ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Επιστήμης Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών
Συστημάτων (MicroLab)

On the Dependability of Transient Neuron Simulations

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Μαυρογιάννη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Αθήνα, Ιούλιος, 2014

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Επιστήμης Υπολογιστών
Εργαστηρίο Μικροϋπολογιστών & Ψηφιακών
Συστημάτων (MicroLab)

On the Dependability of Transient Neuron Simulations

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αλέξανδρου Μαυρογιάννη

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 09/07/2014

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κιαμάλ Πεκμεστζή	Νεκτάριος Κοζύρης
Επίκουρος Καθηγητής	Καθηγητής	Καθηγητής

Αθήνα, Ιούλιος, 2014

.....

Αλέξανδρος Μαυρογιάννης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών
Ε.Μ.Π.

Abstract

Recent trends in many-core multiprocessor design focus on the aggressive integration of multiple cores on a single chip, aiming for performance and power scalability benefits through increased thread-level parallelism. In such environments, reliability is becoming an increasing concern, as increases in the number of cores tend to drastically reduce the mean time to failure of these systems.

Current approaches in many-core and High-Performance Computing (HPC) systems utilize Checkpoint/Restart (C/R) methods to provide fault tolerance. However, these methods are often performed manually or used without addressing the overall dependability profile of the system. Specifically, important concerns often are overlooked, such as automated error resolution, detection of data corruption, thermal-aware chip usage and minimization of time redundancy.

The approach proposed in this thesis addresses these concerns by introducing Depman, a unified runtime environment that resolves common error profiles through C/R and other system-level actions. The resulting framework adapts to variabilities of the system's reliability in order to minimize the total time overhead and provide a high degree of dependability to the managed systems and applications.

Depman was tested through the use of fault injection, performing in accordance to its specification. The adaptive time overhead optimization scheme demonstrated potential for benefits in performance and energy consumption in systems with time dependent failure rates. The resulting work was submitted for publication [32].

Περίληψη

Οι σύγχρονες τάσεις στην σχεδίαση πολυπύρηνων επεξεργαστών επικεντρώνονται στην επιθετική ενσωμάτωση πολλαπλών πυρήνων εντός του ίδιου κυκλώματος, αποσκοπώντας σε οφέλη επίδοσης και ισχύος μέσω του αυξημένου παραλληλισμού επιπέδου νήματος. Σε τέτοια περιβάλλοντα, η αξιοπιστία αποκτά ολοένα και αυξανόμενο ενδιαφέρον, καθώς οι αυξήσεις στον αριθμό πυρήνων οδηγούν στην δραστική μείωση του μέσου χρόνου αποτυχίας αυτών των συστημάτων.

Οι σύγχρονες προσεγγίσεις σε πολυπήρυνα συστήματα και συστήματα υπολογιστικής υψηλών επιδόσεων χρησιμοποιούν μεθόδους Checkpoint/Restart (C/R) για να προσφέρουν ανοχή σε σφάλματα. Παρ'όλα αυτά, αυτές οι τεχνικές συχνά πραγματοποιούνται χειροκίνητα ή ανεξάρτητα απο την συνολική εικόνα της αξιοπιστίας του συστήματος. Συγκεκριμένα, σημαντικά ζητήματα παραβλέπονται, όπως η αυτόματη επίλυση σφαλμάτων, η ανίχνευση αλλοιώσεων δεδομένων, η αξιοποίηση της πλατφόρμας με επίγνωση των θερμικών χαρακτηριστικών της και η ελαχιστοποίηση του προκύπτοντος χρονικού πλεονασμού.

Η προσέγγιση που προτείνεται σε αυτήν την διπλωματική εργασία θίγει αυτά τα ζητήματα με την παρουσίαση του Depman, ενός ενοποιημένου περιβάλλοντος εκτέλεσης που επιλύει συχνά εμφανιζόμενα προφίλ σφαλμάτων μέσω τεχνικών C/R και άλλων δράσεων επιπέδου συστήματος. Το προκύπτον πλαίσιο προσαρμόζεται στη μεταβλητότητα της αξιοπιστίας του συστήματος με σκοπό την ελαχιστοποιήση του πλεονάζων χρόνου, προσφέροντας εναν υψηλό βαθμό αξιοπιστίας στα διαχειριζόμενα συστήματα και εφαρμογές.

Το Depman ελέγχθηκε μέσω της χρήσης έγχυσης σφαλμάτων, επιτυγχάνοντας την ικανοποίηση των προδιαγραφών του. Το σχήμα προσαρμοζόμενης βελτιστοποίησης του πλεονάζων χρόνου επέδειξε ενδεχόμενα οφέλη επίδοσης και ενεργειακής κατανάλωσης σε συστήματα με χρονοεξαρτώμενους ρυθμούς σφαλμάτων. Η προκύπτουσα δουλειά κατατέθηκε προς δημοσίευση.

Acknowledgements

The work described in this thesis was carried out at the Microprocessors Laboratory and Digital Systems Lab of the School of Electrical and Computer Engineering of the National Technical University of Athens, under the supervision of Prof. D.J. Soudris and Mr. D. Rodopoulos.

I would like to express my gratitude to both of them for their guidance and support throughout the project's duration and for introducing me to the challenging field of system dependability. I am truly grateful to them for their invaluable research experience and their positive attitude, which were of paramount importance to the development of this work, as well as for the trust and freedom they placed in me throughout the project's duration.

Finally, I would like to thank my friends and family for the support they have showed me throughout the duration of this thesis.

Αλέξανδρος Μαυρογιάννης

Contents

List of Symbols and Abbreviations

ℓ	Checkpoint Latency
λ	Failures in Time rate
τ	Checkpoint Interval
τ_{opt}	Optimal Checkpoint Interval (time units)
τ'_{opt}	Optimal Checkpoint Interval (simulation steps)
E_{eff}	Energy Efficiency
k	Number of Checkpoints
M	Moving Average Filter Length
$MTBF$	Mean Time Between Failures
$MTTF$	Mean Time to Failure
$MTTR$	Mean Time to Repair
N	Total Number of Checkpointing Cycles
P_{eff}	Performance Efficiency
R	Repair Time
T_r	Rollback Time
TTF	Time to Failure
W	Waste Time

List of Figures

Chapter 1

Introduction

In the recent years, vendors of computing systems consider the integration of multiple processing nodes as a viable solution for the emerging upper bound on single-processor performance [17]. This trend has been observed both in the context of Embedded Computing (EC) and of High Performance Computing (HPC). In EC systems, this aggresive integration is performed on single dies or packages and it is required in order to provide diversification to the respective products [24]. Similarly, in HPC systems, the increasing trend multiple cores per socket integration is apparent and accompanied by a need for diversification, due to the convergence with cloud computing systems [45, 48].

Even though novel transistors exhibit significant improvements in their reliability profiles, especially in the case of soft error resilience [43], the aggressive silicon integration leads to increased failure rates at the circuit and system level[13]. Moreover, there is a class of semiconductor phenomena which exhibit high time-zero and time-dependent variability [39, 19]. As a result, the dependability of contemporary and future computing systems is becoming an increasing concern, as the components of both EC and HPC systems are subject to variability or degradation.

The dependability of a system can be broken down to five basic attributes: availability, reliability, safety, integrity and maintainability. Of those attributes, reliability and safety are the most important in systems with increasing failure rates, as the others are handled by the operating system or the corresponding error correction hardware.

In the context of this thesis, reliability is introduced to a many-core platform, Intel Labs' Single-Chip Cloud Computer (SCC) [23]. As a target application, a time-driven simulator of inferior olive neurons (InfOli) is used

[40]. This is achieved through the introduction of an adaptive dependability manager tool called *Depman*, an abbreviation for *Dependability Manager*. Depman orchestrates a Checkpoint/Restart scheme on target platform and application as a closed-loop control system. The aim of the proposed approach is to achieve fault tolerance, error recovery and error resilience on the present experimental setup.

The Depman tool achieves a high dependability standard of the target many-core system through several means. First of all, it monitors the system for both Detected Unrecoverable Error (DUE) and Silent Data Corruption (SDC) occurences [36]. Upon detection, it attempts to resolve the underlying failures through a series of actions called *countermeasures*. The Time To Failure (TTF) of each error occurence is measured and used to estimate the Mean Time to Failure (MTTF) of the system through a Moving Average filter. Finally, after the system is operational again, the application is optimally restarted in a thermal-aware configuration.

In contrast to recent trends in Checkpoint/Restart methods, Depman does not require the definition of any design-time benchmark parameters. The proposed system automatically estimates the overhead of the C/R implementation and adapts to time-dependent error failure rates of the system. These estimates are used to select an optimal checkpoint rate during the restart operation, in order to minimize the total wall-clock time of the target application. The performed experimental results confirm that adaptive estimation offers performance and energy efficiency improvements over implementations which are agnostic of the system's rate of failure occurence.

In conclusion, we introduce a framework that unifies existing approaches in ensuring the dependability of many-core systems in a closed-loop control system. The resulting framework performs error monitoring and error resolution operations while maintaining thermal-aware chip utilization and minimal overhead in environments with time-dependent failure occurences.

The current thesis is structured as follows: Chapter 2 presents samples of prior art which motivate the novel features of Depman. Chapter 3 introduces the present experimental setup, the implemented Checkpoint/Restart scheme and the error injection techniques that are used. Chapter 4 presents the specifications and implementation of the Depman Tool. Finally, Chapter 5 is dedicated to checkpoint interval optimization and the adaptive estimation of the system's failure rate and Chapter 6 presents conclusions and directions of future work.

Chapter 2

Prior Art

2.1 Introduction

In this chapter we will present the state of the art on supervised Checkpoint/Restart methods on many-core streaming applications. In order to achieve this, the graph shown in Figure 2.1 will be used. This graph shows binary splits of each field of study, where siblings represent complementary design choices. The explored field of study serves as the tentative root of the graph. Each path in this tree structure describes a series of design choices that can be made during the implementation of supervised Checkpoint/Restart models on many-core streaming applications.

As it is apparent in Figure 2.1, the field can be initially divided on whether the Checkpoint/Restart snapshots are taken on the system level, or on the application level. Afterwards, the terms *Multiple Error Profiles* and *Single Error Profile* are used to describe the specification of the system in terms of handling multiple types of errors, such as the distinction of data corruption and unrecoverable errors. The final split is between the terms *Static CIO* and *Dynamic CIO*, which refer to the selection of an optimal checkpoint interval and whether this selection dynamically adjusts to changes of the system or whether it remains static throughout the application runtime. In the rest of this chapter, we will take a closer look at the research landscape, addressing sequentially each level of the tree.

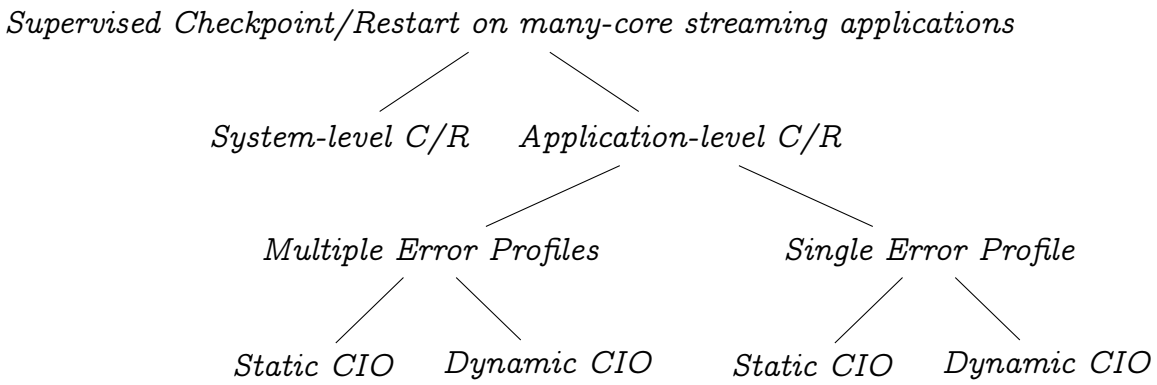


Figure 2.1: Research Landscape Tree

2.2 Checkpoint/Restart

Checkpoint/Restart (or C/R) is the technique by which fault tolerance is introduced to a system through the storage of snapshots of the system state, called checkpoints. In the event of a failure, these checkpoints can be used to restore the system to a stable state. The added fault tolerance is provided at the cost of added time and data redundancy [25], required for the storage of the snapshots. Checkpoint/Restart has been used thoroughly in the context of High Performance Computing systems[21, 5, 1, 6, 28], but the techniques and methods used can be applied to many-core platforms as well. C/R has been used primarily as a software technique, but hardware implementations have been explored as well [35].

Uses of Checkpoint/Restart in distributed systems need to address the issue of checkpoint consistency between all nodes of the system. In such systems, individual nodes often encounter heavy packet loss and communication latencies, requiring the ensurance that all checkpoints taken correspond to the correct state in time of the distributed system. In these cases, checkpointing is either used over unified distributed storage schemes it is managed through coordination schemes [49].

Of all the coordination techniques available, the one requiring the least programming effort is the use of blocking communication, which introduces significant time overhead that scales poorly when new nodes are added to the system. On the other hand, non-blocking techniques avoid this overhead, but they must correctly manage the receipt of early and late messages [49, 5].

2.2.1 System-level and Application-Level C/R

Checkpoint/Restart can be implemented either on the system level, through storage of the platform's state such as memory and register contents, or on the application level, through storage of application-specific data such as data structures and variable values [44]. A unique case of system-level Checkpoint/Restart can be implemented at the user level by extracting the application state from existing userspace systems, such as the Message Passing Interface (MPI).

Several tools have been created to automate the implementation of Checkpoint/Restart. A popular choice for system-level C/R in High Performance Computing (HPC) is Berkeley Labs Checkpoint Restart (BLCR) [21], a Linux kernel module that utilizes MPI callbacks to store distributed snapshots. Addditionally, a user-level choice for MPI applications is Distributed Multithreaded Checkpointing (DMTCP) [1], which operates transparently on the userspace, requiring no interference with the system kernel or the target application.

Application-level C/R methods can potentially outperform their system-level counterparts [44], as the snapshot size is generally smaller. This is due to the fact that application-level methods only require the storage of critical components of the application state at each checkpoint interval, while methods performed by the operating system rely on larger, system-wide snapshots. Moreover, they are highly more portable, as they are transparent to the target platform. However, the implementation of an efficient application-level C/R technique requires more programmer effort, especially in the case of distributed and parallel programs.

Even though the intrusive nature of application-level Checkpoint/Restart implementations usually requires the development of custom solutions, there are several compiler-assisted tools which facilitate this process. One such tool frequently used in HPC applications is The Cornell Checkpointing pre-Compiler[5, 6], also known as C^3. This framework parses programmer-placed directives on the application source code in order to locate potential checkpointing and restarting locations. It then evaluates these locations with call-graph analysis and injects the necessary code snippets to the code base. It also features a runtime environment, used with distributed processes, which manages the checkpointing process through a coordination layer between the Message Passing Interface (MPI) library and the application.

2.2.2 Modelling C/R

Checkpoint/Restart is modeled through several parameters. The *Checkpoint Interval* (τ) is the elapsed time period before taking a checkpoint. The *Checkpoint Latency* (ℓ) , also mentioned as *Checkpoint Overhead* in the literature, is the time interval used to create and store a checkpoint. The actual time required to repair and restore the system and application state is called the *Repair Time* (R) and it is often assumed constant. Finally, the time interval between the detection of an error and the last checkpoint is the *Rollback Time* (T_r) and it represents the lost computational time that needs to be performed again after the restart process. The sequential time ordering of τ, ℓ, T_r and R is called a *checkpointing cycle* and it can be observed in figure 2.2. In this figure, it is assumed that checkpoints are taken equidistantly in time.

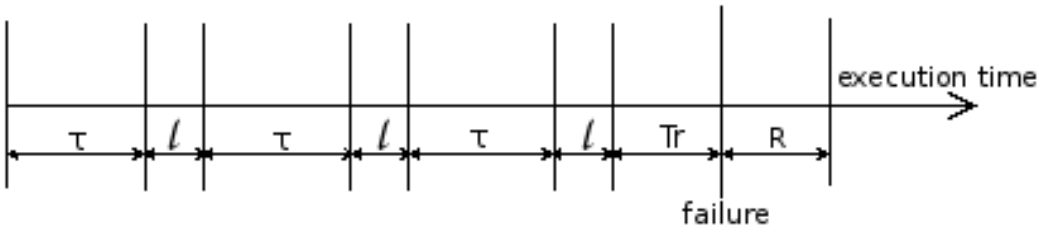


Figure 2.2: Time Modelling of the Checkpointing Cycle

2.3 Error Profiles

An important aspect of systems using C/R methods is the classification of potential errors and the actions taken by the system for their resolution. In the case of multiprocessing systems, faults either manifest themselves as Detected Unrecoverable Failures (DUE), also known as hard errors, or they remain unnoticed as Silent Data Corruptions (SDC) [36].

Detected Unrecoverable Failures can be caused by various categories of faults, such as chip overheating or voltage spikes. The most important characteristic of DUEs is that the application process is either terminated or blocked when they manifest. Due to this reason, our proposed approach will also consider software quality errors that terminate the application process, such as segmentation faults and operating system errors.

Silent Data Corruptions are unintended variations in the system memory or filesystem that are not detected by the corresponding hardware or

firmware. They may be caused by various faults, such as hardware degradation, electromagnetic interference, cosmic radiation, firmware bugs or data transfer noise. Given their elusive nature, the only reliable way to detect and resolve SDC occurences is through triple redundancy of the target application. This method has been mainly used in HPC systems [16], where high resource availability can overcome the increased computational cost. Even though SDC mitigation methods have been introduced at the hardware level [20, 37], the required architectural support renders them unusable in many existing systems.

In many-core systems, task redundancy comes at a greater cost, as the available resources are limited. Therefore, partial SDC mitigation can be achieved through validation of the affected data. In this case, an SDC monitor validates the data of interest in terms of format and plausibility, marking any non-conforming data as *latent errors*. The term latent errors implies the existence of a non-negligible error detection latency, which needs to be taken into account when other system reliability metrics are estimated.

In environments with latent errors, traditional Checkpoint/Restart techniques fail to ensure the recoverability of all potential errors, as they follow the fail-stop model. The fail-stop model assumes that there is no delay between error occurence and detection. Therefore, new checkpoints can be introduced during the error detection procedure, overwriting the checkpoints that correspond to states before the error occurence and rendering the error unrecoverable. In order to avoid this scenario, multiple versions of validated checkpoints need to be kept. This method, called multi-version checkpointing, is used to implement C/R in systems with latent errors [29].

System errors are often modeled through the *Mean Time to Failure* ($MTTF$ or M) metric, which is the mean time period of system uptime after which an error will occur. The inverse metric of the mean time to failure is called the Failures in Time (FIT) rate, and it is often denoted as λ. A FIT rate with a value of one declares that a failure will occur every one billion device hours. The FIT rate is often a more practical metric than the $MTTF$, as the FIT rate of a system can be expressed as the sum of the FIT rates of its components [36].

Additionally, the *Mean Time to Repair* ($MTTR$) is used to model the mean time duration of the system repair process. Finally, the *Mean Time Between Failures* ($MTBF$) metric is defined as the sum of the $MTTF$ and $MTTR$ values and represents the mean time period between the occurences of two failures [36]. $MTBF$ is also used in the bibliography as the mean

time duration of a checkpointing cycle.

Theoretical studies of failure/error rates in HPC and many-cores systems are clearly motivated on the time-dependent variability of these rates throughout the lifetime of the system [9, 14, 46]. However, state-of-the-art C/R implementations in both software and hardware often fail to address the issue of time-dependent failure rates, assuming a constant error distribution instead [2, 34].

2.4 Checkpoint Interval Optimization

The use of Checkpoint/Restart methods inevitably introduces time overhead on the target system, also called *Waste Time*. As the checkpoint interval (τ) is the only easily configurable parameter in the C/R model, several attempts have been made to approximate the optimal checkpoint interval through other parameters of the system. These attempts generally aim to model and minimize the total waste time or the total runtime of the system.

Ling et al. [27] proved through variational calculus that the optimal placement of checkpoint intervals is directly correlated to the system's failure rate. Through this statement, they confirmed previous suggestions [7] which state that time-varying failure rates require time-varying checkpointing intervals. As a result, attempts in the specification of optimal checkpoint intervals can be classified as periodic, which assume Poisson failure distributions, and aperiodic, which can be used on other failure distributions or adapt to the failure rate during runtime.

Checkpoint interval optimization has also been pursued through Markov availability models [18], aiming to maximize the system availability rather than minimize the total waste time. However, the two optimization problems are equivalent, and the same issues arise concerning the distribution of failures.

2.4.1 Periodic Checkpointing Approximations

The first attempt in defining an optimum checkpoint interval was performed by Young [51], whose first-order approximation introduced the attractively simple result of equation 2.1.

$$\tau_{opt} = \sqrt{2 \times MTTF \times \ell} \tag{2.1}$$

This was achieved through the minimization of a cost function that models the waste time of the system. Young's model, however, is based on the assumptions that error occurences follow an exponential distribution and that the repair time of the system (R) is negligible.

An expansion of Young's model that included R was introduced by Daly [10] who later defined a higher order estimate of the optimum checkpoint interval [11]. Daly expanded Young's first order model to include the repair time parameter in the estimation of the optimum checkpoint interval, thus considering the possibility of errors occuring during checkpointing. The result is expressed in equation 2.2.

$$\tau_{opt} = \sqrt{2 \times \ell \times (MTTF + R)} \tag{2.2}$$

Daly also introduced a higher order model, through which it was concluded that the repair time has no contribution to the selection of an optimum checkpoint interval. The complete results include the first three terms of the perturbation solution of the model, thus the simplified model of equation 2.3 was presented.

$$\tau_{opt} = \begin{cases} \sqrt{2 \times MTTF \times \ell} - \ell & \text{for } \ell < MTTF/2 \\ MTTF & \text{for } \ell \geq MTTF/2 \end{cases} \tag{2.3}$$

This simplification has a maximum relative error of 5% problem solution time toward the complete higher order result. Therefore, it can be used as a good rule of thumb for most practical systems, but it still follows the assumption that errors are distributed exponentially. Daly's model has also been enhanced to handle systems with detection latency in multi-version checkpointing techniques [29].

2.4.2 Aperiodic Checkpointing Techniques

In contrast to the previous assumption that system failures can be modelled as Poisson processes with a constant failure rate, real world systems often follow different distributions [42]. Therefore, as mentioned, periodic checkpoint placements cannot provide optimal minimization of the total execution time. In response to this issue, several techniques have been proposed proposed for other distribution types.

First of all, Oliner et al. proposed the idea of cooperative checkpointing [38] as an alternative to the previously mentioned periodic checkpointing techniques. In this method, the runtime system may decide to skip selected checkpoints, based on criteria such as storage or network contention and reliability information. The dynamic adaptation to system parameters enables the efficient and scalable handling of non-exponential failure distributions.

Moreover, Liu et al.[28] have defined a general model which determines the optimal checkpoint placement locations for arbitrary prespecified failure distributions. The performed measurements suggest performance improvements over the cooperative checkpoint model.

Implementations of the aforementioned aperiodic checkpointing techniques require either runtime coordination between the checkpointing mechanism and the system or specification of the checkpoint placement locations in the application. As a result, they cannot be transparently used with existing periodic checkpointing implementations.

Chapter 3

Experimental Setup

3.1 Target Platform & Application

In this chapter we shall explore the details of the target platform, Intel's Single-Chip Cloud Computer, and the target application, the InfOli simulator, in order to determine potential reliability violations of the experimental setup and their countermeasures.

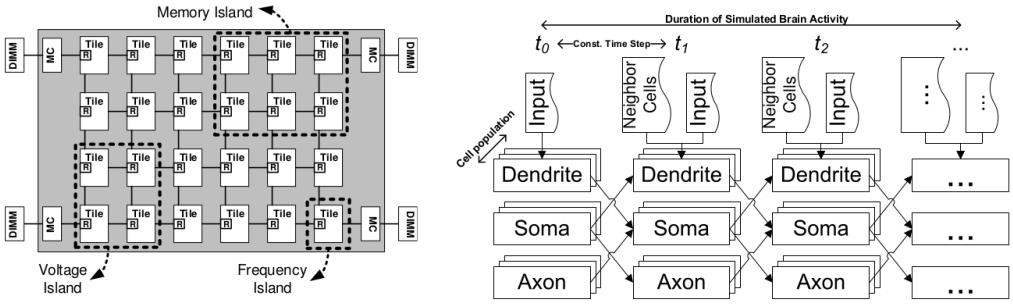
The Single-Chip Cloud Computer (SCC) is an experimental processor developed by Intel Labs as a platform for many-core software research [23, 31]. It is a homogeneous chip with 48 cores, grouped in tiles of two cores and interconnected through a mesh network. Each core has access to a Message-Passing Buffer (MPB), which is used to exchange messages between cores, and its own private memory. The SCC chip is hosted on a board that permits ethernet and PCIe communication with a Management-Console Personal Computer (MCPC). Through these links, the MCPC is able to provide power monitoring features and manage a shared directory, */shared*, between each of the SCC's cores and the MCPC. The chip's message passing capabilities are utilized by a programming model similar to the Message Passing Interface (MPI). This model is enabled by RCCE [30], a C library that is available on the MCPC.

Each tile of the SCC can be configured to a specific frequency value, allowing it to be refered to as a *frequency island*. Moreover, each pair of tiles can be configured to operate on different voltages, allowing them to be considered as *voltage islands*. Finally each quarter of the SCC chip can be considered as a *memory island*, as it is assigned a Dual Inline Memory Module (DIMM) through a Memory Controller (MC). These tile groupings

are apparent in Figure 3.1a

The interactions between the MCPC and the SCC are orchestrated through the *SCCKit* software framework. The framework is responsible for all operations on the SCC board, such as linux image booting, core restarting and reinitialization. The SCCKit also offers a GUI that contains these operations, along with useful monitoring information such as core utilization.

The application that will be used in the context of this thesis is a transient neuroscientific simulation, the InfOli Simulator [40, 8]. The simulator is a C implementation of a biologically plausible computational model of the Inferior Olivary nucleus [12, 4], based on the Hodgkin-Huxley model [22]. The simulated neurons are located in the mendula oblongata region of the brain and they are an important part of the brain's space perception and motor skills. The application utilizes an 8-way interconnectivity scheme among neighboring neurons, allowing exploitation of the SCC chip's message passing model. Moreover, the simulator operates at a constant simulation step of $\Delta t = 50\mu s$.



(a) The frequency, voltage and memory islands of the SCC

(b) Illustration of the InfOli simulator runtime

Figure 3.1: High level view of the target platform and application [40]

The simulator is initiated for a specific grid size and receives inputs through a connectivity file which declares the connections among the neuron cells and, optionally, a file of external current inputs for each neuron. If no external inputs are declared, the simulator generates pseudo-random input currents at every simulation step for each cell. The application produces a series of files, containing the axon voltages of each cell at the end of every simulation step.

Each neuron cell is simulated as a sequence of three individual compartments: the *dendrite* compartment, which receives voltages from neighboring cells, the *soma* compartment, which computes the cell's response based on these inputs, and the *axon* compartment, that propagates the output

to neighboring neurons. In the transient model of the InfOli simulator, communications between neurons are performed at the beginning of each simulation step $t_0, t_1, t_2, \ldots$, resulting to the data flow depicted in Figure 3.1b [40, 8]. Finally, the simulator is a system with memory, as each compartment alters its internal state during operation.

The InfOli simulator have been developed for the SCC in two porting options, the first utilizing data parallelism and the second combining both task and data parallelism. Porting Option 1 allocates all compartments of the same cell to a core, while Porting Option 2 allocates each comparment type to dedicated cores. For example, in a simulation with 48 cells (such as a grid size of 1x48), Porting Option 1 would assign one cell to each of the 48 available cores while Porting Option 2 would use three specialized groups of 16 cores, assigning one compartment type to each group.

Even though the first porting option outperforms the second one on a homogenous chip, the latter one produces more energy efficient mappings when introducing inhomogeneity. This inhomogeneity can be added to the platform by configuring the frequency and voltage islands of the SCC to different values. Therefore, such a configuration can allow the second porting option's groups of 16 cores operate on different voltage-frequency pairs, in a manner which maximizes the utilization of each core.

In the scope of the current thesis, only Porting Option 1 was considered for dependability management, as the second porting option creates an inhomogenous chip utilization which may alter the time to failure of individual cores, eventually lowering the availability of the system. Additionally, each core of the SCC was configured to $800Hz$, $1.1V$ and booted with a custom linux distribution. All tasks performed on the SCC were cross-compiled on the MCPC and all executable, input and output files were exchanged between the SCC and the MCPC through the shared directory.

3.2 Dependability Threats

Scientific simulations such as the InfOli simulator often require long total wall-clock times before providing meaningful results. Therefore, the experimental setup of should conform to high dependability standards in order to minimize the total platform runtime and to ensure the completion of the target application.

The three main dependability threats of systems are *faults*, *errors* and *failures*. Faults are defects in the system that can potentially produce

failures, but it is also possible that no error will manifest on certain input and state conditions. Errors are discrepancies between the system's actual behavior and its specified behavior. Failures are instances where the system behavior varies from its specification. Failures can be caused by errors, but not all errors necessarily result to failures. Failures can be handled through the use of fault tolerance mechanisms, allowing the overall system to continue operating according to its specification [36, 3, 25].

In order to achieve a dependable experimental setup, these threats must be countered through fault tolerance, fault avoidance, error removal and error forecasting. By combining these methods, the experimental setup should have the ability to confidently deliver its specified service.

In the case of the SCC and the InfOli simulator, no fault tolerance is provided against Detected Unrecoverable Errors (DUEs). These errors can be either platform-related, such as core shutdowns or failures, or system-specific, such as memory allocation errors or filesystem permission errors. This issue can be resolved by implementing a Checkpoint/Restart mechanism for the current application and restarting the application upon detection of such errors through a management process on the MCPC.

Furthermore, the experimental setup fails to account for permanent failures of SCC cores during large simulation runtimes. In the event of such a failure, the target application is unable to continue operating with the same configuration on the affected platform. On the other hand, a dependable simulation runtime should enable the simulator to continue operations at any remaining set of available cores, if possible.

Additionally, it is possible for Silent Data Corruptions (SDCs) to be introduced to the simulator's output in many stages throughout its storage. These corruptions may be caused by several factors, such as hardware faults, data transfer noise, poor electrical current distribution and electromagnetic interference. Even though it may not be always possible to detect the occurence of SDCs without using execution redundancy, there are several application-specific criteria that the output can be validated against.

In the case of the InfOli simulator, such critera are the fact that the output voltages of each line of the output file should resemble neuron signals and that the types of the tokens of each line should match the expected ones. Neuroscientifically, it is safe to assume that neuron signals occur in the $[-100, 100]mV$ range, which was verified by finding the minimum and maximum values of previously available output files.

The aforementioned dependability threats require the use of a fault tolerance scheme, such as C/R, in order for the InfOli simulator to achieve progress in environments with error occurences. Additionally, the runtime environment needs to be equipped with error removal capabilities to avoid the infinite occurence of the same errors. This capability can be achieved through the use of platform-level countermeasures.

3.3 Available Countermeasures

There are several means available in the current experimental setup which can alter the platform and system state and potentially stop the reoccurence of errors. These actions, called countermeasures, can be performed in sequences in attempt to resolve the faults that caused the observed failures, before restoring the system and application state to a working condition.

As mentioned before, various control operations can be performed on the SCC board through the SCCKit framework on the MCPC. There are three such operations that can potentially resolve platform-related faults, based on the reliability threats already mentioned.

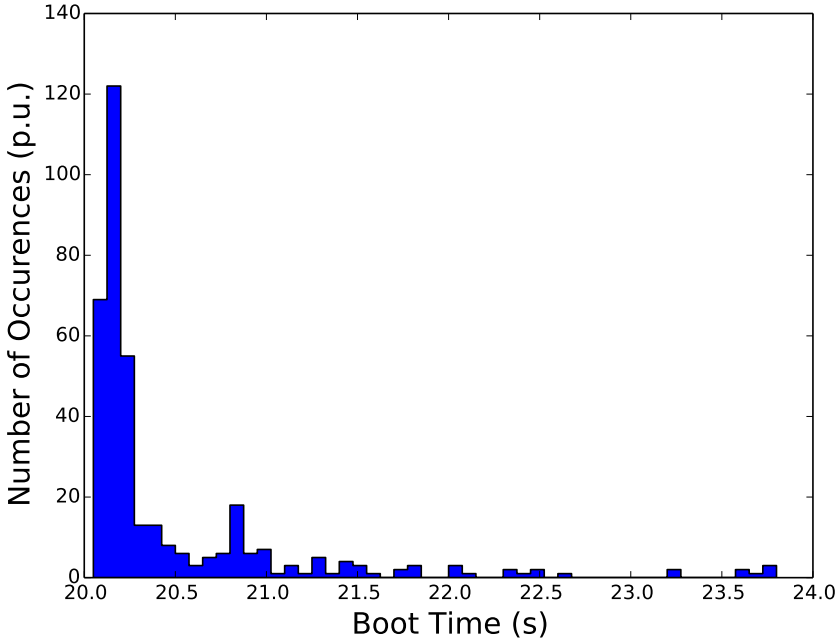


Figure 3.2: Time Distribution of the Boot Linux action

First of all, the SCCKit suite provides access to the *sccBoot* command.

Upon its execution, the specified cores of the SCC board are rebooted with a prespecified custom linux image, after it is first redistributed to the cores' private memories. This countermeasure proves useful in the event of temporarily unreachable cores, especially due to system errors, such as memory corruption or kernel panics. The image propagation generally takes 20 to 25 seconds, as it can be observed in the distribution of Figure 3.2 and the cores can be typically reached by the *ping* command in 5 to 10 seconds.

Another action which can be used as a countermeasure is the *sccReset* command. It enables access to each core's Reset switch, allowing us to pull or release them independently. This is a near-instant process which causes the core to reinitialize and flush its buffers. Even though the faults that this action resolves are very rare, its use introduces minimal overhead and ensures the correct behavior of the *sccBoot* command. Therefore, in the context of the discussed error removal scheme, the *sccReset* command will always be performed before the *sccBoot* command.

Finally, the *sccBmc* command allows use of the board management controller. The most relevant feature in this case is the reinitialization of the SCC platform, which reprograms the device's clock settings and trains the physical system interface. This action may take more than 3 minutes to complete, and therefore should be used only as a last resort. The exact time distribution of the platform reinitialization feature can be observed at Figure 3.3.

3.4 Checkpoint/Restart Implementation for the InfOli simulator

3.4.1 Introduction

In order to provide fault tolerance to the InfOli simulator, a C/R scheme was implemented for both available Porting Options. As mentioned in the previous chapter, most available distributed C/R tools, such as C^3 and $BLCR$ utilize the MPI library to coordinate between the available nodes [5, 6, 1, 21]. As a result, none of the available preexisting tools can be used in the case of the RCCE library and the SCC.

Therefore, it was necessary to implement a custom C/R method. As mentioned in the previous chapter, application-level techniques offer reduced

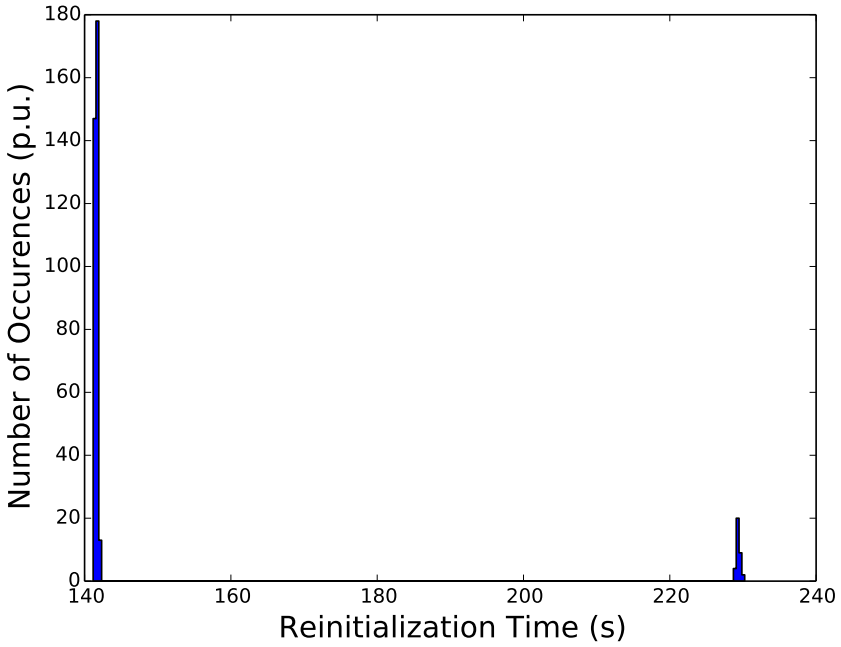


Figure 3.3: Time Distribution of the Platform Reinitialization action

checkpoint latency and checkpoint file sizes compared to their system-level counterparts. This is explained by the reduced storage requirements, caused by the selection of only the absolutely vital portions of the application state during the checkpoint procedure.

Harnessing this performance benefit, an application-level C/R technique was implemented for the InfOli simulator. The stored portions of the application state were determined through analysis of the InfOli simulator code base. These data structures represent the state of each individual simulated neuron cell, containing information of neuroscientific relevance such as current voltage or potassium levels for the axon, dendrite and soma compartments of the cell. Moreover, a local indexing variable used for the communication among neighboring cells also needed to be stored.

Furthermore, the InfOli simulator operates in simulation steps of equal workload, allowing the selection of a checkpoint interval in simulation steps rather than time. In the developed C/R implementation, checkpoints are taken at the beginning of the simulation steps that divide the Checkpoint Interval with no remainder.

The size of the output files is directly related to the *Cells per Core* metric, which indicates how many neuron cells are assigned to each core. Larger numbers of neuron cells require more storage space and increase the check-

point latency metric. The impact of the Cells per Core metric on the latency and interval metrics of the C/R implementation for a fixed checkpoint interval of 500 simulation steps can be observed at Figure 3.4. As expected, it can be observed that an increase in the number of Cells per Core corresponds to increased storage cost of the checkpoint files, resulting to a higher checkpoint latency, and increased processing time of the available cell inventory, corresponding to a higher checkpoint interval.

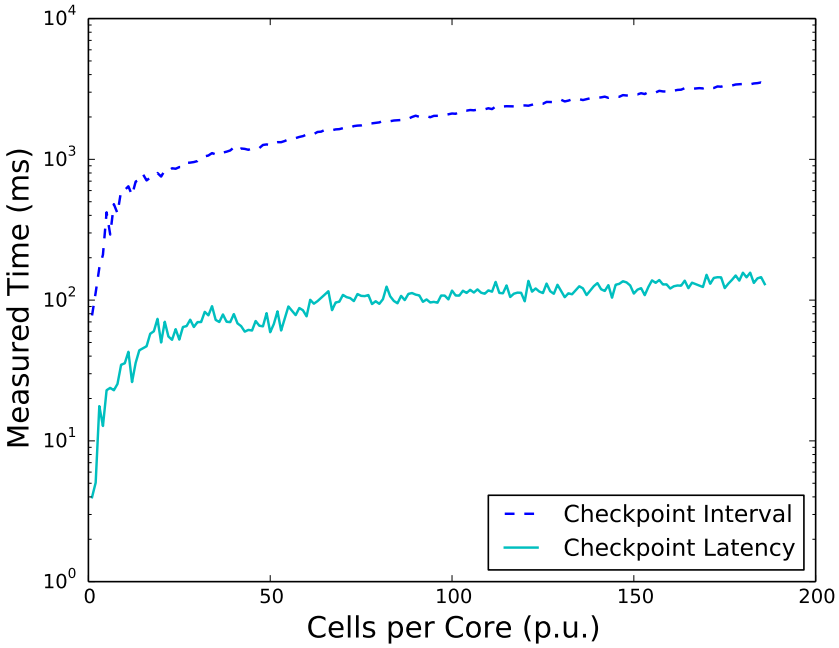


Figure 3.4: Checkpoint Interval and Latency for the C/R scheme of the InfOli simulator

3.4.2 Synchronization

In addition to storing the application state, the C/R implementation also needs to be able to provide valid output files despite the occurence of failures. This fact, along with the large I/O delays in the SCC platform, calls for manual synchronization of the output files before a checkpoint is written. This is achieved through the *fsync* system call, which ensures that the output of every simulation step before the checkpointed one has been successfully stored to the output files.

Through this process, it was observed that periodic synchronization of the output files introduced performance benefits for the InfOli application.

More specifically, the application was previously synchronizing the output files during the simulation exit, forcing their entire size to be written simultaneously. This process quickly achieved full utilization of the PCIe bus, introducing time overhead.

In order to allow the exploration of the introduced performance benefits, the C/R implementation allows the synchronization of the output files to occur at a different interval than the checkpoint files, called the *Sync Interval*. In order to ensure that the output files are synchronized before each checkpoint, the Sync Interval should be a divisor of the Checkpoint Interval. For the purposes of this thesis, the Sync Interval will be assumed to be equal to 100 simulation steps, and all Checkpoint Intervals will be multiples of this number.

In the developed C/R implementation, synchronization of the checkpoint files between all nodes is achieved through blocking communication and double bufferring. More specifically, a barrier function is called when a checkpoint needs to be taken, forcing all nodes to take the checkpoint together. Even though blocking communication introduces more overhead than non-blocking techniques, the InfOli simulator already utilized blocking techniques among all cores during each simulation step. Moreover, checkpoints are stored in a double buffered file, containing two sequential checkpoints at any time. This technique ensures the existence of a valid checkpoint among all nodes, even in the event of errors occurring during the checkpointing process.

3.4.3 Restart Procedure

The main task of the Restart procedure is to extract the neuron state data structures from the checkpoint files, perform the appropriate variable initializations and restore the simulator to the correct simulation step. If the number of running cores is different than that of the checkpointed run, the output files of the InfOli simulator are reconstructed to provide a consistent output. The time cost of the Restart operation in both cases, for a cell inventory of 2400 neurons, is presented at Figure 3.5. The output reconstruction measurements were taken for the scenario where the checkpoint file represents 48 running cores and the restart procedure is performed for 24 cores.

Since the checkpoint files are double buffered, a communication scheme was implemented to determine the maximum recoverable checkpoint for all the nodes. Specifically, each core opens the checkpoint file that contains

the cells required for this core. Afterwards, the maximum simulation step of the two available checkpoints is broadcasted to all the other nodes. By the end of this process, each node is aware of the simulation steps of the available checkpoints and individually determines the maximum simulation step available to every node.

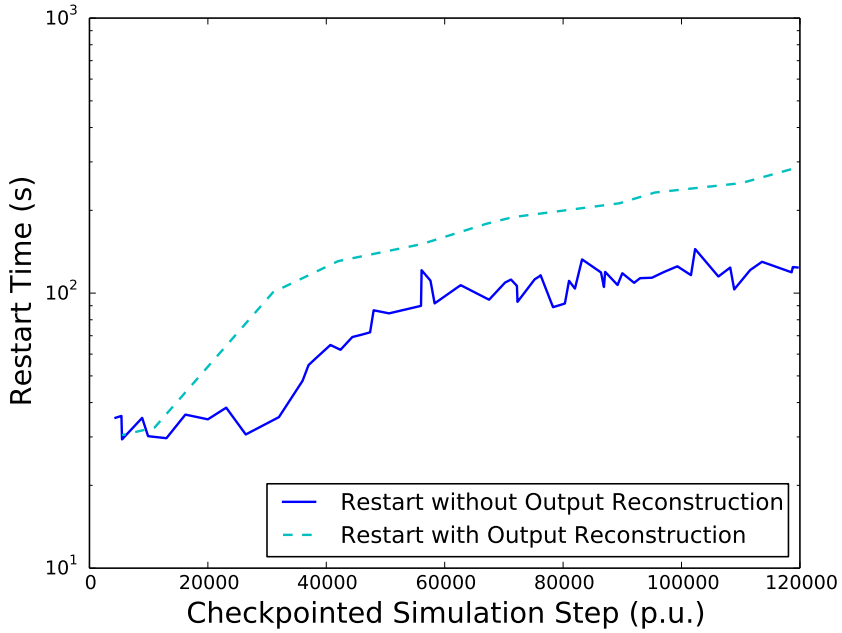


Figure 3.5: InfOli Restart Duration for 2400 Neuron Cells

Given that the desired fault tolerance scheme should be able to handle permanent failures of SCC cores, the C/R implementation needs to allow the InfOli simulator to be restarted at a different amount of cores than the previously checkpointed run. In this case, the consistency of the simulator output needs to be ensured in order to allow the C/R implementation to operate transparently. This is achieved through the output reconstruction process, enabling the application to resume operation in a subset of the available cores or on other platforms. The reconstruction process provides a consistent output when the number of execution units of the simulation is altered.

In order to achieve this feature, the current Cells per Core values need to be added to the checkpoint files. Therefore, when the restart procedure is initiated, every node reads the previous Cells per Core value from the first checkpoint file in order to determine which checkpoint files contain the required cells for this core. After opening them and restoring the application state, each core creates its new output file and populates it using the files

of the previous simulation.

For example, if the simulator is restarted at 24 cores, rather than 48, the Cells per Core metric is doubled and each core opens two of the 48 checkpoint files. Afterwards, each core merges the two corresponding output files in one, resulting in 24 total output files, containing the double number of cells each. The reverse process can also be performed, by splitting the contents of checkpoint and output files rather than merging them.

As it is apparent in Figure 3.5, the reconstruction of the output greatly increases the cost of the restart operation, as the manipulation of the output files is bounded by the performance of the platform's PCIe bus. Furthermore, it can be observed that the Restart Time increases in relation to the checkpointed simulation step, regardless of the use of output reconstruction. This is expected, as the required handling of larger output files introduces an increased I/O cost during the restart operation.

3.5 Injection Campaign

In order to test and benchmark the target experimental setup, the injection of errors on the runtime system is required. The developed injection campaign introduces different types of DUE and SDC errors at user-specifed MTTF intervals, called MTTF_S, where the "s" stands for specified . These intervals are defined as vectors of time and MTTF tuples, allowing the emulation time-dependent behavior of the system's MTTF parameter. Additionally, the injection of each error type is configured with different MTTF_S vectors. The injection campaign is implemented as an independent module of the Depman Tool, which will be further discussed in the next chapter.

The injection of errors at the specified vectors is performed through a Monte Carlo simulation. The injector is invoked at regular time intervals, Δt, during the execution of the InfOli simulator and its restart procedure. The probability of error occurence at each interval is given by equation 3.1, which corresponds to a Weibull distribution of TTF values. The emulation of the specified MTTF_S vectors is demonstrated in Figure 3.6, where the times between 2800 injections are grouped together in bins of $1s$. The measured MTTF value of $10.4s$ successfully matches the specification of $10s$.

$$P_\text{S} = 1 - e^{-\Delta t/\text{MTTF}_\text{S}} \tag{3.1}$$

The implementation of the actual error occurences is performed at the target platform, when possible, using commands of the SCCKit. However, permanent failures and SDC occurences cannot be emulated through the framework. As a result, these errors are injected by altering the input of the corresponding error detectors, causing them to fail. This data manipulation between the SCC and the error detectors is structurally performed at the *Injection Layer* of the runtime environment.

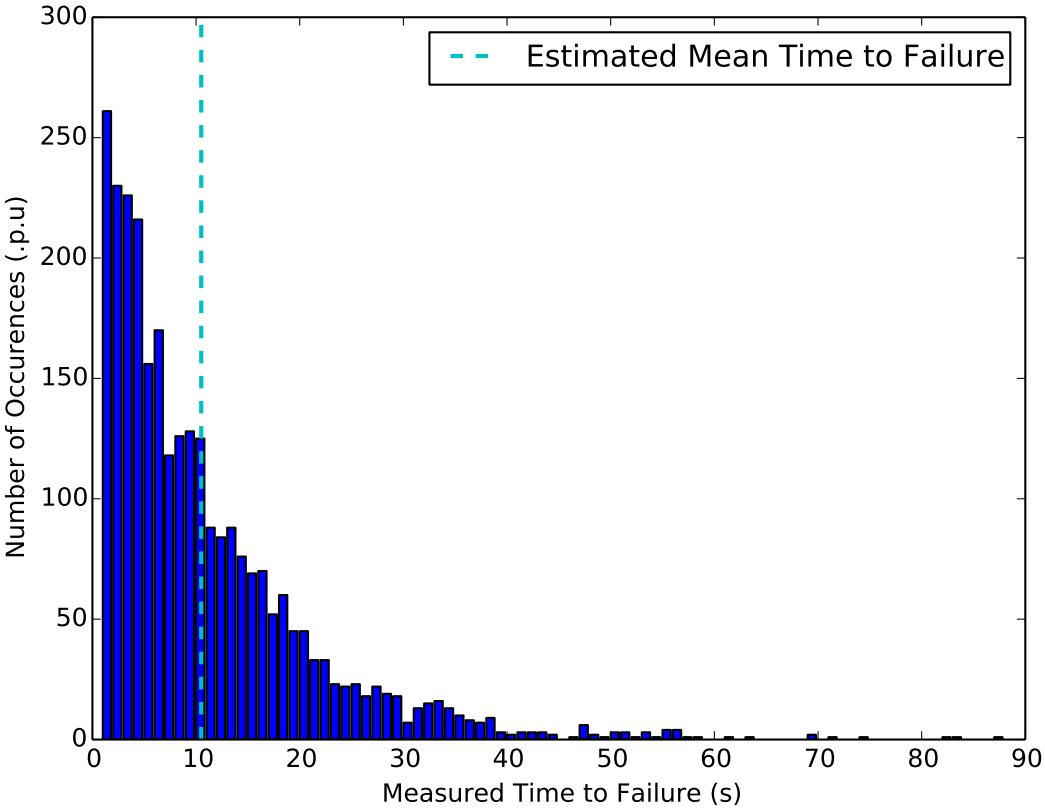


Figure 3.6: TTF Measurements for an Injected $MTTF_S$ of 10 s

Chapter 4

Depman Tool

4.1 Introduction

Depman is a runtime manager that controls the operation of a checkpointed scientific application, the InfOli simulator, on the Single-Chip Cloud Computer. It performs fault correction procedures, restarts the application in a thermal-aware fashion and offers a closed loop optimization of the parameters of the managed C/R system. The interactions of the Depman tool with the target system and application are depicted at Figure 4.1, while an abstract block interpretation can be found at Figure 4.2.

The operations of Depman are structurally organized in three groups of containers: *diagnostics*, which monitor the system for errors, *countermeasures* and the *core tool*. The core tool performs the main control sequence, utilizing the appropriate diagnostics and countermeasures to perform actions related to the current platform or application. It also measures the TTF of the system in order to estimate variations of the current $MTTF$ metric and use this knowledge for checkpoint interval optimization. The latter feature is performed through a moving average estimator and it will be further discussed in the next chapter. A simplified UML class diagram of Depman is depicted at Figure 4.3.

Diagnostics are parallel monitoring tasks that follow the current state of the application and platform for specific errors and define how to resolve them. As mentioned in the previous chapter, countermeasures are the available actions that can be taken by the system in order to resolve platform and application errors. Each diagnostic corresponds to a specific type of error and defines a sequence of countermeasures called a *countermeasure pro-*

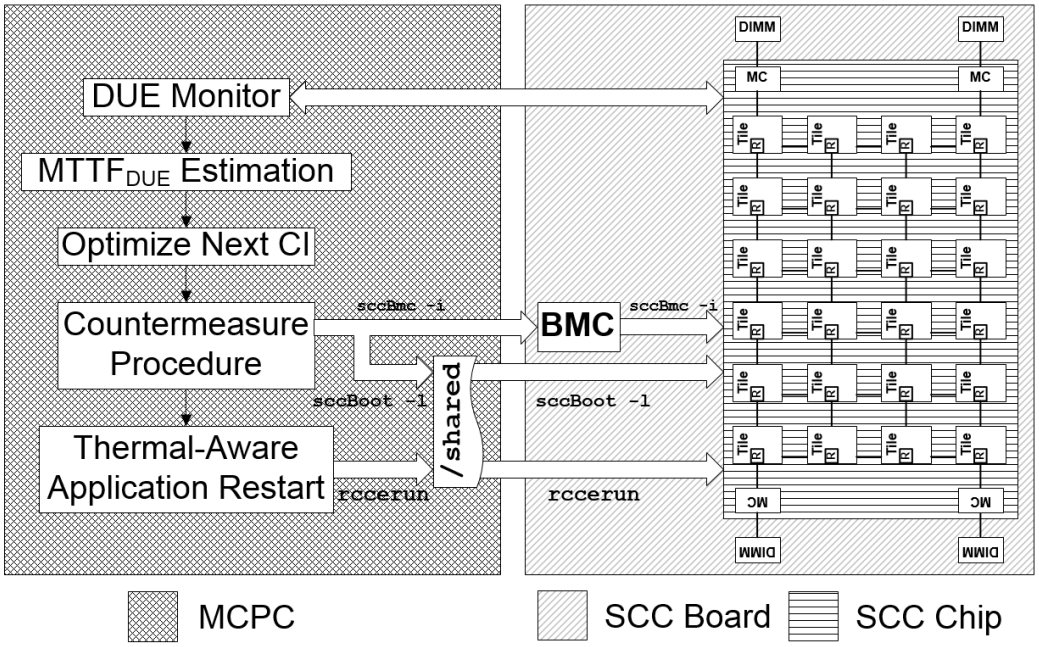


Figure 4.1: Flow Diagram of Depman's Closed Loop Operation for DUE Occurences

cedure that, when executed in order, can potentially resolve the faults that caused them.

Depman also includes an error injection module that follows the specification mentioned in the previous chapter to introduce errors at predetermined MTTF vectors through a Monte Carlo simulation. The injection module has been used in performance measurements of configurations. The module enables multiple errors to be injected at different MTTF vectors, triggering different diagnostics. Each diagnostic can also be triggered by different injectors, such as the injectors of permanent and temporary failures.

Depman was implemented in Python 2.7, using an object oriented design to allow for easy portability to other target platforms and applications. In this design, the core tool is implemented as the *depman* class, which is highly configurable and contains little application and platform dependencies. The source code is licenced under the GPLv3 and it is included in the Appendix.

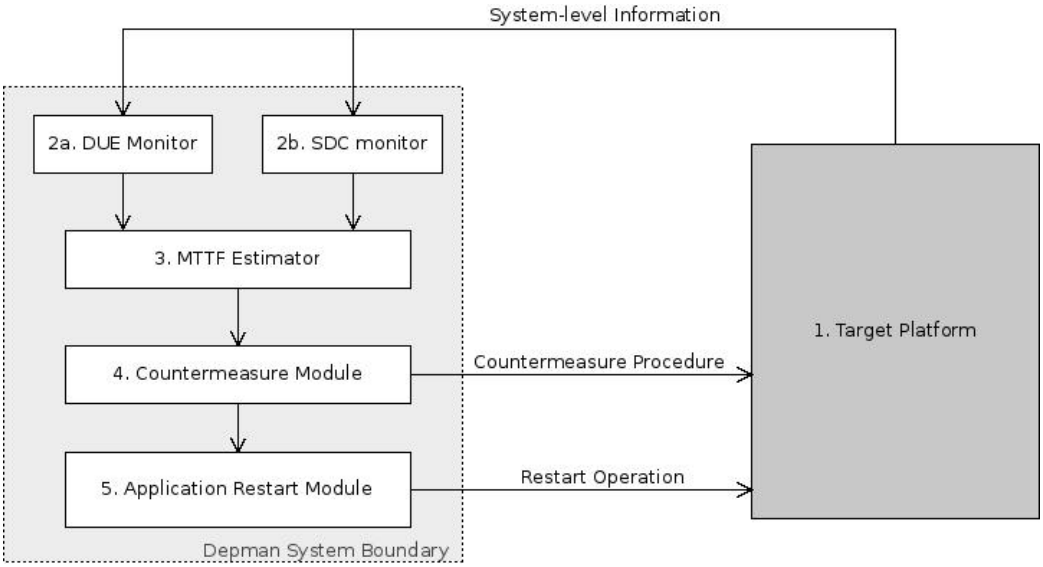


Figure 4.2: Block Diagram of the Depman tool

4.2 Core Tool

The core manager tool spawns the initial application processes, waits for it to exit and then performs the system repair routine by selecting and executing a countermeasure procedure. It estimates the system's reliability metrics during this process, such as the MTTF and MTTR parameters. It then calculates a thermal-aware task allocation, if required, and restarts the application. The tool implemented as the master thread of the depman process. It forks the `rccerun` executable and waits for its termination. The Depman tool terminates when the forked process has returned no error value, no diagnostics have failed since the last repair and no diagnostics have incomplete workloads.

The actions of the core tool after initialization are performed in the *event loop*, which is depicted in pseudocode at Figure 4.4. Initially, the main thread waits for all diagnostic threads to stop their current operation. Diagnostics that still require computational time after the application exits, such as SDC detectors, are marked as incomplete and depman waits for them if the application process has completed.

The latest checkpoint and output files of the application are then validated and stored at a predetermined filesystem location, called the *safe location*. The selection of a checkpoint from the safe location is performed during application restart and it is different for DUE and SDC diagnostics. The

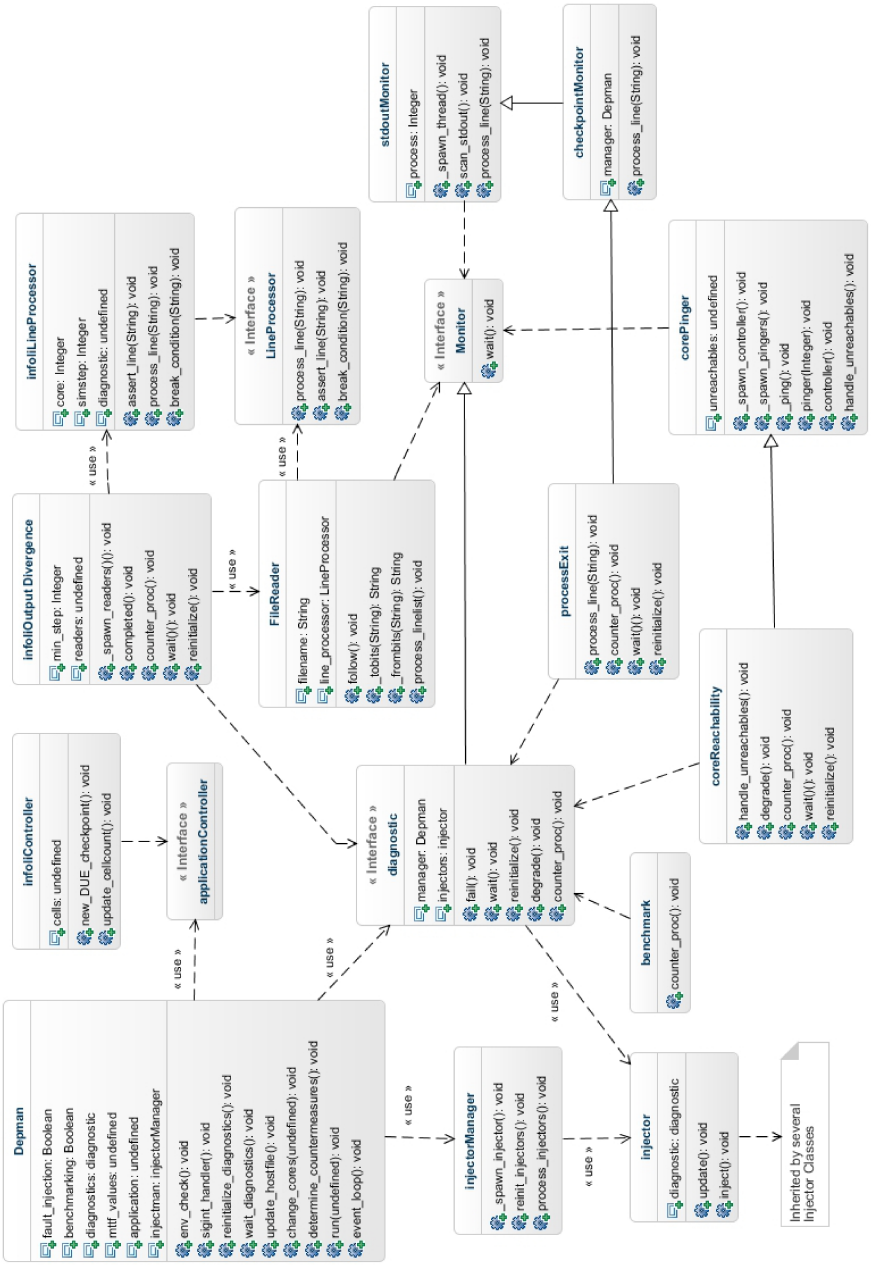


Figure 4.3: Class Diagram of the Depman Tool

```
def event_loop():
    wait(application_process)
    wait(diagnostics)

    repair_timestamp = time()

    if length(failed_diagnostics) == 0:
        if length(incomplete_diagnostics) > 0:
            wait_for_completion(incomplete_diagnostics)
        else:
            exit_depman()

    mttf_estimate.add_value(time() - failure_timestamp)

    new_checkpoint_interval = optimal_CI(mttf_estimate, latency)

    new_DUE_ckpt = look_for_new_checkpoints()
    if not new_DUE_ckpt and countermeasure_procedure is None:
        degrade_diagnostics()
    else if new_DUE_ckpt:
        determine_countermeasure_procedure()

    perform(countermeasure_procedure.pop())
    mttr_estimate.add_value(time() - repair_timestamp)

    reinitialize_diagnostics()

    if injecting_errors:
        reinitialize_injectors()

    failure_timestamp = time()
    event_loop()
```

Figure 4.4: Pseudocode of the Event Loop

exact details of the checkpoint selection and management actions will be discussed in the related subsection.

Afterwards, the measured Time to Failure is added to the set of previous TTF values and a new optimum periodic checkpoint interval is determined based on the new MTTF estimate. The theory and methodology behind the estimation of MTTF and τ_{opt} will be examined in the next chapter.

Finally, the next step of the countermeasure procedure is determined and performed, the time to repair is measured and the injectors and diagnostics modules are reinitialized. It is also worth noting that the fork of the target application is performed by the master thread during initialization and when performing the related restart countermeasure.

4.3 Diagnostics & Countermeasures

4.3.1 Diagnostic Interface

In the previous section, it has been stated that diagnostics are the objects that are responsible for the detection of errors on the target system. Being used as an interface by the core tool, they can be either platform-specific, such as DUE detectors, or application-specific, such as SDC detectors.

Diagnostics contain routines that create and manage any monitoring threads they have, as well as a prespecified countermeasure procedure. The threads perform system monitoring actions such as file or stdout reading and sending IMCP echo requests to individual cores. Upon detection of a failure, the fail() method of the corresponding diagnostic object is called.

Diagnostic objects are implementations of the diagnostic abstract class, which specifies the concrete method fail() and the abstract methods wait() and reinitialize(). The fail() method causes a diagnostic to suspend the running application, through depman's stop() method, while holding the diagnostic lock.

The wait() method of each enabled diagnostic is called by the core tool directly after the application process exits. It is used to block the master thread until the diagnostic thread can finish its current task, determine whether it has failed and stop the monitoring process. This halt is required, as the old application process is terminated at this point in time and no errors can occur until it is restarted. Therefore, the diagnostics perform

cleanup actions during the wait() method and will be reinitialized when a new application process will be forked during the repair procedure.

After this waiting period, the master thread selects the most time-expensive countermeasure procedure of all failed diagnostics and performs the required sequence of countermeasures. More specifically, each countermeasure is enumerated in terms of its execution time and the most expensive sequence of actions in a countermeasure procedure is always located last. More details on this ordering and the exact form of countermeasure procedures are depicted in the related subsection.

When the repair sequence is finished and the application process is restarting, the core tool calls the reinitialize() method of each diagnostic. This method causes the diagnostic threads to reinitialize themselves and start monitoring the currently running application for errors. Based on this separation, we can conclude that depman considers that repair time errors can occur only during the restart procedure of the application and not during the selection and execution of other countermeasures.

The current design of the Depman tool allows diagnostics to inherit or utilize abstract classes called *monitors*, which provide default actions such as stdout piping, file following and core pinging. The existence of the monitor classes enables diagnostic classes to contain only the platform-specific or application-specific code segments, allowing for easier portability to other platforms and applications.

4.3.2 Implemented Diagnostics

The following diagnostics were be implemented for the experimental setup of the thesis, each corresponding to a unique error:

- ProcessExit, which monitors the stdout of the RCCErun process for failure messages indicating a DUE error.
- CoreReachability, which periodically polls the available cores with ICMP echo requests in order to detect unreachable cores.
- infoliOutputDivergence, which scans the validity of the InfOli simulator's output files, aiming to detect SDC occurences.

The ProcessExit diagnostic checks each line of the RCCErun process' standard output for the existence of the "FAILURE" substring. This substring exists in all error messages detected by the RCCE library, and it serves as a useful system-level diagnostic. Potential errors detected by the Pro-

cessExit diagnostic are memory allocation or access errors, file permission errors or application-specific errors that cause the application to exit, such as invalid initialization and configuration.

The wait() method of the ProcessExit diagnostic closes the pipe from the application process, while the reinitiliaze() method starts reading from the pipe of the new application process' output. The diagnostic is implemented as a subclass of the stdoutMonitor class, which provides all the initialization and stdout reading functionality.

The CoreReachability diagnostic periodically checks whether each running core is available through the unix tool `ping`. Upon detection of an unreachable core, the diagnostic fails. It is implemented as a subclass of the corePinger monitor, which utilizes a thread pool to concurrently ping the requested set of cores. More specifically, a *controller* thread inputs the name of each core to a Queue data structure and the *pinger* threads individually pop the head of the queue and ping the corresponding core. When the wait() method is called, the caller thread blocks until the Queue is empty and the controller thread does not reenter the core names in the queue, until the reinitialize() method is called.

The infoliOutputDivergence diagnostic checks for the occurence of SDCs in the output files of the InfOli simulator. In order to achieve this, every line of each output file is validated through several criteria, such as the total number of tokens, the type of each token and the existence of values in acceptable ranges. Its implementation uses a series of FileReader monitor objects, each of which follows a file for changes, splits it into lines and validates every line through the specifications of the diagnostic. The wait() method causes the file reader threads to close the files they are reading, while the reinitialize() method causes them to reopen the files and seek to the appropriate line. Being an SDC detector, the infoliOutputDivergence diagnostic may continue operating after the application has terminated, by being marked as incomplete until the entirety of the output files has been processed.

4.3.3 Countermeasure Procedures

Each diagnostic defines a series of sequences of countermeasures to be attempted as resolutions for the detected failures. As discussed in the previous chapter, the execution time of each countermeasure varies, contributing to the total *Time to Repair* of the system. Therefore, it is crucial that expensive countermeasures are considered only after the faster alternatives

have been already attempted. As a result, the general rule for defining such countermeasure procedures is that the total repair time of each individual sequence should be in an ascending order.

In the present experimental setup, we observed that the most expensive countermeasure is the platform reinitialization (*sccBmc -i*), followed by the Linux Boot process (*sccBoot -l*). The resulting countermeasure procedure for the infoliOutputDivergence and ProcessExit diagnostics is depicted at Figure 4.5. The procedure of the coreReachability diagnostic differs by not containing the first step, the single rccerun invocation.

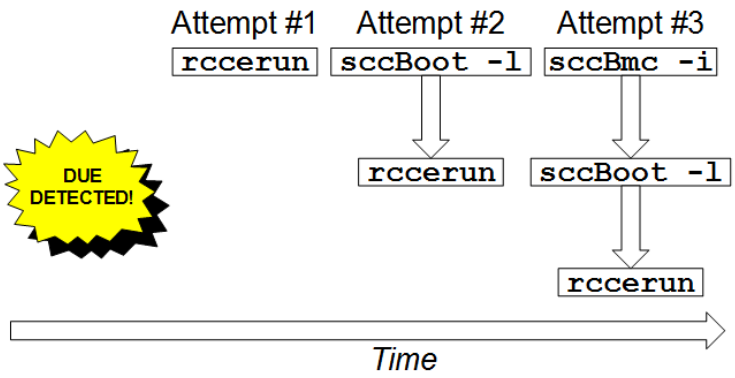


Figure 4.5: Countermeasure Procedure of the ProcessExit diagnostic

4.4 Thermal-Aware Task Reallocation

A possible scenario in the diagnostics & countermeasures model is the case where an error cannot be resolved by any of the available countermeasures, leading to an empty countermeasure procedure. The standard policy of the depman tool is to define a new countermeasure procedure and continue with the error correction tasks . However, this new countermeasure procedure does not ensure the resolution of the underlying fault, as it will be identical to the one just attempted. For this reason, diagnostics can implement a *degrade* function, which is called when the countermeasure procedure is emptied, in an attempt to change parameters of the platform or application that can potentially resolve the recurring fault.

This treatment is applicable in the present experimental setup through the coreReachability diagnostic. When a set of cores is consistently detected as unreachable, even after the entire countermeasure procedure is exhausted,

this set of cores can be considered as permanently defunct. In this case, the degrade function changes the number of execution units that the target application will be executed at, excluding problematic cores. However, the decision of how the remaining tasks will be allocated on the available cores of the chip still needs to be made.

Many-core applications often require the distribution of homogenous workloads among the running cores, especially when blocking communication is used. This happens to be the case with the InfOli simulator, which requires that the total number of simulated cells needs to be a multiple of the running number of cores. For example, if the initial number of cores used is 48 when one permanent core failure occurs, the application can operate on 24 cores, which is the largest possible divisor of 48. Therefore, in homogenous applications such as the InfOli simulator, a permanent failure of one core requires the next restart operation to exclude a portion of the operational cores.

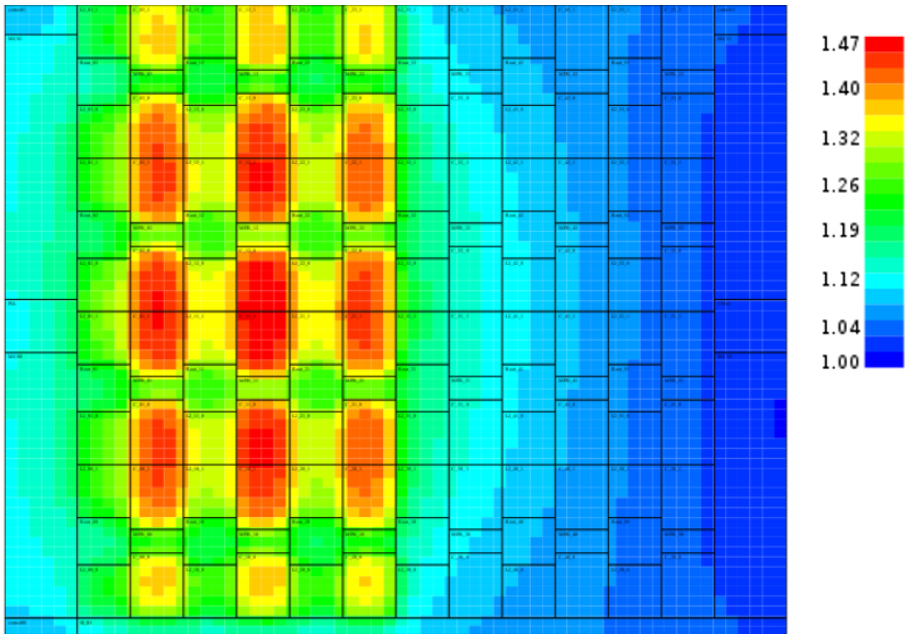
An important dependability concern of task allocations on many-core systems is the thermal homogeneity of the platform. Inhomogenous task distribution causes some areas of the chip to develop hotspots, while other areas remain cooler. As the SCC is a homogenous chip of 48 cores, thermal inhomogeneity can cause some areas of the chip to deteriorate faster than others, resulting to variabilities in the mean time to failure of the chip's individual components.

In order to reduce the effect of inhomogenous task distribution, a greedy thermal-aware task allocation algorithm was developed as the degrade function of the coreReachability diagnostic. The algorithm uses as heuristics the manhattan distance to the closest running core and the distance to the edge of the chip to provide an evenly spaced distribution of tasks on the chip. Even though the algorithm does not guarantee optimality, it provides significant improvements to naive task allocation techniques, such as using only one side of the chip, as presented in Figure 4.7. Additionally, even though permanently defunct cores are not considered for task placement by the algorithm, their thermal conductivity is taken into account as part of manhattan distance calculations.

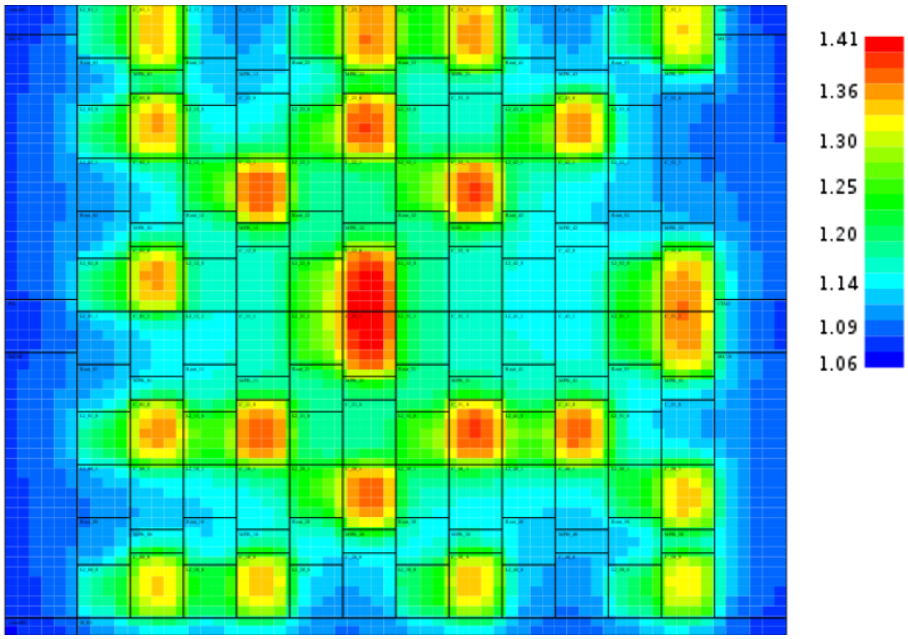
The pseudocode for this algorithm is presented in Figure 4.6 and the steady state temperature modelling simulations of Figure 4.7 were performed using version 5.0 of the HotSpot Temperature Modeling Tool [47] and an existing floorplan of the SCC chip [41]. The temperatures on the figure are indicated in a relative scale where each temperature is divided by the minimum observed temperature of both figures.

```
def SCC_task_allocation(tasks, allowed_cores):
    Grid = new Matrix[8][6]
    for core not in allowed_cores:
        Grid[core] = -1

    x,y = random_available_edge_cell(Grid)

    while tasks > 0:
        Grid[x][y] = -2
        tasks--
        for i,j in (8,6):
            if Grid[i][j] > 0:
                r = manhattan_distance((x,y), (i,j))
                if r < Grid[i][j]:
                    Grid[i][j] = r
                else if (r - 1 < Grid[i][j] < r):
                    Grid[i][j] -= 0.01

        max_list = get_all_max_values(Grid)
        min_list = get_all_minimum_distances_to_edge(max_list)
        x, y = select_random(min_list)

    return core_names_from_coordinates(Grid.find(-2))
```

Figure 4.6: Pseudocode of the Thermal-Aware Task Allocation Algorithm



(a) Naive Task Allocation



(b) Thermal-Aware Task Allocation

Figure 4.7: Temperature Modeling of 24 Running Cores on the SCC

By comparing the thermal modelling images of the two task allocation schemes, it can be observed that the proposed algorithm provides a more even heat distribution on the chip. It is also expected that the thermal gradient of the chip will be less steep, due to this even distribution. Additionally, the maximum temperature on the chip is also reduced from 1.47 units of relative temperature at the naive allocation scheme to 1.41 units at the proposed algorithm.

4.5 Distinctions of SDC and DUE Checkpoints

In contrast to DUEs, the system continues execution after the occurence of SDCs. Therefore, it is possible that checkpoints will be taken before the error is detected. These errors, called *latent errors* require a special C/R technique called multi-version checkpointing in order to ensure that each possible error is optimally recoverable. However, Depman performs validation and storage of checkpoint files only during error detection, rendering multi-version checkpointing as unusable. Therefore, a similar suboptimal technique has been developed, for use in environments with both DUE and SDC occurences.

When an error occurs, the checkpoint files are stored in the safe location and marked by the maximum recoverable simulation step they contain. This leads to a series of checkpoints, sorted in ascending order of simulation steps, one of which needs to be selected for the next restart procedure. Moreover, when an SDC detector diagnostic is operational, it keeps track of the maximum simulation step that has been validated on all output files, called *min_step*, as it is the minimum of the maximum steps of all files.

Upon DUE occurence, the checkpoint corresponding to the maximum available simulation step will be used. Additionally, any checkpoints for simulation steps less than the min_step are discarded, except for the largest one of them. This procedure ensures that the set of checkpoints will be kept as small as possible, while at the same time preserving the best available checkpoint for SDC occurences.

Upon SDC detection, the checkpoint with the largest available simulation step less than min_step will be used. Furthermore, all other checkpoints will be discarded. More specifically, checkpoints corresponding to smaller simulation steps are not needed, as they are covered by the selected checkpoint. On the other hand, checkpoints corresponding to larger simulation

steps cannot be used as they reflect a system state where the observed fault has been introduced.

This handling of stored checkpoints ensures the optimal recovery of DUE occurences, in terms of Repair Time and Rollback Time. However, in environments with only SDC occurences, it will fail to make progress, as no checkpoints will be available when the SDC is detected. Therefore, it is practically useful in systems where SDCs are expected to occur much less frequently than DUEs, allowing the storage of additional checkpoints.

Chapter 5

Checkpoint Interval Optimization

5.1 Introduction

Depman features a closed loop optimization scheme over the C/R implementation of the target application. This is substantiated through the minimization of the total *waste time* of the system, which is the time overhead introduced by the C/R implementation. The total waste time can be defined as the sum of the waste times W_i of each of the N total checkpointing cycles, as described in Equation 5.1.

$$W = \sum_{i=1}^{N} W_i \tag{5.1}$$

Moreover, it is assumed that the C/R implementation on the target application is periodic. Therefore, the calculated checkpoint intervals will be optimal only for Poisson failure distributions, as suggested by Ling et al. [27]. In periodic C/R schemes, each of the W_i times consists of the total time spent on checkpoint storage, the checkpoint latencies (ℓ), and the rollback time (T_r) of that cycle. The waste time of one cycle is modelled in Equation 5.2, assuming k_i checkpoints are taken at the i-th checkpoint cycle.

$$W_i = k_i \times \ell + T_r \tag{5.2}$$

Additionally, the checkpoint interval τ is analogous to k_i for each cycle and

the rollback time T_r can be determined by the TTF of the i-th cycle, TTF_i and τ. As a result, knowledge of the TTF_i of a checkpointing cycle can lead to the calculation of an optimal checkpoint interval for that cycle.

Based on this motivation, Depman calculates an optimal checkpoint interval for the next cycle as part of the repair process. This operation utilizes the MTTF of the current failure distribution of the system, which is estimated using a moving average filter over measurements of the TTF of previous cycles.

In the context of this chapter, we will consider only the cases of DUE occurences for the related measurements, leaving out the injection and detection of SDC occurences from our model. As mentioned in the previous chapter, Depman cannot guarantee optimal recovery after SDC occurences, as their delayed detection requires the use of multi-version checkpointing. Therefore, in the case of SDCs the rollback time presents significant variability and the described optimization methods are not expected to perform optimally. However, the implementation of the depman tool still performs the same phases of estimation and optimization when SDCs are detected, even though they have been designed to target DUEs.

5.2 Optimal Checkpoint Interval

The optimization of the Checkpoint Interval is performed by the Depman tool in three stages: *benchmarking*, *selection* and *transformation to simulation steps*.

The benchmarking phase is performed at the first execution of the target application on the current set of cores. During this phase, the application reports time measurements for the checkpoint latency and checkpoint interval on its standard output. These measurements are retrieved by Depman and averaged, for use in the other optimization phases.

The selection phase determines an optimal checkpoint interval for the next checkpointing cycle in time units. This selection is performed by attempting to minimize W_i for the next cycle, using the estimate of MTTF_{DUE}. The errors are assumed to occur, on average, in the middle of the last sequence of interval and latency. The rollback time can be calculated through this assumption by Equation 5.3. Additionally, the number of checkpoints taken in the i-th checkpointing cycle can be calculated, on average, through Equation 5.4.

$$T_r = \frac{\tau + \ell}{2} \tag{5.3}$$

$$k_i = \frac{\text{MTTF}_i}{\tau + \ell} \tag{5.4}$$

By substituting Equations 5.3 and 5.4 on Equation 5.2, we can calculate the total waste time of the i-th checkpointing cycle using only the MTTF, τ and ℓ parameters. The final result is presented in Equation 5.5.

$$W_i = \frac{\text{MTTF}_i \times \ell}{\tau + \ell} + \frac{\tau + \ell}{2} \tag{5.5}$$

By demanding the first derivative by τ of Equation 5.5 to be zero and solving for τ, we can calculate the checkpoint interval τ_{opt} that minimizes the waste time for that cycle. The final result, presented in Equation 5.6, is the same as the one derived from Daly's model, who used a similar approach to calculate τ_{opt} for all checkpointing cycles, making the same assumptions on the distribution of failures as our model [11].

$$\tau_{opt} = \sqrt{2 \times \text{MTTF}_{\text{DUE}} \times \ell} - \ell \tag{5.6}$$

Once τ_{opt} is calculated in time units, it has to be transformed to simulation steps. This is achieved by using the measured checkpoint interval of the benchmarking phase in time units, τ_{bench}, and the specified checkpoint interval in simulation steps, τ_{steps}. The transformation is performed using the relationships of Equation 5.7. The final value τ'_{opt} is then rounded to the closest multiple of the sync interval, in order to preserve the performance benefit of regular output file synchronization, as discussed in section 3.4.2

$$\tau'_{opt} = \frac{\tau_{opt}}{\tau_{bench}} \times \tau_{steps} \tag{5.7}$$

5.3 Moving Average Estimation

The estimation of the MTTF value used for the optimization of the next cycle is performed through a moving average filter of the measured TTF values from the last M cycles. The estimation of the injected MTTF for the i-th cycle (MTTF_{DUE}) is depicted in Equation 5.8. The selection of

M is performed at design time and it can greatly affect the efficiency of the managed C/R scheme. In the implementation of the Depman tool, the moving average filter was constructed as a double-ended queue with a fixed length of M.

$$\text{MTTF}_{\text{DUE}} = \sum_{i=1}^{N-1} \frac{TTF_i}{M} \tag{5.8}$$

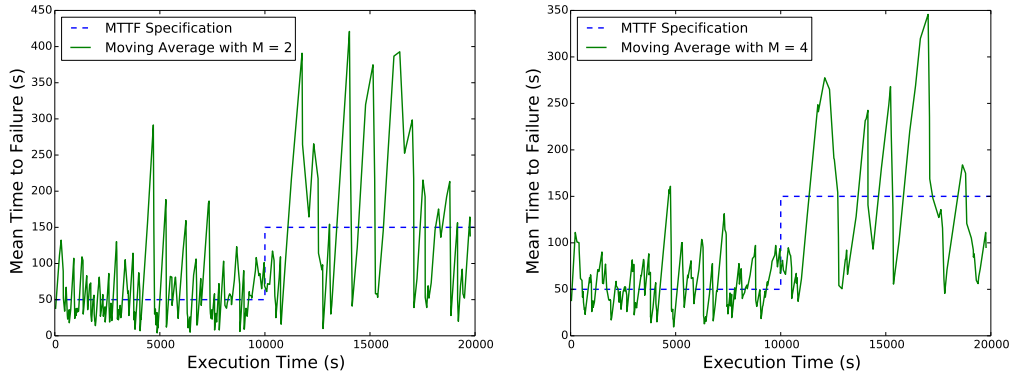
The step responses of the moving average filter for rising and falling MTTF steps at various values of M are available at Figures 5.1 and **??**. The measurements were taken by using the moving average filter and the injection module discussed in Chapter 2. We consider that the injections occur at Weibull distributions of mean time MTTF_{DUE}.

As it can be observed in the figure, larger values of M provide a more accurate steady state convergence to the actual MTTF of the system. Moreover, the amplitude of the MTTF_{DUE} step affects the effectiveness of the estimator, leading to a faster convergence for larger values. Finally, we can observe that rising steps of MTTF_{DUE} lead to more seldom updates of the monitor. This is caused by the fact that the estimation of MTTF_{DUE} is performed upon error detection and it is, therefore, directly affected by the actual injected TTF values. This is a preferable design choice, as the credibility of the estimator is improved when the operating conditions of the system become more hostile, causing the MTTF_{DUE} to decrease. In general, we can observe that increases in the length of the moving average filter produce more credible estimations of MTTF_{DUE}, which favors software implementations.

5.4 Efficiency Evaluation

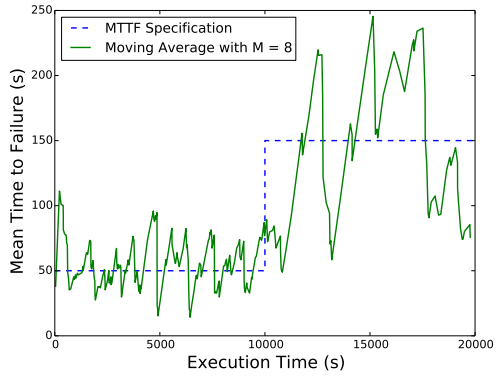
In order to evaluate the efficiency of the presented optimization scheme, two metrics are presented: performance efficiency and energy efficiency. These metrics have been evaluated for both the cases of constant and time-varying MTTF_{DUE} injections on the experimental setup.

The performance efficiency metric is defined as the ratio of the execution time of the target application with no C/R technique or fault injection to the execution time of the application with the proposed fault injection and C/R schemes. Similarly, the energy efficiency is defined as the ratio of the total used energy with no fault injection to the total energy with fault injection and C/R. The two efficiency metrics are depicted in Equations 5.9

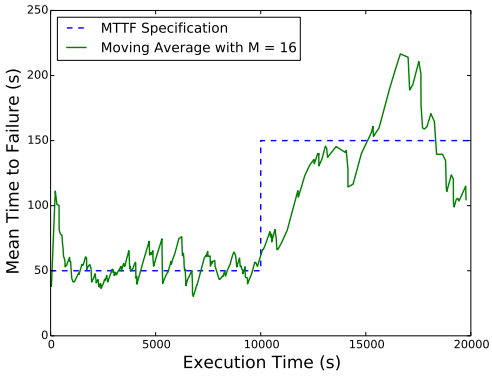


(a) $M = 2$

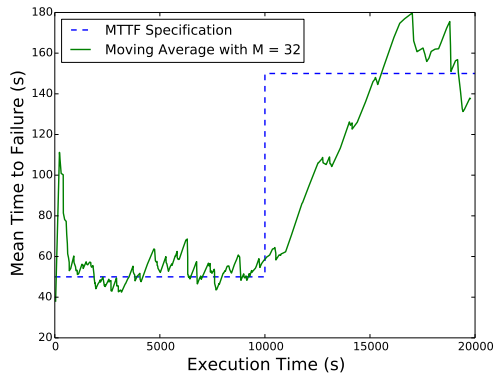
(b) $M = 4$



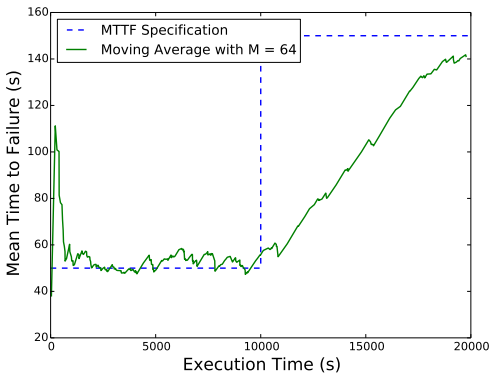
(c) $M = 8$



(d) $M = 16$

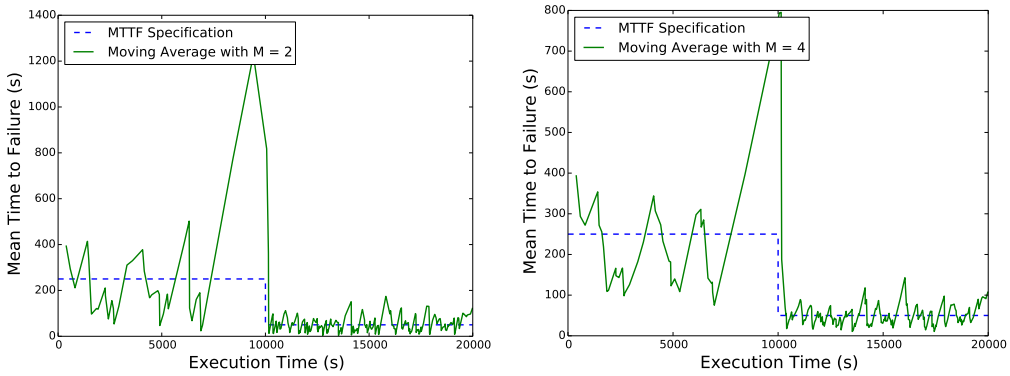


(e) $M = 32$



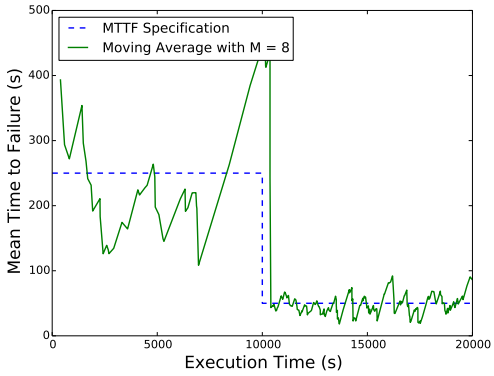
(f) $M = 64$

Figure 5.1: Rising Step Response of the MTTF Estimation

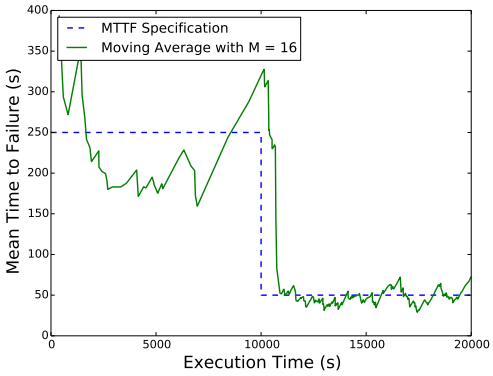


(a) $M = 2$

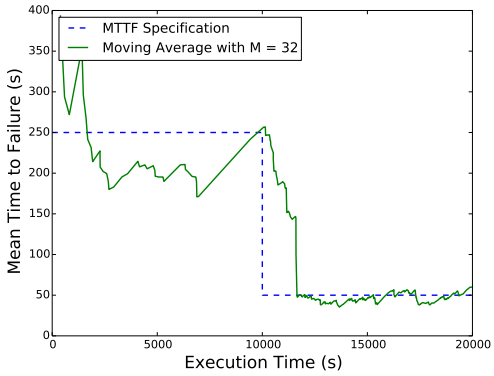
(b) $M = 4$



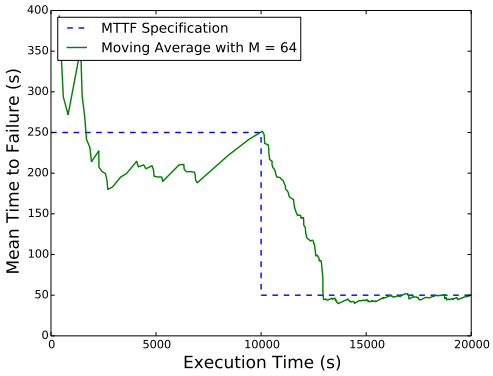
(c) $M = 8$



(d) $M = 16$



(e) $M = 32$



(f) $M = 64$

Figure 5.2: Falling Step Response of the MTTF Estimation

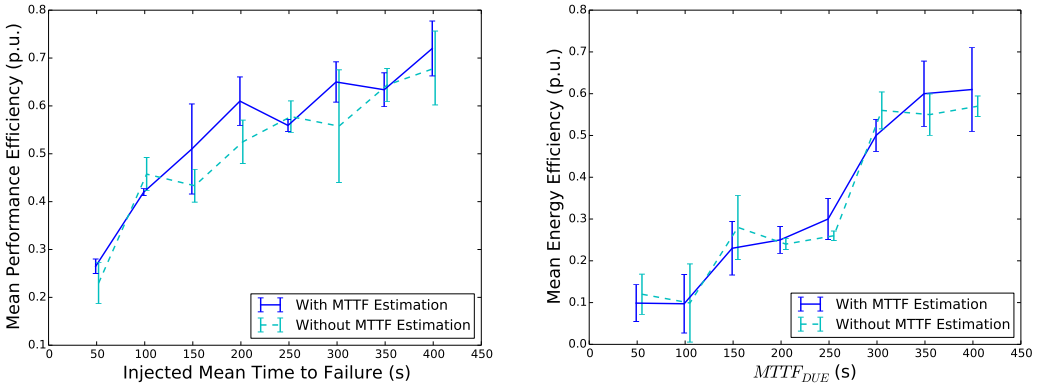


Figure 5.3: Efficiency Evaluation for a constant Injected DUE Rate.

and 5.10. In the experiments presented in this section, we will assume an InfOli simulation of the following parameters: 50 neuron cells per core, 6 seconds of total simulated brain activity, corresponding to 120000 simulation steps, and a naive 8-way interconnectivity scheme between the plane of neurons.

$$P_{eff} = \frac{T_{golden}}{T_{C/R}} \tag{5.9}$$

$$E_{eff} = \frac{E_{golden}}{E_{C/R}} \tag{5.10}$$

5.4.1 Constant $\mathrm{MTTF_{DUE}}$

Assuming a constant MTTF of injected errors, the performance and energy efficiencies of two cases were calculated. The first case assumes that the injected $\mathrm{MTTF_{DUE}}$ is known in advance by the optimization module. The second case utilizes the discussed moving average estimation technique.

The results for both cases are presented in Figure 5.3 for a moving average filter of $M = 32$. We can observe that the efficiency of the optimization scheme is not affected by the use of the moving average filter.

5.4.2 Time-varying $\mathrm{MTTF}_{\mathrm{DUE}}$

In this set of experiments, the efficiency of the rising and falling $\mathrm{MTTF}_{\mathrm{DUE}}$ edges of Figure 5.4 were calculated. The error bars represent a 95% confidence interval of the mean value, calculated by multiplying the standard error of Equation 5.11 by 1.96 and using the result to define the lower and upper bounds of the confidence interval [33]. In Equation 5.11, N represents the total number of measurements used for the estimation, which in the present experiment was 20.

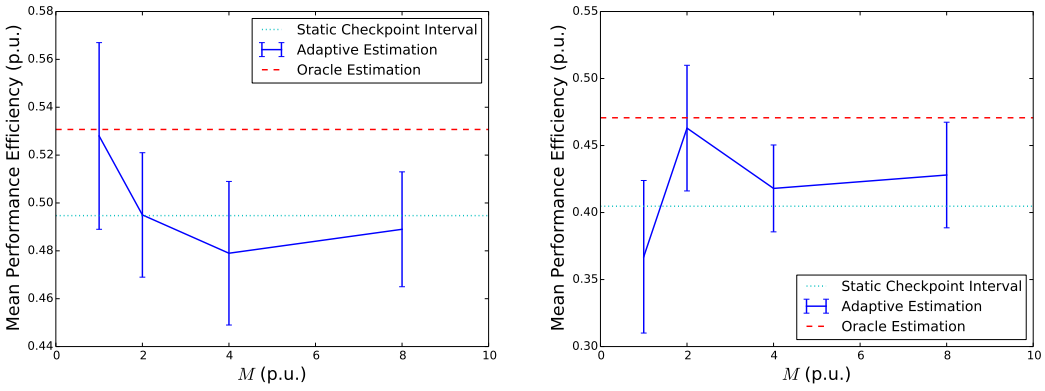
$$\text{standard error} = \frac{\text{standard deviation s}}{N} \tag{5.11}$$

The selection of τ_{opt} was performed in accordance to the technique described in section 5.2 and three different hypotheses have been explored regarding the $\mathrm{MTTF}_{\mathrm{DUE}}$ estimation:

- **Oracle Estimation:** In this case, we assume that the estimation is performed by an *oracle* predictor. The oracle predictor is always aware of the specified $\mathrm{MTTF}_{\mathrm{DUE}}$ of the injector, using it for checkpoint interval optimization. This case effectively provides an upper bound of efficiency, assuming on-the-fly estimation of $\mathrm{MTTF}_{\mathrm{DUE}}$.
- **Static Estimation:** In this case, we assume that the $\mathrm{MTTF}_{\mathrm{DUE}}$ estimation used for CI optimization is always constant and equal to $50s$, even if in reality the injected $\mathrm{MTTF}_{\mathrm{DUE}}$ is time-dependent. This provides a lower bound of efficiency for the discussed optimization scheme.
- **Adaptive Estimation:** This method corresponds to the discussed optimization scheme of this chapter, featuring the moving average estimator.

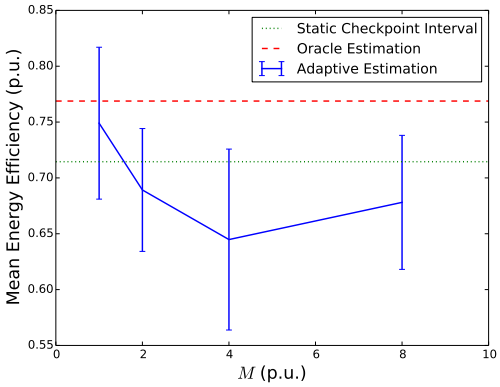
The efficiency values for various values of M are presented in Figure 5.4. From these results, we can determine that knowledge of the current $\mathrm{MTTF}_{\mathrm{DUE}}$ leads to improved efficiency of the C/R optimization scheme, as it is apparent from the difference between the oracle and static estimations.

Furthermore, we can determine that rising $\mathrm{MTTF}_{\mathrm{DUE}}$ edges do not necessarily lead to improvements in the related efficiency metrics. This is speculated to be caused due to several reasons, such as increased moving average error and the fact that performance benefits in rising steps are caused by the reduced total time spent as checkpoint latencies, which is a very small portion of the total execution time to confidently provide an efficiency benefit.

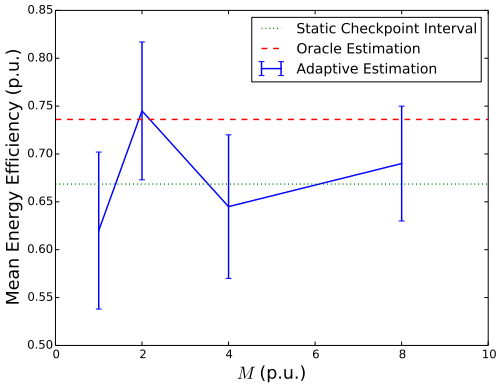


(a) Rising MTTF_{DUE} edge

(b) Falling MTTF_{DUE} edge



(c) Rising MTTF_{DUE} edge



(d) Falling MTTF_{DUE} edge

Figure 5.4: Efficiency Evaluation of `Depman` for Two Cases of Time-Varying MTTF_{DUE}

Finally, in the cases of falling $\mathrm{MTTF}_{\mathrm{DUE}}$ edges, we can observe improvements in the efficiency of the C/R optimization scheme. This corresponds to the event where errors are more frequently introduced to the system. As a result, the C/R scheme benefits from reduced rollback times when the checkpoint intervals are optimally selected. Moreover, the reduction in $\mathrm{MTTF}_{\mathrm{DUE}}$ leads to a faster convergence of the moving average filter to that value, as discussed in the previous sections.

Chapter 6

Conclusions

6.1 General Remarks

This work has been a multifaceted approach toward improving the dependability of many-core systems. The experimental setup consists of the Single-Chip Cloud Computer as the target platform and the InfOli simulator as the target application, upon which a periodic application-level C/R implementation was introduced. The implementation is controlled by the proposed Depman tool which attempts to improve the overall dependability of the system by considering various parameters.

Depman provides error recovery to the target system through the execution of countermeasure procedures. It also enables thermal-aware task allocation after the repair operation and it attempts to minimize the time overhead of the C/R implementation by observing the system's failure rates. For this procedure, the MTTF of the system is estimated on-the-fly by a moving average filter and then used for the calculation of the optimal checkpoint interval.

An error injection module was implemented for the efficiency measurements of the optimization scheme. Using time-varying injection scenarios, the step response of the moving average filter was measured for various filter lengths. These step responses indicate that longer filter lengths lead to more accurate estimate at the steady state, at the cost of increased transient time.

The performance and energy efficiency metrics were measured at scenarios of both constant and time-varying system failure rates. In the case of a constant failure rate, it was observed that on-the-fly estimation does not

introduce any measurable efficiency loss when compared a static optimization scheme, where no estimation is performed. In the case of time-varying failure rates, the falling step scenario showed some efficiency improvements over the static optimization, which were not present in the case of the rising step scenario.

6.2 Future Work

Several modifications and expansions can be performed to the Depman tool in order to achieve better efficiency and a broader functionality spectrum. These future work samples are presented in this section using the numbered components of the block diagram of Figure 4.2, in order to demonstrate the affected portions of the presented framework.

- Given the closed-loop form of the checkpoint interval optimization process, the minimization of time overhead could be approached as a state-space control problem. In this case, the non-observable MTTF variable can be estimated through a state observer [50]. Furthermore, various other options can be considered for the on-the-fly MTTF estimation and the estimation of the benchmarked checkpoint interval and latency metrics, using the appropriate types of filters. These proposals are concerned with component 3, the MTTF estimator.
- the Depman tool can be extended for use on platforms, requiring intrusion to components 2a and 4, and on other applications, requiring changes on components 2b and, possibly, 5. Components 3 and 5 can be also reworked to operate with aperiodic checkpointing techniques [38]. Aperiodic techniques can be used along with a failure distribution estimator in order to improve an optimal checkpoint placement for any failure distribution.
- Despite ensuring a high degree of dependability on the SCC, Depman is susceptible to failures of the host machine, the MCPC. As a result, the Depman tool can be expanded with fault tolerance capabilities through another Checkpoint/Restart implementation. Another potential extension is the dependability management of multiple machines, allowing use of the presented techniques on networks of many-core nodes. These changes demand the design of a meta-system, which utilizes the existing depman tool as a component.
- The principles of the closed-loop optimization scheme can be also applied for uses of time redundancy, in addition to C/R. Redundancy

is used to provide fault tolerance in systems with multiple nodes and it is guaranteed to detect SDC occurences. Therefore, it has been heavily used in HPC systems, along with C/R techniques [15]. As a result, a similar closed-loop approach could be developed for the optimization of both C/R and redundancy by estimating the reliability characteristics of a system's components and using existing optimization approaches [29, 26]. Adapting Depman to a combined redundancy and C/R scheme would require changes in components 2a, 2b and 5, as well as a restatement of the event loop discussed in Chapter 4.

Bibliography

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1--12. IEEE, 2009.

[2] K. Arya, G. Cooperman, A. Dotti, and P. Elmer. Use of checkpoint-restart for complex hep software on traditional architectures and intel mic. *ArXiv e-prints -- 1311.0272*, November 2013.

[3] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11--33, 2004.

[4] Paolo Bazzigaluppi, Jornt R De Gruijl, Ruben S Van Der Giessen, Sara Khosrovani, Chris I De Zeeuw, and Marcel TG De Jeu. Olivary subthreshold oscillations and burst activity revisited. *Frontiers in neural circuits*, 6, 2012.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of mpi programs. *ACM Sigplan Notices*, 38(10):84--94, 2003.

[6] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C 3: A system for automating application-level checkpointing of mpi programs. In *Languages and Compilers for Parallel Computing*, pages 357--373. Springer, 2004.

[7] John Bruno and Edward G Coffman Jr. Optimal fault-tolerant computing on multiprocessor systems. *Acta Informatica*, 34(12):881--904, 1997.

[8] Giorgos Chatzikonstantis. Energy aware mapping of a biologically accurate inferior olive cell model on the single-chip cloud computer. Technical report, National Technical University of Athens, Sept 2013.

[9] AG Colombo, D Costantini, and RJ Jaarsma. Bayes nonparametric estimation of time-dependent failure rate. *IEEE transactions on reliability*, 34(2):109--112, 1985.

[10] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Computational Science—ICCS 2003*, pages 3--12. Springer, 2003.

[11] John T Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303--312, 2006.

[12] Jornt R De Gruijl, Paolo Bazzigaluppi, Marcel TG de Jeu, and Chris I De Zeeuw. Climbing fiber burst size and olivary sub-threshold oscillations in a network setting. *PLoS computational biology*, 8(12):e1002814, 2012.

[13] Anand Dixit, Raymond Heald, and Alan Wood. Trends from Ten Years of Soft Error Experimentation. In *IEEE SELSE Workshop*, 2009.

[14] Awad El-Gohary. Bayesian estimation of the parameters in two non-independent component series system with dependent time failure rate. *Applied mathematics and computation*, 154(1):41--51, 2004.

[15] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for hpc. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 615--626. IEEE, 2012.

[16] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, and Kurt Ferreira. Detection and correction of silent data corruption for large-scale high-performance computing. Poster at the 24th IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2011, Seattle, WA, USA, November 12-18, 2011.

[17] Samuel H Fuller, Lynette I Millett, et al. *The Future of Computing Performance: Game Over or Next Level?* National Academies Press, 2011.

[18] Robert Geist, Robert Reynolds, and James Westall. Selection of a checkpoint interval in a critical-task environment. *Reliability, IEEE Transactions on*, 37(4):395--400, 1988.

[19] T. Grasser, B. Kaczer, W. Goes, H. Reisinger, T. Aichinger, P. Hehenberger, P. J Wagner, F. Schanovsky, J. Franco, M.T. Luque, and

M. Nelhiebel. The paradigm shift in understanding the bias temperature instability: From reaction-diffusion to switching oxide traps. *Electron Devices, IEEE Transactions on*, 58(11):3652--3666, Nov 2011.

[20] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147--160, 1950.

[21] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[22] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.

[23] Howard, J. et al. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE JSSC*, 46(1):173--183, 2011.

[24] Sukriti Jalali. Trends and implications in embedded systems development. *TCS white paper*, 2009.

[25] Jean-Claude Laprie. Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2--11, 1985.

[26] Gregory Levitin, Anatoly Lisnianski, Hanoch Ben-Haim, and David Elmakis. Redundancy optimization for series-parallel multi-state systems. *Reliability, IEEE Transactions on*, 47(2):165--172, 1998.

[27] Yibei Ling, Jie Mi, and Xiaola Lin. A variational calculus approach to optimal checkpoint placement. *Computers, IEEE Transactions on*, 50(7):699--708, 2001.

[28] Yudan Liu, Raja Nassar, CB Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1--9. IEEE, 2008.

[29] Guoming Lu, Ziming Zheng, and Andrew A Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 49--56. ACM, 2013.

[30] Tim Mattson and Rob van der Wijngaart. Rcce: a small library for many-core communication. *Intel Corporation, May*, 2010.

[31] Timothy G Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1--11. IEEE Computer Society, 2010.

[32] Alexandros Mavrogiannis, Dimitrios Rodopoulos, Christos Strydis, and Dimitrios Soudris. Depman: An adaptive run-time dependability manager for many-core checkpoint/restart implementations. In *(submitted)*.

[33] Trent McConaghy, Kristopher Breen, Jeffrey Dyck, and Amit Gupta. *Variation-Aware Design of Custom Integrated Circuits: A Hands-On Field Guide*. Springer, 2013.

[34] A.A. Mendon, R. Sass, Z.K. Baker, and J.L. Tripp. Design and implementation of a hardware checkpoint/restart core. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1--6, June 2012.

[35] Ashwin A Mendon, Ron Sass, Zachary K Baker, and Justin L Tripp. Design and implementation of a hardware checkpoint/restart core. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1--6. IEEE, 2012.

[36] Shubu Mukherjee. *Architecture design for soft errors*. Morgan Kaufmann, 2011.

[37] Hang T Nguyen, Yoad Yagil, Norbert Seifert, and Mike Reitsma. Chip-level soft error estimation method. *IEEE Transactions on Device and Materials Reliability*, 5(3):365--381, 2005.

[38] Adam J Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 14--23. ACM, 2006.

[39] D. Rodopoulos, P. Weckx, M. Noltsis, F. Catthoor, and D. Soudris. Atomistic pseudo-transient bti simulation with inherent workload memory, 2014.

[40] Dimitrios Rodopoulos, Giorgos Chatzikonstantis, Andreas Padelopoulos, Dimitrios Soudris, Chris I. De Zeeuw, and Christos Strydis. Optimal mapping of inferior olive neuron simulations on the single-chip

cloud computer. In *Embeded Computer Systems (SAMOS) 2014 International Conference on*, 2014.

[41] M. Sadri, A. Bartolini, and L. Benini. Single-chip cloud computer thermal model. In *Thermal Investigations of ICs and Systems (THERMINIC), 2011 17th International Workshop on*, pages 1--6, Sept 2011.

[42] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337--350, 2010.

[43] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Quan Shi, R. Allmon, and A. Bramnik. Soft error susceptibilities of 22 nm tri-gate devices. *Nuclear Science, IEEE Transactions on*, 59(6):2666--2673, Dec 2012.

[44] Luís Moura Silva and João Gabriel Silva. System-level versus user-defined checkpointing. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 68--74. IEEE, 1998.

[45] Josh Simons. Hpc cloud bad; hpc in the cloud good. In *Parallel and Distributed Processing Symposium, International*, pages 891--891. IEEE, 2013.

[46] A. Singh and Z. Mourelatos. Time-dependent reliability estimation for dynamic systems using a random process approach. *SAE International Journal on Materials and Manufacturing*, 3(1):339--355, 2010.

[47] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94--125, March 2004.

[48] TOP500 Supercomputer Sites. Cores per socket - performance share, May 2014.

[49] John Paul Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *Distributed Computing and Internet Technology*, pages 221--234. Springer, 2006.

[50] Weiwen Wang and Zhiqiang Gao. A comparison study of advanced state observer design techniques. In *American Control Conference, 2003. Proceedings of the 2003*, volume 6, pages 4754--4759. IEEE, 2003.

[51] John W Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530--531, 1974.

Chapter 7

Appendix

7.1 Source Code

The latest version of the source code of the Depman tool can be found at https://github.com/afein/depman . The software is licensed under the GPLv3 license.