Runtime Interval Optimization and Dependable Performance for Application-Level Checkpointing

Apostolos Kokolis*, Alexandros Mavrogiannis†, Dimitrios Rodopoulos*,‡, Christos Strydis¶ and Dimitrios Soudris*
*MicroLab-ECE-NTUA, Greece, †CMU, USA, ‡ESAT–KU Leuven, Belgium ¶Erasmus Medical Center, Netherlands
Contact Email: depman@microlab.ntua.gr

***Abstract*—As aggressive integration paves the way for performance enhancement of many-core chips and technology nodes go below deca-nanometer dimensions, system-wide failure rates are becoming noticeable. Inevitably, system designers need to properly account for such failures. Checkpoint/Restart (C/R) can be deployed to prolong dependable operation of such systems. However, it introduces additional overheads that lead to performance variability. We present a versatile dependability manager (`DepMan`) that orchestrates a many-core application-level C/R scheme, while being able to follow time-varying error rates. `DepMan` also contains a dedicated module that ensures on-the-fly performance dependability for the executing application. We evaluate the performance of our scheme using an error injection module both on the experimental Intel Single-Chip Cloud Computer (SCC) and on a commercial Intel i7 general purpose computer. Runtime checkpoint interval optimization adapts to a variety of failure rates without extra performance or energy costs. The inevitable timing overhead of C/R is reclaimed systematically with Dynamic Voltage and Frequency Scaling (DVFS), so that dependable application performance is ensured.**

I. Introduction

As an upper bound is emerging for single-processor performance, integration of multiple processors is considered a very viable solution by system vendors [1]. In Embedded Computing (EC), a requirement to diversify the features of the respective products is driving the integration of many processing nodes [2] on a single die or package. In High Performance Computing (HPC), we see a clear trend in the number of cores grouped per socket [3], with consequent amplification of the achieved performance. HPC also experiences diversification, given its increasing convergence with cloud systems [4].

In the meantime, process technology aims at providing semiconductor components of increased reliability. Novel transistors exhibit very good reliability profiles, as for the case of soft errors [5]. However, it is inevitable that transistor numbers will be constantly increasing on silicon dies. Thus, while the reliability profile of a single transistor may be improved, aggressive integration leads to elevated failure rates [6]. Also, there is a certain class of semiconductor phenomena, which amplify time-zero and time-dependent variability [7]. In view of these observations, it is important to emphasize on the dependability of modern/future systems, since their components are bound to exhibit variability and/or degradation.

To that end, computer designers have been enhancing their designs with reliability, availability and serviceability (RAS) schemes to identify and correct erroneous behavior. Software defined methods have been also implemented, such as application- or system-level C/R. However, in most cases, RAS mechanisms come with associated timing overheads, which lead to inevitable *performance variability*. For instance, the disabling of caches has been known to cause an increment in the cycles required for application execution [8]. Similarly, system-level C/R has been known to upper-bound performance, thus creating the so-called “Reliability Wall” [9]. It is clear that C/R techniques should (i) be aware of time-varying error rates *and* (ii) should be designed so that no unnecessary performance overheads intervene with the application.

The contributions of our work reside precisely across these two directions. We present `DepMan`, a run-time dependability manager, which orchestrates an application-level C/R implementation. Our scheme monitors a many-core platform for Detected Unrecoverable Errors (DUEs) and issues a restart upon detection. Additionally, it performs an on-the-fly estimation of the mean time between DUEs, as a useful indication of the platform’s error rate. According to this estimation the checkpoint interval is appropriately reconfigured. `DepMan` also compensates for the performance overheads of C/R, so that dependable performance can be guaranteed for the application. The knob of choice for dependable performance is DVFS, and a variety of strategies are implemented.

For verification purposes, we use a neuroscientific application that performs time-driven simulations of inferior olive neuron cells (InfOli) [10]. This application has large execution times and a high societal relevance, given the insight it provides into olivocerebellar functionality. To demonstrate the versatility of our approach, we use two platforms: (i) The Single-Chip Cloud Computer (SCC), an experimental processor [11] developed by Intel Labs as a concept vehicle for many-core software research. (ii) A general purpose computing platform with an Intel® Core™ i7-2630QM processor. Through experiments, we substantiate the credibility of on-the-fly DUE rate monitoring, as well as the slack recreation capabilities of DVFS against C/R temporal overheads.

The current paper is organized in the following way: Section II presents samples of prior art, comments upon them and motivates the necessity for the `DepMan` scheme. Section III presents the experimental setup for the evaluation of `DepMan`. Section IV introduces the dependability manager, it explains its specifications and modules. An extensive experimental latency and energy evaluation for `DepMan` is presented in Section V. Finally, conclusions are summarized in Section VI.

II. Prior Art and Motivation

A typical distinction of functional reliability violations is between (i) masked errors (ii) correctable errors (iii) detected unrecoverable errors (DUEs) and (iv) silent data corruptions (SDCs) [12]. In the current paper, we focus on transient DUEs. The unrecoverable nature creates the requirement for a rollback mechanism that restores the system from an earlier state. This effectively constitutes a C/R scheme. There is a wide variety of prior art on C/R, featuring application-level [13] and system-level [14] solutions. C/R efficiency is affected by (i) the temporal distance between a DUE occurrence and the latest checkpoint, (ii) the time required to restart and (iii) the latency introduced by checkpoint storage. C/R optimization is explored through the selection of the time interval between consecutive checkpoints, namely the checkpoint interval (CI). Young introduced a first order approximation to the optimum CI [15]. Daly expanded the model to account for errors during restart and introduced a higher order estimate [16]. Aperiodic checkpoint placement techniques have also been explored to account for other failure distributions [17]. An interesting feature of C/R is the adaptability under variable error rates. This has direct implications for the efficiency of C/R implementations: A CI that is disproportionately longer than the failure rate leads to large temporal losses upon restart, while frequent checkpointing amplifies the total introduced latency. Theoretical studies directly motivate the time-variance of error rates [18]. Aiming at energy efficiency, adaptive C/R has been proposed in the domain of embedded real time systems and has been verified with simulations [19].

Generally, errors can be mitigated in many ways: redundancy/voting [20], fine-grain rollbacks [21] or, in an abstract sense, combinations of monitors and knobs [22]. Many of these techniques have negative impact on the performance of the system, as in the case of permanently erroneous cache block disabling [8]. C/R techniques have been known to bound the performance of systems, especially in the exascale era [9].

In reclaiming the temporal overheads of functional reliability techniques (such as C/R), attempts have been made to sprint the system execution in order to recreate timing slack used for mitigation. A custom solution [21] has been proposed to absorb the timing cost of fine-grain rollbacks. A more generic formulation has also been disclosed, leveraging traditional control principles [23]. In any case, there are multiple ways to boost processors in order to reclaim time. DVFS allows performance adjustment across a set of predefined voltage and frequency points. More elaborate techniques, allow an extra speed increment, under certain conditions of many-core chip utilization [24]. Finally, the chip's thermal capacitance can be exploited in order to release additional performance within the boundaries of the thermal design point (TDP) [25].

In view of prior art, it is clear that adaptive C/R has received significant attention, however existing reductions to practice do not surpass simulation-based evaluations [19]. Even though the temporal overhead of C/R is known, reduced effort has gone into practically absorbing its overheads. Our `DepMan` tool, addresses these concerns in the following way: (i) it is a *deployable* C/R implementation that can adapt at runtime to time-varying failure rates by monitoring error rates on-the-fly and selecting an optimal CI. (ii) The inevitable timing overhead of C/R is reclaimed by a dedicated `DepMan` module, which implements a specific DVFS policy. In the Sections that follow we elaborate on these specifications and we evaluate `DepMan`'s functionality through a set of experimental results.

III. Target Setup

A. Target Application

The target application, InfOli, that has been used is a simulator of the crucial set of brain cells, called inferior olive cells, based on the Hodgkin-Huxley model [10]. During the execution, each cell is simulated through its three compartments: the dendrite, the soma and the axon. The simulation is transient (i.e time-driven) with a constant time step. Inputs to the simulator are: the size of the neuron mesh, a connectivity file which declares the static connections between individual cells, a file of external input currents for each cell and an integer representing the number of simulations steps (`simSteps`) between the storage of two consecutive checkpoints. The output consists of a series of files which contain each cell's axon voltages for each simulation step. For the results presented in the current paper, a data-parallel version of the InfOli simulator is used [26], with each thread of execution being assigned a subset of cells. Inter-thread communication is implemented according to the capabilities in each of the target platforms (namely available libraries, etc).

B. Target Platforms

For the purposes of the current paper, `DepMan` has been assessed in two different platforms. The first is the Intel Single-Chip Cloud Computer (SCC) [11]. The chip consists of 48 cores, which are grouped in tiles of two cores each. The tiles are interconnected through a mesh network. Each individual core runs its own copy of Linux. It has its own private memory and a Message-Passing buffer, used to exchange messages between the cores of the chip. The chip has a voltage domain for every four tiles of two cores and 24 frequency domains for each tile of the chip providing the capability for DVFS. Furthermore, the board that hosts the SCC chip communicates with a Management Console Personal Computer (MCPC) through Ethernet and PCIe links providing power monitoring capabilities. Parallel applications are written (and cross-compiled) on the MCPC using libraries resembling the Message Passing Interface [11]. The executable is dispatched to the SCC through the mounted `/shared` directory which is also used for the gathering of inputs/outputs.

The second platform, is a general purpose x86 system running a generic Linux distribution on an Intel® Core™ i7-2630QM processor. As far as DVFS is concerned, the system uses the `acpi-cpufreq` driver and the `userspace` governor [27] is available, enabling us to performs DVFS on demand. The target application is implemented on the x86 system using the traditional Message Passing Interface.

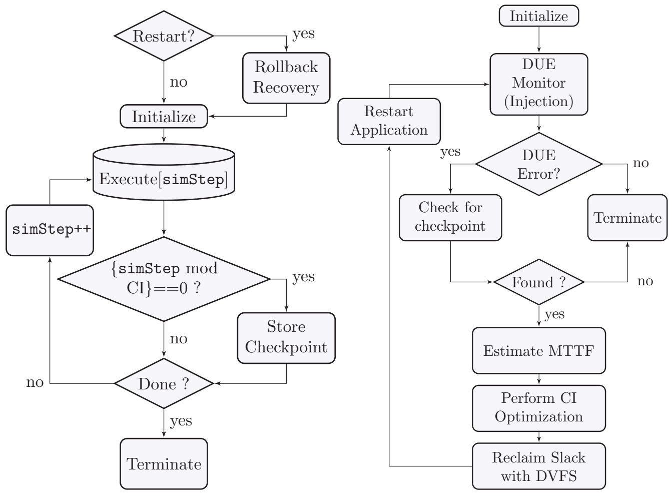


Fig. 1. Application template assumed and functionality of the `DepMan` tool

IV. DepMan Overview

`DepMan` is a run time dependability manager built in Python 2.7, which controls a many-core, application-level C/R implementation. It monitors for DUEs and adapts the CI of the application upon restart. It also contains dedicated modules that reclaim timing slack that is wasted on C/R. `DepMan` can manage any application that follows the template of Figure 1a. This assumes a many-core application that stores checkpoints at regular intervals (CI) and is able to restart from a designated checkpoint. We note that many workloads follow the iterative nature of Figure 1a [21], thus enabling step-wise C/R. `DepMan` responds to stimulus received by the application's standard output, corresponding to a DUE or similar violation. It performs an estimation of the DUE rate in the form of a Mean Time to Failure (MTTF). It adapts the CI and reclaims time wasted on C/R by implementing a specific DVFS policy. `DepMan`'s functionality is shown in Figure 1b.

A. DUE Diagnostics

`DepMan` detects DUEs that cause the application to stop by monitoring the standard output of the application for failure messages. In order to easily perform experiments with `DepMan`, we performed software injection of DUEs. The injection module is evaluated at time intervals Δt and assumes memoryless injection of DUE events at a constant rate as formulated Equation 1. Other error models can be adapted, however this falls beyond the scope of the current paper.

$$P = 1 - e^{-\Delta t/\mathrm{MTTF}} \tag{1}$$

B. Optimal Checkpoint Interval Adaptation

A major capability of `DepMan` is DUE rate estimation during the execution of the application. Given the inherent detectability nature of DUEs we are able to keep a log of the duration between consecutive DUE occurrences, the difference of which provides a Time to Failure (TTF) measurement. MTTF is estimated using a moving average filter that keeps the latest M TTF values as indicated by Equation 2.

$$\mathrm{MTTF} = \sum_{i=0}^{M-1} \frac{\mathrm{TTF}}{M} \tag{2}$$

Before the application is restarted, `DepMan` is able to optimize the managed C/R scheme using an estimation of MTTF. This optimization aims to minimize the total wall-clock execution time of the target application by selecting checkpoint locations based on the expected distribution of failures. This task is performed in three phases: *benchmarking*, *selection*, *transformation to simulation steps*: **(i)** The *benchmarking* phase is performed at design time. During this phase, the checkpoint latency and checkpoint interval are reported on the application's standard output and retrieved by `DepMan`. The average of the reported values is used in all further estimations. **(ii)** After the benchmarking phase, the optimum CI in time units is *selected* using Daly's formulation in Equation 3 [16], where τ is the time interval between two consecutive checkpoints (CI) and λ is the time overhead of each checkpoint, called checkpoint latency (assumed constant for a constant processor frequency). C/R can be modeled as a sequence of error cycles. The wasted time credited to a cycle is the sum of the checkpoint latencies and the time difference between the latest checkpoint and the occurrence of a DUE. The CI optimization is performed as part of the repair process before the application is restarted. As a result, the optimal checkpoint interval should minimize the total wasted execution time till the next DUE error. **(iii)** Finally, this value is then *transformed* to simulation steps using the CI in time units and the simulation steps of the benchmarking phase.

$$\tau_{opt} = \sqrt{2 \times \mathrm{MTTF} - \lambda} - \lambda \tag{3}$$

C. Slack Reclaiming

C/R improves system reliability but costs in additional execution time for storing checkpoints and performing roll-back/recovery. Even when exploiting CI optimization techniques, inevitably, C/R causes a scalability and performance barrier [9]. To enable ensure dependability of application performance, `DepMan` actuates DVFS on the target platform in order to speed up execution when needed. The following terms are used when formulating such policies:

Slack (s) is the time that the application has fallen back due to C/R invocation. The value of s_{ref} indicates the target slack we want to achieve at all times, which is typically set to zero. `DepMan` measures and updates the slack after each DUE error. The parameter n indicates the number of the restart operations that have occurred at any point during application execution. `DepMan` reacts to the value of $e_n = s_{ref} - s_{n-1}$ and proposes a frequency configuration in order to reclaim the, now negative, slack. Finally, the r parameter is used to tune how often a DVFS alteration is actually performed on the platform, measured in number of detected errors: for $r = 1$

TABLE I
Parameters Used in DVFS Policy Formulation

Parameter	Description
s_n	Slack after n DUE occurrences
s_{n-1}	Slack after $n-1$ DUE occurrences
t_{rec}	High frequency duration needed to fully reclaim s_n
f_d	Default operating frequency
f_{opt}	Theoretically "optimal" frequency configuration
f_c	Frequency setting currently applied

DVFS changes will be performed after every DUE occurrence, while for $r = 2$ DVFS changes are performed every two restart operations. The temporal overhead T_o due to C/R events is used collectively in order to accurately estimate the value of slack (s). It is the sum of all the following components:

Rollback Time (T_r): Lost time for computation that needs to be repeated because the application restarts from a previous simulation step. To quantify this time, we calculate the difference between the creation time of the latest valid checkpoint and the time the DUE error occurred. *Time to Repair* (T_{rep}): Time needed for the application to restore its state from a checkpoint file. This time overhead is recorded by the application itself and is stored so that DepMan may use it in the future. *Checkpointing Time* (T_{ckpt}): Time needed for the application to store a checkpoint file. This is recorded by the application and the results are stored to a file accessible by DepMan. In order to calculate the total T_{ckpt}, we multiply the checkpoint latency with the number of stored checkpoints until the DUE error occurred. *Time to Restart* (TTR): Time between the occurrence of a DUE error until DepMan restarts the execution of the application.

D. DVFS Policies

To reclaim slack used on C/R, we explore three DVFS policies. Table I contains variables used in our formulation.

For the SCC platform, where a DVFS alteration is a time consuming procedure, we focus on two different DVFS points. The Default Mode (533 MHz) represents the default execution mode and the Burst Mode (800 MHz) represents the "speed-up" setting. Algorithm 1 describes the slack reclaiming policy for the case of two available DVFS points (Default and Burst).

Algorithm 1: Alternating between two frequency options

```
1 $s_n = s_{n-1} - T_o + \text{TTF} \times (f_c - f_d)$;
2 if (n mod r = 0) then
3     if $s_n < 0$ then
4         $f_c$= Burst Mode Frequency
5     else
6         $f_c$= Default Mode Frequency
7     end
8 end
```

The general purpose system has an on-chip voltage regulator, allowing rapid DVFS changes, without a significant time overhead for each alteration. Given an abundance of DVFS options, DepMan should typically decide on the appropriate frequency that should be selected based on the available slack. In order make the optimum frequency decisions, we take into account the MTTF value: we select a frequency that would result to slack reclaiming in a conservative way during the next estimated TTF. Algorithm 2 shows a liberal solution, whereby the theoretically optimal frequency (f_{opt}) is the one that fully reclaims the slack, assuming a DUE after MTTF seconds (line 2). The closest available option is finally applied (line 3).

Algorithm 2: Demand-driven frequency boosting

```
1 $s_n = s_{n-1} - T_o + \text{TTF} \times (f_c - f_d)$;
2 $f_{opt} = (-s_n + f_d \times \text{MTTF})/\text{MTTF}$;
3 $f_c = \texttt{find}(\max\{f\} : f \leq f_{opt})$;
```

Algorithm 3 maintains a more conservative approach towards high frequency boosting. Upon a DVFS change, a process is forked which idles until the time to fully reclaim slack (t_{rec} – calculated in line 7) has passed. Afterwards, the forked process restores frequency to its default value (line 8). Possibility of positive slack is eliminated, thus saving energy.

Algorithm 3: Demand-driven frequency boosting based on slack, with asynchronous backdrop for energy savings

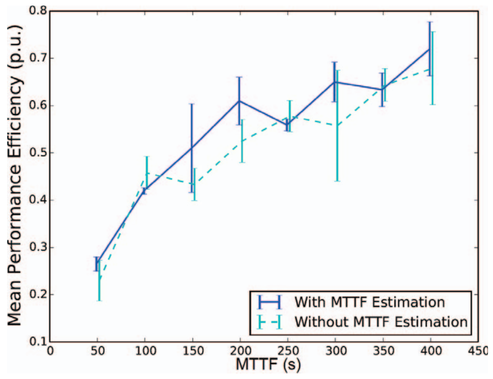
```
1 if $f_c = f_d$ then
2     $s_{n-1} = 0$;
3 else
4     kill[reset_freq_after($t_{rec}$)]
5 end
6 $s_n = s_{n-1} - T_o + \text{TTF} \times (f_c - f_d)$;
7 $f_{opt} = (-s_n + f_d \times \text{MTTF})/\text{MTTF}$;
8 $f_c = \texttt{find}(\max\{f\} : f \leq f_{opt})$;
9 $t_{rec} = -s_n/(f_c - f_d)$;
10 fork[reset_freq_after($t_{rec}$)];
```

We note that Algorithms 2 and 3 will only be tested on the general purpose system. Apart from a huge DVFS latency, the SCC has only two DVFS points that are applicable to the time scales of our experiments. Conversely, the r parameter introduced is only useful on the SCC, since DVFS latency on the general purpose system is negligible ($\sim$ microseconds).

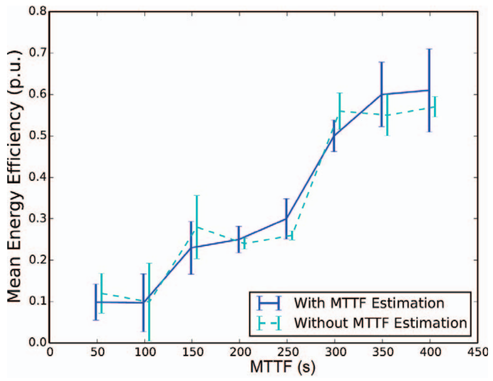
V. Experimental Evaluation

A. On-the-fly MTTF Estimation

The feasibility of on-the-fly MTTF estimation was tested on the SCC platform for a simulation of 50 neuron cells per core. Efficiency is calculated with respect to the reference execution time or energy required to complete the DUE-free, C/R-free and DVFS-free run (i.e. $\frac{E_{\text{ref}}}{E_{\text{tested}}}, \frac{T_{\text{ref}}}{T_{\text{tested}}}$). For various values of injected MTTF, we calculate the performance and energy efficiency and report them in Figure 2. Two cases are presented, (i) a case where on-the-fly MTTF estimation is performed, assuming $M = 32$ and (ii) a case where no on-the-fly estimation is performed and MTTF is known in advance.

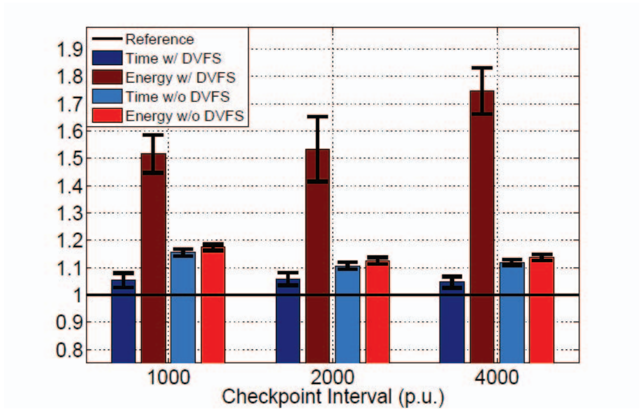


(a)

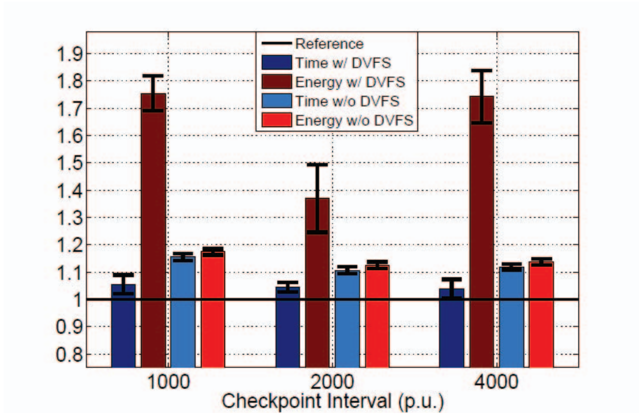


(b)

Fig. 2. Efficiency evaluation of on-the-fly MTTF estimation. Experiments for a constant rate of DUE injection, presented within 95% confidence intervals.



(a) $r = 1$



(b) $r = 3$

Fig. 3. Average time and energy evaluation for DepMan on the SCC (presented within 90% confidence intervals). Reference execution time is 823 s and energy reference 29 kJ. DUEs are injected at an MTTF = 120 s rate.

From these results we substantiate that using a sufficiently long moving average filter ($M = 32$), the on-the-fly estimation of MTTF does not reduce the efficiency of the DepMan scheme. It works adequately with adaptive CI optimization according to Equation 3, as if the MTTF was accurately known in advance.

B. Slack Reclaiming Results

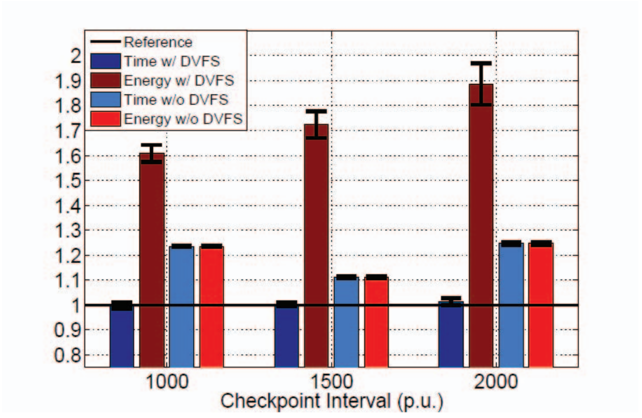
In order to evaluate the DVFS module for slack reclaiming we use DUE-free and C/R-free runs as references for time and energy. For all experiments, the InfOli application performs 120,000 simulation iterations. We evaluate DepMan with and without the DVFS module and normalize time and energy. A variety of constant CIs are used, appearing in number of simSteps in the horizontal axes of Figures 3 and 4. Combined experimental evaluation of run time CI adaptation and slack reclaiming with DVFS are left as future work.

1) SCC platform: In Figure 3 we present execution time and energy consumption measurements for a constant MTTF. The experiment was performed for two values of $r = 1$ and $r = 3$ (number of restarts between DVFS changes). The application workload is a simulation of 64 neuron cells per core and is proportionally larger to the DVFS latency of the SCC platform. DepMan is clearly capable of absorbing most of the slack used by C/R. When the value of r is increased, we typically gain in energy without any performance loss, by avoiding extra overhead of the costly DVFS alterations per DUE error. The high DVFS latency and the overhead of checkpoint storage (through the mounted /shared directory) lead to slight divergence from the target execution time and increased energy overheads of DepMan's DVFS module, especially when the CI is set to 1,000 simSteps.

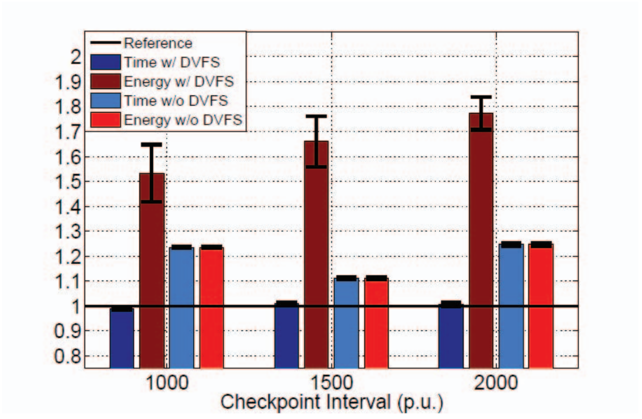
2) General Purpose Platform: This system supports more DVFS points varying from 800 MHz to 2 GHz with a step of 100 MHz. In all cases, we use 1.2 GHz as the default frequency of the experiments (f_d) and an InfOli simulation of 96 cells per core. The V_{dd} range for the processor is taken from its data sheet and is assumed to linearly correlate with frequency. We estimate energy using the measured execution time and the square law of dynamic power (i.e. $P_d \propto f \times V_{dd}^2$).

The evaluation of Algorithm 2 is shown in Figure 4a. The DVFS module is more effective in converging to the reference time. This is enabled by the minimal DVFS latency of the processor, allowing frequency changes after every DUE/restart event (i.e. $r = 1$). Also, the higher number and finer granularity of DVFS options work in favor of dependable performance.

The results for Algorithm 3 appear in Figure 4b. Again, dependable performance is achieved with the DepMan DVFS module. Compared to Algorithm 2, this is a more energy



(a) DVFS policy based on Algorithm 2



(b) DVFS policy based on Algorithm 3

Fig. 4. Average time and energy evaluation for `DepMan` on the general purpose platform (reported within 90% confidence intervals). Reference execution time is 578 s. DUE injection is tuned at MTTF = 20 s.

efficient solution across all inspected CIs. The asynchronous reset of the frequency to f_d upon full reclaiming of slack is definitely reducing the energy budget of DVFS boosting. Even if Algorithm 2 allows for frequencies below f_d, Algorithm 3 is more energy efficient: it avoids slack overcompensation by resetting the frequency back to f_d at exactly the proper time.

VI. Conclusion

We have presented and evaluated `DepMan`, a dependability manager for an application-level C/R scheme. Our framework detects DUEs and estimates their MTTF on-the-fly. Upon restarting, it adapts the checkpoint interval to the currently applicable MTTF, to reduce the C/R overhead. Finally, slack wasted on the C/R procedure is reclaimed with DVFS to ensure dependable performance. We have evaluated `DepMan` on the Intel SCC and on a general purpose computer with an Intel® Core™ i7-2630QM processor. The target application is a simulator for inferior olive neurons. For the evaluation of our scheme, DUEs are injected in the experimental setup using a probabilistic model. The application follows the expected application-level C/R template and `DepMan` enables run time observability and controllability for dependable execution.

For a sufficient summing window, on-the-fly MTTF estimation is accurate for adaptive CI optimization and comes with negligible performance or energy overheads. Three different dependable performance policies have been tested on `DepMan`. All of them show that slack can be mitigated by carefully actuating DVFS. Generally, dependable performance under C/R comes at an energy cost. Conservative reset to default frequency upon slack recuperation appears to be best.

Acknowledgments

This work is partially supported by the FP7-612069-HARPA and ERC-2014-PoC grant BrainFrame (#641000). SCC is available under MTA between ICCS and Intel Corporation.

References

[1] S. Fuller and L. Millett, "Computing performance: Game over or next level?" *Computer*, vol. 44, no. 1, pp. 31–38, Jan 2011.
[2] S. Jalali, "Trends and implications in embedded systems development," TATA Consultancy Services, Tech. Rep., 2009.
[3] TOP500 Supercomputer Cites. Cores per socket – performance share.
[4] J. Simons, "Hpc cloud bad; hpc in the cloud good," in *IPDPS*, 2013.
[5] Seifert, N. et al., "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE TNS*, vol. 59, no. 6, pp. 2666–2673, Dec 2012.
[6] A. Dixit, R. Heald, and A. Wood, "Trends from ten years of soft error experimentation," in *IEEE SELSE Workshop*, 2009.
[7] Rodopoulos, D. et al., "Atomistic pseudo-transient bti simulation with inherent workload memory," *IEEE TDMR*, vol. 14, no. 2, June 2014.
[8] Hardy, D. et al., "The performance vulnerability of architectural and non-architectural arrays to permanent faults," in *MICRO*, 2012.
[9] Xuejun Yang et al., "The reliability wall for exascale supercomputing," *Computers, IEEE Transactions on*, vol. 61, no. 6, June 2012.
[10] 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.
[11] Mattson, Timothy G. et al., "The 48-core scc processor: The programmer's view," in *SC*, 2010.
[12] Nguyen, H.T. et al., "Chip-level soft error estimation method," *IEEE TNS*, vol. 5, no. 3, pp. 365–381, Sept 2005.
[13] Bronevetsky, Greg et al., "C3: A system for automating application-level checkpointing of mpi programs," in *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2004, pp. 357–373.
[14] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *IEEE IPDPS*, 2009.
[15] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.
[16] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Gener. Comput. Syst.*, vol. 22, no. 3, 2006.
[17] Oliner, Adam J. et al., "Cooperative checkpointing: A robust approach to large-scale systems reliability," in *ICS*. ACM, 2006, pp. 14–23.
[18] A. El-Gohary, "Bayesian estimation of the parameters in two non-independent component series system with dependent time failure rate," *Applied Mathematics and Computation*, vol. 154, no. 1, 2004.
[19] Y. Zhang and K. Chakrabarty, "Energy-aware adaptive checkpointing in embedded real-time systems," in *DATE*, 2003, pp. 918–923.
[20] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of R&D*, vol. 6, no. 2, 1962.
[21] Rodopoulos, D. et al., "Demonstrating hw–sw transient error mitigation on the single-chip cloud computer data plane," *TVLSI*, vol. 23, 2015.
[22] Mcnairy, C. et al., "Error framework for a microprocessor and system," Jun. 27 2013, WO Patent App. PCT/US2011/066,658.
[23] Rodopoulos, D. et al., "Tackling performance variability due to ras mechanisms with pid-controlled dvfs," *CAL*, 2014.
[24] Charles, J. et al., "Evaluation of the intel core i7 turbo boost feature," in *IEEE IISWC*, Oct 2009, pp. 188–197.
[25] Raghavan, Arun et al., "Computational sprinting on a hardware/software testbed," in *ASPLOS*, 2013, pp. 155–166.
[26] Rodopoulos, D. et al., "Optimal mapping of inferior olive neuron simulations on the single-chip cloud computer," in *SAMOS*, July 2014.
[27] "The linux kernel governors documentation," https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.