ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Mitigation of performance variability induced by Checkpoint-Restart using DVFS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κοκόλης Απόστολος

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Αθήνα, Ιούλιος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Mitigation of performance variability induced by Checkpoint-Restart using DVFS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Κοκόλης Απόστολος

Επιβλέπων : Δημήτριος Ι. Σούντρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Ιουλίου 2015.

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

Αθήνα, Ιούλιος 2015

.....
Κοκόλης Απόστολος

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

Contents

Abstract

As performance enhancement is accompanied by the aggressive integration of many-cores to a single chip and technology nodes approach deca-nanometer dimensions, the system's failure rate is becoming significant. Inevitably, computer systems must tolerate such failures. Both hardware and software methods are available enabling fault-tolerance to the systems. The Checkpoint/Restart technique provides reliability to the execution of an application. However, Checkpoint/Restart introduce an additional time overhead in order to achieve the fault-tolerance of the execution, that leads to performance variability.

The scope of this thesis is to enhance a runtime manager, Depman, that orchestrates an application level Checkpoint/Restart technique so that such time overheads are absorbed, achieving performance predictability and reliability on the fly, by using Dynamic Voltage and Frequency Scaling (DVFS). A closed-loop implementation controlling the clock frequency is proposed, that quantifies the time overheads induced by the checkpoint restart process and adjusts the frequency levels of the CPU so that execution time converges to the normal.

Depman was also modified to extend its portability to other platforms and applications and was tested using the self fault injection module to both the Intel's Single-Chip Cloud Computer (SCC) and an x86 general computing platform, evaluating both the execution time and energy consumption of our scheme.

Keywords: Dynamic Voltage and Frequency Scaling (DVFS), Checkpoint/Restart (C/R), Execution Sprinting, Dependability, Reliability, Availability and Serviceability, Single-Chip Cloud Computer (SCC)

Acknowledgements

I wish to express my sincere thanks to Professor Dimitrios Soudris for giving me the opportunity to carry out my diploma thesis under his supervision. His guidance and advices, alongside with his invaluable research experience provided me with the motivation and inspiration to accomplish this work.

In addition, I would like to thank Dimitrios Rodopoulos, who stood by me throughout the duration of this thesis with accurate observations and suggestions. He was always willing to support me in every difficulty I faced and share his knowledge with me, so i feel grateful for having the opportunity to cooperate with him. Also, I could not miss to thank Alexandros Mavrogiannis whose work first introduced me to the challenging field of system reliability and gave rise to my collaboration with the Microprocessors Laboratory and Digital Systems Lab of the School of Electrical and Computer Engineering of the National Technical University of Athens.

I give my special thanks to Zisi Iliana and Kiriaki Kokoli for their enjoyable support and cheering me up whenever I needed it. I also thank all my friends who supported me throughout my studies in NTUA.

Finally, I like to thank my family who always encouraged and assisted me to achieve my goals.

Kokolis Apostolos

List of Figures

List of Tables

Chapter 1
Introduction

In the recent years, technology nodes approach deca-nanometer dimensions and even though novel transistors exhibit significant improvement in their reliability profiles, bit level corruption has been a major reliability concern in microprocessor design [5].

Moreover, computer companies incorporate multiple processing nodes on a single chip in order to enhance the performance of their systems, both for Embedded Computing (EC) and High Performance Computing (HPC). This aggressive integration leads to increased failure rates at the circuit and system level [8]. Inevitably, computer systems must tolerate those failures, especially as far as reliability and safety are concerned, since availability,integrity and maintanability are, mostly, guaranteed by the hardware and operating system. As a result fault-tolerance mechanisms should be adopted, in order to provide reliability and availability to these systems.

There are both hardware (HW) and software (SW) available methods for the mitigation of such errors. Computer architects enhance their designs with reliability, availability and serviceability (RAS) schemes to identify and correct such errors. Also software defined methods have been implemented, such as the application or system level Checkpoint Restart (C/R) method, which periodically saves a snapshot of the application's key data structures and performs a rollback-recovery when an error occurs.

However, fault-tolerance mechanisms rarely come with no associated time overheads for the execution of applications. As a result, Error Correcting Codes (ECC) produce a performance degradation, which is often quantified by Performance Vulnerability Factor (PVF) [9], depending on the implentation and the type of the detected error.

This thesis covers the process of introducing fault-tolerance to an application, while at the

same time temporal overheads produced by such RAS mechanisms are absorbed, using Dynamic Voltage and Frequency Scaling (DVFS) techniques, controlled in a closed loop. DVFS is a power management technique that provides the capability to adjust the voltage and frequency to different values, depending upon circumstances and proportionally adjusting the execution speed of the application.

In this context, we evaluate our scheme by modifying an adaptive dependability manager, called Depman [10], that uses an application level Checkpoint/Restart method to provide reliability to an application. So we incorporate the DVFS module in order to provide besides fault-tolerance and error recovery, time overhead mitigation.

Our target application is a time-driven simulator of the inferior olive neurons (Infoli simulator) [11]. Our application scheme is ported and tested to a many-core platform, Intel Lab's Single-Chip Cloud Computer (SCC) [12], as well as an x86 commercial platform.

In contrast to recent trends in C/R and slack reclaiming methods, our implementation estimates the time overheads on the fly and adapts to time-dependent error failure rates.

The current thesis is structured as follows:

- In the next chapter we present the samples of Prior Art which motivated us to get involved with slack reclaiming techniques, systems performance and reliability.
- Next, we introduce the overhead produced by a voltage and frequency alteration both for the SCC and the x86 platform. First, we give a brief overview about voltage regulators, which provide the capability of voltage changes, and the two platforms and then we analyze how we perform the alterations to each platform.
- In Chapter 4 we discuss the operation of Depman, including the DVFS module, for our target application and also we exhibit the function of the DVFS module. Furthermore, we formulate the problem of performance dependability, in view of timing noise and we quantify the time overhead caused by the C/R method, as well as the way in which we calculate the exact time overhead during the execution of the application.

- In addition, in Chapter 5 we depict the results of our application scheme for both platforms and analyze in further detail each implementation. First we demostrate the SCC implementation and then two different implementations for the x86 platform. Also we discuss and evaluate our results.

- Finally, in the last Chapter we list a series of conclusions from the work presented herein. Directions for future work are also pointed out.

Chapter 2

Prior Art

2.1 Introduction

In this Chapter we will present the state of the art concerning this thesis. The way to achieve this is to categorize the main aspects that lead to the need of slack reclaiming. In Figure 2.1 we illustrate the scope of the Chapter.

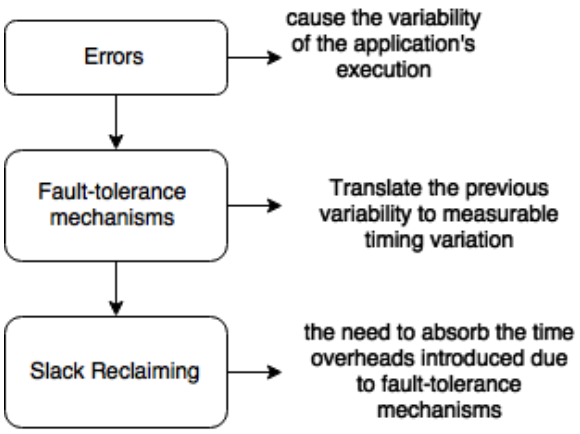


Figure 2.1: Representation of the urgency for slack reclaiming

So, first, we analyze the causes of errors and the their types that violate the reliability of a process. Then we analyze techniques that tolerate these errors. In precise, we analyze the Checkpoint/Restart technique that is occupied in this thesis to enable fault-tolerance. We provide an insight into how the Checkpoint/Restart technique is used in modern systems, its usefullness and the way that it is modelled. Finally, after we have established the need for slack reclaiming because of the time overheads introduced by fault-tolerance mechanisms, we present different approaches for computational sprinting, both parallel and frequency sprinting [4], as well as an approach regarding the slack reclaiming of cycle noise induced by RAS mechanisms [5].

2.2 Error Profilling

Current trends suggest that soft errors, bit-level corruption of computer data, will emerge as a priority for microprocessor designers in the future [13]. As a result, a need for the classification, intuitive understanding and quantitative measurement of errors should be launched, regarding the way that errors affect the system behavior.

Transient faults may arise from radiation [1, 2] ,transistor aging [3] and the constant device miniaturization, as well as near-threshold computing [14, 9]. Also chip overheating and voltage spikes can cause such failures.

The potential errors can be classified into two different categories, the Silent Data Corruptions (SDC) and the Detected Unrecoverable Errors (DUE) [13].

Silent Data Corruptions are faults, which cause the system to generate erroneous outputs. These type of errors are not detected by the corresponding hardware or operating system and are directly related with each specific application. On the other hand, DUEs are detected by hardware or firmware and have the characteristic that when they manifest the application is either terminated or blocked. However, detecting an error does not provide any reliability, but does provide the fail-stop behavior and avoids data corruption. The fail-stop model takes for granted that the appearence of an error causes the simultaneous termination or block of the process. It is likely though that an error is detected after the process termination or , in case of the C/R restart method, during the checkpointing procedure which means that multi-version checkpointing is needed. Both SDCs and DUEs are expressed by two variables. The first is the Mean Time to Failure (MTTF) metric, which is the average period of time between two consecutive failures. The other one is the Failures in Time (FIT), which represents the number of errors detected in a billion device hours.

2.3 Checkpoint Restart

The Checkpoint Restart (C/R) technique enables fault-tolerance through the storage of snapshots either of the system or the application state, known as checkpoints. In case of a

failure these snapshots are used to restore the system to a previous stable state. The C/R is a well known technique used in HPC [15, 16, 17], as well as many-core platforms and distributed systems [18]. In the latter, C/R procedure demands the consistency between the nodes of the system, which can be violated because of heavy packet loss or the network latencies, so checkpointing is either implemented over unified distributed storage schemes or it is managed through coordination schemes [19].

As it is already indicated, checkpointing can be performed both at the system or application level. At system level, the platform's components are stored in checkpoints, such as registers and memory contents, while on application level the crucial application structures are stored in checkpoint files [20]. As a result the application level C/R restart implementation can outperform system level C/R as the size of checkpoints is, generally, smaller, since only the storage of the application's critical components is required. There are several tools which facilitate the process of C/R. As an example is The Cornell Checkpointing pre-Compiler [15] known as C^3, thoroughly used in HPC systems, which parses the programmers framework and indicate potential checkpoint and restarting locations.

In the context of this thesis a Supervised C/R application level technique will be reused and modified [10], called Depman. Depman, initialy, orchestrates a C/R closed loop model to introduce reliability to a many-core platform, the Single Chip Cloud Computer (SCC). Checkpoints are stored periodically for the notable application structures in double buffered files and also it monitors for both DUE's and SDC errors. When an error is detected Depman performs the available countermeasures for the SCC platform, such as core-reboot or platform reset, and restarts the application. No design-time benchmark parameters are required, since it automatically adapts to time-dependent error failure rates of the system. The operation of Depman concerning this thesis will be presented in Chapter 4.

The Checkpoint procedure can be modelled as presented in Figure 2.2.

Where the *Checkpoint Interval,*τ, indicates the time or simulation steps between two sequential checkpoints. The time consumed by the checkpoint to be taken is called $T_{\text{checkpointing}}$, TTR is the time needed for the application to be restarted and T_{repair} is the time for the ap-

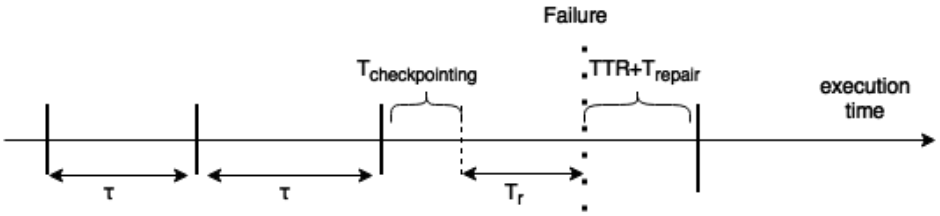


Figure 2.2: Modelling the Checkpoint cycle

plication to restore its state from the checkpoint files. Finally, T_r is the application rollback time that needs to be performed again. As a result, it is clear that the added fault-tolerance provided by the C/R technique comes with the cost of added execution time and data redundancy [21]. The time overheads introduced by the C/R process are discussed in greater detail in Section 4.4.

2.4 Execution Boosting

In embedded systems, the desire for increased number of operations per unit power seems to be a major concern in the near future. However, this type of power efficiency commonly evokes partially compromised resiliency [22]. Moreover, High Performance Computers (HPC) need to adapt fault-tolerance mechanisms in order to improve their reliability and availability, but the time overhead introduced by such mechanisms lead to the concept of "Reliability Wall" [23], meaning that performance and scalability might be violated due to reliability techniques. For example the Checkpoint/Restart technique, as presented in the previous Section, improves the system's reliability but incurs additional execution time for saving the checkpoints and for performing rollback-recovery.

These facts lead to the idea of slack reclaiming and execution sprinting. Since additional time overhead has been imported to the execution of our application, it is important to adapt real-time techniques to achieve the minimum performance degradation, as far as time and energy constraints are concerned. In this direction we have two different types of computational sprinting as presented in Figure 2.3.

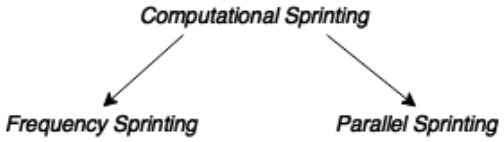


Figure 2.3: Types of computational sprinting [4]

Frequency sprinting refers to the alteration of CPU's clock to opperate on higher frequencies so that the execution is accelerated.

In this scope, Intel has presented the Turbo Boost Feature, which makes the processor opportunistically increase the frequency of the cores depending on the core temperature, the number of active cores and the estimated power consumption [24]. It is known that idle cores consume small amount of active power. So when a portion of cores are inactive the extra power headroom available can be diverted to the active cores so that the execution is quickened without compromising the power and thermal envelope. Of course, the need for dynamically power asymmetric multi-core processors is raised, so that all cores may use the same instruction set, but the frequency of each core can vary independently [24].

The speedup resulting from this method is not identical for all applications. It depends on the application type, i.e. whether the application uses integer or floating point numbers, the last level cache miss rate, the temperature of the platform and the effective frequency for the application. As a result we do not expect a memory intensive application to experience as much performance gain as CPU intensive. The effective frequency levels for memory intensive applications are lower, since the cache miss rate is greater and the execution delays waiting data fetching, so they cannot achieve the full potential of Turbo Boosting. These claims are substantiated in related work [24], where the efficiency of Turbo Boost is depicted.

Another factor that affects the results of Turbo Boosting is the interference between applications running in parallel. As it is presented in related work [24], the speedup achieved with Turbo Boost is greater when two CPU-intensive applications are executed concurrently than when two Memory-intensive applications are running at the same time. Also it is important whether the applications are mapped on the same or different cores.

In every case though, the application executes faster when running on Turbo Boost in the cost of a significant increase in energy consumption.

Parallel sprinting refers to the activation of reserved cores and the distribution of computational load. Of course parallel sprinting results to high performance gains and exploits better the thermal capacitance of the platform. However, the disadvantage of this method is that not all applications support parallel programming and even if they do, sequencial phases of parallel application are, almost, inevitable.

Another approach is to practice both frequency and parallel sprinting where sequential phases of applications can be boosted as well. This method has been applied as it is presented in related work [4, 25]. In that case computational sprinting was adopted in order to increase the system's responsiveness when the computation demands are high, always taking under consideration the thermal limits of the platform.

We need to introduce two new concepts, the Unabridged and the Truncated sprint [4]. The first, refers to a sprint that is able to complete within the thermal constraints of the platform, no matter if it is parallel or frequency sprint, while the second refers to a sprint that needs to be sustained because the thermal limits where infringed. As a consequence we can realize that we can not sprint unconditionally and for a prolonged period of time.

Energy consumption is a critical factor nowadays. Sprinting has the pottential to lead to more energy efficient schemes by "amortizing the fixed uncore power consumption over a large number of active cores and capturing race to idle effects" [4]. However, it is a fact that frequency sprinting requires higher voltage values that increase the power consumption during the execution. The more idle core power is optimized the more energy effective sprinting will be.

It is important to note that in previous work, sprinting has been used to enhance application performance. On the contrary in our work, frequency sprinting is used to reclaim correction overheads. Furthermore, it is important to note that sprinting, as mentioned in related work, tends to press the Thermal Design Power (TDP), whereas in our case we as just using "legal" P-states to enable dependable performance.

Recently, techniques have been proposed and evaluated facilitating observability and controllability of the target platform in order to enhance performance stability [5]. The goal

is to mitigate performance variability at run time level absorbing the cycle noise overheads caused by RAS mechanisms, through a Proportional-Integral-Differential (PID) closed-loop control scheme that modify the voltage and frequency levels to the appropriate values, Figure 2.4. The concept of this idea is to formulate the issue of performance variability and quantify the overhead in what is called cycle noise, control the DVFS process with a PID controller and evaluate the time and energy outcome.

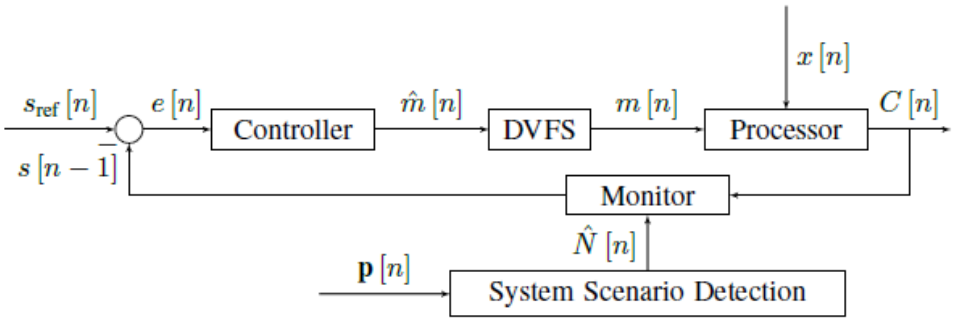


Figure 2.4: Block diagram for the run time performance dependability scheme in view of RAS temporal overheads [5]

As indicated in Figure 2.4 the cycle noise (x), interferes with the timing budget of the application (N) and produces the slack (s) which indicates the recession of the application's execution time. So a frequency multiplier is proposed and the closest available frequency of the processor is selected. This concept was the main springboard of this thesis and its functionality is greatly discussed throughout this thesis. The results of these simulations indicate that time variations are negligible but the energy consumption exhibit a rise [5], in case the aggression of DVFS is not appropriately harnessed.

Chapter 3

DVFS Overhead Characterization

3.1 Voltage Regulators

Dynamic Voltage and frequency scaling is a well known technique used to reduce energy in digital systems. However, the effectiveness of DVFS can be restricted by prolonged voltage transitions. The same applies for our case that we use DVFS to accelerate the execution of the application when needed. So with the growing power and execution time management concerned, as well as the need for per core DVFS, the requirement for efficient voltage regulation has become critical. This Section will present and compare off-chip and on-chip voltage regulators, and also will introduce two rife regulator topologies.

A voltage regulator is needed to keep voltages within the prescribed range that can be tolerated by the electronic device. Its role is to deliver power from the source to the load, with minimum loss and maintain constant voltage during transient response [26]. Most voltage regulators are off-chip devices due to the large power transistors and output filter components that are required. However, lately, great emphasis is given to integrating the voltage regulators on the same chip as the load they feed, on-chip regulators. That is because on-chip regulators result in multiple benefits. They are smaller so they can be integrated on chip, which results to the reduction of Process Control Block (PCB) area required from off-chip regulators, they provide faster voltage switching and offer the potential to provide multiple on-chip power domains to chip multiprocessor systems [27].

There are three important regulator characteristics that should be taken into account when designing on-chip instead of off-chip regulators. These are regulator efficiency, load transient response and voltage switching time [27, 26].

Regulator efficiency concerns the power losses due to the regulator, which depend on the size

of switching power transistors, switching frequency and load conditions. Regulators with higher switching frequencies, such as on-chip regulators, are capable of fast voltage scaling, but incur higher regulator losses. However, on-chip regulators are closer to the load so the losses from parasitic resistors ($I^2 \times R$) between the source and the load are less.

Load transient response determines how much the voltage fluctuates in response to a change in current. For on-chip implementation, the size of the filter components is reduced, leading to efficiency degradation due to voltage fluctuations. That means there is a trade off between operating at high switching frequencies and achieving satisfactory power efficiency. However, on-chip regulators remove impedance restrictions by reducing mid-frequency package resonance issues.

As far as voltage scaling time is concerned, the voltage does not scale immediately, but gradually. So the yield of DVFS has been hindered by slow voltage scaling. On-chip regulators offer the ability for nanosecond voltage scaling and per-core DVFS.

Two widely used regulator topologies are the linear and switching regulators [27, 26]. Linear regulators offer several advantages such as ease of on-chip integration, small size, low cost, no complexity and fast response to load transients. Furthermore, since they are inexpensive and small they are ideal for multiple voltage domains. Unfortunately, the power conversion efficiency of a linear regulator is constrained by its dependency on the Vout/Vin ratio, where Vout is the output and Vin is the input voltage of the regulator. In contrast, switching regulators provide better power conversion efficiency and are less sensitive to the Vout/Vin ratio. Also, they can regulate a wide range of output voltage levels. Different from linear regulators, some types of switching regulators are also capable of providing outputs higher than the input. Hence, switching regulators are better suited for loads employing DVFS, both for turbo boosting and power saving. However, switching regulators also exhibit serious concerns when it comes to on-chip implementation. First of all, the size of a switching regulator is bigger so it is harder to be integrated on a chip. Additionally, switching regulators do not provide clean output voltage, due to the presence of the inductor. So the requirements for high efficiency and high accuracy make the size of the regulator prohibitively large for

on-chip implementation.

3.2 Target Platforms

This Section will be about exploring the details of the target platforms that we will test our application scheme. We test our application in two different platforms. The first one is the Intel's Single-Chip Cloud Computer,SCC, experimental platform and the second one is an x86 architecture commercial laptop distributed by Hewlett-Packard.

3.2.1 Single-Chip Cloud Computer, SCC

The Single-chip Cloud Computer (SCC) is a research chip created by Intel Labs to study many-core CPUs, their architectures, and the techniques used to program them [6, 28]. The processor consists of 48 cores, which are grouped in tiles of two cores each. The tiles are inteconnected through a mesh network. Each tile contains two Intel architecture 32-bit P54C cores, a unified L2 cache memory of 256KB, a router that connects the tile to the mesh and a Message Passing Buffer (MPB), which is used for the message exchange between the cores. The chip allows dynamic voltage and frequency scaling accross the tiles , as will be explained in the next Section 3.3, thus it is appropriate for the purposes of this thesis. Furthermore, the board that hosts the SCC chip communicates with a Management-Console Personal Computer (MCPC) through ethernet and PCIe links. The MCPC is equipped with the SCCKit, which is a software framework for the SCC providing the user the capability to monitor the board remotely. The user can define the power domains of the board, restart the cores, boot linux image on each core, reinitialize the board and ping the available cores. The MCPC and the board can also communicate through a directory called `/shared` which is common for both and can store the output files of an application running on the board. Additionally, a C Library called RCCE is provided with SCC [29]. RCCE is a small library for message passing, similar to the Message Passing Interface (MPI), which is tuned to the needs of many-core chips. RCCE also provides a power management interface to support

power-aware applications. The RCCE source code is crosscompiled with icc/icpc compiler in order to generate the appropriate executable for the SCC board.

3.2.2 x86 architecture platform

The second platform on which we test our implementation, is an x86 architecture HP Pavilion dv6 Notebook, running a Linux 3.8.0-44-generic Ubuntu distribution. Its processor is an Intel(R) Core(TM) $i7-2630QM$ Sandy-bridge. The $i7-2630QM$ is a quad core processor with two simultaneous multi-threading (SMT) contexts per core, providing us eight logical cores. It has a 4GB DDR3 RAM and a 6MB cache.

As far as DVFS is concerned, the system uses the *acpi-cpufreq* driver to perform frequency changes. Also, the *userspace* governor that will be analyzed in Section 3.4.1 is supported, offering the ability to perform our own frequency alterations. The available scaling cpu frequencies range from $800MHz$ to $2GHz$ with a step of $100MHz$.
What is more, the Message Passing Interface (MPI) is supported [30], which is very similar to the RCCE. Our target application is ported both for RCCE and MPI so that we can perform our executions on both platforms.
Finally, contrary to the SCC platform where we use the RCCE power management interface to make voltage and frequency alterations, now the `cpufrequtils` linux package that will be further analyzed in Section 3.4.2 is installed providing the capability for changing the cpu-frequency and monitor the cpu-frequency information using its tools, `cpufreq-set` and `cpufreq-info`.

3.3 SCC voltage/frequency alteration overhead

In this Section we present the time overhead of the voltage alteration on the SCC platform. The SCC platform contains an off-chip Voltage Regulator Controller (VRC) which provides the capability of changing the voltage on a voltage domain of the platform and the frequency of each tile individually. There are seven voltage domains and 24 frequency domains on the platform. Six of the voltage domains comprising four tiles of two cores each in a 2x2 array as shown in Figure 3.1 [6], while the seventh is the entire set of tiles. Each of the frequency domains matches a single tile.

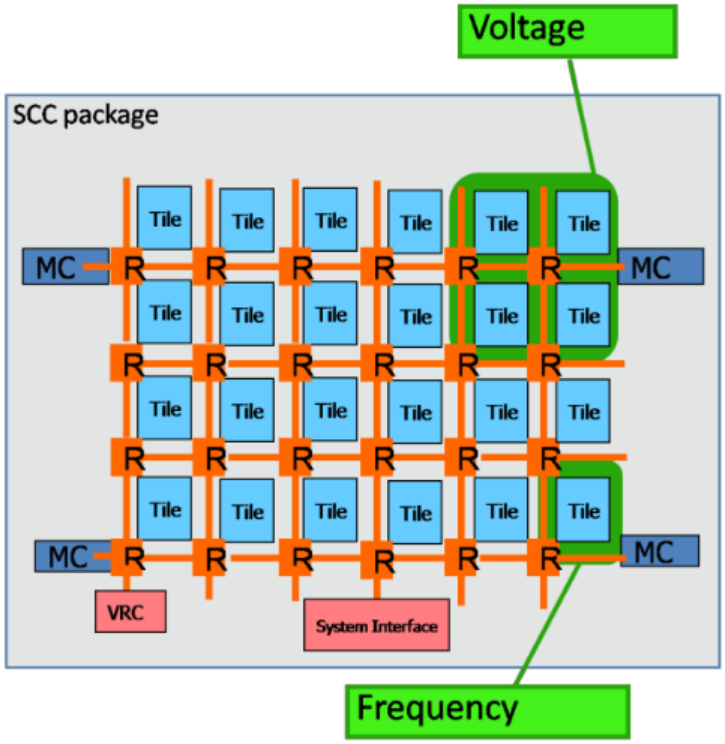


Figure 3.1: SCC Voltage and Frequency domains [6]

In order to change the SCC power we use the RCCE power management call `RCCE_iset_power()`, which sets the tile frequency to the reference clock divided by the supplied divider. In our case the reference clock is 1.6 GHz and the dividers would be either two or three, which lead to a tile frequency of 800 MHz and 533 MHz, respectively. In the case of 800 MHz tile frequency, each voltage domain has a voltage value of about 1.1 Volts, while in case of 533 MHz the voltage of each domain is approximately 0.8 Volts.

To record the measurements that follow we altered the voltage of all domains of the platform from 1.1 Volts (divider two) to 0.8 Volts (divider three) and vice versa for 300 iterations. In each iteration, we measured the total time needed for the voltage to be stabilized to every voltage domain. Also, we recorded the voltage and current of the platform with an interval of approximately 0.3 seconds. To achieve that, we forked two processes:

- the first uses the `RCCE_iset_power()` call to alter the voltage and frequency values and then waits for the voltage to be stabilized to each domain
- the second constantly records the voltage and current of the platform until is terminated by the first.

In Figure 3.2 we present the time overhead of DVFS to change from voltage divider three to voltage divider two, while in Figure 3.3 we present the time overhead to change from voltage divider two to voltage divider three. The y-axis represents the frequency of reporting results in a histogram bin throughout the iterations. The x-axis is the total time needed for the voltage to stabilize to the new value.

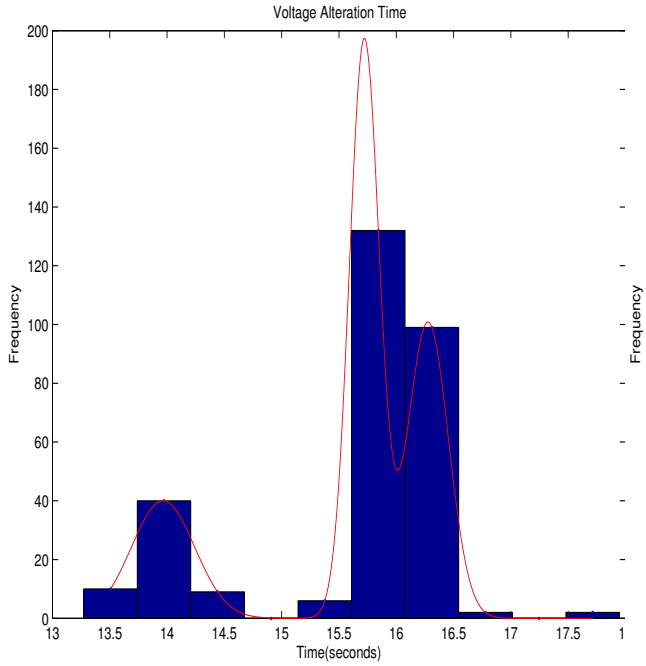


Figure 3.2: Voltage alteration overhead from divider three to divider two

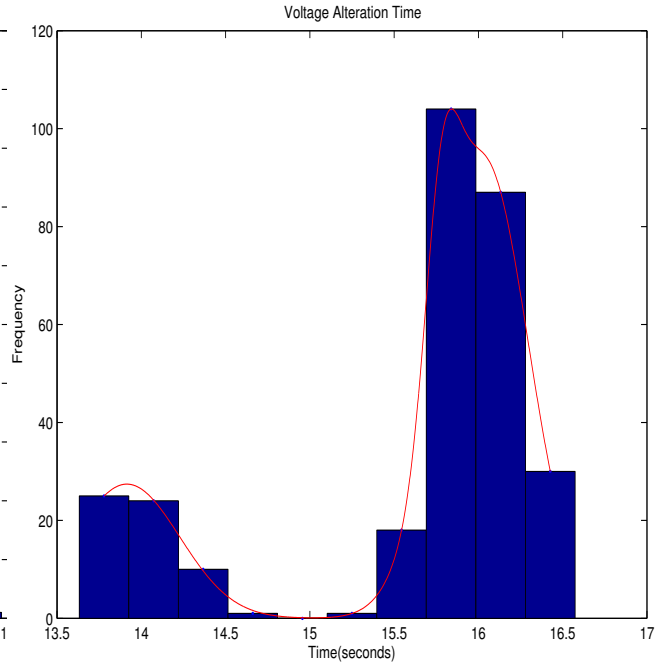


Figure 3.3: Voltage alteration overhead from divider two to divider three

The results are fitted with a Gaussian curve. For Figure 3.2 the fitting curve is

$$f(x) = a_1 \times e^{(-((x-b_1)/c_1)^2)} + a_2 \times e^{(-((x-b_2)/c_2)^2)} + a_3 \times e^{(-((x-b_3)/c_3)^2)} \tag{3.1}$$

while Figure 3.3 is fitted with the curve

$$g(x) = a_1 \times e^{(-((x-b_1)/c_1)^2)} + a_2 \times e^{(-((x-b_2)/c_2)^2)} + a_3 \times e^{(-((x-b_3)/c_3)^2)} \tag{3.2}$$

Both Gaussian curves contain three factors. To evaluate the fit we use two regression models the R-squared and the Root Mean Square Error (RMSE). R-squared is a figure of merit for the fitting. It is the square of the correlation between the response values and the predicted response values. The RMSE is the square root of the variance of the residuals, which are defined as the difference between the observed value of the dependent variable and the predicted value. It indicates the absolute fit of the model to the data, which means how close the observed data points are to the model's predicted values. Whereas R-squared is a relative measure of fit, RMSE is an absolute measure of fit. Lower values of RMSE indicate better fit, while for R-square the closer its value is to one the better the fit is. The values of both RMSE and R-square are show in Table 3.1.

	f(x)	g(x)		f(x)	g(x)
a_1	196.5	39.82	a_3	40.02	27.41
b_1	15.72	15.78	b_3	13.97	13.91
c_1	0.1846	0.1395	c_3	0.3896	0.4437
a_2	100.8	93.3	RMSE	0.9998	0.7151
b_2	16.28	16.03	R-Squared	2.003	1
c_2	0.2535	0.3681			

Table 3.1: R-squared and RMSE values for voltage alteration overhead curves

As a result, we can see that the curves fit the histogram in a very accurate way explaining the behavior of voltage alteration overhead.

Comparing the two figures (Figure 3.2 and Figure 3.3) we can note that there is not appreciable difference in the total time needed for voltage alteration between the two dividers. Changing from divider two to divider three is slightly faster, and that probably is because

the process that uses the `RCCE_iset_power()` call to alter the voltage is executed in a frequency of 800MHz, whereas changing from divider three to divider two the same process is executed in a frequency of 533 MHz. Also reducing the voltage level is a somewhat faster process than increasing it.

Apart from the time of voltage alteration, we also calculated the energy overhead during the alteration between the two dividers. In order to calculate the energy consumption we first determined the power for every 0.3 seconds with the values of voltage and current that we recorded. Then we use the trapezoidal rule to calculate the integral of power during the time of the alteration, which is the energy consumption. As a result we get the following two figures, where y-axis represents the frequency of reporting results in a histogram bin throughout the iterations and in x-axis is the total energy consumption. In Figure 3.4 we represent the energy consumption during the alteration from divider three to divider two, while in Figure 3.5 the energy consumption during the alteration from divider two to divider three.

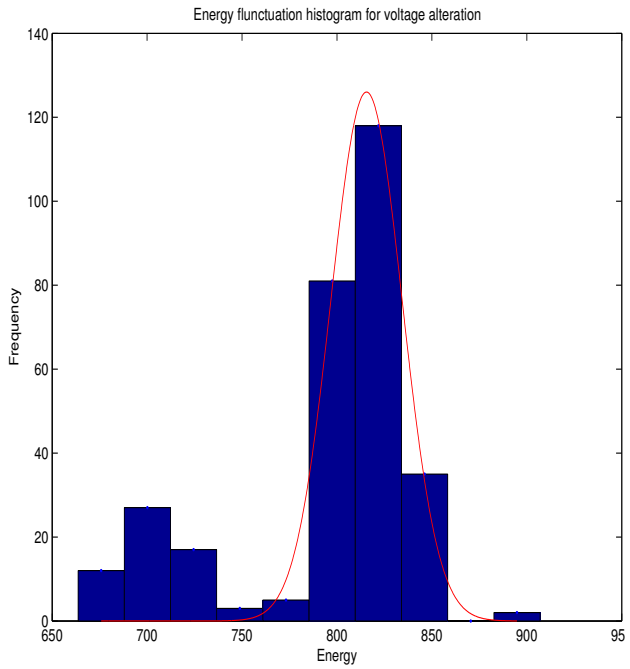


Figure 3.4: Energy fluctuation for voltage alteration from divider three to divider two

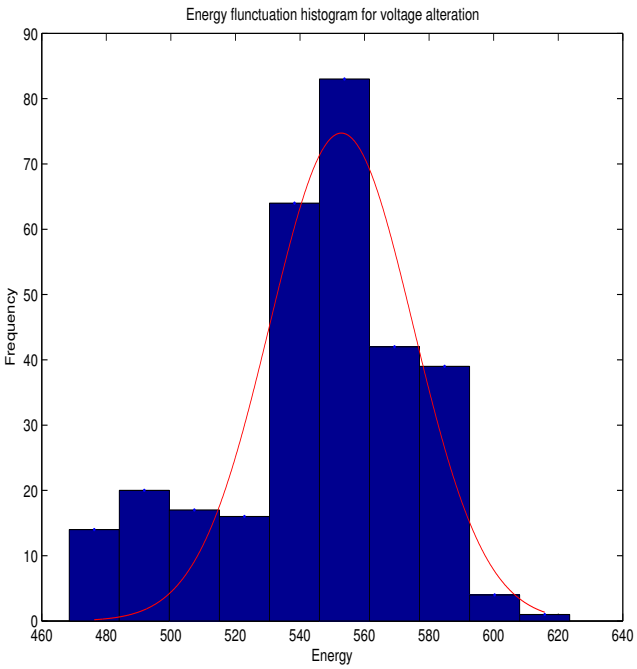


Figure 3.5: Energy fluctuation for voltage alteration from divider two to divider three

Again the results are fitted with a Gaussian curve. For Figure 3.4 the fitting curve is

$$h(x) = a_1 \times e^{(-((x-b_1)/c_1)^2)}$$

and for Figure 3.5 the fitting curve is

$$t(x) = a_1 \times e^{(-((x-b_1)/c_1)^2)}$$

The Gaussian functions have one factor. We evaluate the fit using again the R-squared and RMSE regression models as before and we get the following result as shown in Table 3.2.

	f(x)	g(x)		f(x)	g(x)
a_1	126	74.72	RMSE	0.9132	0.8053
b_1	815.6	552.8	R-Squared	13.13	13.37
c_1	26.54	31.42			

Table 3.2: R-squared and RMSE values for Energy fluctuation curves

The results indicate that although the relative measure of fit is satisfactory, the absolute fit is not as good as it was previously. However, the curves are representative of the energy fluctuation behaviour.

From the two figures (Figure 3.4 and Figure 3.5) we can observe that the energy consumption during the alteration from divider three to divider two is greater. This is because when we want to adjust the voltage level to a higher value on the voltage domains, the current of the platform is also increased.

In particular, we can see this behavior in the following two figures, where in Figure 3.6 we represent the current and voltage fluctuation of the whole SCC platform, not a particular voltage domain, for four random iterations of voltage alteration from divider three to divider two, while in Figure 3.7 we depict the current and voltage fluctuation for four random iterations of voltage alteration from divider two to divider three.

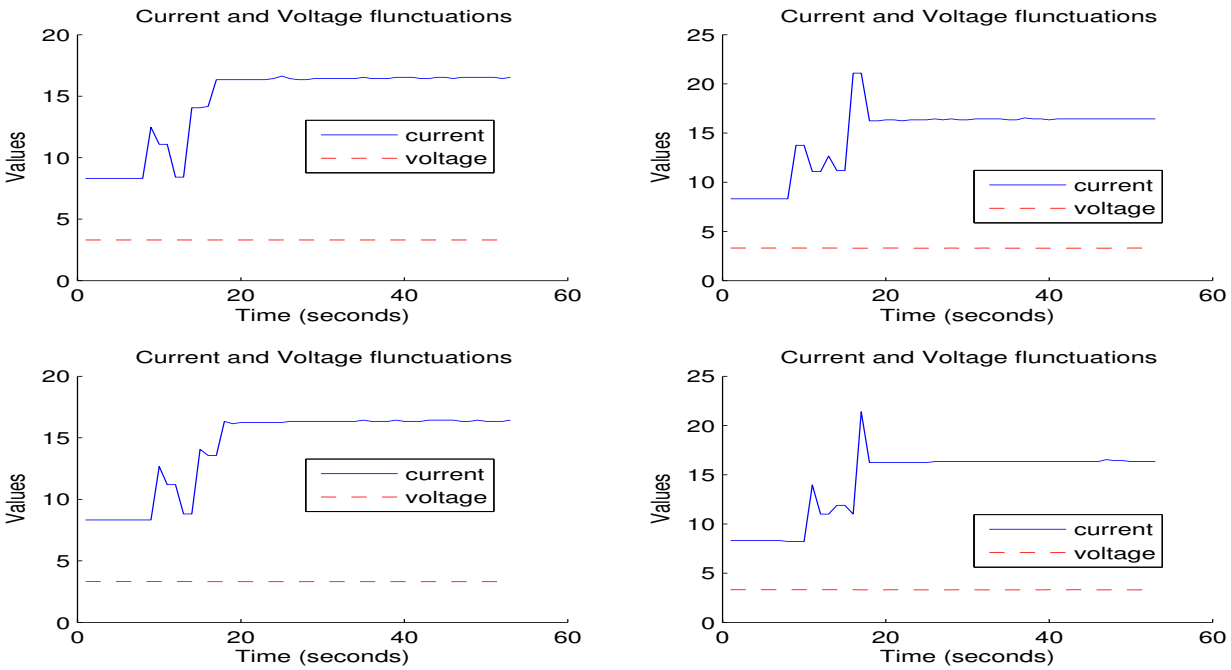


Figure 3.6: SCC voltage and current fluctuation for voltage alteration from divider three to divider two

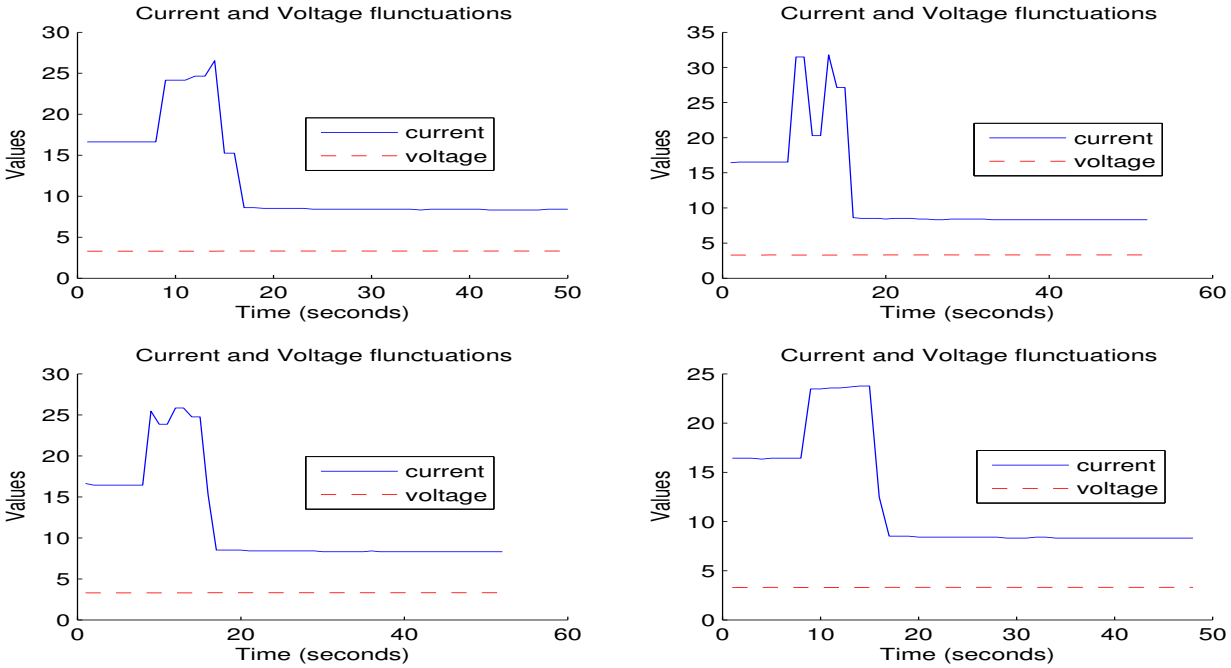


Figure 3.7: SCC voltage and current fluctuation for voltage alteration from divider two to divider three

As a result, in both cases the voltage value of the platform is the same and remains stable throughout the alterations. It is the current that changes its value. In the case of changing

from divider three to divider two, the current switches from a value of 8.3 Ampere to a value of 16.3 Ampere. On the other hand, switching from divider two to divider three, the current value is reduced from 16.3 Ampere to 8.3 Ampere. As a consequence to this behavior the total energy consumption is greater when we increase the voltage and frequency on the voltage islands of the platform since the current of the platform is increasing as well.

In addition, we must note that the current of the platform does not change instantaneously and is resulting to its final value dissimilar in each iteration [31]. This is the reason we must wait until the values are stable in each voltage island before we launch our application again.

In summary, in order to change the voltage value of the voltage domains of SCC adds a time overhead that should also be taken into account during the time reclaiming of our application. Also, we must indicate that executing our application in a higher frequency is a time saving, but energy consuming process, so we must reduce the frequency as soon as the overhead of the restart procedure is reclaimed.

3.4 Voltage/Frequency alteration and overhead for x86 architecture

In this Section we present how we can perform the *DVFS* operation in a machine with x86 architecture. First we will present the available scaling policies of a machine introducing the *CPUfreq governors*, then we will discuss about the available scaling options when we use the *userspace* governor and finally the `cpufrequtils` linux package will be analyzed.

3.4.1 CPUfreq governors

On a given platform, a variety of frequency scaling technologies can be supported and a proper driver must be present to efficiently perform the frequency alterations. The `cpufreq`

infrastructure allows the user to use one CPU-specific driver per platform, plus a number of frequency-changing policies, known as governors [32, 33].
The CPUfreq subsystem has, generally, five available in-kernel governors ,which can change the frequency depending on certain criteria such as CPU usage, energy consumption or user input as presented in [33, 32, 34].

- The first governor is the *performance* governor. This governor statically sets the processor to the highest frequency available. Of course the highest available frequency can be determined by the user by changing the appropriate system file as will be presented later. The goal of this governor is to achieve the maximum system performance by setting the processor clock speed to the maximum value.

- The second governor is the *powersave* governor. In contrast to the previous governor this one sets the cpu frequency to the lowest available value specified by the user. The goal of this governor is to save power by operating at the lowest processor clock speed. However, this governor often does not save any power since the greatest power savings usually come from the savings at idle state. Powersave governor prolongs the execution of a process and the system takes longer to enter an idle state [34].

- The third governor is the *ondemand* governor. This governor sets the cpu-frequency depending on the current usage of the CPU. If the utilization of the CPU exceeds a certain threshold the frequency is set to the highest available. If the utilization is less than the threshold, then the frequency is reduced to the next available, until it reaches the lowest frequency bound. The CPU utilization sampling rate, the threshold and the available frequency borders can be set by the user.

- The fourth governor is the *conservative* governor. This governor operates such as the *ondemand* governor, with the difference that it gracefully increases and decreases the CPU speed rather than setting the frequency to the maximum level when the CPU utilization exceeds a certain threshold.

- The fifth governor is the *userspace* governor. This governor allows the user or any userspace program to adjust the frequency of the CPU. This is the governor that we

will use for the purposes of this thesis, as it gives as the authorization to perform our own frequency adjustments within the available frequency borders.

As mentioned in related work [33, 35] ,voltage scaling is achieved using voltage layer and regulator driver. Every time the CPUfreq driver makes an alteration on the cpu-frequency, the voltage corresponding to this frequency should be selected. This is achieved by iterating the *OPP List* (Operating Performance Point), which is a list of tuples consisting of a frequency value and the voltage required to run at that frequency. Then the device scale function requests the voltage layer to scale the device voltage to the target voltage.

The voltage layer consists of the information of all voltage domains in the system and configures all voltages during voltage layer initialization. Thus, when a voltage change is requested, the voltage layer requests the regulator framework to change the device voltage to the target voltage. Then the regulator driver verifies if the target voltage is within the limits of the voltage domain and regulator supply constraints and performs the alteration.

3.4.2 DVFS options and alteration

The `/sys` filesystem of the linux kernel provides the interface for the CPUfreq changes. Specifically, in the `/sys/devices/system/cpu/cpu*/cpufreq` folder are all the files that contain the available frequency information. When we use the *userspace* governor the folder contains the following files, that are crucial for our implementation.

- First is the `scaling_driver` file which indicates the name of the low-level CPU-specific driver that is being used on this system.
- Next, are the `scaling_cur/max/min_freq` files which contain information about the current, the maximum and the minimum frequency that we use or we can use, respectively. We can change the maximum and minimum frequency limits from these files, but the values should always be within the range indicating by the `cpuinfo_max/min_freq` files.

- Another file that will concern us about the frequency alteration is the *scaling_setspeed* which is a read-write file. When we read this file, it denotes the current CPU frequency. However, the user can write a value to this file and the CPU will change the frequency to the one specified by the user.

- Finally is the `cpuinfo_transition_latency` file which contains the latency value of a frequency alteration.

What we should note here is the great difference between the time overhead caused from the voltage alteration between the SCC platform and an x86 architecture commercial platform. As shown in Figures 3.2 and 3.3 the alteration of voltage and frequency needs about 16 seconds. On the other hand, an x86 machine only needs some μseconds to perform the alteration. For instance, the platform that we will test our application and benchmarks needs 10 μseconds to perform a frequency and voltage change. That means that the cycle noise introduced while performing the *DVFS* operation is widely reduced and the slack reclaiming is expected to be more accurate.

In order to perform our alterations we use a tool called `cpufreq-set` which is included in the `cpufrequtils` package. This executable allows us to change the CPUfreq governor to userspace for each CPU of our machine and also gives us the capability to set the frequency at a current value without having to alterate the value of `/sys/devices/system/cpu/cpu*/cpufreq/scaling_setspeed` all the time. So we can define the frequency value that we want for a certain CPU, or even for all CPUs, and the `cpufreq-set` will set the value to the `scaling_setspeed` file. In this way if we want to perform *DVFS* to all CPUs we shall not change each file entry one by one, but we invoke the `cpufreq-set` command with the CPUs that we want to alter the frequency of.
What is more, `cpufrequtils` contains a tool called `cpufreq-info` which gives us the utility to retrieve cpufreq kernel information at any time and also providing statistics about the cpu-frequencies utilization.

Chapter 4

Depman tool and timing noise quantification

4.1 Introduction

Fault tolerance mechanisms both on Hardware and Software introduce a performance degradation known as *Performance Vulnerability Factor* (PVF), which is the additional execution time of an application because of the invocation of RAS mechanisms.

In this chapter, we first present the target application we use to test our implemention scheme. Then we examine the operation of Depman tool as far as the Checkpoint-Restart procedure is concerned. Also we present the scheme of the *DVFS* module we use to reclaim the time overheads produced by the C/R operation. Additionally we analyze the way in which we inject errors during the application's execution in order to examine the performance of our tool.

Finally, we quantify the timing noise introduced by the Checkpoint-Restart procedure and we explain the ways in which we measure the total execution time overheads, in order to reclaim them on the fly.

4.2 Target Application

The target application that has been used for the purposes of this thesis is a simulator of a crucial set of brain cells, called inferior olive (IO) cells, based on the Hodgkin-Huxley model [11, 7, 36, 37, 38]. Each cell comprises of three individual compartments:

- the dentrite compartment, which is responsible for communicating with the rest cells

of the grid for receiving input voltages

- the soma compartment which is the computational center of the cell, performing all time consuming calculations
- the axon compartment which serves as the output for the neuron.

The simulator receives as input the grid size, a connectivity file which declares the static connections between the neuron cells and,optionally, a file of external input currents for each cell. If the last file is not provided as input to the simulator, pseudo-random input currents are generated for each cell.

The Infoli simulation data flow is briefly explained in Figure 4.1 for each simulation step $t_0, t_1, t_2, ...$

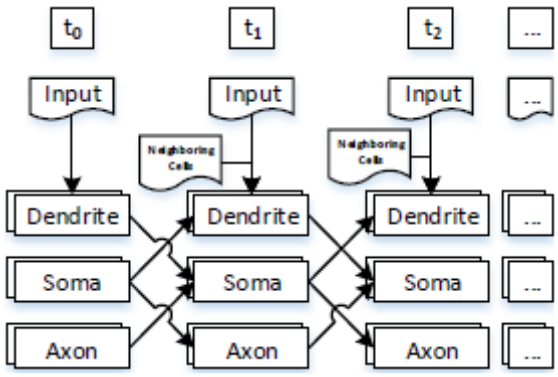


Figure 4.1: Dataflow for the Infoli simulator [7]

First, the dedrite compartment is fed input current as defined by the input file for external current inputs or the generated currents. Then the dendrite compartment records the dentritic voltage levels of its communicating cells, as described by the connectivity file. After the communication is done, each compartment performs its computations to recalculate their biological parameters [7]. For every axonal compartment the new voltage values are recorded to the application output files. The simulation's output is a number of files containing each cell's axon voltages for every simulation step. For the development of our scheme, a porting option utilizing data level parallelism of the Infoli simulator is used. Each core is assigned with entire cells, executing all compartments.

4.3 Depman Tool

4.3.1 Depman Operation

Depman is a runtime manager that controls the operation of a checkpointed application [10], in our case the Infoli simulator, by both handling DUE errors that would cause the application to suspend its execution and by reclaiming the wasted time due to the checkpoint procedure.

Depman was implemented in Python2.7 [39] and has minimum platform and application dependencies, so it is portable to any other platform and application.

The functionality of Depman tool is briefly explained in Figure 4.2.

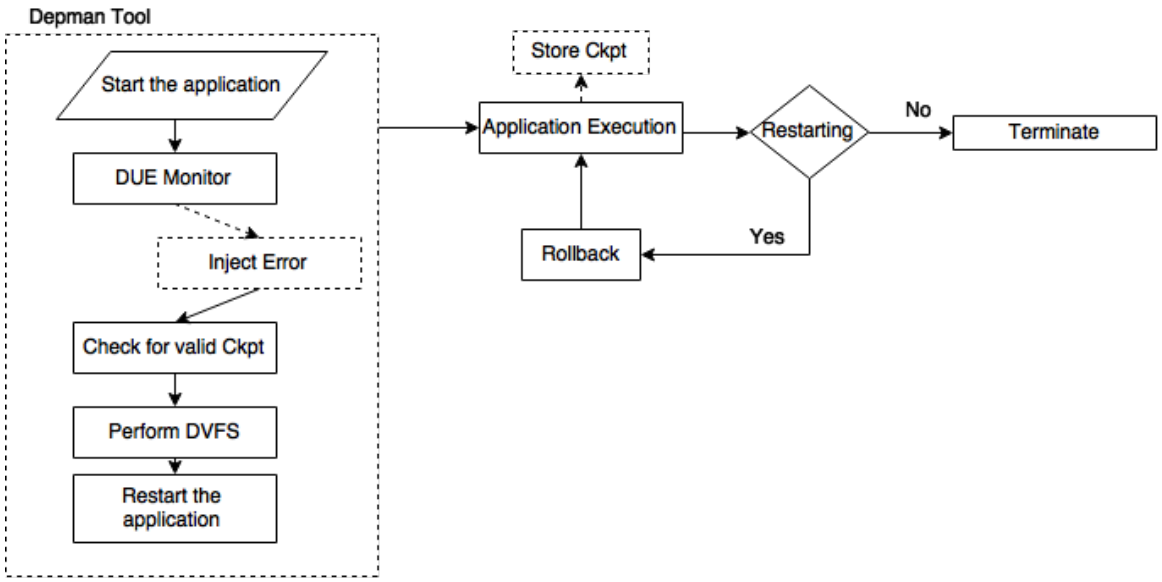


Figure 4.2: Depman tool operation diagram

First, Depman provides the appropriate input to the application and starts its execution. Also, it starts a thread that constantly monitors the application's output for errors. Then it waits until the execution of the application is stopped and checks if the application has ended normally or the DUE monitor has detected an error. For our scheme we self inject the application, but Depman is also operational with real time DUE errors. If an error has been detected Depman checks whether a valid checkpoint for the application has been stored and if it is we perform the DVFS module and then restart the application. The application itself

stores its state to checkpoint files and is capable to restart from a valid checkpoint during the restart procedure.

4.3.2 The Checkpoint Restart procedure

In order to enable fault tolerance, the Infoli simulator uses an application-level C/R method. Therefore, the vital points of the application state should be stored periodically, keeping in mind that the checkpoint files should keep storage requirements to a minimum. The stored points of the Infoli simulator consist of the data structures representing the state of each simulated neuron cell, such as dendrite and soma compartments of the cell and voltage or potassium levels for the axon. Also the simulation step, the number of cores and the number of cells are stored to the checkpoint files. That means the size of the checkpoint file is related with the number of cells of each core. The more cells manipulated by a core, the bulker the checkpoint file will be. The Infoli simulator operates in a number of simulation steps, allowing to select a Checkpoint Interval in simulation steps rather than time. Hence a checkpoint is taken at the beginning of a simulation step that divides the Checkpoint Interval with no remainder [10].

The Checkpointing of all cores is done simultaneously by calling a barrier function when a checkpoint needs to be taken. Furthermore each checkpoint is stored in double-buffered files that contain two sequential checkpoints at any time, so that a valid checkpoint exists even if an error occurs during the checkpoint procedure.

The Restart procedure regards the extraction of the neuron cells from the checkpoint files, the appropriate variables initialization and the continuance of the application from the correct simulation step. Since the checkpoint files are double buffered, cores should perform a communication scheme in order to determine the maximum recoverable simulation step that they can restart from. Each core broadcasts the maximum simulation step that it can restart from and then all cores restart from the minimum simulation step that was broadcasted. Additionally, the simulation could restart with a different number of cores. It can be restarted with less cores, because a number of cores is not responding or we want to

restrict the cores utilization by the application. Also it can be restarted with more cores, because we desire more parallelism for our application in order to achieve computational sprinting. This means that during the Restart procedure each core should identify the appropriate checkpoint files that should recover the neuron cell state from, depending on the number of cells and cores. In both cases the output files should be reconstructed for each core in order to be consistent.

4.3.3 Diagnostics and Self Injection module

The Depman tool is capable of detecting DUE errors that cause the application to stop and perform the appropriate countermeasures to restart the application from the appropriate simulation step. In our implementation scheme, we utilize the ProcessExit diagnostic, which monitors the stdout of the running process for failure messages indicating a Detected Unrecoverable Error (DUE) error [2]. In order to examine the performance of our experimental setup, errors during the runtime of the application are required. That is why a self injection module is used, periodically injecting the application with DUE errors. When a DUE error is injected the injection module calls the `process_line` function of the ProcessExit class containing a key word indicating program failure, the key word is relevant to the target platform. The `process_line` function detects the key word and stops the simulation by running a script to detect and kill the application. The time between errors for the injection module is user defined. In our setup we use both a steady TTF value and a Weibull distributed TTF to test our implementation. When we inject errors using the Weibull distribution the probability of error occurence is given by Equation 4.1, where Δt is the time interval between failures and $MTTFs$ is the user-specified $MTTF$ intervals.

$$P_s = 1 - e^{-\Delta t/MTTFs} \tag{4.1}$$

4.3.4 DVFS module

To enable observability and controllability of our application's performance we use the DVFS module before we restart our application after each DUE error. Our goal is to mitigate the performance variability, caused by DUE errors and the C/R procedure. So we calculate on the fly the timing noise of the application and adjust the voltage and frequency to a new level in order to reclaim the time overheads. This Section will present the main features of the DVFS module, as we use it for the SCC and the x86 platform. A more detailed explanation of the DVFS process will be displayed later in the respective Sections which exhibit the DVFS results for each individual implementation.

The control loop of the DVFS is presented in Figure 4.3

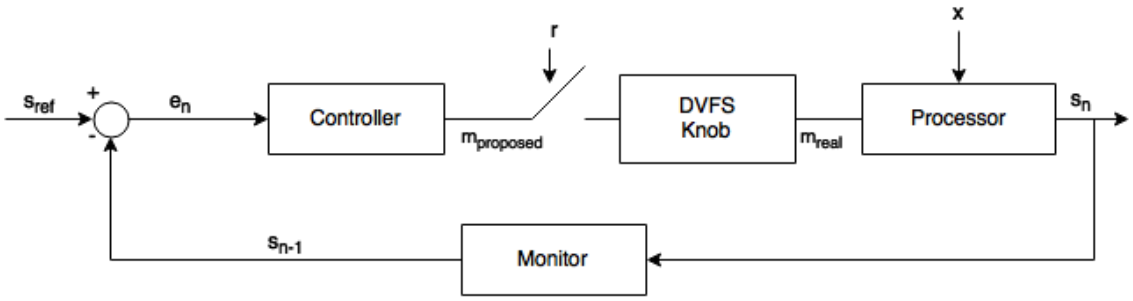


Figure 4.3: Block diagram for the DVFS closed loop implementation

As slack (s) we define the time that the application has fallen back due to the invocation of RAS mechanisms, the C/R procedure, and the restart process that was the result of a DUE error. We want to succeed the slack convergence to s_{ref}, which in our case is zero and indicates the target slack we want to achieve when the application is terminated succesfully. The **Monitor** measures and updates the slack after each DUE error. The parameter n indicates the number of the restart operations that have occured throughout the execution of the application. Then the **Controller** reacts to the value of $e_n = s_{ref} - s_{n-1}$ and proposes a frequency multiplier in order to reclaim the generated slack. Frequency multipliers(m) reflect the DVFS configurations of the processor. So after the Controller proposes a frequency multiplier, the DVFS Knob chooses the nearest available frequency multiplier for the processor and applies the DVFS alteration.

Afterwards the execution of the **Processor** is continued to the new frequency/voltage configuration.

The timing noise indicated by x, is the total time the application is delayed because of the checkpoint restart procedure and contains the checkpointing time and the time for the application to recover from a previous simulation step during the restart procedure. The processor's slack is given as input to the Monitor before it updates the slack containing all timing noise overheads as will be discussed in Section 4.4. Finally, the r parameter is used to tune how often the DVFS Knob performs a voltage/frequency alteration, in a number of detected errors. That means a value of r=1 will perform DVFS changes for every DUE error, while a value of r=2 will perform DVFS changes every two errors and so on.

4.4 Timing Noise Overheads

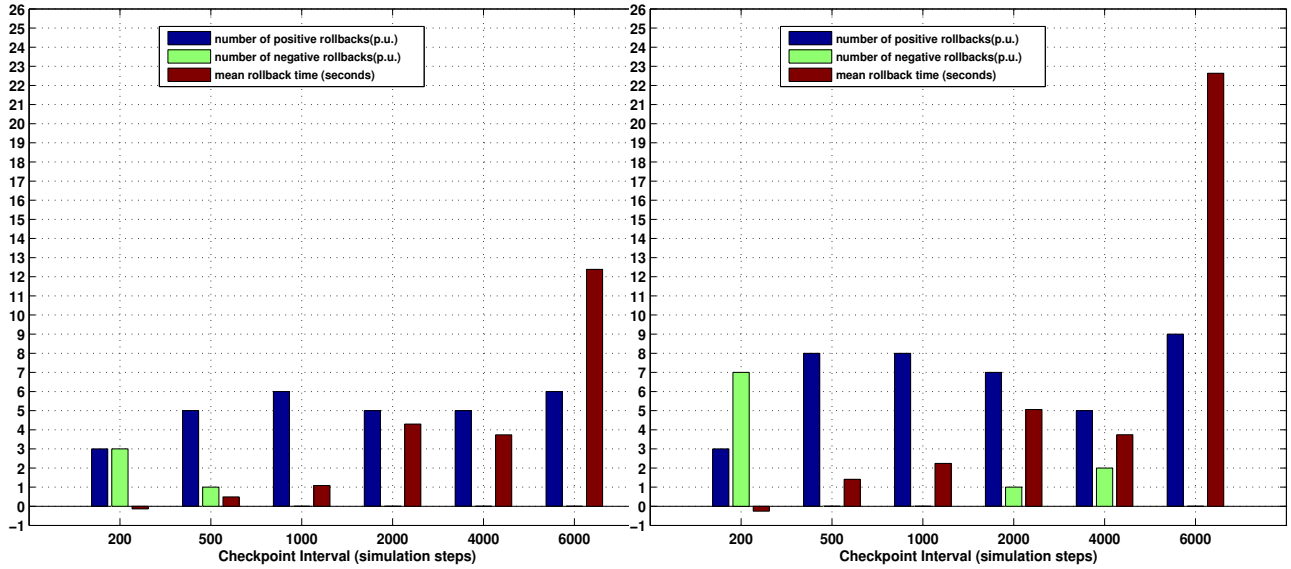
The timing noise imported to the execution of the application due to C/R comes as a consequence to the following factors.

- First of all, the *Rollback Time* (T_r), which represents the lost computation time that needs to be performed again when the application restarts from a previous simulation step because of a detected error.

  In order to recognize the timing noise of the application rollback we need to recognize the exact computation waste time. To achieve that we need to calculate the time difference between the creation of the restarting checkpoint file and the time that the execution of the application was suspended. The checkpoints are stored in double buffered files that contain two sequential checkpoints at any time, so that there is a valid checkpoint for all nodes even if an error occurs while the checkpoint process was ongoing. As a result to this technique we cannot use the checkpoint file itself to measure the time difference between the checkpoint and the error that caused the process to terminate. We need to know the exact time that each checkpoint was stored. That is why when a checkpoint is stored, we also create a file named after the simulation step of the current checkpoint taken so that we can use this file to calculate the T_r

overhead.

The values of T_r vary in relation to *Checkpoint Interval*. As *Checkpoint Interval* is increasing, T_r is increasing too. In our scheme, using self injection to test and benchmark our application setup, we observe the results as illustrated in Figure 4.4 for four different grid sizes, executing in a number of cores indicated by the second grid size factor. These results concern the SCC platform.



(a) grid size 8x24

(b) grid size 8x48

(c) grid size 16x24

(d) grid size 16x48

Figure 4.4: Rollback Time characterization for the SCC

As a result we notice that increasing the *Checkpoint Interval* leads to greater T_r values.

Also, in Figure 4.4 the number of positive and negative T_r is displayed. The reason that we notice negative T_r values is because of the self injection module that we use.

- First, by the time we inject an error to the application *scc diagnostics* notice the injected error and need to terminate the process. We take a timestamp of the process termination at that time, but actually there is some more time needed for us to manually kill the process with a termination script.
  During this time, maybe a new checkpoint is stored and that results to a negative T_r value.
- Another reason that we may observe a negative T_r value is the I/O delays of the SCC platform [31]. So even if the checkpoint was stored before an error occurred, the time to sync the file to our `/shared` folder may result to a negative T_r.

However, if a negative T_r value occurs it is minimal compared to the overall slack of the application restart process and we ignore it without any performance loss for our scheme.
As we can also observe from Figure 4.4, when the *Checkpoint Interval* increases the possibility for a negative T_r reduces. In our measurements we used *Checkpoint Interval* values of 10^3 *Simulation Steps* or more so that we eliminate as far as possible such phenomena.

- Next, the *Time to Restart* (TTR) which is the time to perform the available counter-measures after the simulation stopped its execution, and restart the application. In our experimental setup, after the execution is terminated because of an error we check whether there is a valid checkpoint that the simulation could restart from. Then, if it is, we call our *DVFS* module to update the slack of our application and change the Voltage/Frequency values, if needed, to reclaim the waste time due to the restart procedure. That means that TTR includes both the time overhead for the *DVFS* as was presented in Chapter 3 and the time to test if the application has a previous valid state that could restart from and restart the application.
  In order to calculate the timing noise of TTR we measure the time difference from

the time that *scc diagnostics* notice the error, since the time that the application was successfully restarted.

For the SCC platform TTR overhead is remarkable compared to an x86 commercial platform because of the great *DVFS* overhead when a voltage alteration is performed.

- In addition, the *Repair Time* (T_{repair}) overhead, which is the actual time for our application to restore its state from the checkpoint files and continue the execution from the appropriate simulation step. This time overhead is not considered crucial for our application if the number of active cores is the same before and after the restart procedure, since the time to extract the neuron state data structures from the checkpoint files and restart from the correct simulation step is little. However, if the application restarts with a different number of cores the output files of the Infoli simulator should be reconstructed to provide a consistent output and T_{repair} becomes significant.

  The simulation continues after all nodes have restored their state from the checkpoint files. That means that the T_{repair} is the same for all nodes. After the state restoration,only the node with $core_{id} == 0$ publishes a file in the `/shared` folder, for the SCC , or the designated folder, for the x86 platform, that reports the T_{repair} time of the application, to reduce the I/O traffic. The *DVFS* module can then retrieve the T_{repair} of the application when is called and update the slack of the application including that value.

- Finally, the time overhead introduced because of the checkpoint procedure $T_{\text{checkpointing}}$. Inevitably, the application needs to store its state to a checkpoint file periodically, as it is defined by the *Checkpoint Interval*. This procedure causes extra overhead to the execution. In our simulation $T_{\text{checkpointing}}$ is of the order of miliseconds because the neuron state data structures stored do not occupy a great amount of memory. Still checkpointing may cause a scalability and performance barrier in high performance computer systems as noted in [23].

  The checkpointing of all nodes is done synchronously by setting a barrier before and after the checkpoint is taken. Even though blocking communication and barriers introduce more overhead than non-blocking, the Infoli simulator already utilized blocking

techniques so there is no extra overhead. This means that although we synchronize the checkpointing procedure for all cores both before and after a checkpoint is taken, we see no performance degradation, because the Infoli simulator already uses blocking communication schemes to exchange information between the cells of each core. Since the checkpoint procedure is done at the same time for all nodes, same as before the node with $core_{id} == 0$ publishes a file in the `/shared` folder, for the SCC, or the designated folder, for the x86 platform, that indicates $T_{\text{checkpointing}}$. So when a fault happens and *DVFS* module is called $T_{\text{checkpointing}}$ is retrieved from the file and is multiplied by the number of checkpoints that took place between the restarting simulation step and the previously restarted simulation step (which maybe zero for the first fault).

Chapter 5
Reclaiming Timing Noise

5.1 Introduction

This chapter will present the performance of the depman tool and especially the DVFS module, for both the SCC and the x86 platform. For each individual implementation the DVFS procedure will be analyzed, and the mitigation of the performance variability will be presented and discussed. First, the SCC impementation will be presented and then two similar implementations for the x86 will be introduced. One operating as the SCC implemention, but with more frequency configurations so that we can reduce or increase the frequency levels more frequently, and the other operates in higher frequency levels only for as much time needed to reclaim the slack and then resets frequency to its default value, whether a fault has been detected or not.

5.2 SCC Results

As mentioned before (Section 3.3), a voltage and frequency alteration for the SCC platform is a time consuming procedure, that adds additional time overhead to the execution of our application. That is why we focus on two different DVFS configurations and switch between them depending on whether we have a positive or a negative slack, so that we make a frequency/voltage change as rarely as possible. In the first configuration the SCC operates at a frequency of $533MHz$ and $0.8Volts$ while in the second one the frequency is equal to $800MHz$ and the voltage is $1.1Volts$. Also we perform no DVFS change during the first failure where we only have the T_r and $T_{checkpointing}$, in order to make less alterations.
When we start the execution of the Infoli simulator the SCC is configured at $533MHz$ and

$0.8Volts$, which is the default mode for our case, and the DVFS module operates as shown in Figure 5.1.

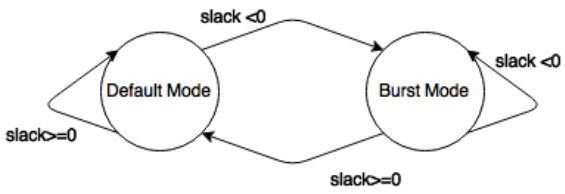


Figure 5.1: State diagram for the SCC DVFS module for slack reclaiming

The application starts its execution at the *Default Mode*. When a DUE error is detected and the application stops then the DVFS module is called and the total slack of the application is calculated as :

$$\begin{aligned} s_{new} = s_{previous} - timeOverheads + \\ + (lastTTF \times curFreq - lastTTF \times defFreq) \end{aligned} \tag{5.1}$$

The time overheads are calculated as mentioned in Section 4.4 and refer to $T_r, TTR, T_{\text{repair}}$ and $T_{\text{checkpointing}}$, while the last factor of the equation depicts the time difference for the execution of our application in case we are running with a different frequency multiplier than the default one. That means we can calculate the time that we have reclaimed from the last failure till the current one. On the SCC platform we only have two execution modes, the default and the burst mode. That means the last factor of Equation 5.1 only shows the reclaimed time during the last TTF. However, as we will see later for the x86 platform, where we have lower frequency configurations than the default one, it can also show the fall back of our execution.

So if the slack is negative that means the application's execution time has fallen back and we need to speedup by operating at *Burst Mode* until the slack is reclaimed. Of course, before we make an alteration the parameter r should be taken into account to see if we must do a frequency change at that time before the application is restarted. We must note here that if the application is executed without any faults at all, then the overhead produced by the checkpointing procedure, $T_{\text{checkpointing}}$, will not be reclaimed, since the DVFS

module will never be called, but this overhead is minimal for the currently inspected case study. Nevertheless, we can modify the DVFS module so that it is triggered every time a checkpoint storage action is taking place, in order to avoid such conditions if needed.

First we test our setup on multiple grid sizes and numbers of cores, for a Checkpoint Interval of 2000 and 4000 and stable TTF values of 120 and 180 seconds. The number of active cores is indicated by the second factor of the grid size.

In Figure 5.2 we illustrate our results. For each grid size we depict the execution time if the application was running without any faults and checkpointing, the total time when we have faults and we apply DVFS changes and the time that would be needed if we had faults but the DVFS module was inactive.

As we can see from our results the DVFS implementation outperforms the execution with faults and no DVFS. That means that the cycle overhead introduced to our application is reclaimed. The only case that we see the DVFS implementation to draw back is for grid size 12x12, TTF 180 seconds and Checkpoint Interval of 4000 simulation steps. The reason is that,in this case, there is not enough time to reclaim the cycle noise. The first DVFS change occurs after the second detected failure which happens near the application termination. So adding the time overhead produced by the voltage alteration we reduce even more the performance of our execution. However, for all the other cases the results are satisfactory and there are cases that we achieve the convergence of our execution time to the time without faults, even though we alter between only two frequency configurations. Also there are cases that DVFS is even better from the default execution. That is because the DVFS module is called after a DUE error and makes a decision about the frequency configuration taking into account the value of the slack. The slack however may have a negative value near to zero. The DVFS module recognizes that the application needs to perform faster so it keeps running on *Burst Mode* reclaiming the total slack and performing even faster than we would expected. Such cases, of largely positive slack effectively correspond to energy loss and must be avoided when possible. We will introduce an implementation for the x86 architecture that avoids a positive overall slack at all times later.

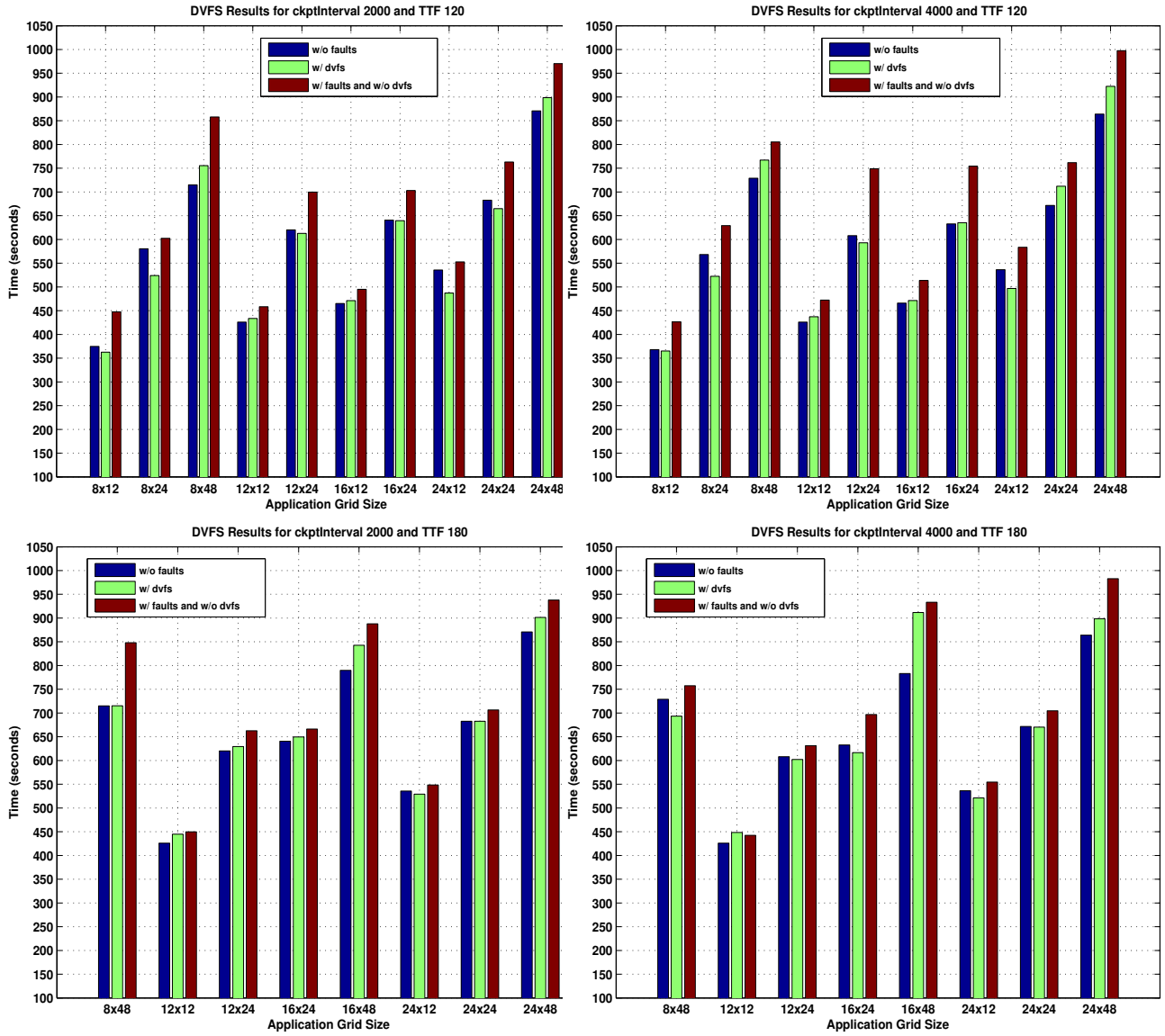


Figure 5.2: DVFS results for multiple grid sizes and active cores, Checkpoint Interval of 2000 and 4000 simulation steps and Time to Failure 120 and 180 seconds

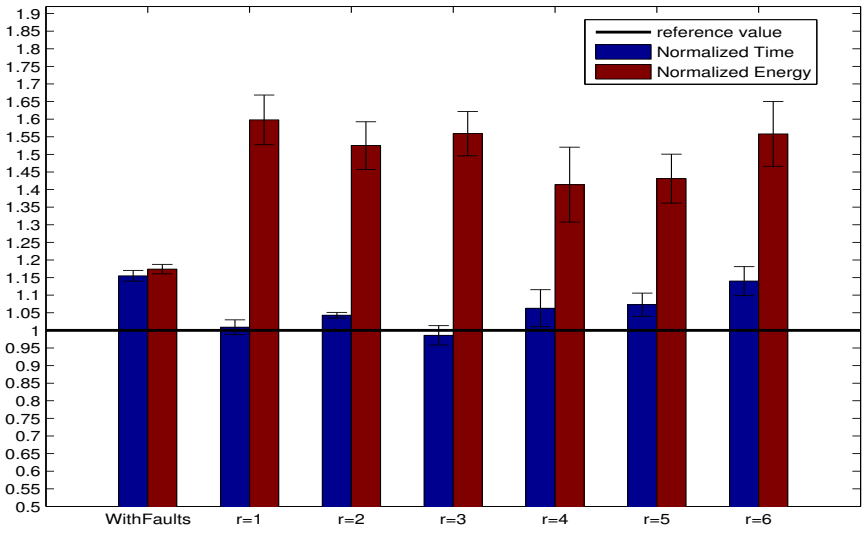
In order to test the DVFS performance even further, we select a grid size of 16x96 which executes for a long period of time, so it is ideal for our experiments, and we select to run with 24 active cores. Our goal now is not only to evaluate the time performance of our scheme but also the energy consumption for different values of the r parameter for stable and Weibull distributed TTF values. For our energy measurements we use the same tactic as in Section 3.3. We forked a process that constantly records the current and voltage of the SCC platform every 0.3 seconds throughout the execution of the application and then we used the trapezoidal integration to calculate the energy consumption.

First the results concerning stable TTF of 122 seconds will be presented for three different

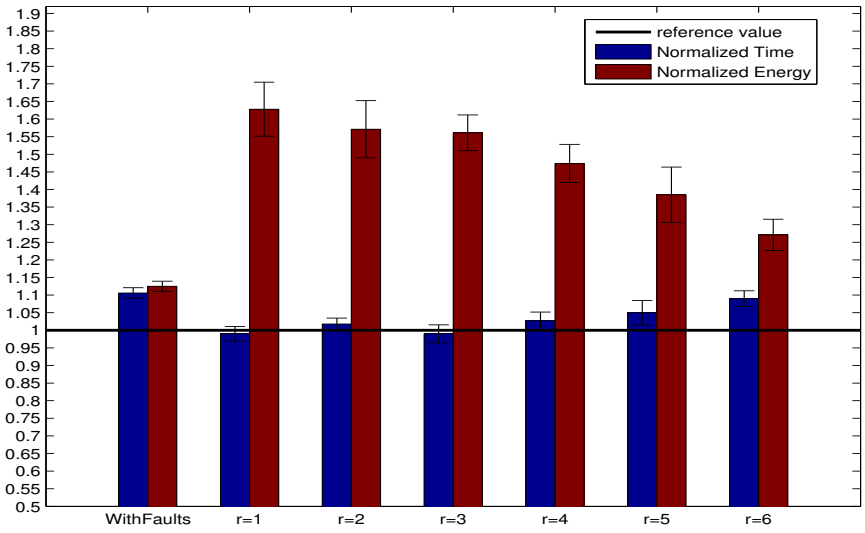
Checkpoint Intervals of 1000,2000 and 4000 and for multiple r values ranging from one to six. As a reference value for our graphs we select the execution time without any faults and the checkpoint procedure, which is 823 seconds of execution time, with energy consumption of 29210 Joules. The y-axis represents the Normalized Time/Energy Overhead, which means that a value of 1.1 for the execution time refers to a value 1.1×823 seconds, and the same with energy but multiplying with the energy reference value.

From Figure 5.3 it is clear that the DVFS module efficiently reclaims the time overheads of the Checkpoint Restart procedure. However, this comes with the cost of more energy consumption. For an application like the one tested that runs for a long period of time the DVFS module is capable of absorbing the cycle noise and converging to the reference value, especially when r=1 the DVFS alteration is performed every time that is needed. On the other hand, by inreasing the value of r seems to generally result in less energy waste with a little or no performance loss. We can not keep increasing r though, because then the slack becomes so massive that there is not enough time to be reclaimed. In our scheme, if r exceeds the value of six the performance of the DVFS is reducing. Furthermore, when the DUE errors are randomly injected there is no guarantee that a great r value will manage to perform satisfactory.

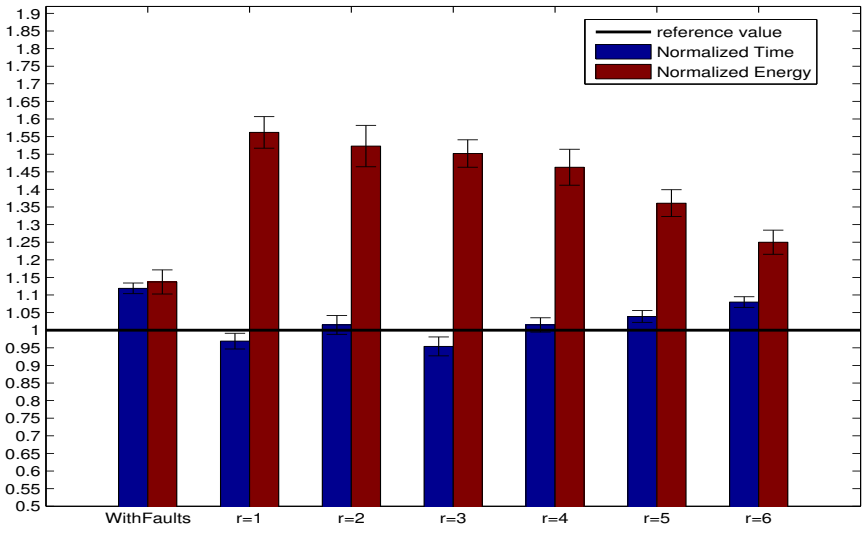
That is why we also test our setup for Weibull distributed TTF values for r=1,2,3 and $MTTFs$ 120 seconds .



(a) Checkpoint Interval =1000,TTF = 122

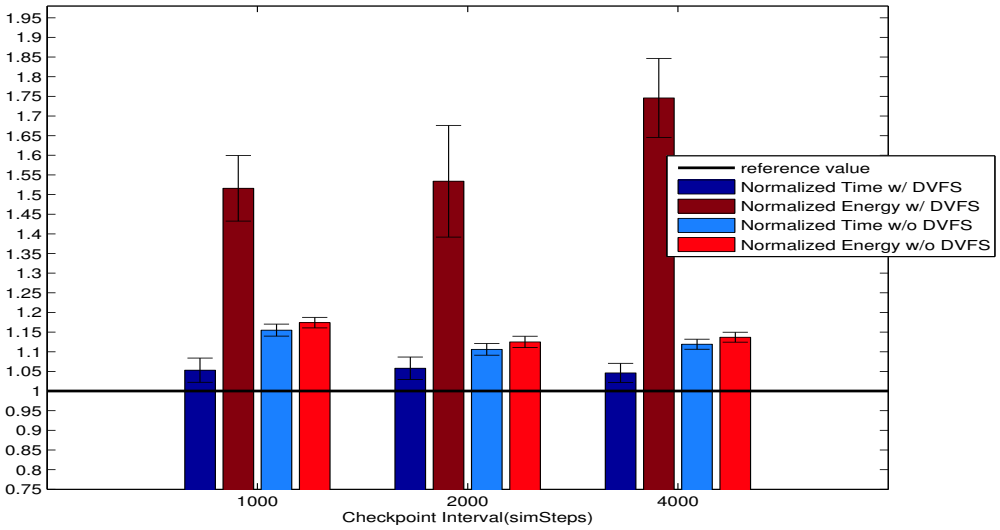


(b) Checkpoint Interval =2000,TTF = 122

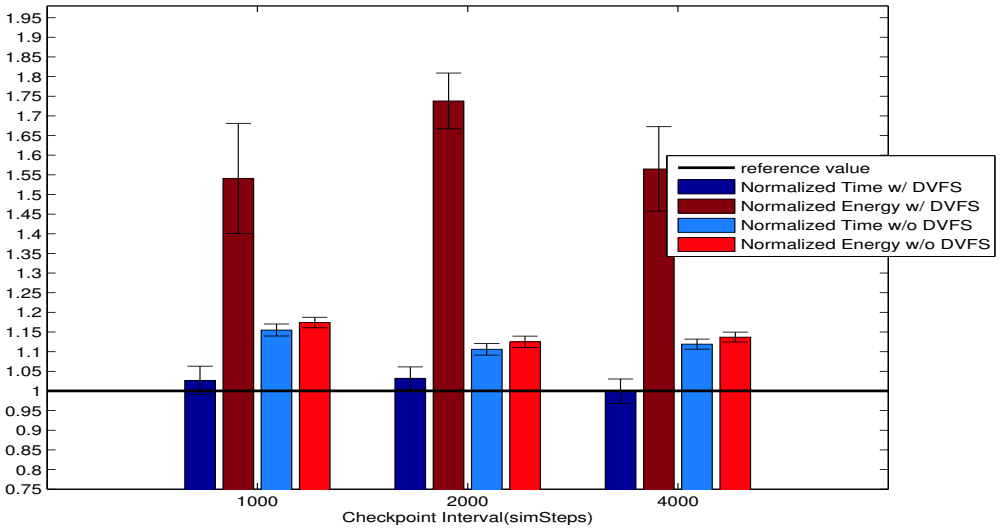


(c) Checkpoint Interval =4000,TTF = 122

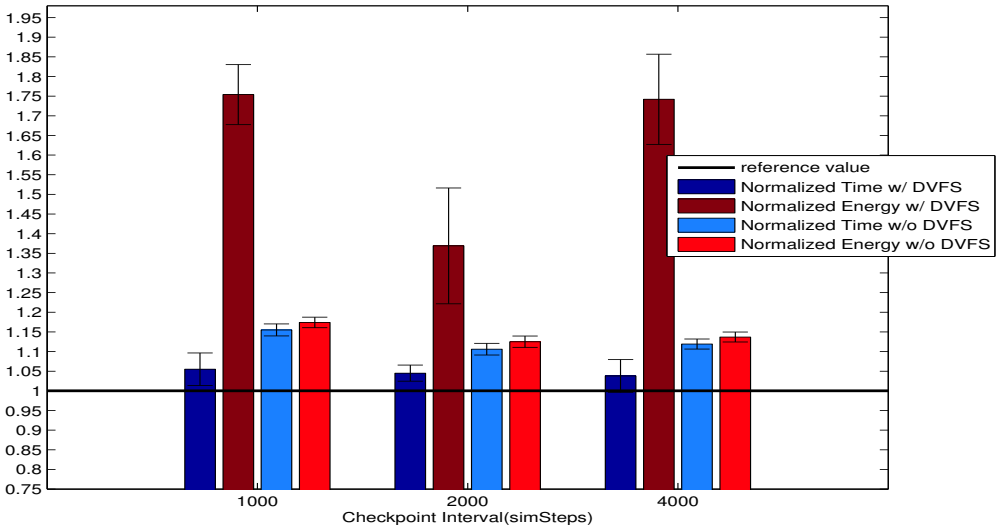
Figure 5.3: Time and Energy results for the SCC platform, for TTF 122 seconds
Time Reference: 823 seconds and Energy Reference: 29210 Joules



(a) r=1



(b) r=2



(c) r=3

Figure 5.4: Time and Energy results for the SCC platform, Weibull distributed TTF values
Time Reference: 823 seconds and Energy Reference: 29210 Joules

The first two columns of each bin of Figure 5.4 represent the execution time and the energy consumption of the application, when the DVFS module is activated, respectively, while the next two represent the time and energy if we run the application with a TTF value of the MTTF we calculated during the previous execution so we can compare our results. Our reference value is the no-fault and no-checkpointing execution of the Infoli simulator for a grid size 16x96 and 24 active cores, which are 823 seconds and 29210 Joules.
Again the results indicate that the DVFS modules can adjust to random TTF with minimum performance cost as far as execution time is concerned. On the other side it seems that the energy consumption can be greater when faults appear randomly, but that largely depends on the failure rate.

5.3 x86 Results

Before we present the results for the x86 implementations we must note the way that Energy was calculated. From the Intel's $2nd$ Generation Datasheet [40] we can see the maximum and minimum voltage values of our processor. Assuming a linear relationship between the Voltage and the CPU frequency we determined that Voltage and frequency are related with the following equation:

$$V(f) = 5.83 \times 10^{-7} \times f + 0.184,\ f\ in\ KHz \tag{5.2}$$

As a result, we can use Equation 5.2 to calculate the Voltage values for each frequency configuration. During the execution of our application we keep track of the time that each frequency configuration was used so that we can measure the Energy Consumption using Equation 5.3.

$$E = c \times f \times V^2 \times Dt \tag{5.3}$$

Of course we do not know the value of constant c but that is no boundary for us to calculate

the Normalized Energy consumption.

5.3.1 First implementation

The x86 platform, in contrast with SCC, has the advantage that frequency alterations are performed rapidly. So we can perform alterations more frequently without any appreciably time overhead and between more frequency configurations. The state diagram describing the function of our first implementation for the DVFS module is given in Figure 5.5.

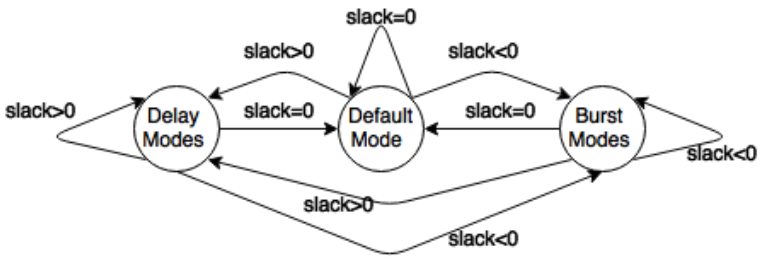


Figure 5.5: State diagram for the first implementation of the x86 DVFS module for slack reclaiming

As we can see from the state diagram in this case we have *Delay, Default and Burst Mode*. That means we have frequency configurations varying from lower to greater values than the default one. Precisely, we have the following frequency configurations: `800MHz, 1000MHZ, 1.2GHz, 1.4GHz, 1.6GHz, 1.8GHz and 2.0GHz`, where `1.2GHz` is defined as the default frequency level. So we have four *Burst Mode* configurations defined by the frequency levels above the default one and two *Delay Mode* configurations defined by the frequency levels lower to the default one. When the value of slack is negative, meaning that the execution of the application is delayed due to timing noise, then we can switch to the appropriate *Burst Mode* configuration. On the other hand, when we have a positive slack, meaning that the application has overtaken the time overheads and is executing even faster than it is needed we can switch to the appropriate *Delay Mode*. We switch to *Default Mode* when the value of slack is zero. In order to update the value of slack we use the Equation 5.1 as described in the previous Section. Having calculated the slack value, we can determine the appropiate *Mode* in which we should continue the execution of the application with the

following equation

$$newFreq = \frac{-s + defFreq \times MTTF}{MTTF} \tag{5.4}$$

With Equation 5.4 we can determine the frequency value that we should continue the execution of our simulation, in order to reclaim the application fall back time before the next expected DUE error, that will cause the application to stop. That means we should calculate the *Mean Time to Failure (MTTF)* before each restart procedure, so that our DVFS module is adaptive to errors.

After the *newFreq* value is calculated we decide the actual frequency level that we must switch to, by finding the closest frequency available that is equal or lower to *newFreq*, and then we make the alteration and then restart the application.

In order to test our scheme we executed the Infoli simulator for three different Checkpoint Intervals (1000,1500,2000), grid size 12×16, Weibull distributed Time to Failure with $MTTFs$ value of 20 seconds. Also, for the same Checkpoint Intervals we executed the Infoli Simulator for TTF of 20 seconds with the DVFS module deactivated so that we can compare the time and energy values. Furthermore, since the frequency and voltage alterations for the x86 platform consumes minimal time we only tested our implementation for $r = 1$. The results are as presented in Figure 5.6.

As we can see from the Figure, the DVFS execution outperforms the execution without the DVFS module, in cost of energy. In comparison with the SCC implementation we can observe that our results converge more accurate to the reference value than it happened before. The main reason for this precision is the less time overhead introduced by the voltage alteration. Also we must note that the error values are less here than there were for the SCC implementation, which means there is less variability to our measurements.

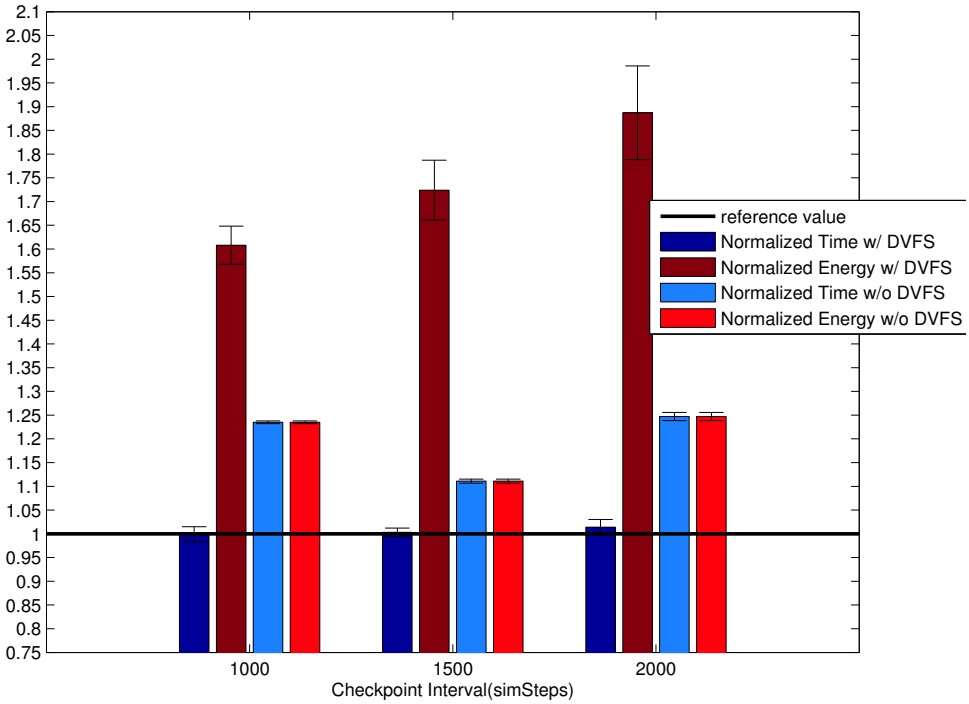


Figure 5.6: Implementation 1: Time and Energy results for the x86 platform, Weibull distributed TTF values
Time Reference: 578 seconds and Energy normalized, based on P $\propto$ f $\times V_{dd}^2$

5.3.2 Second Implementation

We already stated that frequency/voltage alterations on an x86 platform are performed, almost, instantly providing us the capability to perform frequency changes more frequently and between more frequency levels. Another advantage of the x86 platform is that we can implement a frequency change even while the simulation is ongoing.

So far, the implementations we presented have the disadvantage that after an error is detected and the frequency level is set to a new value then the application is executed on the new frequency until a new error, that will cause the DVFS module to reconsider the frequency and make an alteration if needed, is detected. Hence, the execution of the application could be even faster in comparison with the reference case, that is if no error was detected, because it executed on higher frequency levels for more time than it was needed. As was presented by the previous results, we can see that such phenomena are rare when faults follow a statistical distribution. However, it is likely that only one fault is detected during the execution and the execution time would not be able to converge to the reference

value.

As a consequence, we implemented a DVFS module that overtakes such incidents. It's function is described in the following state diagram of Figure 5.7.

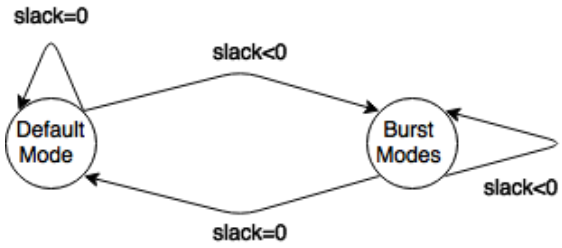


Figure 5.7: State diagram for the second implementation of the x86 DVFS module for slack reclaiming

Here we only have two different opperating *Modes*. There is no need for *Delay Mode*, since we never have positive slack in this case. As a result, we define `800MHz` as the default operation value and all other frequency configurations represent the different *Burst Mode* configurations. The value of slack is updated with Equation 5.1 and the frequency level is determined as in the first implementation, using Equation 5.4.

The difference is when a frequency alteration is performed, we also fork a process which takes as a parameter the total time that the application should execute in *Burst Mode* so that the slack is reclaimed, and after that time has passed it restores frequency to its default value (*Default Mode*). This time is calculated as :

$$reclaimingTime = \frac{-s}{determinedFreq - defFreq} \tag{5.5}$$

So after we determine the theoritical expedient frequency level that we must continue our execution, we determine the actual frequency value that is available for us and then calculate the time that the application needs to run on this frequency, in order to reclaim the time overheads. Therefore, every time the DVFS module is called we should certify whether or not the forked process has restored the frequency value to default. We can not compare the TTF value with the *reclaimingTime* value because race conditions can occur this way,

if for instance TTF is equal to *reclaimingTime* it is not certain that the frequency alteration has already occured or not. Thus, we use the *poll* method contained in the `Python's subprocess module` [39]. This method returns *None* if the process has not terminated yet, or the termination code otherwise. So we *poll* the forked process and in case *None* is returned we kill the process and calculate the remaining slack using the Equation 5.1. Whereas if the process has terminated normally we reset the slack to zero value and calculate the new slack using Equation 5.1. Consequently, this implementation never leads to positive slack values and only boosts the execution for as much time as needed to reclaim the slack. In order to evaluate the implemented scheme we executed the Infoli simulator with grid size 10×14 and all the other configurations remain the same as the previous implementation. The results are as depicted in Figure 5.8.

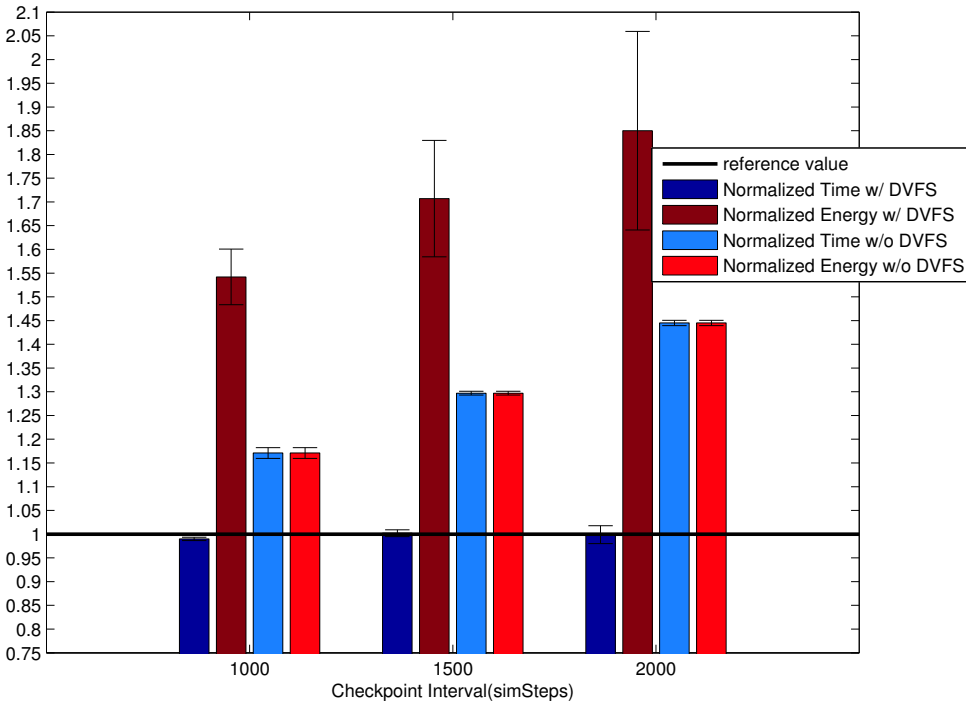


Figure 5.8: Implementation 2: Time and Energy results for the x86 platform, Weibull distributed TTF values
Time Reference: 494 seconds and Energy normalized, based on P $\propto$ f $\times V_{dd}^2$

The results indicate a precise convergence to the reference value as far time is concerned. Again we can observe that slack reclaiming leads to greater power consumption. However, this implementation massively reduces the wasted power due to execution sprinting, since it

sprints the execution for as much time as needed and then restores frequency to its default value.

Chapter 6
Conclusions and Future Work

6.1 Conclusions

This work introduced an approach to enable fault-tolerance to an application without violating any time constraints. For the purposes of this development the Infoli simulator has been employed as the target application, upon which a periodic application level C/R scheme has been adopted. Our scheme was tested and evaluated both on the Single-Chip Cloud Computer and on an x86 commercial platform, to highlight the portability of our work.

The implementation is controlled by the Depman tool which monitors the execution of the application by parsing the system's output for DUE errors. Depman needs to restart the application, until it is terminated normally, when an error occurs by checking whether or not there is a valid checkpoint for the application to restart from. It also sets the voltage and frequency to the appropriate level so that the time overheads introduced by the C/R procedure are reclaimed. In this concept we quantified the total time overhead, that is induced, and controlled it with the DVFS module in a closed-loop. We made three different versions of the DVFS module. One concerning the SCC platform and has the ability to change between Default and Burst Mode depending on the value of slack. The other two concern the x86 platform. The first of them has three different execution Modes, Default, Delay and Burst Mode, and has the ability to change between each one of them depending on the value of slack. The second has two Modes, Default and Burst, but the frequency alteration is performed only for the time needed so that the total slack is reclaimed and then the frequency is set back to default resulting less energy overhead.

In order to test our scheme the error injection module was utilized. That way we injected errors to the execution of our application using static or Weibul-destributed injection scenar-

ios. For the evaluation of this work we measured the execution time and energy consumption throughout various experiments and conclude that the time overheads are greatly reclaimed, and in case of x86 almost converge to the reference values, in cost of more energy consumption.

6.2 Future Work

Copious modifications can be made to both Depman and the DVFS module itself in order to achieve better efficiency or even use it for a different purpose. In this thesis, the burden of our views was about the wasted time restoration caused by the C/R procedure. However, the created scheme can be used vasriously. These potential future work is presented in this section.

- First of all, Depman can be used as the main component to utilize C/R techniques to multiple machines comprising a distributed system. This extension demands the consistency of data between all machines and the sunchronization of checkpoints. So Depman can be used to each of the many-core nodes of the system to provide fault-tolerance and network techniques should be employed to achieve the communication of the nodes.

- Secondly, Depman can be modified not only to work on a closed-loop, but during the execution of the application reacting to numerous events. That way many modules, such as the DVFS module presented here, can be incorporated to Depman and used by the programmers. For example the DVFS module may be called every time a checkpoint is taken or a module that applies checkpoint merging is called.

- Moreover, besides the DVFS technique parallel sprinting may also be applied. Both Depman and Infoli simulator are build in a way that can exploit the use of computational sprinting as another way of slack reclaiming.

- The DVFS module itself can be used independently not only for the slack reclaiming of C/R, but for other fault-tolerance schemes. Also it can be used by the developers in

order to accelerate the execution of their process or even slow it down if energy saving is a major concern. In this direction the DVFS and a module performing parallel sprinting can be implemented as a library, which the programmer may use to adjust the execution speed of his process taking into account the thermal capacitance of the platform.

- What is more the DVFS module can be exploited by web servers, in a way that the performance is boosted during rush hours when the traffic is high so that customers observe less time delay in their services and then performance is degraded when traffic is low so that energy is saved.

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, pp. 305–316, Sept 2005. 5

[2] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. 5, 29

[3] Y. Cao, J. Velamala, K. Sutaria, M.-W. Chen, J. Ahlbin, I. Sanchez Esqueda, M. Bajura, and M. Fritze, "Cross-layer modeling and simulation of circuit reliability," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 33, pp. 8–23, Jan 2014. 5

[4] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational sprinting on a hardware/software testbed." v, 4, 8, 9

[5] D. Rodopoulos, F. Catthoor, and D. Soudris, "Tackling performance variability due to ras mechanisms with pid-controlled dvfs," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2014. v, 1, 4, 9, 10

[6] *The SCC Programmer's Guide-Revision 0.75*. Boston,MA,USA: Intel Corporation, 2010. v, 13, 15

[7] G. Chatzikonstantis, "Energy aware mapping of a biologically accurate inferior olive cell model on the single-chip cloud computer," bachelor thesis, National Technical University of Athens, September 2013. v, 25, 26

[8] A. Dixit, R. Heald, and A. Wood, "Trends from ten years of soft error experimentation," *System Effects of Logic Soft Errors (SELSE)*, 2009. 1

[9] D. Hardy, I. Sideris, N. Ladas, and Y. Sazeides, "The performance vulnerability of architectural and non-architectural arrays to permanent faults," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 48–59, Dec 2012. 1, 5

[10] A. Mavrogiannis, "On the dependability of transient neuron simulations," bachelor thesis, National Technical University of Athens, June 2014. 2, 6, 27, 28

[11] D. Rodopoulos, G. Chatzikonstantis, A. Pantelopoulos, D. Soudris, C. De Zeeuw, and C. Strydis, "Optimal mapping of inferior olive neuron simulations on the single-chip cloud computer," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pp. 367–374, July 2014. 2, 25

[12] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, *et al.*, "A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling," *Solid-State Circuits, IEEE Journal of*, vol. 46, no. 1, pp. 173–183, 2011. 2

[13] S. Mukherjee, J. Emer, and S. Reinhardt, "The soft error problem: an architectural perspective," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 243–247, Feb 2005. 5

[14] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge, "Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits," *Proceedings of the IEEE*, vol. 98, pp. 253–266, Feb 2010. 5

[15] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated application-level checkpointing of mpi programs," *ACM Sigplan Notices*, vol. 38, no. 10, pp. 84–94, 2003. 6

[16] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, p. 494, IOP Publishing, 2006. 6

[17] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. 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*, pp. 1–9, IEEE, 2008. 6

[18] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, May 2009. 6

[19] J. P. Walters and V. Chaudhary, "Application-level checkpointing techniques for parallel programs," in *Distributed Computing and Internet Technology*, pp. 221–234, Springer, 2006. 6

[20] L. M. Silva and J. G. Silva, "System-level versus user-defined checkpointing," in *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pp. 68–74, IEEE, 1998. 6

[21] J.-C. Laprie, "Dependable computing and fault-tolerance," *Digest of Papers FTCS-15*, pp. 2–11, 1985. 7

[22] D. J. Cross, "Power Efficiency Revolution for Embedded Computing Technologies (perfect)," tech. rep., Defense Advanced Research Projects Agency (DARPA), 2010. Available: `http://www.darpa.mil/program/power-efficiency-revolution-for-embedded-computing-technologies`. 7

[23] X. Yang, Z. Wang, J. Xue, and Y. Zhou, "The reliability wall for exascale supercomputing," *Computers, IEEE Transactions on*, vol. 61, pp. 767–779, June 2012. 7, 34

[24] J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova, "Evaluation of the intel x00ae; core x2122; i7 turbo boost feature," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 188–197, Oct 2009. 8

[25] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin, "Computational sprinting," in *Proceedings of the 2012 IEEE*

18th International Symposium on High-Performance Computer Architecture, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012. 9

[26] J. Gjanci, "On-Chip Voltage Regulation for Power Management in System-on-Chip," Master's thesis, B.S. University of Illinois, Chicago, 2006. 11, 12

[27] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pp. 123–134, Feb 2008. 11, 12

[28] *SCC External Architecture Specification (EAS) Revision 1.1.* Intel Corporation, 2010. 13

[29] T. M. (IL) and R. van der Wijngaart (SSG), *RCCE: a Small Library for Many-Core Communication.* Intel Corporation, 2010. 13

[30] M. P. Forum, "Mpi: A message-passing interface standard," tech. rep., Knoxville, TN, USA, 1994. 14

[31] R. Bakker, M. Van Tol, and A. Pimentel, "Emulating asymmetric mpsocs on the intel scc many-core processor," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pp. 520–527, Feb 2014. 21, 33

[32] V. Pallipadi, "Enhanced Intel SpeedStep® Technology and Demand-Based Switching on Linux*," tech. rep., Intel, 10 2010. 22

[33] "The linux kernel documentation." `https://www.kernel.org/doc`. 22, 23

[34] J. Hopper, "Reduce Linux power consumption," tech. rep., IBM, 09 2009. 22

[35] "Texas instruments, dvfs user guide." `http://processors.wiki.ti.com/index.php/DVFS_User_Guide`. 23

[36] J. R. D. Gruijl, P. Bazzigaluppi, M. T. de Jeu, and C. I. D. Zeeuw, "Climbing fiber burst size and olivary sub-threshold oscillations in a network setting," *PLoS computational biology*, p. 8(12):e1002814, 2012. 25

[37] P. Bazzigaluppi, J. R. D. Gruijl, R. S. V. D. Giessen, S. Khosrovani, C. I. D. Zeeuw, and M. T. D. Jeu, "Olivary subthreshold oscillations and burst activity revisited," *Frontiers in neural circuits*, no. 6, 2012. 25

[38] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, p. 117(4):500, 1952. 25

[39] "Python software foundation. python language reference, version 2.7. available at http://www.python.org." 27, 48

[40] "2nd generation intel® core tm processor family mobile and intel® celeron® processor family mobile," September 2012. 43

[41] D. Rodopoulos, A. Papanikolaou, F. Catthoor, and D. Soudris, "Demonstrating hw-sw transient error mitigation on the single-chip cloud computer data plane," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 23, pp. 507–519, March 2015.

[42] "Ubuntu manpages (cpufrequtils)." `http://manpages.ubuntu.com/manpages/lucid/man1/cpufreq-set.1.html`.

Chapter 7
Appendix

7.1 Source Code

The source code of Depman tool for all three implementations can be found at https://github.com/A-Kokolis/thesis-ntua. The code is licenced under the GPLv3 licence and can be modified and redistributed under these terms.