Efficient Fault Tolerance for Selected Scientific Computing Algorithms on Heterogeneous and Approximate Computer Architectures

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
und dem Stuttgart Research Centre for Simulation Technology
der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Alexander Schöll

aus Calw

Hauptberichter:	Prof. Dr. Hans-Joachim Wunderlich
Mitberichter:	Prof. Jie Han, PhD
Tag der mündlichen Prüfung:	16. July 2018

Institut für Technische Informatik
der Universität Stuttgart

2018

Dedicated to my fiancée Lena

Contents

List of Figures

List of Tables

Acknowledgments

During the past years, many people have contributed in different ways to the successful conclusion of this dissertation. It is my pleasure to thank those whose encouragement and support allowed me to grow and to fulfill my goals.

This work would not have been possible without the support of my supervisor, colleagues, and collaboration partners. I am grateful to Prof. Hans-Joachim Wunderlich for sharing his ideas and feedback in countless discussions that greatly contributed to this thesis. I would also like to thank Prof. Jie Han for the fruitful discussions during my stay at the University of Alberta in Canada, and for accepting to be second adviser of my thesis. I thank Prof. Joachim Groß for being my second adviser in the GS SimTech and the interesting discussion during my milestone presentation. It is also my pleasure to thank Claus Braun and Michael A. Kochte who always provided unconditional support and encouragement over all these years.

I am grateful to all my collaboration partners within the SimTech Cluster of Excellence at the University of Stuttgart. I thank Prof. Guido Schneider and Markus Daub for the successful and interesting collaboration. I would also like to thank my collaboration partners at the University of Alberta: Thank you Honglang Jiang, Yiding Liu, Mohammad Saeed Ansari, and Liu Siting. I enjoyed the work with colleagues who were at some time involved in my activities. It was a great pleasure to work with Ahmed Atteya, Rafał Baranowski, Laura Rodríguez Gómez, Zahra Najafi Haghi, Natalia Kaptsova, Chang Liu, Eric Schneider, Dominik Ull, and Marcus Wagner. In the past years, I had many opportunities to work with students who assisted me in my project work, especially Sebastian Brandhofer, Carlos Alberto Franco Salazar, Alisa Kuzmina, Valentin Mihalcut, Sagar Gurudas Nayak, Hisham Saadeddin, Muhammad Tarique Saleem, Amit Pattana Shetti, Stefan Simeonov, and Ediba Žugor. A work like this is not possible without outstanding administrative and technical expertise: Thank you, Mirjam Breitling, Helmut Häfner, and Lothar Hellmeier for your great assistance over the past years.

Words cannot express how grateful I am to my fiancée Lena for her caring support and all the sacrifices that she has made on my behalf.

Stuttgart, July 2018

Alexander Schöll

Abbreviations and Notation

Abbreviations

ABFT	Algorithm-based Fault Tolerance
CG	Conjugate Gradient method
CPU	Central Processing Unit
DWC	Duplication with Comparison
ECC	Error Detecting and Correcting Codes
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
MIPS	Millions of instructions per second
PCG	Preconditioned Conjugate Gradient
SIMD	Single instruction, multiple data
SpMV	Sparse matrix-vector
TMR	Triple Modular Redundancy
VLSI	Very Large Scale Integration

Linear Algebra Notation

$\mathbb{N}_0$	set of natural numbers with 0
$\mathbb{R}$	set of real numbers
$\mathbb{R}_{\neq 0}$	set of non-zero real numbers
$\mathbb{R}^n, \mathbb{R}^{m \times n}$	set of real vectors, and set of real matrices
$\boldsymbol{A}^T$	transpose operation
$a_{i,j}, [\boldsymbol{A}]_{i,j}$	(i, j)-element of matrix $\boldsymbol{A}$
$\boldsymbol{A}_k$	k-th row block matrix of $\boldsymbol{A}$
$[\boldsymbol{A}_k]_{i,j}$	(i, j)-element of k-th row block matrix $\boldsymbol{A}_k$
$[\boldsymbol{A}]_i$	i-th row of matrix $\boldsymbol{A}$
$\|\boldsymbol{x}\|_2$	Euclidean norm of vector $\boldsymbol{x}$
$\boldsymbol{r} \perp \boldsymbol{p}$	vector $\boldsymbol{r}$ is orthogonal to vector $\boldsymbol{p}$
$\boldsymbol{p}^{(i)}$	i-th instance of vector $\boldsymbol{p}$ (e.g., in iteration i of an iterative algorithm)

$fl(x)$	floating-point representation of x
ε_M	machine epsilon

Set Operator Notation

$\cup$	union
$\cap$	intersection
$\setminus$	difference

Boolean Operator Notation

$\neg$	negation
$\wedge$	conjunction
$\vee$	disjunction
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence

Abstract

Scientific computing and simulation technology play an essential role to solve central challenges in science and engineering. The high computational power of heterogeneous computer architectures allows to accelerate applications in these domains, which are often dominated by compute-intensive mathematical tasks. Scientific, economic and political decision processes increasingly rely on such applications and therefore induce a strong demand to compute correct and trustworthy results. However, the continued semiconductor technology scaling increasingly imposes serious threats to the reliability and efficiency of upcoming devices. Different reliability threats can cause crashes or erroneous results without indication. Software-based fault tolerance techniques can protect algorithmic tasks by adding appropriate operations to detect and correct errors at runtime. Major challenges are induced by the runtime overhead of such operations and by rounding errors in floating-point arithmetic that can cause false positives. The end of Dennard scaling induces central challenges to further increase the compute efficiency between semiconductor technology generations. Approximate computing exploits the inherent error resilience of different applications to achieve efficiency gains with respect to, for instance, power, energy, and execution times. However, scientific applications often induce strict accuracy requirements which require careful utilization of approximation techniques.

This thesis provides *fault tolerance* and *approximate computing methods* that enable the reliable and efficient execution of *linear algebra operations* and *Conjugate Gradient solvers* using heterogeneous and approximate computer architectures. The presented fault tolerance techniques detect and correct errors at runtime with low runtime overhead and high error coverage. At the same time, these fault tolerance techniques are exploited to enable the execution of the Conjugate Gradient solvers on approximate hardware by monitoring the underlying error resilience while adjusting the approximation error accordingly. Besides, parameter evaluation and estimation methods are presented that determine the computational efficiency of application executions on approximate hardware.

An extensive experimental evaluation shows the efficiency and efficacy of the presented methods with respect to the runtime overhead to detect and correct errors, the error coverage as well as the achieved energy reduction in executing the *Conjugate Gradient* solvers on approximate hardware.

Zusammenfassung

Wissenschaftliches Rechnen und Simulationstechnologie spielen eine wesentliche Rolle in der Lösung von zentralen Herausforderungen in Wissenschaft und Technik. Die hohe Rechenleistung von heterogenen Rechnerarchitekturen erlaubt es, Anwendungen in diesen Bereichen zu beschleunigen, welche oftmals von rechenintensiven mathematischen Aufgaben dominiert werden. Wissenschaftliche, wirtschaftliche und politische Entscheidungsprozesse stützen sich zunehmend auf solche Anwendungen und erfordern daher ausdrücklich die Berechnung von korrekten und vertrauenswürdigen Ergebnissen. Die zunehmende Miniaturisierung der Halbleiterelektronik konfrontiert jedoch zukünftige Schaltkreise mit ernsthaften Bedrohungen für die Zuverlässigkeit und Effizienz. Verschiedene Zuverlässigkeitsbedrohungen können Abstürze und fehlerhafte Ergebnisse verursachen, welche nicht signalisiert werden.

Software-basierte Fehlertoleranztechniken können algorithmische Aufgaben schützen, in dem sie diesen Algorithmen geeignete Operationen hinzufügen, welche Fehler zur Laufzeit erkennen und korrigieren. Große Herausforderungen werden durch die zusätzliche Laufzeit solcher Operationen und durch Rundungsfehler hervorgerufen, welche in Gleitkommaarithmetik auftreten und zu falsch-positiven Erkennungen führen können. Das Ende der Dennard-Skalierung (*engl. Dennard Scaling*) führt zu zentralen Herausforderungen für die weitere Steigerung der Recheneffizienz zwischen Technologiegenerationen. Approximierendes Rechnen (*engl. Approximate Computing*) nutzt die inhärente Fehlerresilienz verschiedener Anwendungen aus, um Effizienzsteigerungen gegenüber Leistungsaufnahme, Energie und Laufzeiten zu erreichen. Wissenschaftliche Anwendungen stellen jedoch oftmals strenge Anforderungen an die Genauigkeit von Ergebnissen, weshalb ein gewissenhafter Einsatz von Approximationstechniken notwendig ist.

Die vorliegende Arbeit stellt *Fehlertoleranz-* und *Approximationstechniken* vor, welche die zuverlässige und effiziente Ausführung von linearen Algebra Operationen und von *CG-Verfahren* auf heterogenen und approximativen Rechnerarchitekturen erlauben. Die vorgestellten Fehlertoleranztechniken erkennen und korrigieren Fehler zur Laufzeit mit geringer zusätzlicher Laufzeit sowie hoher Fehlerabdeckung. Gleichzeitig ermöglichen diese Fehlertoleranztechniken die Ausführung des CG-Verfahrens auf approximativer Hardware durch die Beobachtung der zugrundeliegenden Fehlerresilienz sowie der entsprechenden Anpassung des Approximationsfehlers. Daneben werden Methoden zur Bewertung und Schätzung von Parametern vorgestellt, welche die Recheneffizienz von Anwendungsausführungen auf approximativer Hardware bestimmen.

Eine ausführliche experimentelle Evaluierung zeigt die Effizienz und Effektivität der verschiedenen vorgestellten Methoden bezüglich der zusätzlichen Laufzeit zur Fehlererkennung und -korrektur, der Fehlerabdeckung sowie der erreichten Energiereduktion in der Ausführung des CG-Verfahrens auf approximativer Hardware.

Chapter 1

Introduction

Simulation technology and scientific computing play an essential role in the majority of scientific domains and have become established techniques to solve central challenges in these fields. The *explanatory* and *predictive power* of computer-based simulation for real-world systems and phenomena constitutes a sustained demand for short execution times [Oberk10, p.9] along with high reliability to obtain *trustworthy results* [Cappe14]. Today, these domains benefit from the compute power of *heterogeneous computer architectures*, which provide high computational performance within reasonable power envelopes [Chen15a,Gao16a]. Such computer architectures combine highly different kinds of processing cores including *multi-core CPUs*, *many-core GPUs* architectures as well as reconfigurable architectures like *field programmable gate arrays* (FPGA) along with communication channels and embedded memories on single chips or packages [Chung10]. Scientific applications are accelerated by *mapping* the different underlying algorithmic parts to matching components in these heterogeneous architectures, which can result in significant reductions of computation time [Lopez15]. The usage of heterogeneous computing architectures in the scientific and engineering domain continues to grow which is reflected in significantly increasing numbers of *high-performance computing* (HPC) systems that rely on these computer architectures [Gao16a].

Over the last decades, *continuous improvements in computer architecture* and *semiconductor technology scaling* have largely driven the increase in computational performance.

Moore's Law [Moore65] has impelled continuously new technology generations with doubling numbers of transistors on a single chip nearly every 18 months. Dennard scaling theory [Denna74] enabled this law as it allowed to increase the transistor density between generations while maintaining a proportional relationship between chip power and chip size. However, the continued technology scaling increasingly imposes challenges that constitute serious threats to the *reliability* and *efficiency* of upcoming semiconductor devices.

Modern nano-scaled semiconductor devices become increasingly vulnerable to a growing spectrum of different reliability threats [Mitra11, ITR] which can cause crashes or erroneous application results without indication. Reliability is a crucial demand of scientific applications since they are required to provide correct and trustworthy results. Future manufacturing processes will allow even smaller chip feature sizes, which makes the integration of efficient and effective *fault tolerance techniques* [Avizi04, Koren07] mandatory. Fault tolerance techniques enable a system to ensure its correct service according to the system specification in the presence of faults. These techniques can be applied to different layers of the system stack ranging from the hardware to the software and application layer, and typically exploit different forms of redundancy [Pradh96]. At the circuit and device layer, different hardware-based fault tolerance techniques including structural, temporal, or information redundancy, as well as self-checking, allow to protect hardware units against different kinds of faults. A widely-used form of information redundancy comprises error detecting and correcting codes that are, for instance, used to protect communication channels and memories. However, these fault tolerance measures are often associated with significant area and energy overheads that may even reduce the system performance. Software-based fault tolerance techniques [Pullu01] target different system layers including operating systems, middleware layers and algorithmic tasks in applications. Different techniques were proposed, which protect the processed data and the program control flow by targeting faults that manifest themselves as errors at these layers. These techniques include replication of computations and data, assertions and embedded signatures for control flow protection as well as different *algorithm-based fault tolerance* (ABFT) [Huang84] schemes to protect different computational tasks. A central challenge in integrating such software-based measures lies in the runtime and energy overhead that is induced by additional operations.

With the *end of Dennard scaling* [Esmae13], subthreshold leakage currents create a power density problem that does not allow anymore to scale the power per transistor at the same rate as the transistor dimensions. Without slowing down or fixing scaling parameters like frequency and supply voltage between technology generations, the power density can grow exponentially which induces unacceptable increases in chip power dissipation and thermal issues. The resulting *power* and *efficiency wall* [Flich16] mainly constituted the rise of multi-core and many-core architectures in the mid-2000s. To overcome the efficiency wall, it is not sufficient to only increase the performance of modern computer architectures by, for instance, increasing the number of cores or computational units on a chip [Esmae13]. The approximate computing paradigm [Han13, Venka15] allows to trade-off precision for efficiency gains with respect to power, energy, execution times, computational performance, and chip area. This computing paradigm targets different efficiency-cost parameters, such as the *power-delay product* of circuits and the *energy-time product* of applications [Kaesl14, p.96]. Applications in multimedia and signal processing, for instance, are often not expected to compute *perfect* results and therefore exhibit a significant error resilience to certain numerical errors. Approximation techniques exploit this inherent error resilience to achieve reductions in runtime, area, power, and energy demand. Different concepts have been proposed that extend the heterogeneous computing paradigm by exploiting approximation techniques for efficiency gains. Such *heterogeneous and approximate computer architectures* combine approximate memories and processing elements with their precise counterparts [Esmae12a, Chand17] and offer error monitoring and compensation at different layers of the system stack [Venka13a].

Fast, efficient, and fault-tolerant computing techniques are essential demands of the scientific computing domain that is dominated by compute-intensive tasks. With energy being a constraining factor, the approximate computing paradigm is a promising solution to tackle upcoming and future energy challenges. A central challenge in extending the application field of approximation techniques to the area of scientific computing is constituted by the demand for correct and trustworthy results. Scientific applications are often not necessarily error-tolerant and induce rather strict requirements on the accuracy of computational results which requires careful utilization of approximation techniques to achieve efficiency gains.

1.1 Contributions of this Thesis

This thesis presents *fault tolerance* and *approximate computing methods* that enable the fault-tolerant and efficient execution of *linear algebra operations* and *Conjugate Gradient solvers* using heterogeneous and approximate computer architectures. These scientific computing algorithms are essential parts of many large-scale applications in science and engineering and are often accelerated by heterogeneous computer architectures. The *approximate computing methods* execute these algorithms on approximate hardware and exploit the presented *fault tolerance* techniques to ensure correct results with low runtime overhead. Besides, this work discusses essential related approaches that also target fault tolerance and approximate computing for scientific and engineering tasks.

A major challenge in ensuring the fault-tolerant execution of scientific computing algorithms is constituted by the performance loss that can be induced by compute-intensive error detection and correction schemes. The fault tolerance techniques presented in this work are algorithm-based and exploit different properties of algorithms to ensure the effective detection and correction of erroneous results with low runtime overhead. The runtime overhead induced by the presented techniques scales with increasing problem size.

The presented methods in this thesis are summarized as follows:

Efficient fault-tolerant sparse matrix-vector multiplications

A technique is presented that enables the *fault-tolerant execution of sparse matrix-vector multiplications* on heterogeneous hardware by detecting and implicitly locating errors in the results, which provides efficient local correction regarding low runtime overhead and high error coverage. An error bound is presented that distinguishes harmful errors caused by, for instance, transient events from acceptable errors.

Efficient fault tolerance for the Conjugate Gradient solvers

Conjugate Gradient solvers are widely used in scientific and engineering applications and solve systems of linear equations iteratively. To ensure the *convergence of these solvers to correct results*, a fault tolerance technique is presented that detects errors with very low runtime overhead by periodically evaluating inherent solver properties.

Enabling the Conjugate Gradient solvers on approximate hardware

Different applications including *Conjugate Gradient* solvers exhibit an error resilience that may change in the course of the iterations. This changing error resilience, as well as the aforementioned tight accuracy demands, constitute major challenges to increase the compute efficiency of solver executions. To enable the *Conjugate Gradient* solvers on approximate computing hardware, an adaptive method exploits the previously addressed *fault tolerance technique* to detect and correct harmful approximation errors while controlling the underlying precision at runtime. The low iteration overhead induced by this fault tolerance technique to monitor intermediate computational results allows reduced energy demand while ensuring convergence to correct results.

Parameter estimation for application executions on approximate hardware
Different parameters must be determined to evaluate the compute efficiency of application executions on approximate computing hardware. These parameters comprise the area, the leakage power, the dynamic power, the delay, and the approximation error. To provide low parameter evaluation runtimes, three parameter estimation methods are presented that rely on circuit simulation-based techniques, model-based evaluations as well as the combination of both approaches. Different parameters are estimated by extrapolating *selected instruction intervals* to complete application executions.

The different presented methods were evaluated with respect to essential aspects including the performance overhead to detect and correct errors, the error coverage as well as the reduction in energy to execute *Conjugate Gradient* solvers on approximate hardware. The experimental evaluation shows the application of these methods while the associated benefits for scientific and engineering applications are discussed.

The scientific computing algorithms targeted in this thesis are categorized in the *sparse linear algebra computational class* [Asano06]. This computational class is widely-used in a large number of areas and continues to grow in importance. Areas in which such sparse linear algebra problems arise include structural mechanics [Smith13, p. 77], thermal engineering [Leng15], computational fluid dynamics [Wozni16], machine learning [Liu15a,Han16], the study of electromagnetic fields [Puzyr13,Dehiy17] as well as semiconductor power grid analysis [Feng10]. Large-scale sparse problems appear in these areas in the context of solving partial differential equations (PDEs), which are discretized by *finite element* or *finite difference methods* [Saad03, p.47]. Iterative methods like *Conjugate Gradient* solvers are well-known techniques to solve such complex problems and are preferred to *direct methods* like the Gaussian elimination [Golub13] since they are typically more efficient regarding computational performance and memory requirements.

At the same time, these linear algebra operations are parallelizable, which makes them well suited for heterogeneous computing systems comprising, for instance, multi-core CPUs and many-core GPUs. Recent works in this area exploit different characteristics of the underlying linear algebra operations to accelerate their execution using these computer architectures [Buato09, Ament10, Helfe12, Li13, Liu16a, Filip17].

1.2 Scientific Computing and Simulation Technology

The research in the science and engineering domains is complemented and propelled by *scientific computing and simulation technology*, which are often called the *third pillar of science* next to theory and experiment [Resch17, p.22]. The underlying computer-based modeling and simulation techniques have become essential means in the exploration and understanding of natural phenomena as well as in the solution of complex engineering problems. Their explanatory and predictive abilities allow to gain a deeper understanding of such phenomena or enable new observations. At the same time, a growing number of problems in different fields constitute an increasing demand to complement or even substitute experiments by computer-based simulations since they are often faster, cheaper, safer and provide increased observability. Such *in-silico experiments* allow the investigation of problems that are infeasible or even impossible to solve by common experimental and theoretical approaches. Besides being time-consuming or highly expensive, different experiments can be associated with unacceptable risks to life and environment. Important examples include natural catastrophes like earth quakes [Boore14] and tropical cyclones [Kim14] as well as the global climate and weather [Hurre13].

To mimic such experiments using simulations, the underlying real-world systems and phenomena are described in models that comprise mathematical and algorithmic formulations. Simulation technology has become a multi-disciplinary domain that combines the models from natural sciences and engineering with the computational methods from numerical mathematics and computer science.

The transformation of scientific computing from a supportive tool into a *leading role* [Oberk10, p.4] demands models that describe real-world systems and phenomena with increasing level of detail [Keyes13]. Significantly increasing amounts of data and growing model complexities require scientific computations and simulations on very large scales. Heterogeneous computer architectures [Chung10] provide the necessary computer power to conduct such complex investigations with reasonable runtimes. The acceleration of complex applications on heterogeneous computer architectures has been widely used in the scientific and engineering computing domain and continues to gain in importance.

Scientific computing applications on heterogeneous computer architectures

The application runtime is a central aspect of scientific computing, which can induce limiting factors for scientific discovery. At the same time, the increasing demand to evaluate problems consisting of multiple interacting physics and phases in different scientific and engineering fields leads to significantly growing model complexities. The underlying *multi-physics*, *multi-phase* and *multi-scale* simulations benefit from the different *architectural strengths* that heterogeneous computer architectures provide.

The computational performance of heterogeneous computer architectures is enabled by the integration of highly diverse kinds of processing cores that close the gap between *serial* or *coarse-grained parallel* tasks and highly *data-parallel* tasks. One of the most widespread examples of heterogeneous computer architectures is the integration of multi-core CPU and many-core GPU architectures on single chips that exhibit highly different architectural features [Chung10, Mitta15]. Modern multi-core CPUs comprise a few tens of latency-optimized cores that offer complex pipeline techniques like *out-of-order multiple instruction* scheduling. In contrast, GPUs rely on large numbers of so-called *single instruction, multiple data* (SIMD) processing elements that are associated with smaller control units, which in return allowed integrating more processing elements. For this reason, such many-core GPUs are optimized for throughput-demanding applications.

To gain high performance from these computer architectures, the different architectural strengths must be leveraged by scientific and engineering applications. For instance, the simulation of multiple interacting physics, phases or scales allows to distinguish the underlying application into different algorithmic parts such as latency-sensitive, coarse-grained parallelizable, and fine-grained parallelizable parts. This mapping of applications to heterogeneous computer architectures can accelerate the application execution which allows reductions in execution runtime. A wide range of works report significant speedups by tailoring highly different applications from these computing domains to heterogeneous computer architectures:

The computational chemistry domain relies on these computer architectures to accelerate simulations of reacting flows [Xu12, Yonke16], molecular and quantum mechanics [Wu12] as well as molecular dynamics [Lashu12]. These applications rely in general on *n-body simulations* which are also applied to other domains including astrophysics [Bastr12]. A closely related important example are Markov-chain molecular Monte-Carlo simulations [Braun12a] that form a core task in thermodynamics and thermal process technology.

Heterogeneous computing has been widely exploited in the computational biosciences over the last decade to accelerate the investigation of biological processes and systems at different scales. The core tasks range from protein [Liu13a] and genome sequencing [Marti16] over the investigation of nervous systems [Hoang13] and biological modeling [Avram17] to the evaluation of biochemical signaling pathways [Braun12b, Schol14].

The investigation of the global climate and weather relies on modeling and predicting the physical, chemical, and biochemical states of the climate system as well as its evolution over time. Different multi-scale and multi-physics models, often called *Earth System Models* in this context are accelerated using heterogeneous computer architectures as presented in [Yang13a, Gan15]. Besides these atmospheric and oceanographic models [Song16], geophysical and seismic models [Cui13, Marti15, Gokhb16, Roten16] are accelerated to understand and predict geological processes like earthquakes.

Essential tasks in the electronic design automation (EDA) domain such as the design, validation, and verification of semiconductor devices rely on heterogeneous computer architectures to enable digital circuits with billions of transistors. A wide range of approaches evaluate such designs at different abstraction levels and map data-parallel simulation workloads to many-core GPUs while they perform scheduling and preprocessing tasks on multi-core CPUs. Important examples include system-level and register-transfer [Nanju10, Vinco12], gate-level [Chatt09, Holst15] and circuit-level simulators [Gulat09, Kapre09]. Besides, essential tasks like fault simulation [Gulat08, Kocht10, Schne16], power analysis [Holst12, Liu13b], and IR-drop estimations [Holst16] are tailored to heterogeneous computer architectures.

Computational structural mechanics (CSM) and computational fluid dynamics (CFD) play an increasing role in traditional engineering domains. Finite-element methods are accelerated on heterogeneous computer architectures to investigate the performance of complex structures and materials [Kessl15, Miao16, Ni16, Shen16]. Computational fluid dynamics methods often rely on numerical methods to investigate gas or liquid flows and heavily rely on heterogeneous computer architectures to solve the underlying the Navier-Stokes [Zabel15, Deng16, Liu16b] and Lattice-Boltzmann [McClu14, Feich15, Valer17] equations. Besides, these numerical methods are applied in safety-critical domains like the aerospace domain to design airframes [Wang14] and jet turbines [Regul16, Gotti16].

In the data sciences, *data-intensive computing* is an emerging area that provides methods to process massive amounts of data in the order of terabytes or petabytes in size, which is often referred to as *Big Data* [Chen14]. Examples of such significant data sources can be found in the area of *particle physics*, for instance, in which laboratories like the *Large Hadron Collider* produce 30 petabytes of data per year [Casti15, p.8]. To solve the challenges that arise from data capturing, curating and analysis [Chen14], different techniques like data batch and stream processing [Chen12, Ranja14], as well as machine learning techniques are accelerated on heterogeneous computer architectures [Oh04, Catan08, Li15, Abadi16]. Besides CPU and GPU architectures, application-specific integrated circuits (ASICs) have been developed that specifically accelerate machine learning tasks. An important example is the so-called *Tensor Processing Unit* [Joupp17] that accelerates core operations in neural networks like matrix-vector multiplications and computing nonlinear functions (i.e., activation functions). At the same time, the coupling of computer architecture progress and machine learning constitutes novel machine learning applications that enable fault classification for semiconductor devices [Rodri16].

Sparse linear algebra operations like matrix multiplications and methods like the *Conjugate Gradient* solver are essential parts of the discussed applications. Besides, these operations and solvers are used in different benchmarks to evaluate the performance of HPC systems. For instance, the *High-Performance Conjugate Gradients* benchmark [Donga15] employs sparse matrix-vector multiplications and the *Preconditioned Conjugate Gradient* algorithm to rank HPC systems by solving a representative thermal engineering problem. Benchmark datasets like the Florida Sparse Matrix Collection [Davis11] comprise several thousand sparse matrices that represent real-world scientific and engineering problems.

The mapping of these sparse linear algebra operations to heterogeneous computer architectures enables fast execution, but such mappings are not sufficient to fulfill the demand for fault-tolerant and efficient computations required by the scientific and engineering domains. Instead, efficient and fault-tolerant variants of these sparse linear algebra operations have to be provided to achieve these goals. The next two sections below introduce the associated *reliability* and *efficiency* challenges and demands.

1.3 Reliability Challenges and Demands

Scientific computing is widely used in different *decision-making processes* to assess the reliability, robustness, and safety of products and technologies as well as the risk of large-scale public and private projects [Oberk10]. *Virtual prototyping* and *virtual testing* are two techniques with increasing importance that employ simulations in different product development phases to reduce the development cost and time. In contrast, the assessment of *high-consequence applications* relies almost entirely on scientific computing as corresponding experiments cannot be performed under representative conditions or impose severe risks and high costs. Such applications include simulations of geological operations like carbon sequestration [Namha16], hydraulic fracturing [Ehler17], underground deposition of nuclear waste [Verma15], and simulations of global climate change [Hurre13]. For this reason, the corresponding political and economic decision-making processes induce a strong demand for scientific computing and simulation technology to provide correct and trustworthy results. A different high-consequence application that is gaining attention is formed by *autonomous driving*, which is associated with very high reliability and safety requirements. This demand for reliable computations constitutes a major challenge as modern computing devices face an increasing number of reliability threats.

The growing spectrum of reliability threats is already a serious challenge for high-performance computing systems. For instance, the study in [Di Ma16] reports that while hardware only causes about 25% of system-wide outages in the *Blue Waters* supercomputer, the *mean time between failures* (MTBF) can be in the order of a few days. These reliability threats can manifest themselves in a large range of unacceptable application outcomes including significantly increased runtimes [Shant11] and visible errors such as numerical deviations from the expected correct result. At the same time, errors can corrupt application results without any indication, which result in Silent Data Corruptions (SDC) [Mukhe11].

Fault tolerance techniques can be employed to detect and correct such unacceptable effects of reliability threats. Due to the strong demand for high performance along with reliable application results in the scientific and engineering computing domains, the integration of effective fault tolerance techniques has become mandatory.

The spectrum of reliability threats ranges from *extrinsic* causes like manufacturing process variations over *intrinsic* causes like aging and wear-out to *environmental* effects such as the increasing susceptibility to transient events [Segur04]. At the same time, the impact of these reliability threats is expected to increase as future manufacturing processes will allow even smaller chip feature sizes resulting in an increased vulnerability to such threats.

Manufacturing Manufacturing-induced process variability can cause the physical and electrical device parameters to deviate from their nominal specifications, which can result in erroneous functional behavior [Borka05, Shin16]. Different device parameters are affected by process variability including the gate width, the threshold voltage, the channel length, as well as the oxide thickness. As the wavelength of light that is used for the lithography process is increasingly exceeding the feature sizes, sub-wavelength lithography variations occur that result in geometrical variations. The so-called line edge roughness (LER) is exhibited in form of randomly varied edges of gate patterns which are caused by fluctuating effects like photon flux variations or the random walk nature of acid diffusion during photoresist removal. Random dopant fluctuations are caused by variations in the impurity atom implantation phase, which can change the threshold voltage. While manufacturing tests are used to identify and filter out defective devices after fabrication, early-life failures and latent defects can induce transient and intermittent faults in the course of the device lifetime.

Device lifetime In the course of the CMOS device lifetime, device parameter variations [Becke10, Mukhe11] continue to occur. Such dynamic variations are caused by aging and stress mechanisms like negative bias temperature instability (NBTI), time-dependent dielectric breakdown (TDDB), electromigration, and hot carrier injection (HCI), and can lead to erroneous functional behavior and performance variability over time. NBTI-induced aging increases the threshold voltage of PMOS transistors, which is caused by applying negative bias voltages at increased temperature. Time-dependent dielectric breakdown (TDDB) is caused by the formation of conducting paths through the gate oxide to the substrate which results in a reduced device oxide insulation. Electromigration is the transport of metal atoms caused by high current densities, which can lead to extrusions or voids that manifest opens, shorts and bridges. Hot carrier injections (HCI) are caused when carriers attain sufficient energy to be injected into the substrate and collisions with substrate atoms cause electron/hole pairs.

Environmental threats Different reliability threats originate from the device environment [Choi09], which includes electrical noise and different kinds of radiation. Radiation-induced reliability threats include ionized particle strikes from heavy ions, neutrons or alpha particles and can cause additional charges. These charges can change the logic state of a circuit by, for instance, switching transistors for short periods of time [Nicol11,Ferle13]. While single-event transients (SET) cause voltage glitches in combinational logic that become bit errors when captured in latches or memory elements, single-bit upsets (SBU) cause bit-flips within a latch or memory element. The number of bit flips depends upon the charge intensity generated by the particle strike which can affect almost all parts of modern CMOS circuits. The actual impact depends on the physical and electrical properties of the semiconductor material which constitutes the critical charge required to induce a bit flip. Shrinking transistor geometries and increasing power densities lead to reduced critical charges, which constitute an increased vulnerability to such transient events in upcoming CMOS devices.

The investigation of fault tolerance techniques is an active research area as its integration has become mandatory. However, different challenges for the integration of fault tolerance arise on the hardware as well as the software level. Different hardware-based fault tolerance approaches rely on different forms of redundancy [Pradh96,Koren07] or apply guard banding to mask errors [Weste15]. However, such techniques are often associated with significant area and energy overheads that are not suitable for highly integrated solutions. Therefore, a growing number of effects caused by transient events will be exposed to the software which has to tolerate them. Future software applications must be capable of detecting errors as well as recovering from them.

On the software level, compute-intensive algorithms from scientific and engineering computing are often designed to provide maximum performance. A central challenge in integrating software-based fault tolerance techniques lies in the runtime and energy overhead that is induced by additional operations. Only techniques with low overheads are suitable to satisfy the performance demands of scientific and engineering computing.

Traditional checkpointing techniques have become a mature approach to tolerate errors in such applications [Pullu01, Herau16]. In general, *checkpointing* techniques write the state of an application periodically to a storage component and restart the application from a prior state if an error is detected. However, such techniques can induce large recovery cost in both transferring checkpoint data and recomputing lost results for high error rates. While these costs might be acceptable when errors are rare, they can become infeasible in the near future with smaller chip feature sizes that can cause increasing error rates. Therefore, checkpointing techniques will become increasingly impractical as they induce significant bottlenecks for the execution of applications [Sloan13, Suraa14, Liu16c].

Fault tolerance techniques that protect the program control-flow rely on assertions and embedded signatures [Oh02], for instance, and avoid fetching and executing incorrect instructions during the program execution. However, these techniques are not able to detect errors that occur in arithmetic computing units, which can corrupt the application result.

Algorithm-based fault tolerance (ABFT) techniques [Huang84] encode input data by adding checksums before performing a linear operation and calculate new checksums for the results. The results are checked for errors by evaluating invariants between checksums that were processed by the operation and the checksums computed for the results. While different ABFT approaches can be highly efficient for *dense* linear algebra operations such as matrix multiplications or decompositions, they can induce significant runtime overheads when they are used to protect *sparse* linear algebra operations. Instead, efficient fault tolerance techniques are required for sparse matrices.

1.4 Efficiency Challenges and Demands

While scientific and engineering computing benefits from the progress in semiconductor scaling and computer architecture, these domains also increasingly face major challenges, which are constituted by the *power* and *efficiency wall*. These challenges affect the high-performance computing domain, in which both the power and energy demand have significantly increased over the last decade and become constraining factors for the design of systems that provide increased performance [Borka10]. Today's most powerful high-performance computing systems are often associated with an annual energy cost that exceeds the acquisition cost of the systems [Subra13, Mitta16a].

To allow the exploration of upcoming scientific and engineering problems with growing complexities, the *computational efficiency* increasingly becomes a central objective besides the computational performance. The constraining factors in the different computational efficiency parameters have to be tackled, which include the *computational runtime*, the *power dissipation*, the *energy demand*, and the *chip area*.

Dennard scaling [Denna74] allowed successive semiconductor technology scalings with doubling numbers of transistors per unit area while the power dissipation stayed in proportion to the area for several decades. By reducing different physical features like doping concentrations and the gate oxide thickness as well as scaling the supply and threshold voltage proportionally to the geometrical dimensions of the transistor, the power density could be maintained constant. Different effects including direct quantum tunneling limit the further reduction of the gate oxide thickness without causing significantly increased leakage currents that result in growing power densities. Increased power densities, in turn, induce increased thermal energy per unit area, which needs to be dissipated to keep the device within its thermal limits. To ensure the correct operation of semiconductor devices, it is essential to operate these devices within a fixed power budget. This insight is often referred to as the *end of Dennard scaling* [Esmae13], which gave rise to different techniques and compute paradigms that target different computational efficiency parameters.

Since the power dissipation of semiconductor devices is a central aspect of the computational efficiency, the investigation of approaches that reduce the power dissipation became an active research field. Such *low-power techniques* target different levels of the system stack ranging from devices and circuits, over architectures to applications. On the device level, transistor technologies like fully depleted silicon-on-insulator FETs (FDSOI) [Beign13] and fin-based FETs (FinFETs) provide reduced leakage currents [Mishr11]. On the circuit level, controllable-polarity field-effect transistors enable the implementation of arithmetic functions [Gaill14] and power gates [Amaru13] with reduced physical resources. Different dynamic voltage and frequency scaling approaches are widely-used techniques on the architecture level, which change the power dissipation as well as the device performance at runtime [Semer02]. Fine-grain power management techniques allow to scale the voltage or frequency for single parts or regions of a device with respect to the resource demands of applications [Ranga09]. On the application level, energy-aware task scheduling and migration techniques minimize the power dissipation by trading off the number of processing units and the frequency of each processing unit [Hsu05]. A related concept relies on completely switching off different device components including processing cores and parts of the memory hierarchy [Esmae12b]. However, these low-power techniques are not sufficient enough to solve the computational efficiency challenges, as they are likely not able to keep up with the power efficiency demand required by semiconductor scaling [Shafi16].

The approximate computing paradigm does not only target the reduction of the power dissipation but instead allows to trade-off precision against individual or combinations of different computational efficiency parameters. The investigation of approximate computing techniques is an active research area and continues to gain in importance. A wide range of approximate computing techniques was proposed for different system stack layers that promise significantly improved computational performance combined with low power and energy demand. Important principles that constitute these approximation techniques include *task skipping* [Sidir11], which allows runtime and energy reductions, *precision reductions* [Jiang15], which result in reduced power dissipation as well as *data estimations* [Migue14], which allow energy reductions by, for instance, avoiding energy-intensive memory operations.

The usefulness and relevance of approximate computing is highly dependent on the spectrum and the number of applications that can benefit from it in the near future. The investigation of approximate computing techniques has often focused on applications that inherently provide some error tolerance or that origin from specialized benchmark collections, which can create a significant discrepancy to real-world application domains. At the same time, real-world applications from the scientific and engineering computing domains highly demand such efficiency gains that are promised by approximate computing. For this reason, the application scope of approximate computing has to be extended to these compute and power-intensive computing domains.

To fully utilize the benefits of approximate computing in general, different major challenges [Venka15] need be tackled to enable applications for approximate computing:

Definition of correct results and result quality As the notion of *acceptable results* constitutes the error resilience of an application, it is inevitable to establish quantitative definitions that allow to measure result quality and to determine result correctness. While different *error metrics* exist for multimedia and signal processing applications, *application-specific metrics* can reflect a wide range of inherent properties in applications that need to be satisfied to accept a result.

At the same time, such application-specific metrics can highly differ between different applications. Besides the definition of metrics, methods are required that ensure correct results by, for instance, monitoring intermediate results when approximate computing techniques are utilized. Such methods may cancel the efficiency gains by introducing additional operations into applications which induce significant runtime and energy overheads.

Significance of compute efficiency gains Different computations in applications offer a wide range of potentials for approximate computing regarding their impact on both the result quality and the compute efficiency. Therefore, it is an essential challenge to identify resilient and sensitive computations in applications as well as to determine the significance of their contribution to the overall compute efficiency. At the same time, different computations may exist in applications, which do not allow approximations. Such computations typically involve pointer arithmetic and control flow operations that may lead to, for instance, application crashes.

Changing error resilience and precision-configurability The error resilience of an application is not a static property, as it can change between different operations within the application. At the same time, the error resilience can depend on the input data as well as it can change over the course of the application execution. An important example are iterative solvers including the *Conjugate Gradient* solvers that exhibit an error resilience that may change in the course of the iterations. For this reason, approximate computing techniques are often required to configure the induced approximation error at runtime.

These challenges especially apply to scientific and engineering computing applications which are important components of *decision-making processes* that impose tight accuracy demands. Such applications often comprise highly different and interacting tasks that offer different opportunities to apply approximate computing. The evaluation of the overall compute efficiency for such complex applications can be associated with significant runtimes, which need to be reduced to determine the actual significance of achieved compute efficiency gains. The error resilience is a major challenge in scientific applications, which can change over the course of the execution as well as for different input data. Besides precision-configurable approximation techniques, efficient monitoring and adaption techniques are required that alter the underlying precision according to such a changing error resilience.

1.5 Outline

The remainder of this work is organized into seven chapters that are structured as follows: Chapter 2 introduces the necessary background and discusses the related work for the subsequent chapters. This includes a concise introduction to the scientific computing algorithms as well as their underlying sparse linear algebra operations that are targeted in this thesis. At the same time, it discusses the essential ideas and concepts from the fields of reliability and fault tolerance as well as heterogeneous and approximate computing.

Chapters 3, 4, 5, and 6 discuss the contributions presented in this thesis. Chapter 3 presents a fault tolerance technique for sparse matrix-vector multiplications that provides both low runtime overhead and high error coverage by implicitly locating errors during error detection steps for efficient error correction. Chapter 4 presents a fault tolerance technique for the *Conjugate Gradient* solvers that periodically evaluates inherent solver properties with low runtime overhead to detect errors. Chapter 5 presents an adaptive method that enables the *Conjugate Gradient* solvers on approximate computing hardware to obtain reduced energy demand while still ensuring correct results. Chapter 6 presents parameter estimation methods that evaluate different compute efficiency parameters for application executions on approximate computing hardware.

Chapter 7 presents and discusses the experimental evaluation of the methods presented in this thesis. Chapter 8 concludes this work, summarizes the obtained findings and discusses the achieved results. The appendices comprise additional material as well as extended experimental results.

Chapter 2

Background and Related Work

This chapter introduces the necessary background and discusses the related work for the subsequent chapters. The selected scientific computing algorithms, as well as their underlying sparse linear algebra operations, are discussed. Besides, this chapter presents important concepts for reliability and fault tolerance in heterogeneous computer architectures. The necessary background for the approximate computing paradigm is introduced upon which the presented approximate computing methods are built.

The corresponding literature is referenced where suitable.

2.1 Sparse Linear Algebra Operations

The contributions in this thesis focus on *sparse matrix operations* and *conjugate gradient solvers*, for which the necessary background is introduced in this section.

2.1.1 Dense and Sparse Matrix Operations

Matrix operations constitute essential computational tasks in many large-scale scientific and engineering applications. One of the most important operations is the matrix-vector multiplication [Golub13], which computes the product of an $(m \times n)$-matrix $\boldsymbol{A}$ and an $(n \times 1)$-*operand vector* $\boldsymbol{b}$ to obtain an $(m \times 1)$-*result vector* $\boldsymbol{r}$ with

$$\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b} \,. \tag{2.1}$$

An element r_i in the *result vector* $\boldsymbol{r}$ is computed as

$$r_i := \sum_{k=1}^{n} a_{i,k} \cdot b_k \,. \tag{2.2}$$

The computational complexity is determined by the number of elements in matrix $\boldsymbol{A}$ with $\mathcal{O}(n \cdot m)$. For quadratic matrices with $n = m$, the complexity is $\mathcal{O}(n^2)$. In comparison to *dense* matrices in which almost all values are non-zero, *sparse* matrices contain a significant portion of zero elements. In the *sparse matrix-vector* (SpMV) multiplication, this sparsity property is exploited to reduce the computational complexity by omitting unnecessary multiplications. With *NNZ* being the number of non-zero elements, the computational complexity is now $\mathcal{O}(NNZ)$. Instead of a quadratic complexity, this SpMV operation can be of linear complexity with $NNZ \approx n$.

Different fault tolerance techniques exploit the *associative property* of matrix-vector multiplication which is

$$\boldsymbol{w}^T(\boldsymbol{A}\boldsymbol{b}) = (\boldsymbol{w}^T\boldsymbol{A})\boldsymbol{b} \tag{2.3}$$

with $\boldsymbol{b}$ being an $(n \times 1)$-operand vector and $\boldsymbol{w}$ being an $(m \times 1)$-operand vector.

An $(m \times n)$-matrix $\boldsymbol{A}$ can be decomposed into *row block matrices* $\boldsymbol{A}_i$ by row-partitioning matrix $\boldsymbol{A}$ into m' submatrices with $1 \leq m' \leq m$. A row block matrix $\boldsymbol{A}_i$ is formed by the i-th submatrix with $1 \leq i \leq m'$. The *row block size* σ_i denotes the number of rows in the *row block matrix* $\boldsymbol{A}_i$. For all row block matrices of an $(m \times n)$-matrix $\boldsymbol{A}$, the sum of all row block sizes is the number of rows in $\boldsymbol{A}$:

$$\sum_{i=1}^{m'} \sigma_i = m \,. \tag{2.4}$$

Example 2.1: The (6×2)-matrix $\boldsymbol{A}$ is row-partitioned into three row block matrices $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, and $\boldsymbol{A}_3$ with row block sizes $\sigma_1 = \sigma_2 = \sigma_3 = 2$:

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \\ a_{5,1} & a_{5,2} \\ a_{6,1} & a_{6,2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \boldsymbol{A}_3 \end{bmatrix} \quad \Leftrightarrow \quad \begin{aligned} \boldsymbol{A}_1 &= \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \\ \boldsymbol{A}_2 &= \begin{bmatrix} a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{bmatrix} \\ \boldsymbol{A}_3 &= \begin{bmatrix} a_{5,1} & a_{5,2} \\ a_{6,1} & a_{6,2} \end{bmatrix} \end{aligned}$$

▸

Let $\boldsymbol{a}$ be a vector with m elements which is partitioned into m' subvectors. Analogously to the description of row block matrices, a *block vector* $\boldsymbol{a}_i$ is a vector formed by the i-th subvector with $1 \leq i \leq m'$.

Let $\boldsymbol{A}$ be an $(m \times n)$-matrix that is partitioned into m' row block matrices, let $\boldsymbol{b}$ be an $(n \times 1)$-*operand vector* and let $\boldsymbol{r}$ be an $(m \times 1)$-*result vector* that is partitioned into m' block vectors. The *block-based matrix-vector multiplication* performs the matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$ as

$$\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b} \quad \Leftrightarrow \quad \begin{bmatrix} \boldsymbol{r}_1 \\ \vdots \\ \boldsymbol{r}_{m'} \end{bmatrix} := \begin{bmatrix} \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_{m'} \end{bmatrix} \boldsymbol{b} \quad \Leftrightarrow \quad \begin{matrix} \boldsymbol{r}_1 := \boldsymbol{A}_1\boldsymbol{b} \\ \vdots \\ \boldsymbol{r}_{m'} := \boldsymbol{A}_{m'}\boldsymbol{b}. \end{matrix} \tag{2.5}$$

2.1.2 Conjugate Gradient Solvers

The *Conjugate Gradient* solvers form a group of algorithms which solve systems of linear equations with the form

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{2.6}$$

with $\boldsymbol{A}$ being a symmetric ($\boldsymbol{A} = \boldsymbol{A}^T$) and positive-definite ($\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} > 0$ with $\boldsymbol{x} \neq 0$) matrix [Saad03]. The vector $\boldsymbol{x}$ denotes the unknowns of the linear system while the right-hand side vector $\boldsymbol{b}$ denotes the constant terms. In the following, the solver method is introduced for real matrices and vectors, namely $\boldsymbol{A} \in \mathbb{R}^{n \times n}, \boldsymbol{b} \in \mathbb{R}^n$, and $\boldsymbol{x} \in \mathbb{R}^n$.

The underlying solver method was originally presented by M.R. Hestenes and E. Stiefel in 1952 [Heste52] while its effectiveness for large, *sparse* matrices was shown in the early seventies [Reid72] when it was formulated as an *iterative* method. Compared to direct methods like the Gaussian-Elimination, this iterative solver method finds a correct result typically faster. The group of Conjugate Gradient solvers is formed by the *Conjugate Gradient solver* (CG) and its variant, the *Preconditioned Conjugate Gradient solver* (PCG). The difference between these two methods lies in the application of a *preconditioner* $\boldsymbol{M} \in \mathbb{R}^{n \times n}$. Preconditioners can accelerate the solving performance significantly by transforming the underlying linear system $\boldsymbol{Ax} = \boldsymbol{b}$ in such a way that the original solution $\boldsymbol{x}$ is computed with a reduced number of solving steps.

The runtime complexity of these solvers depends on both the size *NNZ* and the condition number $\kappa(\boldsymbol{A})$ with $\mathcal{O}(NNZ \cdot \sqrt{\kappa(\boldsymbol{A})})$. The condition number $\kappa(\boldsymbol{A})$ of a symmetric, positive-definite matrix $\boldsymbol{A}$ can be computed as

$$\kappa(\boldsymbol{A}) := \frac{\lambda_{max}}{\lambda_{min}} \tag{2.7}$$

with $\lambda_{max}/\lambda_{min}$ being the ratio of the largest to the smallest eigenvalue. Preconditioners $\boldsymbol{M}$ can diminish the condition number $\kappa(\boldsymbol{A})$ by *indirectly* solving the original system $\boldsymbol{Ax} = \boldsymbol{b}$ as $\boldsymbol{M}^{-1}\boldsymbol{Ax} = \boldsymbol{M}^{-1}\boldsymbol{b}$. Favorably, the preconditioner matrix $\boldsymbol{M}$ resembles the inverse matrix of $\boldsymbol{A}$ such that $\boldsymbol{M}^{-1} \approx \boldsymbol{A}^{-1}$ and $\kappa(\boldsymbol{M}^{-1}\boldsymbol{A}) \ll \kappa(\boldsymbol{A})$. The actual operation to be performed depends on the type of preconditioner and does not necessarily have to include matrix inversions and matrix-vector operations.

Different variants of the PCG solver are established by the spectrum of preconditioning techniques that focus different application scopes [Benzi02]. The *Jabobi* preconditioner uses the diagonal of $\boldsymbol{A}$ to compose the preconditioner matrix $\boldsymbol{M}$ with $\boldsymbol{M}_{\text{Jacobi}} := \textbf{diag}(\boldsymbol{A})$. The application of this preconditioner results in a matrix-vector multiplication $\boldsymbol{M}_{\text{Jacobi}}^{-1}\boldsymbol{r} = \textbf{diag}(a_{11}^{-1}, \cdots, a_{nn}^{-1})\boldsymbol{r}$. This preconditioner has a memory complexity of $\mathcal{O}(n)$ and a runtime complexity of $\mathcal{O}(n)$. Different preconditioners like the *incomplete Cholesky factorization*, the *incomplete LU factorization*, or the *symmetric successive overrelaxation* invoke different and potentially more computationally intensive operations [Benzi02].

The Conjugate Gradient methods solve linear equations by representing the solution $\boldsymbol{x}$ as a combination of different vectors and scalars:

For a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, a set of vectors $V = \{\boldsymbol{v}^{(k)} \mid \boldsymbol{v}^{(k)} \in \mathbb{R}^n \wedge \boldsymbol{v}^{(k)} \neq 0\}$ is *A-orthogonal* if V satisfies

$$\boldsymbol{v}^{(k)}\boldsymbol{A}\boldsymbol{v}^{(j)} = 0 \quad \text{with } k \neq j. \tag{2.8}$$

A *search direction* $\boldsymbol{p}^{(k)}$ is a vector in a set of N mutually A-orthogonal vectors

$$P = \{\boldsymbol{p}^{(k)} \mid \boldsymbol{p}^{(k)} \in \mathbb{R}^n \wedge \boldsymbol{p}^{(k)} \neq 0\}. \tag{2.9}$$

Search directions $\boldsymbol{p}^{(k)}$ are computed by the Conjugate Gradient methods over the course of *solver iterations* k with $1 \le k \le N$. Let $\boldsymbol{x}^{(k)}$ be the *intermediate result* in solver iteration k. A *residual vector* $\boldsymbol{r}^{(k)}$ is the difference between the right-hand side vector $\boldsymbol{b}$ and the product of $\boldsymbol{A}$ and the intermediate result $\boldsymbol{x}^{(k)}$:

$$\boldsymbol{r}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} . \tag{2.10}$$

The *residual* $\delta^{(k)}$ is the euclidean norm of the residual vector in solver iteration k:

$$\delta^{(k)} := \|\boldsymbol{r}^{(k)}\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}\|_2 . \tag{2.11}$$

A set of search directions P forms a basis for $\mathbb{R}^n$, which allows to represent the solution $\boldsymbol{x}$ as a linear combination based on an *initial guess* vector $\boldsymbol{x}^{(0)} \in \mathbb{R}^n$

$$\boldsymbol{x} = \boldsymbol{x}^{(0)} + \sum_{k=0}^{N} \alpha^{(k)} \boldsymbol{p}^{(k)} \tag{2.12}$$

in which $\alpha^{(k)}$ is computed as

$$\alpha^{(k)} := \frac{\boldsymbol{p}^{(k)}\boldsymbol{r}^{(k)}}{\boldsymbol{p}^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)}} \tag{2.13}$$

to ensure optimal step sizes [Saad03]. In case a preconditioner $\boldsymbol{M}$ is applied, $\alpha^{(k)}$ is computed as

$$\alpha^{(k)} := \frac{\boldsymbol{p}^{(k)}\boldsymbol{M}^{-1}\boldsymbol{r}^{(k)}}{\boldsymbol{p}^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)}} . \tag{2.14}$$

In each solver iteration k, an *intermediate result* $\boldsymbol{x}^{(k)}$ is computed from the previous iteration as

$$\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)} . \tag{2.15}$$

Based on the $\boldsymbol{A}$*-orthogonality* between *search directions*, different *inherent relations* exist between the vectors used in the Conjugate Gradient Solvers. For two different iterations j and k in a Conjugate Gradient solver the underlying vectors have the following relations:

$$\boldsymbol{p}^{(j)} \perp \boldsymbol{A}\boldsymbol{p}^{(k)} \qquad \text{for } j \neq k \tag{2.16}$$

$$\boldsymbol{r}^{(j)} \perp \boldsymbol{p}^{(k)} \qquad \text{for } j > k \tag{2.17}$$

$$\boldsymbol{r}^{(j)} \perp \boldsymbol{r}^{(k)} \qquad \text{for } j \neq k \tag{2.18}$$

The interested reader can find the proof for Equations 2.16 to 2.18 in [Golub13], Section 11.3.

The fundamental operations of the PCG solver are shown in Algorithm 1 in pseudo code. For completeness, the CG solver algorithm is described in Appendix A.1. To make a clear distinction between the error induced by approximate hardware and the outcomes of PCG iterations, the term *intermediate result* is used to address $\boldsymbol{x}$, which is often referred to as *approximation* $\boldsymbol{x}$ in literature. The two primary parts of the PCG algorithm are the *preparation of PCG* ranging between Lines 1 and 5 and the *PCG loop* ranging between Lines 6 and 17. The *preparation of PCG* initializes auxiliary vectors based on the initial guess $\boldsymbol{x}^{(0)}$. Based on the initial guess vector $\boldsymbol{x}^{(0)}$, each iteration of the *PCG loop* provides an improved intermediate result $\boldsymbol{x}^{(k)}$ with respect to the exact solution. PCG iterations are executed until an intermediate result $\boldsymbol{x}^{(k)}$ is found with a residual $\delta^{(k)}$ that satisfies the accuracy bounds defined by $\epsilon \in (\mathbb{R}, \mathbb{R}) := (\epsilon_a, \epsilon_r)$. While the *absolute accuracy tolerance* ϵ_a only considers the norm of the residual $\|\boldsymbol{r}^{(k)}\|_2$ to check an intermediate result, the *relative accuracy tolerance* ϵ_r is scaling-invariant which makes the number of required PCG iterations independent from initial guess vector $\boldsymbol{x}^{(0)}$.

The memory complexity of the CG solver is $\mathcal{O}(NNZ)$ with *NNZ* being the number of non-zero elements in $\boldsymbol{A}$. At the same time, the memory complexity of the PCG solver is at least $\mathcal{O}(NNZ)$ and depends on the memory complexity of the utilized preconditioner.

In theory, the Conjugate Gradient solvers converge in a finite number of iterations to the solution. Since these solvers are typically performed using floating-point arithmetic, rounding errors can occur. Over the iterations, rounding errors can accumulate in the search direction vectors $\boldsymbol{p}^{(k)}$ which can cause them to lose $\boldsymbol{A}$-orthogonality. For this reason, the orthogonalities presented above in Equations 2.16 to 2.18 are in practice only *approximately orthogonal*. At the same time, the residual $\boldsymbol{r}^{(k)}$ calculated by the

```
Input: $\boldsymbol{A}$,$\boldsymbol{M}$,$\boldsymbol{b}$, $\boldsymbol{x}^{(0)}$, $\epsilon_a$, $\epsilon_r$, $k_{max}$
Output: The result of solving the system $\boldsymbol{Ax} = \boldsymbol{b}$: $\boldsymbol{x}^{(k+1)}$
Data: $\boldsymbol{p}^{(k)}$, $\boldsymbol{r}^{(k)}$, $\boldsymbol{s}^{(k)}$, $\delta^{(k)}$
/* Preparation of PCG                                          */
1  $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{b} - \boldsymbol{Ax}^{(0)}$ ;          // Initial residual vector
2  $\boldsymbol{s}^{(0)} \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}^{(0)}$;                  // Preconditioning
3  $\boldsymbol{p}^{(0)} \leftarrow \boldsymbol{s}^{(0)}$;                  // Initial search direction
4  $\delta^{(0)} \leftarrow \boldsymbol{r}^{(0)T}\boldsymbol{r}^{(0)}$;                  // Initial residual
5  $k \leftarrow 0$ ;                                   // Iteration count

/* PCG loop                                                      */
6  while $(\delta^{(k)} > \epsilon_a^2) \wedge (\delta^{(k)}/\delta^{(0)} > \epsilon_r^2) \wedge (k < k_{max})$ do
7      $\boldsymbol{w}^{(k)} \leftarrow \boldsymbol{Ap}^{(k)}$;
8      $\gamma \leftarrow \boldsymbol{r}^{(k)T}\boldsymbol{s}^{(k)}$;
9      $\alpha \leftarrow \frac{\gamma}{\boldsymbol{p}^{(k)T}\boldsymbol{w}^{(k)}}$;
10     $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)}$;      // Next intermediate result
11     $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{r}^{(k)} - \alpha\boldsymbol{w}^{(k)}$;        // Update residual vector
12     $\boldsymbol{s}^{(k+1)} \leftarrow \boldsymbol{M}^{-1}\boldsymbol{r}^{(k+1)}$;                // Preconditioning
13     $\delta^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)T}\boldsymbol{r}^{(k+1)}$;              // Update residual
14     $\beta \leftarrow \frac{\boldsymbol{r}^{(k+1)T}\boldsymbol{s}^{(k+1)}}{\gamma}$;
15     $\boldsymbol{p}^{(k+1)} \leftarrow \boldsymbol{s}^{(k+1)} + \beta\boldsymbol{p}^{(k)}$;          // New search direction
16     $k \leftarrow k + 1$;
17 end
```

Algorithm 1: The Preconditioned Conjugate Gradient (PCG) algorithm.

Conjugate Gradient algorithm and the *true residual*

$$\boldsymbol{r}_{\text{true}}^{(k)} := \boldsymbol{b} - \boldsymbol{Ax}^{(k)} \tag{2.19}$$

can increasingly deviate over the iterations due to rounding errors [Cools16]. This deviation is induced by the property that the true residual is never calculated during the iterations in the Conjugate Gradient solvers. Instead, the residual is computed from the recurrence $\boldsymbol{r} = \boldsymbol{r}^{(0)} + \sum_{k=0}^{N} -\alpha^{(k)}\boldsymbol{Ap}^{(k)}$ to reduce the number of matrix-vector multiplications per iteration to one. Due to both the possible loss of $\boldsymbol{A}$-orthogonality and the deviation between true and recursive residuals, the required number of iterations can be significantly increased or the solver may return a wrong result. These properties have to be considered when developing a fault tolerance technique with high error coverage.

The applicability of the Conjugate Gradient Solvers is not necessarily limited to symmetric and positive-definite matrices. To solve the normal equations, both sides of the original equation $\boldsymbol{Ax} = \boldsymbol{b}$ can be pre-multiplied by $\boldsymbol{A}^T$ to solve $\boldsymbol{A}^T\boldsymbol{Ax} = \boldsymbol{A}^T\boldsymbol{b}$. As long as $\boldsymbol{A}$ is non-singular, $\boldsymbol{A}^T\boldsymbol{A}$ is symmetric and positive-definite which allows to exploit the aforementioned solver properties.

The Conjugate Gradient solvers as well as matrix-vector operations constitute compute-intensive tasks in the scientific and engineering computing domain which are often accelerated using heterogeneous computer architectures. Since these modern computing devices become increasingly vulnerable to different reliability threats, the integration of fault tolerance techniques is a mandatory prerequisite. Section 2.2 introduces the formal foundations on reliability and fault tolerance while Section 2.3 discusses related fault tolerance techniques.

2.2 Reliability and Fault Tolerance

This section introduces the definitions and concepts in the area of *reliability and fault tolerance* that is used in the subsequent chapters. The vulnerability of modern CMOS devices is shown using recent assessment results from the literature. This introduction is complemented by discussing the concepts of fault tolerance techniques while presenting the underlying ideas of the fault models that are applied in the experimental evaluations of this thesis.

2.2.1 Definitions

The definitions in this section follow the taxonomy of Avizienis et al. and other authors [Avizi04,Cappe14,Pullu01]. The basic entity in this taxonomy is the *system* which is able to communicate and interact with other systems in its *environment*. An essential property of a system is its functionality that is described by a *system function*. The steps that a system performs to implement its function constitute its *behavior* and can be described as a sequence of internal and external states. The behavior that is perceivable by the other systems in the environment forms the *service* it is delivering.

The delivered service is called a *correct service* when it meets the specified system function. The case in which the delivered service deviates from the system function is called a *failure*. Such deviations are called *errors* and are manifested by a set of external states that differ from the expected correct set of states. Errors are called *detected errors* when the system identifies and signals their presence. The cause of an error is called a *fault*. Hardware faults can result from *defects* that describe distortions or imperfections in the physical structure of the underlying hardware. Besides manufacturing defects, different defect mechanisms exist such as aging or radiation that increase the probability of occurring defects. At the same time, hardware faults can be caused by environmental *transient events* like particle hits (e.g., neutron and alpha particles).

The different of types of faults can be classified according to their persistence into *permanent*, *intermittent* and *transient* faults. A permanent fault is persistent and continues to exist until the fault is repaired. An intermittent fault occurs not continuously but at irregular time intervals. A transient fault occurs once for a short period of time and then typically disappears. Besides their persistence, faults can be classified according to different categories including the *phase of creation, the point of origination, the phenomenological cause, the dimension, the objective,* and the *capability.*

The term *dependability* comprises a global concept that can be determined and measured through the three elements *attributes*, *means to attain dependability*, and *threats*. The attributes comprise availability, reliability, safety, integrity, and maintainability. As one of the most important attributes, the *reliability* describes the probability that a system provides its specified correct service for a certain period of time. The formal definition for the reliability of a system is described in Appendix B. While reliability is one attribute to describe dependability, *fault tolerance* is a means to attain dependability. Fault tolerance techniques try to ensure the correct service according to the system specification in the presence of faults. These techniques are intended to detect and correct errors during operation of the system. This intention is achieved by detecting and notifying about the presence of errors and by recovering, or by compensating for errors by, for instance, using redundancy. Error recovery can include rollback schemes, where the system state is returned to a previous correct state which can be a checkpoint. At the same time, error recovery can rely on roll-forward techniques in which new correct states are created. Besides tolerating faults, dependability is attained by *predicting*, *preventing*, or *removing* faults.

The *threats* that can harm the dependability of systems comprise the aforementioned faults, errors and failures. The *fault tolerance techniques* presented in the subsequent chapters tackle *reliability threats* that are able to manifest themselves as *erroneous outputs* of arithmetic computations. As a result, these techniques contribute to the goal of *dependable computing* by detecting and correcting erroneous outputs in the selected scientific computing algorithms.

2.2.2 Vulnerability Assessment

The vulnerability of modern CMOS-based computing devices to, for instance, the effects of transient events has been assessed in the literature using different approaches.

The vulnerability to different reliability threats can be assessed by *analytical methods* such as the Architectural Vulnerability Factor (AVF) analysis [Mukhe11, p.79], [Wilke14]. *Fault injection* and *fault emulation experiments* are widely used to assess the vulnerability of applications and the effectiveness of fault tolerance techniques [Nicol11]. Fault injection experiments can be performed at different levels of abstraction [Hsueh97], ranging from *hardware fault injection* over *simulation-based fault injection* to *software-based fault injection. Surveys* summarize the failure events under real conditions over certain periods of time. These surveys rely on error logs generated by fault tolerance techniques, which are used to obtain insights into the vulnerability of different system components.

Hardware fault injection experiments inject faults into the physical hardware of the target system. Different *physical experiments* are described in the literature in which circuits are exposed to, for instance, radiation while they perform certain tasks. While such physical experiments may mimic fault mechanisms realistically, they require special instruments like neutron beam generators [Tiwar15a]. At the same time, observing the manifestation of errors caused by specific faults can be challenging or even impossible.

Fault emulation experiments rely on logic emulation that is typically performed using reconfigurable hardware like *field programmable gate arrays* (FPGAs) [Cheng99]. Two widely used fault emulation approaches are the instrumentation of the circuit description and the reconfiguration of the emulated circuit. While circuit instrumentation approaches add fault injection logic to the original circuit description, reconfiguration-based approaches modify the FPGA configuration data (i.e., the underlying *bitstream*) to inject faults.

In contrast to physical experiments, fault emulation and simulation-based fault injection rely on *fault models*. Fault models bundle and collapse the different kinds of faults that affect a system into specific fault classes. These fault classes can be described at different abstraction levels at corresponding degrees of detail.

Simulation-based fault injection experiments rely on fault models that are applied to descriptions of circuits, architectures, or systems to evaluate the impact of faults. Fault injections are mimicked by, for instance, changing the hardware model or the simulated state of the hardware. For gate-level circuit descriptions, the *stuck-at* fault model [Bushn04] is a widely used description to mimic certain manufacturing defects. The *conditional line flip* model [Wunde10] can express the range of traditional circuit fault models by describing fault activations at specific fault sites using Boolean, time-related or arbitrary conditions. The *Resilience Articulation Point* model [Herke14] allows to assess the resilience between different abstraction levels of a system using probabilistic models such as p*robabilistic bit flips*, which are mapped between the levels using so-called abstraction transformation functions.

Software-based fault injection experiments [Segal88,Kanaw92] instrument applications to inject errors at runtime that mimic the manifestation of injected faults. Since faults only cause errors when they are activated and not masked during fault propagation, injecting errors in the targets of actual faults (e.g., the outputs of arithmetic computations), increases the number actually evaluated errors. For this reason, different related works rely on error injection experiments to evaluate the efficiency of fault tolerance techniques as these experiments avoid inactive faults. A widely used error model is the *bit flip model* [Brone08,Wunde10,Herke14,Fang16,Wu16,Loh16] which manipulates single or multiple bits in a value (e.g., in the output of an arithmetic unit) by inverting them or by forcing them to either 0 or 1. This model covers the manifestation of faults ranging from transient to permanent faults in arbitrary system components including *arithmetic units*, *register files* and *communication networks*. For these reasons, the fault tolerance techniques presented in this thesis are evaluated by software-based fault injection experiments using a bit flip error model.

Besides performing fault injection experiments, different works in the literature evaluated system logs collected over a certain period of time to assess the system vulnerability. Di Martino et al. [Di Ma16] assessed the vulnerability of the *Blue Waters* high-performance computing system under typical execution conditions. The authors evaluated the *machine check exceptions* collected in the system logs over a period of

261 days. In this period, 1 544 398 *machine check events* occurred which relates to an average error rate of 250 detected errors/h. Within these events, 28 errors were neither correctable by the applied fault tolerance techniques (i.e., *ECC* and *Chipkill* [Dell97]). For the GPU devices, uncorrectable error events occurred on average every 80 h. An evaluation of memory errors showed that about 70% of the memory errors involved a single bit while about 30% involved between 2 and 8 consecutive bits. Tiwari et al. [Tiwar15b] assessed the vulnerability of the *Titan* high-performance computing system. The authors evaluated the system logs collected over a period 21 months and analyzed the GPU-related events. In this time period, 48 uncorrectable errors were detected, which corresponds on average to one uncorrectable error event per week. In these events, 86% of these uncorrectable errors occurred in device memory, while the remaining 14% occurred in the register files.

In summary, the different vulnerability assessments in the literature show that modern CMOS-based computing devices are highly vulnerable to the effects of transient events, but also to permanent faults. This insight emphasizes the demand for fault tolerance techniques.

2.2.3 Fault Tolerance Strategies

Fault tolerance techniques intend to attain dependability of a system by avoiding failures in the presence of faults. A well-known strategy used by most fault tolerance techniques is to exploit specific forms of *redundancy*. By introducing redundancy, additional hardware, procedures, or information are used that are not directly required by the system to provide a correct service. However, these additional elements come into effect in case of occurring errors by detecting and correcting these errors. The majority of fault tolerance techniques can be classified into the three categories discussed below [Pradh96, Koren07]. These categories comprise *space redundancy*, *time redundancy*, and *information redundancy*.

Space redundancy is often referred to as *structural redundancy* when applied to hardware designs. This strategy adds additional hardware into the design to tolerate errors. The *Dual Modular Redundancy* (DMR) technique replicates a module (e.g., a processing element) once which allows to detect single errors. In contrast to DMR, the *Triple Modular Redundancy* (TMR) technique replicates a module into three units which allows to detect and correct single errors based on a majority decision. However, if more than one module is affected by an error, three different outputs can be obtained, and the voter is not able to provide a correct result.

Time redundancy relies on repeating computations a certain number of times and comparing the different results to detect errors. For instance, the *Duplication with Comparison* (DWC) technique repeats certain computations once and compares the two produced results. Time redundancy techniques can detect transient or intermittent faults. However, permanent faults remain undetected since these faults produce the same wrong output repeatedly. Errors are corrected by repeating the failed computations or by restarting the application from a priorly recorded state (i.e., a *checkpoint*).

Information redundancy techniques encode input data (i.e., the underlying information) to detect errors. For instance, to ensure the integrity of data that is stored in memory or transmitted between systems, *Error Detecting and Correcting Codes* (ECC) can be applied. These codes add some check bits to the original data bits which allow error detection or even correction depending on the underlying encoding scheme. The aforementioned ABFT techniques apply information redundancy to detect errors in the results of algorithms. These techniques add checksums to the input data before executing the algorithm and calculate new checksums for the results. Errors are detected by evaluating certain algorithm-specific invariants between checksums that were processed by the algorithm and the checksums computed for the results. When the encoding and invariant checking steps induce only low runtime overhead, such ABFT techniques can be very efficient, which makes them a favorable option in integrating fault tolerance. The fault tolerance techniques presented in this thesis exploit information redundancy to enable the efficient detection and correction of errors.

2.3 Related Fault Tolerance Techniques for Linear Algebra Operations

The investigation of fault tolerance techniques for linear algebra algorithms is an active research area and continues to gain in importance. Related fault tolerance techniques for matrix-matrix and matrix-vector multiplications are discussed below in Section 2.3.1. Conjugate Gradient solvers cannot be directly protected by these fault tolerance techniques as they do not cover all underlying operations. Related fault tolerance techniques for Conjugate Gradient solvers are discussed below in Section 2.3.2.

2.3.1 Fault Tolerance Techniques for Matrix Multiplications

Different algorithm-based fault tolerance (ABFT) approaches were proposed for *dense* linear algebra operations such as matrix multiplications and decompositions as well as matrix factorizations [Huang84, Braun14, Wu14, Hakka15, Du12, Boute15, Wu16]. ABFT techniques encode input data by adding *checksums* before performing the linear operation and calculate new checksums for the results.

Weights can be introduced during checksum generation to improve the error detection, localization and correction capability of ABFT techniques as introduced in [Jou86]. The computed checksums introduce information redundancy, which is now exploited to check the results for errors. Errors are detected by comparing the checksums that were processed by the operation and the checksums computed for the results. In the following, the protection of matrix-matrix multiplications using ABFT is discussed for square matrices. The underlying concept can be transferred to general matrices without loss of generality.

Weights w_k are used to encode an $(n \times n)$-matrix $\boldsymbol{A}$ by multiplying weights w_k to the matrix elements $a_{i,j}$ with $1 \le k \le n$. Weights that are used to encode a specific matrix $\boldsymbol{A}$ form a *weight vector*, which is denoted by $\boldsymbol{w_A}$. Different techniques were proposed in related works to select weights. One wide-spread approach is to set the weights to 1 [Brone08, Shant12]. Jou and Abraham [Jou86] propose the utilization of exponential weights. For a weight vector $\boldsymbol{w}$ with n elements, the weights w_k are computed as $w_k := 2^{k-1}$ for $1 \le k \le n$. While such exponential weights can be efficiently computed in their binary representation through shift operations, exponential weights can cause overflow problems for very large weights. To address this overflow problem, Luk [Luk86] proposes linear weights w_k that are computed as $w_k := k$ for $1 \le k \le n$. Fasi et al. propose quadratic weight vectors [Fasi16] with $w_k := k^2$ for $1 \le k \le n$.

ABFT techniques protect a matrix-matrix multiplication $\boldsymbol{R} := \boldsymbol{AB}$ with $\boldsymbol{A}$,$\boldsymbol{B}$, and $\boldsymbol{R}$ being $(n \times n)$-matrices by encoding the input operands $\boldsymbol{A}$ and $\boldsymbol{B}$ using a $(1 \times n)$-weight vector $\boldsymbol{w_A}$ and an $(n \times 1)$-weight vector $\boldsymbol{w_B}$. While the columns are encoded in $\boldsymbol{A}$ to obtain *column checksums*, the rows are encoded in $\boldsymbol{B}$ to obtain *row checksums*. Both encodings are performed using matrix-vector multiplications that result in two *checksum vectors*. The resulting checksum vectors are stored in additional columns and rows to form a column checksum matrix $\boldsymbol{A}_{cc}$, a row checksum matrix $\boldsymbol{B}_{rc}$, and a full checksum matrix $\boldsymbol{R}_{fc}$ as follows:

A *column checksum vector* $\boldsymbol{c_c}$ is a $(1 \times n)$-vector that is formed by computing *column checksums* for $\boldsymbol{A}$ as

$$\boldsymbol{c_c} := \boldsymbol{w_A}\boldsymbol{A} \,. \tag{2.20}$$

A *column checksum matrix* $\boldsymbol{A_{cc}}$ is an $(n+1 \times n)$-matrix that is formed by matrix $\boldsymbol{A}$ and its corresponding column checksum vector $\boldsymbol{c_c}$ as

$$\boldsymbol{A_{cc}} := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{c_c} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{w_A}\boldsymbol{A} \end{bmatrix} \tag{2.21}$$

A *row checksum vector* $\boldsymbol{c_r}$ is an $(n \times 1)$-vector that is formed by computing *row checksums* for $\boldsymbol{B}$ as

$$\boldsymbol{c_r} := \boldsymbol{B}\boldsymbol{w_B} \,. \tag{2.22}$$

A *row checksum matrix* $\boldsymbol{B_{rc}}$ is an $(n \times n+1)$-matrix that is formed by matrix $\boldsymbol{B}$ and its corresponding row checksum vector $\boldsymbol{c_r}$ as

$$\boldsymbol{B_{rc}} := \begin{bmatrix} \boldsymbol{B} & \boldsymbol{c_r} \end{bmatrix} = \begin{bmatrix} \boldsymbol{B} & \boldsymbol{B}\boldsymbol{w_B} \end{bmatrix} \tag{2.23}$$

A *full checksum matrix* $\boldsymbol{R_{fc}}$ is an $(n+1 \times n+1)$-matrix that results from the multiplication of a *column checksum matrix* $\boldsymbol{A_{cc}}$ and a *row checksum matrix* $\boldsymbol{B_{rc}}$ as

$$\boldsymbol{R_{fc}} \;:=\; \boldsymbol{A_{cc}}\boldsymbol{B_{rc}} \tag{2.24}$$

Following [Huang84], the elements of the full checksum matrix $\boldsymbol{R_{fc}}$ are evaluted as:

$$\boldsymbol{R_{fc}} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{R}\boldsymbol{w_B} \\ \boldsymbol{w_A}\boldsymbol{R} & \boldsymbol{w_A}\boldsymbol{R}\boldsymbol{w_B} \end{bmatrix} := \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{c_c} \end{bmatrix} \begin{bmatrix} \boldsymbol{B} & \boldsymbol{c_r} \end{bmatrix} \tag{2.25}$$

ABFT encodes the operand matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ according to Equations 2.21 and 2.23 and computes the full checksum matrix $\boldsymbol{R_{fc}}$, which allows to check for errors in the result matrix $\boldsymbol{R}$ by evaluating the following identity:

$$\boldsymbol{w_A}\boldsymbol{R}\boldsymbol{w_B} = \boldsymbol{c_c} \cdot \boldsymbol{c_r} \,. \tag{2.26}$$

Equation 2.26 follows from applying Equations 2.20 and 2.22 on $\boldsymbol{c_c} \cdot \boldsymbol{c_r}$ and substituting $\boldsymbol{AB}$ by $\boldsymbol{R}$ since $\boldsymbol{R} := \boldsymbol{AB}$. Errors are detected when Equation 2.26 does not hold.

Errors typically do not affect the complete result matrix $\boldsymbol{R}$, but only small parts of it. To avoid unnecessary recomputations, *syndrome vectors* are computed to find the location of errors, which allows to perform only necessary computations to correct errors (i.e., inner products) instead. The vector results of $\boldsymbol{R}\boldsymbol{w}_{\boldsymbol{B}}$ and $\boldsymbol{w}_{\boldsymbol{A}}\boldsymbol{R}$ in the full checksum matrix $\boldsymbol{R}_{fc}$ serve as inputs to compute two *syndrome vectors*. For the matrix-matrix multiplication $\boldsymbol{R} := \boldsymbol{A}\boldsymbol{B}$ with $(n \times n)$-matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{R}$, a row syndrome vector and a column syndrome vector are computed:

A *row syndrome vector* $\boldsymbol{s}_{\boldsymbol{r}}$ is an $(n \times 1)$-vector that is computed as

$$\boldsymbol{s}_{\boldsymbol{r}} := \boldsymbol{A}\boldsymbol{c}_{\boldsymbol{r}} - \boldsymbol{R}\boldsymbol{w}_{\boldsymbol{B}} \, . \tag{2.27}$$

and a *column syndrome vector* $\boldsymbol{s}_{\boldsymbol{c}}$ is a $(1 \times n)$-vector that is computed as

$$\boldsymbol{s}_{\boldsymbol{c}} := \boldsymbol{c}_{\boldsymbol{c}}\boldsymbol{B} - \boldsymbol{w}_{\boldsymbol{A}}\boldsymbol{R} \, . \tag{2.28}$$

During error-free matrix-matrix multiplications, both row and column syndrome vectors are zero:

$$\boldsymbol{s}_{\boldsymbol{c}} = \boldsymbol{s}_{\boldsymbol{r}} = 0 \, . \tag{2.29}$$

Proof: Equation 2.29 results from applying Equations 2.27 and 2.28 to Equation 2.26.

□

Non-zero syndrome elements $[\boldsymbol{s}_{\boldsymbol{r}}]_i$ and $[\boldsymbol{s}_{\boldsymbol{c}}]_j$ indicate the location of an error within the corresponding row i and column j in the result matrix $\boldsymbol{R}$ [Jou86].

With matrix size $(n \times n)$, $\mathcal{O}(n^2)$ computations are required to check for errors. For this reason, ABFT is efficient for multiplying dense matrices since $\mathcal{O}(n^3)$ computations are induced by the multiplication itself. Partitionable linear algebra operations such as matrix-matrix operations can be divided into blocks of smaller matrix operations which allows to perform the ABFT technique on the single blocks separately [Rexfo94].

ABFT can also be applied to *matrix-vector multiplications*. Let $\boldsymbol{A}$ denote an $(n \times n)$-matrix, $\boldsymbol{b}$ denote an $(n \times 1)$-vector and let $\boldsymbol{r}$ denote an $(n \times 1)$-vector. With the result and one of the operands being *vectors* instead of *matrices* (i.e., $\boldsymbol{b}$ and $\boldsymbol{r}$ instead of $\boldsymbol{B}$ and $\boldsymbol{R}$), different matrix-vector-multiplications that were required by ABFT to protect matrix-matrix-multiplications evaluate to *inner products* and to *vector-scaling operations*.

To protect a matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{Ab}$ using ABFT, the checksum vectors in Equations 2.20 and 2.22 are evaluated as follows [Brone08]: Equation 2.20 still requires a matrix-vector multiplication to compute the *column checksum vector* $\boldsymbol{c_c}$ and

$$\boldsymbol{c_c} := \boldsymbol{w_A A} \ . \tag{2.30}$$

At the same time, the $(n \times 1)$-vector $\boldsymbol{b}$ replaces the operand matrix $\boldsymbol{B}$, which changes the number of weights in weight vector $\boldsymbol{w_B}$. Since each row in $\boldsymbol{b}$ contains one column, the weight vector $\boldsymbol{w_B}$ now only contains one element, which is denoted as $\omega := [\boldsymbol{w_B}]_1$. Following Equation 2.22, the *row checksum vector* $\boldsymbol{c_r}$ is computed using a vector-scaling operation instead of a matrix-vector multiplication and

$$\boldsymbol{c_r} := \boldsymbol{b} \cdot \omega \ . \tag{2.31}$$

Example 2.2: The (3×3)-matrix $\boldsymbol{A}$ is multiplied by a (3×1)-operand vector $\boldsymbol{b}$ to obtain the (3×1)-result vector $\boldsymbol{r}$:

$$\boldsymbol{r} := \boldsymbol{Ab} = \begin{bmatrix} 1 & 5 & 2 \\ 6 & 2 & 4 \\ 4 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 19 \\ 22 \\ 18 \end{bmatrix}$$

To compute checksums for the columns in $\boldsymbol{A}$, three weights are required, since $\boldsymbol{A}$ contains three rows. Therefore, $\boldsymbol{w_A}$ is a (1×3)-vector. At the same time, one weight is required to compute checksums for the rows in $\boldsymbol{b}$, since $\boldsymbol{b}$ contains one column. This weight is denoted by ω. In this example, all weights are set to 1:

$$\boldsymbol{w_A} := \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \qquad \omega := 1$$

Using these weights, the input operand $\boldsymbol{A}$ is encoded using a matrix-vector multiplication to compute $\boldsymbol{c_c}$. Since only one weight is required to encode $\boldsymbol{b}$, the row checksum vector $\boldsymbol{c_r}$ is computed using a vector-scaling operation instead of a matrix-vector multiplication:

$$\boldsymbol{c_c} := \boldsymbol{w_A A} = \begin{bmatrix} 11 & 10 & 7 \end{bmatrix} \qquad \boldsymbol{c_r} := \boldsymbol{b} \cdot \omega = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix}$$

▶

Following Equation 2.25 and with $\boldsymbol{r}$ and $\boldsymbol{b}$ being $(n \times 1)$-vectors, the *full checksum matrix* $\boldsymbol{R_{fc}}$ is evaluated as a $(n+1 \times 2)$-matrix instead of an $(n+1 \times n+1)$-matrix and

$$\boldsymbol{R_{fc}} \coloneqq \boldsymbol{A_{cc}B_{rc}} \tag{2.32}$$

$$\boldsymbol{R_{fc}} \coloneqq \begin{bmatrix} \boldsymbol{r} & \boldsymbol{r} \cdot \omega \\ \boldsymbol{w_A} \cdot \boldsymbol{r} & \boldsymbol{w_A} \cdot \boldsymbol{r} \cdot \omega \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} \\ \boldsymbol{c_c} \end{bmatrix} \begin{bmatrix} \boldsymbol{b} & \boldsymbol{c_r} \end{bmatrix} \tag{2.33}$$

Compared to the matrix-matrix multiplication, for which ABFT requires a matrix-vector multiplication and two inner products to *detect errors* (cf. Equation 2.26), ABFT only requires *two inner products* and a scalar multiplication to *detect errors for matrix-vector multiplications* by evaluating the identity

$$\boldsymbol{c_c} \cdot \boldsymbol{c_r} = \boldsymbol{w_A} \cdot \boldsymbol{r} \cdot \omega\ . \tag{2.34}$$

Equation 2.34 follows from Equation 2.26 and will typically not be satisfied in case of errors [Sloan13].

Equation 2.34 allows to compute scalar checksums for both operands and results: Using the checksum vectors computed for the operands, $\boldsymbol{c_c}$ and $\boldsymbol{c_r}$, an *operand checksum* c_1 is computed as

$$c_1 \coloneqq \boldsymbol{c_c} \cdot \boldsymbol{c_r} = (\boldsymbol{w_A A}) \cdot (\boldsymbol{b} \cdot \omega)\ . \tag{2.35}$$

By encoding the result $\boldsymbol{r}$ using weight vectors $\boldsymbol{w_A}$ and $\boldsymbol{w_B}$, a *result checksum* c_2 is computed as

$$c_2 \coloneqq \boldsymbol{w_A} \cdot \boldsymbol{r} \cdot \omega\ . \tag{2.36}$$

Example 2.3: (*Continues Example* 2.2)
The *operand checksum* c_1 encodes the inputs $\boldsymbol{A}$ and $\boldsymbol{b}$:

$$c_1 \coloneqq \boldsymbol{c_c} \cdot \boldsymbol{c_r} = \begin{bmatrix} 11 & 10 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \cdot 1 = 59$$

The *result checksum* c_2 encodes the result $\boldsymbol{r}$:

$$c_2 := \boldsymbol{w_A} \cdot \boldsymbol{r} \cdot \omega = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 19 \\ 22 \\ 18 \end{bmatrix} \cdot 1 = 59$$

▸

In the following, the different steps of applying ABFT to matrix-vector multiplications are discussed with respect to the runtime complexity of the different operations. The fault tolerance technique is performed in two parts: The first part comprises a *setup phase* which is executed once for each matrix $\boldsymbol{A}$ to obtain the *column checksum vector* $\boldsymbol{c_c} := \boldsymbol{w_A}\boldsymbol{A}$. The second part comprises the operations for *error detection*, in which the checksum of the operands c_1 is computed and compared to the checksum of the result c_2. With matrix size n, $\mathcal{O}(n^2)$ computations are required for the *setup phase*, while error checking only requires $\mathcal{O}(n)$ computations (i.e., two inner products). This approach can induce low runtime overhead for dense matrices, since $\mathcal{O}(n^2)$ computations are induced by the multiplication itself.

Sparse linear algebra operations are essential tasks in different scientific and engineering applications besides dense linear operations. The straightforward application of ABFT to protect *sparse matrix-vector* (SpMV) multiplications induces significant runtime overheads as the runtime complexity of the SpMV multiplication depends on the number of non-zero elements in the underlying matrix. The runtime complexity of the SpMV multiplication can be *linear*, if each row in the underlying matrix contains only a few non-zero elements. In such a case, the runtime overhead for error detection may even exceed the runtime of the original operation, since the SpMV operation and the error detection operations are in same order of runtime complexity (i.e., $\mathcal{O}(NNZ) \approx \mathcal{O}(n),$ with $NNZ \approx n$). On top of that, when applied to matrix-vector multiplications, the error localization scheme used for matrix-matrix multiplications does not reduce the number of computations compared to complete recomputations [Sloan13]. Both the error localization scheme (i.e., compute $\boldsymbol{s_r} := \boldsymbol{Ab} \cdot \omega - \boldsymbol{r} \cdot \omega$ following Equation 2.27) and complete recomputations (i.e., recompute $\boldsymbol{r} := \boldsymbol{Ab}$) compute a matrix-vector multiplication with equal number of computations.

Nonetheless, the error detection operations described in Equation 2.34 is applied in different related works to detect errors in sparse linear operations such as the SpMV operation [Brone08,Shant12,Sloan12,Sloan13,Fasi16]: Bronevetsky and Supinsky [Brone08], Shantharam et al. [Shant12], and Fasi et al. [Fasi16] apply this approach within the Conjugate Gradient algorithm to detect and correct errors in the underlying SpMV operations.

Different approaches were proposed that focus on reducing the runtime overhead for error detection and correction in sparse linear operations. Sloan et al. propose checksum encodings [Sloan12] for the SpMV operation that omit some computations during error detection. These checksum encodings are based on either sampling some random matrix columns in the computation of the column checksum vector $\boldsymbol{c_c}$ or aggregating selected column sums to form clustered checksums. Both encodings reduce the number of non-zero elements in the column checksum vector $\boldsymbol{c_c}$, which reduces the error detection runtime. In the aggregation scheme, different clustering algorithms exploit structural properties such as dominant diagonality, diagonal bands or blocked diagonality to identify representative column sums for the column checksum vector $\boldsymbol{c_c}$. Besides, two preconditioning techniques are proposed that change or create additional structures within the matrix to make them suitable for this aggregation scheme. The fault tolerance technique corrects errors by rolling back the application to a prior checkpoint. This approach reduces the runtime overhead for error detection, but also reduces the error coverage.

Later, Sloan et al. proposed an error localization scheme [Sloan13] for the SpMV operation that avoids complete recomputations to reduce the runtime overhead for error correction. The proposed fault tolerance technique detects errors using the error detection operations in Equation 2.34 and delimits the corresponding portion in the result vector r_i in which the erroneous result element exists. The erroneous result vector element is corrected by recomputing the SpMV operation for the delimited portion. Such portions in the result vector are determined by an iterative bisection technique, which repeatedly divides the input matrix $\boldsymbol{A}$ into two row block matrices and checks for errors in these block matrices. The bisection is repeated until a certain portion of the vector around the erroneous result vector element is delimited. However, the runtime overhead to provide fault tolerance still depends on the error detection operations, as the result of this check is required before any error localization steps can be performed.

Gao et al. [Gao16b] developed a fault tolerance scheme that protects multiple matrix-vector multiplications executed in parallel by exploiting ideas from Error Detecting and Correcting Codes (ECC). For p matrix-vector multiplications with $\boldsymbol{b}^{(i)} = \boldsymbol{A}^{(i)}\boldsymbol{b}$

and $1 \leq i \leq p$, this scheme calculates a detection matrix $\boldsymbol{D}$ and a sum matrix $\boldsymbol{S}$ from p input matrices $\boldsymbol{A}^{(i)}$. For each input matrix $\boldsymbol{A}^{(i)}$, a checksum vector is computed with $\boldsymbol{c}^{(i)} = \boldsymbol{w}^T \boldsymbol{A}^{(i)}, \boldsymbol{w}^T = (1...1), i = (1, ..., p)$. The single rows in the detection matrix $[\boldsymbol{D}]_j$ are composed by adding specific checksum vectors $\boldsymbol{c}^{(i)}$ according to Hamming parity bit configurations. In case of a single error, the erroneous result vector $\boldsymbol{r}'^{(i)}$ is located and corrected using the sum vector $\boldsymbol{z} = \boldsymbol{Sb}$, with $S = \sum_{j=1}^{p} \boldsymbol{A}^{(j)}$ and $\boldsymbol{r}'^{(i)} = \boldsymbol{z} - \sum_{j=1}^{p} \boldsymbol{r}^{(j)}, r_i = 0$. However, this approach focuses on integer arithmetic which is uncommon in the scientific and engineering computing domain. The challenge to distinguish inevitable rounding errors from harmful errors is not addressed. Besides, the runtime overhead is dominated by two additional matrix-vector multiplications which can be significant for low numbers of input matrices p.

Since linear operations for both dense and sparse matrices are often performed using floating-point arithmetic, rounding errors occur. A more detailed discussion on the nature of rounding errors in floating-point arithmetic can be found in Appendix C. A direct comparison of checksums should be avoided because rounding errors typically cause small differences in these checksums. To avoid false positive error detections in the presence of rounding errors, error checking needs to be performed under consideration of a *rounding error bound.*

A *rounding error bound* $\tau \in \mathbb{R}$ is an upper bound for the maximum difference between an operand checksum $c_1 \in \mathbb{R}$ and a corresponding result checksum $c_2 \in \mathbb{R}$ for

$$|fl(c_1) - fl(c_2)| < \tau \tag{2.37}$$

with $fl(c_1) \in \mathbb{R}$ and $fl(c_2) \in \mathbb{R}$ denoting the corresponding floating-point representations for c_1 and c_2, respectively (cf. Equation D.2).

Errors are detected, when $|fl(c_1) - fl(c_2)|$ exceeds the rounding error bound τ. The challenge to distinguish inevitable rounding errors from harmful errors was addressed in different related works. Huang and Abraham propose to let the user determine such thresholds τ manually [Huang84]. However, this approach requires a deep knowledge of the input data and re-calibrations for subsequent or new problem sets. Different approaches were proposed that determine such rounding error bounds at runtime with respect to the input data. *Analytical rounding error functions* provide upper estimations for the rounding error with respect to the underlying input data. Such rounding error functions were derived for different basic arithmetic operations as well as for linear operations like inner products [Golub13]. To protect matrix-vector multiplications using dense matrices in the presence of rounding errors, an analytical rounding error bound was derived by Chowdhury and Banerjee [Chowd96].

While these analytic rounding error functions focus on the worst-case rounding error [Higha96], probabilistic models of floating-point arithmetic can provide statistical statements on the rounding error behavior. Such models were used by Barlow and Bareiss [Barlo85] to derive probabilistic error bounds for sums and inner products in floating-point arithmetic. Based on this approach, an algorithm-based fault tolerance scheme was proposed for dense matrix-operations such as multiplications and decompositions which determines rounding error bounds at runtime [Braun14].

In summary, the direct application of traditional ABFT is highly inefficient when applied to sparse linear operations such as the SpMV multiplication. Related works [Sloan12] and [Sloan13] focus on reducing the runtime overheads for error detection and correction steps. However, the applied error detection schemes only indicate the existence of errors and not their location. Either complete recomputations or additional error localization steps need to be performed to correct errors. Scientific and engineering applications are typically performed using floating-point arithmetic which is prone to rounding errors. For this reason, suitable rounding error bounds are required to distinguish errors in the magnitude of the rounding error and errors that may be harmful to the application. In Chapter 3, a fault tolerance technique for sparse matrix operations is presented that allows the efficient algorithmic detection and correction of erroneous computation results with high error coverage.

2.3.2 Related Fault Tolerance Techniques for Conjugate Gradient Solvers

While the ABFT techniques discussed above can protect the linear operations in the Conjugate Gradient solvers, they are not able to protect all operations with low runtime overhead. For instance, when the error detection operations in Equation 2.34 are applied to protect an inner product $\boldsymbol{a}^T\boldsymbol{b}$, one additional inner product will be computed (i.e., $(w \cdot \boldsymbol{a}^T)\boldsymbol{b}$ with w being a *weight scalar*). This approach does not allow low runtime overhead since the additional inner product has the same runtime complexity as the original one. Therefore, fault tolerance for the Conjugate Gradient solvers demands different methods to achieve complete and efficient protection.

The vulnerability of the Conjugate Gradient solvers was assessed over the last decade: Bronevetsky and Supinski [Brone08] show the insufficient ability of CG to avoid silent data corruptions in the presence of errors. Shantharam et al. [Shant11] discuss the influence of errors on the performance of linear solvers and demonstrate performance degradations of PCG by factors of up to 200x.

Different fault tolerance approaches were presented to detect and correct transient events causing errors: Oboril et al. [Obori11] present a fault tolerance technique that repeats PCG on an auxiliary problem, if an incorrect solution is detected after the completion of PCG. In such a case, PCG is repeated on the obtained residual $\boldsymbol{Ad} = \boldsymbol{r}$ $[:= (\boldsymbol{b} - \boldsymbol{Ax})]$. While this method aims to avoid repetitions of PCG on the original problem, it awaits the result after complete convergence of PCG before it checks for errors. Sao and Vuduc [Sao13] propose to periodically *stabilize* the solver execution during *inherently reliable system modes*. Such stabilizations exploit the convergence conditions of CG to transform arbitrary iterations to valid iterations. Chen [Chen13] proposes a periodic check of both the residual invariant (i.e., $\boldsymbol{r}^{(k)} \approx \boldsymbol{b} - \boldsymbol{Ax}^{(k)}$) and the orthogonality of consecutive search direction and residual vectors (cf. equation 2.17) for error detection. Detected errors are corrected by rolling the solver back to a previously recorded state (i.e., a *checkpoint*). Chien et al. propose a hierarchical checkpointing technique [Chien16] to extend the fault tolerance technique presented by Chen. This technique targets *latent errors* that can cause endless loops in traditional checkpoint-rollback techniques which rely on single recorded states. An array comprising multiple checkpoints is maintained that holds different application states recorded during the execution. Loh et al. [Loh16] propose to check the orthogonality of consecutive residual and preconditioned residual vectors (i.e., $(\boldsymbol{M}^{-1}\boldsymbol{r}^{(k-1)})^T\boldsymbol{r}^{(k)} = \boldsymbol{s}^{(k-1)T}\boldsymbol{r}^{(k)} \approx 0$) as well as its typical decreasing monotonicity after each iteration to detect errors. The steps performed to update the intermediate result $\boldsymbol{x}^{(k)}$ are duplicated and compared in each iteration to detect errors that may not violate this orthogonality invariant. The solver is recovered from detected errors by restarting the solver from the last computed version of $\boldsymbol{x}^{(k)}$. Tao et al. [Tao16] present a fault tolerance technique that relies on computing and adapting checksums for the vectors in PCG. Vector updates are tracked by adapting the checksums according to the underlying value changes at runtime. The checksum encoding follows the ABFT encoding scheme discussed above in Section 2.3.1. To detect errors, new checksums are computed for the resulting vectors which are compared to the original checksums in user-defined intervals. In case of detected errors, the solver is rolled back to a previously recorded state.

Different related works rely on replication schemes to introduce fault tolerance: Liu et al. [Liu15b] propose to use increasing levels of replication as the solver converges to a solution. After a predefined number of iterations, the state of the PCG solver is replicated and the execution is continued in a Dual Modular Redundancy scheme that periodically crosschecks the underlying intermediate results. The state of the PCG solver is replicated again after a predefined number of iterations and the execution is continued in a Triple Modular Redundancy scheme. Detected errors are corrected by rolling back to a previously recorded state when DMR is used and by recovering an erroneous execution from a majority decision when TMR is used. Dichev and Nikolopoulos [Diche16] propose a Dual Modular Redundancy scheme that replicates the execution of PCG. This scheme periodically synchronizes both executions to detect errors by comparing the computed residual norms (i.e., $\|r_1^{(i)}\| \approx \|r_2^{(i)}\|$). To correct errors that affected only one of both executions, the residual invariant (i.e., $\boldsymbol{r}^{(k)} \approx \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$) is checked in both executions to identify the correct execution which is used to restore the erroneous one. If both executions do not satisfy the residual invariant, the execution is restarted from a previously recorded state.

The discussed approaches rely on strategies and operations that induce significant runtime and energy overheads to obtain fault tolerance. Such overheads can violate the central demand in scientific and engineering computing domain for fast and efficient computations.

The approaches in [Sao13, Chen13] require additional sparse matrix-vector operations to detect errors which induces significant runtime overheads. At the same time, the approaches in [Loh16, Tao16] add at least one additional inner product into each solver iteration. While the parallel execution of replicated PCG instances can reduce the runtime overhead for DMR and TMR schemes presented in [Liu15b, Diche16], the demanded energy is multiplied by a factor constituted by the number of replications. Some approaches recover from errors solely by using checkpointing techniques, which are associated with large recovery cost in recomputing lost intermediate results.

Chapter 4 presents an efficient fault tolerance technique for the Conjugate Gradient solvers that relies on an error detection and correction scheme with low runtime and energy overhead while it achieves high error coverage.

2.4 Heterogeneous Computer Architectures and Approximate Computing

The scientific and engineering computing domains demand feasible execution times along with high reliability. Compute-intensive applications from these domains are often accelerated using *heterogeneous computer architectures* since they provide high computational performance within reasonable power envelopes. This section presents the concept of heterogeneous computer architectures and discusses its benefits for the fields of science and engineering. At the same time, it introduces the necessary background for the *approximate computing paradigm* which is promising to become an integral part of such computer architectures [Esmae12a, Chand17].

2.4.1 Heterogeneous Computer Architectures

Semiconductor technology scaling induces to integrate considerably different computer architectures with different kinds of processing cores, communication channels and embedded memories on single chips or packages [Chen15a]. Heterogeneous computer architectures are a result from this integration process and combine, for instance, *multi-core CPUs*, *many-core GPUs* architectures as well as reconfigurable architectures like field programmable gate arrays (FPGA) [Chung10]. One of the most widespread example for these computer architectures is the integration of CPU and GPU architectures on single chips [Mitta15] such as *Intel*'s *Skylake* architecture [Intel17] and *AMD*'s APU architecture [Bouvi14].

Applications from the scientific and engineering domain often comprise of different computational parts which, for instance, reflect multiple interacting processes from multi-phase, multi-scale or multi-physics problems. Applications are accelerated by mapping the different underlying algorithmic parts to the different components of such heterogeneous architectures which can result in significant reductions of computation time. For instance, while multi-core CPUs are typically optimized for latency-sensitive as well as coarse-grained parallel tasks, many-core GPUs provide high computational throughput.

The computational performance of heterogeneous computer architectures is the most important aspect for applications from the scientific and engineering computing domains. Compared to homogeneous architectures (i.e., that rely on single kinds of processing units), these heterogeneous architectures can achieve significantly increased performance to execute such applications. By integrating these highly diverse kinds of processing cores, the gap between latency-sensitive or coarse-grained parallel tasks and highly data-parallel tasks is closed. These different kinds of tasks were often individually tailored to latency-optimized CPUs and throughput-optimized GPUs before. For this reason, the acceleration of such algorithmic parts is enabled that did not fit in an optimal manner to one of the existing architectures before.

With the growing interest for heterogeneous computer architectures over the last decade, different application programming interfaces were developed that allow to tailor applications to these architectures. For instance, *OpenCL* [Stone10] and *Nvidia CUDA* [NVIDI17] are two wide-spread examples for such programming interfaces. Different software libraries like *SparseLib* [Donga94] and *NIST Sparse BLAS* [Duff02] allow the integration of sparse linear algebra operations and solvers into a wide range of applications. A research field focuses on *building blocks* that accelerate compute-intensive and recurring tasks including sparse linear algebra operations [Kreut16] using such computer architectures. Important examples include software frameworks like MAGMA [Donga12], ViennaCL [Rupp10] and PETCs [Balay16].

2.4.2 The Approximate Computing Paradigm

While the concept of *approximate computing* was already addressed several decades ago [Von N56], the research activities on this computing paradigm have only just intensified in the last couple of years. Approximate computing tackles different computational efficiency cost metrics in the system stack. Important metrics include the *area-time complexity*, the *power-clock cycle product*, and the *energy-time product* [Nebel13,Kaesl14]:

The *area-time complexity*

$$\text{Area-time complexity} := A \cdot T \tag{2.38}$$

measures the resource efficiency for a given (chip) area A and a time T spent to process an instruction. A closely related metric is the *area-time*^{2} *complexity* [Thomp79]

$$\text{Area-time}^2 \text{ complexity} := A \cdot T^2 \ . \tag{2.39}$$

The *power-clock cycle product*

$$\text{Power-clock cycle product} := P \cdot t_c \tag{2.40}$$

provides insights into the energy efficiency of a circuit for P being the chip power dissipation and t_c being the clock cycle.

The *energy-time product*

$$\text{Energy-time product} := E \cdot T \tag{2.41}$$

measures the interplay between computational performance and energy efficiency for E denoting the energy and T denoting the time spent to process an instruction.

A metric to measure the *computational performance* is *MIPS* (millions of instructions per second):

$$\text{MIPS} := \frac{\text{instruction count}}{10^6 \cdot \text{execution time}} \,. \tag{2.42}$$

The *energy-per-instruction* metric

$$\text{Energy-per-instruction} := \frac{E}{\text{instruction count}} = \frac{P \cdot T}{\text{instruction count}} \tag{2.43}$$

measures the power efficiency for E denoting the energy, P denoting the power dissipation, and T denoting the time spent to process an instruction. With T = instruction count/MIPS, the energy-per-instruction metric is related to the *Watt/MIPS*-metric [Rabae12] as follows:

$$\text{Energy-per-instruction} = P \cdot \frac{T}{\text{instruction count}} = \frac{\text{Watt}}{\text{MIPS}} \,. \tag{2.44}$$

At the same time, the *energy-time product* is related to the *Watt-per-MIPS*^{2}-metric as follows:

$$\text{Energy-time product} = P \cdot \left(\frac{T}{\text{instruction count}}\right)^2 = \frac{\text{Watt}}{\text{MIPS}^2} \,. \tag{2.45}$$

Central terms in approximate computing are *accuracy* and *precision.* For an exact result $c \in \mathbb{R}^n$ and a specified maximum distance $d \in \mathbb{R}, d \geq 0$ to the exact result, the *accuracy* is the probability P that a computed result c' is within the specified distance d to the exact value c:

$$\text{Accuracy} := P(\{\|c - c'\|_2 \leq d\}) \,. \tag{2.46}$$

The *precision* is the distance from a computed result c' to the exact value c:

$$\text{Precision} := \|c - c'\|_2 \,. \tag{2.47}$$

Precision is a characteristic of the *approximation technique* (e.g., approximate hardware and its approximate arithmetic) while the accuracy is a requirement defined by the application, in particular by its underlying algorithms and the processed data. For instance, approximate arithmetic structures such as adders or multipliers that exhibit a large degree of approximation are typically associated with a lower precision, meaning that the results of their operations are allowed to deviate more from the exact computational result. Important examples for error metrics [Han13] are the *absolute* and *relative error* and the *error rate.*

For an exact result $c \in \mathbb{R}$ and a computed result $c' \in \mathbb{R}$, the *absolute error* ε_{abs} is

$$\varepsilon_{abs} := |c - c'| \, . \tag{2.48}$$

The *relative error* ε_{rel} is

$$\varepsilon_{rel} := \frac{|c - c'|}{|c|} \, . \tag{2.49}$$

The *error rate* $error_{rate}$ describes the relation between the number of input values and the number of computed results c' that are unequal to the precise result c with

$$error_{rate} := \frac{\text{Number of imprecise results with } \|c - c'\|_p > 0}{\text{Total number of input values}} \, . \tag{2.50}$$

Application-specific error metrics were proposed that are able to evaluate the accuracy of application results [Grigo14a, Grigo14b, Zhang14a, Ringe15, Zhang15a].

Different approximation techniques target the power dissipation of circuits. The three basic power dissipation sources of CMOS devices are the *charging and discharging* of capacitors, *short-circuit* currents and *leakage* currents [Nebel13]. In the following, let V_{DD} denote the supply voltage, let f denote the clock frequency, let I_{leak} denote the leakage current, let A_i denote the activity, let Q_i denote the transported charge during a short-circuit and let C_i denote the capacitance at node i. The power dissipation caused by *charging and discharging* of capacitors P_{CD} is

$$P_{CD} := \frac{1}{2} \cdot f \cdot V_{DD}^2 \cdot \sum_i A_i \cdot C_i \, . \tag{2.51}$$

The power dissipation caused by *short-circuit* currents P_{SC} is

$$P_{SC} := V_{DD} \cdot f \cdot \sum_i Q_i \cdot A_i \, . \tag{2.52}$$

The leakage power P_{leak} is

$$P_{leak} := I_{leak} \cdot V_{DD} \, . \tag{2.53}$$

Based on these sources, an estimation for the average power dissipation is

$$P \approx \frac{1}{2} \cdot f \cdot V_{DD}^2 \cdot \sum_i A_i \cdot C_i + V_{DD} \cdot f \cdot \sum_i Q_i \cdot A_i + I_{leak} \cdot V_{DD} \,. \tag{2.54}$$

To evaluate parameters like power and performance, different techniques compute an *instruction mix* [Patte14] of target applications. Let $i := 1, \cdots, n$ denote instructions and let p_i denote the probability for the execution of instruction i. *An instruction mix* is a set of instructions i along with probabilities p_i that are obtained by counting instruction executions in the target applications. Instead of executing the target applications while measuring different parameters, *benchmark programs* can be derived from the obtained instruction mix. Compared to the original applications, these benchmarks can exhibit a reduced instruction count, for instance, and allow reduced execution and evaluation times.

A wide range of *approximate computing techniques* have been proposed for different system layers from circuits over architectures to software. These techniques exploit the inherent resilience of applications to certain numerical errors. The current scope of applications for approximate computing can be distinguished into different classes depending on their input data sets, their output data sets or the computational patterns they exhibit [Chakr10, Chipp13, Venka15, Shafi16]:

Applications with imprecise inputs include applications that process real-world data sets that are often inherently noisy. Subsequent computations in such applications do not have to be more precise than what the input data allows. The underlying algorithms of these applications are designed to handle noise appropriately, which gives them the ability to tolerate errors due to approximation. A second member of this class are applications that process large data sets with high degrees of redundancy, which enables a certain error tolerance. Such applications include, for instance, voice recognition, motion detection or processing of sensor data [Chipp13, Wang16, Raha17].

Applications with imprecise outputs include applications whose required output quality is defined by, for instance, the limited perceptual abilities of human beings. Deviations from perfect results due to approximation errors can be tolerated as long as they are not perceived by the user. Examples for such applications include audio, image and video processing in multimedia applications [Advan14, Schaf14, Tagli16].

Applications with ambiguous outputs include applications that produce a range of output data that are equally acceptable for the users. These applications include recommendation systems, web searches, modern machine learning techniques [Esmae12c, Grigo14a, Eldri14, Grigo15, Morea15, Zhang15b] and often process data based on heuristics, statistical aggregation, and probabilistic computations. A perfect solution does not necessarily exist for such applications.

Applications with convergent outputs include applications which utilize optimization techniques or iterative methods that refine the output data set until a certain convergence criterion is satisfied [Zhang14a, Lass17]. The output quality may vary depending on the processing steps and the computational precision. Errors due to approximate computations may be compensated by longer execution times. The Conjugate Gradient solvers (cf. Chapter 2.1.2) are an example for such iterative convergent tasks where such a compensation often does not occur.

The approximate computing paradigm has been applied to the whole computing stack including circuits, architectures up to software, programming models and algorithms [Shafi16]. Approximate computing hardware designs are developed with deviations from their exact specification which therefore cause approximation errors up to a certain degree. The spectrum of proposed approximate hardware designs ranges from approximate adder structures [Mahdi10, Gupta11, Miao12, Gupta13, Kim13, Yang13b, Nanu14, Hu15, Beche16] and multipliers [Kyaw10, Kulka11, Farsh13, Bhard14, Chen15b, Liu17] over approximate floating-point components [Zhang14b, Liu16d, Yin16] to structures that allow to configure the underlying precision at runtime [Kahng12, Lin13, Bhard13, Liu14, Hashe15, Camus15, Espos16, Mazah16]. Besides arithmetic units, different approaches target efficiency gains in the memory hierarchy by, for instance, estimating load values to reduce cache miss latencies [Migue14, Yazda16a, Yazda16b, Jain16], skipping memory accesses [Samad14] as well as reducing the refresh rates in DRAM memories [Liu12, Cho14, Jeong16] and voltage [Chen16]. Different design automation methods were proposed that focus on the design, automatic generation and synthesis of approximate circuits [Venka12, Venka13b, Miao13, Nepal14, Yazda15, Soeke16]. Besides the design phase, different approaches analyze the error behavior of approximate hardware [Venka11] and propose formal verification techniques for approximate hardware [Holik16].

Different concepts have been proposed that introduce the approximate computing paradigm into heterogeneous computer architectures. Esmaeilzadeh et al. [Esmae12a] present a computer architecture that allows application developers to select between precise and approximate computation modes. A special instruction set architecture offers approximate variants of precise arithmetic operations that can be used to achieve efficiency-gains for error-resilient instructions within applications. This computer architecture relies on voltage scaling in different components such as register files and caches, as well as in arithmetic operations to trade-off errors for reduced power dissipation. Venkataramani [Venka13a] presents a computer architecture that provides precision-configurable arithmetic units along with error estimation units. Different monitors estimate the accumulation of numeric errors that are induced by rounding at different precision degrees, which can be evaluated by applications using special instructions. Unacceptable errors are corrected by recomputing the affected instructions with increased precision. Chandrasekharan et al. [Chand17] present a computer architecture that combines approximate floating-point units with their precise counterparts. These approximate floating-point units reduce the number of clock cycles required for floating-point operations by storing already performed operations and results in a look-up table, which is checked before each new floating-point operation. If a new floating-point operation is similar to an already performed operation, then the corresponding result value is returned from the lookup table, while the floating-point unit is bypassed. The maximum degree of similarity that triggers such a bypass can be dynamically adapted by the application. Koutsovasilis et al. [Kouts17] present a benchmark suite of 12 compute-intensive tasks from the science and engineering domain that were adapted to heterogeneous and approximate computer architectures. Each implemented task relys on an accurate and an approximate version of its computationally intensive parts. The underlying approximation techniques are based on task skipping, precision-reduction in arithmetic operations and relaxation of sychronization barriers.

On the software level, task skipping and early termination techniques like loop perforation [Sidir11] reduce the number of executed instructions at the cost of result quality. Programming language extensions allow developers to guide the approximation by, for instance, source code annotations that specify approximable data and instructions [Samps11, Carbi13, Rahim13, Vassi15]. Changing quality demands are exploited by compiler-based program transformation techniques that generate multiple versions of programs with different precision levels and memory access frequencies [Samad13]. The approximate nature of neural networks is exploited to mimic certain algorithms and to compute approximate results [Morea15, McAfe15]. At the same time, neural networks and their underlying evaluation algorithms are approximated to obtain efficiency gains [Venka14, Zhang15b, Sarwa16].

The next section discusses related approximate computing schemes, which were proposed for essential tasks in the scientific and engineering computing domain including Cholesky Decomposition [Schaf14], Eigen-Decompositions [Zhang15a], iterative methods [Zhang14a], and inverse matrix p-th roots [Lass17].

2.5 Related Approximate Computing Techniques

This section presents and discusses the related work for the approximate computing methods presented in Chapters 5 and 6. Related works that target the extension of the approximate computing application scope to the scientific and engineering computing domain are discussed below in Section 2.5.1. Different compute efficiency parameters need to be evaluated to asses the execution of applications on approximate computing hardware. Related works that determine such parameters are discussed below in Section 2.5.2.

2.5.1 Related Approximate Computing Techniques for Scientific Computing Tasks

Schaffner et al. [Schaf14] propose an approximate computing technique for a direct Cholesky decomposition-based solver that targets well-conditioned problems arising in video processing applications. The technique exploits the application-specific property that such well-conditioned problems tend to contain insignificant values in the *fill-in* elements of their Cholesky decomposition. During the decomposition process, multiplications are skipped in the computation of such *fill-in* elements that contain operands smaller than a specific threshold value.

Zhang et al. [Zhang14a] propose a monitoring technique for iterative methods that continuously adapts the induced approximation error according to their underlying *optimization functions* [Saad03]. This technique starts the execution using the lowest available degree of precision, which is increased if the underlying optimization function is violated. While some iterative methods rely on evaluating their underlying optimization function, *Krylov-subspace methods* like the Conjugate Gradient solvers typically do not compute this function explicitly to find solutions. The periodic computation of this function induces significant runtime overheads since it requires an additional expensive matrix-vector multiplication. On top of that, such additional matrix-vector multiplications can cancel out potential energy savings.

Later, Zhang et al. [Zhang15a] present a method that enables the iterative *Lanczos algorithm* for approximate computing by dynamically adapting the induced approximation. As a prerequisite, the authors evaluated the error resilience of the underlying algorithmic parts using the resilience identification approach in [Chipp13] to determine candidates suitable for approximation. The presented technique exploits the insight that the *Lanczos algorithm* restarts the search process to re-establish inherent properties (i.e. orthogonality). In case of such restarts, the proposed technique adapts the induced approximation by increasing the underlying precision. Based on the ideas in [Sloan12], the underlying matrix-vector multiplications in the *Lanczos algorithm* are monitored to evaluate the result quality against accuracy demands.

Lass et al. [Lass17] analyze the error resilience of an iterative algorithm that computes the inverse matrix p-th roots (i.e., $\boldsymbol{A}^{-1/p}$) of a positive definite matrix $\boldsymbol{A}$ [Bini05]. The convergence behavior in the presence of approximation errors is evaluated by comparing the intermediate solutions against solutions computed in precise arithmetic. Approximation errors are mimicked by reducing the precision in the underlying computations and data representations. Reducing the precision below a certain degree introduces an oscillating behavior which is dependent on the matrix size and the root factor p.

In summary, the discussed approximate computing schemes target different scientific and engineering applications. The underlying approximation schemes include detecting and skipping computations for insignificant values, monitoring inherent correction cycles to adapt the induced approximation as well as adding monitors to detect violations of inherent application properties. For iterative methods including the Conjugate Gradient solvers, such monitoring techniques must induce only low runtime and energy overhead to avoid reducing or canceling the achieved efficiency gains.

Chapter 5 presents an adaptive method that exploits the fault-tolerant *Conjugate Gradient* solver (i.e., presented in Chapter 4) to enable the solver execution on approximate computing hardware with high compute efficiency gains while still providing correct results. This method propels the extension of the application scope of approximate computing to the scientific and engineering computing domain, which are often associated with low error resilience along with tight constraints on the result accuracy.

2.5.2 Related Parameter Estimation Techniques for Application Executions on Approximate Computing Hardware

To assess the compute efficiency of application executions on approximate computing hardware, different *parameters* need to be determined, which comprise the *area*, the *leakage power*, the *dynamic power*, the *delay*, and the *approximation error*. Using these parameters, the compute efficiency is described by different metrics including the energy per instruction as well as the overall runtime performance, for instance. The different related works can be distinguished into two classes with respect to the targeted parameters. The first class comprises related works that investigate the approximation error and its propagation throughout the application execution, while the second class comprises related works that provide parameter estimation methods targeting, for instance, the leakage and dynamic power dissipation.

Different related works *model the approximation error* induced by approximate computing hardware to investigate the approximation error resilience of applications: Chippa et al. [Chipp13] inject *random bit flip errors* into the output variables of different loops and functions to identify error-resilient algorithmic parts. Algorithmic parts that do not cause application crashes or unacceptable result deviations are further evaluated using software-based models of approximate computing techniques. These models include *loop perforations*, *operand value truncations* and *bit error profiles* that specify the error probability for each bit in an arithmetic unit.

Roy et al. [Roy14] use *random value manipulations* to identify error-resilient application data (e.g., program variables). This approach collects representative value ranges for each variable in a program depending on the variable data type to form a multi-dimensional search space. Different configurations are randomly extracted from such search spaces to evaluate the sensitivity of the application output to certain variables.

Mishra et al. [Mishr14] present a framework that combines annotation-guided code transformations and *instruction set simulation of approximate computing techniques* to determine the error resilience of applications. The simulated approximation techniques include operand value truncation, approximate memories, and approximate network channels.

Single bit flip error injections are utilized by Venkatagiri et al. [Venka16] to identify error-resilient arithmetic instructions. Program analysis methods and heuristics are applied to collect suitable error injection candidates, which are grouped in error equivalence classes. Error injection experiments are performed on a representative of such equivalence classes while the application output is monitored.

Barbareschi et al. [Barba16] use *reduced floating-point precision* to evaluate the error resilience of applications. Existing source code is transformed by user-defined transformation rules to generate approximate source code variants. These source code variants are evaluated using user-defined error metrics to assess the error resilience of the different code parts.

Lee et al. [Lee16] present an approach that generates models for arbitrary approximate hardware designs using *data flow graphs* and *probability mass functions*. Data flow graphs capture the propagation of data and approximation errors from the inputs to the outputs in approximate hardware designs. Probability mass functions are used to model the influence of approximation errors on the dependency between input values and approximate application outputs.

The discussed related works [Chipp13, Roy14, Mishr14, Venka16, Barba16, Lee16] rely on different software-based models of approximate computing hardware, which are used to identify algorithmic parts as well as data that are resilient to approximation errors. To evaluate the significance of the compute efficiency, parameters like the power dissipation and the demanded energy must also be assessed besides the application output error.

Different techniques have been presented that evaluate the *power dissipation* of *application executions*. One approach is to determine the power dissipation of individual instructions using *physical measurements* and to use the obtained results to *estimate the power dissipation* for complete application executions: Tiwari et al. [Tiwar94, Tiwar96] present such a measurement-based technique that generates power models for instruction traces by quantifying the power dissipation of individual instructions and different inter-instruction effects. Such inter-instruction effects include pipeline and write buffer stalls as well as cache misses. The power models are obtained through physical experiments in which the current drawn by the target device is measured while executing the application under consideration in an infinite loop.

Physical experiments are also used by Laurent et al. [Laure04] for a power modeling technique called *functional level power analysis*. To model the power dissipation, different programs are executed on evaluation boards, while the drawn current is measured. The measurements are summarized in a power model using regression. Later, this technique is applied by Senn et al. [Senn04] to estimate the power dissipation for applications. Using static code analysis, different parameters such as the instruction mix are determined. These parameters are translated to power dissipation using the models that were generated by *functional level power analysis*. Rethinagiri et al. [Rethi14] apply the *functional level power analysis* approach to heterogeneous computer architectures comprising multi-core CPUs and field programmable gate arrays (FPGA). The authors present power models for different devices that allow to estimate the power dissipation from different parameters of the executed application and the underlying target design. These parameters include the instruction mix and number of cores.

The discussed related works [Tiwar94, Laure04, Senn04, Rethi14] estimate the power dissipation of complete application executions by modeling the power dissipation of the underlying instructions using physical experiments. While these experiments are performed at full execution speed, they require the implementation of approximate hardware descriptions in physical hardware, which can be associated with high cost.

Different related works estimate the power dissipation of application executions by using *capacitance descriptions*: Brooks et al. [Brook00] provide capacitance equations for different architectural components including caches, fully associative memories, combinational logic, wiring as well as clocking to compute the capacitance of a target device. Using architectural simulations for the target applications, the circuit activity is determined to estimate the power dissipation of application executions.

Capacitance descriptions are also used by Li et al. [Li09] in *McPAT*, which provides models for power, area, and timing estimation targeting architectural-level circuit descriptions. This approach decomposes the target architecture into circuit blocks and computes the capacitance for each module using analytical models to evaluate the power dissipation.

The discussed related works [Brook00, Li09] rely on capacitance descriptions to model the underlying hardware, which is evaluated by architectural simulation to estimate the power dissipation. Different approximate hardware designs are based on changes in the underlying functional description of the circuit (e.g., in the gate-level description of approximate arithmetic units [Yang13b, Chen15b, Liu17]), which are not necessarily reflected in an architectural simulation.

Sampson et al. [Samps11] provide a model that *specifies power and energy reductions* for different architectural components with respect to certain approximation techniques. The presented reduction specifications have been collected from different works in the literature targeting voltage scaling, mantissa reduction in floating-point operations, as well as memory refresh rate reductions. This model provides insights for three configurations of the evaluated approximation techniques. However, precision-configurable approximate computing arithmetic often provides a significantly larger number of configurations, which need to be evaluated to obtain detailed insights into the power dissipation.

Different related works target *simulation-based power estimation techniques* and present techniques to *reduce the simulation runtime* for complete application executions: Hsieh and Pedram [Hsieh98] estimate the power dissipation of an application execution by combining architectural and register-transfer-level simulation. In a first step, architectural simulation is performed to obtain different application parameters including the instruction mix. A new program is generated from these parameters, which exhibits similar performance and power dissipation behavior as the original application. Compared to the original instruction trace, the instruction trace of the generated program is reduced with respect to a user-defined compression ratio. In a second step, the new program is simulated at the register-transfer level to determine the power dissipation. The obtained power dissipation result is used as an estimation for the complete application execution.

Hamerly et al. [Hamer05] present a technique to reduce the runtime of architectural simulation by exploiting repetitive behaviors in programs. A central assumption of this approach is that all iterations of a repetitive *instruction interval* (i.e., a section of continuous application execution) exhibit similar behavior, which allows to evaluate an instruction interval once to represent all remaining instances. The approach identifies such representative instruction intervals in an offline analysis step by examining the execution frequencies for different regions of code. The identified instruction intervals are clustered according to their execution frequency in complete application executions to allow weighted parameter extrapolation.

The authors of [Wunde03] present a method that exploits statistical sampling theory to reduce the runtime of architectural simulation. In the course of a simulation, the approach alternates between detailed and functional simulation modes provided by a custom system-level simulation framework for fixed instruction intervals in an application. The approach selects minimal numbers of instruction intervals for detailed simulations to estimate parameters like performance and power while satisfying a required confidence interval.

The discussed related works [Hsieh98,Wunde03,Hamer05] rely on significantly reduced portions of the original application, namely *synthesized programs* and *selected instruction intervals* to estimate the power dissipation of complete application executions. These approaches assume that the evaluated instructions exhibit similar behavior compared to the original application. Approximation techniques like [Zhang14a] exploit changing degrees of approximation errors to ensure acceptable application outputs. However, changing degrees of approximation errors induce different behaviors over time. Previously repetitive behaviors that are exhibited by iterative methods, for instance, can only be limited to iteration intervals with unchanged approximation error. At the same time, the evaluation of a limited instruction interval does not necessarily reflect the impact of approximation errors on the power dissipation, since the induced approximation error propagates between executed instructions.

Chapter 6 presents parameter estimation methods that assess the compute efficiency of application executions on approximate computing hardware with respect to the area, the leakage power, the dynamic power, the delay, and the approximation error. To evaluate long-running iterative algorithmic parts, highly accurate but slow circuit simulations are combined with light-weight models of approximate computing hardware. This approach exploits the insight that parameter estimations can achieve low estimation runtimes and high estimation accuracy using carefully selected instruction intervals as discussed in [Hsieh98, Wunde03, Hamer05]. The underlying model is based on operand value truncation in accordance to [Chipp13, Mishr14, Barba16]. At the same time, the impact of approximation errors on the power dissipation is considered by evaluating the propagation of approximation errors throughout the application execution.

Chapter 3

Efficient Fault-Tolerant Sparse Matrix-Vector Multiplication

A fault tolerance technique for *sparse matrix-vector multiplications* (SpMV) is presented in this chapter that allows the efficient algorithmic detection and correction of erroneous operation results with high error coverage [Schol16a]. To achieve high runtime efficiency, an important challenge is to derive an error detection and correction scheme with low runtime overhead. At the same time, additional challenges arise to avoid false positive error detections due to rounding errors. To achieve high error coverage, suitable rounding error bounds are required to distinguish errors in the magnitude of rounding errors from presumably harmful errors.

The fault tolerance technique is based on the observation that even high error rates typically do not cause errors in complete matrix-vector multiplication results, but only in small parts. Instead of repeating entire computations or performing expensive error localization steps, the underlying error detection scheme is instrumented to implicitly provide the error locations with high efficiency. This technique enables partial recomputations just for erroneous outputs directly after error detection, which reduces the overall runtime overhead to provide fault tolerance.

This fault tolerance technique is algorithm-based and exploits the properties of sparse matrices to instrument error detection steps with error localization capabilities. Rounding errors are addressed by an analytical error bound that provides suitable rounding error bounds for the SpMV operation.

Related algorithm-based fault tolerance approaches for sparse matrix operations detect and correct errors, but often perform expensive error localization steps or rely on complete recomputations. General *checkpointing* techniques write the state of an application periodically to a fault-tolerant storage and restart the application from a prior state if an error is detected. However, such techniques can induce large recovery cost in both transferring checkpoint data and recomputing lost results for high error rates. Therefore, checkpointing techniques will become increasingly impractical as they induce significant bottlenecks for the execution of applications [Liu11].

Related *algorithmic error detection and correction* approaches compute and evaluate checksums to detect errors and recompute erroneous results by recomputation. Such approaches avoid the cost induced by rolling back to a prior state. Since even under high error rates, only a small portion of the output is corrupted by errors, these approaches reduce error correction cost by locating and recomputing only erroneous results. This *partial recomputation* approach, however, requires additional error localization steps to avoid unnecessary correction cost. Additional error localization costs may be acceptable for small outputs, but become unacceptable for large output sizes.

The remainder of this chapter is organized as follows. Section 3.1 below introduces the formal background for the presented fault tolerance technique. An analytical rounding error bound for sparse matrix operations is discussed below in Section 3.2. Section 3.3 presents the corresponding algorithmic steps. Section 3.4 presents the details of the underlying preprocessing, error detection and correction steps. This chapter concludes with a discussion on the runtime and memory overhead in Section 3.5. The presented fault tolerance technique is evaluated in Chapter 7, Section 7.4.

3.1 Method Overview

The presented fault tolerance technique divides the SpMV operation into *small blocks* with respect to a selected *row block size* (cf. Equation 2.4) and performs checksum-based error detection for each block separately. This block-based error detection scheme was evaluated for dense matrix operations in [Rexfo94]. Blocks for which the *operand* *checksums* do not match the corresponding *result checksums* are marked erroneous. For this reason, this approach delimits error locations to blocks of result elements instead of locating the erroneous result element exactly. These erroneous blocks are corrected by recomputing the SpMV multiplications *partially* for these blocks. Therefore, error locations are already determined during error detection which avoids both complete recomputation and error localization steps. Since this block-based approach exploits the sparsity of the underlying input matrix, the runtime overhead to detect and locate errors is significantly reduced.

The following equations and examples introduce the fault tolerance technique for the matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$, in which $\boldsymbol{A}$ denotes a real $(n \times n)$-matrix and $\boldsymbol{r}$, $\boldsymbol{b}$ denote $(n \times 1)$-vectors. The underlying concept can be transferred to general matrices without loss of generality. The partitioning of the SpMV multiplication into row blocks as well as the generation of checksums for these blocks relies on a *weight matrix* $\boldsymbol{W}$, which is computed as follows:

Let a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be partitioned into m' row block matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \cdots \boldsymbol{A}_{m'}$ with $1 \leq m' \leq n$. The *row block size* σ_k denotes the number of rows in row block matrix $\boldsymbol{A}_k$ with $1 \leq k \leq m'$. For each row block matrix $\boldsymbol{A}_k$, a *weight vector* $\boldsymbol{w}^{(k)}$ is a $(1 \times \sigma_k)$-vector, which contains weights $[w^{(k)}]_i \in \mathbb{R}_{\neq 0}$. The weights are non-zero real numbers used to encode the row block matrices.

A *weight matrix* $\boldsymbol{W}$ is an $(m' \times n)$-matrix that is formed by m' weight vectors $\boldsymbol{w}^{(k)}$ with $1 \leq k \leq m'$ as

$$\boldsymbol{W} := \begin{bmatrix} \boldsymbol{w}^{(1)} & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \boldsymbol{w}^{(2)} & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \boldsymbol{w}^{(k)} & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & \boldsymbol{w}^{(m')} \end{bmatrix}. \tag{3.1}$$

Each row k in a weight matrix $\boldsymbol{W}$ is formed by weight vector $w^{(k)}$. At the same time, each column in $\boldsymbol{W}$ contains one non-zero element, namely one weight from a weight vector.

Example 3.1: The (6×6)-sparse matrix $\boldsymbol{A}$ is multiplied by a (6×1)-operand vector $\boldsymbol{b}$ to obtain the (6×1)-result vector $\boldsymbol{r}$:

Zero elements in matrices and vectors are denoted by $(\,\cdot\,)$.

$$\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b} = \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & 2 \\ \cdot & 5 & 2 & \cdot & \cdot & \cdot \\ \cdot & 2 & 3 & 5 & \cdot & \cdot \\ \cdot & \cdot & 5 & 4 & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 2 & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & 9 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 3 \\ 5 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 26 \\ 42 \\ 36 \\ 8 \\ 18 \end{bmatrix}.$$

Matrix $\boldsymbol{A}$ is partitioned in three row block matrices $\boldsymbol{A}_1, \boldsymbol{A}_2$, and $\boldsymbol{A}_3$ with row block sizes $\sigma_1 = \sigma_2 = \sigma_3 = 2$.

$$\boldsymbol{A} := \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & 2 \\ \cdot & 5 & 2 & \cdot & \cdot & \cdot \\ \cdot & 2 & 3 & 5 & \cdot & \cdot \\ \cdot & \cdot & 5 & 4 & \cdot & 1 \\ \cdot & \cdot & \cdot & \cdot & 2 & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & 9 \end{bmatrix} = \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \boldsymbol{A}_3 \end{bmatrix} \quad \Leftrightarrow \quad \begin{aligned} \boldsymbol{A}_1 &:= \begin{bmatrix} 1 & \cdot & \cdot & \cdot & \cdot & 2 \\ \cdot & 5 & 2 & \cdot & \cdot & \cdot \end{bmatrix} \\ \boldsymbol{A}_2 &:= \begin{bmatrix} \cdot & 2 & 3 & 5 & \cdot & \cdot \\ \cdot & \cdot & 5 & 4 & \cdot & 1 \end{bmatrix} \\ \boldsymbol{A}_3 &:= \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & 2 & \cdot \\ 2 & \cdot & \cdot & 1 & \cdot & 9 \end{bmatrix}. \end{aligned}$$

Using the row block sizes $\sigma_1 = \sigma_2 = \sigma_3 = 2$, a (1×2)-weight vector $\boldsymbol{w}^{(k)}$ is formed for each row block matrix $\boldsymbol{A}_k$.

In this example, all weights are set to 1:

$$\boldsymbol{w}^{(1)} := \begin{bmatrix} 1 & 1 \end{bmatrix} \quad \boldsymbol{w}^{(2)} := \begin{bmatrix} 1 & 1 \end{bmatrix} \quad \boldsymbol{w}^{(3)} := \begin{bmatrix} 1 & 1 \end{bmatrix}.$$

Following Equation 3.1, a (3×6)-weight matrix $\boldsymbol{W}$ is formed by the weight vectors.

$$\boldsymbol{W} := \begin{bmatrix} \boldsymbol{w}^{(1)} & \cdot & \cdot \\ \cdot & \boldsymbol{w}^{(2)} & \cdot \\ \cdot & \cdot & \boldsymbol{w}^{(3)} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix}.$$

▶

A row block matrix $\boldsymbol{A}_k$ is encoded using weight vector $\boldsymbol{w}^{(k)}$ to compute a $(1 \times n)$-*checksum vector* $\boldsymbol{c}^{(k)}$ as

$$\boldsymbol{c}^{(k)} := \boldsymbol{w}^{(k)} \boldsymbol{A}_k \,. \tag{3.2}$$

The structure of a weight matrix $\boldsymbol{W}$ allows to encode each row block matrix $\boldsymbol{A}_k$ within a single sparse matrix-matrix multiplication $\boldsymbol{WA}$. The generated checksum vectors form the $(m' \times n)$-checksum matrix $\boldsymbol{C}$ with

$$\boldsymbol{C} := \boldsymbol{WA} = \begin{bmatrix} \boldsymbol{w}^{(1)} & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & \boldsymbol{w}^{(2)} & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \boldsymbol{w}^{(k)} & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & \cdots & 0 & \boldsymbol{w}^{(m')} \end{bmatrix} \begin{bmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \vdots \\ \boldsymbol{A}_k \\ \vdots \\ \boldsymbol{A}_{m'} \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}^{(1)}\boldsymbol{A}_1 \\ \boldsymbol{w}^{(2)}\boldsymbol{A}_2 \\ \vdots \\ \boldsymbol{w}^{(k)}\boldsymbol{A}_k \\ \vdots \\ \boldsymbol{w}^{(m')}\boldsymbol{A}_{m'} \end{bmatrix} . \tag{3.3}$$

Following Equation 3.2, row k in $\boldsymbol{C}$ contains the checksum vector $\boldsymbol{c}^{(k)}$:

$$\boldsymbol{C} = \begin{bmatrix} \boldsymbol{w}^{(1)}\boldsymbol{A}_1 \\ \boldsymbol{w}^{(2)}\boldsymbol{A}_2 \\ \vdots \\ \boldsymbol{w}^{(k)}\boldsymbol{A}_k \\ \vdots \\ \boldsymbol{w}^{(m')}\boldsymbol{A}_{m'} \end{bmatrix} = \begin{bmatrix} \boldsymbol{c}^{(1)} \\ \boldsymbol{c}^{(2)} \\ \vdots \\ \boldsymbol{c}^{(k)} \\ \vdots \\ \boldsymbol{c}^{(m')} \end{bmatrix} . \tag{3.4}$$

Example 3.2: (*Continues Example* 3.1)
The weight matrix $\boldsymbol{W}$ is multiplied with $\boldsymbol{A}$ to obtain the (3×6)-checksum matrix $\boldsymbol{C}$ following Equation 3.3.

$$\boldsymbol{C} := \boldsymbol{WA} = \begin{bmatrix} 1 & 5 & 2 & \cdot & \cdot & 2 \\ \cdot & 2 & 8 & 9 & \cdot & 1 \\ 2 & \cdot & \cdot & 1 & 2 & 9 \end{bmatrix} .$$

▸

The weight matrix $\boldsymbol{W}$ and the checksum matrix $\boldsymbol{C}$ are used to compute checksums for the operands $\boldsymbol{Ab}$ and checksums for the result $\boldsymbol{r}$. An *operand checksum vector* $\boldsymbol{t}$ is an

$(m' \times 1)$-vector that is computed for the operands in a matrix-vector multiplication $\boldsymbol{Ab}$ as

$$\boldsymbol{t} := \boldsymbol{Cb}\ . \tag{3.5}$$

A *result checksum vector* $\boldsymbol{t}^*$ is an $(m' \times 1)$-vector that is computed for the results of a matrix-vector multiplication $\boldsymbol{Ab}$ as

$$\boldsymbol{t}^* := \boldsymbol{Wr}\ . \tag{3.6}$$

Example 3.3: (*Continues Example* 3.2)
A (3×1)-operand checksum vector $\boldsymbol{t}$ is computed following Equation 3.5 as

$$\boldsymbol{t} := \boldsymbol{Cb} = \begin{bmatrix} 1 & 5 & 2 & \cdot & \cdot & 2 \\ \cdot & 2 & 8 & 9 & \cdot & 1 \\ 2 & \cdot & \cdot & 1 & 2 & 9 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 3 \\ 5 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix}$$

and a (3×1)-result checksum vector $\boldsymbol{t}^*$ is computed following Equation 3.6 as

$$\boldsymbol{t}^* := \boldsymbol{Wr} = \begin{bmatrix} 1 & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 26 \\ 42 \\ 36 \\ 8 \\ 18 \end{bmatrix} = \begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix}.$$

▶

A *syndrome vector* $\boldsymbol{s}$ is an $(m' \times 1)$-vector that is computed from the checksum vectors $\boldsymbol{t}$ and $\boldsymbol{t}^*$ as

$$\boldsymbol{s} := \boldsymbol{t} - \boldsymbol{t}^*\ . \tag{3.7}$$

The syndrome vector allows both error detection and localization. If $\boldsymbol{r}$, $\boldsymbol{A}$, and $\boldsymbol{b}$ are related by $\boldsymbol{r} = \boldsymbol{Ab}$, then the syndrome vector $\boldsymbol{s} \in \mathbb{R}^{m' \times 1}$ is equal to the zero vector $\overline{0}$:

$$\boldsymbol{s} = \overline{0}. \tag{3.8}$$

Proof: Equation 3.8 follows from applying Equations 3.3, 3.5, and 3.6 to Equation 3.7:

$$\begin{aligned} & \boldsymbol{s} := \boldsymbol{t} - \boldsymbol{t}^* = \boldsymbol{Cb} - \boldsymbol{Wr} = (\boldsymbol{WA})\boldsymbol{b} - \boldsymbol{Wr} = \boldsymbol{W}(\boldsymbol{Ab} - \boldsymbol{r}) \\ (\text{with } \boldsymbol{r} = \boldsymbol{Ab}): \quad & \boldsymbol{s} = \boldsymbol{W}(\overline{0}) = \overline{0}. \end{aligned}$$

□

Example 3.4: (*Continues Example* 3.3)
A (3×1)-syndrome vector $\boldsymbol{s}$ is computed from $\boldsymbol{t}$ and $\boldsymbol{t}^*$ following Equation 3.7 as

$$\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}^* = \begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix} - \begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} .$$

▸

If a syndrome vector element $s_k \in \boldsymbol{s}$ is not zero, then the relation between the row block matrix $\boldsymbol{A}_k$, the operand vector $\boldsymbol{b}$ and the result block vector $\boldsymbol{r}_k$ does not hold:

$$s_k \neq 0 \quad \Leftrightarrow \quad \boldsymbol{r}_k \neq \boldsymbol{A}_k \boldsymbol{b} . \tag{3.9}$$

Proof:

$$s_k = t_k - t_k^* = [\boldsymbol{Cb}]_k - [\boldsymbol{Wr}]_k = [\boldsymbol{C}]_k \boldsymbol{b} - [\boldsymbol{W}]_k \boldsymbol{r} .$$

Following Equation 3.3, row k in a checksum matrix $\boldsymbol{C}$ is computed as

$$[\boldsymbol{C}]_k := \boldsymbol{c}^{(k)} = \boldsymbol{w}^{(k)} \boldsymbol{A}_k$$

which allows to write the term $[\boldsymbol{C}]_k \boldsymbol{b}$ as

$$[\boldsymbol{C}]_k \boldsymbol{b} = (\boldsymbol{w}^{(k)} \boldsymbol{A}_k) \boldsymbol{b} .$$

The term $[\boldsymbol{W}]_k \boldsymbol{r}$ is computed as

$$[\boldsymbol{W}]_k \boldsymbol{r} = \sum_{i=1}^{n} w_{k,i} \cdot r_i .$$

From Equation 2.5 follows that the result vector $\boldsymbol{r}$ is partitioned into block vectors $\boldsymbol{r}_1, \boldsymbol{r}_2, \cdots \boldsymbol{r}_{m'}$ such that the block size of $\boldsymbol{r}_k$ is equal to the row block size of $\boldsymbol{A}_k$ with $1 \leq k \leq m'$. Following Equation 3.1, row k in a weight matrix $\boldsymbol{W}$ contains the weight vector $\boldsymbol{w}^{(k)}$ which is enclosed by zero elements. Since only the multiplication of the vector $\boldsymbol{w}^{(k)}$ with the row block $\boldsymbol{r}_k$ can result in a non-zero result:

$$[\boldsymbol{W}]_k \boldsymbol{r} = \sum_{i=1}^{n} w_{k,i} \cdot r_i = \boldsymbol{w}^{(k)} \boldsymbol{r}_k \, .$$

The syndrome vector element s_k only depends on the relationship between the row block matrix $\boldsymbol{A}_k$, the operand vector $\boldsymbol{b}$ and the result block vector $\boldsymbol{r}_k$:

$$s_k = [\boldsymbol{C}]_k \boldsymbol{b} - [\boldsymbol{W}]_k \boldsymbol{r} = (\boldsymbol{w}^{(k)} \boldsymbol{A}_k) \boldsymbol{b} - \boldsymbol{w}^{(k)} \boldsymbol{r}_k = \boldsymbol{w}^{(k)} (\boldsymbol{A}_k \boldsymbol{b} - \boldsymbol{r}_k) \, .$$

Assuming that the weight vectors are not zero, $\boldsymbol{w}^{(k)} \neq \overline{0}$:

$$s_k \neq 0 \quad \Leftrightarrow \quad \boldsymbol{A}_k \boldsymbol{b} - \boldsymbol{r}_k \neq 0 \quad \Leftrightarrow \quad \boldsymbol{r}_k \neq \boldsymbol{A}_k \boldsymbol{b} \, .$$

□

Since the syndrome vector element s_k only relies on the relation between the row block matrix $\boldsymbol{A}_k$, the operand vector $\boldsymbol{b}$ and the result block vector $\boldsymbol{r}_k$, errors in the result of the matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$ can be delimited to $\boldsymbol{r}_k := \boldsymbol{A}_k \boldsymbol{b}$ for which $s_k \neq 0$.

Example 3.5: Assume that the fourth result element r_4 is corrupted during the matrix-vector multiplication from Example 3.1. The second block vector $\boldsymbol{r}'_2$ contains the error:

$$\boldsymbol{r} := \begin{bmatrix} 4 \\ 26 \\ 42 \\ 36 \\ 8 \\ 18 \end{bmatrix} \qquad \boldsymbol{r}' := \begin{bmatrix} 4 \\ 26 \\ 42 \\ 34 \\ 8 \\ 18 \end{bmatrix} \quad \Leftrightarrow \quad \begin{matrix} \boldsymbol{r}'_1 = \begin{bmatrix} 4 \\ 26 \end{bmatrix} \\ \boldsymbol{r}'_2 = \begin{bmatrix} 42 \\ 34 \end{bmatrix} \\ \boldsymbol{r}'_3 = \begin{bmatrix} 8 \\ 18 \end{bmatrix} \end{matrix}$$

In the corresponding syndrome vector $\boldsymbol{s}$, the second syndrome element s_2 is not zero.

$$\boldsymbol{s} = \boldsymbol{t}^* - \boldsymbol{t} = \boldsymbol{W}\boldsymbol{r}' - \boldsymbol{C}\boldsymbol{b} = \begin{bmatrix} 30 \\ 76 \\ 26 \end{bmatrix} - \begin{bmatrix} 30 \\ 78 \\ 26 \end{bmatrix} = \begin{bmatrix} 0 \\ -2 \\ 0 \end{bmatrix}$$

With $s_2 \neq 0$, the error location is delimited to $\boldsymbol{r}'_2$.

▶

When the matrix-vector multiplication is executed in floating-point arithmetic, rounding errors can occur that induce small differences between the checksum vector elements t_k and t_k^*. To distinguish errors in the magnitude of rounding errors from errors that may be harmful to the application, each syndrome element s_k is compared against a rounding error bound, which is computed as follows:

Let $fl(t_k) \in \mathbb{R}$ and $fl(t_k^*) \in \mathbb{R}$ denote the floating-point representations of t_k and t_k^*. A *rounding error bound vector* $\boldsymbol{\tau}$ is an $(m' \times 1)$-vector and its elements $\tau_k \in \mathbb{R}$ are upper bounds for the maximum difference between the floating-point representations of the checksums t_k and t_k^*:

$$|fl(t_k^*) - fl(t_k)| \quad < \tau_k \,. \tag{3.10}$$

Section 3.2 below presents an *analytical rounding error bound* for sparse matrix-vector multiplications.

3.2 Analytical Rounding Error Bound for Sparse Matrices

Rounding errors create a major challenge for fault tolerance techniques, because these inevitable errors can cause false positives when checksums are directly compared for equality. Error detection schemes need to consider the impact of rounding errors as they typically cause small differences in the checksums. Instead of comparing checksums directly, the difference between checksums has to be compared to suitable rounding error bounds τ that cope with such differences. Error bounds τ that are chosen smaller than the actual rounding error cause false positive error detections and can trigger unnecessary corrections. Too large bounds τ can lead to undetected errors (i.e., false negatives) that may harm the final result in the application. For instance, such errors may significantly increase execution times or lead to silent data corruption in the case of iterative solvers [Brone08, Shant11].

An important metric that is used to quantify the rounding error in floating-point arithmetic is the *machine epsilon* [Mulle10]. The *machine epsilon* ε_M is an upper bound on the relative error due to rounding for *normal numbers* in floating point arithmetic. For *directed rounding modes*, the machine epsilon ε_M is computed from the number of significant digits in a floating-point representation $p \geq 2$ (i.e., *precision*) with

$$\varepsilon_M := 2^{-(p-1)} . \tag{3.11}$$

To keep the introduction for the term *machine epsilon* concise, the interested reader finds a comprehensive discussion of floating-point arithmetic in Appendix C.

Chowdhury and Banerjee present an analytical rounding error bound [Chowd96] for dense matrix-vector multiplications that are protected by ABFT as described in Equations 2.34 to 2.36. This rounding error bound estimates the maximum difference between the operand and result checksums that is constituted by the accumulation of rounding errors in the different underlying operations. These operations include the norms of operand $\boldsymbol{b}$, checksum vector $\boldsymbol{c}$ (i.e., $\boldsymbol{w}^T\boldsymbol{A}$) as well as the rows in input matrix $\boldsymbol{A}$. Using repeated applications of the triangle inequality, the submultiplicative inequality and the *Cauchy-Schwarz* inequality, the authors compose the *analytical rounding error bounds* for these operations (i.e., inner products and matrix-vector products) to form an estimation for the maximum difference between these checksums.

Let $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^{n \times 1}$ and $\boldsymbol{r} \in \mathbb{R}^{n \times 1}$ that are related by $\boldsymbol{r} = \boldsymbol{Ab}$. Besides, let $\boldsymbol{c_c}$ denote the $(1 \times n)$-*column checksum vector* (i.e., $\boldsymbol{w}^T\boldsymbol{A}$) and let ε_M denote the machine epsilon. The *rounding error bound based on simplified error analysis* τ_{SEA} is a scalar that is computed as

$$\tau_{SEA} := \left((n + 2 \cdot m - 2) \cdot \sum_{i=1}^{m} \|a_i\|_2 + n \cdot \|\boldsymbol{c_c}\|_2\right) \cdot \varepsilon_M \cdot \|\boldsymbol{b}\|_2 \tag{3.12}$$

with $\|a_i\|_2$ denoting the norm of the i-th *row* in matrix $\boldsymbol{A}$.

The difference between the operand and result checksums is typically smaller for sparse matrices, because sparse matrices contain a large portion of zero elements which do not contribute to the rounding error. For this reason, the error bounds derived by this approach are too loose for sparse problems.

The presented fault tolerance technique relies on an analytical rounding error bound for sparse matrix-vector multiplications that is based on the following insight: Instead of assuming that each block $\boldsymbol{A}_k$ contains non-zero elements in all n columns, the actual number of non-empty columns n'_k is utilized to estimate the maximum difference between the operand and result checksums. This estimation provides tighter error bounds since it considers the actual number of elements that contribute to the rounding error with $n'_k < n$ for sparse row block matrices. The presented fault tolerance technique utilizes the following analytical rounding error bound for sparse matrix-vector multiplications.

Given $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b} \in \mathbb{R}^{n \times 1}$ and $\boldsymbol{r} \in \mathbb{R}^{n \times 1}$. Let $\boldsymbol{A}$ and $\boldsymbol{r}$ be partitioned into m' blocks with σ_k being the number of rows in block k. Let $\boldsymbol{w}^{(k)}$ be the k-th $(1 \times \sigma_k)$-weight vector with $1 \leq k \leq m'$ and let $\boldsymbol{C} \in \mathbb{R}^{m' \times n}$ denote the checksum matrix. Besides, let n'_k denote the number of non-empty columns in row block matrix $\boldsymbol{A_k}$ and let ε_M be the machine epsilon. The *rounding error bound for sparse matrix operations* $\boldsymbol{\tau}$ is an $(m' \times 1)$-vector and its elements τ_k are computed as:

$$\tau_k := \|\boldsymbol{b}\|_2 \cdot \varepsilon_M \cdot \left((n'_k + 2 \cdot \sigma_k) \sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 + n'_k \cdot \|[\boldsymbol{C}]_k\|_2\right) . \tag{3.13}$$

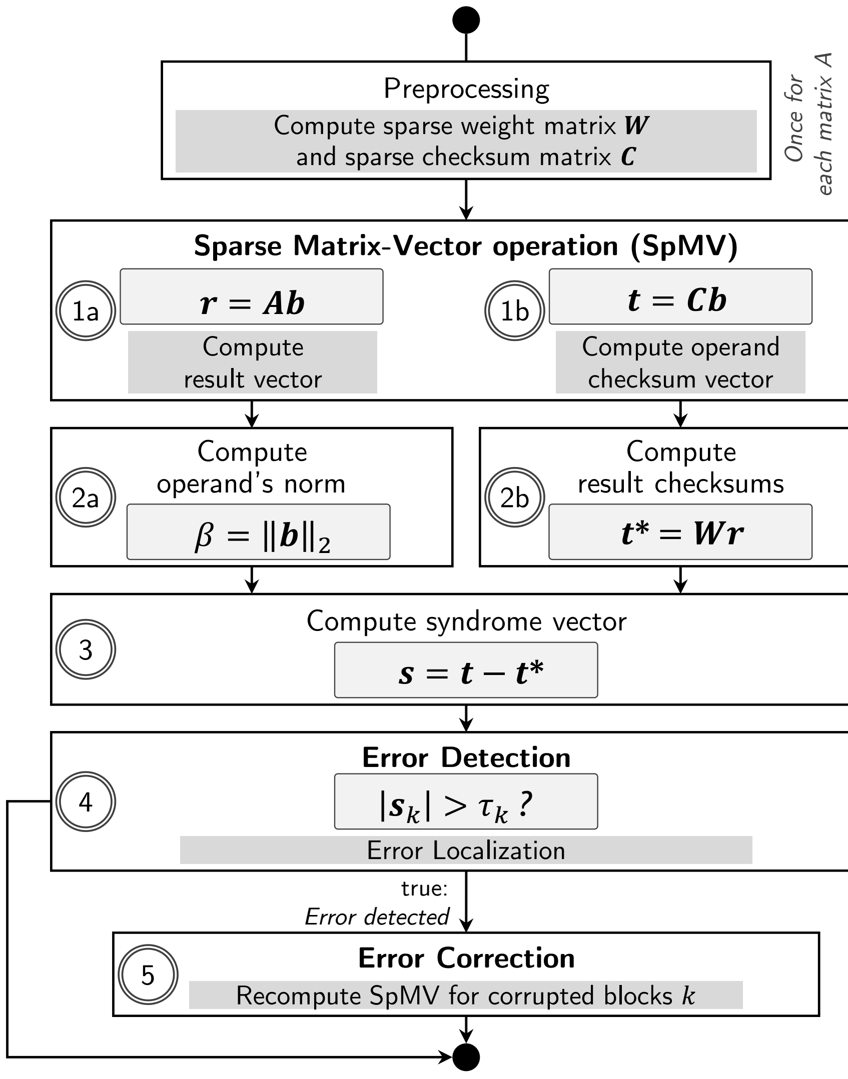
To keep this discussion concise, the interested reader finds the derivation of Equation 3.13 in Appendix D.1.

3.3 Algorithmic Steps

The underlying algorithmic steps of the presented fault-tolerant sparse matrix-vector multiplication are based on the equations introduced in Section 3.1 above. Given a sparse matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$, the fault tolerance technique *preprocesses* the operation and performs error *detection* and *correction* steps in the course of the original operation.

An overview of the algorithmic steps that are performed for each fault-tolerant sparse matrix-vector multiplication is shown in Figure 3.1. Operations that can be executed in parallel to each other are depicted in common rows.

In a *preprocessing step*, the weight matrix $\boldsymbol{W}$ is computed to encode the input matrix $\boldsymbol{A}$ following Equation 3.1. Besides, the checksum matrix $\boldsymbol{C}$ is computed as $\boldsymbol{C} := \boldsymbol{W}\boldsymbol{A}$ following Equation 3.3. The structure of the weight matrix $\boldsymbol{W}$ ensures that each row k


Preprocessing
Compute sparse weight matrix $\boldsymbol{W}$ and sparse checksum matrix $\boldsymbol{C}$
Once for each matrix A
Sparse Matrix-Vector operation (SpMV)
1a
$\boldsymbol{r} = \boldsymbol{A}\boldsymbol{b}$
Compute result vector
1b
$\boldsymbol{t} = \boldsymbol{C}\boldsymbol{b}$
Compute operand checksum vector
2a
Compute operand's norm
$\beta = \|\boldsymbol{b}\|_2$
2b
Compute result checksums
$\boldsymbol{t}^* = \boldsymbol{W}\boldsymbol{r}$
3
Compute syndrome vector
$\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}^*$
4
Error Detection
$|\boldsymbol{s}_k| > \tau_k$?
Error Localization
true: Error detected
5
Error Correction
Recompute SpMV for corrupted blocks k

▲ **Figure 3.1** — Overview of the algorithmic steps in the fault-tolerant sparse matrix-vector multiplication.

in the checksum matrix $\boldsymbol{C}$ contains the *column checksums* for a specific block $\boldsymbol{A}_k$. As discussed in Chapter 2.3.1, different approaches exist to set the weight elements. Unless otherwise stated, the weight elements are set to 1 following [Brone08, Shant12, Sloan12, Sloan13]. As a result, the checksum matrix $\boldsymbol{C}$ inherits the sparsity of the input matrix $\boldsymbol{A}$. By exploiting this sparsity, this block-based approach reduces the runtime overhead to detect errors compared to related approaches that rely on traditional ABFT (cf. Chapter 2.3.1).

In the *first step* of the fault-tolerant sparse matrix-vector multiplication, the original operation $\boldsymbol{r} = \boldsymbol{A}\boldsymbol{b}$ computes the result vector $\boldsymbol{r}$. Parallel to this operation, the operand checksum vector $\boldsymbol{t}$ is computed with $\boldsymbol{t} = \boldsymbol{Cb}$ following Equation 3.5. Each element t_k stores the checksum for k-th block in the original SpMV operation $\boldsymbol{A}_k\boldsymbol{b}$.

The second and third steps compute different error detection variables that are required to check the results. The operand norm β is computed in the second step, which is required to determine the rounding error bounds for error detection. After the first step encoded the operands, new checksums are calculated for the results and stored in the result checksum vector $\boldsymbol{t}^* := \boldsymbol{Wr}$ following Equation 3.6. Each element t_k^* stores the checksum for result block $\boldsymbol{r}_k$. The third step computes the *syndrome vector* $\boldsymbol{s}$ by calculating the difference between the result checksum vector $\boldsymbol{t}^*$ and the operand checksum vector $\boldsymbol{t}$ following Equation 3.7.

To detect errors and delimit error locations, the fourth step compares each element of the syndrome vector s_k against the rounding error bound τ_k that is computed following Equation 3.13. During this error detection step, the location of errors is determined by the portion of result blocks k for which the syndrome exceeds the corresponding rounding error bound. These blocks contain at least one erroneous element. To correct these errors, these erroneous blocks are recomputed in the fifth step. Instead of recomputing the original SpMV operation completely, this error correction scheme recomputes this operation only *partially* in case of errors.

3.4 Error Detection and Correction

The error detection steps are prepared by encoding an input matrix $\boldsymbol{A}$ using a weight matrix $\boldsymbol{W}$ following Equation 3.1. The structure of the weight matrix (i.e., the location of non-zero weights in the matrix) ensures that the matrix $\boldsymbol{A}$ is encoded with respect to its partitioning for each of the m' row block matrices $\boldsymbol{A}_1, \boldsymbol{A}_2, \cdots, \boldsymbol{A}_{m'}$. The resulting checksums are combined in the checksum matrix $\boldsymbol{C}$ that is computed in a sparse matrix-matrix multiplication $\boldsymbol{C} := \boldsymbol{WA}$. The checksum matrix $\boldsymbol{C}$ has to be computed for each input matrix $\boldsymbol{A}$ only once. Further matrix-vector multiplications using matrix $\boldsymbol{A}$ can reuse the already computed checksum matrix $\boldsymbol{C}$. Besides, the structure of the weight matrix ensures that each row in the checksum matrix $[\boldsymbol{C}]_k$ contains the checksums for the corresponding row block matrix $\boldsymbol{A}_k$. The checksum matrix $\boldsymbol{C}$ allows to compute the *operand* checksums in a single additional SpMV operation which results in the *operand* checksum vector $\boldsymbol{t} := \boldsymbol{Cb}$. After the computation of the original operation $\boldsymbol{r} = \boldsymbol{Ab}$, *result* checksums $\boldsymbol{t}^*$ are computed by $\boldsymbol{t}^* := \boldsymbol{Wr}$. The structure of the weight matrix divides the result vector $\boldsymbol{r}$ into m' vector blocks $\boldsymbol{r}_k$ (i.e., $\boldsymbol{r}_k = \boldsymbol{A}_k\boldsymbol{b}$) during the computation of result checksums $\boldsymbol{t}^*$.

The error detection steps compute syndrome vector elements following Equation 3.7 as $s_k := t_k^* - t_k$ and evaluate them against rounding error bounds τ_k following Equation 3.13 to determine error locations already during error detection. According to Equation 3.8, the checksum vectors $\boldsymbol{t}$ and $\boldsymbol{t}^*$ are equal aside from rounding errors in the error-free case. The difference between these checksum vectors constitutes the syndrome vector $\boldsymbol{s} = \boldsymbol{t} - \boldsymbol{t}^*$ which is used to detect errors. Each syndrome element s_k is compared against its corresponding rounding error bound τ_k to distinguish errors in the magnitude of the rounding errors from errors that may be harmful to the application.

In case of errors, the fault tolerance technique recovers from errors by recomputing the original SpMV multiplication *partially.* Compared to recomputing the entire original operation or performing additional error localization steps, this approach induces low runtime overhead.

Parallel to the detection of errors, errors are located by determining the set of result vector blocks $\boldsymbol{r}_k$ for which the syndrome vector elements s_k exceed the corresponding round error bound τ_k. Errors are corrected by recomputing such erroneous blocks $\boldsymbol{r}_k$ with

$$\boldsymbol{r}_k = \boldsymbol{A}_k\boldsymbol{b} \quad \text{for } |s_k| > \tau_k \,. \tag{3.14}$$

Multiple erroneous row blocks $\{\boldsymbol{r}_i, \boldsymbol{r}_j, \ldots\}$ are corrected by recomputing each block individually.

In high error rate scenarios, additional errors can occur during error detection and correction steps. To avoid the propagation of such errors to the application, the error detection steps can be repeated after the error correction steps finished. This *error detection and correction cycle* is repeated until all checksum invariant violations are resolved.

However, in case of false positive error detections, infinite loops can occur as the underlying result and checksum values will not change during error detection and correction cycles. Such infinite loops can be avoided by storing the computed checksum vectors $\boldsymbol{t}$ and $\boldsymbol{t}^*$ after error detection and comparing new checksum vectors to these previously computed checksum vectors. As the impact of false positive errors on the checksums will typically not change between succeeding error detection events, the error detection and correction cycle is stopped, if succeeding checksums do not change. At the same time, errors caused by transient events are still detected reliably as these errors typically affect succeeding checksums to different extents.

3.5 Computational and Memory Overhead

The SpMV multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$ with $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ has a runtime complexity of $\mathcal{O}(NNZ)$ with NNZ being the number of non-zero elements in matrix $\boldsymbol{A}$. With $NNZ \approx n$, this SpMV operation can be of linear complexity (i.e., $\mathcal{O}(NNZ) \approx \mathcal{O}(n)$ and $NNZ \ll n^2$). The memory requirement is constituted by the number of non-zero elements NNZ in matrix $\boldsymbol{A}$, the number of elements in the operand vector $\boldsymbol{b}$ with n elements and the number of elements in the result vector $\boldsymbol{r}$ with n elements.

Both the computational and memory overhead induced by this fault-tolerance technique depends on the number of row block matrices m' into which the original matrix $\boldsymbol{A}$ is divided. During preprocessing, the $m' \times n$-weight matrix $\boldsymbol{W}$ with n elements is multiplied with the $n \times n$-input matrix $\boldsymbol{A}$ with NNZ elements to obtain the $m' \times n$ checksum matrix C. The runtime complexity for the preprocessing step is $\mathcal{O}(NNZ)$. While the space complexity for the weight matrix $\boldsymbol{W}$ is $\mathcal{O}(n)$, the space complexity for the checksum matrix $\boldsymbol{C}$ depends on the block sizes σ_k and the distribution of the non-zero elements in matrix $\boldsymbol{A}$. Let $\max(n')$ be the maximum number of non-zero columns in the row block matrices $\boldsymbol{A}_k$, then the space complexity for the checksum matrix $\boldsymbol{C}$ is $\mathcal{O}(m' \cdot \max(n'))$. The actual number of non-zero elements in the checksum matrix $\boldsymbol{C}$ ranges from n for $m' = 1$ to NNZ for $m' = n$.

Both the weight matrix $\boldsymbol{W}$ and the checksum matrix $\boldsymbol{C}$ have to be computed only once for each input matrix $\boldsymbol{A}$. For this reason, applications that reuse the input matrix $\boldsymbol{A}$ repeatedly achieve even larger benefits from this fault tolerance technique. An important class of such applications are iterative solvers that typically dominate the runtime for many scientific applications.

The error detection steps have a runtime complexity of $\mathcal{O}(m' \cdot \max(n')) + \mathcal{O}(n)$ which is independent of the number of errors. During error detection, the operand checksum vector $\boldsymbol{t}$ is computed with $\boldsymbol{t} := \boldsymbol{C}\boldsymbol{b}$, which has a runtime complexity of $\mathcal{O}(m' \cdot \max(n'))$. The result checksum vector $\boldsymbol{t}^*$ is computed with $\boldsymbol{t}^* := \boldsymbol{W}\boldsymbol{r}$, which has a runtime complexity of $\mathcal{O}(n)$, since the weight matrix $\boldsymbol{W}$ contains at most n non-zero elements. The computation of the operand vector norm $\beta := \|\boldsymbol{b}\|_2$ has a runtime complexity of $\mathcal{O}(n)$. Computing and comparing the syndrome vector elements s_k with the rounding error bounds τ has a runtime complexity of $\mathcal{O}(m')$. Each rounding error bound τ_k is computed with a runtime complexity of $\mathcal{O}(1)$. The space complexity for error detection steps is constituted by the number of elements m' in the syndrome vector $\boldsymbol{b}$ with $\mathcal{O}(m')$.

The runtime complexity to correct errors depends on the number of non-zero elements in those row blocks $\{\boldsymbol{r}_k, \boldsymbol{r}_i, ...\}$ in which an error was detected. With NNZ_k being the number of non-zero elements in block $\boldsymbol{A}_k$, the runtime complexity to correct errors for $\boldsymbol{r}_k$ is $\mathcal{O}(NNZ_k)$. If all blocks are affected, then the runtime complexity for correction is at most $\mathcal{O}(NNZ)$, which corresponds to the runtime complexity of the original operation. Errors typically affect only a small part of the result vector elements even under high error rates. For this reason, the average expected runtime complexity for error correction is $\mathcal{O}(NNZ_k)$.

The total memory overhead to detect and correct errors is $n + m' \cdot (4 + \max(n'))$ elements which are constituted by the $m' \times n$-weight matrix $\boldsymbol{W}$ (i.e., n elements), the $m' \times n$ checksum matrix C (i.e., $m' \cdot \max(n')$ elements), the operand and result checksum vectors as well as the syndrome vector with m' elements each.

The number of blocks m' is an important parameter that determines both the runtime overhead as well as the error coverage. With larger blocks, the runtime overhead for error detection is reduced as the checksum matrix $\boldsymbol{C}$ contains fewer rows which reduces the runtime for computing $\boldsymbol{t}$ (i.e., $\boldsymbol{t} = \boldsymbol{Cb}$). However, in case of errors, the runtime overhead to correct errors increases as the row block matrices $\boldsymbol{A}_k$ contain more elements. At the same time, the error coverage can decrease as errors are more likely to be masked by larger rounding errors in the computation of checksums.

With the linear operations being computed on parallel hardware, the number of sequential steps dominates the runtime. While large blocks reduce the runtime to compute the operand vector $\boldsymbol{t}$, the runtime to compute $\boldsymbol{t}^*$ increases. Each element t_k^* is computed as an inner product which is typically implemented as a *reduction* on parallel computer architectures [Gallo15, p. 18]. With increasing number of elements in each inner product, the number of sequential reduction steps increases. At the same time, smaller blocks result in larger checksum matrices $\boldsymbol{C}$ but reduce the runtime to compute the result checksum vector $\boldsymbol{t}^*$.

Chapter 4

Efficient Fault Tolerance for the Conjugate Gradient Solvers

This chapter presents a fault tolerance technique for the Conjugate Gradient solvers that detects and corrects errors [Schol15], for instance, caused by transient events. The presented technique is suitable for both the CG and the PCG solver as the underlying assumptions are independent of the utilized preconditioning operations. A major challenge in achieving high runtime efficiency is to find an error detection scheme with low runtime overhead and high error coverage. In case of errors, a high error coverage allows low detection and correction latencies which reduces the number of additional iterations required to converge to a correct solution.

The presented approach exploits the property that arbitrary successive iterations in these solvers are related to each other by different inherent relations (cf. Equations 2.16 to 2.18 in Chapter 2.1.2). The convergence of these solvers and the correctness of the final result is ensured, if those relations are maintained throughout the whole execution. By evaluating these relations at runtime to detect and correct errors, repetitions of complete executions are avoided. The underlying error detection criteria are derived from these relations using only operations that induce low overhead compared to the original operations in the solvers. Since expensive operations like sparse matrix-vector multiplications are not required, this error detection scheme is very efficient. At the same time, the runtime overhead for error detection scales favorably with increasing number of non-zero elements in the input matrix. As the underlying relations are satisfied for arbitrary iterations in a solver execution, the induced runtime overhead can be further reduced by periodically evaluating these criteria. The error correction scheme of the proposed fault tolerance technique reduces the overhead to restore the solver execution by identifying the degree of corruption and trading off three different correction methods against each other. Whenever possible, online corrections (i.e., roll-forward recovery) are preferred to two different roll-back schemes.

As Chapter 2.3.2 discussed at the example of related works like [Brone08, Shant11], the Conjugate Gradient solvers are still vulnerable to transient effects. Even single errors can degrade the solver performance by factors of up to 200x and cause silent data corruptions in which the derived solution may not satisfy the original problem $\boldsymbol{Ax} = \boldsymbol{b}$, despite apparent convergence. Different fault tolerance approaches were presented in the related work that tackle the vulnerability of these solvers with different strategies. One approach is to repeat the solver execution if the computed solution $\boldsymbol{x}$ does not satisfy the original problem $\boldsymbol{Ax} = \boldsymbol{b}$. However, this approach waits until the solver converged to a solution before errors are detected. While different approaches detect errors at runtime, they induce significant runtime overheads since they rely on additional expensive matrix-vector multiplications or perform additional inner products after each solver iteration to detect errors. Besides, some approaches solely rely on traditional checkpointing techniques, which induce high cost in recomputing erroneous results compared to immediate correction approaches.

The presented fault tolerance technique is discussed in this chapter as follows. Section 4.1 below introduces the formal background for this technique. The following Sections 4.2 and 4.3 discuss the underlying error detection and corrections schemes. Section 4.4 presents an overview of the corresponding algorithmic steps. Finally, this chapter is concluded by a discussion on the runtime and memory overhead in Section 4.5. This fault tolerance technique is evaluated in Chapter 7.5 by experimental results.

4.1 Method Overview

As discussed in Chapter 2.1.2, the Conjugate Gradient methods solve linear systems of the form $\boldsymbol{Ax} = \boldsymbol{b}$ with $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{b} \in \mathbb{R}^n$ iteratively. In the following, the solver iterations are denoted by i, j, and k.

The presented fault tolerance technique instruments the Conjugate Gradient solvers by additional operations that *periodically* check for errors and generate checkpoints. The error detection scheme evaluates two different criteria, which are referred to as *lambda* and *sigma error checking criteria* below.

The *error checking interval* t determines the number of iterations that are executed between two successive error detection iterations. For two solver iterations k and i, with $k \neq i$, the *lambda* and *sigma* error checking criteria are periodically evaluated with

$$k = i + t \qquad \text{with } t \geq 1 \,. \tag{4.1}$$

The *lambda* error checking criterion relies on the periodic computation of two checksums. Let $\boldsymbol{p}^{(i)} \in \mathbb{R}^n$ be the search direction in iteration i.
A *lambda checksum* $\lambda^{(i)}$ is a scalar that is computed in iteration i as

$$\lambda^{(i)} := \boldsymbol{b}^T \boldsymbol{p}^{(i)} \,. \tag{4.2}$$

The *sigma checksum* $\sigma^{(i)}$ is a scalar that is computed in iteration i as

$$\sigma^{(i)} := \|\boldsymbol{p}^{(i)}\|_2 \,. \tag{4.3}$$

During the execution of the Conjugate Gradient solver, the following equations hold, which describe inherent relations between different solver iterations. Let $\boldsymbol{w}^{(i)} \in \mathbb{R}^n$ be the result of $\boldsymbol{A}\boldsymbol{p}^{(i)}$ in iteration i. Besides, let $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ be the intermediate result in iteration k. The *lambda checksum relation* holds for $\lambda^{(i)}$, $\boldsymbol{w}^{(i)}$, and $\boldsymbol{x}^{(k)}$ in an error-free scenario:

$$\boldsymbol{x}^{(k)T} \boldsymbol{w}^{(i)} = \lambda^{(i)} \qquad \text{for } k > i \,. \tag{4.4}$$

Proof: Let i and k denote two different iterations in a Conjugate Gradient solver with $k > i$. Following Equation 2.17, the residual vector $\boldsymbol{r}^{(k)}$ is orthogonal to each preceding search direction vector $\boldsymbol{p}^{(i)}$ in the error-free case with

$$\boldsymbol{r}^{(k)} \perp \boldsymbol{p}^{(i)} \iff \boldsymbol{r}^{(k)T} \boldsymbol{p}^{(i)} = 0 \quad \text{if } k > i. \tag{4.5}$$

At the same time, $\boldsymbol{r}^{(k)}$ and the intermediate result $\boldsymbol{x}^{(k)}$ are related by

$$\boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} \tag{4.6}$$

Combining equations 4.5 and 4.6 leads to:

$$(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(i)} = 0$$

The left-hand side can be rewritten to:

$$(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(i)} = \boldsymbol{b}^T \boldsymbol{p}^{(i)} - \boldsymbol{x}^{(k)T}(\boldsymbol{A}\boldsymbol{p}^{(i)})$$

Since the result of $\boldsymbol{A}\boldsymbol{p}^{(i)}$ is computed as $\boldsymbol{w}^{(i)}$ in iteration i, it can be replaced as follows

$$(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)})^T \boldsymbol{p}^{(i)} = \boldsymbol{b}^T \boldsymbol{p}^{(i)} - \boldsymbol{x}^{(k)T}\boldsymbol{w}^{(i)}$$

Finally:

$$\boldsymbol{x}^{(k)T}\boldsymbol{w}^{(i)} = \boldsymbol{b}^T \boldsymbol{p}^{(i)}$$

With $\lambda^{(i)} := \boldsymbol{b}^T \boldsymbol{p}^{(i)}$:

$$\boldsymbol{x}^{(k)T}\boldsymbol{w}^{(i)} = \lambda^{(i)} \qquad \text{for } k > i\,.$$

□

Let $\boldsymbol{r}^{(k)} \in \mathbb{R}^n$ be the residual vector in iteration k and let $\boldsymbol{p}^{(i)} \in \mathbb{R}^n$ be the search direction vector in iteration i. The *sigma checksum relation* holds for $\boldsymbol{r}^{(k)}$, and $\boldsymbol{p}^{(i)}$ in an error-free scenario:

$$\boldsymbol{r}^{(k)T}\boldsymbol{p}^{(i)} = 0 \qquad \text{for } k > i. \tag{4.7}$$

Proof: Let i and k denote two different iterations in a Conjugate Gradient solver with $k > i$. Following Equation 2.17, the residual vector $\boldsymbol{r}^{(k)}$ is orthogonal to each preceding search direction vector $\boldsymbol{p}^{(i)}$ in the error-free case with

$$\boldsymbol{r}^{(k)} \perp \boldsymbol{p}^{(i)} \iff \boldsymbol{r}^{(k)T}\boldsymbol{p}^{(i)} = 0 \quad \text{for } k > i.$$

□

In floating-point arithmetic, the normalized form of Equation 4.7 is evaluated and

$$\boldsymbol{r}^{(k)} \perp \boldsymbol{p}^{(i)} \iff \frac{\boldsymbol{r}^{(k)T}\boldsymbol{p}^{(i)}}{\|\boldsymbol{r}^{(k)}\|_2 \|\boldsymbol{p}^{(i)}\|_2} \approx 0 \quad \text{for } k > i. \tag{4.8}$$

Following Equation 4.3, $\sigma^{(i)} := \|\boldsymbol{p}^{(i)}\|_2$

$$\frac{\boldsymbol{r}^{(k)T}\boldsymbol{p}^{(i)}}{\|\boldsymbol{r}^{(k)}\|_2 \cdot \sigma^{(i)}} \approx 0 \qquad \text{for } k > i\,. \tag{4.9}$$

4.2 Error Detection

In each solver iteration, the Conjugate Gradient methods update three vectors, namely the residual vector $\boldsymbol{r}^{(k)}$, the search direction vector $\boldsymbol{p}^{(k)}$ and the intermediate result $\boldsymbol{x}^{(k)}$. Over the course of a solver execution, the residual and search direction vectors affect each other, which allows to detect errors by checking relations between these vectors for successive iterations. As summarized in Chapter 2.1.2, the *orthogonality* properties in Equations 2.16 to 2.18 and the *residual* property in Equation 2.10 must be satisfied to ensure convergence to a correct result. Errors that are caused by transient events, for instance, corrupt these relations and become therefore detectable. Thus, the periodic evaluation of these relations ensures to detect errors that are harmful for the Conjugate Gradient solvers. The orthogonality relations are constituted by the $\boldsymbol{A}$-*orthogonality* between search directions $\{\boldsymbol{p}^{(0)}, \boldsymbol{p}^{(1)}, ..., \boldsymbol{p}^{(N)}\}$ of successive iterations with $\boldsymbol{p}^{(i)} \perp \boldsymbol{A}\boldsymbol{p}^{(k)}$ for $k \neq i$. Checking the $\boldsymbol{A}$-*orthogonality* itself is not feasible, as the periodic evaluation of the required sparse matrix-vector operation induces significant runtime overhead.

To detect errors with low runtime overhead, the Conjugate Gradient solvers are instrumented to evaluate the *lambda* and *sigma error checking criteria* periodically at the error checking interval t. To check these criteria for error detection, inner products with a linear complexity of $\mathcal{O}(n)$ are performed. As the runtime complexity of the Conjugate Gradient solvers depends on the number of non-zero elements *NNZ* in matrix $\boldsymbol{A}$, the runtime overhead for error detection decreases with increasing *NNZ*.

The σ-criterion in Equation 4.7 detects errors that affect at least the residual vectors $\boldsymbol{r}^{(k)}$ by evaluating the orthogonality between successive residual and search direction vectors. The intermediate results $\boldsymbol{x}^{(k)}$ are not related to any other vector by an orthogonality relation. Instead, these vectors are related to the residual vector $\boldsymbol{r}^{(k)}$ by $\boldsymbol{r}^{(k)} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$ following Equation 2.10. The evaluation of this residual relation to detect errors in $\boldsymbol{x}^{(k)}$ induces significant runtime overheads since it relies on an additional matrix-vector multiplication. The λ-checking criterion in Equation 4.4 is derived from combining this residual relation and orthogonality relation between residual and search direction vectors to avoid this matrix-vector multiplication. In accordance with the σ-checking criterion, the λ-checking criterion only relies on inner products. Errors that affect the search directions $\boldsymbol{p}^{(k)}$ become also detectable in this criterion as the intermediate results $\boldsymbol{x}^{(k)}$ are updated using the search directions.

An error detection step is typically not necessary after each iteration as even high error rates do not affect complete sets of iterations in a solver execution, but only small subsets. Additionally, the error checking criteria do not require to evaluate the relations between directly successive iterations (i.e., $k = i + 1$), which allows to reduce the runtime overhead by increasing the *error checking interval t*. While large error checking intervals t allow low runtime overheads by reducing the number of error detection steps, they also can cause a low error coverage. At the same time, small error checking intervals t can enable a high error coverage, but also induce large runtime overheads. To provide a method that determines optimal error checking intervals t that provide minimum solver execution runtime, Z. Chen [Chen13] investigated the dependency between error rates (i.e., the number of expected errors per second), the runtime overhead to obtain correct results, and the selected error checking interval t.

The Conjugate Gradient solvers are typically executed using floating-point arithmetic which is prone to rounding errors. Since rounding errors can cause small differences in the checksums, all checksums are evaluated against a user-defined threshold τ to avoid false positive error detections.

Let $fl(a)$ and $fl(b)$ denote the floating-point representations of checksums $a \in \mathbb{R}$ and $b \in \mathbb{R}$ that are related by $a = b$. The rounding error threshold $\tau \in \mathbb{R}$ is a scalar that is used to evaluate the identity between checksums a and b. The identity $a = b$ is evaluated in floating-point arithmetic as

$$|fl(a) - fl(b)| < \tau \ . \tag{4.10}$$

4.3 Error Correction

If errors are detected during the execution of a Conjugate Gradient solver, error correction steps are required to ensure convergence and eventually a correct result. A direct approach for error correction is repeating the complete solver execution until a correct result is obtained. However, this approach can induce significant runtime overheads as the solvers may require an increased number of iterations to converge in case of errors.

To correct errors, the Conjugate Gradient solvers are instrumented with an *adaptive error correction scheme* that identifies the extent of the detected error and trades off three different correction methods against each other. By selecting a suitable error correction scheme, this method allows low runtime overheads by reducing the number of additional iterations to achieve convergence with a correct result.

In case of a detected error, an *online correction* is performed if the error did not increase the distance between the solution $\boldsymbol{x}$ and the current approximation vector $\boldsymbol{x}^{(k)}$ compared to the last recorded checkpoint. Otherwise, the error recovery scheme performs a *complete roll-back*, if such a correction is not *advantageous* and therefore avoids complete recomputations of the solver. If the utilized checkpoint appears to be corrupted by, for instance, latent errors, a *corrective roll-back* is performed to avoid *endless loops*.

Identifying Degrees of Corruption for Detected Errors

Errors can cause additional solver iterations and silent data corruptions. At the same time, errors also can also take an apparently corrupted intermediate result $\boldsymbol{x}^{(k)}$ closer to the actual solution $\boldsymbol{x}$. If the residual $\boldsymbol{r}^{(k)}$ in the intermediate result $\boldsymbol{x}^{(k)}$ is closer to zero compared to the residual in the intermediate result $\boldsymbol{r}^{(i)}$ of a checkpoint, then $\boldsymbol{x}^{(k)}$ is likely to be closer to the exact solution than $\boldsymbol{x}^{(i)}$ with

$$\begin{aligned} & \|\boldsymbol{r}^{(k)}\|_2 \leq \|\boldsymbol{r}^{(i)}\|_2 \\ \iff & \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}\|_2 \leq \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(i)}\|_2 \end{aligned} \tag{4.11}$$

In such a case, it is *promising* to continue the solver execution using $\boldsymbol{x}^{(k)}$ as it is likely for the solver to require fewer additional iterations to converge compared to a roll-back recovery. Otherwise, if the residual $\boldsymbol{r}^{(k)}$ in the intermediate result $\boldsymbol{x}^{(k)}$ is larger than $\boldsymbol{r}^{(i)}$, then a roll-back to iteration i is more promising to induce fewer iterations. The details of *online correction*, *complete roll-back* and *corrective roll-back* are presented below.

A prerequisite to compute the residuals is the absence of floating-point exceptions such as *NaN* and *Inf* (cf. Appendix C.3) in the underlying vector elements of the intermediate result vector $\boldsymbol{x}^{(k)}$. Such elements are replaced by randomly selected elements in the vector that are free of floating-point exceptions. In case that the complete vector $\boldsymbol{x}^{(k)}$ contains floating-point exceptions, all elements are set to 0.

Online Correction

An erroneous iteration k is *corrected* if the continuation using $\boldsymbol{x}^{(k)}$ is promising to converge in fewer iterations compared to a roll-back to the last checkpoint. *Online correction* re-establishes the *residual* and *orthogonality* relations for successive iterations after iteration k. The following steps are performed for correction: First, the residual $\boldsymbol{r}^{(k)}$ is recomputed in the approximation $\boldsymbol{x}^{(k)}$ as $\boldsymbol{r}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$. Second, the search direction $\boldsymbol{p}^{(k)}$ is computed using the preconditioned residual $\boldsymbol{p}^{(k)} := \boldsymbol{M}^{-1}\boldsymbol{r}^{(k)}$ for PCG and using the residual $\boldsymbol{p}^{(k)} := \boldsymbol{r}^{(k)}$ for CG.

Complete and Corrective Roll-back

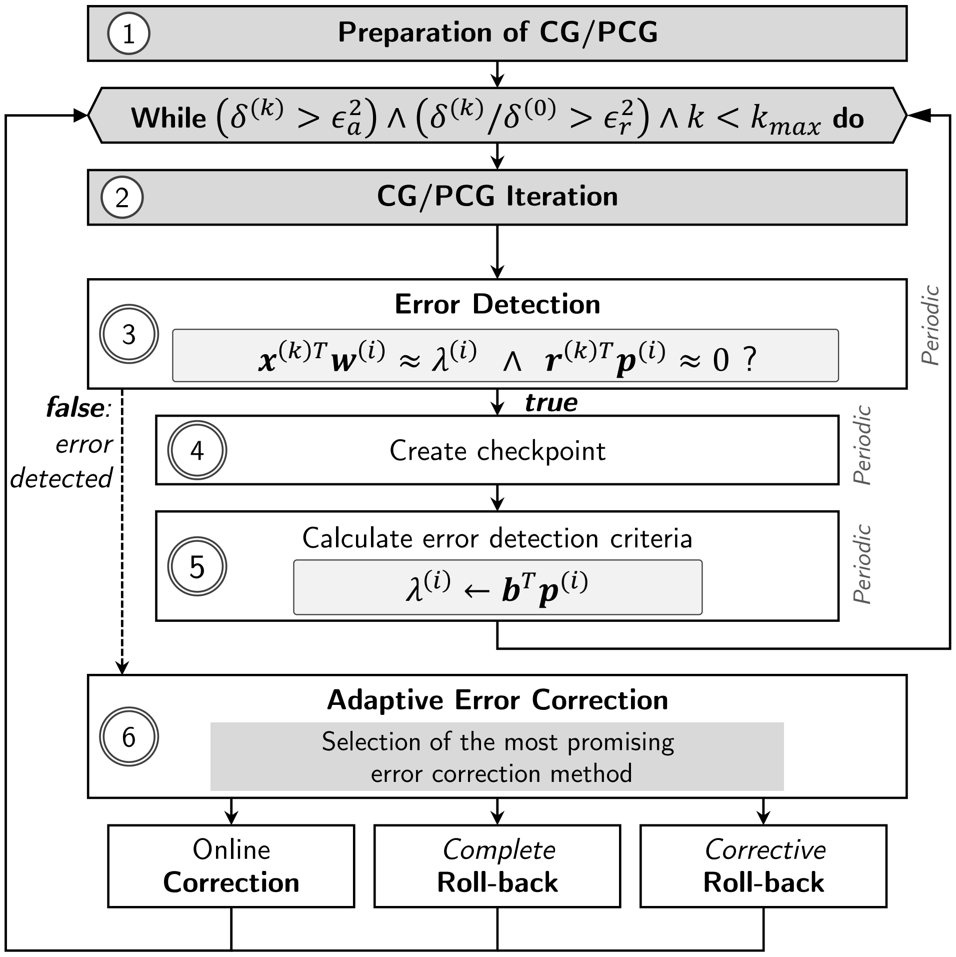
Both roll-back schemes rely on the periodic creation of checkpoints in which the vectors $\boldsymbol{x}^{(k)}$, $\boldsymbol{p}^{(k)}$, and $\boldsymbol{r}^{(k)}$ are written to a *fault-tolerant storage* (e.g., ECC-protected memory [Mitta16b]). During *complete roll-back recovery*, the stored values are copied to the data elements of the current iteration.

Corrective roll-back recovery is performed, if a certain checkpoint is used more than once for error recovery. A checkpoint is periodically created when no error is detected. However, latent errors can propagate to such checkpoints as they may remain undetected until a succeeding error detection step. Rolling back to a checkpoint that contains a latent error can cause endless loops as the solver may be rolled back to the affected checkpoint repeatedly. Therefore, a corrective roll-back is applied when the solver was rolled back to a certain checkpoint before. In this case, only the stored approximation $\boldsymbol{x}^{(i)}$ is restored and the remaining vectors are corrected according to $\boldsymbol{x}^{(i)}$. The residual $\boldsymbol{r}^{(k)}$ is recomputed in the approximation $\boldsymbol{x}^{(k)}$ as $\boldsymbol{r}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$ and the search direction $\boldsymbol{p}^{(k)}$ is set to the preconditioned residual $\boldsymbol{p}^{(k)} := \boldsymbol{M}^{-1}\boldsymbol{r}^{(k)}$ for PCG and the residual $\boldsymbol{p}^{(k)} := \boldsymbol{r}^{(k)}$ for CG, respectively. Afterwards, the solver execution is continued for both roll-back techniques.

4.4 Algorithmic steps

Figure 4.1 shows the algorithmic steps of a *Conjugate Gradient solver* that is instrumented by the presented fault tolerance technique. These steps are suitable for both the Conjugate Gradient Solver and the Preconditioned Conjugate Gradient Solver and are not affected by the underlying preconditioning operations. The first two steps comprise the *preparation of the solver* and the computation of a *solver iteration* which together form the operations in the original solver algorithm (cf. Chapter 2.1.2).

Steps 3 to 6 are added to establish fault tolerance. The presented error detection scheme is performed in the third step. If no error is detected, then a checkpoint is periodically generated in a *fault-tolerant storage* (e.g., ECC-protected memory [Mitta16b]). Both error detection criteria, *lambda* and *sigma* are periodically computed in the fifth step.


1 Preparation of CG/PCG
While $(\delta^{(k)} > \epsilon_a^2) \wedge (\delta^{(k)}/\delta^{(0)} > \epsilon_r^2) \wedge k < k_{max}$ do
2 CG/PCG Iteration
3 Error Detection
$x^{(k)T} w^{(i)} \approx \lambda^{(i)} \wedge r^{(k)T} p^{(i)} \approx 0$?
Periodic
false: error detected
true
4 Create checkpoint
Periodic
5 Calculate error detection criteria
$\lambda^{(i)} \leftarrow b^T p^{(i)}$
Periodic
6 Adaptive Error Correction
Selection of the most promising error correction method
Online Correction
Complete Roll-back
Corrective Roll-back

▲ **Figure 4.1** — Overview of the algorithmic steps in a fault-tolerant (Preconditioned) Conjugate Gradient Solver.

If no errors are detected, then the solver execution is continued. In case of errors, the adaptive error recovery scheme selects the most promising technique in the sixth step, namely *online correction*, *complete roll-back* and *corrective roll-back*.

4.5 Computational and Memory Overhead

The presented fault tolerance technique protects the Conjugate Gradient solvers, *CG* and *PCG*, that have a runtime complexity of $\mathcal{O}(NNZ \cdot \sqrt{\kappa(A)})$ with *NNZ* being the number of non-zero elements and with $\kappa(A)$ being the condition number in matrix A (cf. Chapter 2.1.2). The memory requirement is constant during the solver execution as the vectors and scalars from previous iterations can be overwritten. The memory complexity of the CG solver is $\mathcal{O}(NNZ)$ while it can be larger for the PCG solver as this technique depends on the memory complexity of the utilized preconditioner.

To gain a more realistic insight into the runtime overhead that is induced by the presented technique, it is more feasible to consider the runtime of single iterations as the underlying error detection and correction steps are applied periodically in error checking intervals t. The runtime of a single iteration is typically dominated by a sparse matrix-vector multiplication with runtime complexity $\mathcal{O}(NNZ)$. In the *PCG* solver, the single iterations can be dominated by the preconditioner depending on the underlying preconditioning operations, which can include additional matrix-vector operations or invocations of different solvers [Benzi02].

The error detection scheme of the presented technique is dominated by the periodic computation of four additional inner products, which induce a runtime overhead with linear complexity of $\mathcal{O}(n)$. Assuming sparse matrix-vector multiplications to be the most dominant operations in the solver iterations, the runtime overhead and the scaling behavior depend on the number of non-zero elements *NNZ* in the underlying matrix. Compared to the original solver runtime, the runtime overhead induced by this error detection scheme typically decreases with increasing *NNZ*.

The error detection steps require some memory to store the periodically computed scalars λ and σ that induce a constant memory complexity of $\mathcal{O}(1)$. The corresponding memory requirement remains constant throughout the whole solver execution as the scalars from previous error detection steps can be overwritten. The error correction steps rely on periodically created checkpoints that contain three vectors. For this reason, the memory complexity of the error correction step is linear with $\mathcal{O}(n)$ and is not dependent on the number of solver iterations as previous checkpoints can be overwritten.

When an error is detected, the presented adaptive error correction scheme compares the residual $\boldsymbol{r}^{(k)}$ of the current iteration with the residual in the checkpoint $\boldsymbol{r}^{(i)}$ to select a suitable error correction procedure. The computation of the residual $\boldsymbol{r}^{(k)}$ requires one additional matrix-vector multiplication with runtime complexity $\mathcal{O}(NNZ)$.

CHAPTER 5

Enabling the Conjugate Gradient Solvers on Approximate Computing Hardware

This chapter presents a technique that enables the execution of the Conjugate Gradient solvers using approximate hardware to achieve energy efficiency gains while still ensuring correct solver results [Schol16b]. A major challenge of executing these solvers in the presence of approximation errors is constituted by the tight accuracy demands imposed by the scientific and engineering domains on the result quality. For this reason, different approximation techniques are rendered unsuitable that relax the result accuracy for efficiency gains including task skipping techniques like *loop perforation* (cf. Chapter 2.4). These tight accuracy demands also prohibit *unadaptable* approximation techniques that exploit single degrees of precision (e.g. relying on single approximate hardware designs). Additional solver iterations with increased precision can be required to obtain correct results that can cancel out the gained energy savings. At the same time, different application components typically exhibit different sensitivities to approximation errors, which demand careful adaption of the approximation according to the currently executed component [Chipp13]. Iterative solvers like the Conjugate Gradient solvers additionally exhibit an error resilience that may also change over time [Zhang14a].

Instead of relying on single precision degrees, the presented technique exploits approximate hardware that offers different *approximation levels* (e.g. different degrees of precision with certain numbers of precise bits), which are exchanged according to the changing error resilience. Reduced energy demands and correct results are achieved by dynamically evaluating the underlying error resilience in these solvers. This error resilience evaluation is not only performed *effectively*, but also highly efficiently.

The presented technique is based on the insight that the error resilience can be *estimated* from the *solution progress* with low runtime and energy overhead. Such estimations are translated to approximation levels that are periodically evaluated by the *fault tolerance technique* presented in Chapter 4. This fault tolerance technique is instrumented to detect and correct *harmful* approximation errors, which ensures low iteration overheads to obtain correct results. In a typical solution progress, the *update steps* are being refined over the course of iterations k which therefore become increasingly sensitive to approximation errors. While early iterations often allow a certain degree of approximation as long as the general direction towards the solution is maintained, the induced degree of approximation needs to be reduced as the intermediate results $\boldsymbol{x}^{(k)}$ approach the solution $\boldsymbol{x}$. The degree of error resilience to certain approximation errors can be different for different matrices which demands to *calibrate* this estimation process to changing input matrices. An analysis method is presented that allows this calibration, which is based on recent investigations on the rounding error accumulation behavior in the Conjugate Gradient Solvers [Cools16].

As Chapter 2.5.1 discussed, different related techniques were proposed that focus on specific tasks in the scientific computing domain like Cholesky decompositions [Schaf14], eigendecompositions [Zhang15a], and computing inverse matrix p-th roots [Lass17]. The approach in [Zhang14a] proposes to begin iterative methods using the lowest available degree of precision which is increased if the underlying *optimization function* $E(\boldsymbol{x})$ is violated between iterations. The optimization function for the systems of linear equations $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ is $\min_x E(\boldsymbol{x}) := \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^T\boldsymbol{b}$ with $E'(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}$ [Saad03]. Since the Conjugate Gradient methods do not explicitly compute the optimization function $E(\boldsymbol{x})$ during the solver execution, runtime overheads can be induced by additional evaluations of this functions. Such runtime overheads can be significant, as the optimization function relies on expensive matrix operations.

The presented technique is discussed in this chapter as follows. Section 5.1 presents the formal background for the presented technique. Section 5.2 discusses the details of the error resilience estimation and evaluation steps. The algorithmic steps of the technique are discussed in Section 5.3. Section 5.4 presents and discusses a method that allows to *calibrate* the error resilience estimation process. This chapter concludes with a discussion on the runtime and memory overhead in Section 5.5. The presented technique is evaluated in experimental results in Chapter 7.7.

5.1 Method Overview

As discussed in Chapter 2.1.2, the Conjugate Gradient solvers successively minimize the distance constituted between the initial guess $\boldsymbol{x}^{(0)}$ and the solution of the underlying equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, namely $\boldsymbol{x}$ over time. In the following, it is assumed that $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{x} \in \mathbb{R}^n$, and $\boldsymbol{b} \in \mathbb{R}^n$.

The *update vector* $\boldsymbol{u}^{(k)} \in \mathbb{R}^n$ is the vector between an intermediate result $\boldsymbol{x}^{(k)} \in \mathbb{R}^n$ in iteration k of a Conjugate Gradient solver and its predecessor in iteration $k-1$ with

$$\boldsymbol{u}^{(k)} := \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)} \,. \tag{5.1}$$

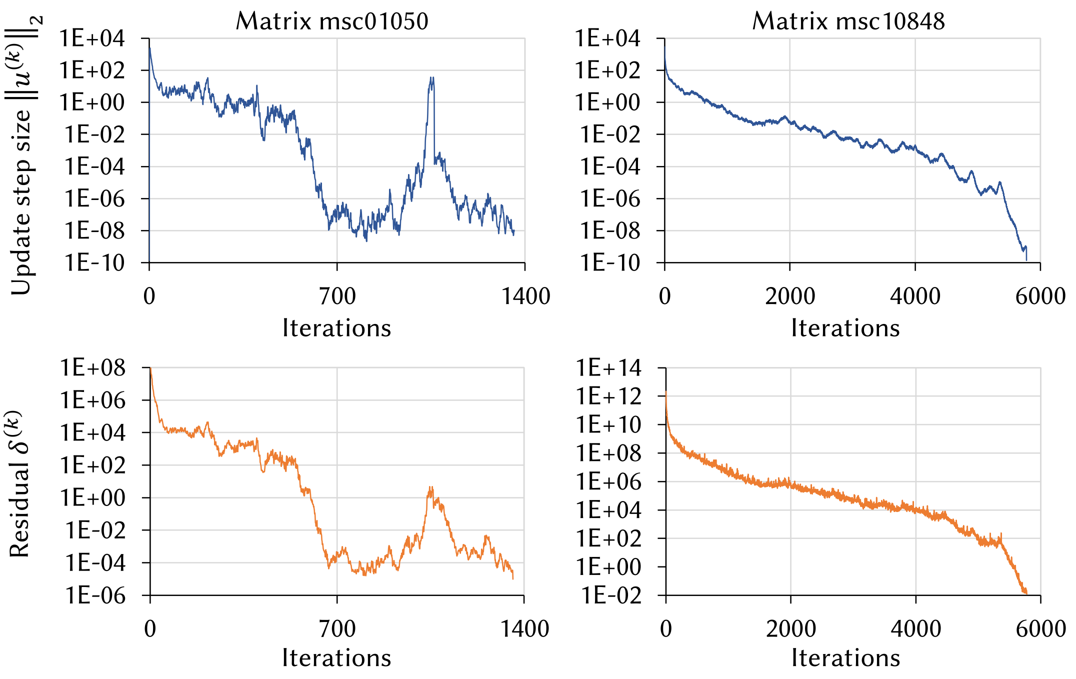
As intermediate results are constructed from preceding intermediate results $\boldsymbol{x}^{(k)}$ with $\boldsymbol{x}^{(k+1)} := \boldsymbol{x}^{(k)} + \alpha^{(k)}\boldsymbol{p}^{(k)}$, the update vectors $\boldsymbol{u}^{(k)}$ can also be computed as

$$\begin{aligned} \boldsymbol{u}^{(k)} &= \boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)} \\ &= \boldsymbol{x}^{(k-1)} + \alpha^{(k-1)}\boldsymbol{p}^{(k-1)} - \boldsymbol{x}^{(k-1)} \\ \boldsymbol{u}^{(k)} &= \alpha^{(k-1)}\boldsymbol{p}^{(k-1)} \end{aligned} \tag{5.2}$$

with $\boldsymbol{p}^{(k-1)} \in \mathbb{R}^n$ being the search direction vector in iteration $k-1$.

The Conjugate Gradient solvers are resilient to a certain degree of approximation as long as the general direction towards the solution is maintained over the course of iterations. As the intermediate results $\boldsymbol{x}^{(k)}$ approach the solution $\boldsymbol{x}$ over time, the induced degree of approximation needs to be reduced as the update vectors are increasingly being *refined* (i.e., $\|\boldsymbol{u}^{(i)}\|_2 < \|\boldsymbol{u}^{(k)}\|_2$ for $i > k$) and therefore become increasingly sensitive to approximation errors. The presented technique relies on an *error resilience estimation scheme* that guides the induced approximation along the solving progress that is reflected in the *update vector length* between successive iterations.

Figure 5.1 shows two examples (cf. Section 7.1) for PCG executions with applied *Jacobi*-Preconditioner. The update vector length $\|\boldsymbol{u}^{(k)}\|_2$ range within several orders of magnitude before converging to a correct result. At the same time, these examples show that such update vectors are typically large during the first solver iterations and approach zero when the solver converges to the solution.



▲ **Figure 5.1** — Comparison of update vectors $\boldsymbol{u}^{(k)}$ and residuals $\delta^{(k)}$ at runtime for two input matrices $\boldsymbol{A}$.

As summarized in Chapter 2.1.2, the Conjugate Gradient solvers require the search directions of successive iterations to be $\boldsymbol{A}$-*orthogonal* to ensure convergence to a correct result. Following Equation 5.2, the update vectors $\boldsymbol{u}^{(k)}$ are computed by scaling the corresponding search directions $\boldsymbol{p}^{(k-1)}$ using the scalar factor $\alpha^{(k-1)}$. For this reason, the $\boldsymbol{A}$-orthogonality also applies to the update vectors $\boldsymbol{u}^{(k)}$ with

$$\boldsymbol{u}^{(i)}\boldsymbol{A}\boldsymbol{u}^{(k)} = 0 \quad \text{for } k \neq i \, . \tag{5.3}$$

The Conjugate Gradient solvers are becoming increasingly sensitive for smaller update vectors since the update direction is now increasingly altered by approximation errors which can *violate* the $\boldsymbol{A}$-*orthogonality* between successive update vectors. Checking the $\boldsymbol{A}$-*orthogonality* periodically is not a feasible solution, as the evaluation of the required matrix-vector multiplication induces significant runtime and energy overhead that may cancel the achieved efficiency gains. Instead, the impact of approximation errors on the update vector *directions* $\boldsymbol{u}^{(k)}$ can be estimated to guide the approximation with low overhead.

When an approximation technique is applied to a solver execution in floating-point arithmetic, then both rounding and approximation errors *accumulate* in the underlying solver data over the course of solver iterations. Therefore, the error resilience of the Conjugate Gradient solvers in iteration k is determined by the maximum approximation error *accumulation* which is not violating the inherent solver convergence invariants. The presented technique exploits the insight that the accumulation of approximation errors can be estimated from the *residual* $\delta^{(k)}$ (i.e., $\delta^{(k)} := \|\boldsymbol{r}^{(k)}\|_2 = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}\|_2$, cf. Equation 2.10), which allows low overhead in evaluating the error resilience. For floating-point arithmetic with machine epsilon ε_M, Cools et al. [Cools16] investigated the *rounding error* accumulation process and presented a mathematical method to determine *error accumulation limits* by tracking the residual $\delta^{(k)}$ over time. Based on this investigation, the presented technique estimates the error resilience and adapts an underlying precision-configurable approximation technique accordingly. Compared to evaluating the update vector length $\|\boldsymbol{u}^{(k)}\|_2$, which requires additional inner products, monitoring the residual $\delta^{(k)}$ induces favorably low overhead as this value is inherently computed by the Conjugate Gradient solvers in each iteration.

Precision-configurable approximation techniques typically provide a finite number of configurations, which are called *approximation levels* below. The set of approximation levels is denoted by L and

$$L := \{l_i \mid i \in \{1, ..., n\}\} \tag{5.4}$$

with l_i denoting an approximation level that corresponds to a specific configuration of the approximation technique.

Using the number of approximation levels n, the presented technique decomposes the value range for the residual $\delta^{(k)}$ into n intervals

$$[0, \rho_1), \quad [\rho_1, \rho_2), \quad [\rho_2, \rho_3), \quad \cdots \quad [\rho_n, \infty) \,. \tag{5.5}$$

As a result of this decomposition, each residual $\delta^{(k)}$ is an element of one interval $[\rho_i, \rho_{i+1})$. To reflect the increasing demand for precision by the Conjugate Gradient solvers as they approach the solution, each interval comprises a specific range of residual values for which a certain approximation level is utilized. At runtime, an *approximation level step function* $H(\delta^{(k)})$ is evaluated to determine the approximation level $l_i \in L$ for the residual $\delta^{(k)}$ in iteration k.

$$H(\delta^{(k)}) : [0, \infty) \quad \rightarrow \quad L \,. \tag{5.6}$$

For a given residual $\delta^{(k)}$, the function $H(\delta^{(k)})$ provides the corresponding approximation level l_i with respect to the underlying interval $\delta^{(k)} \in [\rho_i, \rho_{i+1})$. To satisfy the increasing demand for precision as the solver approaches the solution, this function maps large residuals (e.g. close to the residual in the first iteration $\delta^{(0)}$) to approximation levels l_i with low precision while smaller residuals are mapped to approximation levels with increasing precision. While this function can be defined as a user-defined lookup table, Section 5.4 below presents a technique to determine this function for specific input matrices $\boldsymbol{A}$ based on analyzing the approximation error accumulation. The monitoring of $\delta^{(k)}$ and the computation $H(\delta^{(k)})$ induces very low performance and energy overheads since only scalar operations are required.

5.2 Evaluation of the Estimation

Estimation errors can lead to selecting approximation levels l_i that are too *aggressive* and violate the inherent convergence relations. Without reconfiguring the underlying approximation technique to provide increased precision, wrong results can be obtained despite apparent convergence of the solver while the compute efficiency gain can be canceled by additional iterations to obtain correct results. This scenario can occur when the step function $H(\delta^{(k)})$ does not *exactly* represent the underlying error resilience, which can be caused by selecting a different initial guess vector $\boldsymbol{x}^{(0)}$ or by applying the step function $H(\delta^{(k)})$ to a different linear system $\boldsymbol{A}'\boldsymbol{x} = \boldsymbol{b}'$.

The *approximation level that is applied in solver iteration i* is computed by adding an offset $o \in \mathbb{N}_0$ to the result of $H(\delta^{(k)})$. The *offset function* $O : L \times \mathbb{N}_0 \to L$ maps the result of the step function $H(\delta^{(k)})$ to an approximation level with increased precision. For an offset $o^{(k)}$ in iteration k, the offset function computes the *approximation level in solver iteration k* (i.e., $l^{(k)} \in L$) as

$$l^{(k)} := O\left(H(\delta^{(k)}), o^{(k)}\right) = l_{i-o} \tag{5.7}$$

such that the approximation level l_{i-o} provides increased precision compared to approximation level l_i.

The offset $o^{(k)}$ is adapted by the fault tolerance technique presented in Chapter 4. The introduced approximation errors are periodically evaluated by the fault tolerance technique to detect and correct approximation errors that are too harmful for the solver execution. This fault tolerance technique ensures correct solver results by periodically evaluating the inherent solver convergence relations. At the same time, this technique induces only low energy and runtime overheads since it avoids expensive operations.

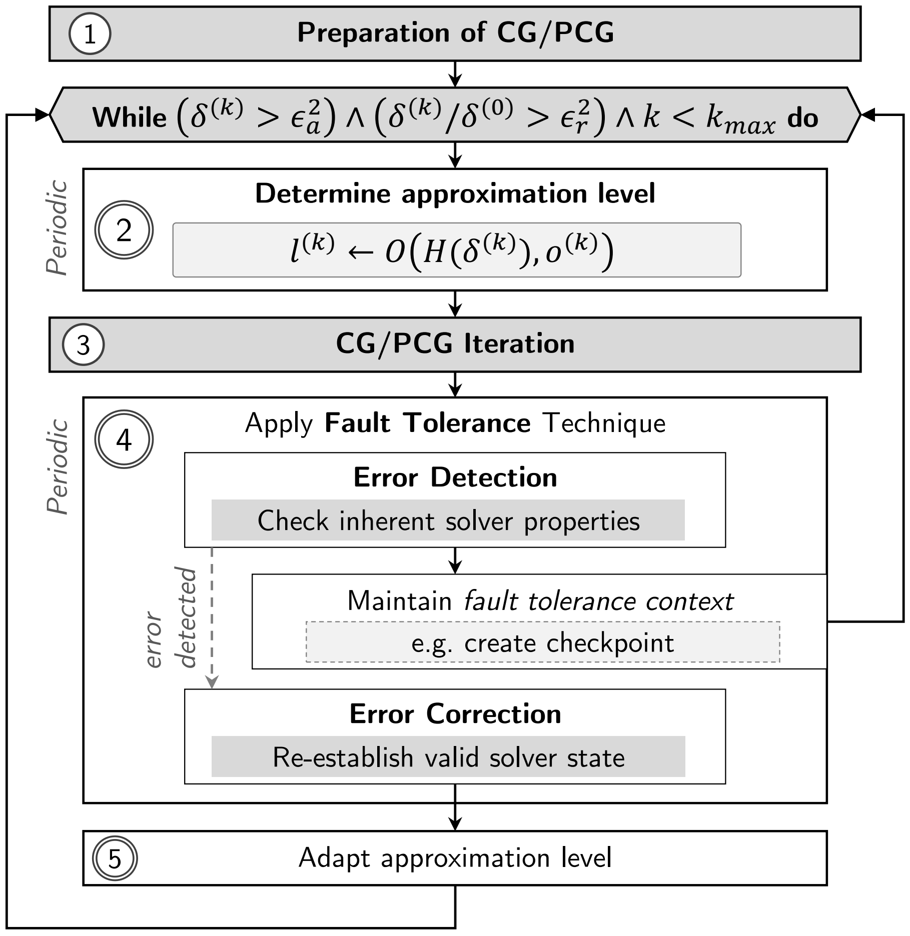
In case of a violation, this fault tolerance technique reestablishes the inherent solver relations by performing error correction steps as explained in Chapter 4.3. Afterwards, the offset is incremented to continue the solver execution with increased precision.

If the result of the step function $H(\delta^{(k)})$ suggests to increase the underlying precision over the course of the solver iterations, then this offset is decremented again. As a result, the utilized approximation level and underlying precision remain unchanged in scenarios in which the offset is larger than zero. This offset reduction avoids unnecessary compute efficiency reductions by maintaining the maximum degree of approximation which does not harm the convergence to correct results. At the same time, unsuitable offset reductions are detected and corrected by the periodically evaluated fault tolerance technique.

5.3 Algorithmic Steps

Figure 5.2 shows the presented technique for the Conjugate Gradient solvers. This technique enables the Conjugate Gradient solvers for approximate computing by instrumenting the underlying algorithm by additional steps that periodically *estimate the error resilience*, select a *corresponding approximation level* and *evaluate such selected levels*.

The first step comprises the *preparation* of the solver algorithm which is constituted by the original solver preparation steps (cf. Chapter 2.1.2). In the second step, the approximation level $l^{(k)}$ is periodically determined based on estimating the current error resilience in iteration k. Using this approximation level $l^{(k)}$, a *solver iteration* is computed in the third step. Afterwards, the approximation level is evaluated periodically in the fourth step by the fault tolerance technique presented before in Chapter 4. If the approximation level is too aggressive (i.e., the underlying precision is unsuitable for solver convergence), the inherent convergence invariants of the Conjugate Gradient solvers are violated, which is detected by the fault tolerance technique as an *error*. In such a case, an offset is introduced in the fifth step which ensures that the solver iterations are continued using lower approximation levels with increased precision. Besides, a valid solver state is recovered by performing the *error correction* scheme of the fault tolerance technique.


1
Preparation of CG/PCG
While $(\delta^{(k)} > \epsilon_a^2) \wedge (\delta^{(k)}/\delta^{(0)} > \epsilon_r^2) \wedge k < k_{max}$ do
Periodic
2
Determine approximation level
$l^{(k)} \leftarrow O(H(\delta^{(k)}), o^{(k)})$
3
CG/PCG Iteration
Periodic
4
Apply Fault Tolerance Technique
Error Detection
Check inherent solver properties
error detected
Maintain fault tolerance context
e.g. create checkpoint
Error Correction
Re-establish valid solver state
5
Adapt approximation level

▲ **Figure 5.2** — Overview of the presented technique for the Conjugate Gradient Solvers.

5.4 Calibrating the Approximation Estimation Process

This section presents a method to determine approximation level step functions $H(\delta^{(k)})$ that ensure energy-efficient executions of the Conjugate Gradient solvers using approximate computing hardware. As the step functions $H(\delta^{(k)})$ are based on estimating the error resilience, a major challenge in determining *suitable* step functions $H(\delta^{(k)})$ is to guide the approximation with respect to the actual error resilience in the solution progress. For instance, if the precision is increased too early, the overall efficiency gain is unnecessarily reduced. If the precision is increased too late, unacceptable errors can accumulate and cause additional iterations that reduce or cancel out the achieved efficiency gains. At the same time, the dynamic behavior of the error resilience can be different for different input matrices.

Based on the investigation by Cools et al. [Cools16] the method presented in this section determines the step functions $H(\delta^{(k)})$ with respect to approximation techniques that offer different *precisions* ε_i. For each approximation level l_i that a configurable approximation technique provides, the corresponding precision ε_i is determined by the *maximum relative error* for non-zero results.

Given a basic arithmetic operation $\mathbf{op}$ (e.g., $\mathbf{op} \in \{+,-,\cdot,/\}$) that is performed on two operands $a \in \mathbb{R}$ and $b \in \mathbb{R}$. Let the corresponding approximate operation using approximation level l_i be denoted by $\mathbf{op}_{\text{apx}}$. For approximation level l_i, the underlying precision ε_i is

$$\varepsilon_i := \max|\delta| \quad \text{subject to: } (a\ \mathbf{op}_{\text{apx}}\ b) = (a\ \mathbf{op}\ b)(1+\delta) \tag{5.8}$$

and

$$(a\ \mathbf{op}_{\text{apx}}\ b) \neq 0 \wedge (a\ \mathbf{op}\ b) \neq 0\ .$$

Over the course of solver iterations k, the residual $\delta^{(k)}$ typically decreases. The underlying idea is to decompose the residual value range $[0,\infty($ into intervals $[\rho_i,\rho_{i+1})$ according to Equation 5.5 such that ρ_i denotes the *minimum residual* to which the approximation error accumulation does not violate the inherent solver properties. As a result, the intervals $[\rho_i,\rho_{i+1})$ determine the residual value range in which the solver is resilient to precision ε_i at runtime.

To determine the minimum residual ρ_i for a precision ε_i of interest, the method executes the Conjugate Gradient solver using this precision ε_i while monitoring the error accumulation caused by both rounding errors and approximation errors. The corresponding approximation level l_i remains *suitable* until the accumulated error violates an *accumulation limit*, which is explained below in detail. In such a case, this approximation level needs to be exchanged by an approximation level with increased precision. Let a violation occur in iteration k with residual $\delta^{(k)}$ for applied precision ε_i. Since a violation occurred, the residual $\delta^{(k)}$ determines the minimum residual (i.e., $\rho_i := \delta^{(k)}$) to which the solver execution is resilient to using precision ε_i. At the same time, this event is tracked in the step function $H(\delta^{(k)})$ by adding a step with

$$H(\delta^{(k)}) := l_i. \tag{5.9}$$

Monitoring and Evaluation of Error Accumulations

The accumulation of arithmetic errors such as rounding and approximation errors can cause the residual vector $\boldsymbol{r}^{(k)}$ to increasingly deviate from the *true residual* vector $\boldsymbol{r}_{\text{true}}^{(k)}$ over the iterations. This deviation is constituted by the property that the true residual vector

$$\boldsymbol{r}_{\text{true}}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)} \tag{5.10}$$

is never calculated over the course of iterations in the Conjugate Gradient solvers. Instead, the residual vector $\boldsymbol{r}^{(k)}$ is calculated by the Conjugate Gradient solvers as:

$$\boldsymbol{r}^{(k)} := \boldsymbol{r}^{(0)} + \sum_{k=0}^{N} -\alpha^{(k)}\boldsymbol{A}\boldsymbol{p}^{(k)} \tag{5.11}$$

with $\alpha^{(k)}$ being computed following Equation 2.13.

Following [Cools16], the true residual $\delta_{\text{true}}^{(k)} := \|\boldsymbol{r}_{\text{true}}^{(k)}\|_2$ begins to *stagnate* (i.e., the value in $\delta_{\text{true}}^{(k)}$ does not decrease further) while the residual $\delta^{(k)} := \|\boldsymbol{r}^{(k)}\|_2$ keeps decreasing after so-called *residual stagnation points* due to rounding error accumulation. Such residual stagnation points are determined by evaluating the difference vector $\Delta\boldsymbol{r}^{(k)}$ between the iterative residual vector $\boldsymbol{r}^{(k+1)}$ and the true residual vector $\boldsymbol{r}_{\text{true}}^{(k+1)}$ with

$$\Delta\boldsymbol{r}^{(k)} := \boldsymbol{r}^{(k)} - \boldsymbol{r}_{\text{true}}^{(k)} \ . \tag{5.12}$$

The *residual difference* is the norm of the difference vector $\Delta\boldsymbol{r}^{(k)}$ with

$$\Delta\delta^{(k)} := \|\Delta\boldsymbol{r}^{(k)}\|_2 \ . \tag{5.13}$$

A *residual stagnation point* is reached in iteration k, if

$$\delta^{(k)} \leq \Delta\delta^{(k)} \ . \tag{5.14}$$

When the rounding error accumulation until iteration k causes the residual difference $\Delta\delta^{(k)}$ to exceed the residual $\delta^{(k)}$, then additional solver iterations after iteration k are not likely to improve the intermediate result towards the solution as long as the underlying precision is not increased. For this reason, residual stagnation points determine error accumulation limits. This insight is exploited by the method presented in this section to determine minimum residuals ρ_i for each precision ε_i that is offered by the underlying approximate computing technique with

$$\rho_i := \min_k \delta^{(k)} \quad \text{with } \delta^{(k)} \leq \Delta\delta^{(k)} \, . \tag{5.15}$$

An *estimation for the residual difference* $\Delta\delta^{(k)}_{\text{est}}$ is presented in [Cools16] to determine the residual difference $\Delta\delta^{(k)}$ with minimum runtime overhead. Computing the residual difference $\Delta\delta^{(k)}$ exactly would require an additional expensive matrix-vector multiplication in each iteration which can cancel achieved compute efficiency gains.

In the first solver iteration, the iterative residual vector $\boldsymbol{r}^{(0)}$ is equal to the true residual vector $\boldsymbol{r}^{(0)}_{\text{true}}$ (cf. Line 1 in Algorithm 1). For this reason, the difference between $\boldsymbol{r}^{(0)}$ and $\boldsymbol{r}^{(0)}_{\text{true}}$ is zero

$$\boldsymbol{r}^{(0)} = \boldsymbol{r}^{(0)}_{\text{true}} \iff \Delta\delta^{(0)} = 0 \, . \tag{5.16}$$

In the remaining solver iterations, the value of the residual difference $\Delta\delta^{(k)}$ is estimated by the term $\Delta\delta^{(k)}_{\text{est}}$ [Cools16] with

$$\Delta\delta^{(k+1)}_{\text{est}} := \Delta\delta^{(k)}_{\text{est}} + 2\alpha^{(k)} \|\boldsymbol{w}^{(k)}\|_2 \cdot \varepsilon_M \tag{5.17}$$

with ε_M being the machine epsilon and $\Delta\delta^{(0)}_{\text{est}} := 0$. Here, the terms $\alpha^{(k)}$ and $\boldsymbol{w}^{(k)}$ denote internal variables of the Conjugate Gradient solvers (cf. Chapter 2.1.2).

Equation 5.17 is reformulated to estimate the difference between true and iterative residual vectors $\Delta\delta^{(k+1)}_{\text{est}}$ caused by *approximation errors* as follows: While the machine epsilon ε_M describes the maximum relative error caused by rounding (cf. Chapter 3.2), the term ε_i describes the maximum relative error caused by a specific configuration of an approximation technique following Equation 5.8, which allows to replace ε_M by ε_i. The method presented in this section estimates the difference between true and iterative residual vectors caused by *approximation errors* using the term $\tau^{(k+1)}$ which is computed as

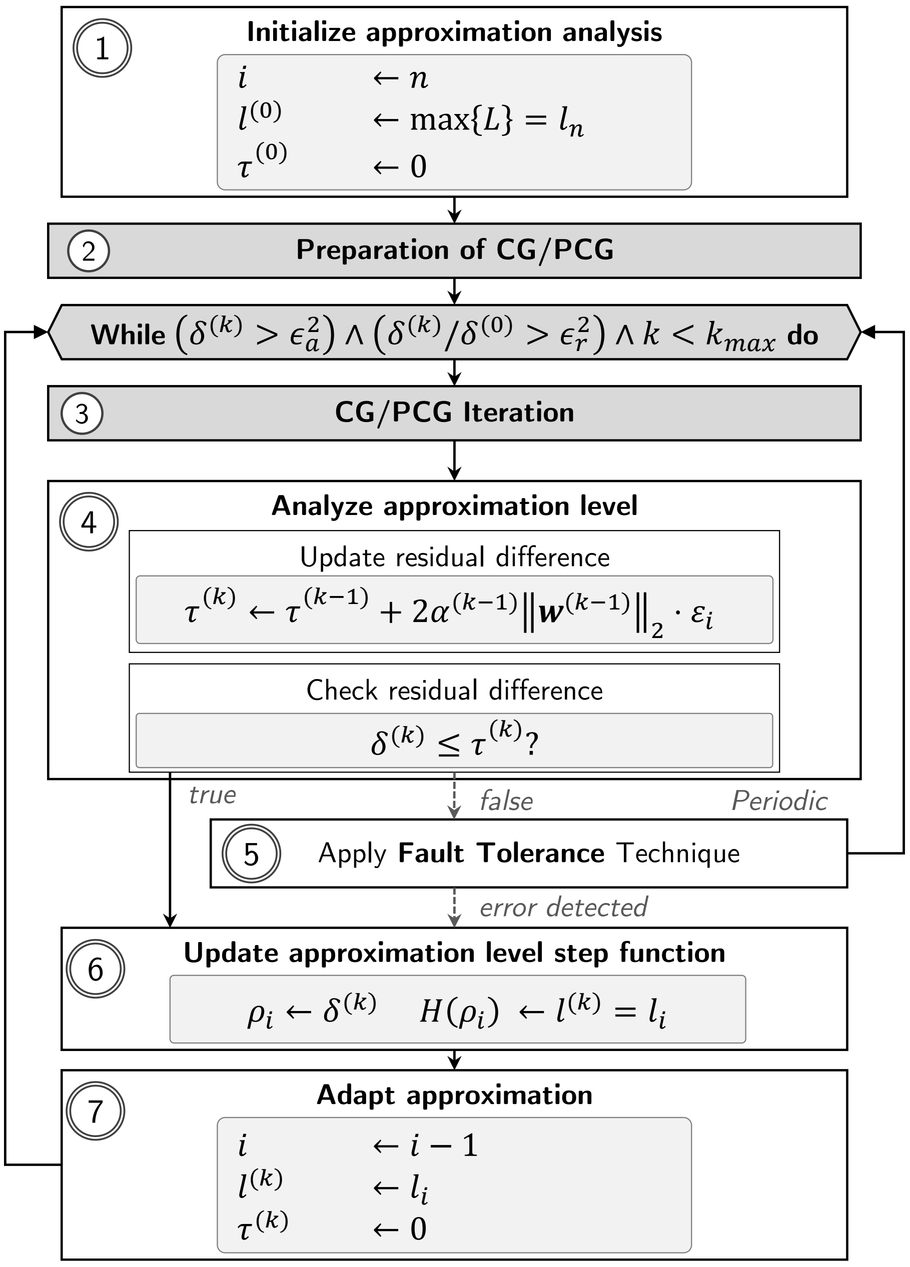
$$\tau^{(k+1)} := \tau^{(k)} + 2\alpha^{(k)} \|\boldsymbol{w}^{(k)}\|_2 \cdot \varepsilon_i \tag{5.18}$$

and $\tau^{(0)} := 0$.

This estimation procedure requires one additional product per iteration to compute the norm $\|\boldsymbol{w}^{(k)}\|_2$, which only has to be applied once per matrix $\boldsymbol{A}$. The minimum residuals ρ_i collected for the different precisions ε_i are applied to different solver executions that are based on the same or highly similar matrices $\boldsymbol{A}$ to avoid this inner product and to reduce the energy demand.

Overview of algorithmic steps

Figure 5.3 shows the algorithmic steps of the presented method to determine step functions $H(\delta^{(k)})$ that guide the induced approximation along the solution progress.


1
Initialize approximation analysis
$i \leftarrow n$
$l^{(0)} \leftarrow \max\{L\} = l_n$
$\tau^{(0)} \leftarrow 0$
2
Preparation of CG/PCG
While $\left(\delta^{(k)} > \epsilon_a^2\right) \wedge \left(\delta^{(k)}/\delta^{(0)} > \epsilon_r^2\right) \wedge k < k_{max}$ do
3
CG/PCG Iteration
4
Analyze approximation level
Update residual difference
$\tau^{(k)} \leftarrow \tau^{(k-1)} + 2\alpha^{(k-1)}\left\|\boldsymbol{w}^{(k-1)}\right\|_2 \cdot \varepsilon_i$
Check residual difference
$\delta^{(k)} \leq \tau^{(k)}?$
true
false
Periodic
5
Apply Fault Tolerance Technique
error detected
6
Update approximation level step function
$\rho_i \leftarrow \delta^{(k)} \quad H(\rho_i) \leftarrow l^{(k)} = l_i$
7
Adapt approximation
$i \leftarrow i - 1$
$l^{(k)} \leftarrow l_i$
$\tau^{(k)} \leftarrow 0$

▲ **Figure 5.3** — Overview of the algorithm that determines the minimum residuals ρ_i for each precision ε_i.

This methods instruments the Conjugate Gradient solvers to find the different minimum residuals ρ_i to which the solvers are resilient at runtime with respect to the different available degrees of precision $\varepsilon_i \in \{\varepsilon_1, \varepsilon_2, ..., \varepsilon_N\}$. The first step initializes this procedure by selecting the approximation level with minimum available precision. The residual difference variable $\tau^{(0)}$ is set to zero since the iterative residual vector $\boldsymbol{r}^{(0)}$ is initialized to $\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$ before the first iteration by the Conjugate Gradient solvers, which equals the true residual vector $\boldsymbol{r}^{(0)}_{\text{true}}$. Steps two and three comprise the *preparation of the solver* and the computation of a *solver iteration* which together form the operations in the original solver algorithm (cf. Chapter 2.1.2).

Step four updates the estimation for the residual difference $\tau^{(k)}$ after each solver iteration as described in Equation 5.18. If the minimum residual ρ_i for the currently chosen precision ε_i has not been found yet (i.e., *residual stagnation point*, cf. Equation 5.14), the fault tolerance technique presented in Chapter 4 is periodically applied in the fifth step, which checks the solver convergence invariants to ensure correct results.

If either the minimum residual ρ_i for the currently chosen precision ε_i has been found or the fault tolerance technique detected a violation, the step function $H(\delta^{(k)})$ is updated in the sixth step using the residual $\delta^{(k)}$ in the current iteration k. After this update step, the step function $H(\delta^{(k)})$ will suggest to use the approximation level l_i with the corresponding precision ε_i as long as the iterative solver residual exceeds the minimum residual ρ_i. Afterwards, the induced approximation is adapted to use increased precision in step seven. The approximation level l_i is exchanged by an approximation level that offers the next increased precision degree. The iterative residual vector $\boldsymbol{r}^{(k)}$ is restored by computing the true residual vector (i.e., $\boldsymbol{r}^{(k)} := \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(k)}$). As restoring the iterative residual vector cancels the difference between the iterative and true residual vector, the difference variable $\tau^{(k)}$ is set to zero.

5.5 Computational and Memory Overhead

The presented methods in this chapter target the Conjugate Gradient solvers that are additionally instrumented by the fault tolerance technique presented in Chapter 4. The computational and memory complexity of these solvers and the overhead introduced by this fault tolerance technique are not changed by the additional operations introduced by the method in this chapter. For this reason, the computational and memory complexity presented in Chapter 4.5 is considered as a basis to discuss the runtime and energy overhead.

The memory overhead for both methods is constituted by storing the step function $H(\delta^{(k)})$ and the offset value. Storing this function requires memory for n scalar tuples (i.e., $(\delta^{(k)}, l_i)$), with n being the number of available approximation levels that the underlying approximation technique offers. The runtime and energy overhead is constituted by estimating and evaluating the underlying error resilience as well as changing the approximation level. The error resilience estimation requires one periodic evaluation of the step function $H(\delta^{(k)})$. With this function being implemented as a lookup table, the runtime and energy overhead induce linear complexity $\mathcal{O}(n)$. The evaluation of the selected approximation level l_i by the fault tolerance technique induces a runtime overhead with linear complexity of $\mathcal{O}(n)$, with n being the matrix size (cf. Chapter 4.5). Adapting the underlying approximation technique (e.g. precision-configurable approximate hardware) induces a constant runtime overhead with $\mathcal{O}(1)$. This update procedure requires, for instance, a single special instruction from an *approximate instruction set architecture* as discussed in [Samps11, Venka13a].

The method that determines step functions $H(\delta^{(k)})$ to guide the approximation at runtime presented above in Section 5.4 differs from the pure approximation guiding method by storing and updating the residual difference $\tau^{(k)}$ and by updating the step function $H(\delta^{(k)})$. The residual difference $\tau^{(k)}$ induces a constant memory complexity with $\mathcal{O}(1)$ as only a single scalar needs to be stored in memory. The runtime and energy overhead is constituted by estimating the residual difference as described in Equation 5.18. This estimation induces an overhead with linear complexity in each iteration with $\mathcal{O}(n)$ as an additional inner product is required. This inner product can be avoided if the gained step functions $H(\delta^{(k)})$ are reused for further solver executions using a certain matrix $\boldsymbol{A}$. The update of the step function $H(\delta^{(k)})$ induces a runtime and energy overhead of linear complexity with $\mathcal{O}(n)$, if this function is being implemented as a lookup table.

Chapter 6

Parameter Estimation for Application Executions on Approximate Computing Hardware

This chapter presents parameter evaluation and estimation methods to assess application executions on approximate computing hardware. The parameters of interest comprise the *area*, the *leakage power*, the *dynamic power*, the *delay*, and the *approximation error*. The investigation of these parameters is required to reveal crucial *compute efficiency* aspects gained by approximate hardware, which includes the *energy per instruction* and the overall *runtime performance*. The parameters of interest can be distinguished according to their dependency on the application execution. *Application-independent* parameters comprise the area and the leakage power, which are obtained during the hardware synthesis process, for instance. *Application-dependent* parameters comprise the dynamic power, the delay, and the approximation error, which depend on the underlying application data (e.g., operand values for arithmetic circuits).

Application-dependent parameters induce a major challenge to achieve low parameter evaluation runtimes for complex and long-running applications, which can comprise billions of executed instructions. Parameter evaluation techniques that rely on circuit simulation to evaluate application-dependent parameters are often rendered infeasible by the execution times that are induced by simulating all executed instructions in an application. Such simulations are often orders of magnitude slower than executing the underlying instructions in silicon [Oluko98].

To reduce the runtime to assess complete application executions, a *simulation-based parameter evaluation method* is combined with a *model-based* method to *estimate* application-dependent parameters for complete application executions. The underlying idea is to combine highly accurate but slow gate-level timing simulations with light-weight software-based models, which describe the numerical approximation error of approximate computing hardware. This *combined parameter estimation method* is applied to *iterative algorithmic parts* such as loops to estimate the application-dependent parameters using a reduced number of simulation results. This approach is based on the insight that such parameters can be estimated with high accuracy by evaluating carefully selected *instruction intervals* (i.e., a section of continuous application execution). Approximation error adaptions in iterative algorithmic parts are exploited to select instruction intervals for simulation-based parameter estimation. At the same time, this method considers the dependency between approximation errors, the power dissipation, and the delay by evaluating the propagation of approximation errors throughout the application execution.

As Chapter 2.5.2 discussed, different related works *model the approximation error* induced by approximate computing hardware to evaluate the error resilience of applications. For instance, related works [Chipp13, Mishr14, Barba16] truncate the operands in arithmetic operations to identify algorithmic parts as well as data that are resilient to approximate computing techniques. Such approaches do not consider parameters like the power dissipation, which are required to evaluate compute efficiency aspects. Different related works analyze the power dissipation for complete application executions. Related works [Tiwar94, Rethi14, Laure04] rely on *physical experiments* to model the power dissipation of different instructions in applications. While these experiments are performed at full execution speed, they require the implementation of approximate hardware designs in physical hardware.

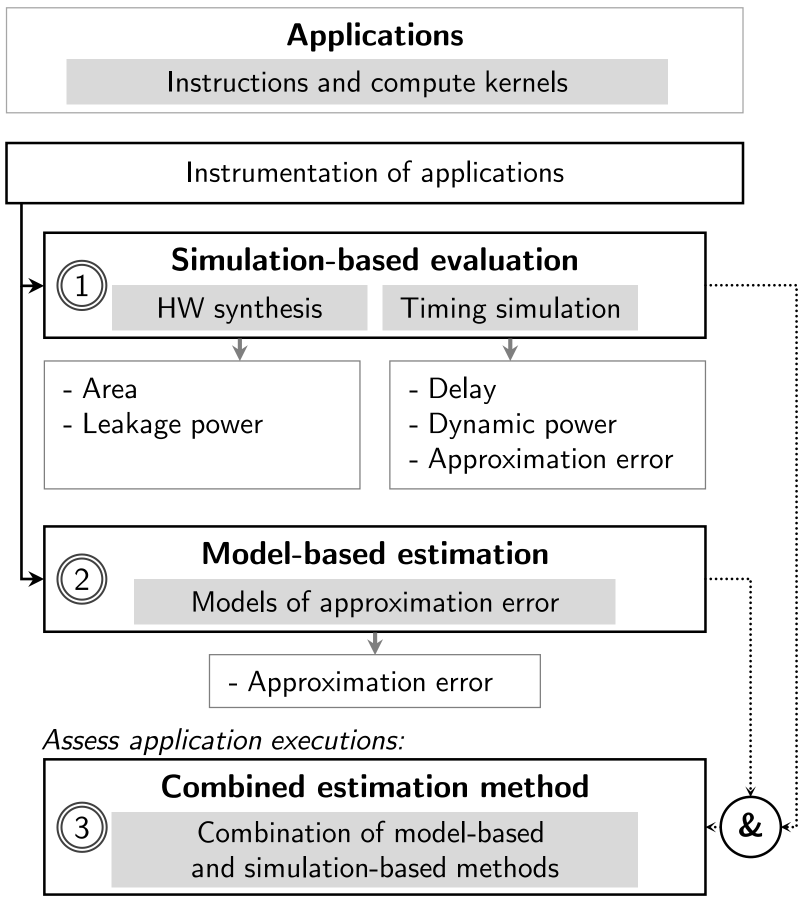
Related works [Hsieh98, Wunde03, Hamer05] present different techniques to reduce the runtime of simulation-based power analysis for complete application executions. Instead of evaluating all instruction executions in an application, these techniques evaluate significantly reduced portions of the original application and estimate the application power dissipation from the obtained results. [Hsieh98] synthesizes new programs based on the *instruction mix* (cf. Chapter 2.4.2) in applications. [Wunde03, Hamer05] evaluate representative instruction intervals using circuit simulation. These related approaches assume that the evaluated instructions represent the complete execution of the original application. However, approximation errors propagate between executed instructions. For this reason, the evaluation of limited instruction intervals does not necessarily reflect the impact of propagated approximation errors on both the power dissipation and the delay. New techniques are required that consider *approximation errors* and their *propagation* throughout the application execution. At the same time, precision-configurable approximate hardware needs to be considered that allows to adapt the degree of the induced approximation error. Such hardware structures are required by approximation techniques like [Zhang14a] and by the technique presented in Chapter 5 to ensure acceptable application outputs.

The remainder of this chapter presents the different methods as follows. Section 6.1 discusses the dependencies between the different *parameter evaluation and estimation methods*. To instrument applications for parameter evaluation, an interface is presented in Section 6.2. The simulation-based parameter evaluation method is presented in Section 6.3. Section 6.4 presents the estimation method that relies on software-based models of approximation techniques. Section 6.5 presents the combined parameter estimation method, which reduces the parameter estimation runtime for iterative algorithms.

The evaluation of the parameter estimation methods presented in this Chapter is discussed in Chapter 7.6. In this evaluation, the presented approach is applied to investigate the approximate computing technique presented in Chapter 5.

6.1 Overview of Parameter Evaluation and Estimation Methods

The presented approach relies on three different parameter evaluation and estimation methods, which are based on gate-level timing simulations, software-based models of approximation techniques as well as their combination. An overview of these methods and their dependencies is shown below in Figure 6.1. Applications comprise *instructions* and *compute kernels* that can form iterative algorithmic parts. Applications of interest need to be *instrumented* to evaluate both application-dependent and application-independent parameters with respect to the underlying approximate computing hardware. A flexible *data representation* is provided that allows the seamless integration of the different methods into applications to evaluate the underlying instructions and data.


Applications
Instructions and compute kernels
Instrumentation of applications
1
Simulation-based evaluation
HW synthesis
Timing simulation
- Area
- Leakage power
- Delay
- Dynamic power
- Approximation error
2
Model-based estimation
Models of approximation error
- Approximation error
Assess application executions:
3
Combined estimation method
Combination of model-based and simulation-based methods
&

▲ **Figure 6.1** — Overview of the parameter estimation flow using the three estimation methods.

Gate-level timing simulations and *hardware synthesis results* form the simulation-based method, which provides detailed insights into the application execution on approximate computing hardware. The second method is the model-based estimation method, which relies on light-weight *software-based models* to mimic the numerical error of approximation techniques.

The *combined estimation method* evaluates the different parameters for iterative algorithmic parts. To provide low runtime, this method evaluates a reduced portion of iterations using timing simulation while it evaluates the remaining iterations using software-based models.

6.2 Instrumentation of Applications

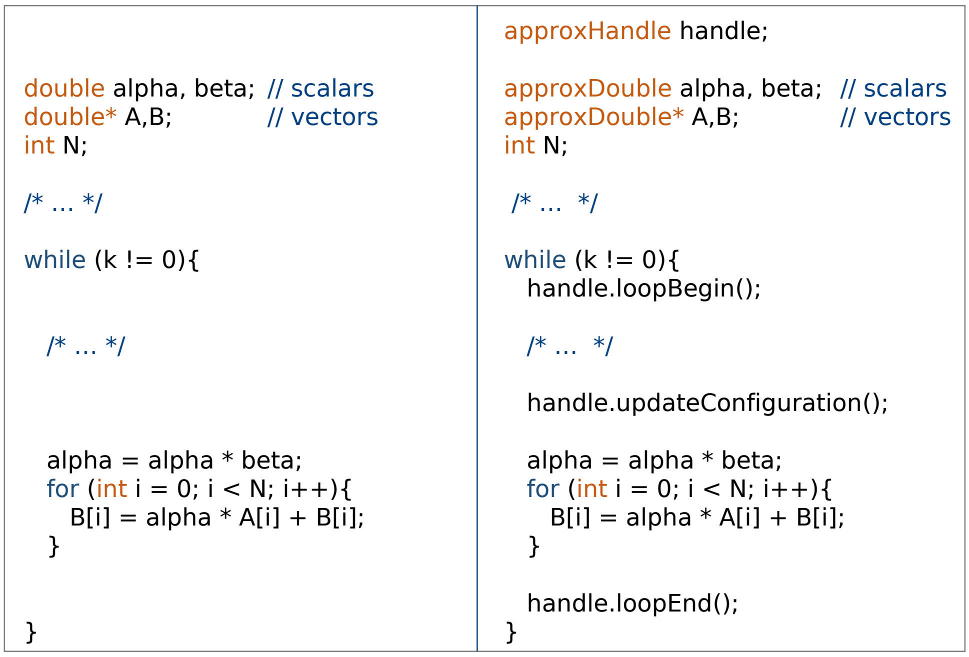
Applications need to be instrumented to investigate their execution on approximate hardware using the three different parameter evaluation methods. As the number of required changes grows with the code size, the instrumentation of existing or new code can become a challenging task. Different interfaces are provided comprising approximate data types and handles that allow the seamless instrumentation of existing or new code. The approximate data types are based on basic arithmetic data types (i.e., integer and floating-point data types), which can be found in any modern programming language (e.g., *C/C++*). Since the remaining instructions remain unchanged, code transformation techniques like [Barba16] can be applied to automatize the code changes.

The interface allows the flexible utilization of the different methods. Different handles allow to switch between the model-based and simulation-based method at runtime. On top of that, these handles denote iterative algorithmic parts in an application as well as configuration changes in the underlying approximation technique (i.e., adaptions in approximation error). The combined estimation method relies on this information to extract and map instruction intervals of iterative algorithmic parts to either model-based estimations or to simulation-based evaluations at runtime. The identification of iterative algorithmic parts like loops can be automatized using techniques like [Hamer05].

Figure 6.2 shows an exemplary instrumentation of a vector addition code that is executed in a loop. A handle is used to denote the begin and end of the loop, which indicates the underlying iterative algorithmic part (i.e., *loopBegin* and *loopEnd*). This handle is also used to denote adaptions of the underlying approximation technique (i.e., *updateConfiguration*), which allows the combined estimation method to evaluate one vector addition execution for each utilized configuration using timing simulation. Besides the initialization of this handle, only variable declarations and function calls need to be adapted to instrument the code.

The application instrumentation approach relies on a flexible data representation to represent the operands of instructions and compute kernels such as linear algebra operations. To illustrate the utilization of this data representation by the different methods, the following notation is utilized. *Computation tuples* S comprise operands that are processed by a common operator and are described as follows:

Let $(a \ \mathbf{op} \ b)$ denote an instruction on input operands a and b which are processed by an operator $\mathbf{op} \in \{+, -, \cdot, /\}$. An *operand vector* O is a vector containing n 2-tuples (a_i, b_i),



```


double alpha, beta;  // scalars
double* A,B;         // vectors
int N;

/* ... */

while (k != 0){


  /* ... */



  alpha = alpha * beta;
  for (int i = 0; i < N; i++){
    B[i] = alpha * A[i] + B[i];
  }


}
```

Original code

```
approxHandle handle;

approxDouble alpha, beta;  // scalars
approxDouble* A,B;         // vectors
int N;

 /* ...  */

while (k != 0){
  handle.loopBegin();

  /* ...  */

  handle.updateConfiguration();

  alpha = alpha * beta;
  for (int i = 0; i < N; i++){
    B[i] = alpha * A[i] + B[i];
  }

  handle.loopEnd();
}
```

Instrumented code

▲ **Figure 6.2** — Original and instrumented code example.

which represent instructions $(a_i \ \mathbf{op}\ b_i)_{1\leq i\leq n}$

$$O := [(a_1,b_1),(a_2,b_2),\dots,(a_n,b_n)] \quad \text{for } (a_i,b_i): \ a_i \ \mathbf{op}\ b_i \ . \tag{6.1}$$

A *computation tuple* S is a 2-tuple that contains one operator $\mathbf{op} \in \{+,-,\cdot,/\}$ and an operand vector O with

$$S :=(\mathbf{op}, O) \ . \tag{6.2}$$

Example 6.1: Given two vectors $\boldsymbol{r} \in \mathbb{R}^{n\times 1}$, $\boldsymbol{x} \in \mathbb{R}^{n\times 1}$ and a scalar $\alpha \in \mathbb{R}$. The vector scaling operation $\boldsymbol{r} := \alpha\boldsymbol{x}$ can be translated to a single computation tuple S with

$$S := (\ \cdot\ , [(\alpha,x_1),(\alpha,x_2),\cdots,(\alpha,x_n)]\ ) \ .$$

▸

The different methods evaluate the operand tuples (a_i,b_i) in operation vectors in parallel, which allows low evaluation runtimes. For this reason, successive arithmetic instructions with data dependencies need to be mapped to multiple computation tuples $S_1, S_2, \cdots, S_n$ in order to resolve such data dependencies between the operand tuples in vector O.

6.3 Simulation-based Parameter Evaluation

The simulation-based parameter evaluation method determines the parameters of interest, namely the area, the leakage power, the dynamic power, the delay, and the approximation error to describe application executions on approximate computing hardware. This method evaluates application-independent parameters from hardware synthesis results, while it evaluates application-dependent parameters using circuit simulation.

The simulation-based estimation method maps the operands $(a_i, b_i) \in O$ in a computation tuple S to a parallel high-throughput gate-level timing simulator accelerated on a GPU [Holst15] to achieve feasible estimation runtimes. For each computation tuple S, a simulation instance evaluates the operands $(a_i, b_i) \in O$ using the circuit description of a target design t. For each target design t, this simulator takes a netlist and the underlying delay annotations in standard delay format (SDF) as input. Such netlists describe precise as well as approximate arithmetic structures. The parallel-pattern simulation technique of this simulator is exploited by the presented method by mapping the operands $(a_i, b_i) \in O$ for a target design t to an *input pattern vector* P^t. The *input patterns* $p_i \in P^t$ are evaluated in concurrent timing simulations on a GPU.

The structure of operand vectors O allows such concurrent timing simulations since the operands $(a_i, b_i) \in O$ do not exhibit data dependencies. This high-throughput simulation technique allows the fast evaluation of complex compute kernels, such as matrix-matrix or matrix-vector multiplications in scientific applications, which often comprise significant amounts of arithmetic operations.

The different application-dependent parameters are determined as follows: For each pattern $p \in P^t$, the output pattern generated during timing simulation is mapped to a *result* r.

To compute the *error due to approximation* e and the *error rate* ρ for *approximate designs*, an additional timing simulation is performed for their precise counterparts. The approximation error e is expressed by the absolute and relative error of a result obtained by an approximate design. The number of non-zero approximation errors is compared to the number of evaluated patterns p to compute the error rate ρ of an approximate design.

The *delay* is determined by the maximum delay that is observed at the circuit outputs for a pattern p.

During a timing simulation, the signal switching activity for each gate output is collected in the circuit. The signal switching activity is used to compute the *Weighted Switching Activity* (WSA) for pattern p as

$$\mathrm{WSA}_p := \textstyle\sum_i f(i) \cdot s(i) \tag{6.3}$$

with $f(i)$ being the fanout and $s(i)$ being the number of transitions at the output of gate i. Standard cell library information such as [Nanga] provides the *energy amount* for signal transitions at gate i, which is denoted by energy_i below.

Using the obtained WSA results and the clock period clk, the *dynamic power* is computed for a pattern p as:

$$\mathrm{Energy}_p := \textstyle\sum_i (f(i) \cdot s(i)) \cdot \mathrm{energy}_i \tag{6.4}$$

$$\text{Dynamic Power}_p := \mathrm{Energy}_p / clk\ . \tag{6.5}$$

6.4 Model-based Parameter Estimation

Different parameters in approximate computing designs determine the probability and magnitude of induced approximation errors. Such parameters have an essential impact on satisfying or violating accuracy bounds of applications. The *model-based estimation method* evaluates models for approximate arithmetic structures that describe the induced approximation error. In the following, a model description is introduced that mimics the induced approximation error of approximate arithmetic structures.

A model for an approximate arithmetic structure $\mathcal{M}$ is a tuple $\mathcal{M} := (t, m, \rho)$, which associates an approximate target design t with an error profile that comprises the error magnitude m and the error rate ρ. The error magnitude is determined by the *maximum relative error* e as described in Equation 5.8. For precision-configurable designs (cf. Chapter 2.4.2), the model description comprises a set of n tuples (t, m, ρ, k), where k denotes the *approximation parameter* to specify configurations and

$$\mathcal{M} := \{(t_1, m_1, \rho_1, k_1), \cdots, (t_n, m_n, \rho_n, k_n)\}\ . \tag{6.6}$$

Approximation techniques are modeled using different approaches. In the following, a model $\mathcal{M}$ is presented for the so-called *truncation and random fill approximation technique*. Truncation-based models are widely used in the literature to describe approximate arithmetic units [Chipp13, Mishr14, Barba16]. This approximation technique performs approximate arithmetic operations by truncating the k least significant bits in the operand values and concatenating the provided *precise* result bits with uniformly random bits. This technique allows to configure the underlying precision by changing the number of approximated bits k. Each value for k corresponds to a specific *approximation level l_i* (cf. Equation 5.4). Higher approximation levels use increasing numbers of approximated bits, while lower approximation levels rely on more precise bits. This technique can be applied to both approximate integer and floating-point arithmetic.

An instruction $(a \textbf{ op } b)$ with input operands $a \in \mathbb{R}$ and $b \in \mathbb{R}$ is approximated as

$$r := \text{approx}\left(\text{trunc}(a) \textbf{ op } \text{trunc}(b)\right) \tag{6.7}$$

with approx(...) describing a function that fills the truncated bits in the result with a random pattern. To adjust the *error rate* in this technique, the truncation operation is skipped in a specific portion of model evaluations.

The model parameters in $\mathcal{M}$ are determined for applying this approximation technique to floating-point operations $\textbf{op} \in \{+, -, \cdot, /\}$ (cf. Appendix C.3) as follows: In floating-point arithmetic, the approximation technique truncates the k least significant bits in the operand mantissas before performing the operation and filling the truncated bits in the result mantissa with a random pattern. This technique allows a user to specify an arbitrary number of configurations (t, m, ρ, k) in which the error rate ρ and the number of truncated bits k for this approximation technique t are user-defined parameters. Following Equation 5.8, the maximum relative error m between an operation $(a \textbf{ op } b)$ and its approximate floating-point representation $fl(a \textbf{ op}_{apx} b)$ is

$$m := \max|\delta| \quad \text{subject to: } fl(a \textbf{ op}_{\text{apx}} b) = (a \textbf{ op } b)(1 + \delta) \tag{6.8}$$

and

$$fl(a \textbf{ op}_{\text{apx}} b) \neq 0 \wedge (a \textbf{ op } b) \neq 0 \,.$$

Given a floating-point operation $(a \textbf{ op } b)$ that is processed by the truncation and random fill approximation technique. For a floating-point representation using p significand bits,

$p - k$ most significant bits in the operand mantissas remain unaffected by truncation. In the worst case, the approximation provides the same result as an operation using $p - k$ mantissa bits. Following Equation 3.11, the maximum relative error m is therefore

$$m := 2^{-(p-k-1)} \,. \tag{6.9}$$

Depending on the approximate target designs, the model description needs to be adapted. Approximate data representations [Chipp13] mimic the approximation error that is introduced to operands and results in approximate memory. Approximate arithmetic operations such as additions and multiplications are modeled by evaluating the original operations while inducing errors to the underlying operands or result values according to a specified *error profile*. Error profiles mimic the induced approximation error by, for instance, adding random numerical errors sampled from probability mass functions [Huang12, Lee16].

6.5 Combined Parameter Estimation

To determine application-dependent parameters like the dynamic power dissipation of an application execution, all instructions in an application can be evaluated by the simulation-based estimation method, which provides an exhaustive insight into the different parameters for all evaluated instructions. Such an exhaustive insight is obtained by mapping all instructions in an application to computation tuples S that are completely evaluated to obtain parameter results for all underlying operands (a_i, b_i). However, such an *exhaustive exploration* that evaluates all instructions in an application execution using gate-level timing simulation can cause long runtimes.

The dependency between operand values, approximation techniques, and application-dependent parameters induces a major challenge to reduce the number of timing simulations while achieving accurate evaluation results. Since the induced approximation error affects different application-dependent parameters, the propagation of approximation errors between executed instructions has to be evaluated. At the same time, precision-configurable approximate hardware needs to be considered, since such hardware adapts the degree of induced approximation error.

The *combined parameter estimation method* combines model-based and simulation-based instruction evaluations, which allows to estimate application-dependent parameters for complete application executions while significantly reducing the number of timing simulations. This approach exploits the insight that carefully selected instruction intervals allow to estimate application-dependent parameters with high accuracy [Hsieh98, Wunde03, Hamer05]. The parameter estimation method targets iterative algorithmic parts that are executed using precision-configurable approximation techniques. Between such iterations, these approximation techniques can adapt the degree of induced approximation error. Instead of only evaluating the selected instruction intervals, all instructions in an application execution are evaluated while the underlying estimation method is switched between model-based evaluations and timing simulations, which allows to evaluate both the impact of approximation error adaptions as well as the approximation error propagation.

For an iteration of an iterative algorithmic part, an *instruction interval* is a vector that contains n computation tuples S

$$I := [S_1, S_2, \cdots, S_n] \tag{6.10}$$

such that I comprises the executed instructions of the iteration.

The term Θ denotes the set of instruction intervals I that are executed for an iterative algorithmic part (i.e., all iterations of the algorithmic part). The estimation method distinguishes the set of instruction intervals Θ into two subsets, *representative instruction intervals* Θ_S that are evaluated by timing simulations and *remaining instruction intervals* Θ_M that are evaluated by model-based estimations. Representative instruction intervals $I \in \Theta_S$ are selected in the two following cases: The first iteration of an iterative algorithmic part forms the first representative instruction interval. On top of that, further representative instruction intervals are formed whenever the induced approximation error is adapted in the underlying approximation technique.

The power dissipation (i.e., comprises both dynamic power and leakage power) and the delay are estimated using representative instruction intervals for complete application execution as follows: For each computation tuple S in a representative instruction interval $I \in \Theta_S$, the average power dissipation $Power_{AVG}$ and the average delay $Delay_{AVG}$ are computed using the power dissipation and delay results that were obtained for the underlying operands (a_i, b_i). The average power dissipation $Power_{AVG}$ and delay results $Delay_{AVG}$ are used as estimations for computation tuples S in subsequent instruction intervals $I \in \Theta_M$ until another representative instruction interval $I \in \Theta_S$ is reached in the sequence of instruction intervals.

For a computation tuple S that contains n operands (a_i, b_i), the average power dissipation $Power_{AVG}$ is computed as

$$Power_{AVG} := \sum_{i=1}^{n} \frac{\text{Dynamic and leakage power to compute operands } i}{n} \ . \tag{6.11}$$

A central *derived* parameter is the energy demand of a computation tuple S. The energy demand of a computation tuple S is estimated from the average power dissipation $Power_{AVG}$ and the average delay $Delay_{AVG}$ to compute one instruction with

$$Energy := n \cdot Power_{AVG} \cdot Delay_{AVG} \ . \tag{6.12}$$

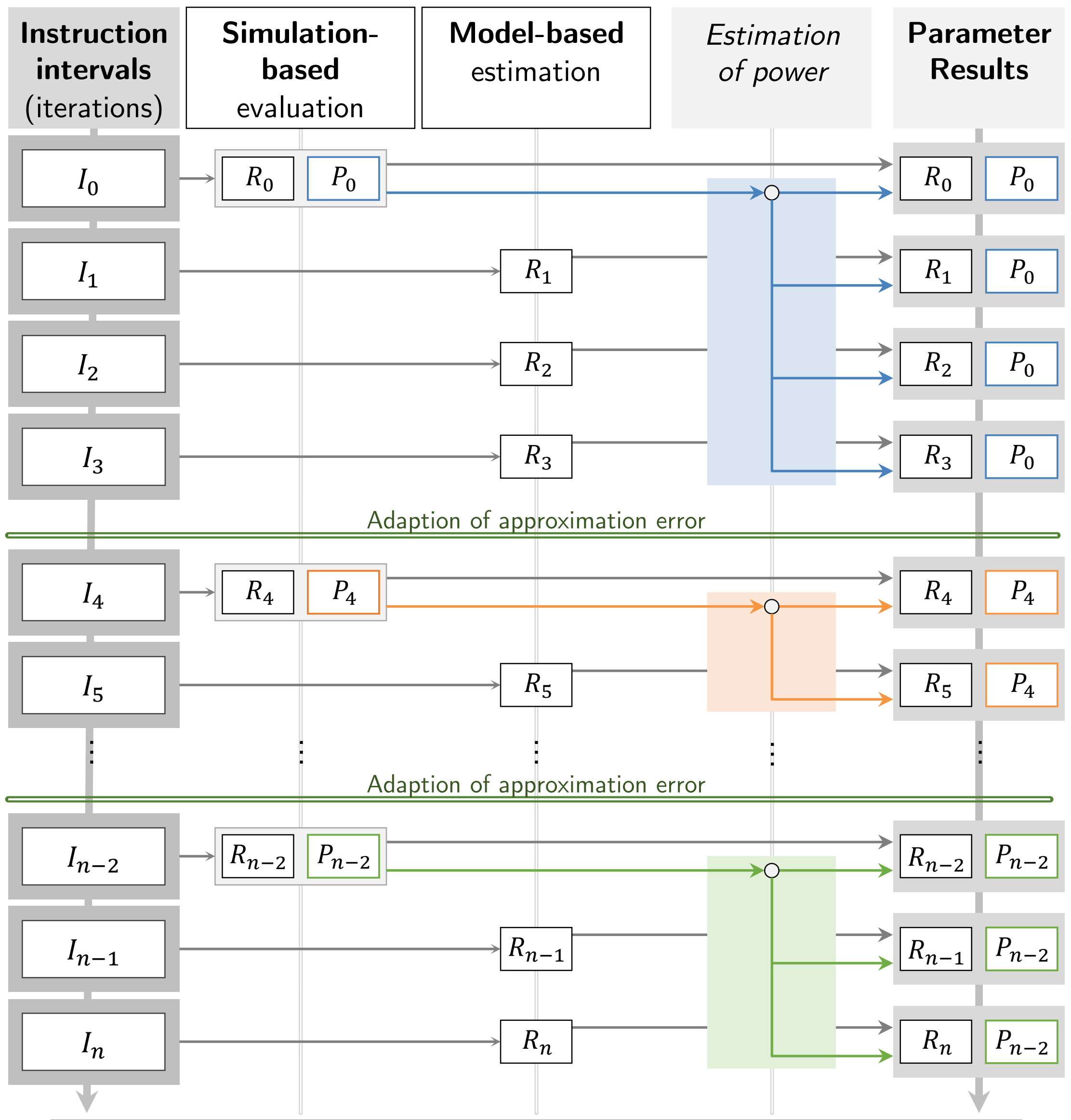
Let $Energy_{i,k}$ denote the estimated energy demand of the i-th computation tuple S_i in the k-th instruction interval of an application execution. To estimate the energy for complete application executions that contain M instruction intervals, the energy demand of all executed computation tuples is summed and

$$\text{Application Energy} := \sum_{k=1}^{M} \sum_{i=1}^{N} Energy_{i,k}, \quad \text{for } S_i \in I_k, I_k \in \Theta \ . \tag{6.13}$$

Using Equation 2.45 in Chapter 2.4.2, the power dissipation results and the time required to process the instructions are used to evaluate the computational performance and energy efficiency (i.e., *Watt-per-MIPS*^{2} metric) of an application execution.

Figure 6.3 illustrates the combined estimation method at the example of a loop, which is executed using a precision-configurable approximation technique.

The instructions in each loop iteration are represented by a sequence of instruction intervals $I_0, \cdots, I_n$ which are sequentially evaluated. The first instruction interval I_0 is a representative instruction interval $I_0 \in \Theta_S$, which is evaluated by the simulation-based parameter evaluation method to determine the power dissipation results P_0 (i.e., power dissipation results of underlying compute tuples S) and the result r_0 (i.e., approximate results in the underlying program variables) for the instructions in I_0. The instruction intervals I_1 to I_3 are remaining instruction intervals $I_1, I_2, I_3 \in \Theta_M$, since these intervals are evaluated with the same approximation error configuration as I_0. The intervals I_1 to I_3 are evaluated by the model-based estimation method to compute the approximation error propagation. The results of these intervals are denoted by r_1 to r_3. The power dissipation results in P_0 are used as an estimation for the power dissipation for intervals I_1 to I_3.



I_k	Instruction interval in iteration k
R_k	Approximate arithmetic results in iteration k
P_k	Power dissipation results in iteration k

▲ **Figure 6.3** — Example for applying the *combined parameter estimation method* to estimate the power dissipation of a loop execution.

After interval I_3, the induced approximation error is adapted. For this reason, the instruction interval I_4 is a representative instruction interval $I_4 \in \Theta_S$, which is evaluated using timing simulations to determine the power dissipation results P_4. The power dissipation results P_4 are used as estimations for all intervals between interval I_4 and the next approximation adaption after I_5. The remaining intervals are evaluated accordingly by switching between the simulation-based and the model-based method after adaptions of the induced approximation error.

CHAPTER 7

Experimental Evaluation and Results

This chapter discusses the experimental evaluation for the methods presented in this thesis. The benchmark data set presented below in Section 7.1 was used to evaluate the different methods. These benchmarks are based on different real-world problems from science and engineering and allow reliable conclusions on the performance of the presented methods in the corresponding applications. The presented fault tolerance methods (cf. Chapter 3 and Chapter 4) were evaluated using the *error model* described below in Section 7.2. The approximation model presented below in Section 7.3 was applied to evaluate the approximate computing technique presented in Chapter 5.

The central findings for the different methods are summarized and discussed in this chapter. The experimental results that were obtained for the fault-tolerant sparse matrix-vector multiplication (cf. Chapter 3) are presented in Section 7.4. Section 7.5 presents the experimental results for the efficient fault tolerance technique that targets Conjugate Gradient solvers (cf. Chapter 4). The parameter evaluation and estimation methods (cf. Chapter 6) are evaluated in Section 7.6. The technique that enables Conjugate Gradient solvers on approximate hardware (cf. Chapter 5) is evaluated below in Section 7.7. To ensure a concise presentation of the experimental results, the detailed data has been moved into Appendix E.

7.1 Benchmark Matrices and Setup

The benchmark data set comprises 30 matrices from the Florida Sparse Matrix Collection [Davis11], which are shown in Table 7.1. Besides the names and the size of the different matrices $N \times N$, the number of nonzero elements *NNZ* are presented. As a side information, the portion of 0*s* within the matrices is shown (i.e. number of zero elements in each matrix divided by the total number of elements).

▼ **Table 7.1** — Overview of evaluated matrices from the Florida Sparse Matrix Collection [Davis11].

Matrix name	Matrix size N	Nonzero elements	Portion of 0s	Description
nos3	960	15,844	98.2808%	Biharmonic operator on plate
bcsstk10	1,086	22,070	98.1287%	Buckling of hot washer
msc01050	1,050	26,198	97.6238%	Symmetric test matrix
bcsstk21	3,600	26,600	99.7948%	Clamped square plate
bcsstk11	1,473	34,241	98.4219%	Ore car (Lumped Masses)
nasa2146	2,146	72,250	98.4312%	Test structure (NASA)
sts4098	4,098	72,356	99.5692%	Structural engineering matrix
bcsstk13	2,003	83,883	97.9092%	Fluid flow
msc04515	4,515	97,707	99.5207%	Test matrix (MSC/Nastran)
ex9	3,363	99,471	99.1205%	Test matrix (FIDAP)
bodyy4	17,546	121,550	99.9605%	Structrual engineering matrix
bodyy5	18,589	128,853	99.9627%	Structrual engineering matrix
bodyy6	19,366	134,208	99.9642%	Structrual engineering matrix
Muu	7,102	170,134	99.6627%	Test matrix (Mathworks)
s3rmt3m3	5,357	207,123	99.2783%	Analysis of cylindrical shells
s3rmt3m1	5,489	217,669	99.2775%	Analysis of cylindrical shells
bcsstk28	4,410	219,024	98.8738%	Solid element model
s3rmq4m1	5,489	262,943	99.1273%	Analysis of cylindrical shells
bcsstk16	4,884	290,378	98.7827%	Dam
Kuu	7,102	340,200	99.3255%	Test matrix (Mathworks)
bcsstk38	8,032	355,460	99.4490%	Airplane engine component
msc23052	23,052	1,142,686	99.7850%	Test matrix (MSC/Nastran)
msc10848	10,848	1,229,776	98.9550%	Test matrix (MSC/Nastran)
cfd2	123,440	3,085,406	99.9798%	Symmetric pressure matrix
nd3k	9,000	3,279,690	95.9510%	3D mesh problem
ship_001	34,920	3,896,496	99.6805%	Ship structure
shipsec5	179,860	4,598,604	99.9858%	Ship section (PARASOL)
G3_circuit	1,585,478	7,660,826	99.9997%	Circuit simulation problem
hood	220,542	9,895,422	99.9797%	Test matrix (INDEED)
crankseg_1	52,804	10,614,210	99.6193%	Static analysis of a crankshaft detail

These matrices comprise *square symmetric* matrices that are positive-definite and contain real numbers. The matrix size (i.e., matrix dimension N) ranges from 960 to 1 585 478 while the number of non-zero elements ranges from 15 844 to 10 614 210. At the same time, the sparsity (i.e. portion of $0s$) of these matrices ranges from 95.95% to 99.9996%. These matrices represent different real-world problems including structural engineering (e.g. *bodyy6*), mechanics (e.g. *bcsstk38*), computational fluid dynamics (e.g. *cfd2*) and semiconductor device simulation (e.g. *G3_circuit*).

The matrices were utilized to solve linear systems $\boldsymbol{Ax} = \boldsymbol{b}$ using the Conjugate Gradient solvers (cf. Chapter 2.1.2). These linear systems were generated as follows: The right-hand side vector $\boldsymbol{b}$ was generated following [Cools16] as $\boldsymbol{b} := \boldsymbol{A}\tilde{\boldsymbol{x}}$ with $\tilde{x}_i := 1/\sqrt{N}$ and N being the matrix size. The right-hand side vector $\boldsymbol{b}$ was changed between experiments using a *random vector* $\boldsymbol{r}$ to avoid repeating solver setups with $\boldsymbol{b} := \boldsymbol{A}\tilde{\boldsymbol{x}}$ and $\tilde{x}_i := 1/\sqrt{N} + r_i/\|\boldsymbol{r}\|_2$. The elements in $\boldsymbol{r}$ were generated using the *Mersenne Twister* pseudorandom number generator [Matsu98] within the IEEE-754 double-precision range [IEEE 08] with $r_i \in [-1.798 \cdot 10^{308}, 1.798 \cdot 10^{308}]$. The elements in the initial guess vector $[x^{(0)}]_i$ were set to b_i. The execution of the solvers was continued until the residual $\delta^{(k)}$ fell below the *absolute accuracy tolerance* ϵ_a or $\delta^{(k)}/\delta^{(0)}$ fell below the *relative accuracy tolerance* ϵ_r. The absolute error tolerance ϵ_a was set to 10^{-6} and the relative error tolerance ϵ_r was set to 10^{-15}. An experiment was considered a failure, if the number of iterations exceeded the iteration limit, namely $10 \cdot N$ iterations with N being the matrix size. Three different representatives of Conjugate Gradient solvers were evaluated, namely the CG solver (i.e. does not rely on a preconditioner), the PCG solver with *Jabobi* preconditioning as well PCG with *incomplete Cholesky factorization* (ICC) as preconditioner. Details on these preconditioners are presented in Appendix A.2.

To accelerate the experiments, all parallelizable linear algebra operations in the evaluated Conjugate Gradient algorithms were mapped to a heterogeneous computing system, which is described in Appendix E.1. Linear algebra operations were mapped to GPU architectures based on GPU-accelerated mathematical libraries, namely the CuBLAS [Nvidia] and CuSPARSE library [Nvidib]. The library calls summarized in Table 7.2 were utilized for the different linear algebra operations. The evaluated matrices were stored in the compressed sparse row storage format (CSR) [DAzev05] to avoid unnecessary overhead in matrix operations due to multiplications by 0. All experiments have been performed in double-precision floating-point arithmetic. Each experiment was executed using a combination of a single CPU core and a single GPU.

▼ **Table 7.2** — Overview of the parallelizable linear algebra operations in the evaluated Conjugate Gradient algorithms and their associated GPU-accelerated library call.

Operation	Library call	Library
Sparse matrix-vector multiplication	cusparseDcsrmv	cuSPARSE
Inner product	cublasDdot	cuBLAS
Vector addition	cublasDaxpy	cuBLAS
Euclidean vector norm	cublasDnrm2	cuBLAS
Jacobi preconditioning	cusparseDcsrmv	cuSPARSE
Incomplete Cholesky preconditioning	cusparseDcsrsv_solve	cuSPARSE

7.2 Error Model

The experimental evaluation focuses on transient events that cause errors in the outputs of arithmetic computations. Erroneous arithmetic outputs may lead to Silent Data Corruptions (SDC). Such corruptions of outputs may occur in the arithmetic components of a processor due to the manifestation of faults in form of errors. This error model does not inject errors in memory elements as hardware fault tolerance techniques like ECC [Mitta16b] are often used in high-performance systems to protect main memories, caches and register files. The model also does not inject errors in the control logic or in the encoding of instructions as it considers them to be protected by appropriate measures like assertions and embedded signatures [Khudi13].

Different implementations of floating-point units exist that may have different error propagation patterns for transient events. In accordance to related works [Brone08, Sloan12, Sloan13, Liu15b, Tao16, Diche16], errors are injected into the outputs of computations at runtime by instrumenting the application to perform random error injections.

To evaluate the fault tolerance technique that targets sparse matrix-vector multiplications (cf. Chapter 3), errors are injected into a randomly selected element within the result vector of the matrix operation. In executions of the Conjugate Gradient solver, errors were injected by randomly selecting both an iteration and one of the operations in the solver to generate erroneous results. If the selected operation computes a vector as result, then one element in this result vector was randomly chosen. All error injection locations were selected based on uniformly distributed random numbers, that were generated using the *Mersenne Twister* pseudorandom number generator [Matsu98].

During error injections, two forms of bit flip errors were evaluated, namely single-bit and multi-bit flip error events. To evaluate multi-bit flip error events, the results of the underlying floating-point instructions are modified by randomized bursts of bidirectional bit flips. The position of a burst is randomly chosen from a uniform distribution within the 64 bits of the floating-point values. Following the survey results in [Di Ma16], the number of bits affected by such bursts are randomly chosen from a normal distribution with mean = 5 and variance = 3. Bit flips were also injected into operations that perform error detection.

7.3 Approximation Model

The software-based model presented in Chapter 6.4 was utilized to evaluate the presented approximation methods. This model was used to approximate the calculation of mantissa values during floating-point multiplications. In each operation, these approximate floating-point operations truncate k least significant bits in the operand mantissas and concatenate the computed *precise* result bits with uniformly random bits, which mimic approximately computed bits.

By changing the number of approximated bits k, the approximate floating-point model mimics precision-configurable approximation techniques that provide different *approximation levels*. Ten approximation levels were utilized that provide different numbers of precise mantissa bits $p \in \{2, 7, 17, 22, 27, 32, 37, 42, 47, 52\}$.

In executions of the Conjugate Gradient solver, this approximation model was applied to the dominant operation in the solver, namely the floating-point multiplications [Zhang14b] in sparse matrix-vector multiplications.

Gate-level hardware descriptions were developed that reflect the behavior of this approximation model in hardware. These hardware descriptions are used to obtain insights into the energy efficiency of solver executions by computing the power dissipation and solver runtime from performing timing simulations. Following the description in [Desch06, chp. 16], combinational gate-level descriptions for a floating-point adder and a floating-point multiplier in double-precision arithmetic were derived. The resulting hardware descriptions were synthesized using combinational two-input gates from the *NanGate* 45 nm library [Nanga].

Both hardware descriptions comprise 128 inputs and 64 outputs, respectively. While the multiplier contains 20,812 two-input gates and a critical path delay of 14.21 ns, the adder contains 5,678 gates and a critical path delay of 14.99 ns. The leakage power of the multiplier is 0.154 mW while it is 0.033 mW for the adder.

Both the software-based model as well as the timing simulations evaluated the underlying floating-point operations in double-precision arithmetic.

7.4 Fault-tolerant Sparse Matrix-Vector Multiplication

This section evaluates the fault tolerance technique for sparse matrix-vector multiplications that was presented in Chapter 3. The experimental results presented below show the *runtime overhead* for both *error detection* and *correction* as well as the achievable *error coverage*. The underlying error detection scheme is compared to the related work approach described in Chapter 2.3.1 (cf. Equation 2.34) that is used by different related works [Brone08, Shant12, Sloan12, Sloan13, Fasi16] to protect sparse matrix-vector multiplications. The presented error correction scheme is compared to complete re-executions of the matrix-vector multiplication.

The presented fault tolerance technique provides different parameters that have been evaluated with respect to their influence on the runtime and the error coverage. Different block sizes σ_k have been selected to evaluate the interplay between the chosen block size, the resulting error coverage, and the induced runtime overhead for error detection and correction. Besides, two different kinds of errors have been evaluated, namely single-bit flip errors as well as multi-bit flip errors. For each experiment with single-bit flip error injections, a second run of the experiment was performed using multi-bit flip error injections instead.

7.4.1 Runtime Overhead

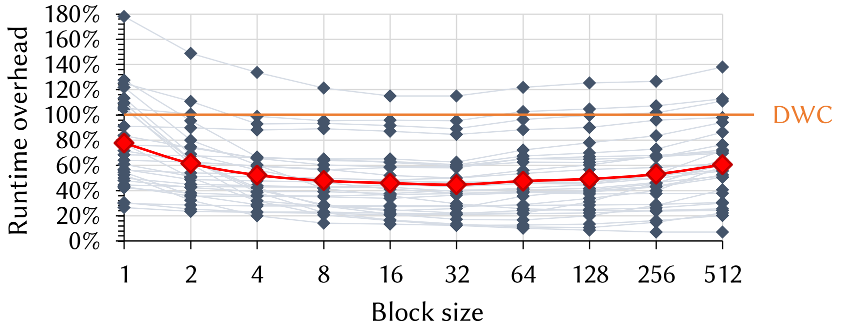
The fault tolerance technique induces some runtime overhead to detect errors as it complements the original matrix-vector multiplication with additional operations. The runtime overhead for error detection is computed as:

$$\text{Runtime overhead} = \left(\frac{\text{Runtime for protected operation}}{\text{Runtime for original operation}} - 1 \right) \cdot 100\%.$$

To obtain the runtime overhead for error detection steps, 1000 experiments were performed for each matrix and each selected block size in which no errors were injected.

The *runtime overhead for error detection* that is induced by the fault tolerance technique depends on the block size σ_k. The runtime overhead has been evaluated for different block sizes ranging from 1 to 512. If a matrix could not be partitioned into m' equally sized row block matrices, then the first $m' - 1$ row blocks were equally sized, while the last row block covered the remaining rows in the matrix.

Figure 7.1 compares the runtime overhead for error detection for each matrix with respect to different block sizes. Each blue data point denotes one runtime overhead result for a specific matrix. The red graph depicts the average runtime overhead over all evaluated matrices with respect to changing the block size. At the same time, the presented fault tolerance is compared to the runtime overhead induced by the *Duplication with Comparison* (DWC) technique (cf. Chapter 2.2.3). The runtime overhead induced by DWC is depicted by the orange graph (i.e. 100% runtime overhead).

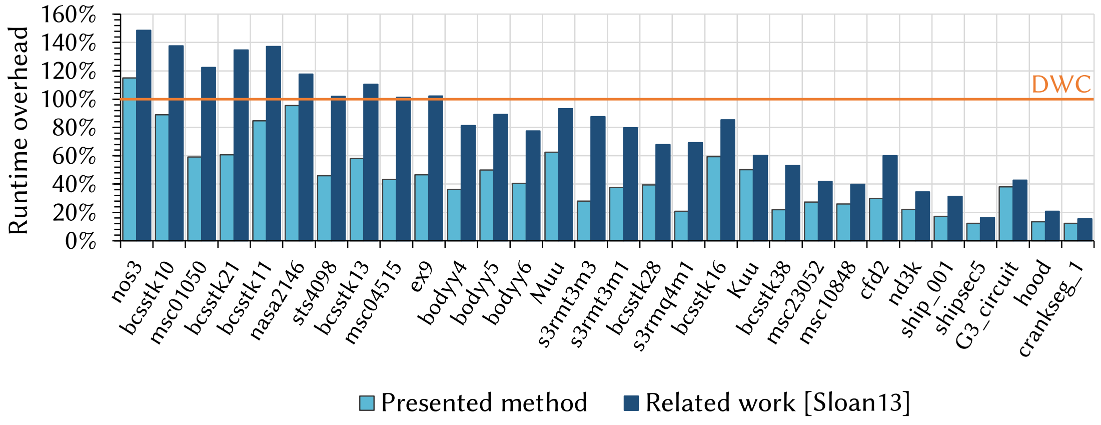


▲ **Figure 7.1** — Runtime overhead of the protected matrix-vector multiplication for different block sizes compared to unprotected executions.

The average runtime overhead is ranging from 44.8% for block size 32 to 77.6% for block size 1. The maximum runtime overhead is observed for block size 1, in which case the checksum matrix $\boldsymbol{C}$ is equal to the input matrix $\boldsymbol{A}$. In that case, the computation of both the original operation $\boldsymbol{Ab}$ and the checksum generation $\boldsymbol{Cb}$ are equal operations and exhibit the same complexity. For block size 1, measured runtime overheads below 100% can be explained by the parallel execution of these two operations.

As block size 32 provides minimum average error detection overhead, the corresponding experiments are further evaluated below. Detailed results for the other block sizes can be found in Appendix E.2.

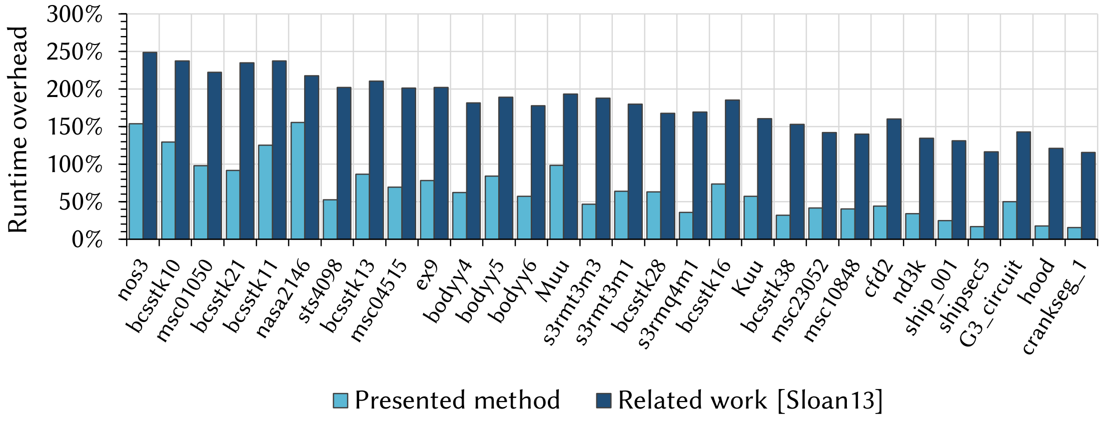
The error detection overhead is shown for each matrix in Figure 7.2 for a block size of 32. In this figure, the error detection overhead of the presented fault tolerance technique is compared against the related work approach that evaluates Equation 2.34 to detect errors [Sloan13]. The orange graph depicts the runtime overhead that is induced by the *Duplication with Comparison* (DWC) technique (i.e. 100% runtime overhead).



▲ **Figure 7.2** — Runtime overhead for error detection in case of a block size of 32.

To ensure a fair comparison, the error checking steps in the related work approach were parallelized with the original matrix-vector multiplication ($\boldsymbol{r} := \boldsymbol{Ab}$). The evaluated matrices are ordered by increasing size. The experimental results show that the runtime overhead decreases for both fault tolerance techniques with increasing matrix size. The runtime overhead for error detection ranges from 12.4% to 115.0% for the presented fault tolerance technique while it ranges from 15.4% to 148.7% for the related work approach. On average, the runtime overhead is reduced by 43.1% when both approaches are compared. The minimum reduction has been measured with 11.4% with matrix *G3_circuit* while the maximum reduction has been measured 69.9% with matrix *s3rmq4m1*. Compared to the DWC technique, which induces runtime overhead of at least 100%, the runtime overhead is reduced for 29 out of 30 matrices. With increasing matrix size, the reduction in runtime overhead compared to DWC becomes increasingly significant.

In case of errors, some additional runtime overhead is introduced by the presented fault tolerance technique to locate and correct detected errors. The runtime overhead for both *detecting and correcting errors* has been evaluated by injecting one error into the matrix-vector multiplication while comparing the runtime to the unprotected operation. Additional experiments have been performed in which errors were injected into error detection operations. Figure 7.3 shows the results of this investigation.



▲ **Figure 7.3** — Runtime overhead for error detection and correction in case of a block size of 32.

In each experiment, one matrix-vector multiplication has been performed while one element in the output vector was injected which triggered error corrections in all evaluated methods. The presented error correction technique is compared to re-executing the complete matrix-vector multiplication in case of detected errors. For each matrix and each selected block size, 1000 error injection experiments were performed.

The runtime overhead for both error detection and correction ranges from 15.6% to 155.6% for the presented fault tolerance technique, while it ranges from 115.5% to 248.7% for the related work approach. The runtime overhead is on average reduced by 63.9%. The minimum reduction is 28.4% with matrix *nasa2146* and the maximum reduction is 86.5% with matrix *crankseg_1* compared to the related work approach.

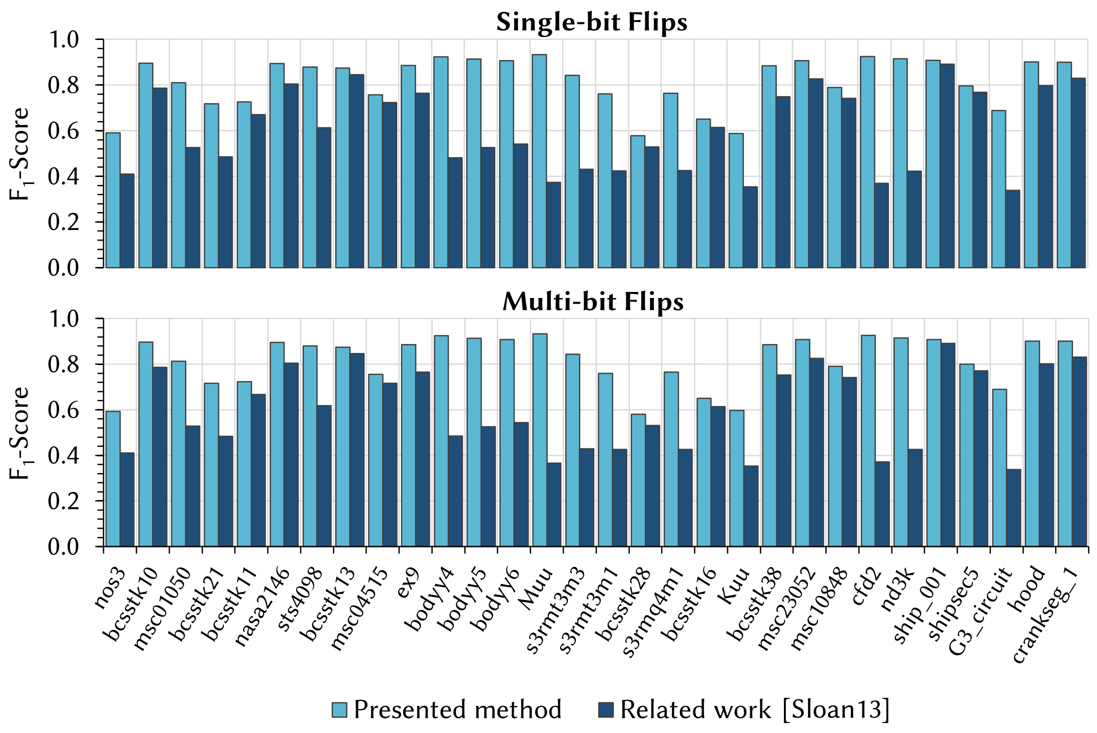
7.4.2 Error Coverage

To evaluate the error coverage for the fault-tolerant sparse-matrix vector multiplication, errors have been injected into the operation while the reaction of the fault tolerance technique was observed. For each matrix and each selected block size, 1000 error injection experiments have been performed to compare the effectiveness of the presented technique against the related work approach [Sloan13]. In each experiment, one matrix-vector multiplication has been performed while one error was injected to the output of the operation or into the error detection operations. The rounding error bound presented in Chapter 3.2 was applied to distinguish harmful from acceptable errors for the presented fault tolerance technique. For the related work approach, the analytical rounding error bound [Chowd96] was applied (cf. Equation 3.12).

From the experimental results, the *balanced F_1-score* [Van R79] is computed which quantifies the error coverage based on the number of successfully detected errors (*true positives*), the number of undetected errors (*false negatives*) and the number of mistakenly identified errors (*false positives*) with

$$F_1 = \frac{2 \cdot \text{true positives}}{2 \cdot \text{true positives} + \text{false negatives} + \text{false positives}}.$$

With an F_1-score close or equal to 1, the number of successfully detected errors is significantly larger than the number of undetected and mistakenly identified errors. Smaller F_1-scores correspond to increased numbers of undetected and mistakenly identified errors. Figure 7.4 shows the results of this investigation by presenting the F_1-scores for the different methods.



▲ **Figure 7.4** — Comparison of error coverage using the F_1-score.

For single-bit error injections, the F_1-score ranges from 0.578 to 0.932 and is on average 0.817. Compared to the related work approach, the F_1-score is on average improved by 35.7%. The minimum improvement is 1.9% with matrix *ship_001*. The maximum improvement is reported with matrix *cfd*, for which the F_1-score is 150% larger compared to the related work approach.

The F_1-score ranges from 0.582 to 0.933 and is on average 0.818 for multi-bit flip error injections. At the same time, the improvement of the F_1-score for the presented method ranges from 1.9% with matrix *ship_001* up to 155% for matrix *Muu*.

7.4.3 Discussion of Experimental Results

The experimental evaluation demonstrates that the presented fault-tolerant sparse matrix-vector multiplication allows the efficient algorithmic detection and correction of erroneous operation results while it provides high error coverage. By selecting a suitable block size, the underlying error detection scheme induces only low runtime overhead. With a block size of 32, the runtime overhead is on average 44.8% and is at most 115.0% for the evaluated matrices. In case of errors, the runtime overhead for both error detection and correction is at most 155.6%. Compared to the related work approach, this runtime overhead is on average reduced by 63.9%. This low runtime overhead can be explained by the implicit localization of errors, which allows partial recomputations just for erroneous outputs directly after error detection.

A high error coverage is achieved while the number of false positive error detections due to rounding errors is minimized. The presented rounding error function distinguishes harmful from acceptable errors in the magnitude of rounding errors and achieves an average F_1-score of 0.817 for single-bit flip errors and 0.818 for multi-bit flip errors. The highest F_1 of 0.993 was measured with matrix *Muu*. Compared to the related work approach, the presented fault tolerance technique improves the F_1-score by up to 155%. This high error coverage can be explained by the combination of the implicit error localization scheme and the presented rounding error function, which is tailored to sparse matrix-vector multiplications.

7.5 Fault Tolerance for Conjugate Gradient Solvers

This section evaluates the fault tolerance technique that protects Conjugate Gradient solvers, which was presented in Chapter 4. This technique has been evaluated with respect to the *error detection runtime overhead* in error-free executions and the *error correction runtime overhead* in case of errors. The vulnerability of unprotected Conjugate Gradient solvers to errors is demonstrated to emphasize the demand for effective fault tolerance measures.

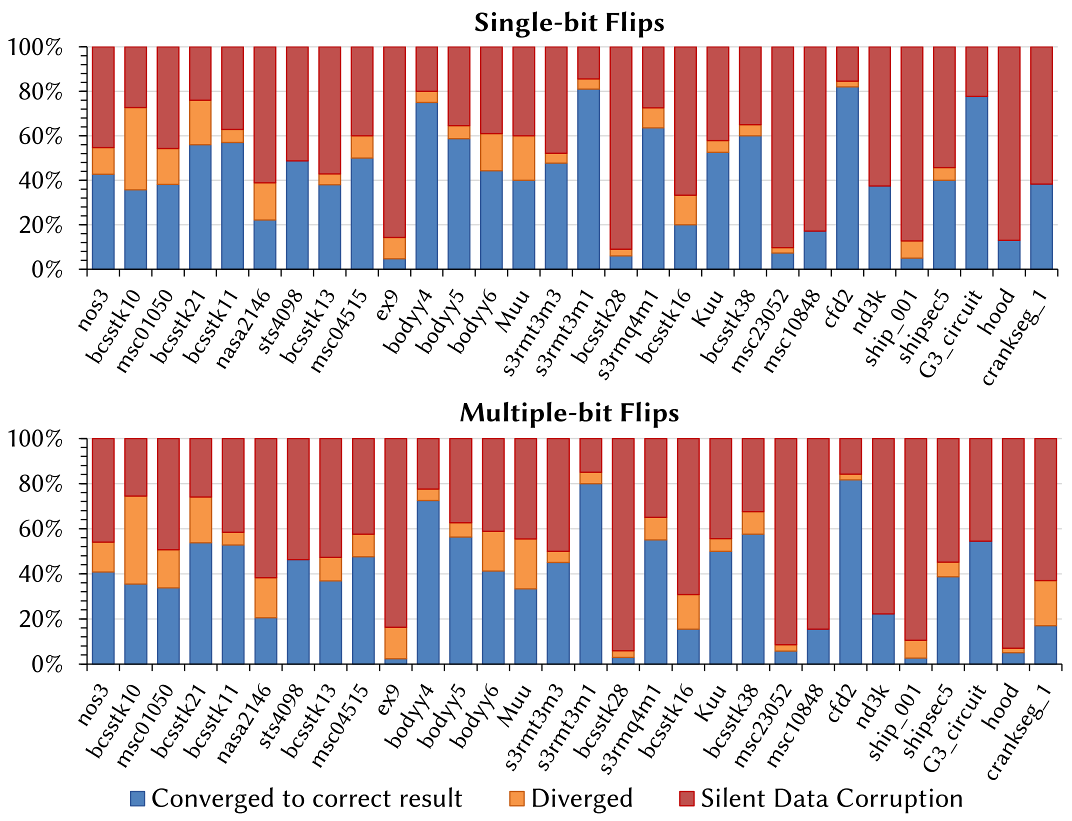
Each experiment comprises a complete run of the solver and a certain number of error injections according to the error model as discussed above in Section 7.2. In the course of the experiments, the influence of different parameters has been evaluated: By changing the *number of injected errors* per experiment, the effectiveness of this fault tolerance technique is evaluated with respect to the error coverage and the error correction runtime overhead. Besides, the influence of the *rounding error threshold* τ (cf. Equation 4.10) on both the error correction runtime overhead as well as the error coverage is evaluated. Two different kinds of errors have been evaluated, namely single-bit flip errors as well as multi-bit flip errors. For each experiment that has been performed with single-bit flip error injections, a second run was performed using multi-bit flip error injections instead.

7.5.1 Vulnerability of Conjugate Gradient Solvers

In a preliminary step, the vulnerability of Conjugate Gradient solvers has been evaluated by performing error injection experiments without applying any additional instrumentation by fault tolerance measures to the solver. In this evaluation, 3000 error injection experiments were performed for each matrix. In the course of each experiment, a complete run of the solver was performed while a single error was injected in a randomly chosen iteration. In such iterations, a result register of a linear algebra operation was randomly selected to inject an error. Figure 7.5 shows the vulnerability of Conjugate Gradient solvers to single and multi-bit flip errors with respect to the proportion between *successfully converged* experiments, *diverged* experiments as well as experiments that resulted in *silent data corruptions (SDC)*. The evaluated matrices are ordered by the number of non-zero elements (*NNZ*). In these figures, the results for applying the *Jacobi*-preconditioner are shown. Further results were obtained for the other two preconditioning cases, which do not show significant differences. These results are presented in Appendix E.3.

In this evaluation, a successfully converged experiment refers to a solver execution that provided a correct result within the iteration limit. Such a correct result $\boldsymbol{x}^{(k)}$ is constituted by a residual $\delta^{(k)}$ that satisfies either the absolute accuracy tolerance ϵ_a or the relative accuracy tolerance ϵ_r (cf. Section 7.1). At the same time, an experiment is referred to as diverged if the number of iterations exceeded the iteration limit. Silent data corruptions are a result of experiments in which the provided result $\boldsymbol{x}^{(k)}$ does not satisfy the underlying linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. In such a case, the provided result $\boldsymbol{x}^{(k)}$

neither satisfies the absolute accuracy tolerance ϵ_a nor the relative accuracy tolerance ϵ_r despite apparent solver convergence.

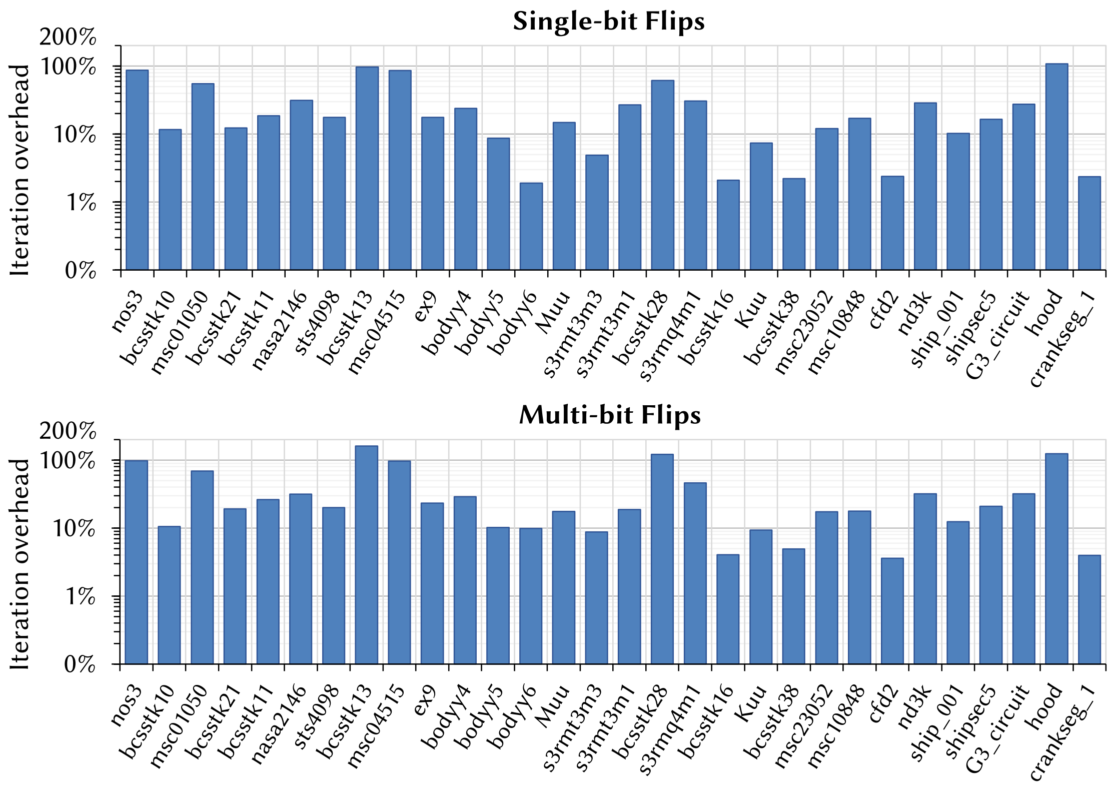


▲ **Figure 7.5** — Proportion of successfully converged experiments, diverged experiments as well as experiments that resulted in silent data corruptions (SDC) in case of errors.

For single-bit errors, the proportion of successfully converged solver executions is on average 42.0% and ranges from 4.8% to 82.1%. Multi-bit error injections reduce the number of successfully converged solver executions on average by 11%. For 20 matrices, the solver only converged in at most 50% of the evaluated experiments to a correct result. For single-bit errors, the proportion of solver executions that exceeded the iteration limit (i.e. diverged) is on average 8.1% and ranges from 0% to 37.0%. Multi-bit flip errors increase the number of diverged experiments on average increased by 17.3%. At the same time, the proportion of silent data corruptions ranges from 14.3% up to 90.9% for single-bit flip errors and from 15.0% up to 94.2% for multi-bit flip errors. While the average number of silent data corruptions is 49.9% for single-bit flip errors and 52.8% for multi-bit flip errors, it exceeds 60% for 10 out of 30 matrices.

The experiments in which the solver converged to a correct result have been further evaluated with respect to the iteration overhead. Figure 7.6 shows the average iteration overhead for these successfully converged experiments compared to error-free solver executions. The depicted runtime overhead was calculated as

$$\text{Iteration overhead} := \left(\frac{\text{Solver iterations in case of errors}}{\text{Solver iterations in error-free case}} - 1\right) \cdot 100\%.$$



▲ **Figure 7.6** — Average iteration overhead to converge to correct results in case of errors.

In case of single-bit flip errors, the iteration overhead required to converge to a correct result ranges from 1.9% to 108.7% and is on average 28.2% when no fault tolerance measure is applied. Multi-bit flip errors increase this overhead on average by 23.2% compared to single-bit flip errors. In that case, the iteration overhead ranges from 3.6% to 161.4%. For three matrices, the average overhead exceeds 100% which corresponds to completely repeating the solver execution.

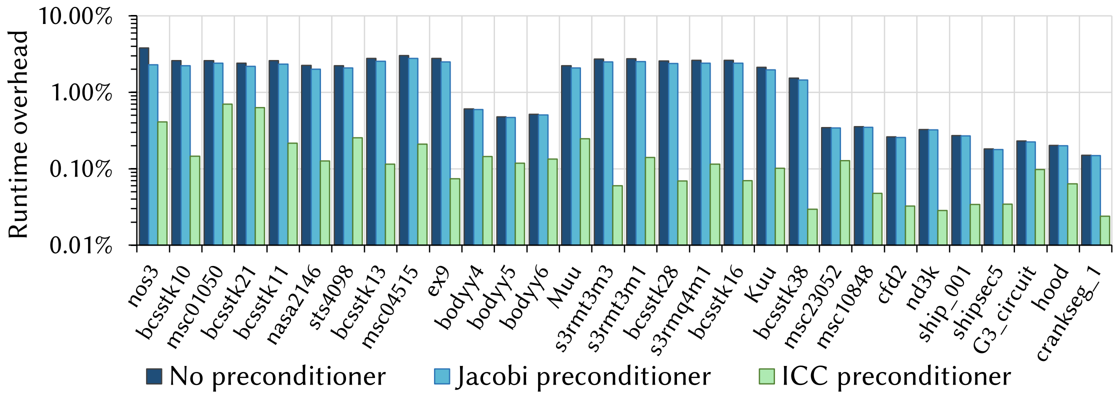
The observed vulnerability of the evaluated Conjugate Gradient solvers constitutes a strong demand for effective fault tolerance measures that induce only low runtime overhead to detect and correct errors. For the evaluated matrices, 47.2% of all solver executions resulted in either significantly increased runtimes or silent data corruptions in case of errors. Even if the solvers converged to a correct result, the iteration overhead to provide solutions was increased by up to 161.4%.

7.5.2 Runtime Overhead for Error Detection

The fault tolerance technique presented in Chapter 4 was applied to the evaluated Conjugate Gradient solvers to investigate the runtime overhead for error detection. In this evaluation, 3000 experiments were performed for each matrix to obtain the results shown below. In these experiments, the error checking interval t was set to ten iterations. Checkpoints were created in intervals of ten iterations. As different kinds of preconditioners can be chosen for Conjugate Gradient solvers, the runtime overhead was obtained for three different cases. While in the first case, no preconditioner was applied, the *Jacobi* preconditioner, and the incomplete Cholesky factorization (ICC) were applied in the other two cases. From the collected runtime information, the runtime overhead is computed as

$$\text{Runtime overhead} := \left(\frac{\text{Protected solver execution runtime}}{\text{Unprotected solver execution runtime}} - 1\right) \cdot 100\%.$$

Figure 7.7 shows the results of this investigation.



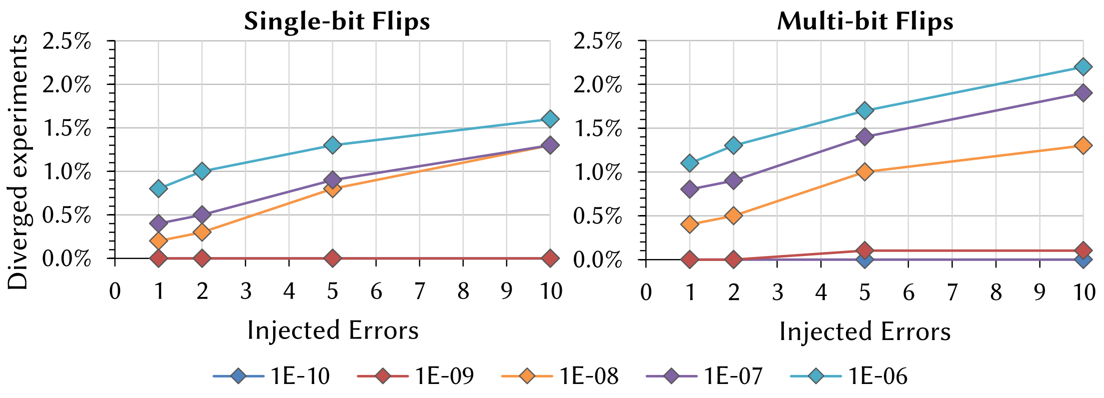
▲ **Figure 7.7** — Runtime overhead for error detection with respect to applying no preconditioner, the *Jacobi* preconditioner, and the *incomplete Cholesky factorization* (ICC) in error-free executions.

The runtime overhead for error detection which is introduced by the presented method differs for the three preconditioners since they introduce different operations to the original solver execution (cf. Appendix A.2). The runtime overhead for error detection is on average 1.7% and ranges from 0.15% to 3.8% when no preconditioner is applied. The average error detection overhead is 1.5% when the *Jacobi* preconditioner is applied and 0.2% when the *incomplete Cholesky factorization* preconditioner is applied. The runtime overhead of the presented fault tolerance method typically becomes smaller with increasing numbers of non-zero elements. For the three largest matrices, *G3_circuit*, *hood* and *crankseg_1*, the overhead of the presented method is only between 0.02% and 0.23%. Therefore, the fault tolerance technique scales very well with increasing problem sizes.

7.5.3 Error Coverage

To evaluate the effectiveness of the presented fault tolerance technique in the presence of errors, 3000 error injection experiments were performed for each matrix. These error injection experiments were performed in accordance to the above presented vulnerability assessment of the Conjugate Gradient solvers with respect to the selected error types and locations. In each experiment, a complete run of the solver was performed while between one and ten errors were injected in a randomly chosen iteration. In these iterations, a result register of a linear algebra operation was randomly selected to inject an error. The evaluation below shows the results for applying the *Jacobi*-preconditioner. Further results were obtained for the other two preconditioning cases, which do not show significant differences. These additional results are shown in Appendix E.3.

Figure 7.8 shows the *maximum portions of diverged experiments* for different numbers of injected errors and different rounding error thresholds $\{10^{-10}, 10^{-9}, 10^{-8}, 10^{-7}, 10^{-6}\}$.



▲ **Figure 7.8** — Maximum portion of execution failures (i.e. number of iterations exceeded iteration limit) with respect to $\tau \in [10^{-10}, 10^{-6}]$.

The range of rounding thresholds τ was determined individually for each matrix to avoid false positives in the error-free case. To find appropriate rounding thresholds τ for the different matrices, the experiments were started with the smallest rounding error threshold $\tau = 10^{-10}$. In case of a false positive, the rounding error threshold was increased, while the threshold value was omitted from the experiments.

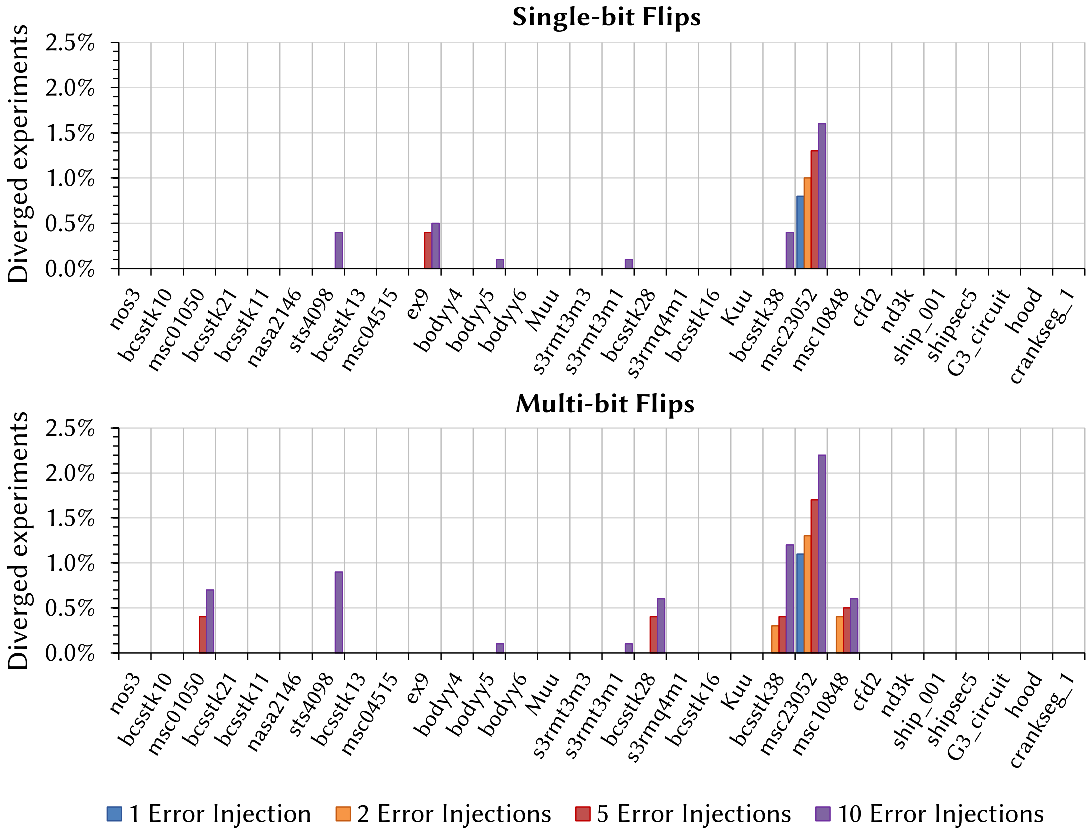
In the course of the experiments, no silent data corruption occurred. For increasing numbers of error injection events, the maximum portions of diverged experiments increases. In case of one error, at least 98.9% of all experiments converged successfully. When the number of error injections is increased to ten per experiment, then at least 97.8% of all experiments still converged to a correct result.

Larger invariant thresholds τ lead to an increasing number of undetected errors that can induce additional iterations to the solver executions. These additional iterations can cause the solver execution to exceed the iteration limit. Smaller invariant thresholds, however, can cause false positive error detections. The threshold invariants $\tau = 10^{-10}$ and $\tau = 10^{-9}$ induced false positive error detections for seven matrices. For the remaining threshold invariants $\tau \in [10^{-8}, 10^{-6}]$, the evaluated matrices did not cause false positive error detections.

Figure 7.9 shows the portion of experiments that exceeded the limit of iterations when the threshold invariant τ was set to 10^{-6}.

Diverged experiments were observed for eight out of 30 matrices. While for single-bit flip error injections, the portion of diverged experiments ranges from 0.1% to 1.6%, it ranges from 0.1% to 2.2% for multi-bit flip error injections.

Compared to the unprotected case, the presented fault tolerance technique significantly increases the number of solver executions that converge to a correct result. For 22 out of 30 matrices, all solver executions converged to a correct result within the iteration limit. For the remaining eight matrices, at least 97.8% converged to a correct result in the presence of up to ten error injection events. Therefore, the presented technique scales very well with increasing error rates.



▲ **Figure 7.9** — Portion of execution failures (i.e. number of iterations exceeded iteration limit) with respect to $\tau = 10^{-6}$.

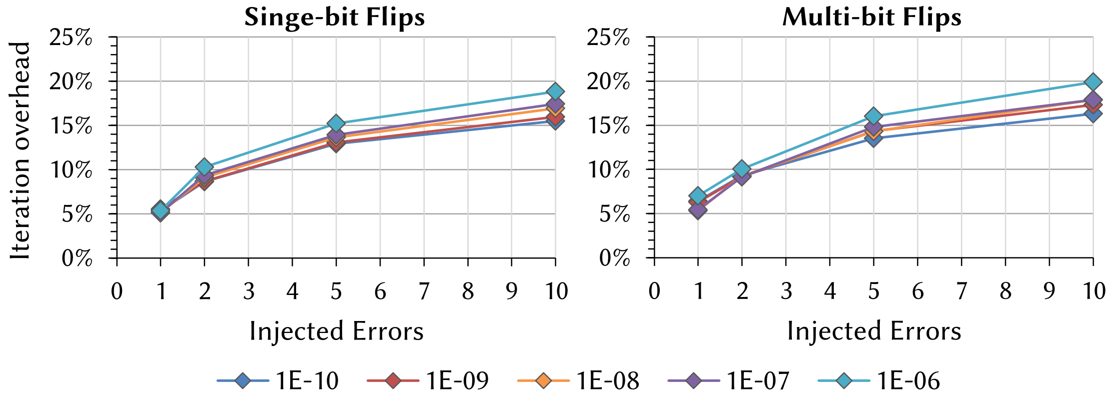
7.5.4 Error Correction Overhead

The overhead for error correction is introduced by additional iterations that are required for convergence to a correct result in case of errors, compared to the error-free execution. This *error correction overhead* is calculated with

$$\text{Err. correction overhead} := \left(\frac{\text{Protected solver iterations with errors}}{\text{Unprotected solver iterations without errors}} - 1 \right) \cdot 100\%.$$

Figure 7.10 shows the average *overhead for error correction* for different numbers of injected errors and different invariant thresholds τ.

For both single and multi-bit flip error injections, the average overhead for error correction increases with increasing numbers of error injection events. For single-bit flip error injections, the overhead ranges from 4.7% for one error injection to 18.82% in case of ten error injections. In case of one error injection event, the overhead ranges from 4.7% to 5.3% when the threshold invariant τ is increased from 10^{-10} to 10^{-6}. At



▲ **Figure 7.10** — Average iteration overhead for error correction with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

the same time, the overhead ranges from 15.5% to 18.8% when the threshold invariant τ is increased from 10^{-10} to 10^{-6} in the case of ten error injection events.

For multi-bit flip error injections, the overhead ranges from 5.3% for one error injection to 19.9% in case of ten error injections. When the threshold invariant τ is increased from 10^{-10} to 10^{-6}, the overhead ranges from 5.3% to 6.8% in case of one error injection event. For ten error injection events, the overhead ranges from 16.3% to 19.9%.

7.5.5 Discussion of Experimental Results

The experimental evaluation demonstrated that the presented fault tolerance technique provides both effective error detection and low runtime overhead. The average runtime overhead for error detection is 1.7% when no preconditioner is applied, 1.5% for the *Jacobi* preconditioner, and 0.2% for the incomplete Cholesky factorization preconditioner. The runtime overhead differs for the evaluated preconditioners since the preconditioners rely on different operations with diverse runtime. Overall, the observed runtime overhead is at most only 3.8% in the error-free case. Besides, the runtime overhead scales very well with increasing problem sizes as it typically decreases for larger problems.

Compared to unprotected solver executions, this fault tolerance technique increases the number of correct results significantly despite the presence of errors. In 22 out of 30 evaluated matrices, all experiments converged to a correct result. For the remaining eight matrices, at least 97.8% of the experiments provided a correct result. In case of one error injection per solver execution, the average runtime overhead to correct errors is

6.2% and remains below 15.7% in the worst case. While the runtime overhead to correct errors increases with increasing number of errors, it is at most 31.1% in case of ten error events.

Both the low runtime overhead to detect and correct errors as well as the high effectiveness make this fault tolerance technique highly suitable for applications in the scientific and engineering domain.

7.6 Parameter Evaluation and Estimation Methods

This section presents the experimental results for the parameter evaluation and estimation methods, which were introduced in Chapter 6. The methods are evaluated at the example of the approximate computing technique presented in Chapter 5, which executes the *Conjugate Gradient* solvers on approximate hardware (cf. Section 7.3) while adapting the precision according to the changing error resilience between solver iterations.

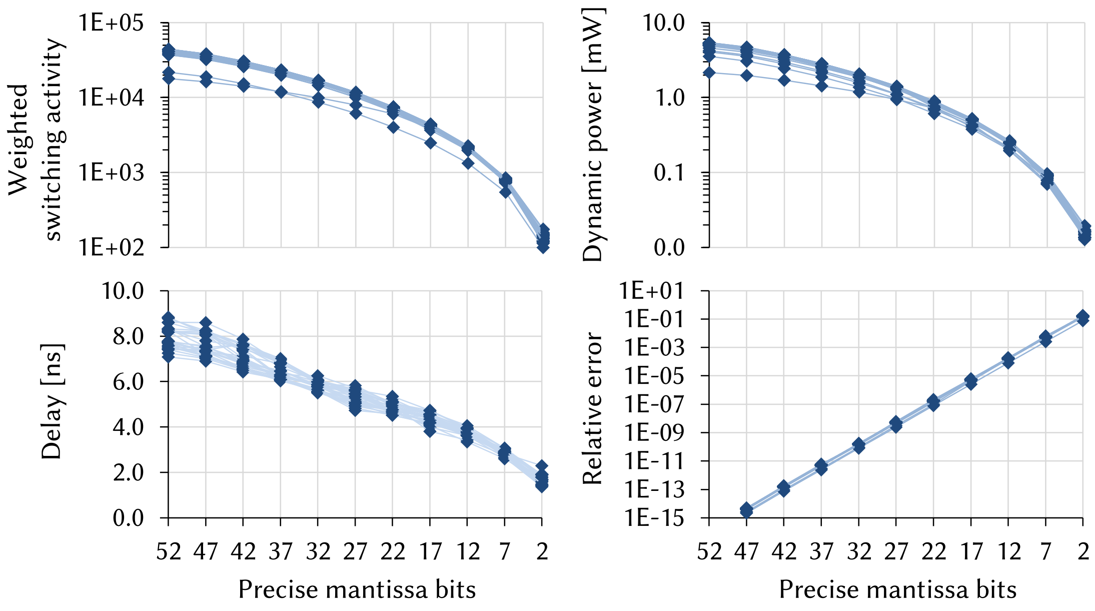
The underlying linear operations in the solver are explored by the *simulation-based* parameter evaluation method to determine parameters comprising area, delay, power, energy, and induced approximation error. The obtained power parameters have been validated against a commercial timing simulation and power estimation tool. The *combined parameter estimation* method has been evaluated by estimating energy and runtime for complete application executions. To validate this approach, the results have been compared to the results of exhaustive simulation-based parameter evaluations with respect to the *estimation error* and the *runtime reduction*.

All linear operations in the Conjugate Gradient solver are instrumented by replacing the underlying floating-point operations by a corresponding interface invocation. In accordance to the experimental evaluation in Section 7.7, the floating-point multiplications in sparse matrix-vector operations are replaced by their approximate counterparts. All benchmark matrices from Table 7.2 were evaluated for ten approximation levels that rely on different numbers of precise mantissa bits following the approximation model described in Chapter 6.4 (i.e. 2, 7, 17, 22, 27, 32, 37, 42, 47, and 52 precise mantissa bits). All timing simulations were performed on circuit descriptions for floating-point addition and multiplication as described in Section 7.3. In the experiments, all floating-point operations were performed in double-precision floating-point arithmetic for the solver execution.

Section 7.6.1 presents the experimental results for the simulation-based parameter evaluation method, in which the activity, the delay, the power dissipation, and the approximation error are compared for the different matrices. Section 7.6.2 presents the validation of the combined parameter estimation method, which combines the simulation-based and model-based methods to estimate the power dissipation for complete application executions.

7.6.1 Simulation-based Parameter Evaluation

The simulation-based parameter evaluation method is applied to determine the average *switching activity*, the average *dynamic power*, the maximum *delay*, and the average *relative error* to compare precise and approximate floating-point multiplications in sparse matrix-vector multiplications. Figure 7.11 compares these observables for the different matrices with respect to different numbers of precise mantissa bits. Each data point denotes an obtained observable for one specific matrix.



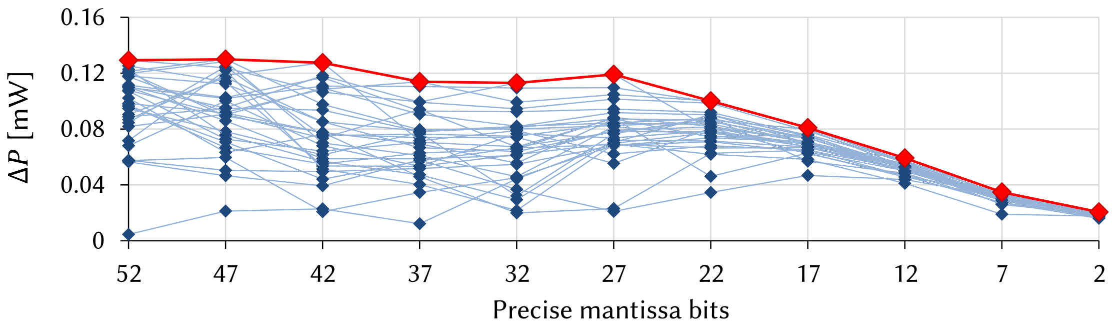
▲ **Figure 7.11** — Comparison of simulation-based evaluation results for the different matrices with respect to floating-point multiplication with different numbers of precise mantissa bits.

The weighted switching activity (WSA) decreases with decreasing number of precise mantissa bits. The WSA ranges from 100 to 44, 162, resulting in a dynamic power ranging from $0.013\,\mathrm{mW}$ to $5.4\,\mathrm{mW}$. The maximum delay corresponds to the time of the last observed transition in the circuit outputs and ranges from $1.4\,\mathrm{ns}$ to $8.8\,\mathrm{ns}$. The relative error increases with decreasing number of precise mantissa bits and ranges from $2.1 \cdot 10^{-15}$ to 0.17.

The simulation-based parameter evaluation method was validated using a commercial simulation and power estimation tool. The power dissipation results obtained by the commercial tool P_{ref} are compared to the results of the simulation-based method P by computing the difference

$$\Delta P := P - P_{\mathrm{ref}}\,. \tag{7.1}$$

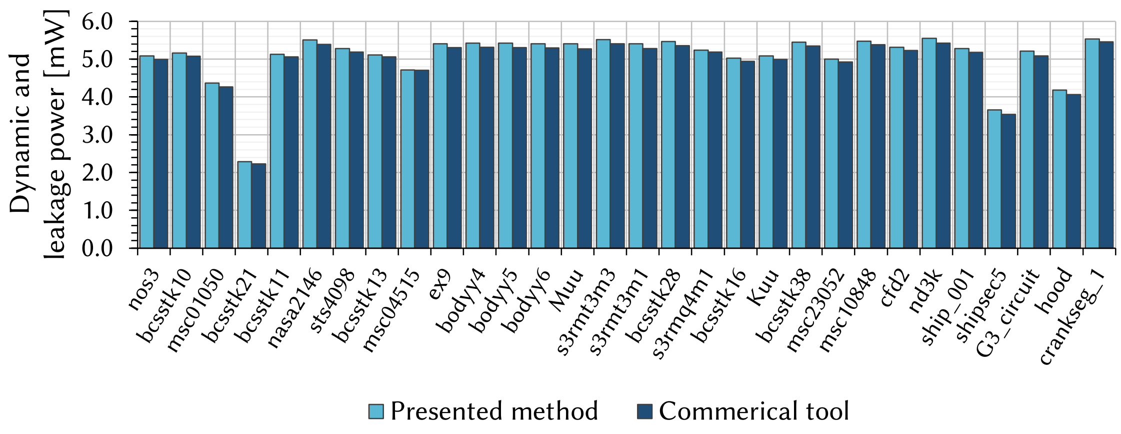
Figure 7.12 shows the results of this investigation. Each blue graph denotes a difference result ΔP for one specific matrix. The red graph denotes the maximum ΔP for a certain number of precise mantissa bits.



▲ **Figure 7.12** — Comparison of the simulation-based parameter evaluation method and the commercial tool for the different matrices with respect to the power dissipation.

Figure 7.13 compares the power results in case of 52 precise mantissa bits.

For all evaluated matrices, the deviation between the simulation-based parameter evaluation method and the commercial tool is on average 5.0% and ranges from 0.1% to 22.5%. This validation shows only small deviations between the results of simulation-based evaluations and the commercial tool, which allows to draw reliable conclusions on the evaluated observables. In all evaluated cases, the power results calculated by the presented method were larger than the results obtained by the commercial tool. Since these calculated power results are pessimistic, trustworthy conclusions on power and energy reductions can be provided using these results.



▲ **Figure 7.13** — Comparison of the simulation-based parameter evaluation method and the commercial tool with respect to power dissipation in case of 52 precise mantissa bits.

For executions of sparse matrix-vector multiplications using different numbers of precise mantissa bits, the *relative error* and *energy demand* have been determined. The relative error ε_{rel} is computed as

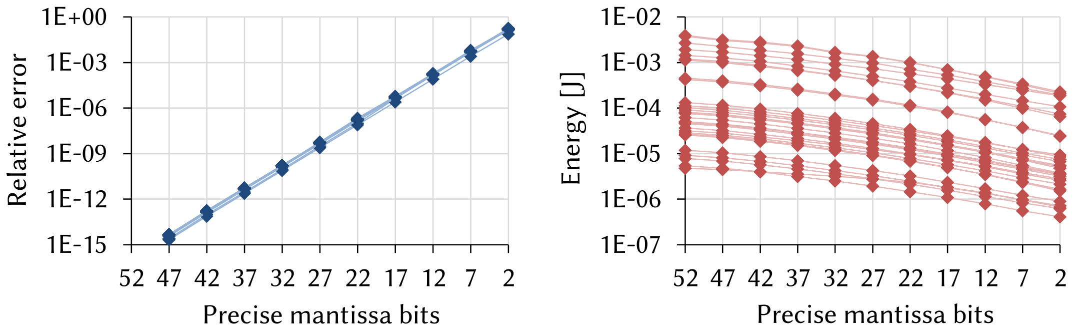
$$\varepsilon_{rel} := \frac{\|\boldsymbol{p} - \boldsymbol{a}\|_2}{\|\boldsymbol{p}\|_2} \tag{7.2}$$

with $\boldsymbol{a}$ being a result vector computed on approximate hardware and $\boldsymbol{p}$ being its precise counterpart. At the same time, the energy demand is computed from the n underlying floating-point operations in a sparse matrix-vector multiplication (i.e., floating-point multiplications and floating-point additions) as

$$\text{Energy} := \sum_{i=1}^{n} \text{Power}_i \cdot T_i \tag{7.3}$$

with Power_i denoting the power dissipation (i.e., leakage and dynamic power) and T_i denoting the critical path delay of the underlying circuit (i.e., 14.99 ns as discussed above in Section 7.3). Figure 7.14 compares the relative error and the energy demand for the different evaluated matrices.

The relative error increases with decreasing number of precise mantissa bits and ranges from $2.1{\cdot}10^{-15}$ to 0.17. The average energy decreases with decreasing number of precise mantissa bits and ranges from $3.9{\cdot}10^{-3}$ J to $4.1{\cdot}10^{-7}$ J. When the number of precise mantissa bits is reduced from 52 to 2, the energy demand is on average decreased by 92.8%.



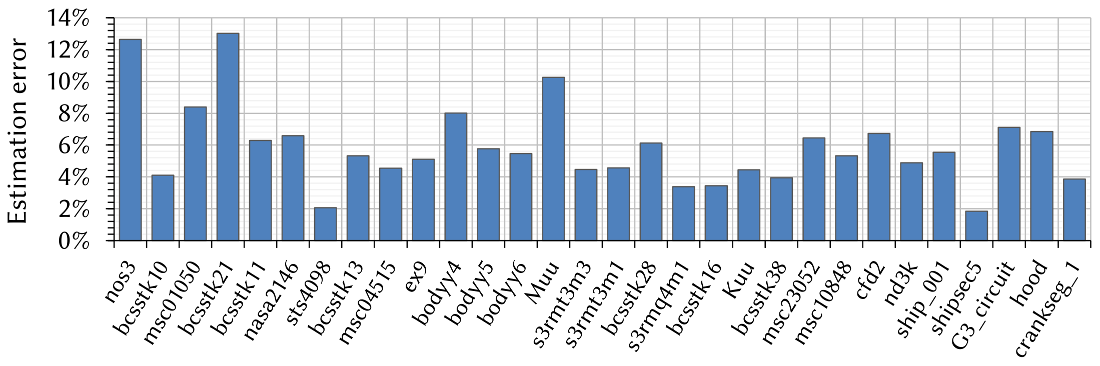
▲ **Figure 7.14** — Comparison of simulation-based parameter evaluations for sparse matrix-vector multiplications with different numbers of precise mantissa bits.

7.6.2 Combined Parameter Estimation

To evaluate the estimation-based parameter estimation method, this method has been applied to assess complete executions of the Conjugate Gradient solver. For each solver iteration k in the execution of the Conjugate Gradient solver, an instruction interval I_k has been formed that contains all linear operations with their underlying floating-point operations (i.e. floating-point additions and multiplications). In the execution of the solver, the evaluation of the instruction intervals has been switched between *simulation-based evaluations* and *model-based estimations* as follows: Simulation-based evaluations are performed for *representative instruction intervals* that comprise the first instruction interval I_0 and all instruction intervals I_k that follow an adaption of the underlying approximation error (i.e., the first iteration after an adaption). Such adaptions of the underlying precision-configurable approximation techniques can occur between solver iterations. The *remaining instruction intervals* are evaluated by the model-based estimation method that mimics the approximation error.

Using the power dissipation results obtained from the performed timing simulations, the combined parameter estimation method estimates the power dissipation of the remaining instruction intervals as follows: As explained in Chapter 6.5 in detail, power dissipation results obtained for an instruction interval are used as estimations for subsequent instruction intervals until the underlying approximation technique is adapted. Based on these power estimations, the method estimates the energy demand for whole solver executions using Equation 6.13. To validate this scheme, the energy estimations were compared to the results obtained from *exhaustive explorations*, in which all instruction intervals during the solver execution are evaluated using timing simulations. The iteration limit was set to 1000 iterations to maintain a feasible evaluation time (i.e. maximum 14 days per experiment).

Figure 7.15 shows the relative error between estimation-based and simulation-based investigations of the energy demand.



▲ **Figure 7.15** — Relative error between estimation-based and simulation-based investigations of the energy demand.

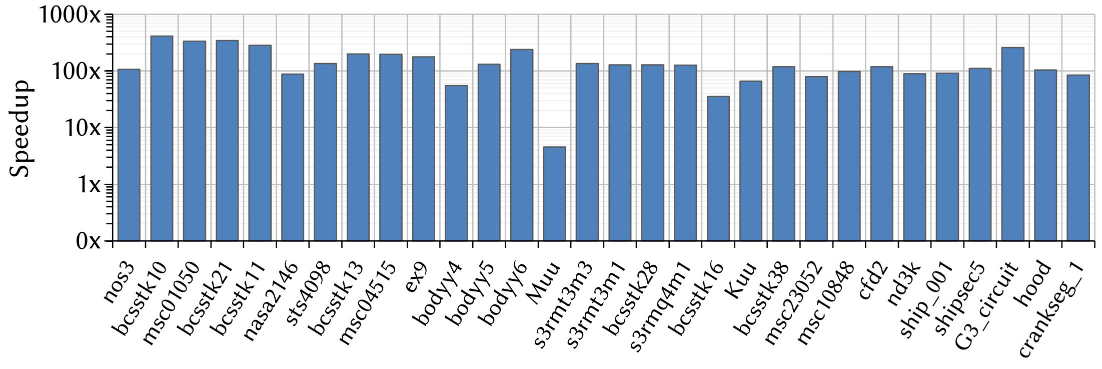
The error in the energy estimation is on average 5.8% and ranges from 1.8% to only 13.0% when the estimation-based and simulation-based methods are compared. For 18 matrices, the energy estimation error is less than 6.0%. In all evaluated cases, the energy estimation is larger than the energy determined by timing simulations. Such small estimation errors allow to draw reliable conclusions from the results obtained by the estimation-based method.

The runtime of the estimation-based method was compared to the runtime of exhaustive explorations by computing the *speedup* as

$$\text{Speedup} := \frac{\text{Runtime of simulation-based method}}{\text{Runtime of estimation-based method}}$$

The results of this evaluation are shown in Figure 7.16.

When compared to exhaustive explorations, the parameter investigation runtime is reduced for all matrices by the estimation-based method. The speedup is on average 149.3x and ranges from 4.5x (i.e. 157.9 s to 34.8 s) to 414.9x (i.e. 5707.4 s to 13.8 s). For 28 out of 30 matrices, the speedup is at least 54.7x. The speedup can be explained by the significantly reduced number of solver iterations that are evaluated by compute-intensive timing simulations. In the worst case, at most 10 solver iterations are mapped



▲ **Figure 7.16** — Speedup of the estimation-based parameter estimation method compared to exhaustive simulation-based parameter evaluations.

to timing simulations, since 10 approximation levels are available for each solver execution. With increasing number of iterations that are required for convergence to correct results, larger speedups are achieved. The minimum speedup of 4.5x is observed with Matrix *Muu* and can be explained by the low number of iterations (i.e. ≈ 17) that are required to solve linear equations for this matrix.

Both small estimation errors along with significant speedups show that the presented approach enables the efficient and effective exploration of parameters.

7.6.3 Discussion of Experimental Results

The experimental results above show that the presented parameter evaluation and estimation methods investigate application executions on approximate computing hardware with low parameter evaluation runtime and high accuracy. In the simulation-based evaluation of floating-point operations, the deviation between the observed power results and the results of a commercial tool is on average 5.0% and ranges from 0.1% to 22.5%. Such small deviations allow to draw reliable conclusions for the determined observables.

For complete application executions, the combined estimation method determines essential parameters like the power dissipation with significant speedups and high accuracy. In the evaluation of complete solver executions, the observed speedup is up to 414.9x while providing energy estimations that only deviate by less than 13.0% when compared to exhaustive explorations that map all underlying operations to timing simulations. On average, this parameter estimation method provides a speedup of 149.3x and an estimation error of 5.8%. The presented approach allows to draw trustworthy results, since all obtained energy estimations were pessimistic compared to the results obtained from exhaustive simulations.

7.7 Conjugate Gradient solvers on Approximate Computing Hardware

This section evaluates the technique that enables the execution of the Conjugate Gradient solvers using approximate hardware, which was presented in Chapter 5. This technique has been evaluated with respect to the influence of approximation errors on the *solver iterations* for correct convergence, the *reduction in energy* as well as the resulting *energy efficiency*. Besides, the utilization of the available precision degrees over the course of the solver progress have been analyzed.

The experimental parameters were set as follows: At runtime, the floating-point multiplications in the sparse matrix-vector multiplication were replaced by their approximate counterparts. For each matrix, 3000 experiments were performed. Each experiment comprised a single execution of the Conjugate Gradient solver with approximated matrix-vector multiplications. The rounding error threshold used in the comparison of floating-point values was set to 10^{-7}. The error resilience in the solver was estimated each 20 iterations and evaluated by the fault tolerance technique (cf. Chapter 5.2) each 10 iterations. For each matrix, the step function $H(\delta^{(k)})$ was calibrated once following the procedure discussed in Chapter 5.4. In each experiment, ten approximation levels were utilized that rely on specific numbers of precise mantissa bits $p \in \{2, 7, 17, 22, 27, 32, 37, 42, 47, 52\}$. All experiments converged to a correct solution.

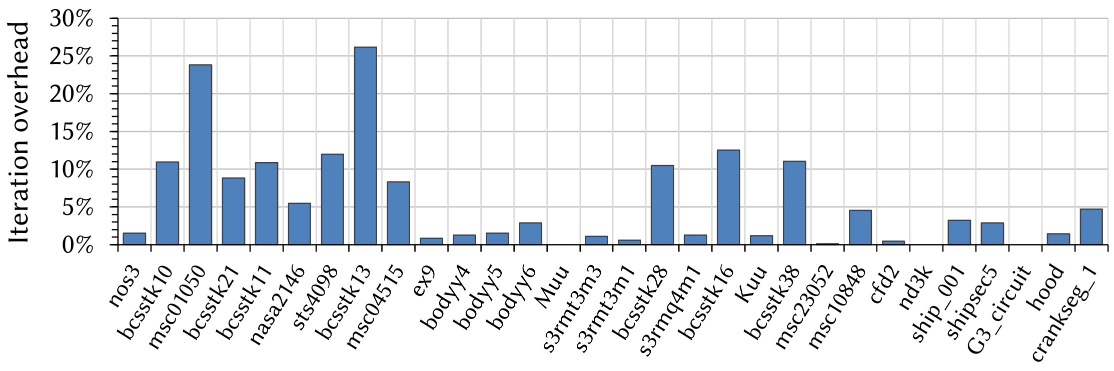
In the experiments, the influence of the utilized preconditioning technique has been evaluated. The experimental results shown below were obtained using the *Jacobi* preconditioner. Further results were obtained for the other two preconditioning cases, which are summarized in Appendix E.4.

7.7.1 Solver Iterations

The introduction of approximation errors into the iterative solver process may induce some additional iterations required for convergence to a correct result. From the evaluated experiments, the iteration overhead is computed as

$$\text{Iteration overhead} := \left(\frac{\text{Iterations for execution on approximate hardware}}{\text{Iterations for execution on precise hardware}} - 1\right) \cdot 100\%.$$

Figure 7.17 shows the additional iterations induced by approximation errors compared to the number of iterations that are required by the solver when executed on precise hardware.



▲ **Figure 7.17** — Average number of iterations on approximate hardware compared to the execution on precise hardware.

The evaluated matrices are ordered by the number of non-zero elements. The increase in the number of iterations is on average only 5.6% and ranges from 0% to 26.1%. For 25 matrices, the number of iterations is only increased by at most 11%. For the matrices *Muu*, *nd3k*, and *G3_circuit*, no iteration overhead was observed.

Low iteration overheads are favorable as each additional iteration demands some additional energy, which can cancel the energy efficiency gain achieved by approximate hardware in the worst case. The iteration overhead that is associated with the presented technique is often very low.

7.7.2 Energy

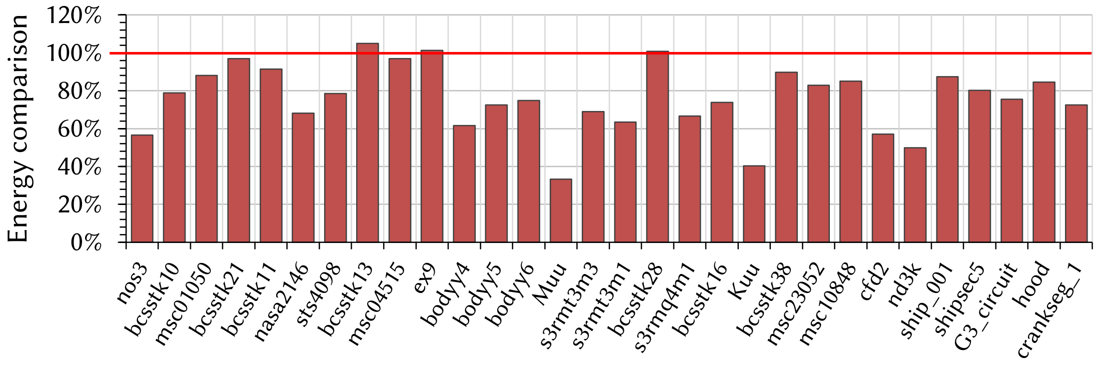
To evaluate the energy demand, the operations in the Conjugate Gradient solver were evaluated by the combined parameter estimation scheme presented in Chapter 6.5 that relies on both gate-level timing simulations and model-based evaluations of the approximation. This method evaluated at least one complete solver iteration for each approximation level using timing simulations to determine the power dissipation.

Following Equations 6.3 and Equation 6.5, the power dissipation is obtained from the activity information provided by the timing simulation and the energy information given in the standard cell library. Using these results, the energy for complete solver executions is estimated using Equation 6.13. In the remaining solver iterations, the software-based approximation model was applied to introduce approximation errors. Both the software-based model and the timing simulations evaluated the underlying floating-point operations in double-precision arithmetic.

The obtained results for energy are compared to the energy that is required to execute the solver on precise hardware without approximation errors as follows:

$$C_{Energy} := \left(\frac{\text{Energy for execution on approximate hardware}}{\text{Energy for execution on precise hardware}} \right) \cdot 100\%. \tag{7.4}$$

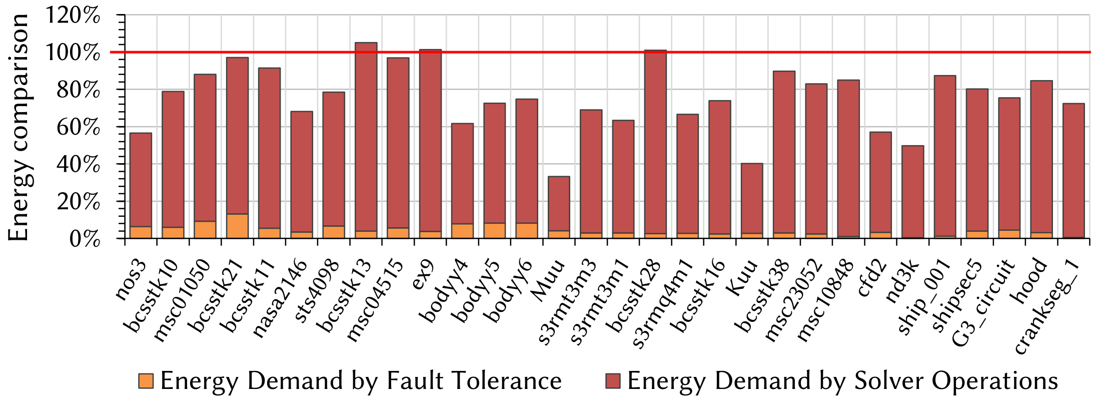
Figure 7.18 shows the demanded energy for executing the solver on approximate hardware compared to the execution on precise hardware.



▲ **Figure 7.18** — Estimated energy demand to execute the solver on approximate hardware compared to the execution on precise hardware.

The case C_{Energy} = 100% denotes the scenario in which the energy demand for executing the solver is equal on approximate hardware and precise hardware. Lower values in C_{Energy} denote reductions in energy to execute the solver while still obtaining correct results. Such a reduction in energy can be observed for 27 matrices compared to executing the solver on precise hardware. For these matrices, the energy demand is on average reduced by 26.9% and in total up to 66.7%. For 14 matrices, the energy demand is reduced by at least 25%.

Figure 7.19 shows the contribution of the underlying fault tolerance technique within the energy demand of solver executions on approximate hardware. This energy demand is compared to the energy demand of solver executions on precise hardware.



▲ **Figure 7.19** — Contribution of the underlying fault tolerance technique within the energy demand for solver executions on approximate hardware.

The energy demand of the fault tolerance technique is on average 6.0% and ranges from 0.7% to 13.6% compared to the total energy demand of solver executions on approximate hardware. This low energy overhead is favorable to execute the Conjugate Gradient solver using approximate hardware with significantly reduced energy demand.

7.7.3 Energy Efficiency

The energy efficiency is evaluated using the *Watt-per-MIPS*^{2} metric that is based on the energy-time product metric (cf. Equation 2.45).

As discussed in Section 7.6.1, the power dissipation changes by adapting the approximation level (i.e. adapting the number of precise mantissa bits). As the power dissipation can change between solver iterations, the *average power dissipation* $\text{Power}_{\text{AVG}}$ in a complete solver execution is computed from the power dissipation in single solver iterations k with

$$\text{Power}_{\text{AVG}} := \sum_{k=1}^{n} \frac{\text{Average power dissipation in iteration } k}{n} \tag{7.5}$$

with n denoting the total number of iterations required for correct convergence.

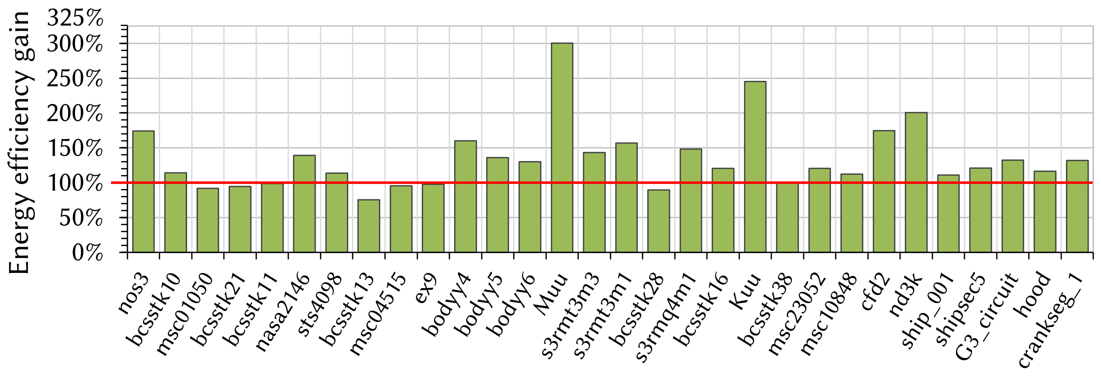
Let T_{solver} denote the solver execution runtime. With T_{solver} = instruction count/MIPS, the energy-time product metric is

$$\text{Energy-time product} = \text{Power}_{\text{AVG}} \cdot \left(\frac{T_{\text{solver}}}{\text{instruction count}} \right)^2 = \frac{\text{Watt}}{\text{MIPS}^2} \, . \tag{7.6}$$

From the obtained power dissipation results, the gain in energy efficiency G_η is computed and compared for executions on approximate and precise hardware as follows:

$$G_\eta := \left(\frac{\text{Watt-per-MIPS}^2 \text{ for execution on precise hardware}}{\text{Watt-per-MIPS}^2 \text{ for execution on approximate hardware}} \right) \cdot 100\% \,. \tag{7.7}$$

The results of this evaluation are shown in Figure 7.20.

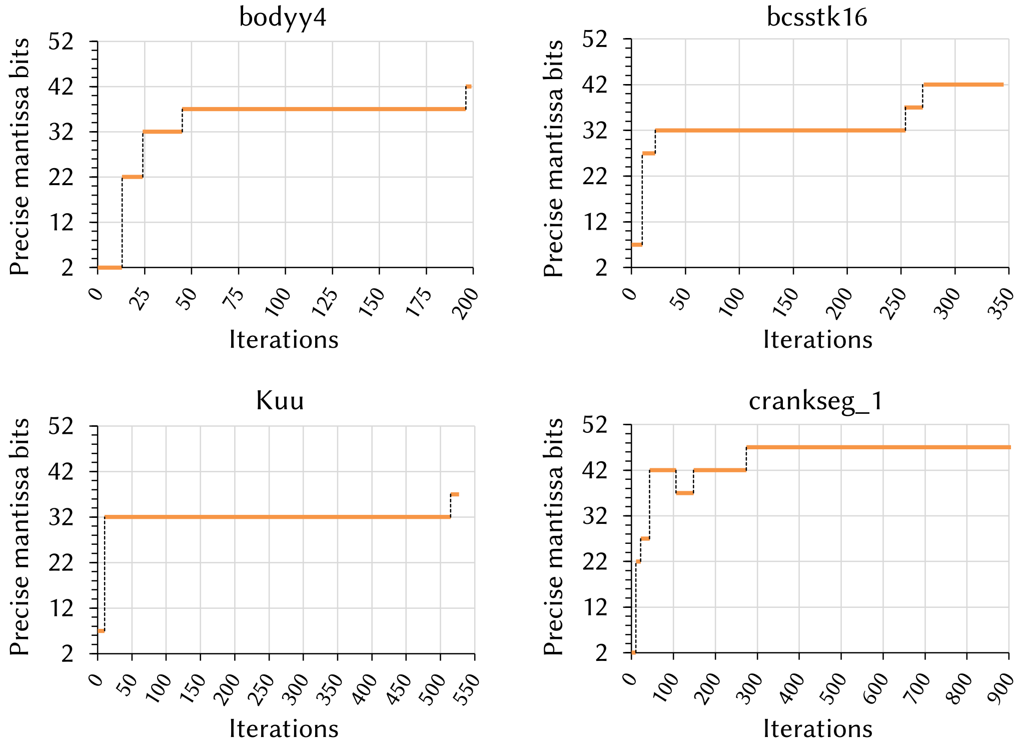


▲ **Figure 7.20** — Gain in energy efficiency for solver executions on approximate hardware compared to executions on precise hardware.

In this evaluation, the case $G_\eta = 100\%$ denotes the scenario in which executing the solver on approximate hardware and precise hardware lead to equal energy efficiency. Values in G_η below 100% refer to decreased energy efficiency when the solver is executed on approximate hardware. At the same time, values in G_η larger than 100% denote improved energy efficiency. The energy efficiency is improved for 23 matrices compared to executing the solver on precise hardware. In case of these matrices, the largest energy efficiency gain is 300.4%, while the average energy efficiency gain is 148.0%. For 14 out of 30 matrices, the energy efficiency is at least increased by 25%.

7.7.4 Utilization of Approximation Levels

Over the course of the solver iterations, the induced approximation error was adapted according to the changing error resilience. Figure 7.21 shows the adaption of approximation levels for single solver executions at the example of matrices *bodyy4*, *bcsstk16*, *Kuu*, and *crankseg_1*. The utilized approximation level is denoted with respect to the number of precise mantissa bits.



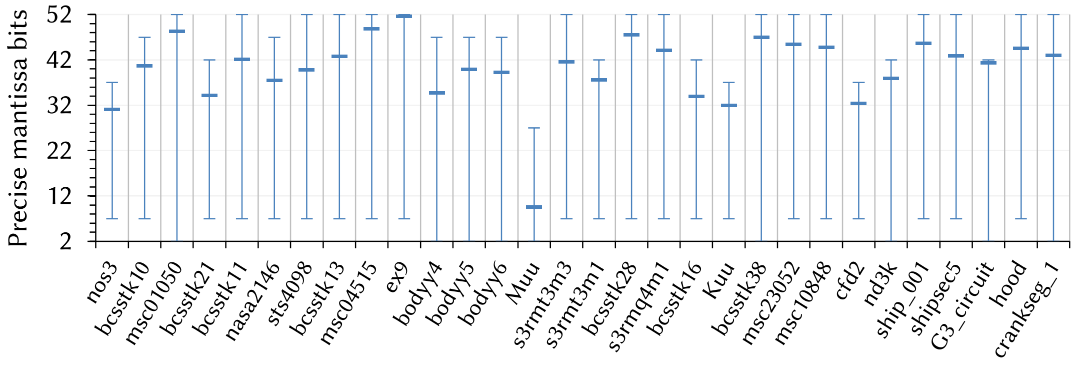
▲ **Figure 7.21** — Adaption of approximation levels in the course of the solver executions.

The available approximation levels were utilized to different extents during the evaluated solver executions. To gain an insight into the approximation level utilization for all matrices, the *average number of precise mantissa bits* p_{AVG} was computed as

$$p_{\text{AVG}} := \sum_{i=1}^{n} \frac{p_i \cdot (\text{Number of iterations spent with } p_i \text{ precise mantissa bits})}{\text{Total number of iterations}} \tag{7.8}$$

with $p_i \in \{2, 7, 17, 22, 27, 32, 37, 42, 47, 52\}$ denoting the number of precise mantissa bits. Figure 7.22 shows the results of this investigation. The whiskers denote the minimum and maximum number of precise mantissa bits that were utilized in the solver executions.

In the evaluation, 16 out of 30 matrices required full precision (i.e. 52 precisely computed mantissa bits) to convergence to correct results. At the same time, 5 matrices required only 47 precise mantissa bits to provide correct results. The minimum number of precise mantissa bits was required by matrix *Muu*, which only required 27 precise mantissa bits. The average number of precise mantissa bits p_{AVG} that was utilized during solver executions ranges from 9.5 to 51.7. Besides, 9 out of 30 matrices used 2 precise mantissa bits in a portion of the solver execution.

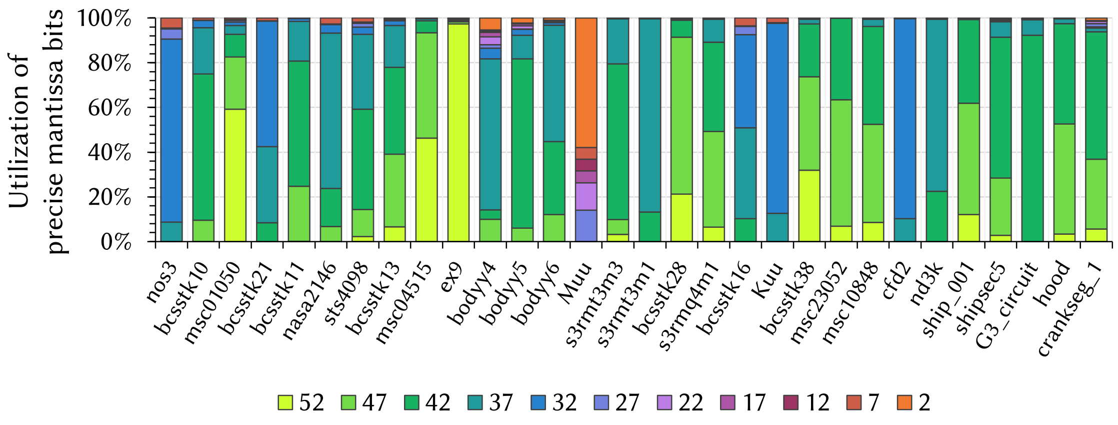


▲ **Figure 7.22** — Minimum, maximum and average precision over the course of the solver executions.

The utilization of the available approximation levels is evaluated by counting the number of iterations that used a specific approximation level. The utilization $U(p_i)$ of an approximation level that relies on p_i precise mantissa bits is computed as:

$$U(p_i) := \left(\frac{\text{Number of iterations spent with } p_i \text{ precise mantissa bits}}{\text{Total number of iterations}} \right) \cdot 100\%.$$

Figure 7.23 shows the results of this evaluation with respect to the number of precise mantissa bits.



▲ **Figure 7.23** — Average utilization of available precisions over the course of the solver execution.

The maximum approximation level (i.e. 2 precise mantissa bits) was utilized in 2.2% of the iterations. The most heavily used approximation level relies on 42 precise mantissa bits and was applied on average in 30.7% of all evaluated solver iterations.

The different matrices use the available precision levels to different extents. While matrices *nos3, nasa2146, ex9, bodyy5,* and *G3_circuit*, for instance, rely on a single precision for the majority of solver iterations, matrices like *bcsstk13, s3rmq4m1,* and *bcsstk16* rely on multiple different precisions over the course of the solver iterations.

7.7.5 Discussion of Experimental Results

The experimental evaluation showed that the adaptive method presented in Chapter 5 enables the execution of the Conjugate Gradient solvers using approximate hardware and, at the same time, often achieves increased energy efficiency while still ensuring correct solver results. When compared to executions on precise hardware (i.e. without approximation errors), the energy efficiency is increased for 23 out of 30 matrices. For the evaluated matrices, the energy efficiency was improved by up to 200.4%. On average, the energy efficiency is increased by 48.0%.

The introduction of approximation errors into the solver execution increased the solver runtime by at most 26.1% compared to executions on precise hardware. On average, the runtime is only increased by 5.6%. This low runtime overhead can be explained by the adaption of the induced approximation error according to the changing error resilience at runtime. Although the runtime of the solver execution is increased, the energy is reduced by up to 66.7%. The average reduction in energy is 26.9%.

The increased energy efficiency comes at the cost of a few additional solver iterations and low energy overhead by the underlying fault tolerance technique. This fault tolerance technique is highly suitable to monitor the solver execution on approximate hardware since its energy demand is on average 29.1x lower than the energy demand of the solver execution. At the same time, the demand for trustworthy results is still satisfied since this technique ensures correct results despite the introduction of approximation errors.

Chapter 8

Conclusion

The high computational power of heterogeneous computer architectures plays an essential role in accelerating complex tasks in scientific computing and simulation technology. The sustained demand for short execution times is satisfied by mapping algorithmic parts to matching components in these computer architectures, which can result in reduced runtimes.

Continuous improvements in computer architecture and semiconductor technology scaling have largely driven the increase in computational performance over the past decades. Continued technology scaling, however, increasingly imposes serious threats to the reliability and efficiency of upcoming semiconductor devices. Modern nano-scaled semiconductor devices become increasingly vulnerable to a growing spectrum of different reliability threats which can cause crashes or erroneous application results without indication. However, the explanatory and predictive power of computed results increasingly supports decision-making processes, which demands high reliability to obtain trustworthy results. Future manufacturing processes will allow even smaller chip feature sizes, which makes the integration of fault tolerance techniques mandatory.

Fault tolerance techniques can be applied to different system layers and typically rely on different forms of redundancy to ensure the correct service in a system. Software-based fault tolerance techniques protect algorithmic tasks in applications, for instance, by targeting faults that manifest themselves as errors at the software and application layer. Different techniques like algorithm-based fault tolerance (ABFT) add appropriate operations to detect and correct errors at runtime. A central challenge in integrating such software-based measures lies in the runtime overhead that is induced by such additional operations.

With the end of Dennard scaling, a power density problem emerged that can cause unacceptable chip power dissipation and thermal issues unless central scaling parameters are fixed between technology generations. The approximate computing paradigm allows to trade-off precision for efficiency gains with respect to power, energy, execution times, computational performance, and chip area. Different concepts have been proposed for *heterogeneous and approximate computer architectures* that combine approximate memories and processing elements with their precise counterparts. While approximate computing is a promising solution to tackle upcoming energy challenges by exploiting the error resilience in applications, scientific applications, however, often induce strict accuracy demands and offer rather low error tolerance. Such strict accuracy demands require careful utilization of approximation techniques.

This thesis has introduced *fault tolerance* and *approximate computing methods* that enable the reliable and efficient execution of *linear algebra operations* and *Conjugate Gradient solvers* using heterogeneous and approximate computer architectures. A *fault tolerance technique for sparse matrix-vector multiplications* has been presented that detects and implicitly locates errors in erroneous operation results with low runtime overhead and high error coverage. This fault tolerance technique exploits the insight that even high error rates typically do not cause errors in complete matrix operation results, but only in small parts. To avoid false-positive error detections in the presence of rounding errors, a rounding error bound has been presented that distinguishes harmful errors from inevitable rounding errors that occur in floating-point arithmetic.

A *fault-tolerant Conjugate Gradient solver* has been introduced that exploits that arbitrary successive iterations in the Conjugate Gradient solvers are related to each other by different inherent relations. To ensure the convergence of these solvers to correct results, the fault tolerance technique detects and corrections errors by evaluating these relations with very low runtime overhead. As the underlying assumptions are independent of the utilized preconditioning operation, the presented technique can protect both, the CG and the PCG solver.

An *adaptive method that enables the Conjugate Gradient solvers on approximate computing hardware* ensures convergence to correct results with reduced energy demand. This method instruments the *fault-tolerant Conjugate Gradient solver* to monitor the changing error resilience at runtime and to adapt the induced approximation error accordingly.

Parameter evaluation and estimation methods have been introduced, which determine the achieved computational efficiency for application executions on approximate hardware with respect to the induced approximation error as well as delay, switching activity, power, and energy. An underlying combined parameter estimation method enables fast and accurate investigations by combining highly accurate gate-level timing simulations with light-weight software-based models to estimate different parameters for long-running application executions.

The presented fault tolerance techniques were evaluated on a heterogeneous computing system with respect to the runtime overhead to detect and correct errors, as well as the error coverage. When the presented *fault-tolerant sparse matrix-vector multiplication* is compared to the traditional ABFT approach, the runtime overhead to detect and correct errors is on average reduced by 63.9%. At the same time, the error coverage is improved by up to 155%. Compared to unprotected executions, the *fault-tolerant Conjugate Gradient solver* induces an average error detection runtime overhead of 1.5%. For the different evaluated benchmark matrices, at least 97.8% of the experiments provided a correct result in case of errors. For 22 out of 30 evaluated matrices, all experiments converged to a correct result. The runtime overhead induced by both presented fault tolerance techniques scales with increasing problem size. The experimental evaluation also showed that the Conjugate Gradient solvers can be executed using approximate hardware by dynamically adjusting the approximation error according to the changing error resilience. When compared to executions on precise hardware, the energy efficiency is increased for 23 out of 30 evaluated matrices. While the average increase in energy efficiency is 48.0%, the maximum increase is 200.4%. The increased energy efficiency comes at the cost of a few additional solver iterations. Experimental results for the parameter evaluation and estimation methods have shown that the underlying approach estimates parameters like the energy for complete application executions with significantly reduced runtime and high accuracy. For the evaluation of complete solver executions, this approach allowed speedups of up to 414.9x and provided energy estimations that only deviate by less than 13.0%.

APPENDIX A

Linear Solvers and Preconditioners

A.1 The Conjugate Gradient Solver

The description of the Conjugate Gradient solver in Algorithm 2 follows [Saad03, pp. 199-200].

```
**Input**: $\boldsymbol{A}$,$\boldsymbol{b}$, $\boldsymbol{x}^{(0)}$, $\epsilon_a$, $\epsilon_r$, $k_{max}$
**Output**: The result of solving the system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$: $\boldsymbol{x}^{(k+1)}$
**Data**: $\boldsymbol{p}^{(k)}$, $\boldsymbol{r}^{(k)}$, $\delta^{(k)}$
/* Preparation of CG                                          */
1 $\boldsymbol{r}^{(0)} \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^{(0)}$ ;            // Initial residual vector
2 $\boldsymbol{p}^{(0)} \leftarrow \boldsymbol{r}^{(0)}$ ;                  // Initial search direction
3 $\delta_0 \leftarrow \boldsymbol{r}^{(0)T}\boldsymbol{r}^{(0)}$;                         // Initial residual
4 $k \leftarrow 0$
/* (Continued on the next page)                                 */
```

```
    /* CG loop                                                    */
5 while $(\delta^{(k)} > \epsilon_a^2) \wedge (\delta^{(k)}/\delta^{(0)} > \epsilon_r^2) \wedge (k < k_{max})$ do
6     $\boldsymbol{w}^{(k)} \leftarrow \boldsymbol{A}\boldsymbol{p}^{(k)}$;
7     $\alpha \leftarrow \frac{\delta^{(k)}}{\boldsymbol{p}^{(k)T}\boldsymbol{w}^{(k)}}$;
8     $\boldsymbol{x}^{(k+1)} \leftarrow \boldsymbol{x}^{(k)} + \alpha\boldsymbol{p}^{(k)}$;        // Next intermediate result
9     $\boldsymbol{r}^{(k+1)} \leftarrow \boldsymbol{r}^{(k)} - \alpha\boldsymbol{w}^{(k)}$;          // Update residual vector
10    $\delta^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)T}\boldsymbol{r}^{(k+1)}$
11    $\beta \leftarrow \frac{\delta^{(k+1)}}{\delta^{(k)}}$;
12    $\boldsymbol{p}^{(k+1)} \leftarrow \boldsymbol{r}^{(k+1)} + \beta\boldsymbol{p}^{(k+1)}$;          // New search direction
13    $k \leftarrow k + 1$;
14 end
```

Algorithm 2: The Conjugate Gradient Solver algorithm.

A.2 Preconditioners

The description of the *Jacobi* preconditioner in Equation A.1 and the description of the *Incomplete Cholesky Factorization* preconditioner in Equation A.2 follow [Golub13, chp. 11.5].

Let $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ be a positive-definite matrix:

The *Jacobi preconditioner* matrix $\boldsymbol{M}_{\text{Jacobi}}$ is

$$\boldsymbol{M}_{\text{Jacobi}} := \mathbf{diag}(A) \tag{A.1}$$

with $\mathbf{diag}(A)$ being the diagonal matrix of matrix $\boldsymbol{A}$.

The preconditioner matrix $\boldsymbol{M}_{\text{ICC}}$ used by the *Incomplete Cholesky Factorization* preconditioner is

$$\boldsymbol{M}_{\text{ICC}} := \boldsymbol{H}\boldsymbol{H}^T \tag{A.2}$$

so that for all nonzero a_{ij}, $[\boldsymbol{H}\boldsymbol{H}^T]_{ij} = a_{ij}$. At the same time, $\boldsymbol{H}$ is a sparse lower triangular matrix so that if

$$\boldsymbol{R} := \boldsymbol{H}\boldsymbol{H}^T - \boldsymbol{A} \tag{A.3}$$

then $a_{ij} \neq 0 \Rightarrow r_{ij} = 0$.

Appendix B

Dependability Attributes

The following definitions follow the taxonomy of [Pradh96] and [Koren07].

Definition B.1 (Lifetime of a system) *Given a system that provides its specified correct service at time $t = 0$, the lifetime T of a system denotes the time until the system fails (i.e. permanent system failure - the delivered service by the system deviates from the correct service permanently).*

The lifetime T is a random variable.

Definition B.2 (Failure probability function) *The failure probability function F(t) is the probability that a system will fail at or before time t with*

$$F(t) := Prob\{T \leq t\}\,. \tag{B.1}$$

Definition B.3 (Probability density function of lifetime T) *With T being the lifetime of a system, f(t) denotes the probability density function with*

$$f(t) := \frac{\mathrm{d}F(t)}{\mathrm{d}t}. \tag{B.2}$$

Being a probability density function, $f(t)$ must satisfy $\forall t \geq 0 : f(t) \geq 0$ and $\int_0^\infty f(t)\mathrm{d}t = 1$. Both functions are related through

$$F(t) = \int_0^t f(s)\mathrm{d}s. \tag{B.3}$$

Definition B.4 (Reliability) *The reliability R(t) of a system is the probability that a system provides its specified correct service at least for the time period t with*

$$R(t) := Prob\{T > t\} = 1 - F(t). \tag{B.4}$$

Definition B.5 (Mean time to failure) *The mean time to failure (MTTF) denotes the average time in which a system provides its specified correct service until a failure occurs. This measure is the expected value of the lifetime E[T] with*

$$MTTF := E[T] = \int_0^\infty t \cdot f(t)\mathrm{d}t = \int_0^\infty R(t)\mathrm{d}t. \tag{B.5}$$

Definition B.6 (Mean time between failures) *For a repairable system, the mean time between failures (MTBF) denotes the average time between failures of a system. With the time to repair the system (i.e. detect and isolate faulty component, replace component, verify successful fault removal) being denoted by the mean time to repair (MTTR), the MTBF is*

$$MTBF := MTTF + MTTR. \tag{B.6}$$

Definition B.7 (Failure rate) *The failure rate λ is the number of failing systems per time unit t compared to the number of surviving systems N.*

$$\lambda(t) = \frac{f(t) \cdot N}{(1 - F(t)) \cdot N} \tag{B.7}$$

Failure rates are expressed using the *FIT rate* which corresponds to the number of failures that can be expected in 10^9 operation hours.

Appendix C

Floating-point Arithmetic

This appendix chapter gives an overview of floating-point arithmetic and discusses the sources of rounding errors that can occur in finite-precision formats. Besides, this appendix chapter briefly discusses the basic formats and concepts that are defined in the IEEE Standard 754™-2008 [IEEE 08].

C.1 Floating-point Numbers

Scientific and engineering computing among other disciplines heavily relies on *real numbers* in their underlying computer-based modeling and simulation techniques. Today, the most widely used approach to represent real numbers is constituted by floating-point arithmetic that follows the IEEE Standard 754™-2008 [IEEE 08]. The discussion of floating-point arithmetic below follows [Mulle10] and [Golub13, chp. 2.7.2].

A floating-point number x is determined by the combination of a *sign* s, a *radix* β, a *normalized significand* m and an *exponent* e with

$$x = (-1) \cdot \beta^e \cdot m \, . \tag{C.1}$$

A set of parameters determines the precision and the value range of the floating-point representation:

- The sign $s \in \{0, 1\}$ corresponds to the sign of x.
- The radix (or base) β is an integer value and $\beta \geq 2$.
- The precision p determines the number of digits in the significand and $p \geq 2$.
- Two integer values e_{min} and e_{max} are used to delimit the value range of the exponent with $e_{min} < 0 < e_{max}$.
- The exponent e is an integer with $e_{min} \leq e \leq e_{max}$.
- The normal significand m is a number $0 \leq |m| < 2$.

Normalization ensures a unique representation for each floating-point number as it inherently selects representations for which the exponent is minimal. In case $1 \leq |m| < 2$, the underlying floating-point number x is a *normal number.* For radix $\beta = 2$, normalization allows to save one bit, which can be implicitly stored and, at the same time, allows to increase the precision p for the significand m by one bit.

In case of $e = e_{min}$ and $|m| < 1$, the underlying floating-point number x does not satisfy the normalization condition $1 \leq |m| < 2$. These numbers are called *denormal numbers* (in the literature, the term *subnormal numbers* is also used). Denormal numbers fill the gap between the smallest representable, normal floating-point number and zero. By filling this gap, denormal numbers allow so-called *gradual underflow*, which causes a *slow loss* of precision instead of an abrupt loss.

C.2 Rounding and Rounding Errors

Generally, floating-point numbers comprise a limited number of digits that allows to represent a finite set of rational numbers. To represent a real number in a certain floating-point format, the number has to be rounded to a suitable adjacent floating-point number. Besides, rounding is also often necessary to represent the result of floating-point operations in the underlying floating-point format. The difference that is introduced by a rounding operation is the so-called *rounding error.*

Different *rounding modes* can be defined that determine how a number is rounded to a finite floating-point number using a *rounding function.* For instance, the four rounding modes that appear in IEEE Standard 754™-2008 are:

- The *round-towards-minus-infinity* rounding mode maps a number x to $RD(x)$. The function $RD(x)$ computes the largest floating-point number less than or equal to x, possibly $-\infty$.
- The *round-towards-plus-infinity* rounding mode maps a number x to $RU(x)$. The function $RU(x)$ computes the smallest floating-point number greater than or equal to x, possibly $+\infty$.
- The *round-towards-zero* rounding mode maps a number x to $RZ(x)$. The function $RZ(x)$ computes the adjacent floating-point number to x that is not greater in magnitude than x.
- The *round-to-nearest* rounding mode maps a number x to $RN(x)$. The function $RN(x)$ computes the closest floating-point number to x. In the case that x is located exactly halfway between two adjacent floating-point numbers, a so called tie-breaking rule is required. A widely-used tie-breaking rule is *round-to-nearest-even* which maps x to the floating-point number with even significand m. Important properties of an appropriate tie-breaking rule are sign symmetry $RN(-x) = -RN(x)$, lack of statistical bias, and reproducibility.

Different properties that are satsified by arithmetic on real numbers do not apply in floating-point arithmetic. In general, floating-point additions and multiplications are neither associative (i.e. $a + (b + c) \neq (a + b) + c$) nor distributive (i.e. $a \cdot (b + c) \neq a \cdot b + b \cdot c$).

C.3 IEEE Standard for Floating-Point Arithmetic

The IEEE Standard 754™-2008 [IEEE 08] defines binary and decimal floating-point number formats and provides methods for floating-point arithmetic in *single, double, extended,* and *extendable precision*, while it recommends formats for data interchange. The computation methods described in this standard ensure identical results of floating-point operations independent of the underlying implementation, which can be performed in hardware, software, or both. The standard defines computation methods for addition, subtraction, multiplication, division, fused multiply-add, square root, compare, and other operations. At the same time, the standard specifies conversion algorithms between integer and floating-format as well as between different floating-point formats. Besides specifications of formats, the standard defines exception conditions such as non-representable numbers (i.e. *Not-a-Number, NaN*), and provides handling solutions for these conditions.

The standard specifies floating-point number formats using four parameters, namely a radix $\beta \in \{2, 10\}$, the precision p, as well as e_{max} and e_{min} which are constrained by $e_{min} = 1 - e_{max}$. It also specifies, that the following floating-point numbers are representable by any floating-point format:

- Signed zero and non-zero floating-point numbers following Equation C.1 for which the sign is either 0 or 1. Besides, the exponent is an integer $e_{min} \leq e \leq e_{max}$ and m is represented by a finite digit string containing p digits d_i, $0 \leq d_i \leq \beta$ and therefore $0 \leq m < \beta$.
- Two infinites, $-\infty$ and $+\infty$.
- Two not-a-number representations (*NaNs*), a quiet *NaN* (*qNaN*) and a signaling *NaN* (*sNaN*).

Floating-point numbers according to IEEE Standard 754™-2008 have a unique encoding using k bits. The standard ensures unique encodings my maximizing the significand while decreasing the exponent e until either $e = e_{min}$ or $m \geq 1$. The encoding of floating-point numbers in k bits relies on the following three fields:

1. A sign bit S.
2. A biased exponent $E = e + bias$ comprising w bits.
3. A trailing significand field T with $t = p - 1$ bits, where the leading digit is implicitly encoded in the biased exponent E.

The three fields are concatenated in the form (S, E, T), such that the sum for the lengths of the bit fields equals k with $1 + w + t = k$. Different types of precision are specified in Table C.1 that determine specific values for k, p, t, w, and $bias$. The encoding of the biased exponent E distinguishes normal and subnormal numbers, as well as it reserves unique encodings for ± 0, $\pm\infty$ and NaNs. Values in the biased exponent E in the range $[1, 2^w - 2]$ encode normal floating-point numbers, while the reserved value 0 encodes ± 0 as well as denormal numbers. Besides, the value $2^w - 1$ encodes $\pm\infty$ and NaNs. The representation r for a floating-point number and its corresponding value v are computed from the fields S, E, and T as follows:

- For $E = 2^w - 1$ and $T \neq 0$, r is *qNaN* or *sNaN* and v is *NaN*.
- For $E = 2^w - 1$ and $T = 0$, r and $v = (-1)^S \cdot (+\infty)$.
- For $1 \leq E \leq 2^w - 2$, r is $(S, (E - bias), (1 + 2^{1-p} \cdot T))$. The value of the floating-point number is $v = (-1)^S \cdot 2^{E-bias} \cdot (1 + 2^{1-p} \cdot T)$. Therefore, normal numbers have an implicit leading significand bit of 1.
- For $E = 0$ and $T \neq 0$, r is $(S, e_{min}, (0 + 2^{1-p} \cdot T))$. The value of the floating-point number is $v = (-1)^S \cdot 2^{e_{min}} \cdot (0 + 2^{1-p} \cdot T)$. Subnormal numbers have an implicit leading significand bit of 0.

▼ **Table C.1** — Parameters of the IEEE 754-2008 standard for floating-point number formats adopted from [IEEE 08].

Parameter	**binary16**	**binary32**	**binary64**	**binary128**
k, storage width in bits	16	32	64	128
p, precision in bits	11	24	53	113
e_{max}, maximum exponent e	15	127	1023	16383
bias, $E - e$	15	127	1023	16383
sign bit	1	1	1	1
w, exponent field width in bits	5	8	11	15
t, trailing significand field width in bits	10	23	52	112

Following Table C.1, the machine epsilon (cf. Equation 3.11) for single and double precision number formats is therefore:

$$\varepsilon_M := 2^{-(p-1)} = 2^{-23} \approx 1.19 \cdot 10^{-7} \quad (\textbf{Single precision}). \tag{C.2}$$

$$\varepsilon_M := 2^{-(p-1)} = 2^{-52} \approx 2.22 \cdot 10^{-16} \quad (\textbf{Double precision}). \tag{C.3}$$

Appendix D

Additional Proofs

D.1 Rounding Error Bound for Sparse Matrices

This section describes the derivation of Equation 3.13 from Chapter 3. The derivation is based on the following assumptions: The matrix-vector multiplication $\boldsymbol{r} := \boldsymbol{A}\boldsymbol{b}$ is computed in floating-point arithmetic with machine epsilon ε_M (cf. Equation 3.11) and $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{b}, \boldsymbol{r} \in \mathbb{R}^{n \times 1}$. For this multiplication, an operand checksum vector $\boldsymbol{t} \in \mathbb{R}^{m' \times 1}$ and a result checksum vector $\boldsymbol{t}^* \in \mathbb{R}^{m' \times 1}$ are computed following Equations 3.5 and 3.6.

The derivation is based on the numerical analysis for the underlying operations (i.e. inner products and matrix-vector products), which shows that the *elements in the rounding error bound vector for sparse matrix operations* τ_k are bounds for the maximum difference between the floating-point representations of the checksums t_k and t_k^*:

$$|fl(t_k^*) - fl(t_k)| < \tau_k \,. \tag{D.1}$$

In the course of the numerical analysis, rounding error bounds for the underlying linear operations are composed to form upper bounds $\hat{\tau}_k$. Finally, it will be shown that the presented rounding error bound τ_k is equal to the rounding error bound $\hat{\tau}_k$ derived during numerical analysis: $\tau_k = \hat{\tau}_k$.

The required rounding error bounds follow the results in [Chowd96], which are briefly introduced below. The interested reader can find a detailed introduction for example in [Golub13], Section 2.7.2.

A central element of such a numerical analysis is the *mapping of real numbers to a floating-point representation.* Let ε_M be the machine epsilon (i.e. the machine rounding error) for floating-point operations and let $x \in \mathbb{R}$.
The function $fl(x)$ maps x to the *floating-point representation* of x:

$$fl(x) := x(1+\delta), \quad \text{with } |\delta| \leq \varepsilon_M\,. \tag{D.2}$$

For any basic arithmetic operation **op** (i.e. $+,-,\cdot,/$) performed in floating-point arithmetic on $x,y \in \mathbb{R}$ using machine epsilon ε_M, the function $fl(x\ \textbf{op}\ y)$ satisfies

$$fl(x\ \textbf{op}\ y) := (x\ \textbf{op}\ y)(1+\delta), \quad \text{with } |\delta| \leq \varepsilon_M\,. \tag{D.3}$$

The order in which multiple floating-point operations are being performed determines the induced rounding error, since floating-point operations are not associative ($fl(x\ \textbf{op}\ fl(y\ \textbf{op}\ z)) \neq fl(fl(x\ \textbf{op}\ y)\ \textbf{op}\ z)$). For this reason, rounding error bounds that cover more than two floating-point operations are determined by the order in which the floating-point operations are being performed.

The rounding error that occurs in summations is bounded as follows. Let $\{x_i \mid x_i \in \mathbb{R}\}$ be a set of n numbers. The sum of this set $fl(\sum_{i=1}^n x_i)$ computed in floating-point arithmetic in the order $i = 1,2,\ldots n$ satisfies:

$$fl(\sum_{i=1}^{n} x_i) := \textstyle\sum_{i=1}^n x_i(1+\delta_i) \tag{D.4}$$

with δ_i being the upper rounding error bound with

$$|\delta_1| < (n-1)\varepsilon_M \text{ and}$$
$$|\delta_i| < (n+1-i)\varepsilon_M\ (1 < i \leq n)$$

and ε_M being the machine epsilon.

The rounding error that occurs in inner products is bounded as follows. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two vectors that contain n elements $x_i, y_i \in \mathbb{R}$. The floating-point representation of the inner product between these two vectors computed in floating-point arithmetic $fl(\boldsymbol{x}^T\boldsymbol{y})$ satisfies

$$fl(\boldsymbol{x}^T\boldsymbol{y}) := fl(\sum_{i=1}^{n} \boldsymbol{x}_i\boldsymbol{y}_i) = \sum_{i=1}^{n} \boldsymbol{x}_i\boldsymbol{y}_i(1+\delta_i) \tag{D.5}$$

with δ_i being the upper rounding error bound with

$$|\delta_1| < n\varepsilon_M \text{ and } |\delta_i| < (n+2-i)\varepsilon_M, (1 < i \leq n) \ .$$

The difference between the result of the inner product $\boldsymbol{x}^T\boldsymbol{y}$ and its floating-point representation $fl(\boldsymbol{x}^T\boldsymbol{y})$ can be bounded by:

$$|fl(\boldsymbol{x}^T\boldsymbol{y}) - \boldsymbol{x}^T\boldsymbol{y}| \leq n \cdot \varepsilon_M \|\boldsymbol{x}\|_2 \|\boldsymbol{y}\|_2 \ . \tag{D.6}$$

In the following, rounding error bounds are derived for the different underlying linear operations that are performed for each fault-tolerant sparse matrix-vector multiplication as presented in Chapter 3.3. The presented analytical rounding error bound τ_k in Equation 3.13 relies on the number of non-empty columns n'_k in a row block matrix $\boldsymbol{A_k}$ to determine the maximum difference between the operand t_k and result checksums elements t^*_k.

The difference between the k-th operand checksum element t_k and its floating-point representation $fl(t_k)$ is bounded by

$$|fl(t_k) - t_k| < \sigma_k \cdot \varepsilon_M \cdot \sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \|\boldsymbol{b}\|_2 + n'_k \cdot \varepsilon_M \cdot \|[\boldsymbol{C}]_k\|_2 \|\boldsymbol{b}\|_2. \tag{D.7}$$

with $\boldsymbol{w}^{(k)}$ being the k-th $(1 \times \sigma)$-weight vector and $1 \leq k \leq m'$. The term σ_k is the number of rows in block k and n'_k is the number of non-zero columns in $\boldsymbol{A}_k$. The term $[\boldsymbol{A}_k]_i$ denotes the i-th row in row block k and the term $[\boldsymbol{C}]_k$ is the k-th row in $\boldsymbol{C}$.

Proof: The floating-point representation of a checksum element $c_{i,j}$ in the checksum matrix $\boldsymbol{C}$ is

$$fl(c_{k,j}) := fl\left(\left[\boldsymbol{w}^{(k)}\boldsymbol{A}_k\right]_j\right) = \sum_{i=1}^{\sigma_k} [\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j} \cdot (1+\delta_i) \tag{D.8}$$

with δ_i being the upper rounding error bound and

$$|\delta_1| < \sigma_k \cdot \varepsilon_M \text{ and } |\delta_i| < (\sigma_k + 2 - i) \cdot \varepsilon_M \text{ for } 1 < i \leq \sigma_k$$

which follows from the direct application of Equation D.4. The floating-point representation of an operand checksum element $fl(t_k)$ is

$$fl(t_k) := fl\left(\sum_{j=1}^{n} fl(c_{k,j}) \cdot b_j\right) = \sum_{j=1}^{n} fl(c_{k,j}) \cdot b_j \cdot (1+\delta_j)$$
$$= \sum_{j=1}^{n}\sum_{i=1}^{\sigma_k}\left([\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j}\right) \cdot b_j \cdot (1+[\xi_k]_{i,j})(1+\delta_j) \tag{D.9}$$

with $[\xi_k]_{i,j}$ being the upper rounding error bound and $|[\xi_k]_{1,j}| < \sigma_k \cdot \varepsilon_M$ and $|[\xi_k]_{i,j}| < (\sigma_k + 2 - i) \cdot \varepsilon_M$ with $1 < i \leq \sigma_k$. The difference between the operand checksum elements t_k and its floating-point representation $fl(t_k)$ is:

$$|fl(t_k) - t_k| = |\sum_{j=1}^{n}\sum_{i=1}^{\sigma_k}[\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot (1+[\xi_k]_{i,j})(1+\delta_j) - \sum_{j=1}^{n} c_{k,j} \cdot b_j| \tag{D.10}$$

By reorganizing the terms, applying the triangle inequality while neglecting quadratic contributions of the machine epsilon $\mathcal{O}(\varepsilon_M^2)$, the difference can be bounded by:

$$|fl(t_k) - t_k| \leq \sum_{j=1}^{n}\sum_{i=1}^{\sigma_k}|[\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot [\xi_k]_{i,j}| + \sum_{j=1}^{n}|c_{k,j} \cdot b_j \cdot \delta_j| \tag{D.11}$$

An upper estimation can be formulated for this difference: With n'_k being the number of non-zero columns in the row block matrix $\boldsymbol{A}_k$, $\delta_j < n' \cdot \varepsilon_M$, which allows to substitute δ_j with $n'_k \cdot \varepsilon_M$. Using $[\xi_k]_{i,j} < \sigma_k \cdot \varepsilon_M$ and the application of the Cauchy-Schwarz inequality:

$$|fl(t_k) - t_k| < \sigma_k \cdot \varepsilon_M \cdot \sum_{i=1}^{\sigma_k}|[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \|\boldsymbol{b}\|_2 + n'_k \cdot \varepsilon_M \cdot \|[\boldsymbol{C}]_k\|_2 \|\boldsymbol{b}\|_2. \tag{D.12}$$

□

The difference between the k-th result checksum element t^*_k and its floating-point representation $fl(t^*_k)$ can be bounded by:

$$|fl(t^*_k) - t^*_k| < \varepsilon_M \cdot (n'_k + \sigma_k) \cdot \|\boldsymbol{b}\|_2 \cdot \sum_{i=1}^{\sigma_k}|[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \tag{D.13}$$

with n'_k being the number of non-zero columns in $\boldsymbol{A}_k$.

Proof: The floating-point representation of a result element $fl([\boldsymbol{r}_k]_i)$ is

$$fl([\boldsymbol{r}_k]_i) := \sum_{j=1}^{n} [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot (1 + [\xi_k]_{i,j}) \tag{D.14}$$

with $[\xi_k]_{i,j}$ being the upper rounding error bound and $|[\xi_k]_{i,1}| < n \cdot \varepsilon_M$ and $|[\xi_k]_{i,j}| < (n + 2 - j) \cdot \varepsilon_M$, $(1 < j \leq n)$, which follows from Equation D.6. The floating-point representation of a result checksum element $fl(t_k^*)$ is

$$fl(t_k^*) := fl\left(\boldsymbol{w}^{(k)} \cdot fl(\boldsymbol{r}_k)\right) = \sum_{i=1}^{\sigma_k} [\boldsymbol{w}^{(k)}]_i \cdot fl([\boldsymbol{r}_k]_i)(1 + \rho_i) \tag{D.15}$$

$$= \sum_{i=1}^{\sigma_k} [\boldsymbol{w}^{(k)}]_i \cdot \sum_{j=1}^{n} [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot (1 + [\xi_k]_{i,j})(1 + \rho_i) \tag{D.16}$$

The difference between the k-th result checksum elements t_k^* and its floating-point representation $fl(t_k^*)$ is:

$$|fl(t_k^*) - t_k^*| = |\sum_{i=1}^{\sigma_k} [\boldsymbol{w}^{(k)}]_i \cdot \sum_{j=1}^{n} [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot (1 + [\xi_k]_{i,j})(1 + \rho_i) - \sum_{i=1}^{\sigma_k} \sum_{j=1}^{n} [\boldsymbol{A}_k]_{i,j} \cdot b_j| \tag{D.17}$$

By reorganizing the terms, applying the triangle inequality while neglecting quadratic contributions of the machine epsilon $\mathcal{O}(\varepsilon_M^2)$, the difference can be bounded by:

$$|fl(t_k^*) - t_k^*| \leq |\sum_{i=1}^{\sigma_k} \sum_{j=1}^{n} [\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot [\xi_k]_{i,j}| + |\sum_{i=1}^{\sigma_k} \sum_{j=1}^{n} [\boldsymbol{w}^{(k)}]_i \cdot [\boldsymbol{A}_k]_{i,j} \cdot b_j \cdot \rho_i| \tag{D.18}$$

An upper estimation can be formulated for this difference. With n'_k being the number of non-zero columns in the row block matrix $\boldsymbol{A}_k$, $[\xi_k]_{i,j} < n'_k \cdot \varepsilon_M$. Using $\rho_i < \sigma_k \cdot \varepsilon_M$ and the application of the Cauchy-Schwarz inequality:

$$|fl(t_k^*) - t_k^*| < \varepsilon_M \cdot (n'_k + \sigma_k) \cdot \|\boldsymbol{b}\|_2 \cdot \sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \; . \tag{D.19}$$

□

An upper bound $\hat{\tau}_k$ is determined for the difference between the floating-point representation of t_k and the floating-point representation of t_k^* by combining the results from Equations D.7 and D.13 such that $|fl(t_k^*) - fl(t_k)| < \hat{\tau}_k$. At the same time, $\hat{\tau}_k = \tau_k$:

$$|fl(t_k^*) - fl(t_k)| < \hat{\tau}_k = \tau_k \; , \tag{D.20}$$

which leads to Equation 3.13.

Proof: For the block k, the difference between the floating-point representations of the operand checksum element t_k and the result checksum element t_k^* is

$$|fl(t_k) - fl(t_k^*)| = |fl(t_k) - t_k + t_k - fl(t_k^*) - t_k^* + t_k^*)| \tag{D.21}$$

With $t_k = t_k^*$ and the application of the triangle inequality:

$$\leq |fl(t_k) - t_k| + |fl(t_k^*) - t_k^*| \tag{D.22}$$

By substituting $|fl(t_k) - t_k|$ and $|fl(t_k^*) - t_k^*|$, the difference can be formulated as

$$\begin{aligned} &< \sigma_k \cdot \varepsilon_M \cdot \sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \|\boldsymbol{b}\|_2 + n'_k \cdot \varepsilon_M \cdot \|[\boldsymbol{C}]_k\|_2 \|\boldsymbol{b}\|_2 \\ &+ \varepsilon_M \cdot (n'_k + \sigma_k) \cdot \|\boldsymbol{b}\|_2 \cdot \sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 \end{aligned} \tag{D.23}$$

After reorganization:

$$\begin{aligned} &|fl(t_k) - fl(t_k^*)| < \\ &\|\boldsymbol{b}\|_2 \cdot \varepsilon_M \cdot ((n'_k + 2 \cdot \sigma_k) \textstyle\sum_{i=1}^{\sigma_k} |[\boldsymbol{w}^{(k)}]_i| \cdot \|[\boldsymbol{A}_k]_i\|_2 + n'_k \cdot \|[\boldsymbol{C}]_k\|_2) \\ &=: \tilde{\tau}_k = \tau_k \, . \end{aligned} \tag{D.24}$$

□

Appendix E

Experimental Setup and Data

This appendix presents the experimental setup with respect to the hardware configuration that was used to provide the experimental results in Chapter 7. Besides, this appendix complements these experimental results by providing additional results for the different experiments.

E.1 Hardware and Software Parameter

Tables E.1, E.2, and E.3 show the hardware and software specifications of the system that was used to conduct the experiments.

▼ **Table E.1** — Host system specification.

CPUs:	2x Intel Xeon E5-2623
GPUs:	4x Nvidia Tesla K80
Memory:	128 GB
Operating system:	Cent OS 6 Linux
GCC version:	4.9.2
CUDA version:	8.0

▼ **Table E.2** — Host CPU specification.

Name:	Intel Xeon E5-2623
Cores:	4
Frequency:	3.0 GHz
Theo. double precision peak performance:	96 GFLOPs
Thermal design power:	105 W

▼ **Table E.3** — GPU specification.

Name:	Tesla K80
GPU cores:	2×GK210
Microarchitecture:	Nvidia Kepler
Stream processors:	2×2496
Clock frequency:	560 MHz
Theo. double precision peak performance:	2.91 TFLOPs
Memory size:	2×12 GByte GDDR5
Memory protection:	ECC (memories, caches and registers)
Memory clock:	2.5 GHz
Memory bandwidth:	480 GB/s (aggregated)
Memory interface:	384 Bit
System interface:	PCI Express Gen3 ×16
Thermal design power:	300 W

E.2 Fault-tolerant Sparse Matrix-Vector Multiplications

Table E.4 shows the runtime of the original matrix-vector multiplications and the runtime overhead of the protected matrix-vector multiplications in the error-free case.

▼ **Table E.4** — Average runtime of the original sparse matrix-vector multiplication T_{SpMV} and average runtime overhead of the protected sparse matrix-vector multiplication O_S in the error-free case for different block sizes.

Matrix	T_{SpMV}	O_S for Block Size σ_k									
name	[ms]	1	2	4	8	16	32	64	128	256	512
nos3	33.5	178.2%	148.8%	133.6%	121.5%	115.1%	115.0%	121.9%	125.4%	126.6%	137.9%
bcsstk10	33.1	125.1%	110.9%	93.0%	93.3%	91.6%	89.0%	96.5%	99.7%	101.6%	111.0%
msc01050	45.0	83.6%	74.0%	65.7%	60.2%	58.8%	59.1%	62.7%	62.3%	66.9%	69.8%
bcsstk21	31.7	127.7%	95.4%	66.7%	64.0%	63.2%	60.7%	67.4%	69.9%	73.0%	86.3%
bcsstk11	34.4	91.1%	89.7%	88.1%	89.2%	87.1%	84.6%	88.4%	90.2%	95.9%	98.0%
nasa2146	37.4	105.4%	100.7%	98.8%	95.5%	95.9%	95.5%	102.8%	104.8%	107.4%	112.7%
sts4098	100.7	56.4%	54.0%	53.4%	46.2%	47.9%	46.0%	46.6%	48.5%	49.9%	55.5%
bcsstk13	42.7	64.0%	62.6%	59.4%	56.7%	58.3%	58.0%	65.3%	64.8%	66.3%	76.5%
msc04515	44.4	44.4%	43.3%	42.9%	42.6%	44.7%	43.2%	45.6%	45.2%	49.8%	64.0%
ex9	44.1	56.9%	48.6%	47.8%	46.6%	47.2%	46.7%	47.0%	46.6%	52.3%	56.4%
bodyy4	52.5	43.2%	35.7%	35.7%	35.2%	33.9%	36.4%	38.5%	38.9%	42.9%	50.6%
bodyy5	54.6	60.9%	51.2%	51.0%	49.8%	48.5%	50.0%	53.4%	56.8%	58.2%	61.3%
bodyy6	55.5	47.9%	42.5%	39.3%	39.1%	37.8%	40.4%	40.7%	41.7%	46.5%	51.4%
Muu	60.6	76.2%	74.8%	65.7%	65.2%	65.1%	62.5%	72.2%	78.1%	83.6%	94.2%
s3rmt3m3	62.1	29.8%	28.5%	27.7%	27.8%	28.1%	28.0%	28.4%	28.7%	29.1%	30.7%
s3rmt3m1	65.1	42.2%	38.1%	36.1%	35.7%	36.9%	37.6%	39.5%	39.7%	43.3%	52.5%
bcsstk28	69.5	49.9%	45.0%	43.9%	42.8%	42.6%	39.4%	41.9%	44.7%	53.1%	72.1%
s3rmq4m1	75.6	26.5%	23.4%	22.7%	21.9%	20.6%	20.9%	21.7%	22.3%	24.2%	25.6%
bcsstk16	81.8	68.3%	67.5%	66.1%	64.8%	62.4%	59.5%	65.2%	67.0%	67.9%	70.7%
Kuu	70.4	71.7%	62.6%	59.7%	57.0%	51.9%	50.1%	56.1%	60.5%	67.5%	72.2%
bcsstk38	145.3	30.6%	25.2%	22.9%	23.0%	21.9%	21.8%	24.3%	25.0%	26.4%	30.0%
msc23052	197.5	77.3%	50.2%	31.9%	27.9%	27.3%	27.4%	28.7%	33.8%	39.9%	59.1%
msc10848	147.8	54.6%	36.5%	28.9%	28.1%	24.0%	25.9%	35.7%	40.5%	45.5%	63.1%
cfd2	402.4	122.0%	80.0%	60.4%	41.8%	36.0%	29.7%	25.6%	24.2%	23.7%	24.3%
nd3k	296.6	104.7%	64.3%	40.7%	28.8%	24.8%	22.3%	22.1%	31.2%	46.0%	60.8%
ship_001	451.9	109.2%	64.7%	42.2%	27.9%	21.2%	17.2%	16.6%	19.9%	28.4%	40.8%
shipsec5	1,250.6	106.2%	61.1%	31.8%	20.9%	16.3%	12.4%	10.2%	8.6%	7.0%	7.3%
G3_circuit	1,298.3	113.3%	68.8%	56.3%	46.6%	41.1%	38.0%	38.6%	37.3%	37.6%	38.0%
hood	1,351.0	109.6%	64.1%	38.6%	22.4%	16.7%	13.3%	11.3%	10.7%	15.0%	22.3%
crankseg_1	930.5	52.4%	32.3%	20.0%	14.3%	13.4%	12.4%	12.8%	13.5%	16.2%	20.3%

Table E.5 shows the runtime of the original matrix-vector multiplications and the runtime overhead of the protected matrix-vector multiplications in case of errors (i.e., overhead for error detection and correction).

▼ **Table E.5** — Average runtime of the original sparse matrix-vector multiplication T_{SpMV} and average runtime overhead of the protected sparse matrix-vector multiplication O_E to detect and correct errors for different block sizes.

Matrix	T_{SpMV}	O_E for Block Size σ_k									
name	[ms]	1	2	4	8	16	32	64	128	256	512
nos3	33.5	183.3%	158.4%	156.7%	154.4%	155.6%	153.9%	156.2%	165.6%	182.6%	195.8%
bcsstk10	33.1	143.7%	142.8%	139.1%	136.7%	134.6%	129.5%	133.3%	137.1%	139.2%	154.9%
msc01050	45.0	107.8%	106.6%	104.6%	97.8%	96.3%	98.0%	100.8%	100.5%	103.9%	104.1%
bcsstk21	31.7	133.6%	115.0%	97.9%	96.5%	96.2%	91.5%	92.4%	93.2%	102.4%	106.4%
bcsstk11	34.4	139.8%	138.0%	130.4%	129.9%	126.0%	125.0%	127.9%	129.5%	133.0%	135.9%
nasa2146	37.4	172.3%	170.1%	167.5%	163.4%	162.4%	155.7%	156.1%	156.1%	165.0%	167.3%
sts4098	100.7	93.5%	84.4%	83.3%	71.2%	64.6%	52.3%	55.2%	61.2%	68.5%	71.7%
bcsstk13	42.7	104.2%	94.7%	91.8%	88.2%	88.0%	86.6%	88.8%	90.1%	92.4%	94.8%
msc04515	44.4	74.3%	73.0%	72.6%	71.4%	70.4%	69.3%	69.6%	72.0%	72.3%	75.5%
ex9	44.1	86.0%	84.9%	81.9%	80.5%	78.1%	78.0%	77.1%	80.8%	81.7%	83.7%
bodyy4	52.5	89.4%	73.2%	69.0%	65.2%	59.7%	62.3%	63.6%	63.6%	65.8%	69.7%
bodyy5	54.6	123.0%	101.2%	97.0%	93.8%	85.9%	84.1%	86.0%	88.1%	88.9%	92.8%
bodyy6	55.5	82.5%	67.8%	60.8%	58.0%	56.9%	57.0%	57.3%	59.2%	60.0%	65.1%
Muu	60.6	128.2%	117.8%	111.8%	103.5%	103.0%	98.4%	99.3%	103.7%	106.0%	115.4%
s3rmt3m3	62.1	57.7%	50.5%	50.6%	51.3%	50.7%	46.7%	47.3%	47.3%	47.9%	49.3%
s3rmt3m1	65.1	76.9%	68.9%	66.6%	65.0%	64.5%	63.7%	63.7%	65.3%	68.9%	77.5%
bcsstk28	69.5	69.7%	67.1%	66.1%	64.4%	63.6%	63.1%	66.4%	68.4%	73.8%	82.7%
s3rmq4m1	75.6	49.8%	42.1%	39.1%	38.1%	37.2%	35.6%	37.8%	38.4%	38.5%	38.6%
bcsstk16	81.8	80.6%	79.8%	79.5%	78.1%	74.4%	73.6%	74.4%	74.8%	75.5%	76.4%
Kuu	70.4	85.0%	73.6%	67.5%	62.0%	60.7%	57.2%	67.9%	74.1%	81.5%	85.3%
bcsstk38	145.3	47.3%	43.3%	40.8%	37.2%	34.8%	32.1%	35.7%	39.7%	45.5%	47.9%
msc23052	197.5	96.3%	67.9%	49.8%	43.3%	41.2%	41.5%	38.4%	48.9%	51.4%	62.5%
msc10848	147.8	67.4%	48.7%	44.1%	42.9%	39.6%	40.3%	45.5%	52.5%	55.3%	67.7%
cfd2	402.4	188.4%	118.6%	87.3%	64.3%	54.2%	44.1%	40.3%	37.7%	35.6%	36.3%
nd3k	296.6	110.3%	66.7%	48.4%	40.0%	35.5%	34.0%	33.8%	38.2%	47.5%	64.6%
ship_001	451.9	113.7%	68.9%	48.8%	37.1%	30.0%	24.7%	23.0%	26.3%	32.7%	45.9%
shipsec5	1,250.6	126.9%	71.7%	41.0%	27.9%	20.8%	16.7%	13.9%	12.4%	11.0%	10.6%
G3_circuit	1,298.3	182.1%	102.7%	77.8%	70.2%	63.8%	50.0%	50.0%	50.6%	50.6%	50.6%
hood	1,351.0	138.8%	79.1%	48.2%	29.6%	22.4%	17.4%	16.3%	16.0%	18.2%	26.3%
crankseg_1	930.5	61.5%	38.7%	25.6%	18.4%	16.7%	15.6%	16.4%	16.5%	19.5%	24.9%

Tables E.6 and E.7 show the error coverage with respect to the balanced F_1-score of the protected sparse matrix-vector multiplication in case of errors.

▼ **Table E.6** — Balanced F_1-score of the protected sparse matrix-vector multiplication in case of single-bit flip errors for different block sizes.

Matrix name	**Balanced F_1-score for Block Size σ_k**									
	1	2	4	8	16	32	64	128	256	512
nos3	0.63	0.61	0.63	0.62	0.64	0.61	0.64	0.62	0.61	0.65
bcsstk10	0.93	0.92	0.94	0.93	0.94	0.92	0.93	0.93	0.91	0.91
msc01050	0.91	0.84	0.84	0.84	0.80	0.87	0.81	0.81	0.83	0.80
bcsstk21	0.79	0.79	0.74	0.72	0.71	0.75	0.77	0.71	0.76	0.73
bcsstk11	0.77	0.76	0.73	0.73	0.72	0.72	0.67	0.70	0.73	0.79
nasa2146	0.92	0.98	0.92	0.96	0.91	0.90	0.92	0.92	0.92	0.96
sts4098	0.92	0.91	0.88	0.85	0.80	0.82	0.88	0.92	0.88	0.88
bcsstk13	0.92	0.89	0.91	0.87	0.83	0.87	0.82	0.84	0.90	0.86
msc04515	0.79	0.80	0.79	0.80	0.77	0.75	0.83	0.85	0.71	0.72
ex9	0.92	0.90	0.94	0.88	0.82	0.78	0.92	0.87	0.86	0.90
bodyy4	0.95	0.92	0.93	0.92	0.91	0.92	0.93	0.92	0.90	0.94
bodyy5	0.95	0.95	0.88	0.88	0.86	0.93	0.93	0.93	0.93	0.93
bodyy6	0.94	0.91	0.96	0.98	0.94	0.97	0.98	0.93	0.93	0.96
Muu	0.96	0.95	0.90	0.91	0.91	0.92	0.88	0.92	0.87	0.90
s3rmt3m3	0.92	0.91	0.88	0.82	0.87	0.85	0.90	0.85	0.87	0.84
s3rmt3m1	0.85	0.77	0.76	0.73	0.77	0.82	0.83	0.78	0.85	0.85
bcsstk28	0.71	0.65	0.69	0.63	0.74	0.68	0.75	0.72	0.67	0.67
s3rmq4m1	0.88	0.81	0.78	0.81	0.88	0.79	0.82	0.79	0.86	0.91
bcsstk16	0.69	0.68	0.69	0.68	0.68	0.70	0.71	0.73	0.64	0.68
Kuu	0.62	0.64	0.58	0.61	0.61	0.63	0.64	0.64	0.64	0.64
bcsstk38	0.92	0.98	0.94	0.92	0.87	0.85	0.93	0.95	0.89	0.94
msc23052	0.94	0.97	0.98	0.95	0.93	0.93	0.97	0.95	0.95	0.95
msc10848	0.82	0.81	0.76	0.75	0.74	0.71	0.73	0.76	0.68	0.76
cfd2	0.95	0.91	0.89	0.91	0.89	0.89	0.87	0.89	0.89	0.87
nd3k	0.94	0.91	0.91	0.92	0.92	0.89	0.91	0.90	0.89	0.79
ship_001	0.94	0.95	0.97	0.96	0.97	0.94	0.97	0.94	0.95	0.95
shipsec5	0.89	0.89	0.90	0.88	0.88	0.87	0.90	0.85	0.88	0.82
G3_circuit	0.77	0.79	0.77	0.78	0.71	0.75	0.74	0.74	0.73	0.74
hood	0.94	0.97	0.98	0.97	0.94	0.99	0.98	0.99	0.94	0.96
crankseg_1	0.93	0.92	0.89	0.92	0.91	0.84	0.85	0.86	0.86	0.86

▼ **Table E.7** — Balanced F_1-score of the protected sparse matrix-vector multiplication in case of multi-bit flip errors for different block sizes.

Matrix	**Balanced F_1-score for Block Size σ_k**									
name	1	2	4	8	16	32	64	128	256	512
nos3	0.63	0.62	0.62	0.63	0.62	0.62	0.65	0.63	0.66	0.65
bcsstk10	0.94	0.94	0.93	0.93	0.94	0.92	0.94	0.93	0.91	0.94
msc01050	0.91	0.87	0.85	0.83	0.82	0.88	0.83	0.82	0.84	0.81
bcsstk21	0.79	0.80	0.78	0.77	0.77	0.76	0.78	0.75	0.76	0.74
bcsstk11	0.77	0.78	0.77	0.77	0.76	0.75	0.72	0.73	0.75	0.80
nasa2146	0.92	0.98	0.98	0.97	0.95	0.93	0.93	0.96	0.98	0.98
sts4098	0.92	0.95	0.93	0.91	0.85	0.88	0.92	0.92	0.91	0.93
bcsstk13	0.92	0.94	0.93	0.90	0.84	0.87	0.88	0.89	0.91	0.90
msc04515	0.79	0.82	0.81	0.81	0.80	0.75	0.84	0.87	0.74	0.78
ex9	0.92	0.96	0.96	0.89	0.87	0.79	0.93	0.91	0.91	0.93
bodyy4	0.96	0.97	0.95	0.95	0.94	0.94	0.97	0.97	0.94	0.94
bodyy5	0.95	0.96	0.92	0.91	0.90	0.95	0.94	0.95	0.96	0.97
bodyy6	0.94	0.97	1.00	0.98	0.98	0.98	0.98	0.98	0.97	0.99
Muu	0.96	0.97	0.93	0.95	0.94	0.93	0.92	0.94	0.91	0.92
s3rmt3m3	0.92	0.93	0.91	0.85	0.89	0.89	0.91	0.87	0.89	0.87
s3rmt3m1	0.86	0.82	0.80	0.78	0.80	0.83	0.83	0.84	0.87	0.86
bcsstk28	0.71	0.70	0.70	0.69	0.76	0.72	0.80	0.73	0.73	0.71
s3rmq4m1	0.88	0.83	0.80	0.83	0.90	0.83	0.82	0.83	0.86	0.95
bcsstk16	0.69	0.70	0.69	0.70	0.70	0.72	0.72	0.73	0.68	0.73
Kuu	0.63	0.66	0.61	0.66	0.65	0.65	0.66	0.66	0.66	0.67
bcsstk38	0.92	0.98	0.97	0.96	0.90	0.87	0.96	0.97	0.95	0.95
msc23052	0.94	1.00	0.99	1.00	0.98	0.99	0.98	0.99	0.98	0.99
msc10848	0.82	0.85	0.77	0.77	0.78	0.75	0.74	0.78	0.71	0.80
cfd2	0.95	0.96	0.94	0.93	0.93	0.92	0.91	0.91	0.92	0.89
nd3k	0.94	0.96	0.92	0.94	0.93	0.93	0.92	0.93	0.91	0.84
ship_001	0.94	0.99	0.99	1.00	1.00	1.00	1.00	0.99	1.00	1.00
shipsec5	0.90	0.90	0.90	0.93	0.89	0.91	0.91	0.87	0.89	0.86
G3_circuit	0.77	0.79	0.80	0.82	0.73	0.76	0.76	0.78	0.76	0.79
hood	0.94	0.98	0.98	0.98	1.00	1.00	1.00	1.00	0.99	1.00
crankseg_1	0.93	0.92	0.94	0.94	0.95	0.89	0.91	0.91	0.88	0.91

E.3 Fault Tolerance for Conjugate Gradient Solvers

Vulnerability of unprotected Conjugate Gradient Solvers

Tables E.8, E.9, E.10, and E.11 show the vulnerability of the evaluated Conjugate Gradient Solvers when no fault tolerance technique is applied in case of errors.

▼ **Table E.8** — Number of successfully converged experiments (Conv.), diverged experiments (Div.), and experiments that resulted in silent data corruptions (SDC) in case of single-bit flip error injections (i.e. one error injection per experiment).

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	Conv.	Div.	SDC	Conv.	Div.	SDC	Conv.	Div.	SDC
nos3	459	109	432	427	120	453	429	71	500
bcsstk10	405	357	238	358	370	272	375	125	500
msc01050	328	165	507	382	162	456	367	33	600
bcsstk21	417	229	354	560	200	240	500	0	500
bcsstk11	542	29	429	571	58	371	550	75	375
nasa2146	203	159	638	222	167	611	269	116	615
sts4098	880	0	120	487	0	513	425	0	575
bcsstk13	122	123	755	381	48	571	238	95	667
msc04515	111	89	800	500	100	400	525	100	375
ex9	0	125	875	48	95	857	25	125	850
bodyy4	800	50	150	750	50	200	775	50	175
bodyy5	563	62	375	588	59	353	650	50	300
bodyy6	563	124	313	444	167	389	374	188	438
Muu	429	0	571	400	200	400	454	91	455
s3rmt3m3	714	0	286	478	44	478	444	334	222
s3rmt3m1	783	87	130	810	47	143	850	75	75
bcsstk28	200	200	600	61	30	909	56	111	833
s3rmq4m1	750	62	188	636	91	273	666	167	167
bcsstk16	71	48	881	200	133	667	200	67	733
Kuu	474	52	474	526	53	421	583	84	333
bcsstk38	393	0	607	600	50	350	545	91	364
msc23052	91	386	523	73	25	902	24	49	927
msc10848	122	82	796	172	0	828	166	42	792
cfd2	897	26	77	821	25	154	880	0	120
nd3k	556	0	444	375	0	625	250	125	625
ship_001	45	46	909	51	77	872	52	104	844
shipsec5	643	71	286	400	57	543	388	68	544
G3_circuit	857	0	143	778	0	222	633	0	367
hood	583	84	333	130	0	870	51	26	923
crankseg_1	266	167	567	383	0	617	225	0	775

▼ **Table E.9** — Number of successfully converged experiments (Conv.), diverged experiments (Div.), and experiments that resulted in silent data corruptions (SDC) in case of multi-bit flip error injections (i.e. one error injection per experiment).

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	Conv.	Div.	SDC	Conv.	Div.	SDC	Conv.	Div.	SDC
nos3	427	120	453	408	132	460	417	83	500
bcsstk10	353	388	259	354	390	256	313	187	500
msc01050	319	181	500	338	169	493	300	50	650
bcsstk21	417	200	383	537	204	259	462	0	538
bcsstk11	526	53	421	528	56	416	462	76	462
nasa2146	180	157	663	205	178	617	250	83	667
sts4098	833	0	167	463	0	537	447	0	553
bcsstk13	132	151	717	368	105	527	227	91	682
msc04515	167	83	750	475	100	425	525	100	375
ex9	0	125	875	23	140	837	25	125	850
bodyy4	800	25	175	725	50	225	775	50	175
bodyy5	533	67	400	563	63	374	647	59	294
bodyy6	428	143	429	412	176	412	286	214	500
Muu	333	0	667	333	222	445	455	90	455
s3rmt3m3	750	0	250	450	50	500	375	250	375
s3rmt3m1	773	91	136	800	50	150	850	75	75
bcsstk28	100	200	700	29	29	942	625	75	300
s3rmq4m1	733	67	200	550	100	350	611	167	222
bcsstk16	49	49	902	154	154	692	211	52	737
Kuu	470	59	471	500	56	444	538	77	385
bcsstk38	393	18	589	575	100	325	529	86	385
msc23052	69	326	605	57	29	914	30	59	911
msc10848	102	82	816	154	0	846	146	68	786
cfd2	897	26	77	816	26	158	750	50	200
nd3k	500	0	500	222	0	778	200	100	700
ship_001	22	89	889	26	79	895	39	108	853
shipsec5	444	56	500	387	65	548	374	70	556
G3_circuit	800	0	200	545	0	455	589	0	411
hood	500	83	417	50	20	930	24	49	927
crankseg_1	271	83	646	170	200	630	154	0	846

▼ **Table E.10** — Average number of iterations in the error-free case I_S, average number of iterations in case of single-bit flip error injections I_E (i.e. one error injection per experiment) and average resulting iteration overhead O_E to converge to correct results.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]
nos3	275	418.9	52.3	250	467.1	86.8	142	261.2	83.9
bcsstk10	3,093	3,240.9	4.8	1,062	1,186.1	11.7	554	611.6	10.4
msc01050	5,656	7,391.5	30.7	1,359	2,110.2	55.3	1,481	2,600.0	75.6
bcsstk21	10,937	12,928.8	18.2	790	887.3	12.3	329	384.0	16.7
bcsstk11	19,259	22,557.2	17.1	5,741	6,810.6	18.6	22,993	24,497.2	6.5
nasa2146	506	720.3	42.3	415	545.0	31.3	359	422.1	17.6
sts4098	15,688	16,758.8	6.8	604	710.4	17.6	563	635.7	12.9
bcsstk13	1,889	3,076.5	62.9	1,573	3,102.6	97.2	1,489	2,561.9	72.1
msc04515	5,631	9,987.6	77.4	4,827	8,978.9	86.0	4,384	7,160.7	63.3
ex9	77,631	99,179.4	27.8	17,290	20,340.6	17.6	15,369	25,776.0	67.7
bodyy4	226	256.2	13.3	213	264.0	24.0	284	345.3	21.6
bodyy5	717	764.1	6.6	538	584.7	8.7	1,466	1,565.9	6.8
bodyy6	2,184	2,644.5	21.1	1,271	1,295.2	1.9	1,090	1,212.3	11.2
Muu	44	47.8	8.6	17	19.5	14.8	12	14.7	22.8
s3rmt3m3	838	876.1	4.5	15,436	16,193.8	4.9	10,935	13,536.1	23.8
s3rmt3m1	76,595	86,501.5	12.9	11,692	14,826.6	26.8	65,550	71,643.7	9.3
bcsstk28	13,776	24,199.0	75.7	5,142	8,309.1	61.6	3,138	6,213.0	98.0
s3rmq4m1	50,410	72,210.5	43.2	8,070	10,550.5	30.7	48,955	72,914.6	48.9
bcsstk16	620	630.7	1.7	279	284.9	2.1	225	228.8	1.7
Kuu	684	697.5	2.0	545	585.5	7.4	243	261.7	7.7
bcsstk38	19,575	21,232.7	8.5	15,001	15,333.8	2.2	37,301	39,232.7	5.2
msc23052	284,012	309,322.2	8.9	217,329	243,577.5	12.1	37,100	41,234.5	11.1
msc10848	110,121	122,343.4	11.1	5,782	6,767.9	17.1	5,147	6,095.3	18.4
cfd2	2,395	2,544.7	6.3	4,984	5,103.3	2.4	2,010	2,239.3	11.4
nd3k	4,214	4,789.4	13.7	7,509	9,665.2	28.7	4,338	4,792.5	10.5
ship_001	96,123	101,334.2	5.4	59,961	66,094.0	10.2	86,456	98,232.7	13.6
shipsec5	8,144	9,123.6	12.0	4,814	5,606.6	16.5	7,156	7,423.9	3.7
G3_circuit	9,391	9,491.5	1.1	3,070	3,914.9	27.5	6,231	11,661.1	87.1
hood	17,592	27,938.2	58.8	7,299	15,234.6	108.7	7,295	7,450.0	2.1
crankseg_1	2,884	3,126.7	8.4	958	980.6	2.4	742	759.4	2.4

▼ **Table E.11** — Average number of iterations in the error-free case I_S, average number of iterations in case of multi-bit flip error injections I_E (i.e. one error injection per experiment) and resulting average iteration overhead O_E to converge to correct results.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]	$\overline{I_S}$	$\overline{I_E}$	$\overline{O_E}$ [%]
nos3	275	488.6	77.7	250	494.3	97.7	142	260.5	83.5
bcsstk10	3,093	3,532.7	14.2	1,062	1,173.5	10.5	554	607.3	9.6
msc01050	5,656	7,901.5	39.7	1,359	2,295.8	68.9	1,481	2,845.8	92.2
bcsstk21	10,937	12,965.5	18.5	790	942.2	19.3	329	399.3	21.4
bcsstk11	19,259	23,751.2	23.3	5,741	7,256.8	26.4	22,993	29,114.2	26.6
nasa2146	506	721.5	42.6	415	545.9	31.5	359	504.8	40.6
sts4098	15,688	17,075.4	8.8	604	725.2	20.1	563	688.0	22.2
bcsstk13	1,889	4,634.6	145.3	1,573	4,111.8	161.4	1,489	2,642.9	77.5
msc04515	5,631	11,122.6	97.5	4,827	9,521.2	97.2	4,384	7,473.0	70.5
ex9	77,631	96,342.6	24.1	17,290	21,333.6	23.4	15,369	25,613.0	66.7
bodyy4	226	286.2	26.6	213	274.4	28.8	284	385.3	35.7
bodyy5	717	774.0	8.0	538	592.7	10.2	1,466	1,968.8	34.3
bodyy6	2,184	2,247.0	2.9	1,271	1,396.8	9.9	1,090	1,213.6	11.3
Muu	44	48.8	10.9	17	20.0	17.6	12	18.5	54.2
s3rmt3m3	838	896.5	7.0	15,436	16,793.9	8.8	10,935	13,437.6	22.9
s3rmt3m1	76,595	92,642.6	21.0	11,692	13,880.0	18.7	65,550	75,203.5	14.7
bcsstk28	13,776	34,631.9	151.4	5,142	11,381.6	121.3	3,138	6,261.5	99.5
s3rmq4m1	50,410	75,214.5	49.2	8,070	11,796.7	46.2	48,955	73,153.8	49.4
bcsstk16	620	630.7	1.7	279	290.4	4.1	225	235.7	4.7
Kuu	684	695.6	1.7	545	596.1	9.4	243	261.7	7.7
bcsstk38	19,575	21,354.6	9.1	15,001	15,739.4	4.9	37,301	42,232.2	13.2
msc23052	284,012	315,043.5	10.9	217,329	254,932.0	17.3	37,100	41,133.2	10.9
msc10848	110,121	133,443.7	21.2	5,782	6,807.5	17.7	5,147	5,892.1	14.5
cfd2	2,395	2,544.7	6.3	4,984	5,162.3	3.6	2,010	2,112.8	5.1
nd3k	4,214	4,988.5	18.4	7,509	9,915.3	32.0	4,338	4,760.2	9.7
ship_001	96,123	101,483.7	5.6	59,961	67,393.1	12.4	86,456	98,242.2	13.6
shipsec5	8,144	9,712.2	19.3	4,814	5,820.0	20.9	7,156	8,423.6	17.7
G3_circuit	9,391	15,486.6	64.9	3,070	4,050.6	31.9	6,231	9,416.1	51.1
hood	17,592	37,947.4	115.7	7,299	16,361.0	124.2	7,295	15,523.6	112.8
crankseg_1	2,884	3,093.3	7.3	958	996.0	4.0	742	754.9	1.7

Runtime overhead for error detection

Table E.12 shows the runtime overhead for error detection that the fault tolerence technique presented in Chapter 4 induces.

▼ **Table E.12** — Average execution time for unprotected solver execution T_S, average execution time for protected solver execution T_P, average runtime overhead for error detection O_P in the error-free case.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	$\overline{T_S}$ [s]	$\overline{T_P}$ [s]	$\overline{O_P}$ [%]	$\overline{T_S}$ [s]	$\overline{T_P}$ [s]	$\overline{O_P}$ [%]	$\overline{T_S}$ [s]	$\overline{T_P}$ [s]	$\overline{O_P}$ [%]
nos3	0.045	0.046	3.83	0.045	0.046	2.30	0.215	0.216	0.41
bcsstk10	0.421	0.432	2.59	0.168	0.172	2.23	1.337	1.339	0.15
msc01050	1.301	1.335	2.61	0.338	0.346	2.41	1.266	1.274	0.70
bcsstk21	1.995	2.044	2.42	0.160	0.163	2.18	0.230	0.231	0.63
bcsstk11	3.092	3.172	2.60	1.025	1.049	2.34	44.488	44.584	0.22
nasa2146	0.078	0.080	2.26	0.072	0.073	2.00	0.984	0.985	0.13
sts4098	4.249	4.343	2.23	0.175	0.179	2.08	1.341	1.344	0.25
bcsstk13	0.403	0.414	2.77	0.366	0.375	2.55	7.701	7.710	0.11
msc04515	1.142	1.177	3.04	1.073	1.103	2.78	12.910	12.937	0.21
ex9	13.141	13.509	2.79	3.252	3.334	2.51	97.834	97.906	0.07
bodyy4	0.320	0.322	0.61	0.309	0.310	0.60	1.685	1.688	0.15
bodyy5	0.996	1.001	0.48	0.761	0.765	0.47	8.172	8.182	0.12
bodyy6	2.827	2.842	0.52	1.679	1.687	0.51	5.458	5.465	0.13
Muu	0.015	0.015	2.23	0.006	0.006	2.09	0.037	0.037	0.25
s3rmt3m3	0.186	0.191	2.73	3.758	3.852	2.49	110.578	110.644	0.06
s3rmt3m1	17.036	17.506	2.76	2.829	2.901	2.53	286.574	286.976	0.14
bcsstk28	3.242	3.326	2.58	1.311	1.342	2.38	27.571	27.590	0.07
s3rmq4m1	11.941	12.253	2.62	2.079	2.129	2.41	264.145	264.448	0.11
bcsstk16	0.142	0.146	2.62	0.070	0.071	2.41	1.936	1.937	0.07
Kuu	0.198	0.202	2.12	0.169	0.173	1.98	1.476	1.477	0.10
bcsstk38	7.930	8.051	1.53	6.393	6.486	1.46	780.942	781.173	0.03
msc23052	601.164	603.248	0.35	466.721	468.315	0.34	213.110	213.382	0.13
msc10848	220.138	220.918	0.35	11.716	11.757	0.35	76.324	76.361	0.05
cfd2	8.058	8.079	0.26	17.130	17.174	0.26	54.529	54.547	0.03
nd3k	8.080	8.106	0.33	14.562	14.609	0.32	95.078	95.105	0.03
ship_001	247.141	247.815	0.27	156.082	156.503	0.27	1,778.494	1,779.100	0.03
shipsec5	44.174	44.254	0.18	26.574	26.622	0.18	206.376	206.447	0.03
G3_circuit	213.697	214.191	0.23	71.989	72.150	0.22	333.831	334.158	0.10
hood	112.940	113.170	0.20	47.689	47.784	0.20	149.683	149.778	0.06
crankseg_1	14.277	14.299	0.15	4.782	4.789	0.15	23.146	23.152	0.02

Error correction overhead

Tables E.13, E.14, E.15, and E.16 show the iteration overhead required for convergence to a correct result in case of single-bit flip error injections. Experiments using rounding error thresholds τ that lead to false positive error detections in error-free executions are indicated by *n/a*.

▼ **Table E.13** — Average iteration overhead for error correction in case of one single-bit flip error injection with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	**No preconditioner**					**Jacobi preconditioner**					**ICC preconditioner**				
$\tau :=$	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]
nos3	9.9	9.9	9.9	8.8	9.6	11.4	8.1	8.1	8.3	5.7	13.1	12.0	11.9	9.4	9.6
bcsstk10	8.3	9.6	9.5	9.9	9.3	9.6	8.3	8.1	5.2	5.0	15.2	10.8	11.1	9.6	8.0
msc01050	n/a	n/a	7.8	3.6	3.0	n/a	n/a	2.1	4.5	13.0	n/a	n/a	9.1	8.7	9.8
bcsstk21	3.5	6.9	7.7	6.2	6.5	8.2	6.1	3.3	3.7	2.6	6.3	6.1	3.9	3.6	2.9
bcsstk11	n/a	n/a	11.5	12.0	9.7	n/a	n/a	13.6	15.6	10.0	n/a	n/a	8.0	9.2	8.6
nasa2146	1.7	2.1	1.1	1.2	1.3	2.3	2.5	2.4	2.4	2.4	n/a	n/a	6.7	5.8	4.3
sts4098	n/a	n/a	2.7	2.5	3.0	3.0	4.1	5.0	5.0	3.7	7.1	6.3	6.1	6.2	4.6
bcsstk13	n/a	n/a	0.5	0.9	2.5	2.1	2.4	9.3	9.7	5.9	8.1	12.2	8.3	12.8	9.9
msc04515	12.7	8.6	5.0	3.9	3.9	3.7	4.6	7.0	12.1	7.3	n/a	n/a	10.1	9.2	11.3
ex9	n/a	n/a	0.2	1.3	3.4	1.7	1.7	0.8	0.4	1.2	n/a	n/a	3.7	3.3	3.2
bodyy4	0.6	1.0	0.6	0.6	0.6	0.6	0.0	0.0	0.7	0.3	n/a	n/a	0.4	0.5	0.7
bodyy5	0.4	1.7	1.4	1.3	1.1	1.8	1.7	1.6	1.6	1.6	n/a	n/a	0.9	1.3	1.6
bodyy6	n/a	n/a	0.2	0.3	0.1	0.7	0.7	1.1	1.0	1.8	0.8	0.8	1.1	1.2	1.5
Muu	1.8	2.3	2.3	2.4	2.3	0.9	0.9	0.9	0.9	0.9	3.8	3.8	3.8	3.8	3.8
s3rmt3m3	0.2	1.1	10.5	10.5	10.8	5.7	5.7	7.3	11.6	11.7	n/a	n/a	12.8	20.0	26.9
s3rmt3m1	3.1	4.1	3.0	4.1	1.6	6.7	7.7	7.8	7.7	8.8	n/a	n/a	11.1	10.0	13.5
bcsstk28	n/a	n/a	3.2	6.8	12.5	n/a	n/a	5.2	3.1	14.4	n/a	n/a	11.6	12.7	15.8
s3rmq4m1	5.1	3.6	6.9	7.1	4.3	9.9	10.9	10.0	8.0	2.7	n/a	n/a	4.2	4.2	5.3
bcsstk16	3.5	14.7	13.1	13.0	12.6	12.3	12.7	10.5	8.5	7.7	12.3	11.8	10.1	8.3	6.9
Kuu	3.8	3.5	3.5	3.5	2.4	9.6	9.3	9.2	9.3	8.5	n/a	n/a	11.4	11.0	9.8
bcsstk38	n/a	n/a	8.3	4.2	5.4	6.0	7.2	5.6	4.0	4.0	n/a	n/a	0.3	0.4	4.9
msc23052	3.0	1.2	0.5	0.3	0.4	4.5	4.7	3.7	3.6	4.1	n/a	n/a	5.5	4.2	15.2
msc10848	n/a	n/a	2.9	1.8	2.7	n/a	n/a	8.1	9.4	11.7	n/a	n/a	13.6	10.2	10.1
cfd2	5.5	11.8	11.5	11.1	11.3	2.7	2.7	2.7	2.4	3.3	12.1	12.1	11.9	11.7	11.7
nd3k	1.9	1.2	1.2	0.3	1.7	1.9	2.5	2.0	2.0	3.3	9.4	10.2	9.1	7.3	8.4
ship_001	n/a	n/a	4.0	3.1	3.9	n/a	n/a	4.0	2.5	2.5	n/a	n/a	0.2	0.7	4.7
shipsec5	n/a	n/a	0.7	1.4	10.3	n/a	n/a	2.0	1.3	2.5	n/a	n/a	1.1	1.6	7.9
G3_circuit	0.6	0.6	0.4	0.3	0.4	2.8	2.7	2.7	2.6	4.2	n/a	n/a	3.3	2.4	2.4
hood	3.6	2.8	3.9	2.6	7.7	0.8	1.4	1.4	1.5	2.7	3.0	2.5	2.4	2.2	2.6
crankseg_1	n/a	n/a	5.6	5.4	7.3	n/a	n/a	4.8	4.8	6.5	n/a	n/a	9.4	12.2	12.1

▼ **Table E.14** — Average iteration overhead for error correction in case of two single-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

MATRIX NAME	NO PRECONDITIONER					JACOBI PRECONDITIONER					ICC PRECONDITIONER				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	18.3	17.7	19.9	15.2	13.9	19.0	18.9	18.6	19.0	13.8	21.9	22.5	20.2	17.2	14.2
bcsstk10	9.7	9.7	9.8	10.0	9.9	14.1	17.2	13.6	12.7	12.8	22.2	18.2	15.8	15.2	14.4
msc01050	n/a	n/a	8.0	5.6	12.8	n/a	n/a	4.6	9.2	16.1	n/a	n/a	17.3	17.7	18.6
bcsstk21	3.7	8.5	8.0	7.9	7.6	13.1	10.7	8.8	7.9	7.0	12.3	10.7	11.1	11.4	11.0
bcsstk11	n/a	n/a	15.9	14.9	15.1	n/a	n/a	24.8	24.4	24.6	n/a	n/a	12.7	13.1	14.8
nasa2146	2.9	2.8	2.9	3.4	3.8	2.9	3.2	3.1	3.0	3.3	n/a	n/a	8.8	6.7	5.4
sts4098	n/a	n/a	6.1	5.8	5.7	5.3	5.3	7.0	6.8	8.1	7.5	6.4	6.2	6.5	6.8
bcsstk13	n/a	n/a	4.3	4.7	3.2	3.8	9.1	16.2	16.6	15.7	20.2	22.0	19.6	21.9	22.8
msc04515	15.2	10.8	10.4	11.7	8.9	14.1	13.2	10.4	12.3	12.1	n/a	n/a	16.1	15.4	11.7
ex9	n/a	n/a	3.2	4.9	5.3	16.8	8.8	1.7	1.2	1.2	n/a	n/a	8.5	5.1	10.8
bodyy4	0.8	1.5	1.5	1.6	1.6	0.8	0.7	0.7	0.8	0.7	n/a	n/a	0.9	1.3	2.1
bodyy5	0.6	1.7	1.6	1.4	1.3	2.2	2.1	2.1	1.7	1.7	n/a	n/a	1.9	2.3	2.8
bodyy6	n/a	n/a	1.2	1.0	2.5	4.1	3.9	3.0	3.3	3.0	3.1	2.7	2.5	2.6	2.7
Muu	2.4	3.1	3.1	3.4	3.5	3.8	3.8	3.8	3.8	4.1	14.6	15.8	15.8	16.3	17.1
s3rmt3m3	2.0	2.5	13.7	13.6	11.6	13.5	14.0	14.7	17.1	28.0	n/a	n/a	19.6	20.3	31.2
s3rmt3m1	5.2	4.6	4.6	4.6	3.7	12.1	16.1	17.0	23.9	17.5	n/a	n/a	19.6	20.1	22.9
bcsstk28	n/a	n/a	4.6	6.8	17.0	n/a	n/a	9.2	8.0	30.9	n/a	n/a	20.8	21.3	26.0
s3rmq4m1	5.3	6.5	7.0	7.5	8.6	10.8	12.2	14.0	14.9	24.8	n/a	n/a	11.5	10.9	12.1
bcsstk16	3.6	15.5	15.5	15.6	14.4	15.8	14.4	13.7	13.6	12.6	13.2	13.4	13.5	14.0	14.4
Kuu	7.7	6.7	6.8	5.9	5.3	13.4	12.6	12.2	12.6	11.3	n/a	n/a	16.9	15.3	13.7
bcsstk38	n/a	n/a	8.4	6.3	14.8	10.3	10.7	8.8	7.5	8.4	n/a	n/a	15.0	15.4	8.1
msc23052	4.4	3.8	2.2	1.3	0.5	4.7	5.1	5.3	5.3	4.4	n/a	n/a	10.4	9.9	15.8
msc10848	n/a	n/a	3.8	3.1	2.7	n/a	n/a	16.5	18.0	13.4	n/a	n/a	20.6	20.6	17.7
cfd2	6.2	12.3	12.1	11.7	12.0	4.3	4.6	4.0	4.0	3.7	12.4	12.3	12.2	12.2	12.3
nd3k	2.8	4.3	4.2	6.3	1.9	7.0	8.5	8.7	5.9	4.5	10.7	10.8	11.1	10.5	11.1
ship_001	n/a	n/a	4.0	3.9	5.0	n/a	n/a	10.7	10.2	7.1	n/a	n/a	0.5	1.3	5.9
shipsec5	n/a	n/a	3.7	10.8	10.3	n/a	n/a	2.4	1.7	2.6	n/a	n/a	1.6	1.6	12.9
G3_circuit	0.7	1.0	0.9	0.8	0.9	3.6	3.3	3.3	3.3	5.2	n/a	n/a	3.9	2.6	3.5
hood	5.7	8.3	9.5	5.5	8.6	3.4	2.4	2.4	2.3	3.0	4.1	4.0	3.9	4.0	4.9
crankseg_1	n/a	n/a	6.1	6.0	10.9	n/a	n/a	9.6	7.5	6.9	n/a	n/a	24.5	27.7	23.6

▼ **Table E.15** — Average iteration overhead for error correction in case of five single-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix	No					Jacobi					ICC				
name	preconditioner					preconditioner					preconditioner				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	23.8	27.1	26.5	23.9	19.5	26.5	26.8	26.6	25.5	25.0	27.5	27.6	29.7	29.3	28.2
bcsstk10	11.5	12.9	12.4	11.5	13.8	21.6	22.4	21.3	22.2	20.5	28.4	27.3	25.3	24.2	22.8
msc01050	n/a	n/a	9.2	5.7	14.9	n/a	n/a	6.7	14.3	16.9	n/a	n/a	29.8	36.2	35.4
bcsstk21	4.5	11.4	9.6	12.4	11.0	21.3	20.2	18.9	19.4	18.5	18.5	20.5	21.4	20.8	22.3
bcsstk11	n/a	n/a	23.5	20.3	21.8	n/a	n/a	29.3	29.6	29.3	n/a	n/a	17.5	17.4	20.5
nasa2146	3.9	4.6	5.3	5.5	6.0	4.6	4.2	4.5	5.5	8.8	n/a	n/a	11.1	9.4	8.4
sts4098	n/a	n/a	8.3	7.2	9.8	6.0	8.2	12.8	13.3	19.7	12.9	13.7	13.4	14.4	15.8
bcsstk13	n/a	n/a	6.6	10.8	16.4	8.0	12.2	22.1	24.0	27.0	22.4	24.8	35.6	37.7	34.7
msc04515	15.7	13.2	12.1	12.1	12.9	14.5	13.9	20.0	19.4	12.6	n/a	n/a	24.7	22.5	28.7
ex9	n/a	n/a	6.6	10.6	7.2	20.2	14.4	2.8	2.4	4.2	n/a	n/a	9.3	9.8	13.2
bodyy4	1.0	1.9	1.8	2.3	3.0	0.9	0.8	0.8	1.5	1.3	n/a	n/a	1.0	1.7	3.9
bodyy5	0.8	2.5	2.7	2.6	2.3	2.4	2.5	2.6	2.7	2.7	n/a	n/a	8.4	11.0	12.8
bodyy6	n/a	n/a	1.4	2.9	2.9	6.3	6.7	6.0	5.3	5.7	6.4	6.6	6.0	5.5	5.2
Muu	2.5	4.2	4.4	4.4	4.7	8.2	8.2	8.2	8.5	8.8	16.7	16.7	17.1	17.1	17.5
s3rmt3m3	3.7	4.8	16.9	16.9	19.5	22.1	18.6	17.8	25.2	28.4	n/a	n/a	23.5	21.3	32.5
s3rmt3m1	11.3	14.0	11.7	10.4	12.0	24.4	26.8	22.8	28.4	29.9	n/a	n/a	25.3	22.5	36.7
bcsstk28	n/a	n/a	6.7	10.0	22.4	n/a	n/a	14.3	10.6	31.4	n/a	n/a	35.1	26.3	26.7
s3rmq4m1	14.4	18.7	18.9	19.5	18.0	17.3	23.3	20.7	25.1	26.4	n/a	n/a	23.5	18.9	23.3
bcsstk16	6.6	21.4	22.0	20.8	20.8	18.5	17.4	15.9	17.1	16.2	20.5	16.6	15.8	17.4	18.0
Kuu	12.2	10.6	10.4	10.0	9.5	17.8	16.3	16.0	15.2	14.9	n/a	n/a	19.5	20.4	19.8
bcsstk38	n/a	n/a	10.6	8.4	27.1	20.0	24.9	19.0	18.8	18.2	n/a	n/a	19.9	15.4	17.5
msc23052	5.2	5.0	2.3	1.4	2.1	10.5	9.1	8.1	8.4	9.2	n/a	n/a	10.7	10.3	17.7
msc10848	n/a	n/a	7.1	10.7	9.9	n/a	n/a	21.2	24.3	24.1	n/a	n/a	21.0	22.8	23.9
cfd2	6.4	12.4	12.1	11.7	12.3	4.8	4.8	4.6	4.3	4.2	12.6	12.7	12.5	12.5	12.8
nd3k	6.2	7.2	7.9	8.8	7.0	9.9	10.8	10.3	7.7	7.2	13.3	12.0	12.9	12.5	12.1
ship_001	n/a	n/a	8.2	5.9	6.0	n/a	n/a	23.6	16.7	12.0	n/a	n/a	0.8	2.1	16.7
shipsec5	n/a	n/a	6.9	10.8	10.5	n/a	n/a	4.7	3.1	2.8	n/a	n/a	1.8	2.2	18.5
G3_circuit	1.3	1.6	1.5	1.6	3.0	4.8	4.2	3.3	3.8	9.5	n/a	n/a	5.2	5.0	5.5
hood	14.8	13.3	14.8	10.2	9.7	5.5	4.1	3.8	3.2	5.2	5.5	4.6	4.1	4.3	5.6
crankseg_1	n/a	n/a	9.2	9.9	12.7	n/a	n/a	15.4	15.2	15.3	n/a	n/a	27.1	28.8	28.7

▼ **Table E.16** — Average iteration overhead for error correction in case of ten single-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	No preconditioner					Jacobi preconditioner					ICC preconditioner				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	24.6	27.3	27.7	25.0	25.6	29.8	30.0	27.3	29.1	29.6	27.7	33.0	29.7	29.5	31.0
bcsstk10	12.8	13.7	14.5	14.7	14.6	25.4	24.9	23.6	23.7	22.7	28.6	28.1	26.5	24.9	24.4
msc01050	n/a	n/a	11.4	5.9	15.6	n/a	n/a	8.3	19.3	20.5	n/a	n/a	36.5	36.7	40.5
bcsstk21	7.0	13.4	17.0	15.7	16.1	22.4	23.3	23.1	23.6	23.6	21.5	23.7	22.5	23.4	23.1
bcsstk11	n/a	n/a	24.2	20.6	25.4	n/a	n/a	31.7	34.1	31.2	n/a	n/a	19.9	17.8	27.4
nasa2146	4.6	5.5	6.4	7.1	6.6	5.3	5.5	5.4	5.9	17.1	n/a	n/a	11.3	10.6	9.8
sts4098	n/a	n/a	10.5	11.6	16.9	7.5	10.1	22.2	20.6	20.5	14.6	19.4	19.1	18.5	21.2
bcsstk13	n/a	n/a	8.9	15.6	17.5	9.0	15.7	25.1	25.5	32.6	24.0	28.0	35.8	38.4	38.7
msc04515	23.8	23.9	21.8	22.6	21.9	15.6	17.9	21.2	20.1	14.9	n/a	n/a	24.8	23.3	35.9
ex9	n/a	n/a	8.8	11.1	8.0	23.3	15.0	4.6	3.4	6.2	n/a	n/a	9.5	10.4	19.4
bodyy4	1.5	3.7	3.5	3.6	3.3	2.3	1.4	1.5	1.7	3.2	n/a	n/a	1.1	2.0	6.5
bodyy5	1.0	4.2	4.0	4.2	4.6	2.8	3.1	3.1	3.3	3.7	n/a	n/a	17.5	12.7	14.9
bodyy6	n/a	n/a	2.8	3.0	4.6	9.6	9.1	8.5	8.9	9.3	8.9	9.7	9.1	10.1	9.8
Muu	5.8	6.9	6.9	6.9	6.8	17.4	17.4	17.4	17.4	17.4	27.1	27.1	35.4	35.4	35.4
s3rmt3m3	9.5	9.4	24.4	21.7	21.1	25.8	27.7	18.0	35.7	29.4	n/a	n/a	24.8	24.9	34.4
s3rmt3m1	11.7	14.1	14.6	16.2	20.2	26.3	28.6	30.7	28.9	30.9	n/a	n/a	25.6	24.9	40.5
bcsstk28	n/a	n/a	8.9	13.4	28.3	n/a	n/a	19.8	16.6	32.7	n/a	n/a	36.5	27.1	27.0
s3rmq4m1	24.1	28.2	25.9	24.2	28.7	28.0	26.7	22.5	28.1	28.2	n/a	n/a	24.8	19.7	34.4
bcsstk16	7.0	25.2	24.9	22.2	26.8	22.6	22.8	22.3	21.4	23.0	23.0	22.0	20.6	22.0	22.7
Kuu	14.0	14.9	14.2	12.6	13.0	18.1	18.8	17.4	18.7	16.0	n/a	n/a	21.9	20.8	20.5
bcsstk38	n/a	n/a	12.8	15.1	31.6	23.8	29.5	23.3	24.6	23.2	n/a	n/a	30.3	33.6	19.4
msc23052	5.8	5.0	3.9	2.5	3.1	11.2	12.3	12.7	12.2	11.1	n/a	n/a	11.7	10.5	17.8
msc10848	n/a	n/a	8.2	10.9	10.8	n/a	n/a	27.9	31.0	27.9	n/a	n/a	31.9	28.1	32.9
cfd2	6.9	12.5	12.4	11.9	12.3	5.6	5.5	5.4	5.0	6.1	12.7	12.8	12.5	12.6	13.6
nd3k	10.6	9.4	8.2	10.0	9.0	13.5	12.3	10.5	7.7	10.2	16.5	14.0	14.8	16.0	13.1
ship_001	n/a	n/a	10.4	10.6	12.6	n/a	n/a	25.9	25.3	24.6	n/a	n/a	1.0	3.9	17.5
shipsec5	n/a	n/a	8.8	11.8	10.5	n/a	n/a	8.0	6.5	8.5	n/a	n/a	2.0	5.0	18.6
G3_circuit	2.1	2.6	2.0	3.0	6.4	5.3	4.9	5.0	5.7	11.9	n/a	n/a	5.3	5.5	7.3
hood	19.2	16.3	15.4	14.2	10.6	5.8	4.7	4.7	4.9	9.0	6.0	5.8	5.6	6.1	5.8
crankseg_1	n/a	n/a	11.5	10.6	23.0	n/a	n/a	21.6	21.3	19.6	n/a	n/a	28.6	29.7	33.7

Tables E.17, E.18, E.19, and E.20 show the iteration overhead required for convergence to a correct result in case of multi-bit flip error injections. Experiments using rounding error thresholds τ that lead to false positive error detections in error-free executions are indicated by *n/a*.

▼ **Table E.17** — Average iteration overhead for error correction in case of one multi-bit flip error injection with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	No preconditioner					Jacobi preconditioner					ICC preconditioner				
$\tau :=$	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]	10^{-10} [%]	10^{-9} [%]	10^{-8} [%]	10^{-7} [%]	10^{-6} [%]
nos3	12.0	12.0	11.0	11.0	9.9	7.0	7.4	7.9	8.0	10.2	12.5	11.5	9.0	9.0	6.7
bcsstk10	9.4	9.5	9.2	9.1	8.9	9.1	7.7	6.4	5.0	6.6	18.9	10.5	8.9	10.1	7.7
msc01050	n/a	n/a	4.6	4.7	12.3	n/a	n/a	6.1	4.0	18.4	n/a	n/a	9.9	10.7	10.5
bcsstk21	0.9	7.3	8.2	7.0	7.2	8.9	6.9	5.0	3.0	5.7	7.0	6.3	4.8	4.4	4.0
bcsstk11	n/a	n/a	11.0	11.4	10.3	n/a	n/a	13.0	15.3	12.8	n/a	n/a	9.7	9.3	9.2
nasa2146	2.6	2.0	1.6	1.4	1.2	2.1	2.4	2.3	2.3	2.4	n/a	n/a	7.9	5.7	4.3
sts4098	3.5	3.4	2.7	1.9	2.6	3.4	4.4	4.8	4.8	4.0	6.7	6.4	6.1	5.9	5.0
bcsstk13	12.4	11.3	10.3	8.4	7.7	11.1	12.4	12.5	10.5	7.0	10.3	7.5	6.9	6.4	4.5
msc04515	10.4	8.3	5.2	4.1	5.8	3.6	8.4	8.1	11.0	13.0	n/a	n/a	11.9	12.7	10.7
ex9	n/a	n/a	0.4	0.7	1.2	n/a	n/a	0.6	0.7	1.1	n/a	n/a	4.7	5.1	5.4
bodyy4	1.1	1.2	0.7	0.5	0.7	0.7	0.7	0.7	0.5	0.7	n/a	n/a	0.7	0.7	0.6
bodyy5	0.2	1.6	1.4	1.2	1.0	1.7	1.6	1.6	1.6	1.5	n/a	n/a	1.1	1.4	1.5
bodyy6	n/a	n/a	0.3	0.2	0.2	1.0	1.0	1.0	1.1	1.1	0.9	0.6	0.6	0.8	0.8
Muu	2.6	2.4	2.5	2.5	2.4	0.8	0.8	0.8	0.8	3.4	4.4	4.4	4.4	4.4	4.4
s3rmt3m3	0.6	0.2	11.4	10.0	10.2	15.3	12.4	7.5	10.3	19.9	n/a	n/a	24.6	25.7	27.8
s3rmt3m1	0.9	1.2	0.8	3.0	1.5	15.7	13.5	10.3	10.6	12.9	n/a	n/a	13.8	13.9	15.0
bcsstk28	n/a	n/a	4.7	5.7	13.0	n/a	n/a	6.8	6.6	14.6	n/a	n/a	15.2	14.9	14.6
s3rmq4m1	3.5	2.7	4.1	4.9	4.7	10.3	11.7	13.3	10.3	7.3	n/a	n/a	5.3	5.2	4.7
bcsstk16	4.2	13.9	12.5	12.6	10.5	11.7	12.1	8.9	8.1	7.7	12.0	11.0	9.8	6.3	6.5
Kuu	3.8	3.5	3.5	3.5	2.4	9.5	9.2	9.1	9.3	8.4	n/a	n/a	11.1	11.4	10.2
bcsstk38	n/a	n/a	1.8	3.0	2.3	11.0	9.6	5.1	3.7	3.7	n/a	n/a	0.1	0.1	0.0
msc23052	0.0	0.0	0.4	1.2	0.2	4.8	4.9	3.8	3.5	4.7	n/a	n/a	6.9	6.9	6.3
msc10848	n/a	n/a	1.0	3.0	12.3	n/a	n/a	8.2	11.3	11.3	n/a	n/a	15.4	12.1	12.0
cfd2	6.8	12.0	12.0	11.8	11.5	2.8	2.8	2.8	2.5	2.5	12.2	12.4	12.2	11.9	11.9
nd3k	0.4	0.5	0.5	2.1	0.0	1.9	2.7	1.9	1.8	4.5	9.3	8.9	8.0	9.1	7.9
ship_001	n/a	n/a	0.4	0.1	2.2	n/a	n/a	2.7	2.4	2.0	n/a	n/a	1.0	1.0	1.2
shipsec5	n/a	n/a	0.1	0.1	1.3	n/a	n/a	1.9	1.3	1.3	n/a	n/a	2.0	2.1	1.5
G3_circuit	0.7	0.9	0.7	0.7	0.4	2.9	2.8	2.8	3.2	3.4	n/a	n/a	3.5	3.0	2.9
hood	3.6	3.1	3.2	2.6	2.1	1.7	1.4	1.5	1.7	4.0	3.0	2.9	2.3	2.1	2.1
crankseg_1	n/a	n/a	0.4	1.0	7.6	n/a	n/a	4.8	4.8	6.4	n/a	n/a	12.1	12.8	13.0

▼ **Table E.18** — Average iteration overhead for error correction in case of two multi-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	No preconditioner					Jacobi preconditioner					ICC preconditioner				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	19.9	21.5	20.3	15.9	12.7	23.2	19.5	19.2	15.9	14.7	21.1	21.7	20.4	18.5	15.5
bcsstk10	10.0	9.9	10.2	9.4	9.3	17.9	16.3	12.9	12.1	12.2	19.9	16.3	13.4	16.2	14.1
msc01050	n/a	n/a	8.2	9.9	13.3	n/a	n/a	7.0	10.2	21.0	n/a	n/a	18.4	18.7	16.6
bcsstk21	1.5	7.3	8.7	8.8	8.6	14.4	12.1	10.4	8.2	8.8	13.5	12.0	12.1	13.0	9.7
bcsstk11	n/a	n/a	18.1	16.7	16.8	n/a	n/a	21.8	21.1	21.7	n/a	n/a	12.9	13.6	13.1
nasa2146	2.8	2.7	2.7	3.3	3.6	2.5	3.0	2.9	2.8	2.9	n/a	n/a	8.8	6.7	5.5
sts4098	8.8	7.9	4.7	5.5	3.2	3.5	5.2	6.2	6.1	7.3	7.4	6.5	6.2	6.2	6.2
bcsstk13	17.2	17.2	16.1	12.8	16.8	14.4	15.1	22.8	17.1	16.7	21.3	16.2	23.6	21.9	26.3
msc04515	13.8	8.3	8.1	10.2	6.5	6.1	10.1	9.2	16.5	14.2	n/a	n/a	15.1	15.2	14.5
ex9	n/a	n/a	0.7	1.2	1.9	n/a	n/a	1.4	1.4	1.3	n/a	n/a	7.8	8.3	9.4
bodyy4	1.6	1.4	1.5	1.4	1.3	0.8	0.8	0.9	0.7	0.8	n/a	n/a	2.5	2.5	1.9
bodyy5	0.5	1.7	1.5	1.2	1.1	2.0	2.0	1.9	1.6	1.6	n/a	n/a	1.6	2.2	2.6
bodyy6	n/a	n/a	1.1	1.3	2.1	4.1	4.4	3.3	3.3	3.5	0.9	0.8	1.0	1.4	1.7
Muu	4.9	4.8	4.8	4.9	5.0	3.1	3.1	3.1	3.1	5.5	13.9	15.1	15.1	15.5	16.3
s3rmt3m3	2.7	2.2	12.0	11.2	18.2	17.0	15.2	15.4	18.9	20.5	n/a	n/a	26.7	26.6	28.8
s3rmt3m1	4.2	3.6	3.2	3.7	4.3	15.8	14.7	15.2	21.5	21.2	n/a	n/a	24.4	24.1	22.2
bcsstk28	n/a	n/a	9.0	9.9	15.8	n/a	n/a	13.9	11.2	17.5	n/a	n/a	24.5	24.4	26.1
s3rmq4m1	4.1	4.2	6.2	6.2	7.5	15.0	18.0	16.1	17.6	23.9	n/a	n/a	11.5	11.7	11.6
bcsstk16	4.4	14.8	14.7	14.8	13.6	14.0	13.1	13.1	13.0	12.2	12.6	12.7	12.8	12.3	13.8
Kuu	10.5	7.7	7.2	6.9	6.6	13.7	13.8	12.5	13.3	12.5	n/a	n/a	16.2	15.8	15.3
bcsstk38	n/a	n/a	7.2	7.3	3.4	13.7	13.7	8.6	6.8	7.9	n/a	n/a	15.0	15.0	16.0
msc23052	0.3	0.3	0.6	1.3	1.3	6.5	4.9	6.5	6.4	5.6	n/a	n/a	10.2	10.2	9.9
msc10848	n/a	n/a	5.1	7.1	16.3	n/a	n/a	19.6	16.6	14.8	n/a	n/a	23.7	22.0	19.3
cfd2	7.1	12.4	12.2	11.9	12.2	4.8	4.7	4.1	4.2	4.2	12.4	12.5	12.4	12.2	12.6
nd3k	5.0	5.7	4.8	5.5	4.7	5.6	6.5	6.1	6.0	4.9	12.2	12.9	11.3	9.7	7.9
ship_001	n/a	n/a	0.4	0.9	3.0	n/a	n/a	5.6	3.5	2.7	n/a	n/a	2.2	2.1	1.7
shipsec5	n/a	n/a	0.3	0.7	3.1	n/a	n/a	2.3	3.0	2.9	n/a	n/a	2.7	2.8	1.6
G3_circuit	1.1	0.9	1.0	1.2	1.2	3.7	3.3	3.3	3.4	5.1	n/a	n/a	3.7	4.1	3.7
hood	6.6	6.6	6.9	5.5	3.9	2.7	2.4	2.2	1.7	4.0	4.0	3.8	4.0	4.3	4.9
crankseg_1	n/a	n/a	1.6	2.0	13.3	n/a	n/a	10.3	8.1	6.9	n/a	n/a	22.8	24.1	22.3

▼ **Table E.19** — Average iteration overhead for error correction in case of five multi-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	No preconditioner					Jacobi preconditioner					ICC preconditioner				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	24.8	25.6	27.3	25.1	19.7	27.1	26.1	28.4	26.8	24.2	27.8	27.4	29.4	29.5	29.6
bcsstk10	12.5	12.5	12.4	12.5	13.1	26.1	24.4	23.3	21.7	20.7	27.8	25.8	24.7	24.9	25.9
msc01050	n/a	n/a	18.4	19.6	24.6	n/a	n/a	9.9	19.0	26.7	n/a	n/a	28.0	35.9	33.5
bcsstk21	6.0	11.4	11.6	12.5	10.8	23.7	20.4	19.4	20.7	17.7	22.2	19.8	20.2	20.9	21.6
bcsstk11	n/a	n/a	19.7	20.0	20.8	n/a	n/a	27.9	28.5	31.2	n/a	n/a	18.0	18.4	18.5
nasa2146	5.7	5.9	5.4	5.5	6.5	4.7	4.3	4.3	5.6	5.9	n/a	n/a	11.3	9.5	8.3
sts4098	9.8	9.2	5.4	7.3	7.3	6.0	7.7	13.6	13.7	12.7	16.6	13.7	13.4	13.6	14.9
bcsstk13	24.0	21.9	17.2	30.5	23.8	14.5	18.1	28.5	30.3	34.3	30.9	34.2	27.7	30.3	30.9
msc04515	20.5	15.7	15.3	13.6	11.2	22.9	26.9	20.0	25.9	25.7	n/a	n/a	27.6	28.5	26.8
ex9	n/a	n/a	1.1	1.6	3.4	n/a	n/a	3.6	3.0	4.2	n/a	n/a	10.6	11.6	14.4
bodyy4	2.1	1.9	1.9	2.5	3.1	0.9	0.8	1.0	1.3	1.5	n/a	n/a	3.8	3.6	3.4
bodyy5	0.6	2.5	2.5	2.6	3.1	2.9	2.9	2.7	2.8	2.7	n/a	n/a	6.6	6.7	8.1
bodyy6	n/a	n/a	1.8	2.4	3.7	6.3	6.7	6.1	5.3	6.0	6.4	6.4	6.1	4.8	5.7
Muu	6.8	6.8	6.9	6.8	6.9	7.3	7.3	7.3	7.6	7.8	16.7	16.7	17.1	17.1	17.5
s3rmt3m3	3.3	4.3	13.4	17.0	20.8	20.8	24.2	21.7	24.2	28.8	n/a	n/a	32.4	32.9	33.6
s3rmt3m1	11.5	9.6	9.8	7.6	10.8	25.6	22.8	21.6	29.8	26.0	n/a	n/a	35.4	35.6	37.0
bcsstk28	n/a	n/a	11.9	13.4	23.4	n/a	n/a	20.5	14.3	28.7	n/a	n/a	31.5	30.7	31.3
s3rmq4m1	20.6	18.0	16.1	16.5	13.0	20.0	29.4	25.1	22.3	28.3	n/a	n/a	21.7	22.3	21.2
bcsstk16	6.0	21.6	22.8	21.0	20.9	18.6	18.5	16.3	16.5	15.9	20.6	16.1	14.9	16.0	16.5
Kuu	12.5	10.3	10.0	9.6	9.3	16.9	16.0	16.3	14.2	14.7	n/a	n/a	20.7	20.9	19.4
bcsstk38	n/a	n/a	19.8	22.2	28.4	20.2	26.4	20.7	21.6	18.9	n/a	n/a	16.4	16.1	16.5
msc23052	2.7	0.8	1.1	2.1	2.1	10.0	11.0	10.6	10.5	9.1	n/a	n/a	13.3	13.5	13.2
msc10848	n/a	n/a	8.0	11.2	24.9	n/a	n/a	23.3	22.6	25.3	n/a	n/a	28.5	29.0	24.6
cfd2	7.1	12.5	12.6	12.0	12.5	5.1	4.9	4.4	4.2	4.3	12.8	12.7	12.8	12.6	13.1
nd3k	5.4	6.6	5.1	7.3	5.9	8.5	8.1	6.2	6.4	7.1	13.6	14.0	11.8	12.7	11.7
ship_001	n/a	n/a	3.0	4.0	19.3	n/a	n/a	22.6	20.3	17.2	n/a	n/a	2.4	2.2	2.1
shipsec5	n/a	n/a	1.3	1.9	4.0	n/a	n/a	4.2	3.3	3.8	n/a	n/a	3.0	3.2	2.3
G3_circuit	2.0	2.3	1.7	1.7	2.9	4.5	4.4	3.6	4.1	11.1	n/a	n/a	4.3	4.3	4.6
hood	15.3	10.8	9.3	9.9	6.7	4.4	3.8	3.4	3.6	4.5	5.0	4.3	4.3	5.4	5.5
crankseg_1	n/a	n/a	3.2	3.9	13.7	n/a	n/a	16.6	15.7	15.3	n/a	n/a	24.3	25.4	24.2

▼ **Table E.20** — Average iteration overhead for error correction in case of ten multi-bit flip error injections with respect to different $\tau \in [10^{-10}, 10^{-6}]$.

Matrix name	No preconditioner					Jacobi preconditioner					ICC preconditioner				
$\tau :=$	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}	10^{-10}	10^{-9}	10^{-8}	10^{-7}	10^{-6}
	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]	[%]
nos3	27.7	26.6	28.3	27.8	24.0	28.9	28.8	28.8	27.8	27.6	29.1	31.7	30.1	33.5	30.3
bcsstk10	17.2	17.3	18.0	18.0	20.4	27.0	26.8	24.3	22.3	22.9	31.2	27.8	28.2	24.9	27.2
msc01050	n/a	n/a	19.5	20.6	30.4	n/a	n/a	13.0	19.9	29.8	n/a	n/a	28.6	36.4	40.4
bcsstk21	10.1	13.9	16.4	16.5	16.0	24.1	22.3	22.1	20.8	22.2	23.0	22.7	22.9	23.0	21.8
bcsstk11	n/a	n/a	23.2	24.1	22.9	n/a	n/a	28.7	29.1	37.1	n/a	n/a	27.0	27.5	27.2
nasa2146	5.8	6.6	6.1	6.4	7.5	5.3	5.3	5.6	6.4	6.4	n/a	n/a	11.7	9.7	9.9
sts4098	13.7	14.1	14.2	15.6	23.5	8.0	9.4	19.5	21.2	20.7	21.4	20.4	19.5	18.3	20.7
bcsstk13	25.6	23.0	21.4	33.1	24.8	15.3	21.6	29.8	32.1	38.4	37.7	34.3	29.5	30.9	33.3
msc04515	29.9	26.4	26.3	24.3	23.8	26.9	28.9	21.6	26.2	27.0	n/a	n/a	36.3	37.6	36.9
ex9	n/a	n/a	1.6	2.0	6.1	n/a	n/a	5.8	4.0	6.8	n/a	n/a	15.0	14.9	17.1
bodyy4	3.8	3.6	3.6	3.5	3.4	1.4	1.4	1.3	1.4	1.7	n/a	n/a	7.0	7.3	6.9
bodyy5	1.9	3.9	4.0	4.2	5.1	3.2	3.1	3.2	3.4	3.8	n/a	n/a	23.8	24.2	28.6
bodyy6	n/a	n/a	2.7	2.5	4.5	9.2	8.6	10.7	9.0	9.7	8.4	8.1	8.1	8.4	8.9
Muu	8.5	9.0	9.1	12.4	12.4	21.0	21.0	21.0	21.0	21.0	27.4	28.6	28.9	33.1	38.1
s3rmt3m3	9.9	9.2	22.3	22.2	22.0	29.3	30.0	26.0	33.4	33.8	n/a	n/a	33.3	34.2	36.3
s3rmt3m1	12.9	11.5	11.7	12.1	19.4	31.1	30.5	26.8	31.8	27.6	n/a	n/a	38.8	39.7	40.0
bcsstk28	n/a	n/a	18.0	20.3	29.2	n/a	n/a	23.6	22.2	30.6	n/a	n/a	33.4	31.7	31.8
s3rmq4m1	22.6	22.3	22.2	24.2	25.7	22.0	31.1	29.7	23.1	34.2	n/a	n/a	30.4	30.8	32.6
bcsstk16	7.5	24.7	25.2	26.0	26.3	22.6	22.8	21.7	22.4	23.3	23.3	22.1	21.1	22.7	22.8
Kuu	16.4	16.6	16.2	15.3	15.5	18.7	19.9	19.2	18.8	19.7	n/a	n/a	21.1	22.2	22.3
bcsstk38	n/a	n/a	23.4	27.1	29.2	20.6	27.4	27.5	26.1	25.2	n/a	n/a	26.6	26.6	30.3
msc23052	8.1	0.8	1.1	6.8	3.7	12.2	13.0	11.7	12.7	12.2	n/a	n/a	14.3	14.3	14.8
msc10848	n/a	n/a	12.6	14.0	28.8	n/a	n/a	27.4	29.2	26.5	n/a	n/a	33.4	29.9	35.5
cfd2	7.6	12.6	12.8	12.2	13.0	5.2	5.1	5.4	5.2	5.5	12.8	12.8	12.9	13.0	13.1
nd3k	8.9	8.3	5.2	8.9	6.6	15.3	13.1	11.2	11.0	8.3	14.9	14.4	15.2	16.0	13.3
ship_001	n/a	n/a	3.9	5.0	26.3	n/a	n/a	22.6	20.7	25.9	n/a	n/a	4.1	4.3	3.3
shipsec5	n/a	n/a	1.7	1.9	10.2	n/a	n/a	7.2	7.2	8.7	n/a	n/a	5.3	5.3	3.7
G3_circuit	2.6	3.2	2.4	2.8	6.0	6.1	5.3	5.0	5.3	11.9	n/a	n/a	9.1	9.5	6.7
hood	15.6	17.1	14.9	15.4	14.3	5.3	5.1	4.5	4.1	6.5	5.6	5.6	5.1	5.9	7.5
crankseg_1	n/a	n/a	4.2	5.0	15.4	n/a	n/a	21.2	20.3	21.7	n/a	n/a	34.0	34.0	27.3

E.4 Conjugate Gradient solvers on Approximate Computing Hardware

Table E.21 shows the solver iterations for executions on precise and approximate hardware.

▼ **Table E.21** — Average number of iterations for executions on precise hardware I_S, average number of iterations for executions on approximate hardware I_{apx}, and resulting iteration overhead O_{apx} to converge to correct results.

MATRIX NAME	NO PRECONDITIONER			JACOBI PRECONDITIONER			ICC PRECONDITIONER		
	$\overline{I_S}$	$\overline{I_{apx}}$	$\overline{O}$	$\overline{I_S}$	$\overline{I_{apx}}$	$\overline{O_{apx}}$	$\overline{I_S}$	$\overline{I_{apx}}$	$\overline{O_{apx}}$
nos3	275	281.4	2.3%	250	253.8	1.5%	142	152.9	7.7%
bcsstk10	3,093	3,095.1	0.1%	1,062	1,178.2	10.9%	554	632.9	14.2%
msc01050	5,656	6,113.5	8.1%	1,359	1,682.4	23.8%	1,481	1,891.7	27.7%
bcsstk21	10,937	10,937.0	0.0%	790	859.6	8.8%	329	383.9	16.7%
bcsstk11	19,259	19,909.1	3.4%	5,741	6,363.7	10.8%	22,993	23,160.5	0.7%
nasa2146	506	559.5	10.6%	415	437.7	5.5%	359	381.2	6.2%
sts4098	15,688	16,149.8	2.9%	604	676.3	12.0%	563	669.5	18.9%
bcsstk13	1,889	1,889.0	0.0%	1,573	1,984.2	26.1%	1,489	1,793.3	20.4%
msc04515	5,631	6,543.0	16.2%	4,827	5,229.2	8.3%	4,384	4,819.6	9.9%
ex9	77,631	80,173.7	3.3%	17,290	17,435.6	0.8%	15,369	15,369.0	0.0%
bodyy4	226	232.5	2.9%	213	215.6	1.2%	284	305.4	7.5%
bodyy5	717	733.6	2.3%	538	546.1	1.5%	1,466	1,546.0	5.5%
bodyy6	2,184	2,226.9	2.0%	1,271	1,307.5	2.9%	1,090	1,201.6	10.2%
Muu	44	44.2	0.5%	17	17.0	0.0%	12	12.1	0.7%
s3rmt3m3	838	859.3	2.5%	15,436	15,603.7	1.1%	10,935	11,728.9	7.3%
s3rmt3m1	76,595	86,385.8	12.8%	11,692	11,759.1	0.6%	65,550	65,550.0	0.0%
bcsstk28	13,776	13,776.0	0.0%	5,142	5,680.7	10.5%	3,138	3,866.8	23.2%
s3rmq4m1	50,410	54,403.8	7.9%	8,070	8,170.8	1.2%	48,955	50,740.8	3.6%
bcsstk16	620	702.6	13.3%	279	313.9	12.5%	225	267.5	18.9%
Kuu	684	704.0	2.9%	545	551.4	1.2%	243	258.0	6.2%
bcsstk38	19,575	20,628.7	5.4%	15,001	16,658.5	11.0%	37,301	37,301.0	0.0%
msc23052	284,012	284,012.0	0.0%	217,329	217,560.0	0.1%	37,100	37,883.7	2.1%
msc10848	110,121	110,121.0	0.0%	5,782	6,043.0	4.5%	5,147	5,571.1	8.2%
cfd2	2,395	2,427.3	1.3%	4,984	5,007.2	0.5%	2,010	2,010.0	0.0%
nd3k	4,214	4,214.0	0.0%	7,509	7,509.0	0.0%	4,338	4,501.2	3.8%
ship_001	96,123	96,123.0	0.0%	59,961	61,899.1	3.2%	86,456	86,456.0	0.0%
shipsec5	8,144	8,144.0	0.0%	4,814	4,953.2	2.9%	7,156	7,156.0	0.0%
G3_circuit	9,391	9,450.8	0.6%	3,070	3,070.0	0.0%	6,231	6,278.5	0.8%
hood	17,592	17,835.3	1.4%	7,299	7,405.1	1.5%	7,295	7,442.5	2.0%
crankseg_1	2,884	2,928.9	1.6%	958	1,003.1	4.7%	742	751.1	1.2%

Energy

Table E.22 shows the energy for executions on precise and approximate hardware.

▼ **Table E.22** — Average energy for executions on precise hardware E_S, average energy for executions on approximate hardware E_{apx}, and resulting energy comparison C_{Energy}.

Matrix name	**No preconditioner**			**Jacobi preconditioner**			**ICC preconditioner**		
	$\overline{E_S}$ [J]	$\overline{E_{apx}}$ [J]	$\overline{C_{Energy}}$ [%]	$\overline{E_S}$ [J]	$\overline{E_{apx}}$ [J]	$\overline{C_{Energy}}$ [%]	$\overline{E_S}$ [J]	$\overline{E_{apx}}$ [J]	$\overline{C_{Energy}}$ [%]
nos3	$4.2 \cdot 10^{-4}$	$2.4 \cdot 10^{-4}$	56.6%	$4.0 \cdot 10^{-4}$	$2.2 \cdot 10^{-4}$	56.5%	$3.8 \cdot 10^{-4}$	$3.1 \cdot 10^{-4}$	80.3%
bcsstk10	$6.0 \cdot 10^{-3}$	$3.3 \cdot 10^{-3}$	54.6%	$2.3 \cdot 10^{-3}$	$1.8 \cdot 10^{-3}$	78.9%	$2.1 \cdot 10^{-3}$	$2.0 \cdot 10^{-3}$	95.7%
msc01050	$8.4 \cdot 10^{-3}$	$8.5 \cdot 10^{-3}$	101.3%	$1.9 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	88.0%	$3.6 \cdot 10^{-3}$	$3.5 \cdot 10^{-3}$	96.6%
bcsstk21	$2.1 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	80.0%	$1.9 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	97.0%	$9.9 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$	110.4%
bcsstk11	$5.7 \cdot 10^{-2}$	$4.1 \cdot 10^{-2}$	71.2%	$1.9 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	91.4%	$1.1 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	98.8%
nasa2146	$3.2 \cdot 10^{-3}$	$1.8 \cdot 10^{-3}$	57.1%	$2.7 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	68.2%	$4.3 \cdot 10^{-3}$	$4.2 \cdot 10^{-3}$	96.5%
sts4098	$8.2 \cdot 10^{-2}$	$8.4 \cdot 10^{-2}$	102.6%	$4.3 \cdot 10^{-3}$	$3.4 \cdot 10^{-3}$	78.6%	$6.9 \cdot 10^{-3}$	$6.7 \cdot 10^{-3}$	97.3%
bcsstk13	$1.4 \cdot 10^{-5}$	$1.4 \cdot 10^{-5}$	100.0%	$1.1 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	105.0%	$2.0 \cdot 10^{-2}$	$2.1 \cdot 10^{-2}$	106.2%
msc04515	$4.5 \cdot 10^{-2}$	$3.8 \cdot 10^{-2}$	83.6%	$4.3 \cdot 10^{-2}$	$4.1 \cdot 10^{-2}$	96.9%	$6.7 \cdot 10^{-2}$	$7.8 \cdot 10^{-2}$	115.4%
ex9	$3.3 \cdot 10^{-1}$	$3.3 \cdot 10^{-1}$	101.5%	$1.4 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	101.3%	$2.3 \cdot 10^{-1}$	$2.3 \cdot 10^{-1}$	98.7%
bodyy4	$3.2 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$	69.3%	$3.4 \cdot 10^{-3}$	$2.1 \cdot 10^{-3}$	61.6%	$6.8 \cdot 10^{-3}$	$6.7 \cdot 10^{-3}$	98.0%
bodyy5	$1.1 \cdot 10^{-2}$	$8.0 \cdot 10^{-3}$	73.8%	$9.3 \cdot 10^{-3}$	$6.7 \cdot 10^{-3}$	72.5%	$3.7 \cdot 10^{-2}$	$3.9 \cdot 10^{-2}$	103.9%
bodyy6	$3.5 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$	86.1%	$2.3 \cdot 10^{-2}$	$1.7 \cdot 10^{-2}$	74.8%	$2.9 \cdot 10^{-2}$	$2.6 \cdot 10^{-2}$	86.6%
Muu	$5.8 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$	27.4%	$2.4 \cdot 10^{-4}$	$8.0 \cdot 10^{-5}$	33.3%	$3.0 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	60.7%
s3rmt3m3	$6.8 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$	34.3%	$3.0 \cdot 10^{-1}$	$2.1 \cdot 10^{-1}$	69.0%	$1.8 \cdot 10^{-1}$	$1.8 \cdot 10^{-1}$	104.4%
s3rmt3m1	$9.3 \cdot 10^{-1}$	$8.1 \cdot 10^{-1}$	86.9%	$2.3 \cdot 10^{-1}$	$1.5 \cdot 10^{-1}$	63.4%	$2.5 \cdot 10^{0}$	$2.5 \cdot 10^{0}$	98.9%
bcsstk28	$3.1 \cdot 10^{-1}$	$2.7 \cdot 10^{-1}$	85.3%	$1.0 \cdot 10^{-1}$	$1.0 \cdot 10^{-1}$	100.9%	$1.2 \cdot 10^{-1}$	$1.2 \cdot 10^{-1}$	101.5%
s3rmq4m1	$6.8 \cdot 10^{-1}$	$6.1 \cdot 10^{-1}$	90.0%	$1.7 \cdot 10^{-1}$	$1.1 \cdot 10^{-1}$	66.6%	$1.6 \cdot 10^{0}$	$1.5 \cdot 10^{0}$	91.4%
bcsstk16	$1.2 \cdot 10^{-2}$	$1.1 \cdot 10^{-2}$	88.5%	$6.7 \cdot 10^{-3}$	$4.9 \cdot 10^{-3}$	73.9%	$1.0 \cdot 10^{-2}$	$8.9 \cdot 10^{-3}$	87.6%
Kuu	$1.7 \cdot 10^{-2}$	$9.6 \cdot 10^{-3}$	57.7%	$1.4 \cdot 10^{-2}$	$5.7 \cdot 10^{-3}$	40.3%	$1.2 \cdot 10^{-2}$	$1.0 \cdot 10^{-2}$	85.2%
bcsstk38	$2.4 \cdot 10^{-1}$	$2.4 \cdot 10^{-1}$	99.6%	$4.5 \cdot 10^{-1}$	$4.1 \cdot 10^{-1}$	89.8%	$9.3 \cdot 10^{-5}$	$9.3 \cdot 10^{-5}$	100.0%
msc23052	$2.5 \cdot 10^{1}$	$2.3 \cdot 10^{1}$	93.7%	$2.1 \cdot 10^{1}$	$1.8 \cdot 10^{1}$	82.9%	$3.1 \cdot 10^{-4}$	$3.0 \cdot 10^{-4}$	96.7%
msc10848	$2.1 \cdot 10^{-4}$	$2.1 \cdot 10^{-4}$	100.0%	$6.1 \cdot 10^{-1}$	$5.2 \cdot 10^{-1}$	85.0%	$1.1 \cdot 10^{0}$	$1.1 \cdot 10^{0}$	99.2%
cfd2	$1.9 \cdot 10^{-1}$	$1.4 \cdot 10^{-1}$	74.2%	$1.0 \cdot 10^{0}$	$5.7 \cdot 10^{-1}$	57.0%	$4.3 \cdot 10^{-1}$	$4.0 \cdot 10^{-1}$	93.1%
nd3k	$9.3 \cdot 10^{-1}$	$4.3 \cdot 10^{-1}$	46.8%	$1.7 \cdot 10^{0}$	$8.5 \cdot 10^{-1}$	49.8%	$1.6 \cdot 10^{0}$	$6.9 \cdot 10^{-1}$	43.7%
ship_001	$5.7 \cdot 10^{-4}$	$5.7 \cdot 10^{-4}$	100.0%	$1.7 \cdot 10^{1}$	$1.5 \cdot 10^{1}$	87.3%	$8.4 \cdot 10^{-4}$	$8.4 \cdot 10^{-4}$	100.0%
shipsec5	$4.8 \cdot 10^{-4}$	$4.8 \cdot 10^{-4}$	100.0%	$1.2 \cdot 10^{0}$	$1.0 \cdot 10^{0}$	80.3%	$6.8 \cdot 10^{-4}$	$6.8 \cdot 10^{-4}$	100.0%
G3_circuit	$8.7 \cdot 10^{0}$	$7.1 \cdot 10^{0}$	81.6%	$3.5 \cdot 10^{0}$	$2.6 \cdot 10^{0}$	75.5%	$9.2 \cdot 10^{0}$	$9.1 \cdot 10^{0}$	99.5%
hood	$6.3 \cdot 10^{0}$	$5.7 \cdot 10^{0}$	90.4%	$5.5 \cdot 10^{0}$	$4.6 \cdot 10^{0}$	84.5%	$1.1 \cdot 10^{1}$	$1.1 \cdot 10^{1}$	100.0%
crankseg_1	$2.5 \cdot 10^{0}$	$2.4 \cdot 10^{0}$	97.3%	$8.4 \cdot 10^{-1}$	$6.1 \cdot 10^{-1}$	72.4%	$1.6 \cdot 10^{0}$	$1.6 \cdot 10^{0}$	100.0%

Table E.23 shows the contribution of the fault tolerance technique within the energy demand for solver executions on approximate hardware.

▼ **Table E.23** — Average energy for executions on approximate hardware E_{apx}, average energy for fault tolerance evaluations E_{FT}, and relative energy contribution of fault tolerance p.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	$\overline{E_{apx}}$ [J]	$\overline{E_{FT}}$ [J]	$\overline{p}$ [%]	$\overline{E_{apx}}$ [J]	$\overline{E_{FT}}$ [J]	$\overline{p}$ [%]	$\overline{E_{apx}}$ [J]	$\overline{E_{FT}}$ [J]	$\overline{p}$ [%]
nos3	$2.4 \cdot 10^{-4}$	$2.2 \cdot 10^{-5}$	9.3%	$2.2 \cdot 10^{-4}$	$2.5 \cdot 10^{-5}$	11.3%	$3.1 \cdot 10^{-4}$	$1.2 \cdot 10^{-5}$	3.9%
bcsstk10	$3.3 \cdot 10^{-3}$	$2.5 \cdot 10^{-4}$	7.7%	$1.8 \cdot 10^{-3}$	$1.4 \cdot 10^{-4}$	7.6%	$2.0 \cdot 10^{-3}$	$5.6 \cdot 10^{-5}$	2.8%
msc01050	$8.5 \cdot 10^{-3}$	$3.5 \cdot 10^{-4}$	4.2%	$1.6 \cdot 10^{-3}$	$1.7 \cdot 10^{-4}$	10.5%	$3.5 \cdot 10^{-3}$	$1.3 \cdot 10^{-4}$	3.8%
bcsstk21	$1.7 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$	11.7%	$1.9 \cdot 10^{-3}$	$2.6 \cdot 10^{-4}$	13.6%	$1.1 \cdot 10^{-3}$	$7.9 \cdot 10^{-5}$	7.3%
bcsstk11	$4.1 \cdot 10^{-2}$	$2.2 \cdot 10^{-3}$	5.5%	$1.7 \cdot 10^{-2}$	$1.0 \cdot 10^{-3}$	6.0%	$1.1 \cdot 10^{-1}$	$2.3 \cdot 10^{-3}$	2.1%
nasa2146	$1.8 \cdot 10^{-3}$	$9.3 \cdot 10^{-5}$	5.1%	$1.9 \cdot 10^{-3}$	$9.5 \cdot 10^{-5}$	5.1%	$4.2 \cdot 10^{-3}$	$6.4 \cdot 10^{-5}$	1.5%
sts4098	$8.4 \cdot 10^{-2}$	$4.0 \cdot 10^{-3}$	4.7%	$3.4 \cdot 10^{-3}$	$2.9 \cdot 10^{-4}$	8.6%	$6.7 \cdot 10^{-3}$	$2.2 \cdot 10^{-4}$	3.2%
bcsstk13	$1.4 \cdot 10^{-5}$	$2.2 \cdot 10^{-7}$	1.6%	$1.2 \cdot 10^{-2}$	$4.6 \cdot 10^{-4}$	3.8%	$2.1 \cdot 10^{-2}$	$4.1 \cdot 10^{-4}$	1.9%
msc04515	$3.8 \cdot 10^{-2}$	$2.3 \cdot 10^{-3}$	6.1%	$4.1 \cdot 10^{-2}$	$2.5 \cdot 10^{-3}$	5.9%	$7.8 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$	2.5%
ex9	$3.3 \cdot 10^{-1}$	$1.0 \cdot 10^{-2}$	3.1%	$1.4 \cdot 10^{-1}$	$5.3 \cdot 10^{-3}$	3.8%	$2.3 \cdot 10^{-1}$	$3.7 \cdot 10^{-3}$	1.6%
bodyy4	$2.3 \cdot 10^{-3}$	$2.0 \cdot 10^{-4}$	8.7%	$2.1 \cdot 10^{-3}$	$2.7 \cdot 10^{-4}$	12.8%	$6.7 \cdot 10^{-3}$	$3.0 \cdot 10^{-4}$	4.5%
bodyy5	$8.0 \cdot 10^{-3}$	$6.8 \cdot 10^{-4}$	8.5%	$6.7 \cdot 10^{-3}$	$7.7 \cdot 10^{-4}$	11.5%	$3.9 \cdot 10^{-2}$	$1.1 \cdot 10^{-3}$	3.0%
bodyy6	$3.0 \cdot 10^{-2}$	$2.0 \cdot 10^{-3}$	6.5%	$1.7 \cdot 10^{-2}$	$1.9 \cdot 10^{-3}$	11.0%	$2.6 \cdot 10^{-2}$	$1.3 \cdot 10^{-3}$	5.0%
Muu	$1.6 \cdot 10^{-4}$	$2.1 \cdot 10^{-5}$	13.1%	$8.0 \cdot 10^{-5}$	$1.0 \cdot 10^{-5}$	12.7%	$1.8 \cdot 10^{-4}$	$6.0 \cdot 10^{-6}$	3.2%
s3rmt3m3	$2.4 \cdot 10^{-3}$	$1.6 \cdot 10^{-4}$	7.0%	$2.1 \cdot 10^{-1}$	$8.9 \cdot 10^{-3}$	4.3%	$1.8 \cdot 10^{-1}$	$2.3 \cdot 10^{-3}$	1.3%
s3rmt3m1	$8.1 \cdot 10^{-1}$	$2.4 \cdot 10^{-2}$	3.0%	$1.5 \cdot 10^{-1}$	$6.9 \cdot 10^{-3}$	4.6%	$2.5 \cdot 10^{0}$	$3.0 \cdot 10^{-2}$	1.2%
bcsstk28	$2.7 \cdot 10^{-1}$	$4.9 \cdot 10^{-3}$	1.9%	$1.0 \cdot 10^{-1}$	$2.7 \cdot 10^{-3}$	2.6%	$1.2 \cdot 10^{-1}$	$2.5 \cdot 10^{-3}$	2.1%
s3rmq4m1	$6.1 \cdot 10^{-1}$	$1.6 \cdot 10^{-2}$	2.6%	$1.1 \cdot 10^{-1}$	$4.7 \cdot 10^{-3}$	4.2%	$1.5 \cdot 10^{0}$	$1.9 \cdot 10^{-2}$	1.3%
bcsstk16	$1.1 \cdot 10^{-2}$	$2.3 \cdot 10^{-4}$	2.1%	$4.9 \cdot 10^{-3}$	$1.7 \cdot 10^{-4}$	3.4%	$8.9 \cdot 10^{-3}$	$1.1 \cdot 10^{-4}$	1.2%
Kuu	$9.6 \cdot 10^{-3}$	$3.7 \cdot 10^{-4}$	3.8%	$5.7 \cdot 10^{-3}$	$3.9 \cdot 10^{-4}$	6.8%	$1.0 \cdot 10^{-2}$	$1.5 \cdot 10^{-4}$	1.4%
bcsstk38	$2.4 \cdot 10^{-1}$	$5.1 \cdot 10^{-3}$	2.2%	$4.1 \cdot 10^{-1}$	$1.3 \cdot 10^{-2}$	3.2%	$9.3 \cdot 10^{-5}$	$8.7 \cdot 10^{-7}$	0.9%
msc23052	$2.3 \cdot 10^{1}$	$4.6 \cdot 10^{-1}$	2.0%	$1.8 \cdot 10^{1}$	$5.1 \cdot 10^{-1}$	2.9%	$3.0 \cdot 10^{-4}$	$2.5 \cdot 10^{-6}$	0.8%
msc10848	$2.1 \cdot 10^{-4}$	$1.2 \cdot 10^{-6}$	0.6%	$5.2 \cdot 10^{-1}$	$7.1 \cdot 10^{-3}$	1.4%	$1.1 \cdot 10^{0}$	$5.0 \cdot 10^{-3}$	0.5%
cfd2	$1.4 \cdot 10^{-1}$	$5.0 \cdot 10^{-3}$	3.5%	$5.7 \cdot 10^{-1}$	$3.3 \cdot 10^{-2}$	5.8%	$4.0 \cdot 10^{-1}$	$1.1 \cdot 10^{-2}$	2.8%
nd3k	$4.3 \cdot 10^{-1}$	$2.5 \cdot 10^{-3}$	0.6%	$8.5 \cdot 10^{-1}$	$6.1 \cdot 10^{-3}$	0.7%	$6.9 \cdot 10^{-1}$	$8.0 \cdot 10^{-5}$	0.0%
ship_001	$5.7 \cdot 10^{-4}$	$3.8 \cdot 10^{-6}$	0.7%	$1.5 \cdot 10^{1}$	$2.3 \cdot 10^{-1}$	1.5%	$8.4 \cdot 10^{-4}$	$3.8 \cdot 10^{-6}$	0.4%
shipsec5	$4.8 \cdot 10^{-4}$	$1.4 \cdot 10^{-5}$	3.0%	$1.0 \cdot 10^{0}$	$5.0 \cdot 10^{-2}$	5.0%	$6.8 \cdot 10^{-4}$	$1.9 \cdot 10^{-5}$	2.9%
G3_circuit	$7.1 \cdot 10^{0}$	$2.8 \cdot 10^{-1}$	4.0%	$2.6 \cdot 10^{0}$	$1.5 \cdot 10^{-1}$	5.9%	$9.1 \cdot 10^{0}$	$2.9 \cdot 10^{-1}$	3.2%
hood	$5.7 \cdot 10^{0}$	$1.5 \cdot 10^{-1}$	2.6%	$4.6 \cdot 10^{0}$	$1.7 \cdot 10^{-1}$	3.6%	$1.1 \cdot 10^{1}$	$1.4 \cdot 10^{-1}$	1.3%
crankseg_1	$2.4 \cdot 10^{0}$	$1.2 \cdot 10^{-2}$	0.5%	$6.1 \cdot 10^{-1}$	$5.6 \cdot 10^{-3}$	0.9%	$1.6 \cdot 10^{0}$	$4.1 \cdot 10^{-3}$	0.3%

Energy Efficiency

Table E.24 shows the energy efficiency gain for executions approximate hardware compared to its precise counterpart.

▼ **Table E.24** — Energy efficiency following Equation 7.6 for executions on precise hardware η_S, energy efficiency for executions on approximate hardware η_{apx}, and resulting energy efficiency gain G_η.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	$\overline{\eta_S}$	$\overline{\eta_{apx}}$	$\overline{G_\eta}$	$\overline{\eta_S}$	$\overline{\eta_{apx}}$	$\overline{G_\eta}$	$\overline{\eta_S}$	$\overline{\eta_{apx}}$	$\overline{G_\eta}$
nos3	$1.1\cdot10^{-1}$	$6.7\cdot10^{-2}$	172.7%	$9.9\cdot10^{-2}$	$5.7\cdot10^{-2}$	174.3%	$5.4\cdot10^{-2}$	$4.7\cdot10^{-2}$	115.7%
bcsstk10	$1.8\cdot10^{1}$	$1.0\cdot10^{1}$	183.1%	$2.5\cdot10^{0}$	$2.1\cdot10^{0}$	114.3%	$1.2\cdot10^{0}$	$1.3\cdot10^{0}$	91.5%
msc01050	$4.7\cdot10^{1}$	$5.2\cdot10^{1}$	91.4%	$2.5\cdot10^{0}$	$2.8\cdot10^{0}$	91.8%	$5.4\cdot10^{0}$	$6.6\cdot10^{0}$	81.0%
bcsstk21	$2.3\cdot10^{2}$	$1.9\cdot10^{2}$	125.1%	$1.5\cdot10^{0}$	$1.6\cdot10^{0}$	94.7%	$3.3\cdot10^{-1}$	$4.2\cdot10^{-1}$	77.6%
bcsstk11	$1.1\cdot10^{3}$	$8.1\cdot10^{2}$	135.8%	$1.1\cdot10^{2}$	$1.1\cdot10^{2}$	98.7%	$2.6\cdot10^{3}$	$2.5\cdot10^{3}$	100.5%
nasa2146	$1.6\cdot10^{0}$	$1.0\cdot10^{0}$	158.4%	$1.1\cdot10^{0}$	$8.2\cdot10^{-1}$	139.0%	$1.6\cdot10^{0}$	$1.6\cdot10^{0}$	97.6%
sts4098	$1.3\cdot10^{3}$	$1.4\cdot10^{3}$	94.6%	$2.6\cdot10^{0}$	$2.3\cdot10^{0}$	113.7%	$3.9\cdot10^{0}$	$4.5\cdot10^{0}$	86.4%
bcsstk13	$2.6\cdot10^{-2}$	$2.6\cdot10^{-2}$	100.0%	$1.8\cdot10^{1}$	$2.4\cdot10^{1}$	75.5%	$3.0\cdot10^{1}$	$3.8\cdot10^{1}$	78.2%
msc04515	$2.5\cdot10^{2}$	$2.5\cdot10^{2}$	103.0%	$2.1\cdot10^{2}$	$2.2\cdot10^{2}$	95.3%	$2.9\cdot10^{2}$	$3.7\cdot10^{2}$	78.8%
ex9	$2.6\cdot10^{4}$	$2.7\cdot10^{4}$	95.4%	$2.4\cdot10^{3}$	$2.4\cdot10^{3}$	97.9%	$3.5\cdot10^{3}$	$3.5\cdot10^{3}$	101.3%
bodyy4	$7.3\cdot10^{-1}$	$5.2\cdot10^{-1}$	140.2%	$7.3\cdot10^{-1}$	$4.6\cdot10^{-1}$	160.2%	$1.9\cdot10^{0}$	$2.0\cdot10^{0}$	94.9%
bodyy5	$7.8\cdot10^{0}$	$5.9\cdot10^{0}$	132.5%	$5.0\cdot10^{0}$	$3.7\cdot10^{0}$	135.9%	$5.5\cdot10^{1}$	$6.0\cdot10^{1}$	91.2%
bodyy6	$7.7\cdot10^{1}$	$6.8\cdot10^{1}$	113.9%	$2.9\cdot10^{1}$	$2.2\cdot10^{1}$	129.9%	$3.2\cdot10^{1}$	$3.1\cdot10^{1}$	104.7%
Muu	$2.5\cdot10^{-2}$	$7.0\cdot10^{-3}$	363.2%	$4.1\cdot10^{-3}$	$1.4\cdot10^{-3}$	300.4%	$3.7\cdot10^{-3}$	$2.2\cdot10^{-3}$	163.8%
s3rmt3m3	$5.7\cdot10^{0}$	$2.0\cdot10^{0}$	284.1%	$4.6\cdot10^{3}$	$3.2\cdot10^{3}$	143.4%	$1.9\cdot10^{3}$	$2.2\cdot10^{3}$	89.3%
s3rmt3m1	$7.1\cdot10^{4}$	$7.0\cdot10^{4}$	102.0%	$2.7\cdot10^{3}$	$1.7\cdot10^{3}$	156.7%	$1.6\cdot10^{5}$	$1.6\cdot10^{5}$	101.1%
bcsstk28	$4.3\cdot10^{3}$	$3.7\cdot10^{3}$	117.3%	$5.2\cdot10^{2}$	$5.8\cdot10^{2}$	89.7%	$3.8\cdot10^{2}$	$4.7\cdot10^{2}$	79.9%
s3rmq4m1	$3.4\cdot10^{4}$	$3.3\cdot10^{4}$	103.0%	$1.4\cdot10^{3}$	$9.3\cdot10^{2}$	148.3%	$8.0\cdot10^{4}$	$7.6\cdot10^{4}$	105.6%
bcsstk16	$7.6\cdot10^{0}$	$7.6\cdot10^{0}$	99.8%	$1.9\cdot10^{0}$	$1.5\cdot10^{0}$	120.3%	$2.3\cdot10^{0}$	$2.4\cdot10^{0}$	96.1%
Kuu	$1.1\cdot10^{1}$	$6.8\cdot10^{0}$	168.5%	$7.8\cdot10^{0}$	$3.2\cdot10^{0}$	245.2%	$3.0\cdot10^{0}$	$2.7\cdot10^{0}$	110.5%
bcsstk38	$4.6\cdot10^{3}$	$4.9\cdot10^{3}$	95.2%	$6.8\cdot10^{3}$	$6.8\cdot10^{3}$	100.3%	$3.5\cdot10^{0}$	$3.5\cdot10^{0}$	100.0%
msc23052	$7.1\cdot10^{6}$	$6.6\cdot10^{6}$	106.7%	$4.6\cdot10^{6}$	$3.8\cdot10^{6}$	120.5%	$1.1\cdot10^{1}$	$1.1\cdot10^{1}$	101.3%
msc10848	$2.3\cdot10^{1}$	$2.3\cdot10^{1}$	100.0%	$3.5\cdot10^{3}$	$3.1\cdot10^{3}$	112.5%	$5.5\cdot10^{3}$	$5.9\cdot10^{3}$	93.1%
cfd2	$4.6\cdot10^{2}$	$3.5\cdot10^{2}$	132.9%	$5.0\cdot10^{3}$	$2.9\cdot10^{3}$	174.6%	$8.7\cdot10^{2}$	$8.1\cdot10^{2}$	107.4%
nd3k	$3.9\cdot10^{3}$	$1.8\cdot10^{3}$	213.8%	$1.3\cdot10^{4}$	$6.4\cdot10^{3}$	200.7%	$6.8\cdot10^{3}$	$3.1\cdot10^{3}$	220.4%
ship_001	$5.4\cdot10^{1}$	$5.4\cdot10^{1}$	100.0%	$1.0\cdot10^{6}$	$9.2\cdot10^{5}$	110.9%	$7.3\cdot10^{1}$	$7.3\cdot10^{1}$	100.0%
shipsec5	$3.9\cdot10^{0}$	$3.9\cdot10^{0}$	100.0%	$6.0\cdot10^{3}$	$4.9\cdot10^{3}$	121.1%	$4.8\cdot10^{0}$	$4.8\cdot10^{0}$	100.0%
G3_circuit	$8.1\cdot10^{4}$	$6.7\cdot10^{4}$	121.7%	$1.1\cdot10^{4}$	$8.0\cdot10^{3}$	132.5%	$5.7\cdot10^{4}$	$5.7\cdot10^{4}$	99.8%
hood	$1.1\cdot10^{5}$	$1.0\cdot10^{5}$	109.1%	$4.0\cdot10^{4}$	$3.4\cdot10^{4}$	116.6%	$7.7\cdot10^{4}$	$7.9\cdot10^{4}$	98.0%
crankseg_1	$7.1\cdot10^{3}$	$7.0\cdot10^{3}$	101.2%	$8.1\cdot10^{2}$	$6.1\cdot10^{2}$	131.8%	$1.2\cdot10^{3}$	$1.2\cdot10^{3}$	98.8%

Utilization of approximation levels

Tables E.25, E.26, and E.27 show the utilization of the available approximation levels of the underling hardware over the course of the solver progress.

▼ **Table E.25** — Utilization of available precisions (i.e. number of precise mantissa bits p) for executions on approximate hardware when no preconditioner is used.

Matrix name	Utilization of p precise mantissa bits										
	52 [%]	47 [%]	42 [%]	37 [%]	32 [%]	27 [%]	22 [%]	17 [%]	12 [%]	7 [%]	2 [%]
nos3	0.00	0.00	0.00	41.15	54.42	0.37	0.00	0.00	0.00	4.06	0.00
bcsstk10	3.00	0.42	0.10	0.00	95.98	0.03	0.06	0.00	0.00	0.38	0.01
msc01050	34.01	36.04	17.79	6.95	2.71	1.21	0.46	0.22	0.27	0.09	0.26
bcsstk21	0.00	0.00	0.00	0.01	40.72	59.13	0.01	0.00	0.01	0.00	0.12
bcsstk11	25.90	70.89	1.37	0.83	0.54	0.32	0.08	0.01	0.01	0.01	0.06
nasa2146	0.00	0.00	13.84	28.06	55.92	0.18	0.00	0.00	0.00	2.00	0.00
sts4098	97.97	0.64	0.34	0.26	0.18	0.14	0.14	0.12	0.10	0.03	0.09
bcsstk13	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
msc04515	5.92	5.04	58.62	18.09	11.98	0.17	0.00	0.00	0.00	0.19	0.00
ex9	95.53	0.99	0.73	0.79	1.45	0.39	0.07	0.01	0.00	0.00	0.03
bodyy4	0.00	0.00	1.52	80.55	11.92	0.50	0.00	0.00	0.00	5.51	0.00
bodyy5	0.00	0.00	77.25	17.31	3.42	0.27	0.00	0.00	0.00	1.76	0.00
bodyy6	3.99	47.40	44.78	1.75	0.61	0.85	0.05	0.00	0.00	0.56	0.00
Muu	0.00	0.00	0.00	0.00	0.00	64.86	0.09	2.61	0.00	2.70	29.73
s3rmt3m3	0.00	0.00	3.11	42.85	35.22	11.33	3.52	0.72	0.32	0.24	2.68
s3rmt3m1	8.04	4.08	68.90	18.88	0.08	0.00	0.00	0.00	0.00	0.02	0.00
bcsstk28	98.08	0.72	0.45	0.16	0.13	0.12	0.10	0.06	0.08	0.06	0.03
s3rmq4m1	0.00	16.83	64.53	18.51	0.10	0.00	0.00	0.00	0.00	0.03	0.00
bcsstk16	20.79	1.30	4.20	67.97	3.31	0.31	0.07	0.01	0.01	1.98	0.07
Kuu	0.00	0.00	9.31	86.48	2.27	0.16	0.00	0.00	0.00	1.77	0.00
bcsstk38	53.09	45.74	0.27	0.14	0.11	0.09	0.09	0.17	0.16	0.01	0.12
msc23052	50.62	49.07	0.12	0.10	0.07	0.01	0.00	0.00	0.00	0.00	0.00
msc10848	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
cfd2	0.00	0.00	0.00	0.00	99.04	0.08	0.00	0.00	0.00	0.88	0.00
nd3k	0.00	8.38	19.42	71.45	0.01	0.28	0.02	0.06	0.03	0.03	0.32
ship_001	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
shipsec5	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
G3_circuit	0.00	0.00	22.94	76.41	0.49	0.01	0.00	0.00	0.00	0.00	0.15
hood	0.00	0.00	0.00	0.01	94.51	5.40	0.01	0.00	0.00	0.07	0.00
crankseg_1	95.36	1.33	0.91	0.57	0.63	0.44	0.15	0.17	0.02	0.02	0.40

▼ **Table E.26** — Utilization of available precisions (i.e. number of precise mantissa bits p) for executions on approximate hardware when the *Jacobi* preconditioner is used.

Matrix name	Utilization of p precise mantissa bits										
	52 [%]	47 [%]	42 [%]	37 [%]	32 [%]	27 [%]	22 [%]	17 [%]	12 [%]	7 [%]	2 [%]
nos3	0.00	0.00	0.00	8.73	81.88	4.50	0.40	0.00	0.00	4.50	0.00
bcsstk10	0.00	9.61	65.30	20.70	3.31	0.09	0.00	0.00	0.00	0.99	0.00
msc01050	59.22	23.32	10.22	3.87	1.49	0.79	0.12	0.22	0.03	0.69	0.02
bcsstk21	0.00	0.00	8.42	34.12	56.08	0.12	0.00	0.00	0.00	1.27	0.00
bcsstk11	0.17	24.51	56.08	17.72	1.22	0.13	0.01	0.00	0.00	0.16	0.00
nasa2146	0.00	6.77	16.94	69.49	3.88	0.32	0.00	0.00	0.00	2.60	0.00
sts4098	2.29	12.05	44.84	33.45	3.25	1.79	0.58	0.00	0.00	1.75	0.00
bcsstk13	6.64	32.39	38.86	18.78	2.14	0.57	0.11	0.00	0.00	0.52	0.00
msc04515	46.32	47.03	5.44	0.75	0.23	0.02	0.00	0.00	0.00	0.21	0.00
ex9	97.34	0.95	0.55	0.66	0.34	0.08	0.01	0.00	0.00	0.08	0.00
bodyy4	0.00	9.95	4.28	67.51	4.70	1.57	3.58	1.93	0.65	0.49	5.35
bodyy5	0.00	5.99	75.76	10.53	2.65	0.14	1.40	0.93	0.20	0.20	2.20
bodyy6	0.00	12.03	32.71	52.04	1.14	0.00	0.62	0.37	0.08	0.08	0.93
Muu	0.00	0.00	0.00	0.00	0.00	14.04	12.28	5.26	5.26	5.26	57.89
s3rmt3m3	3.26	6.60	69.63	20.09	0.13	0.07	0.07	0.00	0.07	0.07	0.00
s3rmt3m1	0.00	0.00	13.21	86.41	0.18	0.09	0.01	0.00	0.00	0.09	0.00
bcsstk28	21.24	70.24	7.56	0.32	0.24	0.19	0.02	0.00	0.00	0.19	0.00
s3rmq4m1	6.51	42.75	39.90	10.26	0.04	0.14	0.14	0.01	0.13	0.14	0.00
bcsstk16	0.00	0.00	10.23	40.62	41.65	3.70	0.31	0.00	0.00	3.49	0.00
Kuu	0.00	0.00	0.00	12.58	85.00	0.26	0.00	0.00	0.00	2.16	0.00
bcsstk38	31.91	41.85	23.53	2.10	0.15	0.17	0.07	0.06	0.07	0.01	0.07
msc23052	6.92	56.55	36.45	0.04	0.01	0.01	0.01	0.00	0.00	0.01	0.00
msc10848	8.59	43.90	43.77	3.25	0.12	0.11	0.08	0.00	0.00	0.18	0.00
cfd2	0.00	0.00	0.00	10.25	89.44	0.03	0.00	0.00	0.00	0.28	0.00
nd3k	0.00	0.00	22.55	76.91	0.15	0.02	0.12	0.06	0.02	0.02	0.17
ship_001	12.11	49.78	37.42	0.52	0.07	0.03	0.02	0.00	0.02	0.02	0.00
shipsec5	2.86	25.57	62.94	6.89	0.55	0.24	0.24	0.24	0.24	0.24	0.00
G3_circuit	0.00	0.00	92.26	6.90	0.02	0.00	0.36	0.03	0.03	0.03	0.36
hood	3.39	49.15	44.93	2.17	0.19	0.01	0.00	0.00	0.00	0.15	0.00
crankseg_1	5.63	31.19	57.05	1.71	0.43	1.45	0.99	0.28	0.13	0.01	1.14

▼ **Table E.27** — Utilization of available precisions (i.e. number of precise mantissa bits p) for executions on approximate hardware when the *Incomplete Cholesky factorization* preconditioner (ICC) is used.

Matrix name	Utilization of p precise mantissa bits										
	52 [%]	47 [%]	42 [%]	37 [%]	32 [%]	27 [%]	22 [%]	17 [%]	12 [%]	7 [%]	2 [%]
nos3	0.00	0.00	0.00	33.44	54.98	1.84	0.11	0.67	0.69	0.69	7.58
bcsstk10	0.00	31.48	48.78	13.41	1.77	0.59	1.43	0.43	0.16	0.16	1.78
msc01050	45.53	21.49	18.59	9.34	2.87	0.57	0.57	0.41	0.06	0.37	0.20
bcsstk21	0.00	0.00	1.26	16.32	46.06	30.56	2.78	0.25	0.00	2.78	0.00
bcsstk11	88.57	11.03	0.11	0.09	0.05	0.05	0.06	0.04	0.00	0.00	0.00
nasa2146	13.98	69.44	1.82	5.11	3.47	3.09	3.09	0.00	0.00	0.00	0.00
sts4098	2.02	31.75	42.86	13.70	3.02	1.78	1.21	1.10	0.68	0.16	1.72
bcsstk13	0.00	5.75	29.32	34.34	27.31	2.73	0.10	0.00	0.00	0.46	0.00
msc04515	21.52	75.25	2.29	0.44	0.24	0.02	0.00	0.00	0.00	0.24	0.00
ex9	95.73	2.15	1.08	0.51	0.13	0.09	0.06	0.08	0.08	0.08	0.00
bodyy4	18.56	46.03	11.99	7.45	6.36	0.52	4.02	1.05	0.00	4.02	0.00
bodyy5	91.04	1.81	1.65	1.37	1.29	0.83	0.48	0.74	0.79	0.00	0.00
bodyy6	2.86	3.23	18.15	49.18	24.31	0.00	0.74	0.30	0.12	0.09	1.01
Muu	0.00	0.00	0.00	0.00	0.00	2.39	9.78	6.52	6.52	6.52	68.26
s3rmt3m3	88.67	5.79	2.61	1.31	0.58	0.38	0.17	0.09	0.18	0.21	0.00
s3rmt3m1	99.84	0.04	0.02	0.01	0.03	0.02	0.02	0.02	0.02	0.00	0.00
bcsstk28	0.00	0.69	1.47	9.15	19.42	39.16	24.41	5.42	0.22	0.07	0.00
s3rmq4m1	6.50	67.82	23.28	2.22	0.04	0.04	0.03	0.03	0.03	0.03	0.00
bcsstk16	0.00	0.00	3.87	64.41	26.86	0.40	0.00	0.00	0.00	4.45	0.00
Kuu	0.00	0.00	42.65	47.74	4.59	0.54	0.01	0.00	0.00	4.46	0.00
bcsstk38	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
msc23052	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
msc10848	12.53	57.54	27.24	1.57	0.38	0.22	0.08	0.13	0.09	0.21	0.00
cfd2	0.00	0.00	0.00	0.00	0.09	98.85	0.00	0.09	0.00	0.00	0.97
nd3k	0.99	0.00	0.00	97.82	0.00	0.00	0.72	0.03	0.03	0.03	0.37
ship_001	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
shipsec5	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
G3_circuit	95.49	3.47	0.39	0.21	0.00	0.20	0.02	0.00	0.00	0.20	0.01
hood	6.70	53.28	39.18	0.51	0.17	0.01	0.00	0.00	0.00	0.15	0.00
crankseg_1	62.22	20.09	7.45	3.93	1.02	0.61	0.74	1.34	1.21	0.24	1.14

Table E.28 shows the minimum, maximum, and average number of precise mantissa bits p in the course of solver executions.

▼ **Table E.28** — Minimum, maximum, and average number of precise mantissa bits p in the course of solver executions.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	min p	avg p	max p	min p	avg p	max p	min p	avg p	max p
nos3	7.00	33.03	37.00	7.00	31.05	37.00	2.00	30.88	37.00
bcsstk10	2.00	32.57	52.00	7.00	40.75	47.00	2.00	41.43	47.00
msc01050	2.00	46.04	52.00	2.00	48.29	52.00	2.00	46.35	52.00
bcsstk21	2.00	29.00	37.00	7.00	34.22	42.00	7.00	30.41	42.00
bcsstk11	2.00	47.95	52.00	7.00	42.15	52.00	17.00	51.37	52.00
nasa2146	7.00	34.28	42.00	7.00	37.52	47.00	22.00	45.19	52.00
sts4098	2.00	51.64	52.00	7.00	39.84	52.00	2.00	41.07	52.00
bcsstk13	52.00	52.00	52.00	7.00	42.84	52.00	7.00	37.25	47.00
msc04515	7.00	40.65	52.00	7.00	48.85	52.00	7.00	47.78	52.00
ex9	2.00	51.33	52.00	7.00	51.68	52.00	7.00	51.54	52.00
bodyy4	7.00	34.78	42.00	2.00	34.71	47.00	7.00	42.60	52.00
bodyy5	7.00	40.14	42.00	2.00	39.96	47.00	12.00	50.36	52.00
bodyy6	7.00	44.29	52.00	2.00	39.24	47.00	2.00	36.86	52.00
Muu	2.00	18.76	27.00	2.00	9.54	27.00	2.00	6.51	27.00
s3rmt3m3	2.00	32.50	42.00	7.00	41.57	52.00	7.00	50.79	52.00
s3rmt3m1	7.00	42.05	52.00	7.00	37.61	42.00	7.00	51.97	52.00
bcsstk28	2.00	51.71	52.00	7.00	47.50	52.00	7.00	27.43	47.00
s3rmq4m1	7.00	41.90	47.00	7.00	44.13	52.00	7.00	45.89	52.00
bcsstk16	2.00	39.63	52.00	7.00	33.97	42.00	7.00	34.48	42.00
Kuu	7.00	36.80	42.00	7.00	32.08	37.00	7.00	37.51	42.00
bcsstk38	2.00	49.41	52.00	2.00	47.06	52.00	52.00	52.00	52.00
msc23052	2.00	49.50	52.00	7.00	45.51	52.00	52.00	52.00	52.00
msc10848	52.00	52.00	52.00	7.00	44.79	52.00	7.00	45.83	52.00
cfd2	7.00	31.78	32.00	7.00	32.44	37.00	2.00	26.75	32.00
nd3k	2.00	38.64	47.00	2.00	38.02	42.00	2.00	36.88	52.00
ship_001	52.00	52.00	52.00	7.00	45.65	52.00	52.00	52.00	52.00
shipsec5	52.00	52.00	52.00	7.00	42.87	52.00	52.00	52.00	52.00
G3_circuit	2.00	38.07	42.00	2.00	41.41	42.00	2.00	51.60	52.00
hood	7.00	31.71	37.00	7.00	44.61	52.00	7.00	45.24	52.00
crankseg_1	2.00	51.20	52.00	2.00	43.01	52.00	2.00	47.45	52.00

E.5 Parameter Evaluation and Estimation

Tables E.29 and E.30 present the validation results of the presented simulation-based parameter evaluation method that were obtained for the underlying operations of the sparse matrix-vector multiplications.

▼ **Table E.29** — Difference between dynamic power results obtained by the simulation-based method and the commercial tool chain with different numbers of precise mantissa bits.

Matrix name	*k* precise mantissa bits										
	52 [mW]	47 [mW]	42 [mW]	37 [mW]	32 [mW]	27 [mW]	22 [mW]	17 [mW]	12 [mW]	7 [mW]	2 [mW]
nos3	0.096	0.071	0.044	0.057	0.066	0.069	0.066	0.067	0.053	0.032	0.019
bcsstk10	0.090	0.117	0.085	0.067	0.064	0.078	0.078	0.069	0.052	0.032	0.018
msc01050	0.102	0.063	0.076	0.074	0.074	0.055	0.081	0.062	0.047	0.032	0.019
bcsstk21	0.058	0.050	0.049	0.052	0.055	0.068	0.071	0.064	0.050	0.029	0.017
bcsstk11	0.068	0.100	0.068	0.053	0.062	0.070	0.077	0.067	0.051	0.031	0.018
nasa2146	0.118	0.112	0.085	0.078	0.082	0.092	0.090	0.075	0.053	0.031	0.019
sts4098	0.097	0.067	0.055	0.063	0.065	0.088	0.082	0.071	0.055	0.032	0.019
bcsstk13	0.057	0.060	0.021	0.035	0.045	0.068	0.074	0.071	0.054	0.032	0.019
msc04515	0.004	0.021	0.023	0.012	0.044	0.070	0.090	0.070	0.054	0.033	0.018
ex9	0.107	0.073	0.064	0.070	0.067	0.078	0.086	0.080	0.053	0.032	0.019
bodyy4	0.110	0.095	0.118	0.099	0.094	0.101	0.098	0.079	0.056	0.034	0.020
bodyy5	0.117	0.092	0.108	0.114	0.099	0.104	0.100	0.080	0.056	0.033	0.020
bodyy6	0.110	0.102	0.111	0.110	0.109	0.109	0.099	0.081	0.057	0.035	0.020
Muu	0.129	0.124	0.106	0.099	0.113	0.119	0.080	0.073	0.053	0.029	0.017
s3rmt3m3	0.110	0.086	0.059	0.059	0.061	0.079	0.082	0.067	0.050	0.030	0.018
s3rmt3m1	0.121	0.130	0.098	0.080	0.080	0.082	0.085	0.074	0.052	0.032	0.018
bcsstk28	0.111	0.102	0.117	0.093	0.092	0.094	0.092	0.075	0.052	0.031	0.018
s3rmq4m1	0.056	0.047	0.039	0.058	0.037	0.021	0.034	0.047	0.044	0.031	0.019
bcsstk16	0.087	0.118	0.127	0.072	0.079	0.062	0.068	0.065	0.046	0.027	0.016
Kuu	0.098	0.091	0.078	0.046	0.020	0.023	0.062	0.065	0.052	0.033	0.019
bcsstk38	0.108	0.090	0.109	0.091	0.082	0.084	0.088	0.074	0.051	0.031	0.019
msc23052	0.082	0.090	0.073	0.094	0.067	0.088	0.046	0.063	0.045	0.034	0.018
msc10848	0.089	0.095	0.093	0.078	0.081	0.087	0.086	0.070	0.050	0.030	0.018
cfd2	0.084	0.125	0.075	0.076	0.075	0.084	0.084	0.070	0.052	0.032	0.019
nd3k	0.125	0.114	0.054	0.053	0.077	0.074	0.078	0.069	0.050	0.031	0.018
ship_001	0.095	0.078	0.061	0.077	0.032	0.077	0.088	0.057	0.048	0.027	0.019
shipsec5	0.121	0.089	0.077	0.069	0.055	0.087	0.074	0.076	0.057	0.029	0.019
G3_circuit	0.123	0.076	0.057	0.047	0.029	0.072	0.071	0.067	0.059	0.031	0.020
hood	0.120	0.128	0.070	0.061	0.046	0.082	0.075	0.071	0.053	0.026	0.019
crankseg_1	0.072	0.122	0.051	0.040	0.022	0.069	0.062	0.058	0.041	0.019	0.018

▼ **Table E.30** — Runtime of simulation-based parameter evaluation with respect to different numbers of precise mantissa bits.

Matrix name	*k* precise mantissa bits										
	52 [s]	47 [s]	42 [s]	37 [s]	32 [s]	27 [s]	22 [s]	17 [s]	12 [s]	7 [s]	2 [s]
nos3	0.96	0.89	0.81	0.73	0.66	0.61	0.57	0.57	0.51	0.48	0.49
bcsstk10	1.34	1.32	1.13	1.03	0.93	0.94	0.78	0.73	0.71	0.73	0.68
msc01050	1.76	1.56	1.41	1.28	1.24	1.07	1.00	0.94	0.95	0.88	0.86
bcsstk21	1.33	1.27	1.28	1.12	1.08	1.03	1.02	0.94	0.92	0.87	0.92
bcsstk11	2.18	1.94	1.75	1.58	1.50	1.33	1.24	1.16	1.18	1.06	1.06
nasa2146	4.14	3.76	3.44	2.98	2.67	2.47	2.21	2.05	1.93	1.92	1.82
sts4098	4.05	3.68	3.39	2.95	2.64	2.45	2.18	2.04	1.95	1.92	1.83
bcsstk13	4.74	4.38	3.89	3.50	3.22	2.87	2.65	2.53	2.35	2.27	2.27
msc04515	5.18	4.85	4.27	3.83	3.55	3.17	2.91	2.75	2.57	2.47	2.47
ex9	5.51	5.09	4.47	3.98	3.61	3.20	2.93	2.75	2.56	2.45	2.41
bodyy4	6.91	6.17	5.57	4.91	4.37	4.07	3.60	3.36	3.23	3.06	2.98
bodyy5	7.35	6.61	5.94	5.21	4.65	4.28	3.85	3.58	3.43	3.24	3.19
bodyy6	7.62	7.00	6.16	5.49	4.99	4.46	4.05	3.88	3.59	3.44	3.42
Muu	9.54	8.69	7.64	6.91	6.09	5.51	5.07	4.67	4.42	4.28	4.14
s3rmt3m3	11.56	10.46	9.38	8.26	7.42	6.62	6.01	5.66	5.25	5.05	4.94
s3rmt3m1	12.27	11.13	9.97	8.81	7.91	7.09	6.50	5.98	5.65	5.48	5.31
bcsstk28	12.37	11.17	9.99	8.84	7.95	7.09	6.50	5.98	5.64	5.46	5.29
s3rmq4m1	14.98	13.70	12.20	11.00	9.77	8.90	8.06	7.56	7.09	6.89	6.66
bcsstk16	15.83	14.27	12.78	11.33	10.21	9.16	8.41	7.74	7.37	7.02	6.90
Kuu	18.18	16.61	14.76	13.24	11.77	10.72	9.72	9.14	8.57	8.25	8.03
bcsstk38	19.87	17.94	15.87	14.17	12.56	11.35	10.25	9.53	8.96	8.66	8.39
msc23052	63.77	57.58	51.08	45.44	40.25	36.27	32.99	30.45	28.86	27.59	26.95
msc10848	68.07	61.43	54.53	48.35	43.00	38.74	35.05	32.44	30.62	29.35	28.70
cfd2	169.44	153.25	136.16	121.08	107.47	96.82	87.93	81.44	77.05	73.72	72.14
nd3k	181.45	164.28	145.41	128.99	114.53	103.17	93.76	86.59	81.71	78.50	76.54
ship_001	240.26	219.21	195.80	175.16	156.92	142.44	130.22	121.37	115.36	110.66	108.26
shipsec5	453.45	421.02	382.62	347.92	317.27	292.34	271.93	257.41	246.73	239.94	235.01
G3_circuit	381.23	348.65	311.10	281.41	257.54	232.39	212.81	199.20	188.86	181.69	177.99
hood	567.75	515.07	458.85	408.98	365.60	330.83	301.72	280.89	266.83	256.05	251.06
crankseg_1	589.40	530.65	470.41	416.51	369.59	333.51	301.69	280.16	263.73	253.42	247.50

Tables E.31 and E.32 present the simulation-based parameter evaluation results obtained for executions of sparse matrix-vector multiplications.

▼ **Table E.31** — Energy demand for sparse matrix-vector multiplication with respect to different numbers of precise mantissa bits.

Matrix name	*k* precise mantissa bits										
	52 [µJ]	47 [µJ]	42 [µJ]	37 [µJ]	32 [µJ]	27 [µJ]	22 [µJ]	17 [µJ]	12 [µJ]	7 [µJ]	2 [µJ]
nos3	5.4	4.8	4.0	3.2	2.5	1.9	1.5	1.1	0.8	0.6	0.4
bcsstk10	7.8	6.9	5.7	4.6	3.6	2.8	2.1	1.6	1.2	0.8	0.6
msc01050	9.1	8.1	6.7	5.4	4.3	3.3	2.5	1.8	1.3	1.0	0.7
bcsstk21	4.8	4.4	4.0	3.6	3.2	2.8	2.3	1.8	1.4	1.0	0.7
bcsstk11	11.9	10.5	8.7	7.0	5.5	4.3	3.2	2.4	1.7	1.2	0.9
nasa2146	26.8	23.6	19.5	15.6	12.2	9.4	7.0	5.0	3.5	2.3	1.5
sts4098	25.6	22.6	18.7	15.0	11.8	9.1	6.8	4.9	3.5	2.4	1.7
bcsstk13	28.8	25.4	21.1	16.9	13.4	10.4	7.9	5.8	4.2	3.0	2.1
msc04515	31.5	28.1	23.4	19.3	15.5	12.3	9.5	7.1	5.1	3.6	2.6
ex9	36.3	32.1	26.6	21.4	16.8	13.1	9.9	7.3	5.3	3.8	2.8
bodyy4	45.6	40.1	33.2	26.6	21.0	16.2	12.2	8.9	6.4	4.5	3.2
bodyy5	48.3	42.5	35.1	28.2	22.2	17.2	12.9	9.5	6.8	4.8	3.5
bodyy6	50.2	44.2	36.5	29.3	23.0	17.8	13.4	9.9	7.1	5.1	3.7
Muu	62.0	54.6	45.0	36.0	28.3	21.8	16.4	11.8	8.3	5.5	3.8
s3rmt3m3	76.7	67.6	55.8	44.7	35.0	27.0	20.3	14.9	10.7	7.3	5.1
s3rmt3m1	79.9	70.4	58.0	46.5	36.6	28.1	21.1	15.6	11.1	7.7	5.5
bcsstk28	81.4	71.6	59.1	47.4	37.1	28.6	21.3	15.3	10.6	7.2	4.8
s3rmq4m1	91.3	80.7	67.1	54.2	42.9	33.3	25.2	18.6	13.4	9.4	6.6
bcsstk16	101.4	89.3	73.7	59.3	46.9	36.4	27.5	20.2	14.4	10.2	7.4
Kuu	112.7	99.9	83.0	67.4	53.4	41.3	31.7	23.7	17.3	12.5	9.4
bcsstk38	131.2	115.3	95.1	76.2	59.6	45.9	34.4	25.0	17.7	12.2	8.5
msc23052	424.6	373.6	308.3	246.9	193.2	148.9	111.3	80.1	55.7	37.5	24.9
msc10848	452.8	397.9	327.6	261.5	204.2	156.5	116.3	82.7	56.5	37.3	24.2
cfd2	1128.6	993.6	820.7	662.8	523.1	405.7	304.3	220.3	157.0	109.2	74.3
nd3k	1194.2	1051.4	859.2	683.6	533.6	404.4	299.8	217.7	150.7	99.4	65.1
ship_001	1432.8	1268.0	1052.2	834.3	655.2	501.8	377.8	277.5	200.7	144.1	107.4
shipsec5	1930.7	1693.6	1416.1	1159.9	912.5	735.0	565.3	436.4	331.1	247.4	186.2
G3_circuit	2684.9	2246.7	1849.9	1554.7	1235.5	977.9	714.4	533.5	381.0	270.5	193.2
hood	3743.1	3033.4	2702.6	2259.0	1644.1	1354.1	998.2	697.9	489.9	340.1	226.0
crankseg_1	3959.9	3141.9	2793.7	2334.8	1682.2	1378.1	1012.5	701.1	482.3	323.7	208.9

▼ **Table E.32** — Relative error for sparse matrix-vector multiplication with respect to different numbers of precise mantissa bits.

Matrix	k precise mantissa bits										
name	52	47	42	37	32	27	22	17	12	7	2
		$[10^{-15}]$	$[10^{-13}]$	$[10^{-12}]$	$[10^{-10}]$	$[10^{-9}]$	$[10^{-7}]$	$[10^{-6}]$	$[10^{-4}]$	$[10^{-3}]$	$[10^{-1}]$
nos3	0.0	4.38	1.81	5.91	1.50	5.98	2.16	6.11	1.73	6.52	1.60
bcsstk10	0.0	4.38	1.56	5.17	1.67	5.28	1.71	5.43	1.76	5.44	1.62
msc01050	0.0	4.02	1.42	4.90	1.61	4.98	1.57	5.51	1.51	4.88	1.48
bcsstk21	0.0	2.58	0.95	2.99	0.93	3.09	1.01	4.20	1.54	5.05	1.73
bcsstk11	0.0	4.91	1.66	5.46	1.64	5.47	1.82	5.51	1.84	5.77	1.64
nasa2146	0.0	4.69	1.64	5.29	1.68	5.39	1.75	5.54	1.76	5.60	1.63
sts4098	0.0	4.48	1.57	5.17	1.68	5.25	1.69	5.33	1.73	5.54	1.64
bcsstk13	0.0	4.53	1.60	5.07	1.60	5.22	1.72	5.41	1.74	5.60	1.65
msc04515	0.0	4.05	1.52	4.69	1.48	4.88	1.57	5.30	1.89	5.89	1.70
ex9	0.0	4.64	1.65	5.31	1.68	5.38	1.74	5.54	1.78	5.61	1.65
bodyy4	0.0	4.71	1.64	5.28	1.69	5.38	1.73	5.53	1.77	5.63	1.65
bodyy5	0.0	4.71	1.64	5.26	1.69	5.37	1.73	5.53	1.77	5.63	1.65
bodyy6	0.0	4.71	1.64	5.25	1.69	5.36	1.72	5.53	1.76	5.64	1.66
Muu	0.0	4.79	1.71	5.33	1.61	5.01	1.91	5.97	1.80	5.69	1.50
s3rmt3m3	0.0	4.66	1.64	5.23	1.67	5.37	1.73	5.51	1.75	5.61	1.65
s3rmt3m1	0.0	4.65	1.65	5.16	1.67	5.34	1.73	5.50	1.77	5.45	1.65
bcsstk28	0.0	4.69	1.64	5.26	1.68	5.39	1.74	5.52	1.76	5.61	1.64
s3rmq4m1	0.0	4.11	1.47	4.62	1.51	4.81	1.54	4.98	1.59	4.91	1.50
bcsstk16	0.0	4.65	1.63	5.13	1.69	5.31	1.72	5.41	1.76	5.52	1.61
Kuu	0.0	4.12	1.46	4.89	1.57	4.85	1.48	5.00	1.68	5.29	1.65
bcsstk38	0.0	4.68	1.64	5.24	1.67	5.37	1.72	5.52	1.77	5.59	1.64
msc23052	0.0	4.65	1.62	5.19	1.67	5.33	1.71	5.49	1.75	5.59	1.62
msc10848	0.0	4.72	1.64	5.26	1.69	5.39	1.73	5.55	1.77	5.62	1.64
cfd2	0.0	4.53	1.59	5.11	1.63	5.23	1.67	5.36	1.71	5.47	1.59
nd3k	0.0	4.70	1.65	5.28	1.68	5.40	1.73	5.53	1.77	5.60	1.64
ship_001	0.0	3.89	1.37	4.39	1.41	4.50	1.44	4.64	1.48	4.74	1.38
shipsec5	0.0	2.13	0.75	2.43	0.78	2.46	0.78	2.50	0.81	2.58	0.75
G3_circuit	0.0	3.69	1.37	4.58	1.37	4.58	1.67	4.45	1.51	4.93	1.40
hood	0.0	4.34	1.52	4.88	1.56	5.00	1.60	5.11	1.64	5.22	1.53
crankseg_1	0.0	4.65	1.62	5.18	1.66	5.31	1.70	5.53	1.78	5.65	1.64

Table E.33 compares the results of the combined parameter estimation method against the results of the simulation-based parameter evaluation methods for complete solver executions.

▼ **Table E.33** — Comparison of runtime for estimation-based T_S and simulation-based methods T_E, as well as energy estimation error e of the estimation-based method.

Matrix name	No preconditioner			Jacobi preconditioner			ICC preconditioner		
	T_S [s]	T_E [s]	e [%]	T_S [s]	T_E [s]	e [%]	T_S [s]	T_E [s]	e [%]
nos3	10.3	1,281.9	11.5	10.9	1,165.4	12.6	11.6	847.1	13.1
bcsstk10	13.2	5,481.3	7.3	13.8	5,481.3	4.1	14.9	3,982.2	6.0
msc01050	15.9	5,290.9	6.7	16.6	5,290.9	8.4	18.1	7,435.1	6.8
bcsstk21	14.3	6,349.7	14.1	15.0	5,016.2	13.0	16.0	2,653.7	6.9
bcsstk11	19.7	5,435.3	7.4	20.3	5,435.3	6.3	22.3	8,021.4	7.9
nasa2146	33.5	3,676.1	6.6	34.1	3,015.0	6.6	38.6	4,449.1	4.3
sts4098	33.5	7,658.6	8.6	34.2	4,625.8	2.1	38.5	7,143.7	8.3
bcsstk13	38.4	7,785.4	0.6	39.1	7,785.4	5.3	44.1	13,549.8	1.4
msc04515	42.7	8,508.9	6.3	43.4	8,508.9	4.5	48.9	14,784.7	2.2
ex9	46.3	7,936.6	6.9	47.1	7,936.6	5.1	52.9	14,548.0	6.4
bodyy4	54.8	3,173.7	6.6	56.4	2,991.1	8.0	63.1	6,353.1	2.6
bodyy5	57.6	9,531.7	7.4	59.2	7,304.9	5.8	66.4	22,361.0	3.9
bodyy6	60.0	14,352.7	3.3	61.7	14,352.7	5.5	69.1	23,412.4	5.6
Muu	34.0	221.8	3.6	34.8	126.0	10.2	45.3	262.0	4.2
s3rmt3m3	84.7	8,321.9	8.0	85.5	11,276.3	4.5	98.4	24,983.3	0.2
s3rmt3m1	94.4	12,084.8	10.6	95.1	12,084.8	4.6	109.0	26,753.0	4.6
bcsstk28	92.7	11,095.8	8.2	93.4	11,095.8	6.1	107.4	25,855.1	6.8
s3rmq4m1	113.1	13,922.4	10.3	113.9	13,922.4	3.4	130.8	31,683.4	3.9
bcsstk16	116.5	7,323.4	6.9	117.3	3,929.3	3.4	135.4	7,423.7	3.7
Kuu	134.4	11,139.4	6.5	135.2	8,875.7	4.4	156.1	9,222.2	12.7
bcsstk38	148.5	17,043.7	6.9	149.4	17,043.7	3.9	171.9	40,454.3	2.5
msc23052	602.6	47,631.8	6.8	604.4	47,631.8	6.5	677.6	122,641.8	9.2
msc10848	481.6	45,569.3	8.0	482.8	45,569.3	5.3	561.5	125,444.5	3.5
cfd2	1,229.2	145,999.1	6.9	1,237.4	145,999.1	6.7	1,430.6	347,344.1	12.6
nd3k	1,258.1	111,753.6	3.6	1,259.2	111,753.6	4.9	1,468.9	322,549.7	2.0
ship_001	1,801.5	158,407.9	1.9	1,804.2	158,407.9	5.6	2,084.1	440,928.9	2.9
shipsec5	3,493.1	377,650.4	10.8	3,504.8	377,650.4	1.8	4,034.4	918,990.7	3.0
G3_circuit	3,121.0	826,055.6	3.3	3,220.5	826,055.6	7.1	3,579.5	1,284,599.7	3.8
hood	4,109.5	412,303.0	6.0	4,123.9	412,303.0	6.9	4,779.8	1,082,567.0	5.5
crankseg_1	4,094.7	361,439.9	5.1	4,098.5	346,259.4	3.9	4,783.0	778,949.4	4.1

Bibliography

[Abadi16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016. [page 9]

[Advan14] S. Advani, N. Chandramoorthy, K. Swaminathan, K. Irick, Y. C. P. Cho, J. Sampson, and V. Narayanan. Refresh Enabled Video Analytics (REVA): Implications on Power and Performance of Dram Supported Embedded Visual Systems. In *32nd IEEE International Conference on Computer Design (ICCD)*, pages 501–504. 2014. [page 47]

[Amaru13] L. Amarú, P.-E. Gaillardon, J. Zhang, and G. De Micheli. Power-gated Differential Logic Style Based on Double-gate Controllable-polarity Transistors. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(10):672–676, 2013. [page 14]

[Ament10] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 583–592. 2010. [page 6]

[Asano06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, and S. W. Williams. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. [page 5]

[Avizi04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. [pages 2 and 26]

[Avram17] E. Avramidis and O. E. Akman. Optimisation of an Exemplar Oculomotor Model Using Multi-objective Genetic Algorithms Executed on a GPU-CPU Combination. *BMC Systems Biology*, 11(1):40, 2017. [page 8]

[Balay16] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016. [page 44]

[Barba16] M. Barbareschi, F. Iannucci, and A. Mazzeo. An Extendible Design Exploration Tool for Supporting Approximate Computing Techniques. In *International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. 2016. [pages 53, 56, 98, 101, and 105]

[Barlo85] J. L. Barlow and E. Bareiss. On Roundoff Error Distributions in Floating Point and Logarithmic Arithmetic. *Computing*, 34(4):325–347, 1985. [page 40]

[Bastr12] S. Bastrakov, R. Donchenko, A. Gonoskov, E. Efimenko, A. Malyshev, I. Meyerov, and I. Surmin. Particle-in-cell Plasma Simulation on Heterogeneous Cluster Systems. *Journal of Computational Science*, 3(6):474–479, 2012. [page 8]

[Beche16] A. Becher, J. Echavarria, D. Ziener, S. Wildermann, and J. Teich. A LUT-Based Approximate Adder. In *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 27–27. 2016. [page 48]

[Becke10] B. Becker, S. Hellebrand, I. Polian, B. Straube, W. Vermeiren, and H.-J. Wunderlich. Massive Statistical Process Variations: A Grand Challenge for Testing Nanoelectronic Circuits. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 95–100. 2010. [page 11]

[Beign13] E. Beigné, A. Valentian, B. Giraud, O. Thomas, T. Benoist, Y. Thonnart, S. Bernard, G. Moritz, O. Billoint, Y. Maneglia, et al. Ultra-wide Voltage Range Designs in Fully-depleted Silicon-on-insulator FETs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 613–618. 2013. [page 14]

[Benzi02] M. Benzi. Preconditioning Techniques for Large Linear Systems: a Survey. *Journal of Computational Physics*, 182(2):418–477, 2002. [pages 22 and 82]

[Bhard13] K. Bhardwaj and P. S. Mane. ACMA: Accuracy-configurable Multiplier Architecture for Error-resilient System-on-chip. In *8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–6. 2013. [page 48]

[Bhard14] K. Bhardwaj, P. S. Mane, and J. Henkel. Power-and Area-efficient Approximate Wallace Tree Multiplier for Error-resilient Systems. In *15th International Symposium on Quality Electronic Design (ISQED)*, pages 263–269. 2014. [page 48]

[Bini05] D. A. Bini, N. J. Higham, and B. Meini. Algorithms for the Matrix p-th Root. *Numerical Algorithms*, 39(4):349–378, 2005. [page 51]

[Boore14] D. M. Boore, J. P. Stewart, E. Seyhan, and G. M. Atkinson. NGA-west2 Equations for Predicting PGA, PGV, and 5% Damped PSA for Shallow Crustal Earthquakes. *Earthquake Spectra*, 30(3):1057–1085, 2014. [page 6]

[Borka05] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10–16, 2005. [page 11]

[Borka10] S. Borkar. The Exascale Challenge. In *International Symposium on VLSI Design Automation and Test (VLSI-DAT)*, pages 2–3. 2010. [page 13]

[Boute15] A. Bouteiller, T. Herault, G. Bosilca, P. Du, and J. Dongarra. Algorithm-Based Fault Tolerance for Dense Matrix Factorizations, Multiple Failures and Accuracy. *ACM Transactions on Parallel Computing*, 1(2):10:1–10:28, 2015. [page 32]

[Bouvi14] D. Bouvier, B. Cohen, W. Fry, S. Godey, and M. Mantor. Kabini: An AMD Accelerated Processing Unit System on a Chip. *IEEE Micro*, 34(2):22–33, 2014. [page 43]

[Braun12a] C. Braun, S. Holst, H.-J. Wunderlich, J. M. Castillo, and J. Gross. Acceleration of Monte-Carlo Molecular Simulations on Hybrid Computing Architectures. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD'12)*, pages 207–212. 2012. [page 8]

[Braun12b] C. Braun, M. Daub, A. Schöll, G. Schneider, and H.-J. Wunderlich. Parallel Simulation of Apoptotic Receptor-Clustering on GPGPU Many-Core Architectures. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'12)*, pages 1–6. 2012. [page 8]

[Braun14] C. Braun, S. Halder, and H.-J. Wunderlich. A-ABFT: Autonomous Algorithm-Based Fault Tolerance for Matrix Multiplications on Graphics Processing Units. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, pages 443–454. 2014. [pages 32 and 40]

[Brone08] G. Bronevetsky and B. de Supinski. Soft Error Vulnerability of Iterative Linear Algebra Methods. In *Proceedings of the International Conference on Supercomputing*, pages 155–164. 2008. [pages 29, 32, 35, 37, 41, 66, 68, 74, 114, and 116]

[Brook00] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94. 2000. [page 54]

[Buato09] L. Buatois, G. Caumon, and B. Levy. Concurrent Number Cruncher: a GPU Implementation of a General Sparse Linear Solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24(3):205–223, 2009. [page 6]

[Bushn04] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, volume 17. Springer Science & Business Media, 2004. [page 29]

[Camus15] V. Camus, J. Schlachter, and C. Enz. Energy-Efficient Inexact Speculative Adder with High Performance and Accuracy Control. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 45–48. 2015. [page 48]

[Cappe14] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 Update. In *Supercomputing Frontiers and Innovations*, volume 1. 2014. [pages 1 and 26]

[Carbi13] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *ACM SIGPLAN Notices*, volume 48, pages 33–52. 2013. [page 49]

[Casti15] L. R. F. Castillo. The large hadron collider. In *The Search and Discovery of the Higgs Boson*. Morgan & Claypool Publishers, 2015. [page 9]

[Catan08] B. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *Proceedings of the 25th International Conference on Machine Learning*, pages 104–111. 2008. [page 9]

[Chakr10] S. T. Chakradhar and A. Raghunathan. Best-effort Computing: Re-thinking Parallel Software and Hardware. In *47th ACM/IEEE Design Automation Conference (DAC'10)*, pages 865–870. 2010. [page 47]

[Chand17] A. Chandrasekharan, D. Große, and R. Drechsler. Proact: A Processor for High Performance On-demand Approximate Computing. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 463–466. ACM, 2017. [pages 3, 43, and 49]

[Chatt09] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven Gate-level Simulation with GP-GPUs. In *Proceedings of the 46th Annual ACM/IEEE Design Automation Conference (DAC)*, pages 557–562. ACM, 2009. [page 8]

[Chen12] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 25. IEEE Computer Society Press, 2012. [page 9]

[Chen13] Z. Chen. 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*, pages 167–176. 2013. [pages 41, 42, and 78]

[Chen14] C. P. Chen and C.-Y. Zhang. Data-intensive Applications, Challenges, Techniques and Technologies: A Survey on Big Data. *Information Sciences*, 275:314–347, 2014. [page 9]

[Chen15a] B. Chen, B. Bottoms, D. Armstrong, and A. Isobayashi. ITRS 2.0: Heterogeneous Integration. *Solid State Technology*, 58(3):13–17, 2015. URL `http://www.itrs2.net/itrs-reports.html`. [pages 1 and 43]

[Chen15b] K. Chen, F. Lombardi, and J. Han. Matrix Multiplication by an Inexact Systolic Array. In *Proceedings of the IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH'15)*, pages 151–156. 2015. [pages 48 and 54]

[Chen16] Y. Chen, X. Yang, F. Qiao, J. Han, Q. Wei, and H. Yang. A Multi-accuracy-Level Approximate Memory Architecture Based on Data Significance Analysis. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 385–390. 2016. [page 48]

[Cheng99] K.-T. Cheng, S.-Y. Huang, and W.-J. Dai. Fault Emulation: A New Methodology for Fault Grading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(10):1487–1495, 1999. [page 28]

[Chien16] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna, et al. Exploring Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience. *International Journal of High Performance Computing Applications*, pages 1–27, 2016. [page 41]

[Chipp13] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*, pages 1–9. 2013. [pages 47, 51, 52, 53, 56, 83, 98, 105, and 106]

[Cho14] K. Cho, Y. Lee, Y. H. Oh, G.-c. Hwang, and J. W. Lee. eDRAM-based Tiered-Reliability Memory with Applications to Low-power Frame Buffers. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 333–338. 2014. [page 48]

[Choi09] J. G. Choi and P. H. Seong. Reliability of Electronic Components. In *Reliability and Risk Issues in Large Scale Safety-critical Digital Control Systems*, pages 3–24. Springer, 2009. [page 12]

[Chowd96] A.-R. Chowdhury and P. Banerjee. A New Error Analysis Based Method for Tolerance Computation for Algorithm-Based Checks. *IEEE Transactions on Computers*, 45(2):238–243, 1996. [pages 40, 66, 120, and 160]

[Chung10] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip Heterogeneous Computing: Does the Future include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. 2010. [pages 1, 7, and 43]

[Cools16] S. Cools, W. Vanroose, E. F. Yetkin, E. Agullo, and L. Giraud. On Rounding Error Resilience, Maximal Attainable Accuracy and Parallel Performance of the Pipelined Conjugate Gradients method for Large-Scale Linear Systems in PETSc. In *Proceedings of the Exascale Applications and Software Conference*, pages 3:1–10. 2016. [pages 25, 84, 87, 91, 92, 93, and 113]

[Cui13] Y. Cui, E. Poyraz, K. B. Olsen, J. Zhou, K. Withers, S. Callaghan, J. Larkin, C. Guest, D. Choi, A. Chourasia, et al. Physics-based Seismic Hazard Analysis on Petascale Heterogeneous Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 70. ACM, 2013. [page 8]

[DAzev05] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized Sparse Matrix Multiply for Compressed Row Storage Format. In *Computational Science*, volume 1 of *Lecture Notes in Computer Science*, pages 99–106. 2005. [page 113]

[Davis11] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011. [pages 9 and 112]

[Dehiy17] R. Dehiya, A. Singh, P. K. Gupta, and M. Israil. Optimization of Computations for Adjoint Field and Jacobian Needed in 3D CSEM Inversion. *Journal of Applied Geophysics*, 136:444–454, 2017. [page 5]

[Dell97] T. J. Dell. A White Paper on the Benefits of Chipkill-correct ECC for PC Server Main Memory. *IBM Microelectronics Division*, 11, 1997. [page 30]

[Deng16] L. Deng, H. Bai, F. Wang, and Q. Xu. CPU/GPU Computing for An Implicit Multi-Block Compressible Navier-Stokes Solver on Heterogeneous Platform. In *International Journal of Modern Physics: Conference Series*, volume 42. 2016. [page 9]

[Denna74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. [pages 2 and 14]

[Desch06] J.-P. Deschamps, G. J. Bioul, and G. D. Sutter. *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. John Wiley & Sons, 2006. [page 115]

[Di Ma16] C. Di Martino, Z. Kalbarczyk, and R. Iyer. Measuring the Resiliency of Extreme-Scale Computing Environments. In *Principles of Performance and Reliability Modeling and Evaluation*, pages 609–655. Springer, 2016. [pages 10, 29, and 115]

[Diche16] K. Dichev and D. S. Nikolopoulos. TwinPCG: Dual Thread Redundancy with Forward Recovery for Preconditioned Conjugate Gradient Methods. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 506–514. 2016. [pages 42 and 114]

[Donga12] J. Dongarra, T. Dong, M. Gates, A. Haidar, S. Tomov, and I. Yamazaki. MAGMA: A New Generation of Linear Algebra Library for Gpu and Multicore Architectures. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, 2012. [page 44]

[Donga15] J. Dongarra, M. A. Heroux, and P. Luszczek. HPCG Benchmark: A New Metric for Ranking High Performance Computing Systems. *Technical Report, Electrical Engineering and Computer Science Department*, 2015. [page 9]

[Donga94] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington. A Sparse Matrix Library in C++ for High Performance Architectures. 1994. [page 44]

[Du12] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. Algorithm-Based Fault Tolerance for Dense Matrix Factorizations. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 225–234. 2012. [page 32]

[Duff02] I. S. Duff, M. A. Heroux, and R. Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002. [page 44]

[Ehler17] W. Ehlers and C. Luo. A phase-field approach embedded in the theory of porous media for the description of Dynamic Hydraulic Fracturing. *Computer Methods in Applied Mechanics and Engineering*, 315:348–368, 2017. [page 10]

[Eldri14] S. Eldridge, F. Raudies, D. Zou, and A. Joshi. Neural Network-based Accelerators for Transcendental Function Approximation. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI '14, pages 169–174. 2014. [page 48]

[Esmae12a] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312. 2012. [pages 3, 43, and 48]

[Esmae12b] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *IEEE Micro*, 32(3):122–134, 2012. [page 15]

[Esmae12c] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460. 2012. [page 48]

[Esmae13] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Power Challenges May End the Multicore Era. *Communications of the ACM*, 56(2):93–102, 2013. [pages 2, 3, and 14]

[Espos16] D. Esposito, G. Castellano, D. D. Caro, E. Napoli, N. Petra, and A. G. M. Strollo. Approximate Adder with Output Correction for Error Tolerant Applications and Gaussian Distributed Inputs. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1970–1973. 2016. [page 48]

[Fang16] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. A Systematic Methodology for Evaluating the Error Resilience of GPGPU Applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3397–3411, 2016. [page 29]

[Farsh13] F. Farshchi, M. S. Abrishami, and S. M. Fakhraie. New Approximate Multiplier for Low Power Digital Signal Processing. In *17th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 25–30. 2013. [page 48]

[Fasi16] M. Fasi, J. Langou, Y. Robert, and B. Uçar. A Backward/Forward Recovery Approach for the Preconditioned Conjugate Gradient Method. *Journal of Computational Science*, 17:522–534, 2016. [pages 32, 37, 38, and 116]

[Feich15] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, and T. Aoki. Performance Modeling and Analysis of Heterogeneous Lattice Boltzmann Simulations on CPU-GPU Clusters. *Parallel Computing*, 46:1–13, 2015. [page 9]

[Feng10] Z. Feng and Z. Zeng. Parallel Multigrid Preconditioning on Graphics Processing Units (GPUs) for Robust Power Grid Analysis. In *Proceedings of the 47th ACM/EDAC/IEEE Design Automation Conference (DAC'10)*, pages 661–666. 2010. [page 5]

[Ferle13] V. Ferlet-Cavrois, L. W. Massengill, and P. Gouker. Single Event Transients in Digital CMOS-A Review. *IEEE Transactions on Nuclear Science*, 60(3):1767–1790, 2013. [page 12]

[Filip17] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):30, 2017. [page 6]

[Flich16] J. Flich, G. Agosta, P. Ampletzer, D. A. Alonso, C. Brandolese, A. Cilardo, W. Fornaciari, Y. Hoornenborg, M. Kovač, B. Maitre, et al. Enabling HPC for QoS-sensitive Applications: the MANGO Approach. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 702–707. 2016. [page 3]

[Gaill14] P.-E. Gaillardon, L. Amaru, J. Zhang, and G. De Micheli. Advanced System on a Chip Design Based on Controllable-polarity FETs. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, page 235. European Design and Automation Association, 2014. [page 14]

[Gallo15] E. Gallopoulos, B. Philippe, and A. Sameh. *Parallelism in Matrix Computations*. Scientific Computation. Springer, 2015. [page 72]

[Gan15] L. Gan, H. Fu, W. Luk, C. Yang, W. Xue, X. Huang, Y. Zhang, and G. Yang. Solving the Global Atmospheric Equations through Heterogeneous Reconfigurable Platforms. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2):11, 2015. [page 8]

[Gao16a] Y. Gao and P. Zhang. A Survey of Homogeneous and Heterogeneous System Architectures in High Performance Computing. In *IEEE International Conference on Smart Cloud (SmartCloud)*, pages 170–175. 2016. [page 1]

[Gao16b] Z. Gao, P. Reviriego, and J. A. Maestro. Efficient Fault Tolerant Parallel Matrix-Vector Multiplications. In *IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS'16)*, pages 25–26. 2016. [page 38]

[Gokhb16] A. Gokhberg and A. Fichtner. Full-waveform Inversion on Heterogeneous HPC Systems. *Computers & Geosciences*, 89:260–268, 2016. [page 8]

[Golub13] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, 4th edition, 2013. [pages 5, 20, 24, 39, 150, 153, and 160]

[Gotti16] K. C. Gottiparthi, R. Sankaran, and J. C. Oefelein. High Fidelity Large Eddy Simulation of Reacting Supercritical Fuel Jet-in-Cross-Flow using GPU acceleration. In *52nd AIAA/SAE/ASEE Joint Propulsion Conference*, page 4791. 2016. [page 9]

[Grigo14a] B. Grigorian and G. Reinman. Dynamically Adaptive and Reliable Approximate Computing using Light-weight Error Analysis. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS'14)*, pages 248–255. 2014. [pages 46 and 48]

[Grigo14b] B. Grigorian and G. Reinman. Improving Coverage and Reliability in Approximate Computing using Application-Specific, Light-Weight Checks. In *Workshop on Approximate Computing Across the System Stack (WACAS)*. 2014. [page 46]

[Grigo15] B. Grigorian, N. Farahpour, and G. Reinman. BRAINIAC: Bringing Reliable Accuracy into Neurally-implemented Approximate Computing. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 615–626. 2015. [page 48]

[Gulat08] K. Gulati and S. P. Khatri. Towards Acceleration of Fault Simulation Using Graphics Processing Units. In *Proceedings of the 45th Annual ACM/EDAC/IEEE Design Automation Conference (DAC'08)*, pages 822–827. 2008. [page 8]

[Gulat09] K. Gulati, J. F. Croix, S. P. Khatr, and R. Shastry. Fast Circuit Simulation on Graphics Processing Units. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 403–408. 2009. [page 8]

[Gupta11] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy. IMPACT: Imprecise Adders for Low-power Approximate Computing. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, (ISLPED), pages 409–414. 2011. [page 48]

[Gupta13] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy. Low-power Digital Signal Processing Using Approximate Adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, 2013. [page 48]

[Hakka15] D. Hakkarinen, P. Wu, and Z. Chen. Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1323–1335, 2015. [page 32]

[Hamer05] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction-Level Parallelism*, 7:1–28, 2005. [pages 55, 56, 98, 99, 101, and 107]

[Han13] J. Han and M. Orshansky. Approximate Computing: An Emerging Paradigm for Energy-efficient Design. In *18th IEEE European Test Symposium (ETS)*, pages 1–6. 2013. [pages 3 and 46]

[Han16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. 2016. [page 5]

[Hashe15] S. Hashemi, R. Bahar, and S. Reda. Drum: A Dynamic Range Unbiased Multiplier for Approximate Applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 418–425. 2015. [page 48]

[Helfe12] R. Helfenstein and J. Koko. Parallel Preconditioned Conjugate Gradient Algorithm on GPU. In *Proceedings of the 15th International Congress on Computational and Applied Mathematics (ICCAM)*, volume 236, pages 3584 – 3590. 2012. [page 6]

[Herau16] T. Herault and Y. Robert. *Fault-Tolerance Techniques for High-Performance Computing*. Springer, 2016. [page 13]

[Herke14] A. Herkersdorf, H. Aliee, M. Engel, M. Glaß, C. Gimmler-Dumont, J. Henkel, V. Kleeberger, M. A. Kochte, J. M. Kühn, D. Mueller-Gritschneder, S. R. Nassif, H. Rauchfuss, W. Rosenstiel, U. Schlichtmann, M. Shafique, M. B. Tahoori, J. Teich, N. Wehn, C. Weis, and H. Wunderlich. Resilience Articulation Point (RAP): Cross-layer Dependability Modeling for Nanometer System-On-Chip Resilience. *Microelectronics Reliability*, 54(6-7):1066–1074, 2014. [page 29]

[Heste52] M. R. Hestenes and E. Stiefel. *Methods of Conjugate Gradients for Solving Linear Systems*, volume 49. Journal of Research of the National Bureau of Standards, 1952. [page 21]

[Higha96] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 48. Siam, 1996. [page 40]

[Hoang13] R. V. Hoang, D. Tanna, L. C. Jayet Bray, S. M. Dascalu, and F. C. Harris Jr. A Novel CPU/GPU Simulation Environment for Large-scale Biologically Realistic Neural Modeling. *Frontiers in Neuroinformatics*, 7:19, 2013. [page 8]

[Holik16] L. Holık, O. Lengál, A. Rogalewicz, L. Sekanina, Z. Vašıcek, and T. Vojnar. Towards Formal Relaxed Equivalence Checking in Approximate Computing Methodology. *2nd Workshop on Approximate Computing (WAPCO'16)*, 2016. [page 48]

[Holst12] S. Holst, E. Schneider, and H.-J. Wunderlich. Scan Test Power Simulation on GPGPUs. In *IEEE 21st Asian Test Symposium (ATS)*, pages 155–160. 2012. [page 8]

[Holst15] S. Holst, M. E. Imhof, and H.-J. Wunderlich. High-throughput Logic Timing Simulation on GPGPUs. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):37, 2015. [pages 8 and 103]

[Holst16] S. Holst, E. Schneider, X. Wen, S. Kajihara, Y. Yamato, H.-J. Wunderlich, and M. A. Kochte. Timing-Accurate Estimation of IR-Drop Impact on Logic-and Clock-Paths During At-Speed Scan Test. In *IEEE Asian Test Symposium (ATS)*, pages 19–24. 2016. [page 8]

[Hsieh98] C.-T. Hsieh and M. Pedram. Microprocessor Power Estimation Using Profile-driven Program Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(11):1080–1089, 1998. [pages 55, 56, 98, and 107]

[Hsu05] C.-H. Hsu and W.-C. Feng. A Power-aware Run-Time System for High-Performance Computing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 2005. [page 15]

[Hsueh97] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *Computer*, 30(4):75–82, 1997. [page 28]

[Hu15] J. Hu and W. Qian. A New Approximate Adder with Low Relative Error and Correct Sign Calculation. In *Proceedings of the 2015 Design, Automation*

& Test in Europe Conference & Exhibition (DATE), pages 1449–1454. 2015. [page 48]

[Huang12] J. Huang, J. Lach, and G. Robins. A Methodology for Energy-Quality Trade-off using Imprecise Hardware. In *Proceedings of the 49th ACM/EDAC/IEEE Annual Design Automation Conference (DAC'12)*, pages 504–509. 2012. [page 106]

[Huang84] K.-H. Huang and J. A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984. [pages 2, 13, 32, 33, and 39]

[Hurre13] J. W. Hurrell, M. M. Holland, P. R. Gent, S. Ghan, J. E. Kay, P. J. Kushner, J.-F. Lamarque, W. G. Large, D. Lawrence, K. Lindsay, et al. The Community Earth System Model: A Framework for Collaborative Research. *Bulletin of the American Meteorological Society*, 94(9):1339–1360, 2013. [pages 6 and 10]

[IEEE 08] IEEE Standards Committee. 754-2008 IEEE Standard for Floating-Point Arithmetic. *IEEE Computer Society*, 2008. [pages 113, 153, 155, and 157]

[ITR] The International Technology Roadmap for Semiconductors 2013 Edition. URL `http://www.itrs.net/Links/2013ITRS/Home2013.htm`. [page 2]

[Intel17] Intel Corporation. 7th Generation Intel Core Fact Sheet, 2017. [page 43]

[Jain16] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise Loads and Stores: The Case for an Asymmetric Compute-memory Architecture for Approximation. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. 2016. [page 48]

[Jeong16] D. Jeong, Y. H. Oh, J. W. Lee, and Y. Park. An eDRAM-Based Approximate Register File for GPUs. *IEEE Design & Test*, 33(1):23–31, 2016. [page 48]

[Jiang15] H. Jiang, J. Han, and F. Lombardi. A Comparative Review and Evaluation of Approximate Adders. In *Proceedings of the Great Lakes Symposium on VLSI*, pages 343–348. 2015. [page 15]

[Jou86] J.-Y. Jou and J. A. Abraham. Fault-tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures. *Proceedings of the IEEE*, 74(5):732–741, 1986. [pages 32 and 34]

[Joupp17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers. In-datacenter performance analysis of a tensor processing unit. *International Symposium on Computer Architecture (ISCA)*, 2017. [page 9]

[Kaesl14] H. Kaeslin. *Top-down Digital VLSI Design: From Architectures to Gate-level Circuits and FPGAs.* Morgan Kaufmann, 2014. [pages 3 and 44]

[Kahng12] A. B. Kahng and S. Kang. Accuracy-Configurable Adder for Approximate Arithmetic Designs. In *Proceedings of the 49th Annual ACM/EDAC/IEEE Design Automation Conference*, pages 820–825. 2012. [page 48]

[Kanaw92] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A Tool for The Validation of System Dependability Properties. In *The 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 336–344. 1992. [page 29]

[Kapre09] N. Kapre and A. DeHon. Performance Comparison of Single-precision Spice Model-evaluation on FPGA, GPU, Cell, and Multi-core Processors. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 65–72. 2009. [page 8]

[Kessl15] A. Keßler, K. Schrader, and C. Könke. Distributed FE Analysis of Multiphase Composites for Linear and Nonlinear Material Behaviour. In *High Performance Computing in Science and Engineering*, pages 661–669. Springer, 2015. [page 9]

[Keyes13] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, et al. Multiphysics Simulations: Challenges and Opportunities. *The International Journal of High Performance Computing Applications*, 27(1):4–83, 2013. [page 6]

[Khudi13] D. S. Khudia and S. Mahlke. Low Cost Control Flow Protection Using Abstract Control Signatures. In *ACM SIGPLAN Notices*, volume 48, pages 3–12. ACM, 2013. [page 114]

[Kim13] Y. Kim, Y. Zhang, and P. Li. An Energy Efficient Approximate Adder with Carry Skip for Error Resilient Neuromorphic VLSI Systems. In *ICCAD*, pages 130–137. 2013. [page 48]

[Kim14] H.-S. Kim, G. A. Vecchi, T. R. Knutson, W. G. Anderson, T. L. Delworth, A. Rosati, F. Zeng, and M. Zhao. Tropical Cyclone Simulation and Response to CO2 Doubling in the Gfdl Cm2. 5 High-resolution Coupled Climate Model. *Journal of Climate*, 27(21):8034–8054, 2014. [page 6]

[Kocht10] M. A. Kochte, M. Schaal, H.-J. Wunderlich, and C. G. Zoellin. Efficient Fault Simulation on Many-core Processors. In *Proceedings of the 47th ACM/IEEE Design Automation Conference (DAC)*, pages 380–385. ACM, 2010. [page 8]

[Koren07] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Elsevier, 1 edition, 2007. [pages 2, 12, 30, and 151]

[Kouts17] P. Koutsovasilis, C. Kalogirou, C. Konstantas, M. Maroudas, M. Spyrou, and C. D. Antonopoulos. AcHEe: Evaluating Approximate Computing and Heterogeneity for Energy Efficiency. *Parallel Computing*, 2017. [page 49]

[Kreut16] M. Kreutzer, J. Thies, A. Pieper, A. Alvermann, M. Galgon, M. Röhrig-Zöllner, F. Shahzad, A. Basermann, A. R. Bishop, H. Fehske, et al. Performance Engineering and Energy Efficiency of Building Blocks for Large, Sparse Eigenvalue Computations on Heterogeneous Supercomputers. In *Software for Exascale Computing-SPPEXA 2013-2015*, pages 317–338. Springer, 2016. [page 44]

[Kulka11] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *24th International Conference on VLSI Design (VLSI Design)*, pages 346–351. 2011. [page 48]

[Kyaw10] K. Y. Kyaw, W. L. Goh, and K. S. Yeo. Low-Power High-Speed Multiplier for Error-Tolerant Application. In *IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–4. 2010. [page 48]

[Lashu12] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A Massively Parallel Adaptive Fast Multipole Method on Heterogeneous Architectures. *Communications of the ACM*, 55(5):101–109, 2012. [page 8]

[Lass17] M. Lass, T. D. Kühne, and C. Plessl. Using Approximate Computing for the Calculation of Inverse Matrix p-th Roots. *IEEE Embedded Systems Letters*, 2017. [pages 48, 50, 51, and 84]

[Laure04] J. Laurent, N. Julien, E. Senn, and E. Martin. Functional Level Power Analysis: An Efficient Approach for Modeling the Power Consumption of Complex Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. 2004. [pages 53, 54, and 98]

[Lee16] S. Lee, D. Lee, K. Han, E. Shriver, L. K. John, and A. Gerstlauer. Statistical Quality Modeling of Approximate Hardware. In *17th International Symposium on Quality Electronic Design (ISQED)*, pages 163–168. 2016. [pages 53 and 106]

[Leng15] C. Leng, X.-D. Wang, T.-H. Wang, and W.-M. Yan. Multi-parameter Optimization of Flow and Heat Transfer for a Novel Double-layered Microchannel Heat Sink. *International Journal of Heat and Mass Transfer*, 84:359–369, 2015. [page 5]

[Li09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480. 2009. [page 54]

[Li13] R. Li and Y. Saad. GPU-accelerated Preconditioned Iterative Linear Solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013. [page 6]

[Li15] P. Li, Y. Luo, N. Zhang, and Y. Cao. HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Machine Learning Algorithms. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348. 2015. [page 9]

[Lin13] C.-H. Lin and C. Lin. High Accuracy Approximate Multiplier with Error Correction. In *IEEE 31st International Conference on Computer Design (ICCD)*, pages 33–38. 2013. [page 48]

[Liu11] H. Liu, T. Davies, C. Ding, C. Karlsson, and Z. Chen. Algorithm-based Recovery for Newton's Method without Checkpointing. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1541–1548. 2011. [page 58]

[Liu12] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM Refresh-power through Critical Data Partitioning. *ACM SIGPLAN Notices*, 47(4):213–224, 2012. [page 48]

[Liu13a] Y. Liu, A. Wirawan, and B. Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117, 2013. [page 8]

[Liu13b] X.-X. Liu, H. Wang, and S. X.-D. Tan. Parallel Power Grid Analysis using Preconditioned GMRES solver on CPU-GPU Platforms. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 561–568. IEEE, 2013. [page 8]

[Liu14] C. Liu, J. Han, and F. Lombardi. A Low-power, High-performance Approximate Multiplier with Configurable Partial Error Recovery. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, page 95. 2014. [page 48]

[Liu15a] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky. Sparse Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814. 2015. [page 5]

[Liu15b] J. Liu, M. C. Kurt, and G. Agrawal. A Practical Approach for Handling Soft Errors in Iterative Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 158–161. 2015. [pages 42 and 114]

[Liu16a] Y. Liu and B. Schmidt. LightSpMV: Faster CUDA-Compatible Sparse Matrix-Vector Multiplication Using Compressed Sparse Rows. *Journal of Signal Processing Systems*, pages 1–18, 2016. [page 6]

[Liu16b] X. Liu, Z. Zhong, and K. Xu. A Hybrid Solution Method for CFD Applications on GPU-accelerated Hybrid HPC Platforms. *Future Generation Computer Systems*, 56:759–765, 2016. [page 9]

[Liu16c] L. Liu, L. Ci, W. Liu, et al. Control-Flow Checking Using Branch Sequence Signatures. In *IEEE International Conference on IEEE Cyber, Physical and Social Computing (CPSCom)*, pages 839–845. IEEE, 2016. [page 13]

[Liu16d] W. Liu, L. Chen, C. Wang, M. O'Neill, and F. Lombardi. Design and Analysis of Inexact Floating-Point Adders. *IEEE Transactions on Computers*, 65(1):308–314, 2016. [page 48]

[Liu17] W. Liu, L. Qian, C. Wang, H. Jiang, J. Han, and F. Lombardi. Design of Approximate Radix-4 Booth Multipliers for Error-Tolerant Computing. *IEEE Transactions on Computers*, 2017. [pages 48 and 54]

[Loh16] F. Loh, K. K. Saluja, and P. Ramanathan. Fault Tolerance through Invariant Checking for Iterative Solvers. In *29th International Conference on VLSI Design and 15th International Conference on Embedded Systems (VLSID)*, pages 481–486. 2016. [pages 29, 41, and 42]

[Lopez15] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso. A Survey of Performance Modeling and Simulation Techniques for Accelerator-based Computing. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):272–281, 2015. [page 1]

[Luk86] F. T. Luk. Algorithm-based Fault Tolerance for Parallel Matrix Equation Solvers. In *Real-Time Signal Processing VIII*, pages 49–56. 1986. [page 32]

[Mahdi10] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired Imprecise Computational Blocks for Efficient VLSI Implementation of Soft-computing Applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(4):850–862, 2010. [page 48]

[Marti15] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. Navaux. Towards Seismic Wave Modeling on Heterogeneous Many-core Architectures Using Task-based Runtime System. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on*, pages 1–8. IEEE, 2015. [page 8]

[Marti16] H. Martínez, S. Barrachina, M. Castillo, J. Tárraga, I. Medina, J. Dopazo, and E. S. Quintana-Ortí. A Framework for Genomic Sequencing on Clusters of

Multicore and Manycore Processors. *International Journal of High Performance Computing Applications*, page 1094342016653243, 2016. [page 8]

[Matsu98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998. [pages 113 and 114]

[Mazah16] S. Mazahir, O. Hasan, R. Hafiz, M. Shafique, and J. Henkel. An Area-efficient Consolidated Configurable Error Correction for Approximate Hardware Accelerators. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. 2016. [page 48]

[McAfe15] L. McAfee and K. Olukotun. EMEuro: A Framework for Generating Multipurpose Accelerators Via Deep Learning. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 125–135. 2015. [page 49]

[McClu14] J. E. McClure, J. F. Prins, and C. T. Miller. A Novel Heterogeneous Algorithm to Simulate Multiphase Flow in Porous Media on Multicore CPU-GPU Systems. *Computer Physics Communications*, 185(7):1865–1874, 2014. [page 9]

[Miao12] J. Miao, K. He, A. Gerstlauer, and M. Orshansky. Modeling and Synthesis of Quality-energy Optimal Approximate Adders. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 728–735. 2012. [page 48]

[Miao13] J. Miao, A. Gerstlauer, and M. Orshansky. Approximate Logic Synthesis under General Error Magnitude and Frequency Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 779–786. 2013. [page 48]

[Miao16] X. Miao, X. Jin, and J. Ding. An Approach to Enhance the Performance of Large-scale Structural Analysis on CPU-MIC Heterogeneous Clusters. *Concurrency and Computation: Practice and Experience*, 2016. [page 9]

[Migue14] J. S. Miguel, M. Badr, and N. E. Jerger. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 127–139. 2014. [pages 15 and 48]

[Mishr11] P. Mishra, A. Muttreja, and N. K. Jha. FinFET circuit design. In *Nanoelectronic Circuit Design*, pages 23–54. Springer, 2011. [page 14]

[Mishr14] A. K. Mishra, R. Barik, and S. Paul. iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*. 2014. [pages 52, 53, 56, 98, and 105]

[Mitra11] S. Mitra, K. Brelsford, Y. M. Kim, H.-H. K. Lee, and Y. Li. Robust System Design to Overcome CMOS Reliability Challenges. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 1(1):30–41, 2011. [page 2]

[Mitta15] S. Mittal and J. S. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys (CSUR)*, 47(4):69, 2015. [pages 7 and 43]

[Mitta16a] S. Mittal. A Survey of Techniques for Approximate Computing. *ACM Computing Surveys (CSUR)*, 48(4):62, 2016. [page 13]

[Mitta16b] S. Mittal and J. S. Vetter. A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(4):1226–1238, 2016. [pages 80 and 114]

[Moore65] G. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965. [page 2]

[Morea15] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 603–614. 2015. [pages 48 and 49]

[Mukhe11] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, 2011. [pages 10, 11, and 28]

[Mulle10] J.-M. Muller, S. Torres, R. Nathalie, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, and D. Stehle. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010. [pages 66 and 153]

[NVIDI17] NVIDIA Corporation. CUDA Software Development Kit (SDK), 2017. URL `https://developer.nvidia.com/cuda-downloads`. [page 44]

[Namha16] A. Namhata, S. Oladyshkin, R. M. Dilmore, L. Zhang, and D. V. Nakles. Probabilistic Assessment of Above Zone Pressure Predictions at a Geologic Carbon Storage Site. *Scientific Reports*, 6, 2016. [page 10]

[Nanga] Nangate Inc. 45nm open cell library. *http://www.nangate.com.* [pages 104 and 115]

[Nanju10] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC simulator on GPUs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 149–154. 2010. [page 8]

[Nanu14] D. Nanu, P. Roshini, D. Sowkarthiga, and K. S. Al Ameen. Approximate Adder Design Using CPL Logic for Image Compression. *International Journal of Innovative Research and Development*, 2014. [page 48]

[Nebel13] W. Nebel and J. Mermet. *Low Power Design in Deep Submicron Electronics*, volume 337. Springer Science & Business Media, 2013. [pages 44 and 46]

[Nepal14] K. Nepal, Y. Li, R. I. Bahar, and S. Reda. ABACUS: A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 361:1–361:6. 2014. [page 48]

[Ni16] P. Ni and S. Law. Hybrid Computational Strategy for Structural Damage Detection with Short-term Monitoring Data. *Mechanical Systems and Signal Processing*, 70:650–663, 2016. [page 9]

[Nicol11] M. Nicolaidis. Soft Errors In Modern Electronic Systems. In *Frontiers in Electronic Testing*. Springer, 2011. [pages 12 and 28]

[Nvidia] Nvidia Corporation. cuBLAS Tookit Documentation. http://docs.nvidia.com/cuda/cublas/. [page 113]

[Nvidib] Nvidia Corporation. cuSPARSE Tookit Documentation. http://docs.nvidia.com/cuda/cusparse/. [page 113]

[Oberk10] W. L. Oberkampf and C. J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, 2010. [pages 1, 6, and 10]

[Obori11] F. Oboril, M. Tahoori, V. Heuveline, D. Lukarski, and J.-P. Weiss. Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 144–153. 2011. [page 41]

[Oh02] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002. [page 13]

[Oh04] K.-S. Oh and K. Jung. GPU Implementation of Neural Networks. *Pattern Recognition*, 37(6):1311–1314, 2004. [page 9]

[Oluko98] K. Olukotun, M. Heinrich, and D. Ofelt. Digital System Simulation: Methodologies and Examples. In *Proceedings of the ACM/IEEE-CAS/EDAC Design Automation Conference (DAC)*, pages 658–663. 1998. [page 98]

[Patte14] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (5th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2014. [page 47]

[Pradh96] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, 1996. [pages 2, 12, 30, and 151]

[Pullu01] L. L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House, 2001. [pages 2, 13, and 26]

[Puzyr13] V. Puzyrev, J. Koldan, J. de la Puente, G. Houzeaux, M. Vázquez, and J. M. Cela. A parallel Finite-element Method for Three-dimensional Controlled-source Electromagnetic Forward Modelling. *Geophysical Journal International*, 193(2):678–693, 2013. [page 5]

[Rabae12] J. M. Rabaey and M. Pedram. *Low Power Design Methodologies*. Springer Science & Business Media, 2012. [page 45]

[Raha17] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan. Energy-Efficient Reduce-and-Rank Using Input-Adaptive Approximations. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(2):462–475, 2017. [page 47]

[Rahim13] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A Variability-aware OpenMP Environment for Efficient Execution of Accuracy-configurable Computation on Shared-FPU Processor Clusters. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10. 2013. [page 49]

[Ranga09] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread Motion: Fine-grained Power Management for Multi-core Systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM, 2009. [page 14]

[Ranja14] R. Ranjan. Streaming Big Data Processing in Datacenter Clouds. *IEEE Cloud Computing*, 1(1):78–83, 2014. [page 9]

[Regul16] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. Kelly, and D. Radford. Acceleration of a Full-scale Industrial CFD Application with OP2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1265–1278, 2016. [page 9]

[Reid72] J. K. Reid. The Use of Conjugate Gradients for Systems of Linear Equations Possessing "Property A". *SIAM Journal on Numerical Analysis*, 9(2):325–332, 1972. [page 22]

[Resch17] M. Resch, A. Kaminski, and P. Gehring. The Science and Art of Simulation I: Exploring-Understanding-Knowing, 2017. [page 6]

[Rethi14] S. K. Rethinagiri, O. Palomar, O. S. Unsal, A. Cristal, R. B. Atitallah, and S. Niar. PETS: Power and Energy Estimation Tool at System-level. In *Fifteenth International Symposium on Quality Electronic Design, (ISQED)*, pages 535–542. 2014. [pages 54 and 98]

[Rexfo94] J. Rexford and N. Jha. Partitioned Encoding Schemes for Algorithm-Based Fault Tolerance in Massively Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):649–653, 1994. [pages 34 and 58]

[Ringe15] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and Debugging the Quality of Results in Approximate Programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411. 2015. [page 46]

[Rodri16] L. Rodríguez Gómez and H.-J. Wunderlich. A Neural-Network-Based Fault Classifier. In *Proceedings of the 25th IEEE Asian Test Symposium (ATS'16)*, pages 144–149. 2016. [page 9]

[Roten16] D. Roten, Y. Cui, K. B. Olsen, S. M. Day, K. Withers, W. H. Savran, P. Wang, and D. Mu. High-frequency Nonlinear Earthquake Simulations on Petascale Heterogeneous Supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 82. IEEE Press, 2016. [page 8]

[Roy14] P. Roy, R. Ray, C. Wang, and W. F. Wong. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In *ACM SIGPLAN Notices*, volume 49, pages 95–104. 2014. [pages 52 and 53]

[Rupp10] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL-a High Level Linear Algebra Library for GPUs and multi-core CPUs. In *International Workshop on GPUs and Scientific Applications*, pages 51–56. 2010. [page 44]

[Saad03] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Siam, 2003. [pages 5, 21, 23, 50, 84, and 149]

[Samad13] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24. 2013. [page 49]

[Samad14] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, pages 35–50. 2014. [page 48]

[Samps11] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power

Computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. 2011. [pages 49, 54, and 96]

[Sao13] P. Sao and R. Vuduc. Self-stabilizing Iterative Solvers. In *Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 4:1–4:8. 2013. [pages 41 and 42]

[Sarwa16] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy. Multiplier-less Artificial Neurons Exploiting Error Resiliency for Energy-efficient Neural Computing. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 145–150. 2016. [page 50]

[Schaf14] M. Schaffner, F. K. Gürkaynak, A. Smolic, H. Kaeslin, and L. Benini. An Approximate Computing Technique for Reducing the Complexity of a Direct-solver for Sparse Linear Systems in Real-time Video Processing. In *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*, pages 1–6. 2014. [pages 47, 50, and 84]

[Schne16] E. Schneider, M. A. Kochte, S. Holst, X. Wen, and H.-J. Wunderlich. GPU-Accelerated Simulation of Small Delay Faults. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(5), 2016. [page 8]

[Schol14] A. Schöll, C. Braun, M. Daub, G. Schneider, and H.-J. Wunderlich. Adaptive Parallel Simulation of a Two-Timescale-Model for Apoptotic Receptor-Clustering on GPUs. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'14)*, pages 424–431. 2014. [page 8]

[Schol15] A. Schöll, C. Braun, M. A. Kochte, and H.-J. Wunderlich. Low-Overhead Fault-Tolerance for the Preconditioned Conjugate Gradient Solver. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'15)*, pages 60–66. October 2015. [page 73]

[Schol16a] A. Schöll, C. Braun, M. A. Kochte, and H.-J. Wunderlich. Efficient Algorithm-Based Fault Tolerance for Sparse Matrix Operations. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, pages 251–262. 2016. [page 57]

[Schol16b] A. Schöll, C. Braun, and H.-J. Wunderlich. Applying Efficient Fault Tolerance to Enable the Preconditioned Conjugate Gradient Solver on Approximate Computing Hardware. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'16)*, pages 21–26. 2016. [page 83]

[Segal88] Z. Segall, D. F. Vrsalovic, D. P. Siewiorek, D. A. Yaskin, J. Kownacki, J. H. Barton, R. Dancey, A. Robinson, and T. Lin. FIAT-Fault Injection Based Automated Testing Environment. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 102–107. 1988. [page 29]

[Segur04] J. Segura and C. F. Hawkins. *CMOS Electronics: How It Works, How It Fails*. John Wiley & Sons, 2004. [page 11]

[Semer02] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Eighth International Symposium on High-Performance Computer Architecture*, pages 29–40. 2002. [page 14]

[Senn04] E. Senn, J. Laurent, N. Julien, and E. Martin. SoftExplorer: Estimation, Characterization, and Optimization of the Power and Energy Consumption at the Algorithmic Level. In *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 342–351. 2004. [pages 53 and 54]

[Shafi16] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel. Invited: Cross-layer Approximate Computing: From Logic to Architectures. In *53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. 2016. [pages 15, 47, and 48]

[Shant11] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the Impact of Soft Errors on Iterative Methods in Scientific Computing. In *Proceedings of the International Conference on Supercomputing*, pages 152–161. 2011. [pages 10, 41, 66, and 74]

[Shant12] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution. In

Proceedings of the ACM International Conference on Supercomputing, pages 69–78. 2012. [pages 32, 37, 38, 68, and 116]

[Shen16] Y. Shen and C. E. Cesnik. Hybrid Local FEM/global LISA Modeling of Damped Guided Wave Propagation in Complex Composite Structures. *Smart Materials and Structures*, 25(9):095021, 2016. [page 9]

[Shin16] C. Shin. Variation-aware Advanced CMOS Devices and SRAM. *Springer Series in Advanced Microelectronics*, 56, 2016. [page 11]

[Sidir11] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance Vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 124–134. 2011. [pages 15 and 49]

[Sloan12] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12. 2012. [pages 37, 38, 40, 51, 68, 114, and 116]

[Sloan13] J. Sloan, R. Kumar, and G. Bronevetsky. An Algorithmic Approach to Error Localization and Partial Recomputation for Low-Overhead Fault Tolerance. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*, pages 1–12. 2013. [pages 13, 36, 37, 38, 40, 68, 114, 116, 118, and 119]

[Smith13] I. Smith, D. Griffiths, and L. Margetts. *Programming the Finite Element Method.* Wiley, 4 edition, 2013. [page 5]

[Soeke16] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler. BDD Minimization for Approximate Computing. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 474–479. IEEE, 2016. [page 48]

[Song16] M. Song, W. Li, W. Li, E. Liu, and D. Yu. A Hybrid Parallel Computing Model to Support Scalable Processing of Big Oceanographic Spatial Data. In *International Conference on Geo-Informatics in Resource Management and Sustainable Ecosystems*, pages 276–285. Springer, 2016. [page 8]

[Stone10] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–73, 2010. [page 44]

[Subra13] B. Subramaniam, W. Saunders, T. Scogland, and W.-C. Feng. Trends in Energy-efficient Computing: A Perspective from the Green500. In *International Green Computing Conference (IGCC)*, pages 1–8. 2013. [page 13]

[Suraa14] M. P. R. Suraana and N. Thoutam. A Review on Evaluation of Multi-level Checkpointing System in Distributed Environment. *Intl. Journal of Electronics, Communication and Soft Computing Science & Engineering (IJECSCSE)*, 3(7):25–32, 2014. [page 13]

[Tagli16] G. Tagliavini, A. Marongiu, D. Rossi, and L. Benini. Always-on Motion Detection with Application-level Error Control on a Near-threshold Approximate Computing Platform. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 552–555. 2016. [page 47]

[Tao16] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen. New-Sum: A Novel Online ABFT Scheme for General Iterative Methods. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 43–55. 2016. [pages 41, 42, and 114]

[Thomp79] C. D. Thompson. Area-Time Complexity for VLSI. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 81–88. 1979. [page 44]

[Tiwar15a] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, and P. Navaux. Understanding GPU Errors on Large-Scale HPC Systems and the Implications for System Design and Operation. In *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. 2015. [page 28]

[Tiwar15b] D. Tiwari, S. Gupta, G. Gallarno, J. Rogers, and D. Maxwell. Reliability Lessons Learned from GPU Experience with the Titan Supercomputer at

Oak Ridge Leadership Computing Facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 38–50. 2015. [page 30]

[Tiwar94] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design ICCAD*, pages 384–390. 1994. [pages 53, 54, and 98]

[Tiwar96] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee. Instruction Level Power Analysis and Optimization of Software. In *International Conference on VLSI Design*, pages 326–328. 1996. [page 53]

[Valer17] P. Valero-Lara and J. Jansson. Heterogeneous CPU+GPU approaches for mesh refinement over Lattice-Boltzmann simulations. *Concurrency and Computation: Practice and Experience*, 29(7), 2017. [page 9]

[Van R79] C. Van Rijsbergen. *Information Retrieval.* Butterworths, 1979. [page 120]

[Vassi15] V. Vassiliadis, K. Parasyris, C. Chalios, C. D. Antonopoulos, S. Lalis, N. Bellas, H. Vandierendonck, and D. S. Nikolopoulos. A Programming Model and Runtime System for Significance-Aware Energy-Efficient Computing. *SIGPLAN Not.*, 50(8):275–276, 2015. [page 49]

[Venka11] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: Modeling and Analysis of Circuits for Approximate Computing. In *Proceedings of tje IEEE/ACM International Conference Computer-Aided Design (ICCAD)*, pages 667–673. 2011. [page 48]

[Venka12] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. SALSA: Systematic Logic Synthesis of Approximate Circuits. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 796–801. 2012. [page 48]

[Venka13a] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality Programmable Vector Processors for Approximate Computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–12. 2013. [pages 3, 49, and 96]

[Venka13b] S. Venkataramani, K. Roy, and A. Raghunathan. Substitute-and-simplify: A Unified Design Paradigm for Approximate and Quality Configurable Circuits. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 1367–1372. 2013. [page 48]

[Venka14] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan. AxNN: Energy-efficient Neuromorphic Systems Using Approximate Computing. In *Proceedings of the IEEE/ACM International Symposium Low Power Electronics and Design (ISLPED)*, pages 27–32. 2014. [page 50]

[Venka15] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan. Approximate Computing and the Quest for Computing Efficiency. In *Proceedings of the 52nd ACM/EDAC/IEEE Annual Design Automation Conference (DAC'15)*, page 120. 2015. [pages 3, 15, and 47]

[Venka16] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve. Approxilyzer: Towards a Systematic Framework for Instruction-level Approximate Computing and Its Application to Hardware Resiliency. In *Proceedings of the 49th Annual IEEE/ACM International Symposium of Microarchitecture (MICRO)*, pages 1–14. 2016. [pages 52 and 53]

[Verma15] A. K. Verma, P. Gautam, T. Singh, and R. Bajpai. Numerical Simulation of High Level Radioactive Waste for Disposal in Deep Underground Tunnel. In *Engineering Geology for Society and Territory-Volume 1*, pages 499–504. Springer, 2015. [page 10]

[Vinco12] S. Vinco, V. Bertacco, D. Chatterjee, and F. Fummi. SAGA: SystemC Acceleration on GPU Architectures. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC'12)*, pages 115–120. 2012. [page 8]

[Von N56] J. Von Neumann. Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components. *Automata studies*, 34:43–98, 1956. [page 44]

[Wang14] Z. Wang. High-order Computational Fluid Dynamics Tools for Aircraft Design. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 372(2022):20130318, 2014. [page 9]

[Wang16] Q. Wang, Y. Li, and P. Li. Liquid State Machine Based Pattern Recognition on FPGA with Firing-Activity Dependent Power Gating and Approximate Computing. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 361–364. 2016. [page 47]

[Weste15] N. H. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson India, 2015. [page 12]

[Wilke14] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli. Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 293–305. 2014. [page 28]

[Wozni16] B. D. Wozniak, F. D. Witherden, F. P. Russell, P. E. Vincent, and P. H. Kelly. Gimmik–generating bespoke matrix multiplication kernels for accelerators: Application to high-order computational fluid dynamics. *Computer Physics Communications*, 202:12–22, 2016. [page 5]

[Wu12] X. Wu, A. Koslowski, and W. Thiel. Semiempirical Quantum Chemical Calculations Accelerated on a Hybrid Multicore CPU-GPU Computing Platform. *Journal of Chemical Theory and Computation*, 8(7):2272–2281, 2012. [page 8]

[Wu14] P. Wu and Z. Chen. FT-ScaLAPACK: Correcting soft errors on-line for ScaLAPACK Cholesky, QR, and LU factorization routines. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 49–60. 2014. [page 32]

[Wu16] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen. Towards Practical Algorithm Based Fault Tolerance in Dense Linear Algebra. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 31–42. 2016. [pages 29 and 32]

[Wunde03] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–95. 2003. [pages 55, 56, 98, 99, and 107]

[Wunde10] H.-J. Wunderlich and S. Holst. *Generalized Fault Modeling for Logic Diagnosis*, volume 43, pages 133–155. Springer-Verlag Heidelberg, 2010. [page 29]

[Xu12] M. Xu, F. Chen, X. Liu, W. Ge, and J. Li. Discrete Particle Simulation of Gas-solid Two-phase Flows with Multi-scale CPU-GPU Hybrid Computation. *Chemical Engineering Journal*, 207:746–757, 2012. [page 8]

[Yang13a] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng. A Peta-scalable CPU-GPU Algorithm for Global Atmospheric Simulations. In *ACM SIGPLAN Notices*, volume 48, pages 1–12. 2013. [page 8]

[Yang13b] Z. Yang, A. Jain, J. Liang, J. Han, and F. Lombardi. Approximate XOR/XNOR-based Adders for Inexact Computing. In *Proceedings of the IEEE Conference on Nanotechnology (IEEE-NANO'13)*, pages 690–693. 2013. [pages 48 and 54]

[Yazda15] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language Support for Approximate Hardware Design. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 812–817. 2015. [page 48]

[Yazda16a] A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry. Mitigating the Memory Bottleneck With Approximate Load Value Prediction. *IEEE Design & Test*, 33(1):32–42, 2016. [page 48]

[Yazda16b] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):62:1–62:26, 2016. [page 48]

[Yin16] P. Yin, C. Wang, W. Liu, and F. Lombardi. Design and Performance Evaluation of Approximate Floating-Point Multipliers. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 296–301. 2016. [page 48]

[Yonke16] N. Yonkee and J. C. Sutherland. PoKiTT: Exposing Task and Data Parallelism on Heterogeneous Architectures for Detailed Chemical Kinetics, Transport, and Thermodynamics Calculations. *SIAM Journal on Scientific Computing*, 38(5):S264–S281, 2016. [page 8]

[Zabel15] S. Zabelok, R. Arslanbekov, and V. Kolobov. Adaptive Kinetic-fluid Solvers for Heterogeneous Computing Architectures. *Journal of Computational Physics*, 303:455–469, 2015. [page 9]

[Zhang14a] Q. Zhang, F. Yuan, R. Ye, and Q. Xu. ApproxIt: An Approximate Computing Framework for Iterative Methods. In *Proceedings of the 51st ACM/IEEE Design Automation Conference (DAC'14)*, pages 1–6. 2014. [pages 46, 48, 50, 56, 83, 84, and 99]

[Zhang14b] H. Zhang, W. Zhang, and J. Lach. A Low-power Accuracy-configurable Floating Point Multiplier. In *32nd IEEE International Conference on Computer Design (ICCD)*, pages 48–54. 2014. [pages 48 and 115]

[Zhang15a] Q. Zhang, Y. Tian, T. Wang, F. Yuan, and Q. Xu. ApproxEigen: An Approximate Computing Technique for Large-Scale Eigen-Decomposition. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 824–830. 2015. [pages 46, 50, 51, and 84]

[Zhang15b] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. ApproxANN: An Approximate Computing Framework for Artificial Neural Network. In *Proceedings of the Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 701–706. 2015. [pages 48 and 50]

Index

Publications of the Author

- C. Braun, M. Daub, A. Schöll, G. Schneider and H.-J. Wunderlich, Parallel Simulation of Apoptotic Receptor-Clustering on GPGPU Many-Core Architectures, *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'12)*, Philadelphia, Pennsylvania, USA, 4-7 October 2012, pp. 1-6.

- A. Schöll, C. Braun, M. Daub, G. Schneider and H.-J. Wunderlich, Adaptive Parallel Simulation of a Two-Timescale-Model for Apoptotic Receptor-Clustering on GPUs, *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM'14)*, Belfast, United Kingdom, 2-5 November 2014, pp. 424-431, SimTech Best Paper Award.

- A. Schöll, C. Braun, M.A. Kochte and H.-J. Wunderlich, Efficient On-Line Fault-Tolerance for the Preconditioned Conjugate Gradient Method, *Proceedings of the 21st IEEE International On-Line Testing Symposium (IOLTS'15)*, Elia, Halkidiki, Greece, 6-8 July 2015, pp. 95-100.

- A. Schöll, C. Braun, M.A. Kochte and H.-J. Wunderlich, Low-Overhead Fault-Tolerance for the Preconditioned Conjugate Gradient Solver, *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'15)*, Amherst, Massachusetts, USA, 12-14 October 2015, pp. 60-65.

- H.-J. Wunderlich, C. Braun and A. Schöll, Fault Tolerance of Approximate Compute Algorithms, *Proceedings of the 34th VLSI Test Symposium (VTS'16)*, Caesars Palace, Las Vegas, Nevada, USA, 25-27 April 2016.

- A. Schöll, C. Braun, M.A. Kochte and H.-J. Wunderlich, Efficient Algorithm-Based Fault Tolerance for Sparse Matrix Operations, *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, Toulouse, France, 28 June-1 July 2016, pp. 251-262.

- H.-J. Wunderlich, C. Braun and A. Schöll, Pushing the Limits: How Fault Tolerance Extends the Scope of Approximate Computing, *Proceedings of the 22nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS'16)*, Sant Feliu de Guixols, Catalunya, Spain, 4-6 July 2016, pp. 133-136.

- A. Schöll, C. Braun and H.-J. Wunderlich, Applying Efficient Fault Tolerance to Enable the Preconditioned Conjugate Gradient Solver on Approximate Computing Hardware, *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT'16)*, University of Connecticut, USA, 19-20 September 2016, pp. 21-26, DFT 2016 Best Paper Award.

- A. Schöll, C. Braun and H.-J. Wunderlich, Hardware/Software Co-Characterization for Approximate Computing, *Workshop on Approximate Computing*, Pittsburgh, Pennsylvania, USA, 6 October 2016.

- A. Schöll, C. Braun and H.-J. Wunderlich, Energy-efficient and Error-resilient Iterative Solvers for Approximate Computing, *Proceedings of the 23nd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS'17)*, Thessaloniki, Greece, 3-5 July, 2017.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Alexander Schöll