Soft Error Resilient Time Stepping Schemes on Regular Grids

Master Thesis

Author(s):
Kohler, Manuel Christoph

Publication date:
2016

Permanent link:
https://doi.org/10.3929/ethz-a-010583223

Soft Error Resilient Time Stepping Schemes on Regular Grids

Master's Thesis by

Manuel Christoph Kohler
manuel.christoph.kohler@alumni.ethz.ch

supervised by

Prof. Dr. Peter Arbenz
arbenz@inf.ethz.ch

submitted to

Department of Computer Science
ETH Zurich

January, 12. 2016

Abstract

Monte Carlo (MC) and multilevel Monte Carlo (MLMC) methods are used for estimating statistical moments of random variables. Large cluster computer suitable for such calculations are subject to frequent failures due to their myriad of components. The problem is currently solved by checkpoint/restart, i.e., by periodically taking a snapshot of the application's state and restarting from the last one whenever an error has been detected. This solution is projected to come to a halt in the near future because of unfortunate developments of failure rate, snapshot size and the storage bandwidth. Therefore, application-specific solutions are taken into account for exascale computing. For MC methods, we already investigated the behavior regarding fail-stop failures. But another development concerning the high performance computing (HPC) community is the occurrence of soft errors. These are bits being flipped occasionally mainly in the main memory and caches due to neutron and alpha particle strikes.

We propose two soft error models based on empirical evidence, and a combination of principles for resilience against them. A tool for artificial soft error injection according to both error model is provided, allowing the investigation of an application's resilience. The principles for soft error resilience include triple modular redundancy (TMR) for masking errors in small chunks of data, error detection in large blocks of data through checksums, and, measures for recovery after error detection. The principles are well-suited for time-stepping schemes on regular grids like they are often appearing in MC and MLMC simulations. Following two major implementation approaches, experiments are conducted with a simple partial differential equation (PDE) solver. Depending on the error model and the chosen approach, the method reveals a high soft error resilience with a low resource overhead.

Contents

1 Introduction

This introductory section discuss the background of the thesis in Section 1.1 and embeds it into theory as well as related work in Section 1.2 and 1.3, respectively. In Section 2, we introduce our new method for soft error resilient applications and elaborate on the assumptions and principles. Section 3 is dedicated to an implementation and a tool for testing the implementation under the influence of errors. Empirical results of these programs are presented in Section 4. In Section 5, we discuss the outcome of this project and point towards future research. Finally, we conclude our work in Section 6.

1.1 Background

Monte Carlo (MC) methods are used for estimating statistical moments of random variables. An MC simulation consists of many samples which are the solutions of partial differential equations (PDEs) with randomly drawn initial and boundary condition. The expected solution with respect to the distribution of the random conditions is approximated by averaging over the samples. MC methods suffer from a slow convergence rate which can be improved by the recently proposed multilevel Monte Carlo (MLMC) methods [19], though, still requiring excessive computational resources. MC and MLMC methods have inherent fault-tolerant properties [39] which we have investigated in various projects [29, 30, 40, 41]. In this projects, we were always concerned with fail-stop failures, in other words, a process participating in the calculation of a sample either succeeds or crashes. Other fault models are possible, in particular, soft errors have caught attention by the high performance computing (HPC) community. Soft errors are the manifestation of transient hardware faults and often observed as bits in the main memory or the caches which flip their value. The effect is only noticeable in large simulations such as MLMC which claim a considerable amount of memory over a long period of time. Systems often survive soft errors but the result may have an obvious corruption or a subtle deviation from the correct solution. The goal is to explore measures to detect and hopefully correct soft errors in MLMC methods. In this project, we take a first step by investigating soft error resilience in MC and MLMC samples, or more precisely, in time-stepping schemes on regular girds.

1.2 Dependable Computing

We recapitulate the basic concepts of dependable computing and topics specific to soft errors by following the structures and notions of Avizienis et al. [2], Dubrova [10], Lee & Anderson [32], Mukherjee [37], and Benso and Prinetto [3]. Section 1.2.1 contains an overview of the terms and concepts related to faults, errors and failures in general and soft errors in particular. In Section 1.2.2, possible solutions for these problems are sketched. Finally, in Section 1.2.3, we cover the three basic methods of error injection for testing purposes.

1.2.1 Terms and Definitions

A *system* consists of *components* which interact according to a system *design*. Every component is itself either a system or *atomic*. Anything outside of the system is part of the *environment* and interacts with the system through its *interface* [32, pp. 13-17].

The intended behavior of the system's interface is recorded in a *specification* [2, pp. 12-13]. A system is said to be *correct* if it behaves according to its specification. The specification could itself be erroneous, but we assume that it is *exact*, i.e., the specification is authoritative to determine the correctness of the system [32, p. 33].

Any deviation from the specification is called a *failure* [2, p. 13]. Systems equipped with sanity checks may *signal* a problem to the environment instead of failing silently. As a consequence, the problem of *false alarms* is introduced, that is, a problem is signaled but the behavior would be indeed correct.

Failures are part of the system's interface and are caused by erroneous internal states called *errors* [2, p. 13]. Errors itself are caused by *faults* which are the cause of all trouble [2, p. 13]. They are categorized according to various viewpoints into system and environment faults, design and component faults, hardware and software faults, transient and permanent faults, malicious and non-malicious faults, etc. [2, pp. 7-12]. Following the recursive definition of a system, a failure of a component may itself be regarded as a fault in the enclosing system.

Important for this work are *transient hardware faults* that remain active in a bounded period of time. Two major sources are alpha particles and neutrons [37, pp. 20-30]. Alpha particles often arise from radioactive impurities in the chip's packaging material. Neutron strikes have their origin in cosmic rays. Though in different ways, a charge is inducted into the circuitry by these causes. If it exceeds the critical charge of a transistor, a faulty bit is produced, i.e., an error in the system. Transient hardware faults are often only observed by their manifestation in these so-called *soft errors*.

Masking describes the phenomenon that a soft error in a component does not necessarily cause its failure and, thus, induces a fault in the enclosing system [37, pp. 52-59]. It occurs at various levels of a computer system: On a circuitry level, the soft error may be masked logically, e.g., it affects the input of an `OR` gate which has another high input anyway. On a chip level, a soft error in an inactive processing unit is masked trivially. Similarly, a soft error in the memory hierarchy is masked when the corresponding datum is overwritten or never read.

Every part of a circuit may be subject to soft errors. The impact on a component depends on its size and masking rate [37, pp. 36-38]. For example, a soft error in the clock signal of circuitries could have dramatic consequences, but those are rarely hit because of their small spatial extent. The same is true for registers and other small components. The masking rates for logical circuits and of whole processing units are normally very high (cf. Section 1.3), hence, they are less vulnerable despite their size. In contrast, SRAM cells (for caches) and DRAM cells (for main memory) have a small masking rate and occupy large portions of the chip's size. The result is a comparably high rate of soft errors in SRAM and DRAM devices. Hardware measures were introduced to overcome this problem: Cache lines of SRAM chips are often protected with *error detecting codes (EDC)* and some DRAM chips are even protected with *error correcting codes (ECC)*. But the vulnerability increases because the charge held by an SRAM or DRAM cell decreases as technology evolves. The high density of cells in a chip may also lead to multiple bits being flipped from one particle strike. This is called *multi cell upset (MCU)* as opposed to *single event upsets (SEU)* where only a single bit is changed [52, p. 11]. Multiple SEU or MCUs may slip through hardware protection and become part of the applications state as soft errors.

The rate at which soft errors are observed is called *soft error rate (SER)* and measured in *failures in time (FIT)* or *mean time between failures (MTBF)* [37, pp. 9-11]. The FIT is defined as the number of soft errors in one billion hours of operation. This unit has the property that the rate of a system is the sum of the rates of its components. Though inconsistent with the nomenclature introduced here, the MTBF is the mean time between two successive soft errors.

Soft errors may be categorized by their impact on fault tolerant systems (cf. Section 1.2.2). A soft error is called *benign* if the faulty bit is masked (naturally or by fault tolerant methods). *Silent data corruptions (SDCs)* are errors which remain undetected and lead to failures. *Detected unrecoverable errors (DUEs)* are errors which are detected and signaled.

1.2.2 Fault Tolerance and Redundancy

Dependability is the ability of a system to provide its service with an acceptable failure rate [2, p. 5]. This goal is approached through concepts such as fault prevention, fault removal and fault forecasting [10, pp. 16-17]. Another major part of dependable computing is concerned with *fault tolerance* or *fault resilience*, i.e., the design of systems capable of continuing their services in spite of faults. Paradoxically, fault tolerance increases the complexity of the system and, therefore, the probability of a design or implementation fault. Thus, strategies for fault tolerance must be carefully considered to actually increase the system's dependability [32, p. 20, p. 60].

Because it is difficult to anticipate every possible impact of faults in a system, they are assumed to behave according to a *fault model* [10, pp. 13-14]. A fundamental part of the fault model is the identification of unreliable components. Those are said to lie in the *sphere of replication (SoR)* and everything passing through the SoR must be protected in order to provide resilience according to the model [51].

Two important aspects of protection are fault detection and recovery [10, pp. 15-16]. *Fault detection* is a measure to detect the occurrence of faults or their manifestation in errors based on tests. *Recovery* means the system is restored to an error-free state after a fault or error has been detected. If a system is capable of providing its services despite faults, it *masks* the faults.

Common to most fault tolerant systems is the fact that they exploit some kind of *redundancy*, that is "the provision of functional capabilities that would be unnecessary in a fault-free environment" [10, p. 2]. Redundancy can further be divided into time and space redundancy. *Time redundancy* means an action is repeated. *Space redundancy* refers to the existence of components solely for fault tolerance. It is further split into hardware, software and information redundancy indicating the type of these extra components. *Information redundancy* achieves fault-tolerance through *coding*, i.e., the representation of an object for transmission or storage [32, pp. 106-108].

A standard approach for fault tolerance is checkpoint/restart in which a snapshot of the application's state is pushed to permanent storage from time to time [13]. When an error is detected, the system recovers by deploying the saved state and restarting from the last checkpoint. With the growing size of HPC systems, the MTBF decreases, causing a higher checkpoint frequency. At the same time, the duration for creating and restoring a checkpoints increases as the ratio between the checkpoint size and the storage bandwidth deteriorates. Eventually, HPC system with checkpoint/restart will have an efficiency close to zero when the system is permanently creating and recovering from checkpoints. This phenomenon is well-studied and expected to become reality in the near future [48].

1.2.3 Fault Injection

As any software, fault tolerant software must be tested. It is often impractical to wait for real faults to happen, hence, faults have to be introduced artificially into the system. This is often referred to as *fault injection* [3, p. 1]. The three major methods for fault injection are introduced below.

In the *hardware-based fault injection*, the hardware is augmented with tools that inject the faults [3, pp. 28-31]. This is the most realistic but also the most expensive method. No error model is required and the tests run in real-time because the faults injected are indistinguishable from real faults. However, it requires special purpose hardware, physical access to the hardware under test and might even damage the system.

The *software-based fault injection* approach is used mainly for software testing and works by modifying the state of the system either by the system itself or by another system running in parallel [3, pp. 31-33]. These methods run in near real-time but do not allow accurate timing measurements. The fault model is limited to entities visible in software, e.g., memory and registers. In contrast, caches or processing units are out of scope.

Finally, *simulation-based fault injection* is done by simulating the entire system in software [3, pp. 33-34]. This method requires the development of a simulator. Simulators often run slower than the real execution but allow faults to be injected according to any fault model.

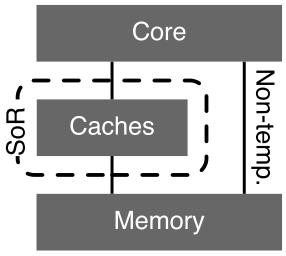
1.3 Related Work

Excellent introductions to the current and future problems of high performance computing (HPC) are provided by Snir et al. [52], Schroeder and Gibson [48], Besseron et al. [5], and Capello et al. [7, 8].

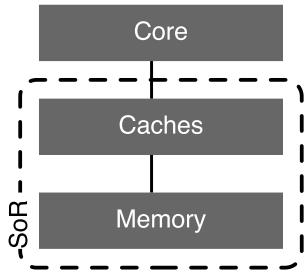
Despite their threat for future HPC systems, it is difficult to reach concrete failure rates for soft errors. Different error sources, methodologies in measurements, technologies, hardware types and parts make it difficult to compare the rates. Additionally, vendors often remain silent on that topic. Nevertheless, we would like to mention some known statistics about soft error rates: SRAM chips have been reported to exhibit failure rates of 10^{-4} FIT/bit [52]. Similar failure rates are reported for registers in the central processing units (CPUs) but have a smaller effect on the overall system due to their comparatively small number [52, pp. 11-12]. The parts of the chips for combinatorial logic have lower FIT rates and, in addition, are estimated to have a masking rate of about 99% [52, pp. 12-13]. DRAM are subject to a rate of up to $7.5 \cdot 10^4$ FIT/device [49], although, most of them are corrected by hardware mechanisms resulting in 10-20 FIT/device [52, p. 13]. The effects of increasing DRAM chip sizes is widely accepted to cancel each other out [49, 52]. The probability of MCUs is disclosed to have an exponential decrease in the number of bits affected [37, pp. 31-32].

A vast number of solutions have been proposed to overcome the alarming soft error rates. First of all, soft errors need to be detected, normally involving some kind of tests similar to software testing methods for scientific applications [22, 54]. Other methods are more specific to fault tolerance, notably by backwards error analysis [6], checks for control-flow [44] as well as loops and program invariants [31, 46, 53].

Regarding recovery, a considerable part of the literature is concerned with improvements of the checkpoint/restart methods, for example the idea of providing additional processes which have redundant checkpoint information in their memory [42, 43]. Similar solutions are the replication of full processes or state machines [16, 51]. An older but nevertheless promising approach is called application-based fault tolerance (ABFT), i.e., a high-level coding for fault resilience [24]. The original variant had some problems regarding numerical stability which have since been improved [45, 50] and tested for specific problems [11]. Another option is to verify the credibility of the computation with a cheaper but less accurate method [4].



(a) In the cache error model the SoR comprises solely the caches which can be bypassed by non-temporal instructions.



(b) In the memory error model the SoR includes the caches as well as the memory.

Figure 1: The sphere of replication (SoR) in the two error models.

The community is also eager to analyze the impact of soft errors on floating point numbers in general or on specific types of problems. Floating point numbers have a particular bit-level representation [21] which turns out to have some kind of fault tolerant properties [11, 12, 34, 35]. The same phenomenon is also reported for whole problems [17, 18, 36].

Finally, a couple of error injection tools have emerged [1, 9, 33, 38] and the first problems are being analyzed under their influence [23].

2 A Recipe for Soft Error Resilience

The method introduced in this section is a recipe for soft error resilient software. Two error models are developed in Section 2.1. In Section 2.2, we detail the basic principles of the method.

2.1 Error Models

In this section we develop two error models based on the empirical evidence provided in Section 1.3. As mentioned before, any part of the hardware may be hit by a soft error. Therefore, we have to make some assumption upon which our method is to be built.

Our basic error model assumes fully reliable computations and registers while the main memory or the caches may be subject to soft errors according to the following rules: The MTBF is drawn independently from an exponential distribution with parameter λ. Based on the remark on multiple cell upsets (MCUs) in Section 1.3, the number of successive bits affected is drawn independently from a geometrical distribution with parameter p. Finally, the soft error is uniformly distributed among the bits of the caches or main memory.

Given the dissimilar SERs of SRAM and DRAM chips, two error models shall be distinguished based on whether the main memory may also be subject to soft errors as depicted in Figure 1. We call the model *memory error model* if this is the case and *cache error model* otherwise.

Cache Error Model In the cache error model we assume that the main memory is reliable – the SoR comprises only the caches as shown in Figure 1a. Furthermore we assume that the dirty bits which indicate modifications of the cache lines are not affected by soft errors. The semantic implication is that soft errors in unmodified cache lines are always transitional in the sense that they are evicted eventually and never written back to the main memory.

Data which is never part of a cache line cannot be corrupted. This includes data which is stored and loaded from memory without touching the caches by non-temporal load and store instructions [27, section 7.5]. These instructions are normally used to optimize performance by avoiding cache pollution with data that is not used again in the near future. This model also has a consequence regarding the recovery after an error has been detected: Assuming the data was written to main memory with non-temporal store instruction, corrupted cache lines can be flushed and the data reloaded from main memory [20].

Memory Error Model In the memory error model we assume that the main memory may be subject to soft errors – the SoR includes, both, the caches as well as the main memory as shown in Figure 1b. Any data which is stored in the memory hierarchy may be subject to soft errors. Notice that the assumptions of the memory error model are a strict superset of the assumptions in the cache error model in the sense that an application which is resilient to soft errors of the memory error model is also resilient to soft errors in the cache memory model but not vice versa.

Soft errors in the main memory may be permanent and require some kind of redundancy to recover. Simple reloading of the data like in the cache error model is impossible and other measures must be taken to recover.

2.2 Principles

In this section, we propose three basic principles for the design of soft error resilient applications. The first one is error masking with triple modular redundancy as described in Section 2.2.1. The second principle is error detection with checksums which is suitable for larger blocks of data as elaborated in Section 2.2.2. The third principle, discussed in Section 2.2.3, is concerned with the recovery which depends on the error model.

The methods described in this section are fully resilient to one SEU in the sense that they guarantee correctness bit by bit in contrast to other methods which guarantee a certain numerical accuracy. Multiple SEUs or MCUs may cause DUEs or SDCs in some cases as elaborated for the individual principles. We use the name *kernel* for functions which provide soft error resilience by following these principles.

2.2.1 Masking Through Triple Modular Redundancy

The first principle of our method is storage in *triple modular redundancy (TMR)*. It allows a complete masking of soft errors in both error models. TMR is ideal to protect a few values or small blocks such as configuration values and constants.

TMR is a coding technique where every datum is stored three times. The term originates from hardware redundancy [10, p. 58] but is used here for the equivalence in coding. It is a simple way to not only

detect errors but to completely mask them. A voting logic is applied after reading the three values: If all three values match, it is assumed that no soft error occurred. If only two values match, it is assumed that the remaining one was subject to an error and a value corresponding to the former two is returned. When all three values mutually disagree, a DUE is signaled. Examples of possible TRM store and load implementations are depicted in Algorithm 1.

Algorithm 1 A sketch of a triple modular redundancy (TMR) store and load implementation. We assume that, beside the `load()` and `store()` functions, no additional load and store instructions are introduced when compiling the sample source codes.

```
#define TMR 3

inline void store_tmr(double value_address[TMR], double value) {
  // store the values three times
  store(&value_address[0], value);
  store(&value_address[1], value);
  store(&value_address[2], value);
}


inline double load_tmr(double value_address[TMR]) {
  // load the three values
  double value1 = load(&value_address[0]);
  double value2 = load(&value_address[1]);
  double value3 = load(&value_address[2]);

  // voting logic
  if(value1 == value2)
    return value1;
  else if(value2 == value3)
    return value2;
  else if(value3 == value1)
    return value3;
  else
    throw DUE(); // indicate a detected unrecoverable error (DUE)
}
```

TRM may fail to mask errors if multiple SEUs or MCUs change at least two of the three values identically. All other cases are either masked or signaled by TMR.

2.2.2 Detection Through Checksums

The second principle of our method is the checksum coding technique for error detection in large blocks of data. A *checksum* of a set of values is a small redundant information produced by a checksum operation applied on those values. Checksums provide a measure to detect errors as explained in the well-known example of unreliable communication channels: Before a message is sent, a checksum is calculated and sent along with the original message. Upon reception of the message, the receiver recalculates the checksum and compares it with the one received along with the message. The receiver assumes to have read the correct message if the calculated checksum matches the received one. Otherwise, it may request the retransmission of the message. In our case, the reliable CPU core combines the role of the sender and the receiver. It transmits messages to itself through the memory hierarchy which corresponds to the

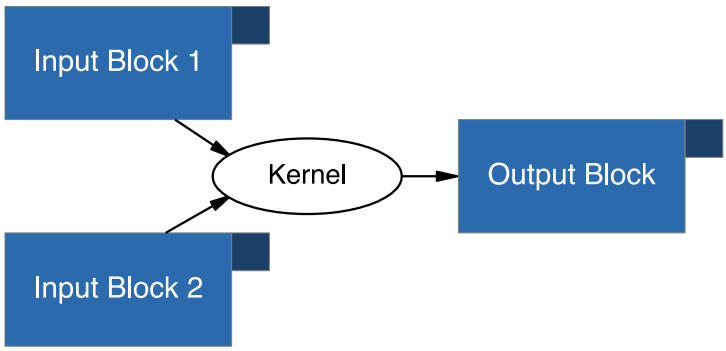


Figure 2: An example of a convolution kernel (white) with two input blocks and one output block. The values of the block (lighter blue) are protected with a checksum (darker blue).

unreliable channel. We propose to use of fast checksum operations which are only capable of detecting errors and require other measures for recovery.

We introduce the notion of *protected blocks* to simplify the description of this method. Given a checksum operation, a *protected block* is a set of *values* in the memory hierarchy with an associated *reference checksum* obtained by applying the checksum operation on those values when they were generated. A *test checksum* is calculated by performing the checksum operation again on all the values of a protected block. A protected block is said to be *correct* if the test checksum matches the reference checksum, otherwise it is said to be *erroneous*.

An important aspect of this principle is the movement of data in and out of the core. Whenever a value enters or leaves the core, the corresponding checksum must be updated. In particular, no unintended load or store instruction is permitted. In general, protected blocks must be loaded or written completely because checksums are attributed to the entire block. When writing to a protected block, the reference checksum is calculated while storing the values and also stored in memory. When reading from a protected block, the test checksum is calculated along with the loading of the values and compared with the reference checksum. If both checksums match, the protected block is assumed to be free of errors. Otherwise the protected block either has to be recovered as will be described in Section 2.2.3 or a DUE must be signaled.

Soft errors in checksums are masked by storing them outside of the core in TMR, hence, the data is to blame whenever the checksums mismatch. The checksum operation is anticipated to have low impact on the runtime by occupying unused computational resources [44]. Also, checksums belong to the class of *separable codings*, i.e., the original data and the redundant supplement are separated, thus, there is no need for encoding and decoding in the kernels. Registers in the core are occupied for all the input, output and intermediate values as well as the checksums of every input and output block. Thus, the number of registers available in a core is a limiting factor for the usability of protected blocks.

Though seemingly rigid, a huge variety of kernels can be achieved depending on the number of input and output blocks and the operation performed on them. The following counterintuitive applications of protected blocks are important later in the text:

- Protected blocks do not have to be stored contiguously in memory.
- Multiple protected blocks may overlap. This means that some values could be covered by multiple reference checksums.
- If only one part of a protected block is required for a calculation, the remaining values can be loaded

superfluously to complete the block load.

- For a convolution kernel, the same block might be read multiple times with an offset within the same kernel invocation by treating them as different input blocks each covered with its own test checksum.

- Another degree of freedom depends on the concrete checksum operation. Checksums such as cyclic redundancy check (CRC) [10, p. 121] must be applied to the values in the same order. Other checksum such as the bitwise `XOR` operation may be applied to the values in any permutation due to their associative and commutative nature.

The probability that multiple SUEs or MCUs remain undetected equals the probability of them causing a checksum collision which depends on the checksum operation. A single SEU is always detected with any reasonable checksum operation.

Examples of two kernels are provided in Algorithm 2. The first kernel (on line 1) is a generation kernel which fills a buffer with data. At the beginning, a reference checksum is initialized. Then, the following two steps are performed for every element of the buffer: First, a value is generated based on the index of the element. Second, the value is stored in the buffer and the reference checksum is updated. In the end, the reference checksum is stored. The second kernel (on line 21) is a reduction kernel which reduces the elements of an input buffer to a single value. At the beginning, the test checksum for the input block and the reduction value are initialized. Then, the following two steps are performed for every input value: First, the value is loaded from the memory hierarchy and the test checksum is updated. Second, the reduction operation is performed. In the end, the test checksum is compared with the reference checksum. The final result of the reduction is store to the memory hierarchy. The kernel indicates possible checksum mismatches in its return value.

2.2.3 Recovery Through Reconstruction and Reloading

A fault tolerant application must not only detect errors but also recover from them. Protected blocks are only a measure for detecting errors. The goal of the recovery process is to remove the soft errors after they have been detected. The methods for the recovery of blocks introduced in this section are reloading and reconstruction for the cache and memory error model, respectively.

Reloading is the method of recovery in the cache error model. A detected soft error is guaranteed to be of transient nature in the cache error model. Therefore, it is possible to flush the cache and reload the data from main memory in order to recover. This method has two necessary conditions: First, the erroneous input block must not be overwritten with an output block before the kernel detects the soft error. Second, the values reloaded from the main memory must be free of soft errors, which may be enforced with non-temporal store instructions as noted in Section 2.1.

Reconstruction is the method of recovery in the memory error model. A detected soft error may be of permanent nature in the memory error model, hence, reloading the erroneous input block from main memory may provoke the same error again. Nevertheless, the erroneous input block may be reconstructed, as illustrated in Figure 3, because it is itself the output block of another kernel denoted as the *recovery kernel*. The method has two necessary conditions in order to succeed: First, the input blocks of the recovery kernel – which we call *recovery blocks* – must still be available. Second, the recovery blocks must be free of soft errors. Both conditions are trivially met if the recovery kernel has no input blocks.

Algorithm 2 Examples of a generation and a reduction kernel. The functions `store_tmr()` and `load_tmr()` refer to Algorithm 1. We assume that, beside the `load()` and `store()` functions, no additional load and store instructions are introduced when compiling the sample source codes.

```
void generation_kernel(double* values_address, size_t values_length,
                       double checksum_address[TMR]) {
  // initialize the reference checksum for the output block
  double reference_checksum = checksum_init();

  // for every element in the output block
  for(size_t i = 0; i < values_length; ++i) {
    // perform a generating operation
    double value = generation_operation(i);

    // store output value and update its reference checksum
    store(values_address + i, value);
    reference_checksum = checksum_operation(reference_checksum, value);
  }

  // save the output reference checksum
  store_tmr(checksum_address, reference_checksum);
}


bool reduction_kernel(double* values_address, size_t values_length,
                      double checksum_address[TMR], double* reduction_address) {
  // initialize the test checksum for the input block
  double test_checksum = checksum_init();

  // set the identity element for the reduction operation
  double reduction_value = reduction_identity();

  // for every element in the input
  for(size_t i = 0; i < values_length; ++i) {
    // read the next value of the input block and update the test checksum
    double value = load(values_address + i);
    test_checksum = checksum_operation(test_checksum, value);

    // perform the reduction operation
    reduction_value = reduction_operation(reduction_value, value);
  }

  // compare the test checksum with the reference checksum for the input
  bool correct = (test_checksum == load_tmr(checksum_address));

  // store the reduction value with TMR
  store_tmr(reduction_address, reduction_value);

  return correct;
}
```

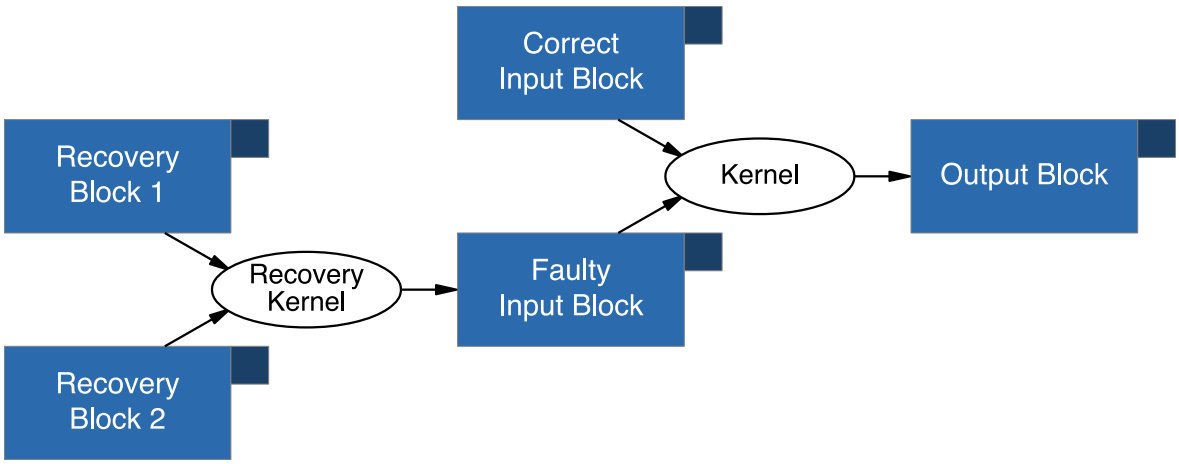


Figure 3: The process of reconstruction. If a kernel detects a erroneous input block, it may be reconstructed by repeating the recovery kernel if the recovery blocks 1 and 2 are still available.

Both methods have a memory overhead independent of the occurrence of errors. The overhead arises if the data could be overwritten immediately after reading but must be kept for recovery. In the case of reloading, solely the input blocks of the kernel which detected the error must be kept. In the case of reconstruction, the recovery blocks must be kept in addition. Thus, the absolute memory overhead depends on the size of the input blocks.

Both methods also have a runtime overhead in the case of an error: On one hand, it arises from the fact that either the cache has to be flushed or the recovery kernel has the be re-executed. On the other hand, the kernel which detected the soft errors has to be re-executed independent on the recovery method. Again, both factors depend on the block sizes. There is no need for recovery and, therefore, no runtime overhead in the absence of errors.

The size of protected blocks is an important design choice because it influences the memory and the runtime overhead. Sufficiently big blocks could be divided into multiple smaller blocks each of them protected with its own reference checksum. At a first glance, reducing the block size not only reduces the memory overhand but also the runtime overhead. But kernels applied on tiny blocks come with the risk of harming the performance by many separate kernel invocations. Hence a balance between the two competing aspects of block sizes is inevitable.

In Algorithm 3, an example of a driver function with recovery capabilities for both error models is shown. A buffer is filled with data by a generation kernel and reduced to a single value by a reduction. In the cache error model, the generation kernel is invoked exactly once while the reduction kernels is repeated upon errors with intermediate flushing of caches for reloading. In the memory error model where reconstruction is required, the generation and the reduction kernels are repeated as long as the latter one reports errors.

3 A Test Implementation

A wide range of applications can be designed by adopting kernels implementing the aforementioned principles. They include time stepping schemes on regular grids which are not only the origin of our research but also the leading example in this thesis. But the method could also be applied to other

Algorithm 3 An example of a driver function which supports recovery in both error models. The `generation_kernel()` and `reduction_kernel()` refer to Algorithm 2. A possible implementation of the `cache_flush()` function is covered in Section 3.1.5. We assume that, beside the `load()` and `store()` functions, no additional load and store instructions are introduced when compiling the sample source codes.

```
1  void driver() {
2    double values[SIZE]; // values of the block
3    double checksum[TMR]; // checksum of the block
4    double reduction[TMR]; // final result
5
6  #if CACHE_ERROR_MODEL
7    // invoke generation kernel
8    generation_kernel(values, SIZE, checksum);
9
10   // while the reduction kernel returns errors ...
11   while(!reduction_kernel(values, SIZE, checksum, reduction)) {
12     // flush the caches
13     caches_flush(values, SIZE);
14   }
15 #elif MEMORY_ERROR_MODEL
16   do {
17     // invoke the generation kernel
18     generation_kernel(values, SIZE, checksum);
19   } while(!reduction_kernel(values, SIZE, checksum, reduction));
20   // ... while the reduction kernel return errors
21 #endif
22
23   output(reduction);
24 }
```

applications that can be formulated by protected blocks.

We design and implement two applications to empirically test our method for soft error resilient time stepping schemes. The first is a solver which tries to find the solution to a test problem by implementing the method covered in Section 2.2. The second is the injector which injects soft errors into the solver for testing purposes.

3.1 Solver

3.1.1 Test Problem

The solver tries to approximate the solution of a two-dimensional parabolic heat equation with initial and Dirichlet boundary conditions using finite differences. Find $U : \Omega \times [0, t_{\text{end}}] \to \mathbb{R}$ subject to

$$\begin{aligned} \frac{\partial U(\mathbf{x}, t)}{\partial t} + \rho \Delta_{\mathbf{x}} U(\mathbf{x}, t) = 0, \qquad & \mathbf{x} \in \Omega,\ t \in [0, t_{\text{end}}] \subset \mathbb{R} \\ U(\mathbf{x}, 0) = f(\mathbf{x}, 0), \qquad & \mathbf{x} \in \Omega \\ U(\mathbf{x}, t) = f(\mathbf{x}, t), \qquad & \mathbf{x} \in \partial\Omega,\ t \in (0, t_{\text{end}}] \subset \mathbb{R} \end{aligned}$$

on the spatial domain $\Omega = [0, 1]^2 \subset \mathbb{R}^2$ where $\Delta_{\mathbf{x}}$ is the Laplace operator w.r.t. spacial coordinates, $\rho > 0$ is a constant and $f(\mathbf{x}, t) := x_1^2 + x_2^2$.

The test problem is discretized in space on a regular $N \times N$ grid using finite differences and in time using explicit Euler integration method with T equidistant time steps. $U_{i,j}^k$ denotes the approximated value of U at position $\mathbf{x}_{i,j} := (i/(N-1), j/(N-1))$ for $0 \leq i, j \leq N-1$ and time $t_k := k \cdot t_{\text{end}}/(T-1)$ for $0 \leq k \leq T-1$. The initial condition is imposed by

$$U_{i,j}^0 := f\left(\mathbf{x}_{i,j}, 0\right) \tag{1}$$

for all $1 \leq i, j \leq N-1$. For each time step $0 < t \leq T-1$ the following stencil is applied to each of the interior grid points $\{(i, j) \mid 1 < i, j < N-1\}$

$$U_{i,j}^k := U_{i,j}^{k-1} + c\left(U_{i+1,j}^{k-1} + U_{i-1,j}^{k-1} + U_{i,j+1}^{k-1} + U_{i,j-1}^{k-1} - 4 \cdot U_{i,j}^{k-1}\right) \tag{2}$$

with $c = \rho/(1/N)^2$. The Dirichlet boundary conditions are carried along for the boundary positions $\{(i, j) \mid i = 0 \vee i = N-1 \vee j = 0 \vee j = N-1\}$ with

$$U_{i,j}^k := f(\mathbf{x}_{i,j}, t_k) \tag{3}$$

A way to think about this stencil is with double buffering, i.e., introducing two domains which are swapped after each time step. One domain – denoted as the *read domain* – contains the approximate solution at time step $k-1$ and the other domain – the *write domain* – is prepared to contain the result at time step k. Now, the center of the stencil is placed on every interior point of the read domain and produces the corresponding value in the write domain as illustrated in Figure 4. Then, the two domains swap roles for the next time step. The process is repeated until the write domain contains the solution at time step $T-1$. Note that these two domains do not necessarily have to represent memory regions, even though this would be a simple implementation.

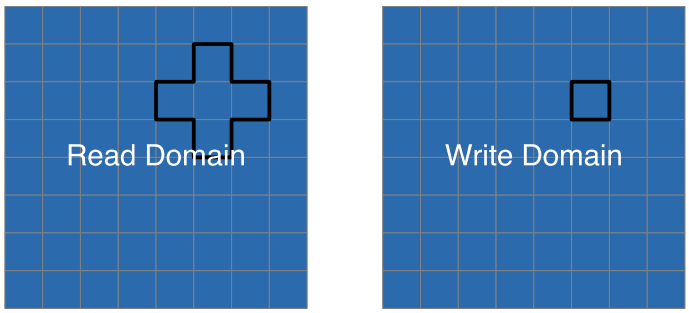


Figure 4: Visualization of the stencil applied to the read domain (left side) resulting in the corresponding value in the write domain (right side).

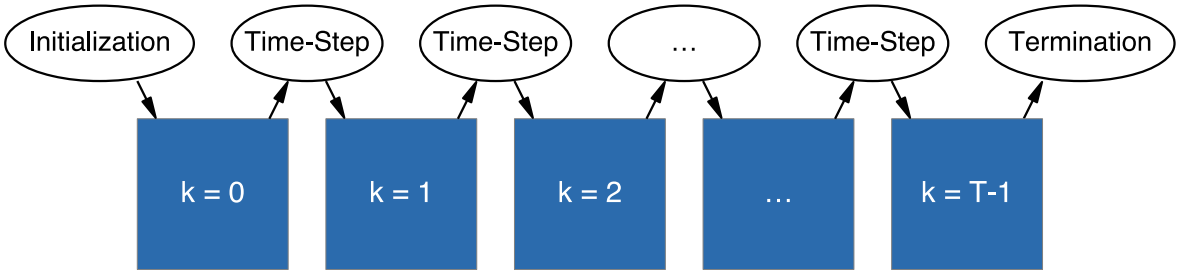


Figure 5: The simulation phase of the solver as a sequence of applications of the three kernels (white) with their input and output blocks (blue). The initialization kernel creates the initial condition at time step 0. Repeated application of the time-step kernel create the result of time step k from the result of time step $k-1$. The termination kernel outputs the final result.

The model problem is designed to have useful properties for testing: First, it is simple to implement in a reference solver for verification. Second, two neighboring points in one time step and the same point at different time steps almost always differ in value. Conversely, if the solution of a run is identical with the reference solution it is almost certainly correct. Third, the solution can be investigated visually because of its two-dimensional nature. Finally, due to a single conserved quantitiy, the number of registers occupied in the implementation is low.

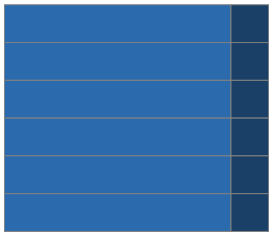
The heat equation is simply an example and other problems with more conserved quantities and higher dimensions could be implemented analogously. We try to anticipate problems arising with other problems such as register occupation.

3.1.2 Kernels

The simulation phase consists of three kernels: An initialization kernel, a time-step kernel and a termination kernel as shown in Figure 5. All kernels solely operate with the principles described in Section 2.2. Thus, the simulation phase is resilient with respect to the error models defined in Section 2.1. Soft errors injected anywhere in the simulation phase have a high probability of being detected.

The initialization kernel is similar to the generation kernel in Algorithm 2 and produces the initial values (cf. (1)) for the simulation. This initialization kernel can be repeated whenever needed because it has no input block as stated in Section 2.2.3.

(a) The domain approach where the whole simulation domain (lighter blue) is treated as a single protected block with a single checksum (darker blue).



(b) The line approach where every horizontal line (lighter blue) is treated as its own protected block each with an individual checksum (darker blue).

Figure 6: The two approaches of defining protected block in the model problem.

The time-step kernel implements the stencil (2) of the test problem. This kernel has one output block for writing the solution of the current time step. Further, it has three input blocks, i.e., one for every line of the stencil in the domain of the preceding time step. The stencil is moved line-by-line across the domain. The three values of the stencil's middle line do not have to be reloaded every time the stencil moves to the right. Only the rightmost is loaded from the memory and reused as the middle and left value when the stencil moves one and two positions to the right, respectively. Some values have to be loaded superfluously because not every line of the stencil reaches every point on the reading domain. The kernel also conserves the boundary values (cf. (3)) by copying them from the read domain to the write domain since they are time-independent.

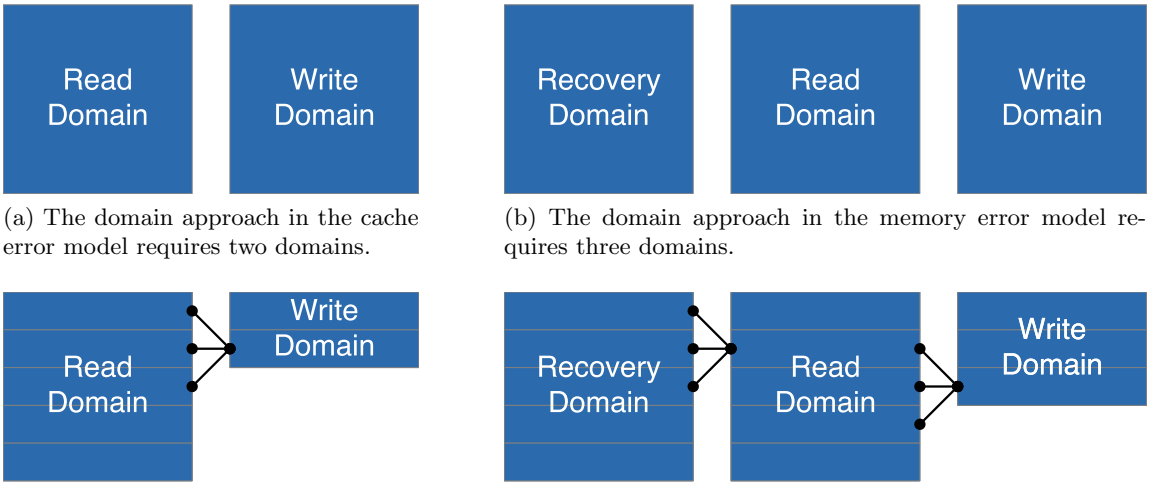
The termination kernel is similar to the reduction kernel in Algorithm 3 but instead of writing the result to permanent storage it calculates the CRC checksum of the result. From the methods perspective calculating the CRC checksum of the result is similar to save the result to permanent storage but has the advantage of an execution time that is independent of storage devices and claims considerable less disk space when many automated experiments are conducted.

3.1.3 Protected Blocks

In the model problem, there are various sets of values which could be treated as protected blocks by means of our method. For the empirical tests we investigated the approaches depicted in Figure 6: The first one – the *domain approach* – treats whole discretized domains of the simulation as a single protected block. The second one – the *line approach* – treats every horizontal line of the domains as an individual protected block. Both approaches may be applied to both error models, resulting in a total of four major combinations. An illustration of their memory consumption is provided in Figure 7.

Domain Approach In the domain approach, a whole domain of the simulation is a protected block as shown in Figure 6a. This approach is optimal in the sense that as many computations as possible are performed by the kernels without unnecessarily repeating kernel invocations.

However, this approach has a severe memory overhead. In the cache error model, two disjoint protected blocks, each of the size of the discretized domains must be allocated as depicted in Figure 7a. In the memory error model, even three blocks of the same size are required in order to recover the input blocks as illustrated in Figure 7b.



(a) The domain approach in the cache error model requires two domains.

(b) The domain approach in the memory error model requires three domains.

(c) The line approach in the cache error model requires one domain plus two lines.

(d) The line approach in the memory error model requires two domains plus three lines.

Figure 7: Memory consumption of the four combinations arising from the domain and line approaches in the cache and memory error models. For the line approach, the dependent lines of the stencil (indicated black) determine the number of additional lines. The checksums are omitted in this graphic.

This approach also has a runtime overhead in case of an error. In the cache error model, the entire input block must be flushed and the kernel re-evaluated. In the memory error model, the input block has to be reconstructed first by executing the recovery kernel and, then, the kernel itself is re-evaluated.

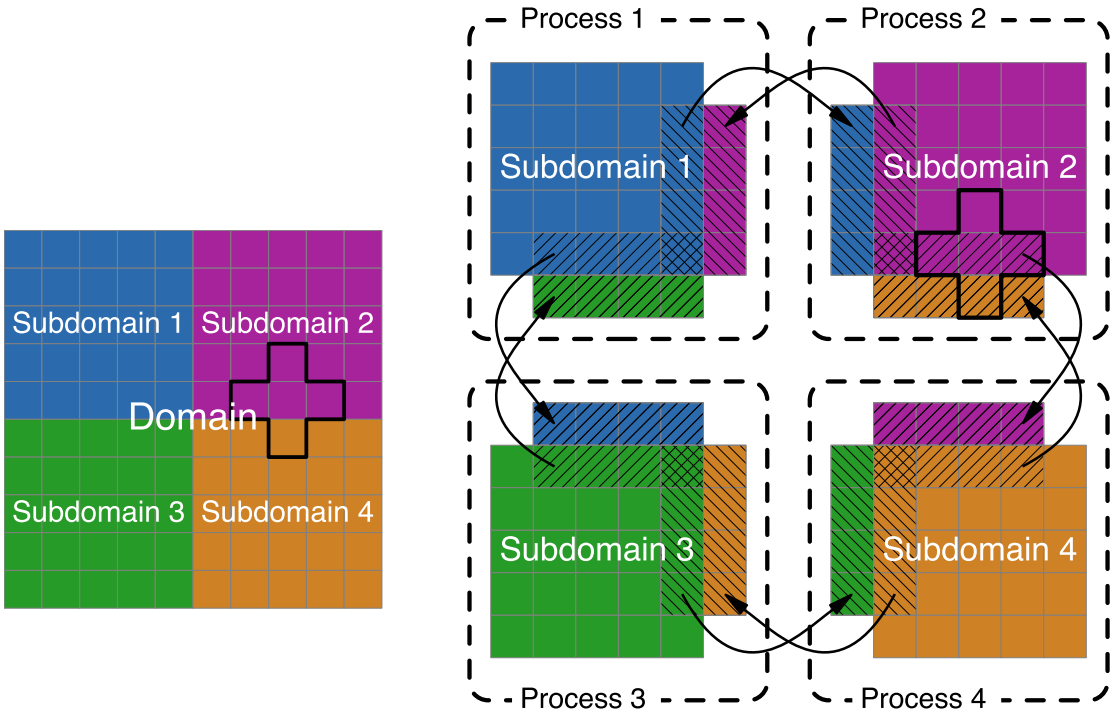
Line Approach In the line approach, every horizontal line of the domain is treated as a protected block as shown in Figure 6b. The three kernels of the simulation phase must be slightly modified to work with these smaller blocks. The smaller block sizes may increase the runtime overhead by repeated kernel invocations. However, there is a smaller memory overhead for recovery and a smaller runtime overhead in the case of an error compared to the domain approach as described next.

In case of an error, individual lines can be recovered which decreases the memory overhead by approximately one discretized domain compared to the domain approach. In the cache error model, two additional lines must be kept for recovery as depicted in Figure 7c. In the memory error model, the erroneous line must be reconstructed from the three corresponding recovery blocks. In the worst case, the erroneous line is detected in the uppermost input block of the read domain and, thus, requires three additional lines to recover as shown in Figure 7d.

3.1.4 Distributed Computing

The solver supports distributed memory environments through the message passing interface[1] (MPI). A single-core solver would be much simpler, although uninteresting because only large simulations are subject to soft errors. Special care must be taken to increase the dependability when implementing this

[1] http://mpi-forum.org/



(a) The entire simulation domain partitioned into four subdomains (colors).

(b) The assignment of these subdomains to different processes and their extensions with the ghost layers consisting of the boundary values (hatched) of the neighboring subdomains.

Figure 8: Dividing the simulation domain into subdomains and the consequence of exchanging boundary values. The example stencil placed in subdomain 2 clearly depends on a boundary value of subdomain 4.

extension as mentioned in Section 1.2.2.

The simulation is distributed onto multiple cores by dividing the simulation domain into rectangular subdomains and assigning each subdomain to an individual MPI process as shown in Figure 8. The subdomains are kept as square as possible for the given number of processes with the help of the MPI function `MPI_Dims_create()`. Values on the boundary must be exchanged with the processes responsible for neighboring subdomains because the stencil on the boundary of one subdomain depends on boundary values of the neighboring subdomains. For this purpose, the boundary values are sent to the neighbor processes and appended to their subdomains as so-called *ghost layers* as depicted in Figure 8b.

The complexity arises from the fact that a soft error may hit a boundary value before, during or after the exchange. A fault-tolerant exchange kernel is introduced which exchanges the boundary values between time steps as shown in Figure 9. The exchange is done by nonblocking MPI communication on a cartesian communicator created with `MPI_Cart_create()`. The precise implementation of the kernel depends on the chosen approach.

Domain Approach In the domain approach, the boundary values are exchanged as protected blocks. This is achieved by augmenting the time-step kernel with input and output blocks for each neighboring

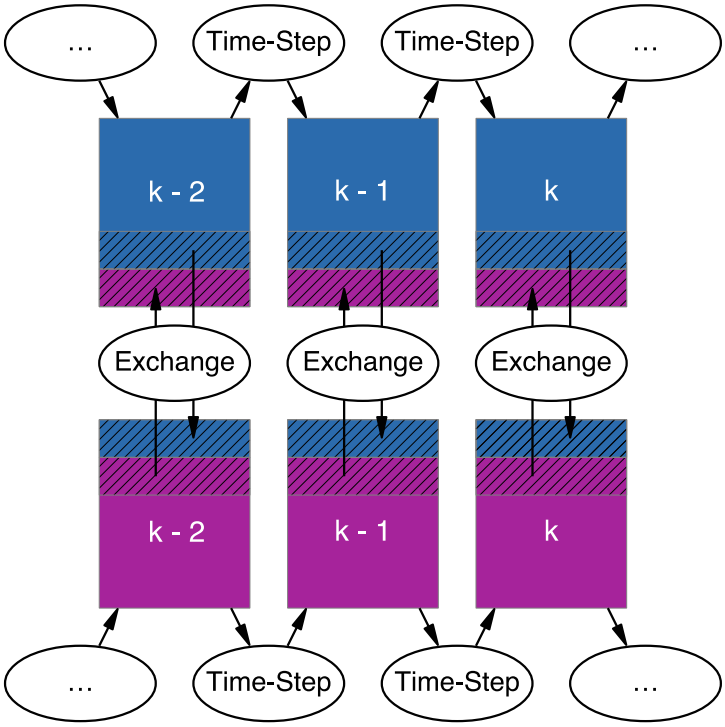


Figure 9: Application of the exchange kernel (white ellipse labelled "Exchange") to exchange boundary values (hatched) between the applications of the time-step kernel (white ellipse labelled "Time-Step") in the example of two processes with their corresponding subdomains (blue and magenta, respectively).

subdomain. The implementation is optimized with regards to register occupation and runtime overhead, but only the basic procedure is covered here for brevity.

In case of the output blocks, each additional block covers the boundary values of one neighboring subdomain, and, therefore, overlaps with the block covering the whole subdomain of the process. Whenever a boundary value is written, the reference checksums of the corresponding neighboring blocks are updated in addition to that covering the entire subdomain. After the time step, the values and the reference checksums of the blocks are sent to their corresponding neighbor processes.

In case of the input blocks, the additional blocks cover the ghost layers and are disjoint from the block covering the process's subdomain. Hence, whenever a value is loaded it is either covered by the test checksum of the corresponding ghost layer block or by the one covering the entire subdomain. After the time step, the test checksums of all the input blocks are compared with the reference checksums and an error is detected if at least one mismatch occurs.

No soft error detection is performed between writing the boundary values on one process and their incorporation as ghost layers in the next time step on another process. The principle is sometimes referred to as end-to-end argument [47]. Instead, a process replies with an error message whenever an error is detected after the time step, triggering the recalculation of the last time step on its neighbors and the retransmission of their ghost layers.

Line Approach The solution of transmitting protected block is not applicable here. The reason is that local values are successively overwritten as the stencil traverses the domain. If an error message would be

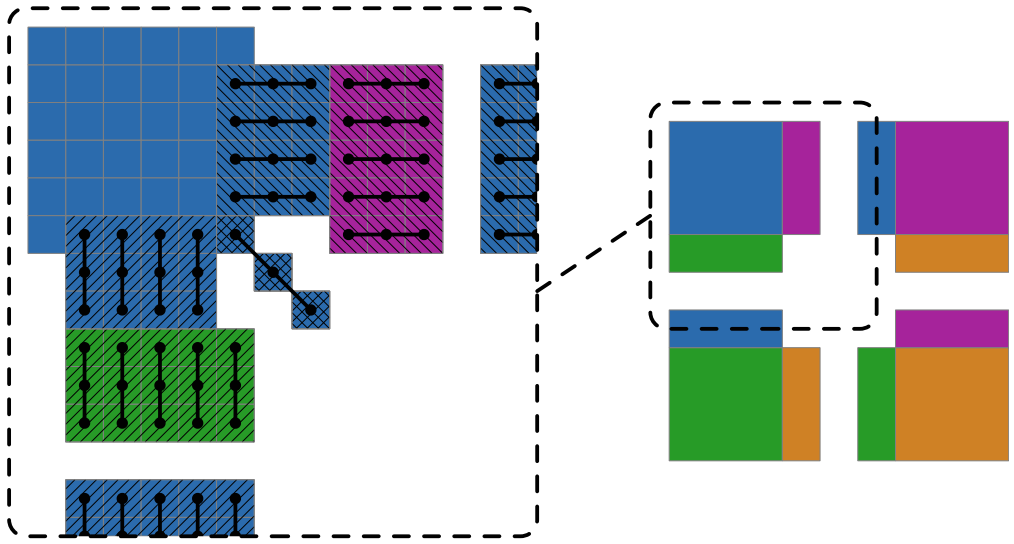


Figure 10: A visualization of the layout for boundary value exchange in the line approach. The boundary values (hatched) of the subdomains (colors) are always stored and transmitted with triple modular redundancy (TMR; connected boxes).

received afterwards, it would be impossible to recover from the error. Instead, the values are exchanged in TMR.

Whenever a kernel accesses the boundary value of the subdomain, it is stored or loaded in TMR. Similarly, the boundary values are also exchanged in TMR between time steps. The memory layout of the boundary values and ghost layers is partitioned into different regions as illustrated in Figure 10. After a ghost layer is received in TMR, the problem of recovery becomes a local problem and no error messages are needed.

The solution has a memory, a network and a runtime overhead independently of the occurrence of soft errors. The memory and network overhead comes from the fact that the boundary values and ghost layers are stored and sent in TMR which triples their memory consumption and network volume. The runtime overhead arises because storing in TMR requires multiple store instructions and loading requires multiple loading instructions with a voting logic. Other solutions with other properties would be possible as well.

3.1.5 Implementation Details

The kernels are written in the C++11 programming language, even though it is impossible to guarantee correctness a priori. The problem is that the C++11 standard does not comprehend entities such as cores, registers, caches, load and store instructions which are the basis for our method. Instead "conforming [compiler] implementations are required to emulate (only) the observable behavior" [28]. But the observable behavior under the existence of soft errors changes when load and store instructions are introduced by the compiler, e.g., for pushing variables onto the stack because of register shortage.

The kernels are instead implemented with Intel intrinsics [26], i.e., compiler directives which translates one-to-one to single instruction multiple data (SIMD) assembly instructions. Intrinsics are also subject to rearrangements in general but a kernel cleanly programmed with intrinsics turns out to conserve the required properties as opposed to normal C++ source code. The correctness can be shown a posteriori

by disassembling the kernels and showing that neither harmful rearrangements have occurred nor load or store instructions were introduced.

Despite the fact that intrinsics mainly exist for SIMD vectorization, the kernels are not vectorized. Vectorizing kernels is an elementary method in HPC but would go beyond the scope of this thesis. Here, intrinsics are merely a possibility to advise the compiler to create a correct program in the sense of our method. A serious flaw of this design choice, though, is that the non-temporal store – as required in the cache error model – cannot be performed because it only exists for vectorized code. This restriction does not affect the correctness of the solver as long as the soft errors are injected artificially (cf. Section 3.2.3) but may influence its performance (cf. Section 4.2).

The kernels use the intrinsic instruction `_mm_xor_pd()` (bitwise `XOR` on packed double precision floating point numbers) as their checksum operation for protected blocks. This leads to a 64-bit interleaved parity code capable of detecting any SEU, MCU up to 63 bit wide and other soft errors as long as not all bit positions are hit an even number of times. The TMR load implementation from Algorithm 1 is optimized to provide its functionality with a single branching operation by using the bit masks returned by the SIMD comparison instructions. The `cache_flush()` operation used in Algorithm 3 can be implemented with a simple loop over the intrinsic `_mm_clflush()` instruction.

The solver may throw an exception indicating detected unrecoverable errors (DUEs) in some situations. Such a situation occurs if operations fail repeatedly: Some operations could be repeated ad infinitum, e.g, reloading, reconstruction and retransmission. However, sanity checks are included in the kernels in the form of retry limits. A DUE is signaled when the retry limit for an operation is exceeded. This could, however, lead to a false alarm when the operation is indeed repeatedly hit by soft errors. Another possibility for DUEs are soft errors in recovery blocks.

3.1.6 Optimization

The goal of this thesis was not to implement a fast solver for finite differences, but basic code optimizations are nevertheless required to obtain meaningful measurements. To keep the text concise, results of the various optimization steps are only briefly covered here.

First of all, optimization is mostly done for the hotspots of the code. The initialization and termination kernels are only optimized roughly since they are executed at most a few times. Conversely, parts of the time step kernel are highly optimized as described below.

A major performance optimization is done by eliminating branch instructions in the time-step kernel. It is not surprising that many branching instructions are located in this kernel considering the different treatment of Dirichlet or ghost borders and the special handling of border values in the line based approaches. However, these special cases all lay on the first or last few iterations of the two loops for traversing the (sub-)domain. In addition, the outcome of the branching instructions is defined as soon as the neighbors of the process are determined. Therefore, sixteen nearly branchless variants of the kernels are compiled, i.e., one for every combination of either Dirichlet boundaries or ghost layers on each of the four sides of the (sub-)domain. In the line approach, each variant has five sub-variants depending on the line position in the (sub-)domain.

Minor performance optimizations performed in the solver include loop reordering, re-associated computation in the stencil, inlining of code and compiler optimizations. Optimization related to the memory

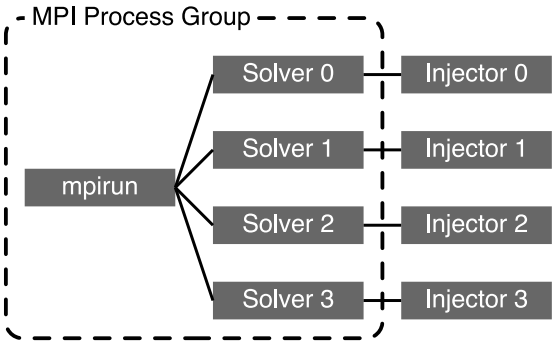


Figure 11: The process hierarchy in the example of four solvers: Processes related to MPI form a group comprising `mpirun` and the solvers which are direct children. The injectors are supplementary children of the corresponding solvers, i.e., they trace their own parents for soft error injection.

hierarchy such as blocking was omitted since the time-step kernel accesses the memory three-way-linearly and the processor mentioned in Section 4.1 have elaborate prefetching mechanisms [25].

3.2 Injector

The injector is a tool for simulating soft errors in the solver. It rests upon the software-based injection method as it is a good compromise between development costs and flexibility. It supports two operation modes for the two error models.

3.2.1 Process Setup

The `ptrace()` system call is normally used by debuggers and system call tracers (e.g., GDB[2] or `strace`[3]) but is applied here for error injection. It allows a program – called the *tracer* – to monitor the behavior and possibly alter the state of another process – called the *tracee*. The tracer is informed upon events of the tracee such as the delivery of signals and system call invocations. Additionally, the tracer may spontaneously decide to stop the tracee. Whenever the tracee is stopped, the tracer may inspect and alter the state of the tracee including the memory and the registers. The tracer is able to resume the normal execution of the tracee or instruct the tracee to take a single step in the code. The disadvantage of this injection method is that a process may only be debugged by at most one process which lead to difficult debugging scenarios during the development of the solver.

Every solver process is paired with an injector process. At the beginning, `mpirun` start the injectors which themselves fork and substitutes their parent process with an instance of the solver. Though counterintuitive and unnatural from a debugging point of view, it is possible for a child process to debug its parent under Linux. This setup is more natural from the MPI perspective: `mpirun` has the solvers – which are the MPI processes – as their direct children with their injectors only appended as supplementary processes as depicted in Figure 11. Otherwise, `mpirun` is often unable to terminate the processes properly when `MPI_Abort()` is invoked. Most of the time, only one process of a pair is running: The injector is normally suspended and only awakes for soft error injection during which the solver is stopped.

[2] http://www.gnu.org/software/gdb/
[3] http://sourceforge.net/projects/strace/

3.2.2 Restrictions

The injection of soft errors must be restricted in space and time in order to reliably provide functionality such as runtime measurements and determining the correctness of the simulation within the solver. These features require minimal support in the solver which is also described here.

For the restriction in time, the solver sends itself two `SIGUSR1` signals which cause the injector to be informed. The first signal is sent when the simulation begins to indicate the start of the soft error injections and the second after the simulation ends to stop them.

In-between the start and the stop signal, soft errors are injected into the memory regions according to the restriction in space. For this purpose, the solver maintains a well-defined structure of memory regions and their content. The injector is able to peek at this structure from the solvers memory space with `ptrace(PTRACE_PEEKDATA, ...)`. As a result, soft errors injections in specific memory regions are possible for testing, e.g., boundary values or checksums.

We claim that soft error resilient kernels contribute significantly to the overall resilience of the system. MLMC simulations spend most of their execution time in scientific computation and use a considerable part of their memory for scientific data. With newer hardware including x86-64 platforms, function arguments and iteration variables are often kept in registers, thus, avoiding soft errors according to our models. Hence, we claim that soft errors would indeed hit scientific data most of the time and soft error resilient kernels are a major step towards resilient simulations. This conjecture is not verified because of many open questions related to memory regions and their write protection, instruction cache behavior, system calls and kernel space, erratic behavior on the test cluster and others.

3.2.3 Soft Error Injection

Between the start and the stop signal, the injector alternates between waiting for a timeout and injecting a soft error. The timeout is drawn independently from an exponential distribution according to the basic error model. After the timeout occurs, the injector wakes up and stops the solver by sending a `SIGTRAP` signal. Then, the solver undergoes a soft error injection before it is released again with `ptrace(PTRACE_CONT, ...)`. This results in soft errors appearing out of thin air from the solver's perspective. The precise method for the soft error injection depends on the error model as follows.

Memory Error Model In the memory error model, the soft error injection is simple: A word is retrieved with `ptrace(PTRACE_PEEKDATA, ...)`, altered and stored back with `ptrace(PTRACE_POKEDATA, ...)`. The caches are flushed before the solver resumes, therefore, the errors initially reside only in the main memory after the injection. The location and width of the soft error follow exactly the model assumptions, except that MCUs are only supported within words. Benign errors may occur when the words poked is never read.

Cache Error Model In the cache error model, the situation is more difficult as the cache is completely transparent to the software: The hardware provides neither a mechanism for retrieving information about the current state of the cache nor a way to alter the cache without setting the dirty-bit and ultimately writing the changes back to main memory. To the best of our knowledge, the only way to simulate soft

errors in the cache with software based error injection is to alter the destination register of load instructions [55]: After the solver is stopped, the injector steps through the instructions of the solver one by one with `ptrace(PTRACE_SINGLESTEP, ...)` until a load instruction is detected. This load instruction is then executed with another call to `ptrace(PTRACE_SINGLESTEP, ...)`, the value of its destination register retrieved with `ptrace(PTRACE_GETFPREGS, ...)`, altered and written back to the same register with `ptrace(PTRACE_SETFPREGS, ...)`. At this point, the solver is in the same state as if the value loaded had been subject to a soft error in the cache. Benign errors are impossible in this method.

Basic disassembly capabilities are needed to check whether the instruction is a load instruction and if so to determine its source address and destination register. These are achieved by the help of the libudis86[4] library version 1.7.2. After every step of the solver, the program counter (`RIP` register) which points to the instruction to be executed next is retrieved with `ptrace(PTRACE_GETREGS, ...)` and dereferenced with `ptrace(PTRACE_GETTEXT, ...)`. The machine readable instruction is disassembled by `ud_disassemble()` and its operands are tested for exactly one memory source and exactly one register destination operand. The memory source operand must further lay within the memory regions specified by the space restrictions. MCUs are subject to the same restrictions as for the memory error model.

This method, however, also has several issues: First, it simulates soft errors in caches in general but is not capable of simulating soft errors in specific cache levels. Second, it neither provides a uniform distribution of soft errors over the cache lines nor rises the probability of a soft error with the duration in which the data resides in the cache as the model would state. Third, the soft error disappears when reloading the same address, even though, the cache would sustain the soft error until the corresponding cache line is evicted. Fourth, the likelihood of a load instruction being hit by a soft error highly depends on its previous instructions: It is very low if the instruction is preceded by another load instruction because the injector would have to stop the solver at this very instruction in contrast to a load instructions preceded by many non-load instructions. This could however be improved [55]. Finally, single-stepping through the solver from the halting point to the first load instruction is an order of magnitude slower than the normal execution. Despite these drawbacks, the method gives insights into the behavior of the solver under the cache error model.

4 Results

In this section, we present experimental results of the four combinations arising from the two error models and the two approaches regarding the definition of the protected blocks. We start, in Section 4.1, describing the precise environment in which the tests are conducted. Afterwards, we present the actual results without and with error injection in Section 4.2 and 4.3, respectively.

4.1 Test Environment

The following tests were all performed on the EULER cluster[5] consisting of Hewlett-Packard ProLiant BL460c Gen8 Server Blades each with two 12-core Xeon E5 2680v3[6] processors and 64GB of main memory [14, 15]. The processors run at a base frequency of 2.5GHz and have Turbo Boost enabled. The instruction throughput for double precision floating point numbers is 1 addition and 2 multiplication per cycle [25].

[4] http://udis86.sourceforge.net/
[5] http://www.ethz.ch/id/services/list/comp_zentral/cluster/index_EN
[6] http://ark.intel.com/products/81908/

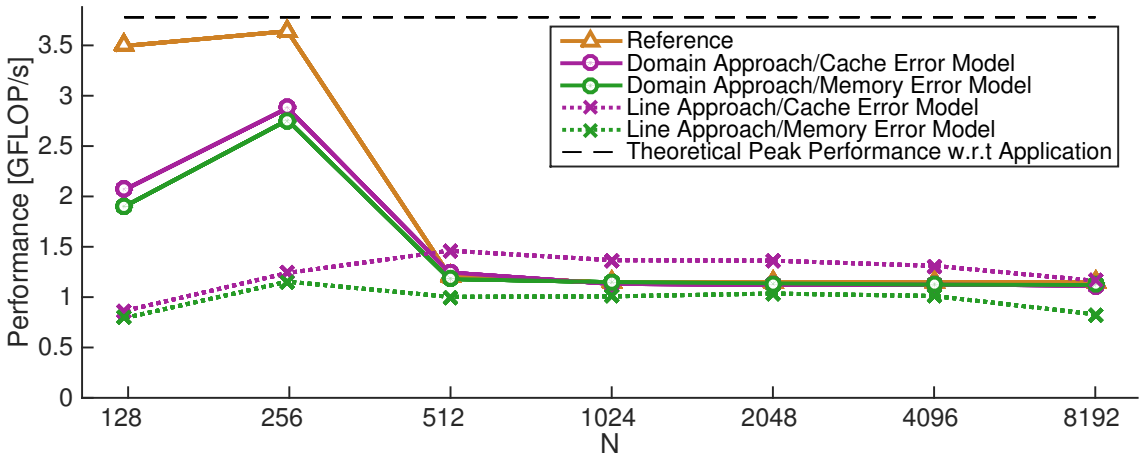


Figure 12: Single-core performance of the four combinations together with a reference implementation in terms of double precision floating point operations per second versus the problem size N.

The Cent OS[7] operating system version 6.6 is running on the cluster with the Platform LSF[8] batch system version 9.1.2.0. It has support for MPI through Open MPI[9] version 1.6.5. The C++11 source code is compiled with the GCC[10] compiler version 4.8.2.

Special care was taken to avoid influences of other workloads during the measurements. The number of solver processes was rounded up to a multiple of 24 to occupy a full node of the cluster. Dummy tasks were assigned to the spare cores, preventing influences of Turbo Boost as well as effects in the (possibly shared) caches due to other applications. Furthermore, every solver-injector pair is assigned to a single core with Linux CPU affinity to keep the profile of the injector low.

4.2 Results without Error Injection

In this section we quantitatively cover the properties of the four combinations in the absence of errors. The intension is to demonstrate the legitimacy of the result under the existence of errors. For that purpose, we show the single-core performance together with the strong and weak scaling behavior.

In Figure 12, we show the single-core performance for different problem sizes $128^2 \leq N^2 \leq 8192^2$. In addition to the four combinations, the performance of a fault-intolerant reference implementation is depicted. For small problem sizes, the performance of the reference solution is just below the theoretical peak performance of 3.78 GFLOP/s, the domain approach reaches nearly 3 GFLOP/s and the line approach around 1 GFLOP/s. With problem sizes greater than $2 \cdot 10^6$ the four combinations and the reference solution converge and remain steady at about 1.2 GFLOP/s.

The plot in Figure 13 contains the strong scaling behavior with $1 \leq C \leq 96$ cores and a fixed problem size of $N^2 = 8192^2$ (up to 1.5GB of data in total) measured over $T = 100$ time steps. In Figure 14, we show the weak scaling behavior with a problem size of approximately $N^2 = C \cdot 8192^2$ (up to 1.5GB of data per

[7] http://www.centos.org/
[8] http://www.ibm.com/systems/de/platformcomputing/products/lsf/
[9] http://www.open-mpi.org/
[10] http://gcc.gnu.org/

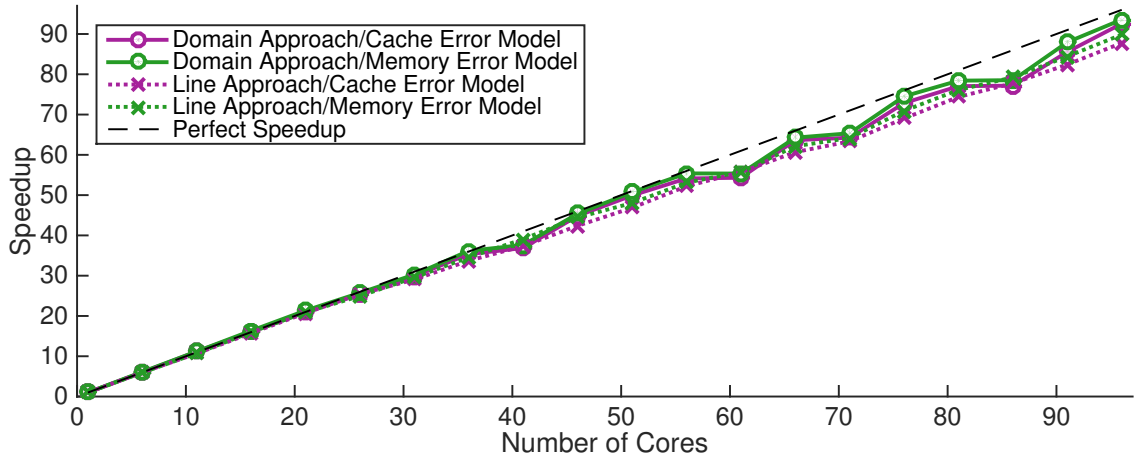


Figure 13: Strong scaling behavior of the four combinations in terms of speedup against number of cores.

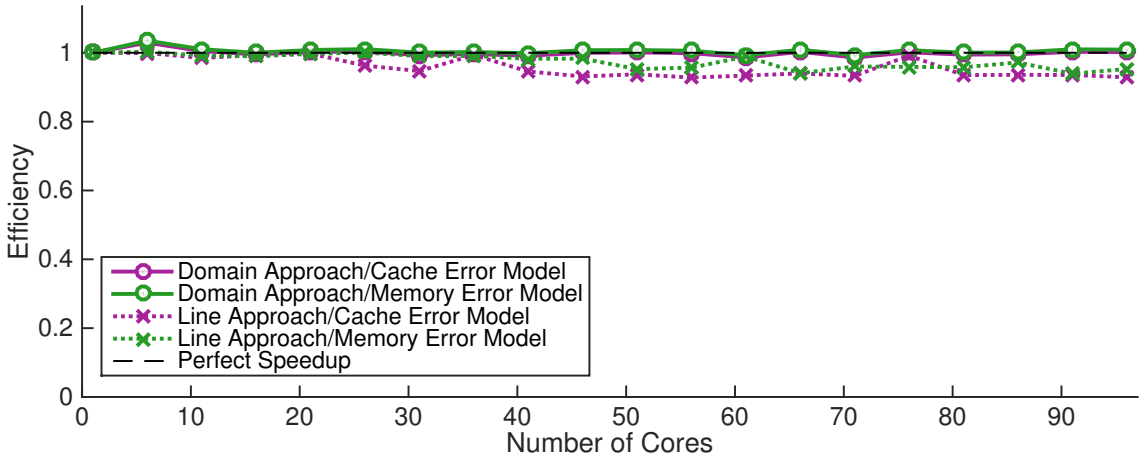


Figure 14: Weak scaling behavior of the four combinations in terms of efficiency versus number of cores.

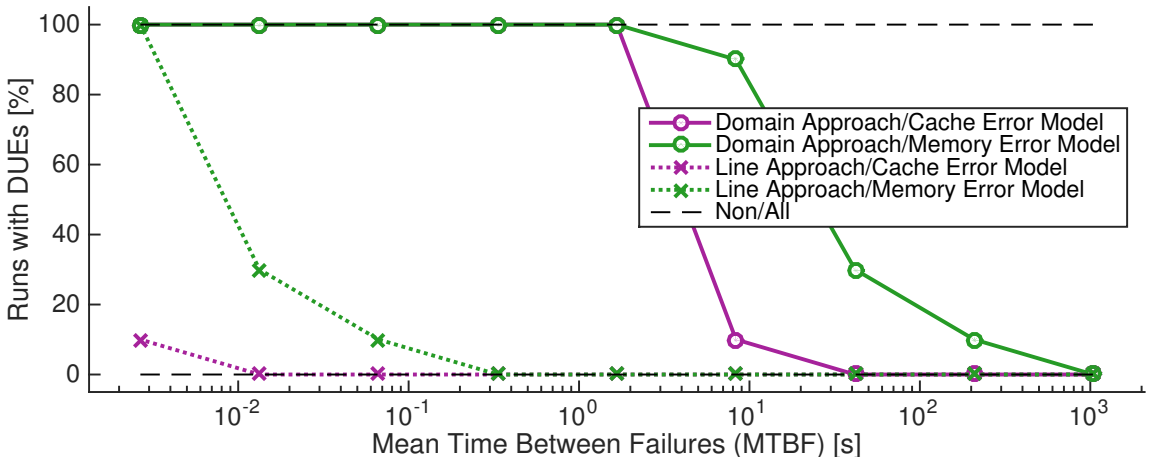


Figure 15: The percentage of runs suffering from detected unrecoverable errors (DUEs) as a function of the mean time between failures (MTBF).

core) also measured over $T = 100$ time steps. All four combinations use their single-core execution time as the reference for the speedup and the efficiency.

Both, the weak and the strong scaling plot, disclose a near perfect scaling. However, the performance is subject to some fluctuation because of the following reason: If the number of cores C is a square number then the simulation domain can be evenly partitioned along both axes resulting in square subdomains. Otherwise, it is partitioned into narrower and taller rectangles culminating in vertical stripes if C is prime. The more a subdomain deviates from a square the larger its surface becomes. This in turns leads to a performance loss because more boundary values must be exchanged in-between time steps. The effect is extenuated in the line approach where taller subdomains imply more horizontal lines and, thus, more protected blocks. This results in more kernel invocations which cause an additional performance deterioration.

4.3 Results with Error Injection

In this section, we quantitatively cover the properties of the solver under the influence of soft errors. More precisely, detected unrecoverable errors (DUEs), silent data corruptions (SDCs) and the runtime overhead are investigated. These properties are plotted against the mean time between failures (MTBF), i.e., the inverse of λ set for the injector's exponential distribution for the whole simulation. Ten runs where started for every value of λ, each with $C = 96$ cores, a problem size of $N^2 = C \cdot 8192^2$ and $p = .5$ for the geometric distribution for MCUs . We encountered problems with MTBF below 10^{-3} seconds with `SIGTRAP` stopping signals not being recognized by the injector but directly delivered to the solver. That is why only higher MTBFs are shown. Note that the solver has a natural masking rate of approximately 50% and 67% in the line and domain approach, respectively, when errors are injected according to the memory error model. The reason is that soft errors in one out of two and two out of three domains, respectively, are often benign because they are never read.

The DUE occurrences are plotted in Figure 15. The solver may throw exceptions indicating DUEs as mentioned in Section 3.1.2. All combinations exhibit the same behavior: With a large MTBF they always finish without any problem. At some point, they start detecting DUEs and the percentage of runs detecting them increases as the MTBF decreases. Finally, all runs encounter DUEs if the MTBF is

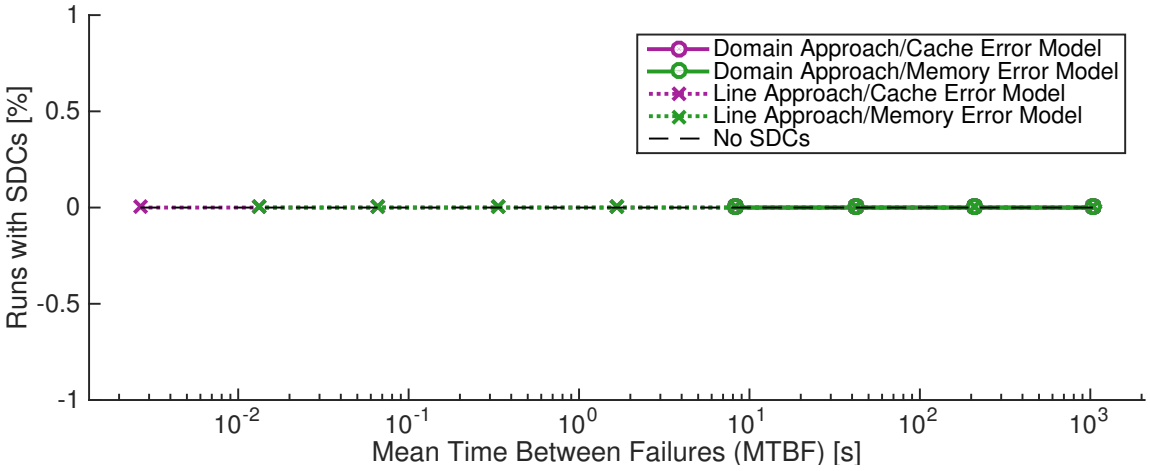


Figure 16: The percentage of runs with silent data corruptions (SDC) against the mean time between failures (MTBF).

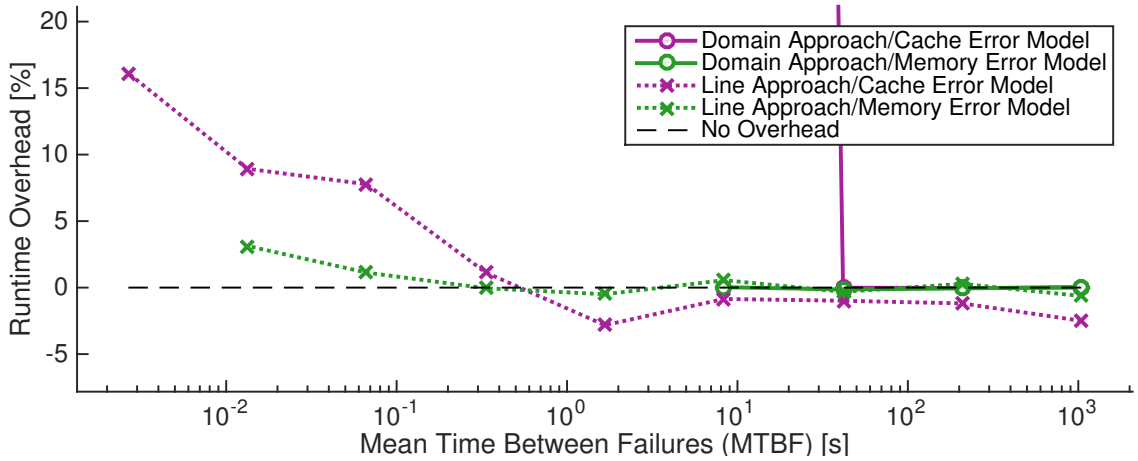


Figure 17: The runtime overhead of the four combinations as the ratio between the runtime under the influence of errors and error-free runtime as a function of the mean time between failures (MTBF).

roughly two orders of magnitude lower than for the first detections. An exception is the line approach under the cache error model for which the behavior cannot be determined within the sampled error rates, although, we expect the same behavior. The position of this transition depends on the combination: The domain approach is more sensible to soft errors than the line approach. And both approaches are more resilient in the cache error model than in the memory error model. Runs with DUEs are excluded from the remaining results because they abort at any point in the simulation and their runtime becomes meaningless.

The SDCs are depicted in Figure 16. In principle, it is possible that combinations of soft errors remain undetected because of checksum collision (cf. Section 2.2.2) or that the same bit position is hit in TMR (cf. Section 2.2.3). But all runs produced the correct solution of the test problem with regard to the CRC checksum as described in Section 3.1.2.

Next, the runtime overhead due to recovery of the four combinations is plotted in Figure 17. The domain

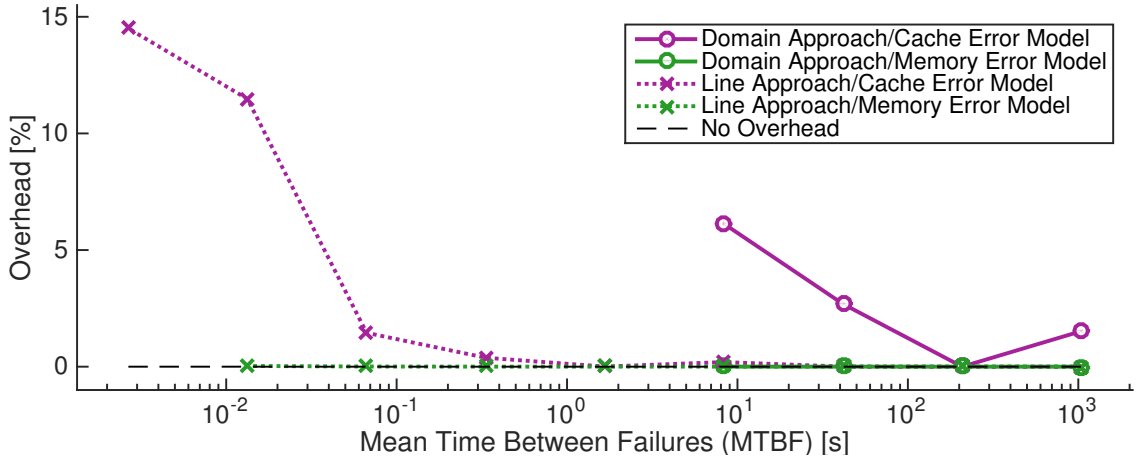


Figure 18: Relative execution time spent in the injector.

approach under the cache error model exhibits an enormous overhead of 530% (not shown in the plot). A closer investigation revealed that flushing the cache was the major cause. The same approach in the memory error model has a negligible overhead as long as some runs finish without DUEs. The line approach has a very low runtime overhead for MTBF rates greater than 1s. Below that, the overhead increases to 16% and 3% in the cache and memory error model, respectively, as the MTBF decreases.

Depending on the error model, a part of the runtime overhead may be due to the injector as shown in Figure 18. In the memory error model, peeking and poking the data is negligible. In the cache error model, the injector steps through the instructions of the solver one by one as described in Section 3.2.3. The resulting performance penalty is clearly visible: Up to 14.5% of the total execution time is spent within the injector in the line approach. In the case of the domain approach, the effect is with 6% small compared to the runtime overhead.

5 Discussion

In this section, we discuss the advantages and disadvantages of the error models and the approaches in Section 5.1 and Section 5.2, respectively. We also give an outlook on future research in Section 5.3.

5.1 Cache Error Model versus Memory Error Model

The two error models are compared by means of their memory and runtime overhead, the resilience of both approaches against them and their testability. A final note on the choice of the memory model is made at the end of this section.

The cache error model outperforms the memory error model with respect to the memory overhead but not to runtime overhead. Both approaches consume less memory because no recovery blocks are required. The comparison is more difficult for the runtime overhead because the domain approach under the memory error model encounters DUEs in every run before the trend of the runtime becomes visible. Given the remaining combinations, the overhead of the cache error model is considerable compared to the memory

error model, although, it is difficult to account for the influence of the injector.

Both approaches survive less errors before they encounter DUEs under the memory error model. The main reason is that the probability of a erroneous recovery block increases as the MTBF decreases. The only requirement – in the cache error model – to flush the cache and reload the data is advantageous in terms of feasibility but has a substantial runtime overhead with large amounts of data.

The memory error model has an advantage regarding testability. The complexity of the injector plays a considerable role because every fault tolerant application must also be tested. The cache error model is handicapped in this regards because the requirement for stepping through the instructions of the solver one by one, disassembling and inspecting them comes with expenses. The process of peeking, modifying and poking a word for the memory error model is less involved.

Of course, the error model is not chosen freely but should reflect the soft error properties of the clusters. The combinations for the memory error model must be chosen whenever the cluster encounters soft errors in the main memory on a regular basis. When this is unlikely, the combinations for the cache error model could be chosen. Given the fact that both approaches survive higher soft error rates than real clusters encounter, the simpler memory error model might be a good choice whenever sufficient memory is available.

5.2 Domain Approach versus Line Approach

In this section, we review the domain and the line approach with respect to the implementation complexity, memory and runtime overhead, soft error resilience and transmission volume. The discussion is summarized at the end of the section.

The situation regarding the implementation complexity is twofold. The design and implementation complexity is an important aspect of fault-tolerant systems because complex strategies for fault resilience run into danger of decreasing the overall system reliability as elaborated in Section 1.2.2. On one hand, the single-core versions of the initialization, time-step and termination kernels are less complex in the domain approach due to the fact that they involve fewer input and output blocks. On the other hand, the retransmission request in the domain approach must be carefully considered. To be fair, the alternative TMR transmission which caused the recovery problem to be local was introduced as a solution to a problem specific to the line approach but could also be adopted in the domain approach.

The situation is uniform with respect to the performance. Although the domain approach is slightly faster for small problem sizes, both methods have approximately the same performance and scale equally well for realistic problem sizes.

The domain approach has a considerable memory overhead of one discretized domain in addition to the overhead possibly imposed by the error model. This sums up to an overhead of 300% in the unfortunate combination of the domain approach for the memory error model. The smaller blocks of the line approach have a clear advantage because memory allocated for recovery can be released and reused in a much more fine-grained manner.

The same argument for small block sizes holds for soft errors resilience. They encounter not only less DUEs for the same SER but also the runtime overhead in the error case was reduced. Injection rates of up to one soft error per second have no impact with the line approach. The domain approach also

survives an unrealistically high MTBF of fifteen minutes without any problem.

The TMR transmission of the line approach triples the network volume. In the scenario of the test problem with a single conserved quantity in two dimensions, tripling the transmission volume is irrelevant because large cluster computers usually provide fast connections between their nodes. For problems with more conserved quantity and more dimensions, the transmission method should be reconsidered.

Since soft errors are only a concern in large simulations, memory is normally a scarce resource. The line approach outperforms the domain approach in this regard. Despite an increased implementation complexity, the line approach is likely to be the better choice independently of the error model.

5.3 Future Research

The implementation of the solver is the product of a variety of design decisions. Many options have been considered and even tested, but others remain to be investigated. The latter include: more interesting test problems with more conserved quantities in higher dimensions, other transmission methods, vectorization of the kernels allowing non-temporal store instruction and other checksum operations, etc.

A major limitation of the thesis is that the soft error injections have been restricted in space and time. Real soft errors may occur anywhere at any time possibly provoking erratic behavior of the solver. The claim that protection of scientific data and their computation against soft errors improves the overall system reliability to a great extent remains to be verified.

The solver runs without any special requirements if the injector is the only source of soft errors. If a real soft error would occur, the hardware might detect it and abort the solver or the whole system. Therefore, hardware-side soft error detection must be turned off in order for the solver to develop its full potential. How this could be achieved goes beyond the scope of this thesis.

Finally, the thesis is motivated in MLMC methods but only covers the example of a finite difference solver. Although the solver is a major part of MLMC methods, further research would be necessary to accomplish a fully soft error resilient multilevel Monte Carlo simulation.

6 Conclusion

We have briefly covered the principles of reliable computing, in particular, soft errors were systematically located in this field of research and the need for soft error resilient methods was emphasized by empirical studies. The project was motivated on the basis of previous research and given a concise overview of the many related projects.

Based on the empirical studies, we have deduced two error models: The cache error model that assumes solely transient errors in the cache and the memory error model which permits permanent errors in the whole memory hierarchy.

We have developed a recipe for soft error resilient applications based on three principles: First, individual values or small blocks can be stored in triple modular redundancy (TMR), thus completely masking soft errors. Second, a protected block consisting of a set of values and a checksum enables detection of soft

errors in large chunks of memory with low runtime overhead. Third, a system may recover from soft errors by either reloading erroneous data from the main memory or reconstructing the fault memory region depending on the error model.

Using this recipe, a test application has been implemented for the example of a two-dimensional heat equation with initial and Dirichlet boundary condition which was discretized on a regular grid. The simulation phase including the ghost-layer exchange in distributed memory environments is soft error resilient. Two main approaches have been distinguished based on the size of protected blocks: The domain approach treats a whole discretized domain as a single protected block while the line approach treats every individual line of the discretized domain as a separate protected block.

The test application is accompanied by a soft error injection tool based on the `ptrace()` system call. This injector simulates soft errors of both error models and was used to test the behavior of the solver under the influence if soft errors. The injector carried out its work smoothly up to a few hundred errors per second. With a higher error frequency, problems with the timing of signals occurred possibly indicating a bug.

The cache error model has a clear advantage regarding the memory overhead but is accompanied with a complex testing procedure. Despite a slight increase in complexity, the line approach has a lower runtime overhead and a higher soft error resilience compared to the domain approach.

The main contribution of this thesis is a software method capable of detecting and possibly correcting soft errors on a bit-by-bit level. The concrete implementation of a time stepping scheme is subject to many design decisions. Only a few have been implemented in this thesis, many others remain to be investigated in the future. Also, unrestricted soft error injection deserves attention in a future project.

Acknowledgements

I would like to thank the following people for helping me in various phases during this thesis: Peter Arbenz for the opportunity of this project and the many hours of constructive discussions, Stefan Pauli and Jonas Šukys for clarifications regarding the feasibility of resilient multilevel Monte Carlo methods in the early stage of the project, James Elliott and Christian Engelmann for consulting us on soft errors in general and their simulation, Urban Borstnik for his support regarding the EULER cluster computer, Sebastian Kohler for the review of the thesis and, last be not least, Nathalie Pasche for the moral support.

References

[1] B. Atkinson et al. “Fault Injection Experiments with the CLAMR Hydrodynamics Mini-App”. In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Nov. 2014, pp. 6–9. DOI: 10.1109/ISSREW.2014.51.

[2] A. Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004). 03444, pp. 11–33. DOI: 10.1109/TDSC.2004.2.

[3] A. Benso and P. Prinetto. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Springer Science & Business Media, Oct. 2003.

[4] A. R. Benson, S. Schmit, and R. Schreiber. "Silent error detection in numerical time-stepping schemes". In: *International Journal of High Performance Computing Applications* (Apr. 2014). DOI: 10.1177/1094342014532297.

[5] X. Besseron et al. "Fault-tolerance and availability awareness in computational grids". In: *Fundamentals of Grid Computing: Theory, Algorithms and Technologies* (2009). 00003, p. 143.

[6] D. Boley et al. "Floating point fault tolerance with backward error assertions". In: *IEEE Transactions on Computers* 44.2 (Feb. 1995), pp. 302–311. DOI: 10.1109/12.364541.

[7] F. Cappello et al. "Toward exascale resilience". In: *International Journal of High Performance Computing Applications* (2009).

[8] F. Cappello et al. "Toward Exascale Resilience: 2014 update". In: *Supercomputing frontiers and innovations* 1.1 (June 2014), pp. 5–28. DOI: 10.14529/jsfi140101.

[9] D. Costa et al. "Xception^{TM}: A Software Implemented Fault Injection Tool". In: *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Ed. by A. Benso and P. Prinetto. Frontiers in Electronic Testing 23. Springer US, 2003, pp. 125–139.

[10] E. Dubrova. *Fault-tolerant design*. Springer, 2013.

[11] J. Elliott, M. Hoemmen, and F. Mueller. "Tolerating Silent Data Corruption in Opaque Preconditioners". In: *arXiv preprint arXiv:1404.5552* (2014).

[12] J. Elliott et al. *Quantifying the impact of single bit flips on floating point arithmetic*. Tech. rep. ORNL/TM-2013/282, Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, 2013. 6, 9, 2013.

[13] E. N. Elnozahy (Mootaz) et al. "A Survey of Rollback-recovery Protocols in Message-passing Systems". In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408. DOI: 10.1145/568522.568525.

[14] ETH HPC Team. *ClusterWiki – Euler II fully operational*. URL: http://clusterwiki.ethz.ch/brutus/Euler_II_fully_operational (visited on 07/28/2015).

[15] ETH HPC Team. *ClusterWiki – Introducing EULER*. URL: http://clusterwiki.ethz.ch/brutus/Introducing_EULER (visited on 07/28/2015).

[16] K. Ferreira et al. "Evaluating the Viability of Process Replication Reliability for Exascale Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. New York, NY, USA: ACM, 2011, 44:1–44:12. DOI: 10.1145/2063384.2063443.

[17] A. Geist and C. Engelmann. "Development of naturally fault tolerant algorithms for computing on 100,000 processors". In: *Journal of Parallel and Distributed Computing* (2002).

[18] A. Geist and D. A. Reed. "A survey of high-performance computing scaling challenges". In: *International Journal of High Performance Computing Applications* (Aug. 2015). DOI: 10.1177/1094342015597083.

[19] M. B. Giles. "Multilevel Monte Carlo Path Simulation". In: *Operations Research* 56.3 (June 2008), pp. 607–617. DOI: 10.1287/opre.1070.0496.

[20] J. N. Glosli et al. "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability". In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007, p. 58.

[21] D. Goldberg. "What Every Computer Scientist Should Know About Floating-point Arithmetic". In: *ACM Comput. Surv.* 23.1 (1991), pp. 5–48. DOI: 10.1145/103162.103163.

[22] G. Gonnet and M. Wodzislawski. "Testing in large-scale scientific computation: the short circuit method". In: *Proc. of the CAST* 2010 (2010).

[23] Q. Guan et al. “F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability”. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. 2014, pp. 1245–1254. DOI: 10.1109/IPDPS.2014.128.

[24] K.-H. Huang and J. A. Abraham. “Algorithm-based fault tolerance for matrix operations”. In: *IEEE Transactions on Computers* 100.6 (1984). 00879, pp. 518–528.

[25] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C*. 2014.

[26] Intel Corporation. *Intel® C++ Intrinsic Reference*. 2014.

[27] Intel Corporation. *Intel® Xeon® Processor E5 v3 Product Family Datasheet, Vol. 2*.

[28] International Organization for Standardization (ISO). *ISO International Standard ISO/IEC 14882:2014(E) – Programming Language C++ [Working draft]*. 2014.

[29] M. Kohler. “Fault-Tolerant Multicore Master/Worker Application with ULFM”. Semester Project. Zurich, Switzerland: ETH Zurich, Computer Science Department, Dec. 2014.

[30] M. Kohler. “Fault Tolerant Multilevel Monte Carlo Implementation with User Level Failure Mitigation”. Bachelor’s Thesis. Zurich, Switzerland: ETH Zurich, Computer Science Department, Sept. 2013.

[31] M. Leeke et al. “A methodology for the generation of efficient error detection mechanisms”. In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. June 2011, pp. 25–36. DOI: 10.1109/DSN.2011.5958204.

[32] P. A. Lee and T. Anderson. *Fault Tolerance*. Ed. by A. Avižienis, H. Kopetz, and J. C. Laprie. Vol. 3. Dependable Computing and Fault-Tolerant Systems. Vienna: Springer Vienna, 1990.

[33] D. Li, J. S. Vetter, and W. Yu. “Classifying Soft Error Vulnerabilities in Extreme-scale Scientific Applications Using a Binary Instrumentation Tool”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, 57:1–57:11.

[34] S. Li and X. Li. “Soft error propagation in floating-point programs”. In: *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International*. 2010, pp. 239–246. DOI: 10.1109/PCCC.2010.5682305.

[35] B. R. Lowery. “Relative error due to a single bit-flip in floating-point arithmetic”. In: *arXiv preprint arXiv:1304.4292* (2013).

[36] J. Meng et al. “Exploiting the forgiving nature of applications for scalable parallel execution”. In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Apr. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470469.

[37] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann, Aug. 2011.

[38] T. Naughton et al. “Fault Injection Framework for System Resilience Evaluation: Fake Faults for Finding Future Failures”. In: *Proceedings of the 2009 Workshop on Resiliency in High Performance*. Resilience ’09. New York, NY, USA: ACM, 2009, pp. 23–28. DOI: 10.1145/1552526.1552530.

[39] S. Pauli, P. Arbenz, and C. Schwab. “Intrinsic fault tolerance of multi level Monte Carlo methods”. In: *ETH Zurich, Computer Science Department, Tech. Rep* (2012).

[40] S. Pauli, M. Kohler, and P. Arbenz. “A fault tolerant implementation of Multi-Level Monte Carlo methods.” In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. Ed. by M. Bader et al. IOS Press, 2014, pp. 471–480.

[41] S. M. Pauli. “Fault Tolerance in Multilevel Monte Carlo Methods”. PhD Thesis. Zurich, Switzerland: ETH Zurich, Computer Science Department, 2014.

[42] J. Plank, Y. Kim, and J. Dongarra. "Algorithm-based diskless checkpointing for fault tolerant matrix operations". In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers*. 00068. June 1995, pp. 351–360. DOI: 10.1109/FTCS.1995.466964.

[43] J. Plank and K. Li. "Faster checkpointing with N+1 parity". In: *Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers*. 00000. June 1994, pp. 288–297. DOI: 10.1109/FTCS.1994.315631.

[44] G. Reis et al. "SWIFT: software implemented fault tolerance". In: *International Symposium on Code Generation and Optimization, 2005. CGO 2005.* Mar. 2005, pp. 243–254. DOI: 10.1109/CGO.2005.34.

[45] J. Rexford and N. Jha. "Algorithm-based fault tolerance for floating-point operations in massively parallel systems". In: *1992 IEEE International Symposium on Circuits and Systems, 1992. ISCAS '92. Proceedings*. Vol. 2. 00010. 1992, 649–652 vol.2. DOI: 10.1109/ISCAS.1992.230168.

[46] S. Sahoo et al. "Using likely program invariants to detect hardware errors". In: *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008.* June 2008, pp. 70–79. DOI: 10.1109/DSN.2008.4630072.

[47] J. H. Saltzer, D. P. Reed, and D. D. Clark. "End-to-end Arguments in System Design". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984). 02473, pp. 277–288. DOI: 10.1145/357401.357402.

[48] B. Schroeder and G. A. Gibson. "Understanding failures in petascale computers". In: *Journal of Physics: Conference Series* 78.1 (July 2007). 00279, p. 012022. DOI: 10.1088/1742-6596/78/1/012022.

[49] B. Schroeder, E. Pinheiro, and W.-D. Weber. "DRAM errors in the wild: a large-scale field study". In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 37. ACM, 2009, pp. 193–204.

[50] P. Shivakumar et al. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 389–398.

[51] A. Shye et al. "Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07.* June 2007, pp. 297–306. DOI: 10.1109/DSN.2007.98.

[52] M. Snir et al. "Addressing failures in exascale computing". In: *International Journal of High Performance Computing Applications* (Mar. 2014). DOI: 10.1177/1094342014522573.

[53] N. Sri Hari Krishna et al. "Using Loop Invariants to Fight Soft Errors in Data Caches". In: *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*. ASP-DAC '05. New York, NY, USA: ACM, 2005, pp. 1317–1320. DOI: 10.1145/1120725.1121052.

[54] E. J. Weyuker. "On Testing Non-Testable Programs". In: *The Computer Journal* 25.4 (Nov. 1982), pp. 465–470. DOI: 10.1093/comjnl/25.4.465.

[55] N. Wulf et al. "SCIPS: An emulation methodology for fault injection in processor caches". In: *2011 IEEE Aerospace Conference*. Mar. 2011, pp. 1–9. DOI: 10.1109/AERO.2011.5747450.