over time. Gains in memory overhead for redundancy should translate rather directly to energy savings, though we would require an implementation of WITSEC, our adaptable ECC scheme, to precisely measure this effect. Future work includes measuring the energy and performance of a hardware WITSEC implementation, to compare it against hardware integrated ECC, and exploring different error models, e.g. from different technologies. For example, if errors of multiple bits were to be as probable as single bit

errors, it could be interesting to combine ZEC directly with DEC or TEC rather than only SEC, while using related work such as VS-ECC instead of WITSEC could be more adapted to target burst error models.

Finally, it is clear that strong PMU capabilities enable powerful runtime tools. A runtime-aware architecture needs to facilitate dynamic cross-level optimisations. For this, such an architecture needs to disclose information such as memory access patterns to the higher levels of the hardware-software abstraction stack. At the very least, runtime optimisations such as the one proposed in this chapter would benefit hugely from more control on how instructions are sampled.

Chapter 7

Conclusions

The work presented in this thesis has demonstrated various ways in which runtime systems can be used to exploit redundancy efficiently, tackling problems caused by DRAM soft error rates, and thus improving the reliability of a computing system. This chapter summarises the main conclusions from these contributions, and presents some lines of future research that this work has opened. It then lists the publications that are an outcome of this thesis, as well as financial and technical acknowledgements.

7.1 Conclusions

7.1.1 Overlapping Algorithmic Recoveries

At the algorithmic level, the runtime system has allowed to mask the overhead of recovering from an error. Using a real-world DUE fault model, the retirement of memory pages by the OS due to a DUE in memory, we have presented an algorithmic recovery that is both an exact and a forward recovery, FEIR. This algorithmic recovery relies on straightforward redundancy relations identified between data structures of an algorithm, for any Krylov subspace solver. Such invariants can be easily derived, and are often used in related work as detectors for SDC. Compared to the state-of-the-art, the properties of the recovery guarantee a better convergence of the protected solver, and thus a better mean time to solution. This better convergence is achieved by recovering lost data exactly, as opposed to restart-based techniques that constitute the state-of-the-art of algorithm-specific recoveries. We further preserve all computations done until the error is detected, as opposed to

backwards error recoveries such as checkpointing, which are the state-of-the-art for more general techniques.

While the proposed recovery mechanism FEIR outperforms the state-of-the-art in terms of time to solution, this technique still suffers from an overhead for recovering lost data. Using the expressiveness of the OmpSs task-based programming model, and leveraging its runtime scheduler, we encapsulate the recovery computations in a task, and run this recovery asynchronously. Without further programmer intervention, the runtime system overlaps the recovery tasks with the solver's computations by exploiting load imbalance that exists in the solver. This technique, AFEIR, outperforms FEIR by offering even lower average times to solution for all except the highest simulated fault rates. The overhead of both techniques grows with the size of memory pages, demonstrating that for large memory pages, a finer grain hardware retirement technique implemented at the OS level would be beneficial.

Future avenues of research opened by this work include extrapolating redundancy relations from invariants, to build algorithmic recoveries for other algorithms. This will expand the use of low-overhead recoveries exploiting DUE. Potential invariants have already been identified in QR, LU, and matrix multiplication [Huang and Abraham 1984; Davies and Chen 2013; Heroux *et al.* 2005].

This contribution underlines the potential of algorithmic-based resilience. Programming models could be extended to support algorithmic recoveries out of the box, by providing a way to express how to recover lost data. Instantiating recovery tasks only when errors are detected would remove the last small overhead incurred by algorithmic recoveries. Furthermore, providing a unifying framework to express both checking the validity and recovering corrupted data would enable more wide-spread use of algorithmic resilience techniques. Such a programming model extension has been proposed to perform checksum verification and checkpointing of task inputs and outputs [Subasi *et al.* 2015]. Beyond allowing the programmer to express resilience mechanisms easily, the programming models' runtime system should ensure that the control flow of the application remains valid, by recovering from any potential crash, identifying lost or corrupted data, and triggering the appropriate recoveries.

Finally, as the application can recover from memory page retirement at a very high rate, more risk can be taken with data placement in memory, using devices

with lower fault rates or checkpointing only for the memory that can't be recovered algorithmically. Alternately, a new strategy against SDC becomes available. The application could diminish drastically its probability of silent errors by treating corrected errors as DUE. By doing so, the probability of SDC due to an ECC miscorrection is avoided. This approach allows the error coverage for multiple bit flips to be drastically increased.

7.1.2 Identifying Memory Vulnerability

At the runtime level, we identify and quantify the risk of suffering from an error in data stored in memory by introducing a new metric, FEA. This metric is an upper bound on error rates, tighter and correlating much better with error rates than other metrics. Thus the FEA metric is useful to guide many reliability-based decisions. Furthermore, an important insight was gained from the metric: false DUE should be ignored in an error analysis of memory. This insight has lead to a simple OS-level proposal for delaying the reporting of memory DUE. This proposal allows to reduce failure rates of applications due to DUE in memory, with drastic reductions (close to 50%) for a number of algorithms that can not compute values in-place. Such algorithmic restrictions create memory access patterns overwriting data in memory, and thus data that is stored in memory that is in fact inconsequential.

Two of the research lines made possible by this work already have been or are in the process of being investigated. The first of these research direction leverages the powerful correlation between the FEA metric and the vulnerability of data – that is the probability of an application failure due to an error at a given bit. The work done on this topic is presented in the last chapter of this thesis.

Another research direction born from the work on the FEA metric exploits the knowledge of inconsequential data stored in memory, as that data does not need to be refreshed. Indeed, refreshing data in DRAM is costly both in power and in memory bandwidth. These problems are exacerbated by decreasing power budgets, increasing storage sizes, and technical difficulties in scaling the DRAM technology to smaller sizes. This research direction is currently work in progress, but preliminary work shows promising results that correlate with the reductions in the DUE rates obtained by the proposal presented in this work.

The AVF, and thus related metrics such as MVF and FEA, typically overestimate error rates due to faults in data by up to an order of magnitude. Further work could include extending the FEA metric to gain more insight into what data needs to be protected in priority. The hardware could also be extended to gather information on memory accesses, and directly provide the runtime with an estimation of this vulnerability metric.

7.1.3 Adapting ECC Dynamically

The last contribution consists of a hardware proposal for a dynamically adaptable ECC scheme using addressable memory as supplementary redundancy, WITSEC. Complementing this ECC scheme, we propose as part of this third contribution a methodology to measure dynamically at runtime the vulnerability of data. Specifically, the proposed methodology estimates the FEA metric by extrapolating memory access patterns from sampled loads and store instructions. This estimate is demonstrated to be an upper bound on the FEA metric, as it is the vulnerability as perceived from the CPU level.

The whole dynamic methodology has been implemented on a real machine, using the PMU capabilities of the POWER8 processor. The code of this implementation is released online, and has been evaluated by combining WITSEC and results from fault injection experiments in native runs. This demonstrates that the methodology can be implemented on commodity hardware with a low execution time overhead. Furthermore, the evaluation revealed that using WITSEC to extend protection of memory regions that are identified as most vulnerable allows for a wide variety of trade-offs between ECC overhead and failure rates. The ECC can be applied to targeted regions, and at a much smaller granularity than increasing the ECC strength uniformly, to decrease the failure rate at a fraction of the cost of a uniformly stronger ECC.

This methodology can be improved for better accuracy, by correlating access patterns with code phases using techniques such as piece-wise linear regressions [Servat *et al.* 2014], or by computing which accesses are served by caches and which reach memory using models such as those developed by Yu *et al.* [2014]. This could be especially relevant for other purposes that require less real-time performance, such as profiling the application statically. A static vulnerability profile can help decide data placement between two hardware locations with different er-

ror rates, or choose which data structures of the application should be protected using application-level techniques such as discussed in the previous chapters of this work: algorithmic recoveries, checksum verifications, etc. Finally, such a profile can also indicate whether a significant amount of data is stored in memory with no further use, indicating that the algorithm could be optimised with for example a loop-fusion approach to reduce its memory footprint.

The methodology developed for memory access pattern identification can be used for resilience beyond the computation of the FEA metric and estimation of vulnerability of data. Similar work has identified memory access pattern and their evolution over time for correlation with application source code for advanced analytics [Giménez *et al.* 2014; Servat *et al.* 2015]. Such an application-specific memory access pattern identification, correlated with the task information, is a very powerful tool. This information could be used for example to check that tasks behave within their normal parameters, accessing only data that is listed explicitly in data dependencies or that is normally accessed by this task. Deviating from this behaviour could raise an important flag at the runtime level, as a symptom that an error could have occurred, increasing the recall of symptom-based error detection techniques [Hari *et al.* 2009]. More powerful yet, such instrumentation allows to know which data has been modified by a task affected by an error. In this way, the effects of tasks whose control flow might have changed from the expected behaviour can be identified. This is a very promising step towards ensuring the containment of an error, which is a thorny issue in a shared memory environment.

7.1.4 Redundancy-Aware Runtime Systems

Redundancy is the only protection against random faults, and optimising its use will allow for less reliable DRAM memories, either from future process technologies or improving the yield of current ones. The work presented in this thesis demonstrates that hardware-software cooperation enables to use redundancy much more efficiently. Reporting errors at a fine granularity allows to correct them in a timely and low-overhead fashion, while powerful PMUs allow invaluable information to be gathered on an application. Hardware that can be dynamically adjusted at runtime allows the software to adjust the redundancy in memory, or reduce its energy consumption, instead of simply detecting and optimising the use of existing redundancy.

All such advanced techniques allow to better tolerate faults in memory, and need to be implemented at the runtime level. Runtime systems already embody the cooperation between hardware and software, as they have information on both to support programming models' features and provide them with performance, performance portability, support for heterogeneous computing platforms, etc. Furthermore, exposing all the resilience and instrumentation capabilities of modern hardware directly to the end programmer would be counter-productive and defeat the purpose of layered abstractions provided by programming models and instruction sets. Thus, in order to continue reaping the benefits of Moore's law and enable the next era of computing, runtime systems need to be aware of resilience constraints, managing and providing redundancy to safely tolerate future fault rates.

7.2 Publications

This section lists the publications linked to the research presented in this thesis. The first list contains the publications derived directly from the work presented, while the second list presents related publications. The final list presents the code developed in this thesis that is made publicly available.

Publications of the Thesis:

- **L. Jaulmes**, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero (2015), "Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM, 53:1–53:12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807599. **Nominated for the best paper award.**

- **L. Jaulmes**, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas (2018), "Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers," *IEEE Transactions on Parallel & Distributed Systems*, vol. 29, no. 9, pp. 1961–1974, ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2817524

- **L. Jaulmes**, M. Moretó, M. Valero, and M. Casas (2019c), "Memory Vulnerability: A Case for Delaying Error Reporting," presented at the 12th

Workshop on Programmability and Architectures for Heterogeneous Multicores, Multiprog 2019

- **L. Jaulmes**, M. Moretó, M. Valero, and M. Casas (2019a), "A Vulnerability Factor for ECC-protected Memory," currently under review at IOLTS 2019

- **L. Jaulmes**, M. Moretó, M. Valero, and M. Casas (2019b), "Adapting ECC Protection Dynamically using Online Estimation of Memory Vulnerability," currently under review at PACT 2019

Other Publications:

- M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, **L. Jaulmes**, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, J. Labarta, and M. Valero (2015), "Runtime-Aware Architectures," in *European Conference on Parallel Processing*, Euro-Par 2015, Springer, Berlin, Heidelberg, pp. 16–27, ISBN: 978-3-662-48095-3. DOI: 10.1007/978-3-662-48096-0_2

- D. Richards and **L. Jaulmes** (2014), "CoMD in Chapel: The Good, the Bad, and the Ugly," in *Chapel Lightning Talks*, Birds-of-a-Feather session at SC'14

Publicly Available Code:

- **L. Jaulmes** (2016). Resilient CG implementation, GitHub, `https://github.com/lucjaulmes/resilient_cg` (visited on 01/23/2019)

- **L. Jaulmes** (2018). Online sampling-based vulnerability estimator, GitHub, `https://github.com/lucjaulmes/online_vulnerability` (visited on 11/01/2018)

- **L. Jaulmes** (2019). OmpSs Fault Tolerance Benchmarks, GitHub, `https://github.com/lucjaulmes/ompss_fault_tolerance_benchmarks` (visited on 01/23/2019)

7.3 Financial and Technical Support

This thesis has been supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2013/06982, by the RoMoL ERC Advanced Grant (GA

321253), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P), and by the Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272).

The authors would like to thank Francesc Martínez Palau for his precious help and support with the TaskSim infrastructure. The authors would also like to thank Harald Servat and Ramon Bertran for their help in deciphering respectively the Intel and POWER8 PMUs' capabilities.

Appendix A
Online Vulnerabiliy Analysis Reproducibility Artifacts

In this appendix we describe how to reproduce the results from the online vulnerability analysis framework. We describe the common runtime-instrumentation part in Section A.1, the sampling-based vulnerability estimation (specific to POWER8) in Section A.2, and the error injection in Section A.3.

A.1 Runtime Instrumentation of Applications

We rely on preloading to instrument code at runtime without application-level knowledge. This allows to intercept any calls to library functions, in particular memory allocation and freeing, and task-creation functions from the task-based data-flow programming model. These functions' (e.g. `malloc`, `free`) are wrapped by functions in our code that record the address ranges used, and call the initially intended code.

To identify the ROI we use the same mechanism, by modifying the benchmark applications to call external library functions at the start and end of their ROI, as for example in the PARSEC benchmarks. We use two functions that for this purpose: `start_measure()` and `stop_measure(int it)`, which by default measure and report the time elapsed between the two calls. `it` is the number of iterations, `-1` can be used to indicate it is not applicable.

We used gcc $\geq$ v6.2, and the Mercurium source-to-source v2.0.0 compilers to compile benchmarks. The framework, which is publicly available [Jaulmes 2018], has been tested on Linux (with kernels $\geq$ v3.0.101), with the Nanos++ runtime system v0.10. The code works on any hardware, even though the sampling (and

thus estimating vulnerability) features require a POWER8 processor. These features are enabled by default, and can be disabled, for example to compile on x86 systems, by running:

```
$ make NO_SAMPLING=1
```

To execute an application with the preloaded instrumentation, run:

```
$ LD_PRELOAD=libvulnerability.so <benchmark>
```

This will output statistics such as execution time (in µs) to standard output, and the list of identified memory regions in the `./maps` file.

A.2 Estimating Vulnerability through Sampling

Estimating vulnerability through sampling is part of the library described in Section A.1, we only describe here what is specific to sampling and vulnerability estimation. This part relies the sampling capabilities of the POWER8 PMU capabilities and the KHT algorithm.

A.2.1 Sampling configuration

The core of the vulnerability estimate mechanism is the EBB handler depicted in Figure 6.7, which allows to read the virtual address of pseudo-randomly selected registers. It shifts the stack pointer by 512B (as recommended in Section *2.2.2. The Stack Frame* of the ABI Specification [IBM 2015a]) to save General Purpose Registers (GPRs) 1 – 4. With those, the handler reads the SPRs listed in Table A.1 and saves their values in a thread-local buffer. If the buffer is full, the handler deactivates sampling, otherwise it resets the marked instruction count to overflow in `sample_period` events. Finally, it restores the GPRs 1–4 and stack pointer.

The sampling setup is performed in the library's constructor function, thus before the program is loaded. The PMU setup to enable EBB handlers on *marked* loads and stores is described in Section 6.5.1, and we detail the register-level configuration corresponding to this setup in Table A.2. This setup is achieved using the `perf_event_open` system call, with the values of Table A.3. Once setup, the sampling can be enabled and disabled by flipping bits in the `MMCR0` and `BESCR` SPRs directly using the `mfspr` and `mtspr` (Move From/To SPR) instructions, thus

Table A.1: Registers used for sampling

register	value
SDAR	effective address of instruction target
SIER	metadata of instruction (LD or ST, etc.)
PMC5	number of instructions executed
TB	wall-clock time with 512MHz resolution

Table A.2: Registers used for PMU setup

Setup (with `perf_event_open` for `MMCR` registers)	
register	value
MMCR0	`0x0000000050184043`
MMCR1	`0x00000000000000e0`
MMCRA	`0x0000000000000001`
EBBHR	address of EBB handler
Enabling counters (start of sampling phase)	
PMC4	`0x80000000 - sample_period`
BESCRS	`0x8000000100000000`
MMCR0	`0x0000000004000000`
Disabling counters (end of sampling phase)	
BESCRR	`0x8000000100000000`
MMCR0	`0x0000000080000000`

without requiring a system call. These instructions are also used to set the `EBBHR` register. The `BESCRS` and `BESCRR` registers are shorthands allowing to respectively Set and Reset bits in the `BESCR` register.

Estimating vulnerability through sampling requires the KHT algorithm [Fernandes and Oliveira 2008], which is available online [Fernandes 2008], and a POWER8 processor. The instrumentation framework has the same outputs as described in Section A.1, with the addition of average vulnerability per region in the regions list, and statistics per sampling phase on the standard output, such as number of samples and distribution of vulnerability across regions.

The library's parameters listed below can be adjusted at compile-time to trade overhead for precision of vulnerability estimates.

- The sampling period can be set by overriding the value in the Makefile:

```
$ make SAMPLE_PERIOD=2
```

- The phases durations and intervals are set by the arguments passed to `timer_settime`, in `vulnerability.c`:

Table A.3: `perf` configuration to setup EBB

Values marked with * are `or`-ed together in a single `perf_event_open` call, and the value marked with † is similarly added to every call.

config	value	Description
`MRK_INST_CMPL`	`0x00401e0*`	Count *marked instructions* in PMC4
`RUN_INST_CMPL`	`0x00500fa`	Count *run instructions* in PMC5
`RUN_CYC`	`0x00600f4`	Count *run cycles* in PMC6
sampling mode	`0x4000000*`	Select *Random Instruction Sampling* mode, with *Loads and Stores* as eligible instructions
EBB	`1 << 63 †`	Use EBBs, enable PMCs 5-6 and allow unprivileged access to `MMCR0` (with `MMCR0.PMCC=0b10`)

```
424 /* Re-arm timer for 'payoff' phase */
425 struct itimerspec spec_toggle = {.it_interval = {0}, .it_value = {.tv_nsec =
        ↪  500000000}};
426 timer_settime(next_toggle, 0, &spec_toggle, NULL);
```

```
430 /* Re-arm timer for learning phase */
431 struct itimerspec spec_toggle = {.it_interval = {0}, .it_value = {.tv_nsec =
        ↪  500000000}};
432 timer_settime(next_toggle, 0, &spec_toggle, NULL);
```

- The parameters passed to the KHT algorithm are also set in `vulnerability.c`:

```
966 const size_t cluster_min_size = 10;
967 const double cluster_min_deviation = 2.0, delta_theta = 0.25, delta_rho =
        ↪ 4096, kernel_min_rel_height = 0.9, n_sigmas = 2.0;
```

A.3 Error Injections

We only describe the specificities of error injection here, as it is part of the library described in Section A.1. Error injections can run on any platform, taking care to disable vulnerability estimates on platforms that do not support it.

To inject errors, the library must be loaded to instrument the benchmark as in Section A.1. Parameters for the error injection are set as classic command line parameters, passed through the `INJECT` environment variable. In the example below, we inject a double bit flip (`-n 2`) in a random position inside the first identified memory region (`-v 0`), which we undo at the end of the ROI (`-u`). The error is injected at a random time within 12000ns of the ROI's start (`-m 12000`).

```
$ export INJECT="-n 2 -v 0 -u -m 12000"
$ LD_PRELOAD=libvulnerability.so <benchmark>
```

The library will print which bit(s) it will flip, and at which time in the ROI. When flipping the bits, the library prints a message, containing the current vulnerability rating of the targeted memory region if sampling is enabled. This message allows to verify both that the error was indeed injected, and whether it would be corrected or not given a vulnerability threshold for WITSEC's extended protection. The benchmarks' output and result verification differ for each benchmark, and include the crashing of the program due to the injected error.

A.3.1 Installation

Simply compile and install Nanos++ and Mercurium as per their documentation, then run `make` in the root directories of all the libraries and benchmarks.

How software can be obtained. Nanos++ and Mercurium are available online at `https://pm.bsc.es/ompss-downloads/`.
The library implementing our online vulnerability estimates and a modified KHT are available on github [Jaulmes 2018].

Hardware dependencies. The online vulnerability estimates through sampling require the POWER8 PMU capabilities.

Software dependencies. The online vulnerability estimates require that the ELF v2 ABI is used, which is assumed in the EBB handler. Furthermore, setting up the counters through `perf_event_open` requires a reasonably recent Linux kernel (tested with v3.10.0).

Appendix B
TaskSim Simulation Reproducibility Artifacts

During tracing, TaskSim automatically detects the ROI of PARSEC benchmarks. We have added symbols aliasing PARSEC ROI markers to the ones used in our benchmarks to support detection of ROIs out of the box with TaskSim:

```
void __parsec_roi_begin() __attribute__((noinline, alias("start_measure")));
void __parsec_roi_end() __attribute__((noinline, alias("stop_measure")));
```

To use the vulnerability statistics with which TaskSim was extended, the configuration file must be extended to contain the values listed below in the selected section. For example, to perform statistics at memory level, the `Perfect` module:

```
[Perfect]
# none, time, access, or histogram
vulnerability_stat = time
# granularity of stats
vulnerability_ignoredlsb = 6
# cycles count between stats outputs
vulnerability_period = 100000000
# discretisation of vulnerability (histogram only):
# bin N has vuln s.t.: N/n_bins <= vuln < (N+1)/n_bins
vulnerability_bins = 100
```

This can be done at several different modules in the memory hierarchy, such as cache controllers or cores, or the Ramulator module if TaskSim relies on Ramulator for its memory model. The `time` statistic allows to compute the vulnerability (MVF and FEA) by reporting the average number of cycles before each event: store, load with intent to overwrite, load, and end of simulation. The `access` statistic reports the number of loads and stores, which allows to compute the DVF

and LD/ST ratios. The `none` statistic disabled computing the vulnerability and the `histogram` shows the distribution of vulnerability values across each memory region, instead of the average.

The simulations were run with the TaskSim multicore simulation infrastructure, which is currently not publicly available, compiled using gcc 7.1, on a Linux environment, with the Nanos++ task-based data-flow runtime system. The required hardware for tracing is an x86 or ARM processor, while any hardware can be simulated. TaskSim is run with the values listed above set appropriately in the configuration file:

```
$ tasksim <configuration file> <trace file>
```

TaskSim outputs statistics to the standard output, including a filename of the form `vulnerability.<unique suffix>`, which contains the vulnerability statistics requested in the configuration file.

Bibliography

Abdoo, D. G. and Cabello, J. D. (1996), "Error correction system for n bits using error correcting code designed for fewer than n bits," U.S. Patent 5490155 A, U.S. Classification 714/763, 714/785, 714/E11.046; International Classification G06F11/10; Cooperative Classification G06F11/1028; European Classification G06F11/10M1P.

Advanced Micro Devices (AMD), Inc. (2018), "AMD64 Architecture Programmer's Manual Volume 2: System Programming," Publication # 24593, Revision 2.30.

Agullo, E., Giraud, L., Salas, P., and Zounon, M. (2016a), "Interpolation-Restart Strategies for Resilient Eigensolvers," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. C560–C583, ISSN: 1064-8275. DOI: 10.1137/15M1042115.

Agullo, E., Cools, S., Fatih-Yetkin, E., Giraud, L., and Vanroose, W. (2018), "On soft errors in the Conjugate Gradient method: sensitivity and robust numerical detection," Inria Bordeaux Sud-Ouest, Research Report RR-9226.

Agullo, E., Giraud, L., Guermouche, A., Roman, J., and Zounon, M. (2013), "Towards resilient parallel linear Krylov solvers: recover-restart strategies," INRIA, Research Report RR-8324.

Agullo, E., Giraud, L., Guermouche, A., Roman, J., and Zounon, M. (2016b), "Numerical recovery strategies for parallel resilient Krylov linear solvers," *Numerical Linear Algebra with Applications*, vol. 23, no. 5, pp. 888–905, ISSN: 1099-1506. DOI: 10.1002/nla.2059.

Agullo, E., Giraud, L., and Zounon, M. (2015), "On the Resilience of Parallel Sparse Hybrid Solvers," in *22nd International Conference on High Performance Computing*, HiPC, pp. 75–84. DOI: 10.1109/HiPC.2015.9.

Alameldeen, A. R., Wagner, I., Chishti, Z., Wu, W., Wilkerson, C., and Lu, S.-L. (2011), "Energy-efficient Cache Design Using Variable-strength Error-correcting Codes," in *Proceedings of the 38th Annual International Symposium*

on Computer Architecture, ISCA '11, ACM, pp. 461–472, ISBN: 978-1-4503-0472-6. DOI: 10.1145/2000064.2000118.

Alvarez, L., Casas, M., Labarta, J., Ayguadé, E., Valero, M., and Moreto, M. (2018), "Runtime-Guided Management of Stacked DRAM Memories in Task Parallel Programs," in *Proceedings of the 32nd International Conference on Supercomputing*, ICS '18, ACM, pp. 218–228, ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205312.

Alvarez, L., Moretó, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguadé, E., and Valero, M. (2015), "Runtime-Guided Management of Scratchpad Memories in Multicore Architectures," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, PACT '15, pp. 379–391, ISBN: 978-1-4673-9524-3. DOI: 10.1109/PACT.2015.26.

Baek, S., Cho, S., and Melhem, R. (2014), "Refresh Now and Then," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 3114–3126, ISSN: 0018-9340. DOI: 10.1109/TC.2013.164.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Van der Vorst, H. (1994), *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial Mathematics, ISBN: 978-0-89871-328-2. DOI: 10.1137/1.9781611971538.

Baumann, R. (2005), "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, ISSN: 0740-7475. DOI: 10.1109/MDT.2005.69.

Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S. (2011), "FTI: High performance Fault Tolerance Interface for hybrid systems," in *Proceedings of the Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1–12. DOI: 10.1145/2063384.2063427.

Bautista-Gomez, L., Zyulkyarov, F., Unsal, O., and McIntosh-Smith, S. (2016), "Unprotected Computing : A Large-Scale Study of DRAM Raw Error Rate on a Supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, IEEE, 55:1–55–11. DOI: 10.1109/SC.2016.54.

Bellens, P., Perez, J. M., Badia, R. M., and Labarta, J. (2006), "CellSs: a Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 ACM/IEEE Conference on High Performance Networking and Computing*, SC '06, pp. 5–5, ISBN: 0-7695-2700-0. DOI: 10.1109/SC.2006.17.

Berrocal, E., Bautista Gomez, L., Di, S., Lan, Z., and Cappello, F. (2017), "Toward General Software Level Silent Data Corruption Detection for Parallel Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3642–3655, ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2735971.

Beveridge, J. and Wiener, B. (1997), *Multithreading Applications in Win32: The Complete Guide to Threads*. Addison-Wesley Longman Publishing Co., Inc., ISBN: 978-0-201-44234-2.

Binder, D., Smith, E. C., and Holman, A. B. (1975), "Satellite Anomalies from Galactic Cosmic Rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, ISSN: 0018-9499. DOI: 10.1109/TNS.1975.4328188.

Bland, W., Du, P., Bouteiller, A., Herault, T., Bosilca, G., and Dongarra, J. J. (2013), "Extending the scope of the Checkpoint-on-Failure protocol for forward recovery in standard MPI," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 17, pp. 2381–2393, ISSN: 1532-0634. DOI: 10.1002/cpe.3100.

Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995), "Cilk: an Efficient Multithreaded Runtime System," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '95, ACM, pp. 207–216, ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958.

Bohr, M. (2007), "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785534.

Bougeret, M., Casanova, H., Rabie, M., Robert, Y., and Vivien, F. (2011), "Checkpointing strategies for parallel jobs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 33:1–33:11, ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063428.

Bridges, P. G., Ferreira, K. B., Heroux, M. A., and Hoemmen, M. (2012a), "Fault-tolerant Linear Solvers via Selective Reliability." arXiv: 1206.1390.

Bridges, P. G., Hoemmen, M., Ferreira, K. B., Heroux, M. A., Soltero, P., and Brightwell, R. (2012b), "Cooperative Application/OS DRAM Fault Recovery,"

in *Euro-Par 2011 Parallel Processing Workshops*, Springer-Verlag, pp. 241–250, ISBN: 978-3-642-29739-7. DOI: 10.1007/978-3-642-29740-3_28.

Bronevetsky, G. and Supinski, B. R. de (2008), "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd International Conference on Supercomputing*, ICS '08, ACM, pp. 155–164. DOI: 10.1145/1375527.1375552.

Brumar, I., Casas, M., Moreto, M., Valero, M., and Sohi, G. S. (2017), "ATM: Approximate Task Memoization in the Runtime System," in *Proceedings of the IEEE 31st International Parallel and Distributed Processing Symposium*, IPDPS, pp. 1140–1150. DOI: 10.1109/IPDPS.2017.49.

BSC Programming Models (2018). OmpSs Specification, http://pm.bsc.es/ompss-docs/spec/OmpSsSpecification.pdf (visited on 11/19/2018).

Bueno, J., Martinell, L., Duran, A., Farreras, M., Martorell, X., Badia, R. M., Ayguade, E., and Labarta, J. (2011), "Productive Cluster Programming with OmpSs," in *European Conference on Parallel Processing*, Euro-Par 2011, Springer, pp. 555–566, ISBN: 978-3-642-23400-2. DOI: 10.1007/978-3-642-23400-2_52.

Bueno, J., Martorell, X., Badia, R. M., Ayguadé, E., and Labarta, J. (2013), "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 359–368, ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465017.

Caheny, P., Alvarez, L., Derradji, S., Valero, M., Moretó, M., and Casas, M. (2018a), "Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 5, pp. 1174–1187, ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2787123.

Caheny, P., Alvarez, L., Valero, M., Moretó, M., and Casas, M. (2018b), "Runtime-assisted Cache Coherence Deactivation in Task Parallel Programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, 35:1–35:12.

Caheny, P., Casas, M., Moretó, M., Gloaguen, H., Saintes, M., Ayguadé, E., Labarta, J., and Valero, M. (2016), "Reducing cache coherence traffic with hierarchical directory cache and NUMA-aware runtime scheduling," in *Proceedings of the 25th International Conference on Parallel Architecture and Compilation Techniques*, PACT '16, pp. 275–286. DOI: 10.1145/2967938.2967962.

Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M. (2009), "Toward Exascale Resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, ISSN: 1094-3420. DOI: 10.1177/1094342009347767.

Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., and Snir, M. (2014), "Towards Exascale Resilience: 2014 update," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 5–28, ISSN: 2313-8734. DOI: 10.14529/jsfi140101.

Casas, M., Moretó, M., Alvarez, L., Castillo, E., Chasapis, D., Hayes, T., Jaulmes, L., Palomar, O., Unsal, O., Cristal, A., Ayguadé, E., Labarta, J., and Valero, M. (2015), "Runtime-Aware Architectures," in *European Conference on Parallel Processing*, Euro-Par 2015, Springer, Berlin, Heidelberg, pp. 16–27, ISBN: 978-3-662-48095-3. DOI: 10.1007/978-3-662-48096-0_2.

Casas, M., Supinski, B. R. de, Bronevetsky, G., and Schulz, M. (2012), "Fault resilience of the algebraic multi-grid solver," in *Proceedings of the 26th International Conference on Supercomputing*, ICS '12, ACM, pp. 91–100, ISBN: 978-1-4503-1316-2. DOI: 10.1145/2304576.2304590.

Castillo, E., Alvarez, L., Moreto, M., Casas, M., Vallejo, E., Bosque, J. L., Beivide, R., and Valero, M. (2018), "Architectural Support for Task Dependence Management with Flexible Software Scheduling," in *IEEE 24th International Symposium on High Performance Computer Architecture*, HPCA, pp. 283–295. DOI: 10.1109/HPCA.2018.00033.

Castillo, E., Moretó, M., Casas, M., Alvarez, L., Vallejo, E., Chronaki, K., Badia, R., Bosque, J. L., Beivide, R., Ayguadé, E., Labarta, J., and Valero, M. (2016), "CATA: Criticality Aware Task Acceleration for Multicore Processors," in *Proceedings of the IEEE 30th International Parallel and Distributed Processing Symposium*, IPDPS, pp. 413–422. DOI: 10.1109/IPDPS.2016.49.

Chamberlain, B., Callahan, D., and Zima, H. (2007), "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, ISSN: 1094-3420. DOI: 10.1177/1094342007078442.

Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., and Valero, M. (2015), "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite," *ACM Transactions on Architecture and*

Code Optimization, TACO, vol. 12, no. 4, 41:1–41:22, ISSN: 1544-3566. DOI: 10.1145/2829952.

Chen, G., Kandemir, M., Irwin, M. J., and Memik, G. (2005), "Compiler-directed Selective Data Protection Against Soft Errors," in *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, ACM, pp. 713–716, ISBN: 978-0-7803-8737-9. DOI: 10.1145/1120725.1121000.

Chen, Z. (2011), "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, ACM, pp. 73–84, ISBN: 978-1-4503-0552-5. DOI: 10.1145/1996130.1996142.

Chen, Z. (2013), "Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, ACM, pp. 167–176, ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442533.

Cho, Y.-C., Bae, Y.-C., Moon, B.-M., Eom, Y.-J., Ahn, M.-S., Lee, W.-Y., Cho, C.-R., Park, M.-H., Jeon, Y.-J., Ahn, J.-O., Choi, B.-K., Kang, D.-K., Yoon, S.-H., Yang, Y.-S., Park, K.-I., Choi, J.-H., Lee, J.-B., and Choi, J.-S. (2013), "A Sub-1.0V 20nm 5Gb/s/pin post-LPDDR3 I/O interface with Low Voltage-Swing Terminated Logic and adaptive calibration scheme for mobile application," in *Symposium on VLSI Circuits Digest of Technical Papers*, VLSIC, pp. C240–C241, ISBN: 978-1-4673-5531-5. DOI: 10.1109/VLSIC.2013.6578678.

Chronaki, K., Rico, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2015), "Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures," in *Proceedings of the 29th International Conference on Supercomputing*, ICS '15, ACM, pp. 329–338, ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751235.

Chronaki, K., Rico, A., Casas, M., Moretó, M., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2017), "Task Scheduling Techniques for Asymmetric Multi-Core Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 2074–2087, ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2633347.

Davies, T. and Chen, Z. (2013), "Correcting soft errors online in LU factorization," in *Proceedings of the 22nd international symposium on High-performance par-*

allel and distributed computing, HPDC '13, ACM, pp. 167–178, ISBN: 978-1-4503-1910-2. DOI: 10.1145/2493123.2462920.

Davis, T. A. and Hu, Y. (2011), "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, 1:1–1:25, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

Degalahal, V., Ramanarayanan, R., Vijaykrishnan, N., Xie, Y., and Irwin, M. (2004), "The effect of threshold voltages on the soft error rate [memory and logic circuits]," in *5th International Symposium on Quality Electronic Design, 2004. Proceedings*, pp. 503–508. DOI: 10.1109/ISQED.2004.1283723.

Dell, T. J. (1997), "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," IBM Microelectronics Division, white paper.

Di Martino, C., Kalbarczyk, Z., Iyer, R. K., Baccanico, F., Fullop, J., and Kramer, W. (2014), "Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters," in *44th International Conference on Dependable Systems and Networks*, DSN 2014, IEEE, pp. 610–621. DOI: 10.1109/DSN.2014.62.

Di, S. and Cappello, F. (2016), "Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2809–2823, ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2517639.

Di, S., Robert, Y., Vivien, F., and Cappello, F. (2017), "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 244–259, ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2546248.

Dimić, V., Moretó, M., Casas, M., and Valero, M. (2017), "Runtime-Assisted Shared Cache Insertion Policies Based on Re-reference Intervals," in *European Conference on Parallel Processing*, Euro-Par 2017, Springer, pp. 247–259, ISBN: 978-3-319-64203-1. DOI: 10.1007/978-3-319-64203-1_18.

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., and Planas, J. (2011), "OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures.," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, ISSN: 0129-6264. DOI: 10.1142/S0129626411000151.

Ecma International (2015), "ECMAScript 2015 Language Specification," Standard ECMA-262.

Elliott, J., Hoemmen, M., and Mueller, F. (2014), "Evaluating the Impact of SDC on the GMRES Iterative Solver," in *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS, pp. 1193–1202, ISBN: 978-1-4799-3800-1. DOI: 10.1109/IPDPS.2014.123.

Elpida Memory, Inc. (2005), "Low Power Function of Mobile RAM: Partial Array Self Refresh (PASR)," Technical Note E0597E10, Ver. 1.0.

Etsion, Y., Cabarcas, F., Rico, A., Ramirez, A., Badia, R. M., Ayguadé, E., Labarta, J., and Valero, M. (2010), "Task Superscalar: An Out-of-Order Task Pipeline," in *Proceedings of the 43rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 43, pp. 89–100. DOI: 10.1109/MICRO.2010.13.

European Technology Platform for High Performance Computing ETP4HPC (2017), "Third Strategic Research Agenda," European Multi-annual HPC Technology Roadmap SRA 3.

Fagg, G. E., Bukovsky, A., and Dongarra, J. J. (2001), "Fault Tolerant MPI for the HARNESS Meta-computing System," in *Computational Science — ICCS 2001*, Alexandrov, V. N., Dongarra, J. J., Juliano, B. A., Renner, R. S., and Tan, C. J. K., Eds. (2001), ICCS 2001, Springer, pp. 355–366, ISBN: 978-3-540-42232-7. DOI: 10.1007/3-540-45545-0_44.

Fernandes, L. A. F. (2008). Kernel-Based Hough Transform, http://www2.ic.uff.br/~laffernandes/projects/kht/ (visited on 10/31/2018).

Fernandes, L. A. F. and Oliveira, M. M. (2008), "Real-time line detection through an improved Hough transform voting scheme," *Pattern Recognition*, vol. 41, no. 1, pp. 299–314, ISSN: 0031-3203. DOI: 10.1016/j.patcog.2007.04.003.

Ferreira, K., Stearley, J., Laros, J. H., Oldfield, R., Pedretti, K., Brightwell, R., Riesen, R., Bridges, P. G., and Arnold, D. (2011), "Evaluating the viability of process replication reliability for exascale systems," ACM Press, p. 1, ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063443.

Fougstedt, C., Szczerba, K., and Larsson-Edefors, P. (2017), "Low-Power Low-Latency BCH Decoders for Energy-Efficient Optical Interconnects," *Journal of Lightwave Technology*, vol. 35, no. 23, pp. 5201–5207, ISSN: 0733-8724. DOI: 10.1109/JLT.2017.2764679.

Giménez, A., Gamblin, T., Rountree, B., Bhatele, A., Jusufi, I., Bremer, P.-T., and Hamann, B. (2014), "Dissecting On-node Memory Access Performance: A Se-

mantic Approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE Press, pp. 166–176, ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.19.

Gioiosa, R., Sancho, J. C., Jiang, S., Petrini, F., and Davis, K. (2005), "Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers," in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, SC '05, pp. 9–9. DOI: 10.1109/SC.2005.76.

Gong, S.-L., Kim, J., Lym, S., Sullivan, M., David, H., and Erez, M. (2018), "DUO: Dual Use of On-chip Redundancy for High Reliability," in *IEEE 24th International Symposium on High Performance Computer Architecture*, HPCA. DOI: 10.1109/HPCA.2018.00064.

Grass, T., Allande, C., Armejach, A., Rico, A., Ayguadé, E., Labarta, J., Valero, M., Casas, M., and Moreto, M. (2016), "MUSA: A Multi-level Simulation Approach for Next-Generation HPC Machines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pp. 526–537. DOI: 10.1109/SC.2016.44.

Gropp, W., Lusk, E., and Skjellum, A. (1994), *Using MPI: Portable Parallel Programming with the Message-passing Interface*. MIT Press, ISBN: 978-0-262-57104-3.

Gropp, W. and Snir, M. (2013), "Programming for Exascale Computers," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 27–35, ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.96.

Gupta, M., Sridharan, V., Roberts, D., Prodromou, A., Venkat, A., Tullsen, D., and Gupta, R. (2018), "Reliability-Aware Data Placement for Heterogeneous Memory Architecture," in *IEEE 24th International Symposium on High Performance Computer Architecture*, HPCA, pp. 583–595. DOI: 10.1109/HPCA.2018.00056.

Hamming, R. (1950), "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, pp. 147–160,

Hari, S. K. S., Li, M.-L., Ramachandran, P., Choi, B., and Adve, S. V. (2009), "mSWAT: low-cost hardware fault detection and diagnosis for multicore systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium*

on Microarchitecture, MICRO 42, ACM, pp. 122–132. DOI: 10.1145/1669112.1669129.

Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Salinger, A. G., Thornquist, H. K., Tuminaro, R. S., Willenbring, J. M., Williams, A., and Stanley, K. S. (2005), "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 397–423, ISSN: 0098-3500. DOI: 10.1145/1089014.1089021.

Heroux, M. A., Dongarra, J., and Luszczek, P. (2013), "HPCG Technical Specification," Sandia National Laboratories, Sandia Report SAND2013-8752.

Hestenes, M. R. and Stiefel, E. (1952), "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436,

Hillis, W. D. (1981), "The Connection Machine," Massachusetts Institute of Technology Artificial Intelligence Laboratory, A.I. Memo No. 646.

Hocquenghem, A. (1959), "Codes Correcteurs d'Erreurs," *Chiffres*, vol. 2, pp. 147–156,

Hong, S. (2010), "Memory technology trend and future challenges," in *International Electron Devices Meeting Technical Digest*, IEDM, pp. 12.4.1–12.4.4. DOI: 10.1109/IEDM.2010.5703348.

HPEC (2013), "How Memory RAS Technologies Can Enhance the Uptime of HP ProLiant Servers," Technical white paper 4AA4-3490ENW.

Hsiao, M. Y. (1970), "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, ISSN: 0018-8646, 0018-8646. DOI: 10.1147/rd.144.0395.

Huang, K.-H. and Abraham, J. A. (1984), "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. 33, no. 6, pp. 518–528, ISSN: 0018-9340. DOI: 10.1109/TC.1984.1676475.

IBM Corporation (1999), "Enhancing IBM Netfinity Server Reliability: IBM Chipkill Memory," white paper.

IBM Corporation (2015a), "OpenPOWER ABI for Linux Supplement for the Power Architecture 64-bit ELF V2 ABI," Specification, Version 1.1.

IBM Corporation (2015b), "Power ISA," Specification, Version 2.07 B.

Intel Corporation (2011), "Intel® Xeon® Processor E7 Family: Reliability, Availability, and Serviceability," Data Center Group, Intel, white paper.

Intel Corporation (2017), "Intel® 64 and IA-32 Architectures Software Developer Manual Volume 3: System Programming Guide," 325384, version 052.

International Organization for Standardization (ISO), Technical Committee SO/IEC JTC 1/SC 22 (2011), "Standard for Programming Language C++," International Standard ISO/IEC 14882:2011.

Isen, C. and John, L. (2009), "ESKIMO - energy savings using semantic knowledge of inconsequential memory occupancy for DRAM subsystem," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 337–346. DOI: 10.1145/1669112.1669156.

Jaulmes, L. (2016). Resilient CG implementation, GitHub, https://github.com/lucjaulmes/resilient_cg (visited on 01/23/2019).

Jaulmes, L. (2018). Online sampling-based vulnerability estimator, GitHub, https://github.com/lucjaulmes/online_vulnerability (visited on 11/01/2018).

Jaulmes, L. (2019). OmpSs Fault Tolerance Benchmarks, GitHub, https://github.com/lucjaulmes/ompss_fault_tolerance_benchmarks (visited on 01/23/2019).

Jaulmes, L., Casas, M., Moretó, M., Ayguadé, E., Labarta, J., and Valero, M. (2015), "Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM, 53:1–53:12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807599.

Jaulmes, L., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. (2018), "Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers," *IEEE Transactions on Parallel & Distributed Systems*, vol. 29, no. 9, pp. 1961–1974, ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2817524.

Jaulmes, L., Moretó, M., Valero, M., and Casas, M. (2019a), "A Vulnerability Factor for ECC-protected Memory," currently under review at IOLTS 2019.

Jaulmes, L., Moretó, M., Valero, M., and Casas, M. (2019b), "Adapting ECC Protection Dynamically using Online Estimation of Memory Vulnerability," currently under review at PACT 2019.

Jaulmes, L., Moretó, M., Valero, M., and Casas, M. (2019c), "Memory Vulnerability: A Case for Delaying Error Reporting," presented at the 12th Workshop on

Programmability and Architectures for Heterogeneous Multicores, Multiprog 2019.

JEDEC Solid State Technology Association (2007), "Test Method for Alpha Source Accelerated Soft Error Rate," JEDEC Standard JESD89-2A.

JEDEC Solid State Technology Association (2013), "DDR4 SDRAM," JEDEC Standard JESD79-4A, Revision A.

JEDEC Solid State Technology Association (2014), "Low Power Double Data Rate 4 (LPDDR4)," JEDEC Standard JESD209-4.

JEDEC Solid State Technology Association (2016), "0.6 V Low Voltage Swing Terminated Logic (LVSTL06)," JEDEC Standard JESD8-29.

Jia, Z., Treichler, S., Shipman, G., Bauer Michael, B., Watkins, N., Maltzahn, C., McCormick, P., and Aiken, A. (2017), "Integrating External Resources with a Task-Based Programming Model," in *24th IEEE International Conference on High Performance Computing*, HiPC, pp. 307–316. DOI: 10.1109/HiPC.2017.00043.

Kale, L. V. and Krishnan, S. (1993), "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, ACM, pp. 91–108, ISBN: 0-89791-587-9. DOI: 10.1145/165854.165874.

Kang, U., Yu, H.-S., Park, C., Zheng, H., Halbert, J., Bains, K., Jang, S., and Choi, J. S. (2014), "Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling," in *The Memory Forum*.

Kaul, H., Anders, M., Hsu, S., Agarwal, A., Krishnamurthy, R., and Borkar, S. (2012), "Near-threshold voltage (NTV) design — Opportunities and challenges," in *Proceedings of the 49th annual Design Automation Conference*, DAC, pp. 1149–1154. DOI: 10.1145/2228360.2228572.

Kim, J., Sullivan, M., and Erez, M. (2015), "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," in *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA, IEEE, pp. 101–112, ISBN: 978-1-4799-8930-0. DOI: 10.1109/HPCA.2015.7056025.

Kim, Y., Seshadri, V., Lee, D., Liu, J., and Mutlu, O. (2012), "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM," in *Proceedings of the*

39th Annual International Symposium on Computer Architecture, ISCA '12, pp. 368–379. DOI: 10.1109/ISCA.2012.6237032.

Kim, Y., Yang, W., and Mutlu, O. (2016), "Ramulator: a Fast and Extensible DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, ISSN: 1556-6056. DOI: 10.1109/LCA.2015.2414456.

Kleen, A. (2010), "mcelog: memory error handling in user space," presented at the Linux Kongress, Lehmanns, pp. 159–166, ISBN: 978-3-86541-398-7.

Krishnan, P. (2009), "Hardware Breakpoint (or watchpoint) usage in Linux Kernel," in *Proceedings of the Linux Symposium*, pp. 149–158.

Kumar, S., Hughes, C. J., and Nguyen, A. (2007), "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, ACM, pp. 162–173, ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250683.

Langou, J., Chen, Z., Bosilca, G., and Dongarra, J. (2007), "Recovery Patterns for Iterative Methods in a Parallel Unstable Environment," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 102–116, ISSN: 1064-8275. DOI: 10.1137/040620394.

Levy, S., Ferreira, K. B., DeBardeleben, N., Siddiqua, T., Sridharan, V., and Baseman, E. (2018), "Lessons Learned from Memory Errors Observed over the Lifetime of Cielo," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, 43:1–43:12.

Li, D., Chen, Z., Wu, P., and Vetter, J. S. (2013), "Rethinking Algorithm-based Fault Tolerance with a Cooperative Software-hardware Approach," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, ACM, 44:1–44:12, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503226.

Li, D., Vetter, J. S., and Yu, W. (2012), "Classifying Soft Error Vulnerabilities in Extreme-scale Scientific Applications Using a Binary Instrumentation Tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press, 57:1–57:11, ISBN: 978-1-4673-0804-5. DOI: 10.1109/SC.2012.29.

Li, X., Adve, S. V., Bose, P., and Rivers, J. A. (2007), "Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions," in *37th Inter-*

national Conference on Dependable Systems and Networks, DSN 2007, pp. 266–275. DOI: 10.1109/DSN.2007.15.

Liang, X., Chen, J., Tao, D., Li, S., Wu, P., Li, H., Ouyang, K., Liu, Y., Song, F., and Chen, Z. (2017), "Correcting Soft Errors Online in Fast Fourier Transform," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, ACM, 30:1–30:12, ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126915.

Lin, C.-H., Shen, D.-Y., Chen, Y.-J., Yang, C.-L., and Wang, M. (2012), "SECRET: Selective error correction for refresh energy reduction in DRAMs," in *IEEE 30th International Conference on Computer Design*, ICCD, pp. 67–74. DOI: 10.1109/ICCD.2012.6378619.

Liu, H., Chen, Y., Liao, X., Jin, H., He, B., Zheng, L., and Guo, R. (2017), "Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures," in *Proceedings of the International Conference on Supercomputing*, ICS '17, 26:1–26:10, ISBN: 978-1-4503-5020-4. DOI: 10.1145/3079079.3079089.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005), "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, ACM, pp. 190–200, ISBN: 978-1-59593-056-9. DOI: 10.1145/1065010.1065034.

Luo, Y., Govindan, S., Sharma, B., Santaniello, M., Meza, J., Kansal, A., Liu, J., Khessib, B., Vaid, K., and Mutlu, O. (2014), "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *44th International Conference on Dependable Systems and Networks*, DSN 2014, IEEE, pp. 467–478, ISBN: 978-1-4799-2233-8. DOI: 10.1109/DSN.2014.50.

Malek, A., Vasilakis, E., Papaefstathiou, V., Trancoso, P., and Sourdis, I. (2017), "Odd-ECC: On-demand DRAM Error Correcting Codes," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, ACM, pp. 96–111, ISBN: 978-1-4503-5335-9. DOI: 10.1145/3132402.3132443.

Manivannan, M., Negi, A., and Stenström, P. (2013), "Efficient Forwarding of Producer-Consumer Data in Task-Based Programs," in *42nd International Con-*

ference on Parallel Processing, ICPP, pp. 517–522. DOI: 10.1109/ICPP.2013.64.

Manivannan, M., Papaefstathiou, V., Pericàs, M., and Stenström, P. (2016), "RADAR: Runtime-assisted dead region management for last-level caches," in *IEEE 22nd International Symposium on High Performance Computer Architecture*, HPCA, pp. 644–656. DOI: 10.1109/HPCA.2016.7446101.

Manivannan, M. and Stenström, P. (2014), "Runtime-Guided Cache Coherence Optimizations in Multi-core Architectures," in *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS, pp. 625–636, ISBN: 978-1-4799-3800-1. DOI: 10.1109/IPDPS.2014.71.

May, T. C. and Woods, M. H. (1978), "A New Physical Mechanism for Soft Errors in Dynamic Memories," in *16th International Reliability Physics Symposium*, pp. 33–40. DOI: 10.1109/IRPS.1978.362815.

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

Menabrea, L. F. (1842), "Notions sur la Machine Analytique de M. Charles Babbage," *Bibliothèque Universelle de Genève*, nouvelle série, vol. 41, pp. 352–376,

Micron Technology, Inc. (2017), "ECC Brings Reliability and Power Efficiency to Mobile Devices," white paper.

Mitra, S., Bose, P., Cheng, E., Cher, C.-Y., Cho, H., Joshi, R., Kim, Y. M., Lefurgy, C. R., Li, Y., Rodbell, K. P., Skadron, K., Stathis, J., and Szafaryn, L. (2014), "The resilience wall: Cross-layer solution strategies," in *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pp. 1–11. DOI: 10.1109/VLSI-TSA.2014.6839639.

Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. de (2010), "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.18.

Moore, G. (1965), "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117,

Mukherjee, S. S., Weaver, C., Emer, J., Reinhardt, S. K., and Austin, T. (2003), "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, IEEE Computer Society, pp. 29–, ISBN: 0-7695-2043-X. DOI: 10.1109/MICRO.2003.1253181.

Mukherjee, S. (2008), *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., ISBN: 978-0-12-369529-1.

Nair, P. J., Kim, D.-H., and Qureshi, M. K. (2013), "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, pp. 72–83, ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485929.

Nair, P. J., Sridharan, V., and Qureshi, M. K. (2016), "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, ISCA '16, pp. 341–353. DOI: 10.1109/ISCA.2016.38.

Nethercote, N., Walsh, R., and Fitzhardinge, J. (2006), "Building Workload Characterization Tools with Valgrind," in *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*, IISWC 2006, ISBN: 1-4244-0508-4. DOI: 10.1109/IISWC.2006.302723.

Nichols, B., Buttlar, D., Farrell, J., and Farrell, J. (1996), *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, Inc., ISBN: 978-1-56592-115-3.

Oaks, S. and Wong, H. (2009), *Java Threads*, 3rd, ISBN: 978-0-596-00782-9.

Oh, T.-Y., Chung, H., Cho, Y.-C., Ryu, J.-W., Lee, K., Lee, C., Lee, J.-I., Kim, H.-J., Jang, M. S., Han, G.-h., Kim, K., Moon, D., Bae, S., Park, J.-Y., Ha, K.-S., Lee, J., Doo, S.-Y., Shin, J.-B., Shin, C.-H., Oh, K., Hwang, D., Jang, T., Park, C., Park, K., Lee, J.-B., and Choi, J. S. (2014), "A 3.2Gb/s/pin 8Gb 1.0V LPDDR4 SDRAM with integrated ECC engine for sub-1V DRAM core operation," in *International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC, IEEE, pp. 430–431, ISBN: 978-1-4799-0918-6. DOI: 10.1109/ISSCC.2014.6757500.

Ohsawa, T., Kai, K., and Murakami, K. (1998), "Optimizing the DRAM Refresh Count for Merged DRAM/Logic LSIs," in *Proceedings of the 1998 International*

Symposium on Low Power Electronics and Design, ISLPED '98, ACM, pp. 82–87, ISBN: 978-1-58113-059-1. DOI: 10.1145/280756.280792.

OpenMP Architecture Review Board (2005), "OpenMP Application Programming Interface," Specification, Version 2.5.

OpenMP Architecture Review Board (2008), "OpenMP Application Programming Interface," Specification, Version 3.0.

OpenMP Architecture Review Board (2013), "OpenMP Application Programming Interface," Specification, Version 4.0.

Pan, A. and Pai, V. S. (2015), "Runtime-driven Shared Last-level Cache Management for Task-parallel Programs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM, 11:1–11:12, ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807625.

Papaefstathiou, V., Katevenis, M. G., Nikolopoulos, D. S., and Pnevmatikatos, D. (2013), "Prefetching and Cache Management Using Task Lifetimes," in *Proceedings of the 27th International Conference on Supercomputing*, ICS '13, ACM, pp. 325–334, ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465443.

Paul, S., Cai, F., Zhang, X., and Bhunia, S. (2011), "Reliability-Driven ECC Allocation for Multiple Bit Error Resilience in Processor Cache," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 20–34, ISSN: 0018-9340. DOI: 10.1109/TC.2010.203.

Planas, J., Badia, R. M., Ayguadé, E., and Labarta, J. (2013), "Self-Adaptive OmpSs Tasks in Heterogeneous Environments," in *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS, pp. 138–149, ISBN: 978-1-4673-6066-1. DOI: 10.1109/IPDPS.2013.53.

Qureshi, M. K., Kim, D.-H., Khan, S., Nair, P. J., and Mutlu, O. (2015), "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," in *45th International Conference on Dependable Systems and Networks*, DSN 2015, pp. 427–437. DOI: 10.1109/DSN.2015.58.

Reed, I. S. (1954), "A class of multiple-error-correcting codes and the decoding scheme," *Transactions of the IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 38–49, ISSN: 2168-2690. DOI: 10.1109/TIT.1954.1057465.

Restle, P. J., Park, J. W., and Lloyd, B. F. (1992), "DRAM variable retention time," in *International Electron Devices Meeting Technical Digest*, IEDM, pp. 807–810. DOI: 10.1109/IEDM.1992.307481.

Richards, D. and Jaulmes, L. (2014), "CoMD in Chapel: The Good, the Bad, and the Ugly," in *Chapel Lightning Talks*, Birds-of-a-Feather session at SC'14.

Rico, A., Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M. (2012), "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, ISSN: 1544-3566. DOI: 10.1145/2086696.2086715.

Rico, A., Duran, A., Cabarcas, F., Etsion, Y., Ramirez, A., and Valero, M. (2011), "Trace-driven simulation of multithreaded applications," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, pp. 87–96. DOI: 10.1109/ISPASS.2011.5762718.

Saad, Y. (2003), *Iterative Methods for Sparse Linear Systems*, ser. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics, ISBN: 978-0-89871-534-7. DOI: 10.1137/1.9780898718003.

Sánchez Barrera, I., Moretó, M., Ayguadé, E., Labarta, J., Valero, M., and Casas, M. (2018), "Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies," in *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, ACM, pp. 207–217, ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205310.

Sanchez, D., Yoo, R. M., and Kozyrakis, C. (2010), "Flexible Architectural Support for Fine-grain Scheduling," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, ACM, pp. 311–322, ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736055.

Schuster, S. E. (1978), "Multiple Word/Bit Line Redundancy for Semiconductor Memories," *IEEE Journal of Solid-State Circuits*, vol. 13, no. 5, pp. 698–703, ISSN: 0018-9200. DOI: 10.1109/JSSC.1978.1051122.

Seol, H., Shin, W., Jang, J., Choi, J., Suh, J., and Kim, L.-S. (2016), "Energy Efficient Data Encoding in DRAM Channels Exploiting Data Value Similarity," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, ISCA '16, pp. 719–730. DOI: 10.1109/ISCA.2016.68.

Servat, H., Llort, G., González, J., Giménez, J., and Labarta, J. (2014), "Identifying Code Phases Using Piece-Wise Linear Regressions," in *Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS, pp. 941–951. DOI: 10.1109/IPDPS.2014.100.

Servat, H., Llort, G., González, J., Giménez, J., and Labarta, J. (2015), "Low-Overhead Detection of Memory Access Patterns and Their Time Evolution," in *European Conference on Parallel Processing*, Euro-Par 2015, Springer, pp. 57–69, ISBN: 978-3-662-48095-3. DOI: 10.1007/978-3-662-48096-0_5.

Shewchuk, J. R. (1994), "An introduction to the conjugate gradient method without the agonizing pain," School of Computer Science, Carnegie Mellon University, Computer Science Technical Report CMU-CS-94-125.

Sinharoy, B., Norstrand, J. A. V., Eickemeyer, R. J., Le, H. Q., Leenstra, J., Nguyen, D. Q., Konigsburg, B., Ward, K., Brown, M. D., Moreira, J. E., Levitan, D., Tung, S., Hrusecky, D., Bishop, J. W., Gschwind, M., Boersma, M., Kroener, M., Kaltenbach, M., Karkhanis, T., and Fernsler, K. M. (2015), "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, vol. 59, no. 1, 2:1–2:21, ISSN: 0018-8646. DOI: 10.1147/JRD.2014.2376112.

Son, Y. H., Lee, S., O, S., Kwon, S., Kim, N. S., and Ahn, J. H. (2015), "CiDRA: A cache-inspired DRAM resilience architecture," in *IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA, pp. 502–513. DOI: 10.1109/HPCA.2015.7056058.

Sorin, D. J., Martin, M. M. K., Hill, M. D., and Wood, D. A. (2002), "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pp. 123–134. DOI: 10.1109/ISCA.2002.1003568.

Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K. B., Stearley, J., Shalf, J., and Gurumurthi, S. (2015), "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XX, ACM, pp. 297–310, ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694348.

Sridharan, V. and Liberty, D. (2012), "A Study of DRAM Failures in the Field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press, 76:1–76:11, ISBN: 978-1-4673-0804-5. DOI: 10.1109/SC.2012.13.

Stroustrup, B. (2013), *The C++ Programming Language*, 4th. Addison-Wesley Professional, ISBN: 978-0-321-56384-2.

Stuecheli, J., Kaseridis, D., C.Hunter, H., and John, L. K. (2010), "Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, IEEE Computer Society, pp. 375–384, ISBN: 978-0-7695-4299-7. DOI: 10.1109/MICRO.2010.22.

Subasi, O., Arias, J., Unsal, O., Labarta, J., and Cristal, A. (2015), "NanoCheckpoints: A Task-Based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, PDP '15, pp. 99–102. DOI: 10.1109/PDP.2015.17.

Subasi, O., Di, S., Bautista-Gomez, L., Balaprakash, P., Unsal, O., Labarta, J., Cristal, A., and Cappello, F. (2016), "Spatial Support Vector Regression to Detect Silent Errors in the Exascale Era," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pp. 413–424. DOI: 10.1109/CCGrid.2016.33.

Sunami, H. (2008), "The Role of the Trench Capacitor in DRAM Innovation," *IEEE Solid-State Circuits Society Newsletter*, vol. 13, no. 1, pp. 42–44, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2008.4785691.

Tam, D. K., Azimi, R., Soares, L. B., and Stumm, M. (2009), "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, ACM, pp. 121–132, ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508259.

Tan, X., Bosch, J., Jiménez-González, D., Álvarez-Martínez, C., Ayguadé, E., and Valero, M. (2016), "Performance Analysis of a Hardware Accelerator of Dependence Management for task-based Dataflow Programming Models," in *International Symposium on Performance Analysis of Systems and Software*, ISPASS, pp. 225–234. DOI: 10.1109/ISPASS.2016.7482097.

Tan, X., Bosch, J., Vidal, M., Álvarez, C., Jiménez-González, D., Ayguadé, E., and Valero, M. (2017), "General Purpose Task-Dependence Management Hardware for Task-Based Dataflow Programming Models," in *Proceedings of the IEEE 31st International Parallel and Distributed Processing Symposium*, IPDPS, pp. 244–253. DOI: 10.1109/IPDPS.2017.48.

Tang, D., Carruthers, P., Totari, Z., and Shapiro, M. W. (2006), "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *36th International Conference on Dependable Systems and Networks*, DSN 2006, IEEE, pp. 365–370. DOI: 10.1109/DSN.2006.13.

The MPI Forum (1993), "MPI: a message passing interface," in *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, SC '93, ACM, pp. 878–883, ISBN: 978-0-8186-4340-8. DOI: 10.1145/169627.169855.

Valero, M., Moreto, M., Casas, M., Ayguadé, E., and Labarta, J. (2014), "Runtime-Aware Architectures: A First Approach," *Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, ISSN: 2313-8734. DOI: 10.14529/jsfi140102.

Wang, Y., Wang, R., Herdrich, A., Tsai, J., and Solihin, Y. (2016), "CAF: Core to Core Communication Acceleration Framework," in *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques*, PACT '16, pp. 351–362, ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967954.

Weaver, C., Emer, J., Mukherjee, S. S., and Reinhardt, S. K. (2004), "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pp. 264–275. DOI: 10.1109/ISCA.2004.1310780.

Wijngaart, R. F. V. d. and Mattson, T. G. (2014), "The Parallel Research Kernels," in *IEEE High Performance Extreme Computing Conference*, HPEC, pp. 1–6. DOI: 10.1109/HPEC.2014.7040972.

Wong, M., Klemm, M., Duran, A., Mattson, T., Haab, G., Supinski, B. R. de, and Churbanov, A. (2010), "Towards an Error Model for OpenMP," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, IWOMP 2010, Springer, pp. 70–82, ISBN: 978-3-642-13216-2. DOI: 10.1007/978-3-642-13217-9_6.

Wulf, W. A. and McKee, S. A. (1995), "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, ISSN: 0163-5964. DOI: 10.1145/216585.216588.

Yang, X., Wang, Z., Xue, J., and Zhou, Y. (2012), "The Reliability Wall for Exascale Supercomputing," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 767–779, ISSN: 0018-9340. DOI: 10.1109/TC.2011.106.

Yoon, D. H. and Erez, M. (2010), "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, ACM, pp. 397–408, ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736064.

Yu, L., Li, D., Mittal, S., and Vetter, J. S. (2014), "Quantitatively Modeling Application Resilience with the Data Vulnerability Factor," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, IEEE Press, pp. 695–706, ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.62.

Yuan, L., Liu, H., Jia, P., and Yang, Y. (2015), "An adaptive ECC scheme for dynamic protection of NAND Flash memories," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP, pp. 1052–1055. DOI: 10.1109/ICASSP.2015.7178130.

Ziegler, J. F. and Lanford, W. A. (1979), "Effect of Cosmic Rays on Computer Memories," *Science*, vol. 206, no. 4420, pp. 776–788, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.206.4420.776.

List of Figures

List of Tables

Glossary

ABFT ..... Algorithmic-Based Fault Tolerance
ACE ..... Architecturally Correct Execution
AFEIR ..... Asychronous Forward Exact Interpolation Recovery
AVF ..... Architectural Vulnerability Factor [Mukherjee *et al.* 2003]
BCH ..... Bose–Chaudhuri–Hocquenghem error correcting code
BiCGStab ..... Bi-Conjugate Gradient Stabilised
BSC ..... Barcelona Supercomputing Center
CE ..... Corrected Errors
CG ..... Conjugate Gradient
DDDC ..... Double Device Data Correction
DDR ..... Double Data Rate
DEC ..... Double Error Correct
DIMM ..... Dual Inline Memory Module
DRAM ..... Dynamic Random-Access Memory
DUE ..... Detected Uncorrected Error
DVF ..... Data Vulnerability Factor [Yu *et al.* 2014]
EBB ..... Event-Based Branch
ECC ..... Error Correcting Code
ECP ..... Error Correcting Pointer
FEA ..... False Error Aware memory vulnerability factor
FEIR ..... Forward Exact Interpolation Recovery
FT-MPI ..... Fault-Tolerant MPI
GMRES ..... Generalised Minimal RESidual
GPR ..... General Purpose Register
GPU ..... Graphical Processing Unit
HPC ..... High-Performance Computing
LD ..... Load

LPDDR ..... Low Power DDR
MPI ..... Message Passing Interface
MSHR ..... Miss Status Handling Registers
MTBE ..... Mean Time Between Errors
MTBF ..... Mean Time Between Failures
MVF ..... Memory Vulnerability Factor
N-EC ..... N-Error Correct
OS ..... Operating System
PCG ..... Preconditioned CG
PE ..... Processing Element
PMU ..... Performance Monitoring Unit
RBS ..... Redundant Bit Steering
ROI ..... Region Of Interest
RS ..... Reed-Solomon error correcting code
SDC ..... Silent Data Corruption
SEC ..... Single Error Correct
SECDED ..... Single-Error Correct Double-Error Detect
SER ..... Soft Error Rate
SPD ..... Symmetric Positive Definite
SPR ..... Special-Purpose Registers
SSC ..... Single-Symbol Correct
ST ..... Store
TDG ..... Task Dependency Graph
TEC ..... Triple Error Correct
VRT ..... Variable Retention Time
WITSEC ..... WITSEC Is Targeted Stronger Error Correction
ZEC ..... Zero Error Correct