Reliability-Oriented Resource Management for High-Performance Computing

Giuseppe Massari, Miriam Peta, Alessandro Campi, Federico Reghenzani, Federico Terraneo, Giovanni Agosta, William Fornaciari[a,*], Sebastian Ciesielski, Michal Kulczewski, Wojciech Piatek[b]

[a] *DEIB - Politecnico di Milano, Via G. Ponzio 34/5, Milano, Italy*
[b] *Poznan Supercomputing and Networking Center, Street Jana Pawła II 10, Poznań, Poland*

Abstract

Reliability is an increasingly pressing issue for High-Performance Computing systems, as failures are a threat to large-scale applications, for which an even single run may incur significant energy and billing costs. Currently, application developers need to address reliability explicitly, by integrating application-specific checkpoint/restore mechanisms. However, the application alone cannot exploit system knowledge, which is not the case for system-wide resource management systems. In this paper, we propose a reliability-oriented policy that can increase significantly component reliability by combining checkpoint/restore mechanisms exploitation and proactive resource management policies.

1. Introduction

High-Performance Computing (HPC) is a critical and strategic computing infrastructure for both the industrial and scientific sectors [2, 4]. The push towards Exascale-grade computing systems leads to increased power and energy requirements for supplying HPC infrastructures. For such kinds of systems, a 20MW power envelope is widely considered an energy efficiency goal. As a result, any overhead added to the computation, even if relatively low from a latency point of view, is bound to cause massive energy consumption as well as unacceptable costs.

One important source of overhead is represented by *faults* – adverse events due to system component failures or other issues, that lead to errors in the computation, and therefore to the need to redo all or part of it. As can be easily understood, in a scenario where such a failure leads to the need to redo a fraction f of the computation, with a consequent performance loss and additional consumption of energy. Recently, Walker et *al.* [25] provided an interesting analysis of the performance-reliability trade-off in HPC systems.

*Corresponding author

ACCEPTED MANUSCRIPT - AUTHORS' VERSION
DOI: 10.1016/j.suscom.2023.100873

As HPC platforms evolve towards Exascale, the issue of reliability is becoming increasingly pressing. This is due to the sheer amount of available resources coupled with the increase in heterogeneity, resulting from the widespread use of accelerators. Indeed, transient or permanent failures that would seldom occur in a smaller machine become commonplace at extreme scales, increasing the failure rates of very large jobs, if the system and the application are unable to cope with the failure of a single processing element. More specifically, preventing the occurrence of permanent faults, by improving the hardware components' reliability, becomes a primary goal if we consider the impact on maintenance effort, temporary unavailability of resources, and thus costs. In this work, we focused on permanent faults. In general, we can state that an efficient strategy requires implementing a mix of reactive and proactive solutions [1]. Reactive methods monitor the occurrence of failures and try to mitigate their negative effects (i.e., the need to kill and restart the application) by exploiting, e.g., checkpoints. Performing a checkpoint consists of saving the current status of execution of the application on a "image" stored on a persistent storage device. This means that, in the case of faults, we can resume the execution of the application from a safe state, thus limiting the loss of valuable execution time.

Proactive methods, instead, leverage fault probability predictions to allocate the most reliable resources to jobs that would be more dramatically affected by a failure. The former approach, which imposes significant checkpoint overheads, may need to rely on the application cooperation to identify safe checkpoints, where a process can be frozen and moved without dramatically impacting the overall application execution [8]. Whereas the latter approach is limited by the ability to perform correct predictions in an environment where users tend to overestimate execution times [18].

In this work, we developed a run-time resource management framework to handle the trade-off between reliability and performance. From the implementation perspective, we extended the Barbeque Run-Time Resource Manager [3] with a reliability-aware resource allocation policy (*Reliam*) and a reliability monitoring module for the processing resources. The policy relies on a predictive model, which correlates the probability of a fault with the thermal history of the component. To validate the proposed policy, we employed a simulator of HPC systems, DCworms, by extending it with suitable thermal and reliability models. By employing a representative benchmark application, we obtained performance and power profiles on a suitably instrumented computing node and used them to drive the simulation of a multi-node system.

The rest of this paper is organized as follows. In section 2 we review the literature on resource management and reliability for HPC systems. In section 3 we introduce the proposed framework, while in section 4 we describe our extensions to DCworms. Finally, in section 5 we discuss the experimental campaign, and in section 6 we draw some conclusions and highlight future research directions.

2. Related Works

Current approaches to reliability management, in HPC systems, are mainly based on hardware redundancy and checkpoint/restore mechanisms. In case of faulty executions, the management system can react by moving the workload to a healthy set of resources and resume the execution from the last safe checkpoint images. From the implementation perspective, several improvements have been proposed to increase the checkpoint efficiency, including incremental checkpoint [19] and optimal placement of checkpoints [15]. In general, the drawback is in fact represented by the overhead introduced by the need to periodically save on disk a snapshot of the application execution, which could require from tens to hundreds of megabytes. Among the *proactive* approaches, we can find proposals focusing on the optimization of the placement of virtual machines (VMs), with respect to multiple objectives [7, 26, 27], including balancing the resource utilization. These can be considered coarse-grained approaches since a VM can host multiple applications. Alternatively, more fine-grained approaches include task scheduling and application-level resource allocation policies. Currently, in HPC, most of the resource management effort is performed by a global job scheduler, such as SLURM, which allocates computing nodes to jobs at start time. At this level, some works have attempted to introduce reliability-aware schedulers [10, 18], while in [6] the authors provide a review of the thermal-aware task schedulers. Keeping the temperature under control is in fact a key factor in reducing the occurrence of faults [22]. In this regard, HPC systems usually install expensive cooling systems to manage the temperature and humidity levels of the environments. In the embedded systems domains this problem has been largely explored, often by exploiting reactive control-theory-based approaches. Mohammed et al. In [17] authors proposed a temperature-aware task scheduler for many-cores Systems on Chip, relying on task migration, dark silicon and DVFS to improve system reliability. Huang et al. [11] presented a task allocation technique for embedded Multi-Processor Systems on Chip (MPSoC) Platforms aimed at slowing down the wear out of the hardware components, estimating the lifetime reliability of multiprocessor platforms by using an analytical model. These works are related to ours in the concept, although they have been intended for a different technology. As for the HPC scenario, relatively few works regarding resource reliability have been presented. A model-free controller has been proposed to control the temperature and reliability [21]. Gottumukkala et al. [9] designed a reliability-aware resource allocation algorithm relying on the prediction of the time between failure based on the Weibull distribution, in order to minimize the performance loss due to failures. The authors concluded by stating how the CPU load and temperature can improve the reliability prediction accuracy. Moreover, suitable resource management strategies could consider the level of fault tolerance to optimize a performance metric, considering run-time reliability predictions.

Our work followed this direction. More in detail, we implemented a framework combining proactive and reactive reliability management strategies, as proposed also in [8]. We extended the BarbequeRTRM framework by introducing

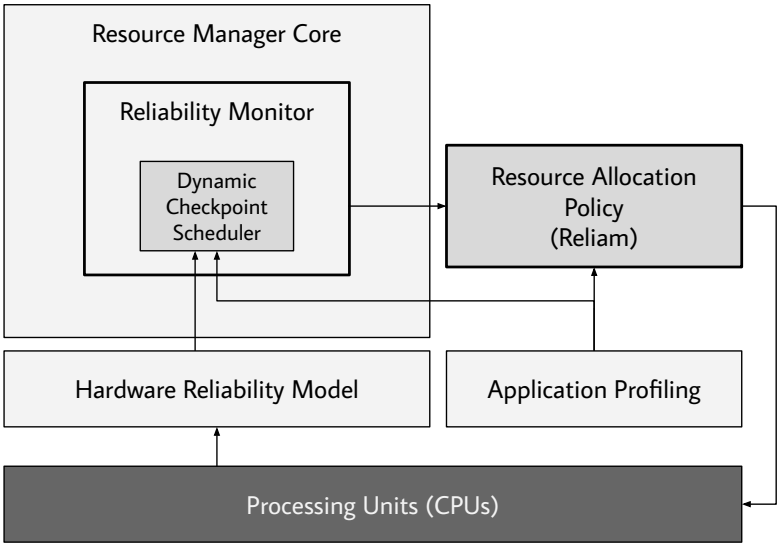


Figure 1: The core components of the reliability-aware resource management framework.

additional policies and integrating external monitoring components. Accordingly, the framework allows us to monitor at run-time the status of applications and computing resources, in order to take decisions aiming at minimizing the application performance loss, while maximizing the lifetime of the computing resources. Furthermore, the reliability control is backed up by a checkpoint scheduler able to tune the checkpoint rate per application, with respect to a user-defined overhead maximum threshold.

3. Run-time Reliability Management

As already introduced, in this work we present a reliability management framework, based on the extension of the Barbeque Run-Time Resource Manager (BarbequeRTRM). In Figure 1, we sketched the components involved in the process. The overall strategy is based on mixing proactive and reactive approaches. The idea in fact is to minimize the probability of failures while providing support for reacting in case of occurrence, through the well-established Checkpoint/Restore mechanisms, taking into account also the impact on the application execution time.

Specifically, the *Reliability Monitor* gets data from a Hardware Reliability Model, in charge of estimating the probability of the processing units to fail (predicted number of *FIT: Failures-In-Time*). The goal here is two-fold: 1) to trigger the execution of the policy, in the case of a high probability of failures, to prevent the occurrence of permanent faults, through suitable resource

allocation strategies; 2) to drive the per-application rate of the checkpoints. In this regard, our goal is to find a balance between maximization of the reliability and minimization of the overhead due to the periodical checkpoints of the running applications. The two goals are in a trade-off: higher reliability requirements push for performing the application checkpoints at a high rate, but the consequence is an impact on the overall application execution time. The *Dynamic Checkpoint Scheduler* has been introduced with the goal of defining the scheduling of the checkpoint processes, on a per-application basis, considering the actual application requirements, as explained later.

Moreover, the reliability information provided by the model has the twofold objective of triggering an immediate re-execution of the resource allocation policy and providing a picture of the status of the processing resources. This enables the possibility of allocating the processing resources, according to the reliability profile, as will be detailed in the next subsection.

3.1. The Reliam Resource Allocation Policy

Reliam is a resource allocation policy whose goal is minimizing the wear out of the computing resources, improving the reliability of the system and, consequently, of the running applications. The execution of the policy is periodically invoked in order to allow an adaptive behaviour, dependent on the run-time status of the hardware components as well as the overall workload. It deals with the adaptation of the CPU resources allocation according to the actual requirements of the workload, while meeting the system's reliability requirements, resorting to task migration and process freezing.

Reliam aims at balancing the trade-off between performance and reliability, through the combination of two main basic blocks, one consisting in the *resource assignment*, i.e., how much CPU resource to allocate, the other in the *resources binding*, which will be widely discussed in the following sections.

3.1.1. Resource assignment

The resource assignment component of Reliam aims at minimizing the system performance loss caused by the reliability control, analyzed in the next section. It consists of a series of Proportional Integral and Derivative (PID) controllers, one for each running application, whose objective is to allocate the *optimal* amount of computing resources.

We define the CPU *quota* Q as the percentage, in terms of time, of CPU resources of the machine:

$$Q = C * 100 \tag{1}$$

where C is the total number of CPU cores, and the observed CPU *usage* as the allocated CPU quota effectively exploited by an application during its execution. The objective of the controller is to assign to each application a CPU quota *as close as possible* to the CPU usage it would be observed in the case of unlimited resources (Q_I).

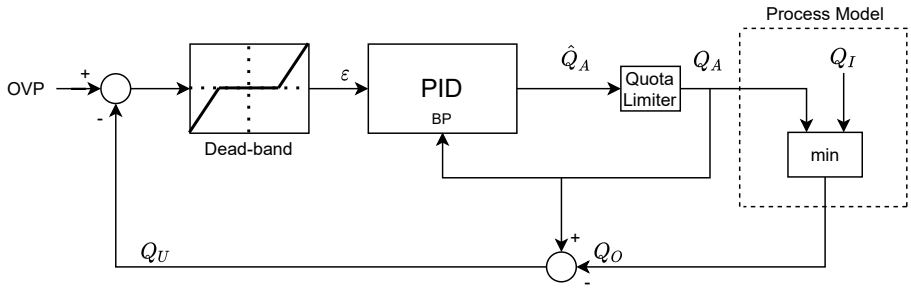


Figure 2: CPU quota allocation control loop.

The main challenge of such a problem is represented by the loss of observability (Q_I is not measurable), occurring when the allocation of the CPU quota gets under-assigned. More specifically, defining Q_A as the CPU quota *assigned* to an application and Q_O as its *observed usage*, we can compute the unused CPU quota, Q_U, as:

$$Q_U = Q_A - Q_O \tag{2}$$

From the equation above, the lack of observability becomes clear. In the case in which Q_A is greater than Q_O, Q_U represents the over-assignment of the CPU quota allocation. Conversely, if Q_U is equal to zero, it is impossible to infer whether the amount of assigned resources actually matches Q_I or it results in an under-assignment. For this reason, we introduce the concept of *over-provisioning*, OVP, i.e. a small excess of assignment needed to identify the assignment as the correct one. More specifically, OVP is a constant value such that, when $0 < Q_U \leq OVP$, the value of Q_A is considered correct.

Moreover, the finite number of resources available in the system is taken into account by the *quota limiter*. Once the CPU quota to assign is computed for all of the running applications, if the total allocated quota exceeds the total available one, all the allocations get reduced proportionally and the new assignment is back-propagated to the state of each controller as part of the PID anti-windup strategy. The resulting control loop is summarized by Figure 2. The idea behind the adaptive assignment is to provide the running applications with a number of computational resources such to optimize the use of the available ones, which might be reduced in number by the action of the reliability control, discussed in the next section.

3.1.2. Resource binding

The resource binding component of Reliam is the one proactively improving the reliability of the system. At each invocation, the policy is entrusted with the ability to take three kinds of decisions:

1. Force to idle some processing units (keep unused to cool down);

2. Redefine the binding (or mapping) of the processing units (task migration);

3. Freeze an application execution.

Such decisions are made possible by the presence of a reliability monitor, able to estimate the probability of failure of each interested processing unit. After being queried by the policy, the monitor returns a *reliability value*, in our case $1/FIT$, such that, for each hardware component, the lower the value, the higher the probability for it to fail. The user can configure a critical value, such that a suitable management action can be triggered, on the basis of the predictions of the reliability monitor.

Algorithm 1 Resource binding.

```
1: sorted_res ← SortComputingResourcesByDescendingReliabilityValue()
2: for all r ∈ sorted_res do
3:     if val_r < CRITICAL_VALUE then
4:         SetIdle(r)
5:     end if
6: end for
7: for all frozen_app do
8:     if AmountResumedResources() ≥ QuotaReqsOf(frozen_app) then
9:         Thaw(frozen_app)
10:    end if
11: end for
12: for all app do
13:    if all resources bound to app are idled then
14:        Freeze(app)
15:    else
16:        ComputeResourceQuotaFor(app)
17:    end if
18: end for
19: BindAppsToSortedResources()
```

Algorithm 1 shows how resource binding is performed. After sorting the computing resources by descending reliability value, Reliam idles the ones exhibiting a critical probability of failure. If, in the current invocation, one or more resources are resumed, among the ones that were forced to idle for reliability purposes in the previous invocation, the applications previously frozen, if any, are thawed. The thawing is carried out only if the resumed resources are enough to cover the quota requirements of the frozen applications: if only a core is resumed, while two applications are frozen, if they require $Q = 100$ each, only one of them gets thawed.

Afterwards, the CPU quota allocation takes place. If in the previous invocation of the policy, an application had been bound solely to processing units,

currently found unreliable, such application is frozen, otherwise, the PID controller logic exposed in the previous section is actuated. The decision of freezing an application that is stressing a group of cores ensures that the idling of the involved resources impacts only the performance of that specific application and not the entire workload. On the contrary, a CPU quota reassignment could reduce the amount of available quota to allocate to the other running applications. Moreover, the freezing operation prevents the overheating of new CPU cores by the same application, while the system is already deprived of a share of the quota.

Finally, *non-idled* cores are bound to the active applications in descending order of reliability value. This decision ensures that if the cumulative number of resources required by the running applications is lower than the number of available resources, only the most reliable ones will be bound.

3.2. Dynamic Checkpoint Scheduler

In systems in which many parallel executions are in place, one can expect that each of them, depending on the delivered service, might have different performance and reliability requirements. For instance, while, for a time-critical application, avoiding timing failures is a mandatory requirement, in the case of a batch application, having a recent checkpoint to restart from in the event of a failure is a much more important requirement. In general, we want to give to the user the possibility of specifying which requirement dominates.

With the Dynamic Checkpoint Scheduler, we propose an adaptive solution, providing the possibility to specify an application-specific upper bound on the checkpointing overhead. This is the maximum performance loss tolerated by the application, expressed in terms of percentage over the overall execution time. The Application Profiling module collects statistics at run-time on checkpoint and execution times, that both the resource allocation policy and the checkpoint scheduler can exploit. The Dynamic Checkpoint Scheduler, therefore, aims at scheduling the checkpoints on the basis of both user requirements and hardware reliability.

At application launch, the user is required to set the maximum checkpoint overhead tolerated by the application, expressed as the ratio of checkpoint and application code execution times. If the setting of the ratio is not provided, a default value is considered. Therefore, the checkpoints get scheduled as summarized by Algorithm 2.

For each application, the scheduler computes an estimation of the checkpoint latency based on the arithmetic mean of the previous dumps. Hence, the expected overhead is computed. The routine considers the ratio between the cumulative amount of time spent performing checkpoints and the total time elapsed since the start of the application. This choice allows us to reduce the impact of the variance on the computation of the overhead. Finally, if the expected overhead is lower than the specified one, the checkpoint is launched and the statistics are updated.

Algorithm 2 Dynamic Checkpoint Scheduler pseudo-code.

```
while True do
    app ← GetNextRunningApplication();
    meanChkTime ← GetCheckpointTimeMean(app);
    elapsedTime ← GetElapsedTimeSinceStart(app);
    totalTime ← elapsedTime + meanChkTime;
    performedChk ← GetNrPerformedCheckpoints(app);
    totalChkTime ← (performedChk +1) * meanChkTime;
    expectedOverhead ← totalChkTime/totalTime;
    if expectedOverhead ≤ specifiedOverhead(app) then
        Dump(app);
        UpdateStatistics(app);
    end if
end while
```

4. Large Scale Simulation with DCworms

After discussing the design of our framework, as well as the tuning of the policy parameters on the resource manager running on a single-node system, we move our focus on validating the approach on a larger scale. For the experimental setup, we relied on an HPC system simulator: *DCworms* [13]. More specifically, we extended the current version of the simulator, by adding thermal and reliability models, other than re-implementing the policies previously tested on the single-node system configuration.

DCworms is a Data Center Workload and Resource Management Simulator, developed by the Poznan Supercomputing and Networking Center, which allows the modelling of large-scale computing systems and workloads together with the evaluation of various management strategies. It is designed as an object-oriented, plugin-based, event-driven simulator. Thus, it provides easy extension capabilities that allow us to plug in the reliability policies, as well as to integrate the necessary measurements based on performance counters. Essentially, a plugin will be activated when an event fires, such that we can inspect the state of the simulation to retrieve the counters, compute the next actions based on the policy, and then apply them. Additionally, DCworms supports the inclusion of a wide range of performance and energy models (by using plugins) that allow describing the behaviour of the evaluated system. They enable the estimation of corresponding factors like power, temperature, and processors' utilization, by playing an important role while performing management actions. For this work, DCworms has been extended to support checkpoint/restore mechanisms which can be characterized in performance, and with reliability monitors and fault injectors.

4.1. Thermal Model

An important extension to DCworms is the integration in the simulator of a custom thermal model capable of simulating core temperatures, enabling the

assessment of reliability metrics that depend on temperature, thus aligning DCworms to recent trends in HPC modelling and simulation [12]. A thermal model suitable for HPC infrastructure simulation needs to satisfy several requirements which led us to design a custom solution. First of all, the need to simulate a large number of cores and CPUs to model an HPC infrastructure poses stringent performance requirements on thermal simulations. Moreover, since for similar performance considerations the simulation of the cores is coarse grain, current HPC simulators, such as DCworms, do not provide enough micro-architectural statistics to be able to reconstruct the power dissipated in each functional unit of every core, in order to construct the fine-grain power information that is needed as input by existing thermal models [24].

Both requirements suggest that a suitable solution could be found in the form of coarse grain thermal models, that focus on capturing only the main dynamics of the thermal phenomena, have low demands in terms of the input power level of detail, and simulate at high performance while providing sufficient accuracy to compute reliability metrics.

In this work, we have developed, starting from the theoretical model of [24], a coarse-grain thermal model for a quad-core CPU. This model takes as input the power dissipated by each of the four cores, computed by a suitable power model, and produces as output an approximation of the temperature of each of the cores, taking into account the core coupling through the heat spreader and heat sink. Although the model does not simulate the entire chip floorplan at a high granularity, it is nonetheless capable of reproducing with sufficient accuracy both the fast thermal dynamics [14] caused by the small thermal capacitance of the silicon die coupled with the non-negligible thermal resistance to the heat sink, as well as the slow thermal dynamics of the heat sink itself.

The parameters of the designed model have been fitted from experimental data on an Intel Core-i5 6600K, which has been instrumented for the purpose as follows. First of all, all thermal control policies except for the hardware thermal shutdown have been disabled, in order to capture thermal transients without policies unexpectedly altering the CPU voltage or frequency. This includes both the *Thermald* thermal control policy that is part of current Linux distributions, as well as the turbo boost policy that is part of the processor. An instrumentation program reading the core temperatures directly from the CPU Model-Specific Registers (MSRs) at a 200Hz sampling rate has been used to sidestep the rate limiting imposed by the Linux kernel on reading temperature information. This proved vital to correctly capture the fast thermal transients. Power measurements have instead been performed by means of a shunt resistor inserted in the CPU power supply path, connected to a National Instruments acquisition board. The acquired information, consisting of step responses applied to the four cores by means of microbenchmark programs designed to cause a constant and known power consumption has been used to fit and subsequently validate the thermal model. Figure 3 shows one experiment, consisting of the temperature of the last core during a sequence of step responses applied to cores 1 through 4 in sequence. As can be seen, from 0 to 300s the model can correctly reproduce the effect of thermal coupling through the heat sink caused by the

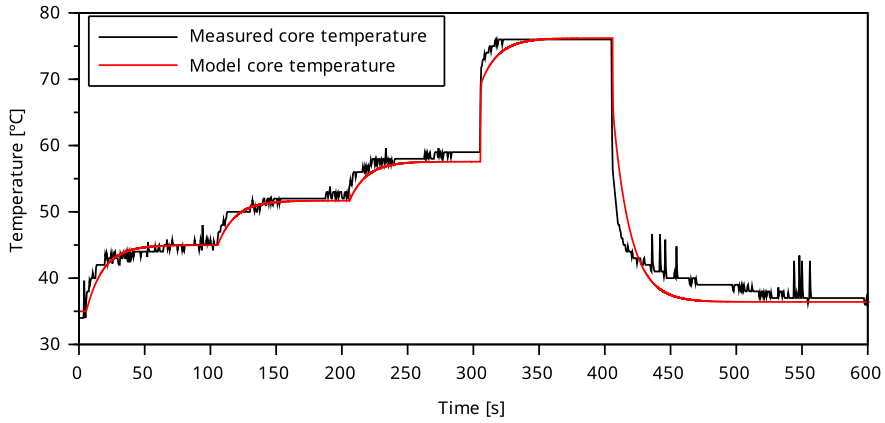


Figure 3: Comparison of the temperature estimate produced by the developed thermal model with experimental measurements.

heating of the other cores, while from 300 to 400 seconds temperature increases further as also this core is dissipating power.

Based on the performed validation, the developed thermal model provides an average error lower than 2°C, and a maximum error of 9°C, while taking less than 1μs to simulate a quad-core CPU for 5ms.

4.2. Reliability Model

In this work, we focused on *permanent faults*, which are a major issue in HPC data centres due to the large number of machines which, in turn, decreases the total Mean-Time-To-Failure (MTTF) of the whole cluster. Moreover, they dominate the overall failures in HPC [23]. Reducing the core temperature is a key goal to increase the reliability of the hardware [28]. Therefore, we implemented in DCworms the same reliability model used in the RECIPE project (see Figure 1) for the real system. This model needs to collect data on the temperature of the processors, in order to predict the probability of failures. The output of the thermal model, described in the previous Section 4.1, is used as input for the reliability model. The reliability model requires only coarse-grain temperature data, from the spatial perspective. This is an advantage, as it matches the capabilities of the ubiquitous on-chip temperature sensors made available by modern CPUs, making the policy easily implementable in a real HPC infrastructure.

The reliability model relies on the well-established Arrhenius' equation, which makes explicit the relationship between the time to failure of a component (processor in this case) and its temperature:

$$AF = A \cdot \exp\left[-\frac{\Delta H}{k} \cdot \left(\frac{1}{T_{curr}} - \frac{1}{T_{base}}\right)\right]$$

The model estimates the *acceleration factor* (AF), representing the variation of time to failure due to the variation of temperature from the test temperature

T_{base} to the current temperature T_{curr}. The parameters include the Boltzmann constant ($k = 8.617 \cdot 10^{-5} eV/K$), the *activation energy* (ΔH), which depends on the material of the component and the type of failure, and a scaling factor A. The parameter for the Intel Skylake processor are [20]: $A = 1$ and $\Delta H = 0.7eV$. Moreover, the variation of temperature has been considered in terms of additional stress, introduced by the current temperature (T_{curr}), with respect to a baseline value $T_{base} = 55°C = 328.15K$. For the setup based on DCworms, T_{curr} is provided by the thermal model. From the Arrhenius' equation, we can derive the following relation with the average failure rate [5]:

$$\lambda_{\text{real}} = AF \cdot \lambda_{\text{expected}}$$

The value $\lambda_{\text{expected}}$ is provided by the manufacturer. Therefore, reducing the core temperature (T_{curr}) makes the acceleration factor AF smaller and, in turn, the fault rate λ_{real} lower.

5. Experimental Results

In the experiments, we show how the applications and the overall hardware system can benefit from a run-time adaptive approach, with respect to static or application-driven ones. Finally, by exploiting DCworms, we provide an evaluation of the proposed system management strategy in terms of scalability over a multi-node HPC infrastructure. Furthermore, it will be shown how the reliability-performance trade-off is taken into maximum consideration through the machine and application awareness of the policy and by the possibility of setting an upper bound on the overhead due to the checkpoint task.

5.1. Application Characterization

To drive the experiments, we characterized the Cloverleaf mini-application [16] from the UK Mini-App Consortium, which employs an explicit second-order method for the resolution of compressible Euler equations, a representative application for the HPC domain. We collected power traces and performance results on the same quad-core machine used for the Thermal model in section 4.1, on both the serial and OpenMP implementations of Cloverleaf. These data are used to drive the thermal model and the simulation of the execution in DCworms.

5.2. Reliam Evaluation

We performed two sets of experiments to test the effectiveness of Reliam. The first one consisted of the characterization of the policy completion time, compared to the baseline behaviour. In the second one, we evaluated the effectiveness of the reliability decisions of the policy on a large scale. For this purpose, we exploited DCworms in order to simulate a multi-node parallel system.

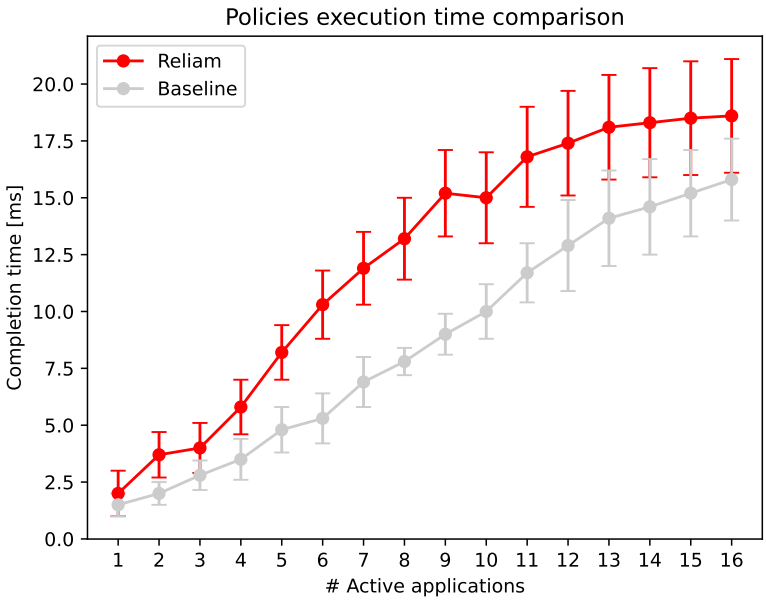


Figure 4: Comparison between mean completion time of an invocation of the Reliam and the baseline policy, varying the number of active applications.

5.2.1. Execution time

For the characterization of the Reliam policy execution time, we used the workstation already exploited for the thermal model characterization. We collected the completion time of Reliam and a baseline policy, consisting of an assignment of a fair amount of CPU quota, without considering the binding problem. We considered several scenarios, characterized by different amounts of active applications to schedule (from 1 to 16). We used the *Fluidanimate* PARSEC benchmark as workload, launched in a single-thread configuration, to process 500 frames of the *native* input set, for a running time of 100 seconds. Figure 4 shows the mean values and standard deviation of the completion time of a single invocation of the two policies, according to the different scenarios. We repeated the experiment 50 times for each scenario. From the figure, one can observe how the two curves have a logarithmic progression, although the slope of Reliam is slightly bigger than the baseline one. The maximum difference encountered in the tested cases consists of a ∼55-60% higher execution time for the Reliam policy, in the scenarios characterized by the utilization of half of the computing resources (8-9 applications). Reasonably, this is due to the fact that such scenarios determine the highest number of CPU cores being affected by thermal variations. On the contrary, this execution penalty drops to 17% for the scenarios characterization by high utilization of resources (14-16

applications). In such cases, the number of thermal variations is reasonably lower, since the fully utilized CPUs are at a homogeneously high temperature. In any case, the policy execution time does not represent a noticeable latency for the workload execution times, since performed in the background. In general, the results show that Reliam is able to actuate a reliability-aware adaptive CPU allocation in a time span of the order of a few milliseconds in all the tested cases, providing a prompt response to the applications and sensible scalability for the system.

5.2.2. Large-scale thermal results

The following experiments aimed at testing our approach at scale. In this regard, we performed two experiments employing DCworms to simulate a large-scale system to assess the impact of the Reliam policy on both homogeneous and heterogeneous clusters.

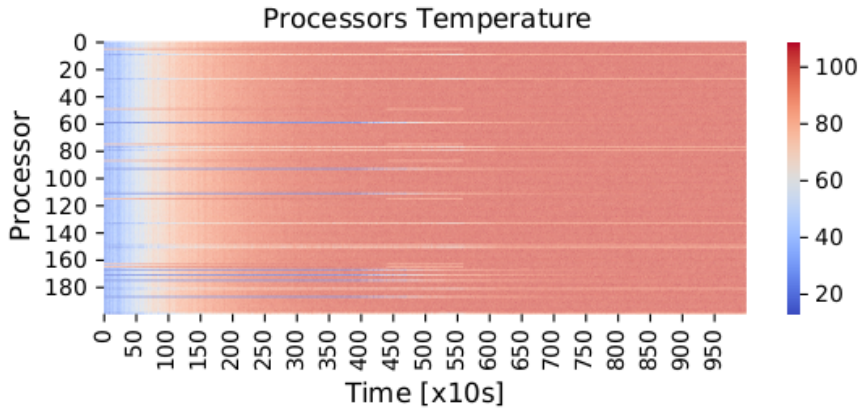
In the homogeneous computing scenario, we evaluated the impact of the resource-binding component of the policy by simulating a cluster of 128 nodes, each with 64 processors. The system is subject to a workload of 1024 jobs with randomly generated characteristics (in terms of the number of tasks and resource consumption for each task). We monitor the temperature evolution for each processor, as well as the estimated failure rate (FIT), when using the baseline configuration, without the Reliam policy, and when using a configuration that enables the Reliam policy.

Applying the policy increases the execution times, but offers better temperature and FIT trends. In fact, the interesting result is the reduction in the temperatures of the processors (temperatures are 16°C lower on average) and the reduction of FIT of the processors of 26%. In this execution, the total execution time increases by ∼10%. Looking at the impact on performance, under specific circumstances, the Reliam policy can reduce the overall execution time. The probability of this occurrence grows as the task execution times and failure probability increase, since this increases as well the probability of a task having to be restarted due to an actual failure. It is possible to understand the outcome by comparing Figures 5a, 5b, 6a, and 6b that show the temperatures and the FIT values of the baseline and of the experiments.

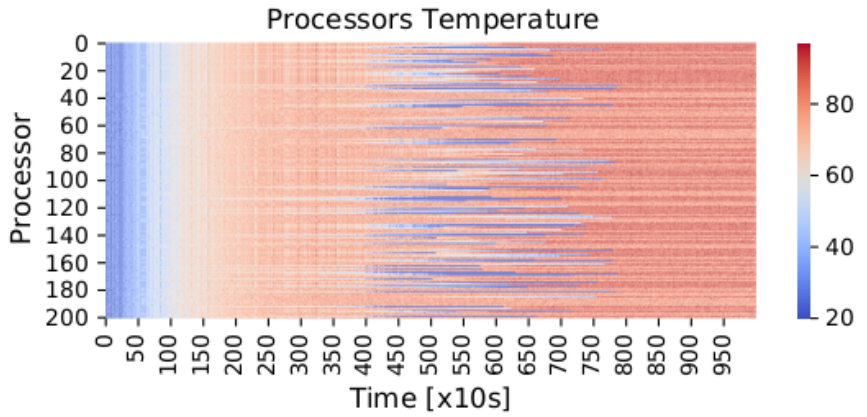
The second example demonstrates the results obtained by applying our policy to a cluster of 128 nodes of 16 processors and 1 GPU. The policy, also in this scenario is able to achieve a reduction of the processors' temperatures of an average of 15°C and a reduction of FIT of the processors of 27%, in return for a relatively minor execution time increase (9%). Figures 7a, 7b, 8a, and 8b show the trends of temperatures and FIT during the experiments in one node.

5.3. Dynamic Checkpoint Scheduler Evaluation

In this section, we report the evaluation of the overhead due to the Dynamic Checkpoint Scheduler, introduced in Section 3.2. For the execution of the Dynamic Checkpoint scheduler in the resource management framework, we used the same system exploited for testing the overhead characterization of the



(a) Baseline configuration.

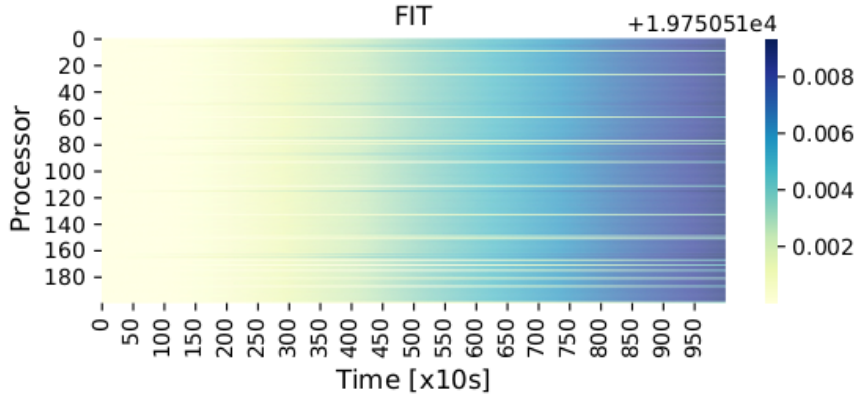


(b) Reliam enabled.

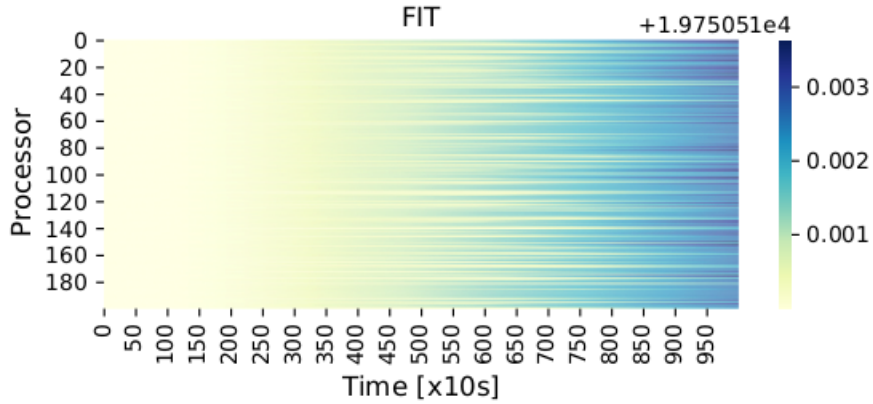
Figure 5: Evolution of temperature in a node with 128 nodes of 64 processors running the test workload in the configuration with the Reliam policy. Temperatures are reported in centigrade degrees. Only 200 randomly chosen processors are shown (the same processors in both).

Reliam policy. We ran the pseudo-application BT of the NAS Parallel Benchmark Suite considering two different workloads, the first one characterized by a memory occupancy of $0.8GB$, and the second one of $12.8GB$. The goal of the experiment was to observe the cumulative overhead obtained using the Dynamic Checkpoint Scheduler, considering both checkpoint overhead and re-execution time in the case of occurrence of failures ($T_{failure}$).

Defined T_{exc_tot}, T_{chk_tot} and $T_{rexc}(t)$, respectively, the total time spent executing the application, the total time spent performing checkpoints and the time spent re-executing the application in case of failure, we define $T_{failure}$ as



(a) Baseline configuration.



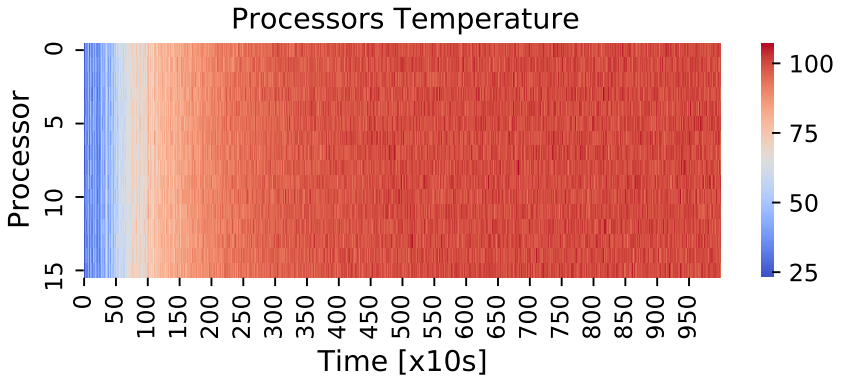
(b) Reliam enabled.

Figure 6: Evolution of the FIT for the same scenario reported in Figure 5.

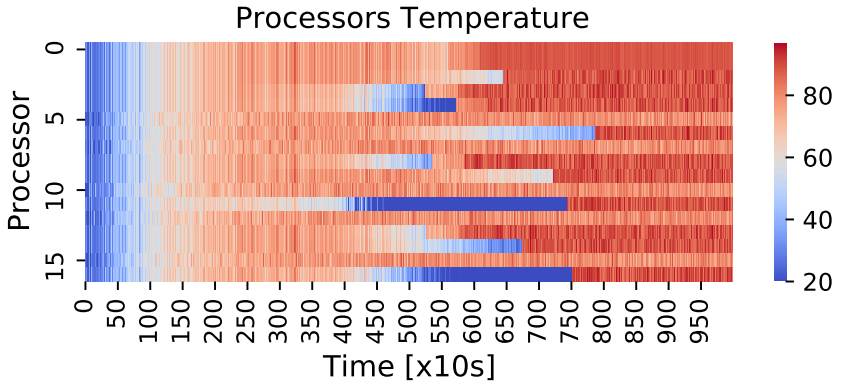
follows:

$$\begin{cases} T_{TOT} = T_{exc\_tot} + T_{chk\_tot} \\ T_{failure} = \int_0^{T_{TOT}} \lambda \cdot T_{rexc}(t)dt \end{cases} \tag{3}$$

We computed the arithmetic mean of the checkpoint times obtained during the execution of the benchmark, observing that the standard deviation never exceeds the 10% of the mean. This result allowed us to re-write, without any loss of information, T_{chk_tot} as $n * T_{chk\_mean}$, where n is the total number of



(a) Baseline configuration.



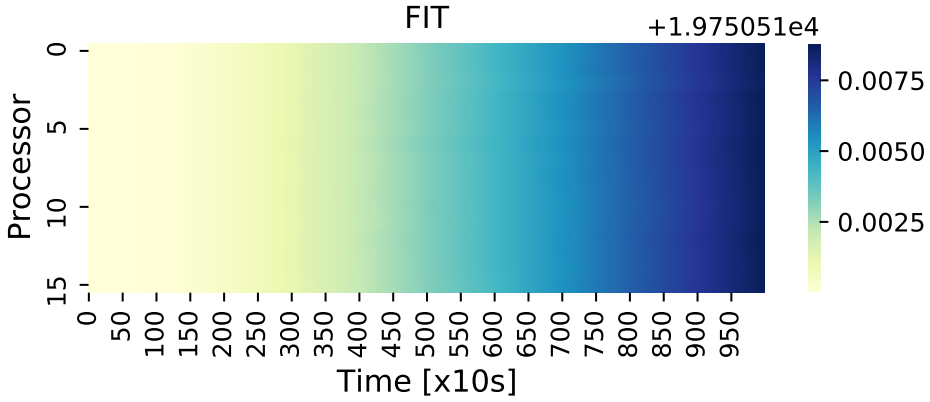
(b) Reliam enabled.

Figure 7: Evolution of temperature in a single node running the test workload in the baseline configuration (a) and in the configuration with Reliam enabled (b). Temperatures are reported in centigrade degrees ($^{\circ}C$)

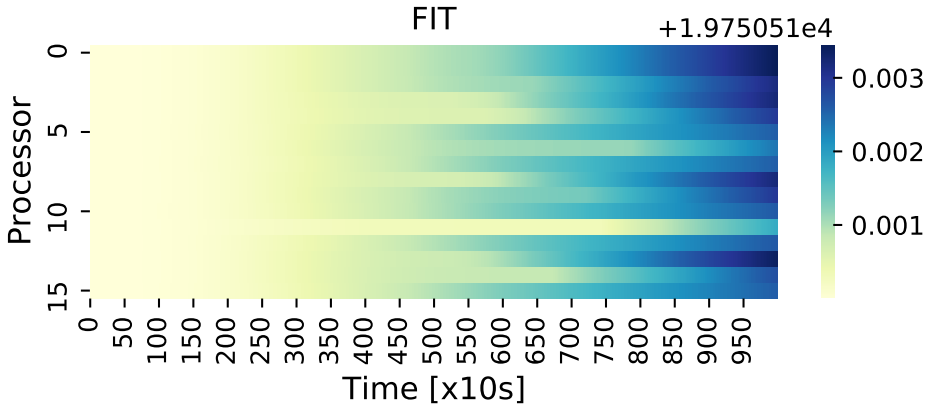
performed checkpoints, and to do the same approximation also for the time passed between two consecutive checkpoints ($T_{TOT} = n * T_{period}$).

Defined $t_{start\_exc_i}$ and $t_{end\_exc_i}$ as the starting and the ending time of the execution of the application code in period i, we know that:

$$t_{end\_exc_i} - t_{start\_exc_i} = T_{period} - T_{chk\_mean} = T_{exc\_mean} \qquad \forall i \qquad (4)$$



(a) Baseline configuration.



(b) Reliam enabled.

Figure 8: Evolution of the FIT for the same scenario reported in Figure 7.

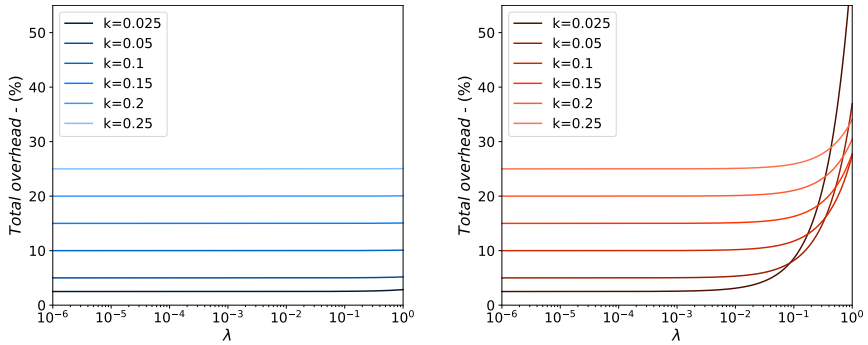
This result allows us to compute $T_{failure}$ as:

$$T_{failure} = \int_0^{T_{TOT}} \lambda \, T_{rexc}(t) dt \tag{5}$$

$$= \sum_{i=1}^{n} \int_{t_{end\_exc_i}}^{t_{start\_exc_i}} \lambda \, t_i dt_i \tag{6}$$

$$= \sum_{i=1}^{n} \lambda \, \frac{t_i^2}{2} \Big|_{t_{start\_exc_i}}^{t_{end\_exc_i}} \tag{7}$$

$$= n \, \lambda \, \frac{\mathbf{E}[T_{exc}]^2}{2} \tag{8}$$



(a) Memory occupancy: 0.8GB.

(b) Memory occupancy: 12.8GB.

Figure 9: sum of $T_{failure}$ and T_{chk_tot} normalized on application total execution time without failures.

Figures 9a and 9b show the sum of $T_{failure}$ and T_{chk_tot} normalized on the application code total execution time in absence of failures. We selected six different checkpoint overhead upper bounds (k) and observed the trend of the total overhead varying the failure rate (λ) of the system, expressed in FIT (hour basis). The experiments show that, for a smaller workload, Figure 9a, the total overhead is always proportional to the choice of k, for all considered values of λ. However, Figure 9b shows that, for bigger workloads, the previously observed trend inverts when the system is highly unreliable ($\lambda \approx 10^{-1}$). This happens because, if the failure rate is high, re-execution times may impact the performance more than the overhead of a more frequent checkpoint. Moreover, the bigger the workload, the lower the failure rate at which the trend inversion is found. It follows that, in the case of significant workloads, the choice of k must be consistent with the reliability level of the system.

By looking at the results reported in [9], for example, we can approximately compute that their methodology set the reliability of the system in the $10^{-2} \leq \lambda \leq 10^{-1}$ range. Assuming to run, on the same system, a workload that is compatible with the one we used to profile the overhead shown in Figure 9b, we can state that by setting a value $k \approx 0.05$, we can get the same achievement with an execution overhead $\leq 5\%$. Higher values of k would dramatically delay the finish time of the workload execution.

6. Conclusion

In this work, we introduced a reliability-aware approach to the resource management of HPC systems, integrating resource allocation policies, monitoring interfaces and models. Our work introduces fine-grained control actions, in coordination with the usage of the checkpoint/restore mechanism. In this regard, our approach aims at keeping under control the overhead introduced by the checkpoint operations, by monitoring the time required with respect to the

overall execution time of the applications and tuning the checkpoint period on a per-application basis, considering the maximum overhead value set by the user. Furthermore, we characterized the overhead introduced by the execution of the management policy itself, showing the sustainability of a real multi-core-based system. The results, obtained by deploying the framework on a simulated HPC infrastructure, have shown how we are able to achieve a significant reduction of temperature, with a consequent dramatic impact on the reliability of the processing resources, and thus of the overall system.

In future developments, we aim at including the FIT estimation in the checkpoint rate tuning, such that, for tight reliability predictions, the dynamic checkpoint scheduler could also consider the restore overhead while evaluating the checkpoint rate value to set.

7. Acknowledgements

This work has received funds from the projects TEXTAROSSA (G.A. 956831), as part of the H2020 and EuroHPC initiatives, and the European Pilot (G.A. 101034126), as part of the European High-Performance Computing Joint Undertaking (JU). Other funds come from the National (Italian) Resilience and Recovery Plan (PNRR) and the National Center for HPC, Big Data and Quantum Computing.

References

[1] Giovanni Agosta, William Fornaciari, David Atienza, Ramon Canal, Alessandro Cilardo, José Flich Cardo, Carles Hernandez Luz, Michal Kulczewski, Giuseppe Massari, Rafael Tornero Gavilá, and Marina Zapater. The recipe approach to challenges in deeply heterogeneous high performance systems. *Microprocessors and Microsystems*, 77:103185, 2020.

[2] Marco Aldinucci et al. The italian research on hpc key technologies across eurohpc. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 178–184, 2021.

[3] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. Effective runtime resource management using linux control groups with the BarbequeRTRM framework. *ACM Trans. Embed. Comput. Syst.*, 14(2):39:1–39:17, March 2015.

[4] Florian Berberich, Janina Liebmann, Jean-Philippe Nominé, Oriol Pineda, Philippe Segers, and Veronica Teodor. European HPC Landscape. In *2019 15th International Conference on eScience (eScience)*, pages 471–478. IEEE, 2019.

[5] Joseph B. Bernstein. Chapter 3 - failure mechanisms. In Joseph B. Bernstein, editor, *Reliability Prediction from Burn-In Data Fit to Reliability Models*, pages 31–48. Academic Press, Oxford, 2014.

[6] Muhammad Tayyab Chaudhry, Teck Chaw Ling, Atif Manzoor, Syed Asad Hussain, and Jongwon Kim. Thermal-aware scheduling in green data centers. *ACM Comput. Surv.*, 47(3), feb 2015.

[7] Nasim Donyagard Vahed, Mostafa Ghobaei-Arani, and Alireza Souri. Multiobjective virtual machine placement mechanisms using nature-inspired metaheuristic algorithms in cloud environments: A comprehensive review. *International Journal of Communication Systems*, 32(14):e4068, 2019. e4068 IJCS-19-0062.R1.

[8] William Fornaciari, Giovanni Agosta, David Atienza, Carlo Brandolese, Leila Cammoun, Luca Cremona, Alessandro Cilardo, Albert Farres, José Flich, Carles Hernandez, Michal Kulchewski, Simone Libutti, José Maria Martínez, Giuseppe Massari, Ariel Oleksiak, Anna Pupykina, Federico Reghenzani, Rafael Tornero, Michele Zanella, Marina Zapater, and Davide Zoni. Reliable power and time-constraints-aware predictive management of heterogeneous exascale systems. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '18, pages 187–194, New York, NY, USA, 2018. ACM.

[9] N. R. Gottumukkala, C. B. Leangsuksun, N. Taerat, R. Nassar, and S. L. Scott. Reliability-aware resource allocation in hpc systems. In *2007 IEEE International Conference on Cluster Computing*, pages 312–321, 2007.

[10] Narasimha Raju Gottumukkala, Chokchai Box Leangsuksun, Narate Taerat, Raja Nassar, and Stephen L Scott. Reliability-aware resource allocation in hpc systems. In *2007 IEEE International Conference on Cluster Computing*, pages 312–321. IEEE, 2007.

[11] L. Huang, F. Yuan, and Q. Xu. Lifetime reliability-aware task allocation and scheduling for mpsoc platforms. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 51–56, 2009.

[12] Arman Iranfar, Federico Terraneo, William Andrew Simon, Leon Dragić, Igor Piljić, Marina Zapater, William Fornaciari, Mario Kovač, and David Atienza. Thermal characterization of next-generation workloads on heterogeneous MPSoCs. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 286–291, 2017.

[13] K. Kurowski, A. Oleksiak, W. Piątek, T. Piontek, A. Przybyszewski, and J. Węglarz. Dcworms – a tool for simulation of energy efficiency in distributed computing infrastructures. *Simulation Modelling Practice and Theory*, 39:135 – 151, 2013. S.I.Energy efficiency in grids and clouds.

[14] Alberto Leva, Federico Terraneo, Irene Giacomello, and William Fornaciari. Event-Based Power/Performance-Aware Thermal Management for High-Density Microprocessors. *IEEE Transactions on Control Systems Technology*, 26(2):535–550, 2018.

[15] Yudan Liu, Raja Nassar, Chockchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen Scott. A reliability-aware approach for an optimal checkpoint/restart model in hpc environments. In *2007 IEEE International Conference on Cluster Computing*, pages 452–457. IEEE, 2007.

[16] Andrew Mallinson, David A Beckingsale, WP Gaudin, JA Herdman, JM Levesque, and Stephen A Jarvis. Cloverleaf: Preparing hydrodynamics codes for exascale. *The Cray User Group*, 2013, 2013.

[17] Mohammed Sultan Mohammed, Ahlam Khaled Al-Dhamari, Ab Al-Hadi ab Rahman, Norlina Paraman, Ali A.M. Al-Kubati, and M. N. Marsono. Temperature-aware task scheduling for dark silicon many-core system-on-chip. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 1–5, 2019.

[18] Mina Naghshnejad and Mukesh Singhal. A hybrid scheduling platform: a runtime prediction reliability aware scheduling platform to improve hpc scheduling performance. *The Journal of Supercomputing*, 76(1):122–149, 2020.

[19] Nichamon Naksinehaboon, Yudan Liu, Chokchai Leangsuksun, Raja Nassar, Mihaela Paun, and Stephen L Scott. Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783–788. IEEE, 2008.

[20] RECIPE Project. RECIPE Fault Prediction Tools. Deliverable D3.3, European Union's Horizon 2020, 2020.

[21] Federico Reghenzani, Simone Formentin, Giuseppe Massari, and William Fornaciari. A constrained extremum-seeking control for cpu thermal management. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, CF '18, page 320–325, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Osman Sarood, Esteban Meneses, and Laxmikant V Kale. A'cool'way of improving the reliability of hpc machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[23] Vilas Sridharan and Dean Liberty. A study of dram failures in the field. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[24] Federico Terraneo, Alberto Leva, William Fornaciari, Marina Zapater, and David Atienza. 3D-ICE 3.0: efficient nonlinear MPSoC thermal simulation with pluggable heat sink models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[25] Craig Walker, Braeden Slade, Gavin Bailey, Nicklaus Przybylski, Nathan DeBardeleben, and William M. Jones. Exploring the tradeoff between reliability and performance in hpc systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2021.

[26] Jing Xu and José Fortes. A multi-objective approach to virtual machine management in datacenters. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, page 225–234, New York, NY, USA, 2011. Association for Computing Machinery.

[27] Jing Xu and Jose A. B. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, pages 179–188, 2010.

[28] Junlong Zhou, Xiaobo Sharon Hu, Yue Ma, Jin Sun, Tongquan Wei, and Shiyan Hu. Improving availability of multicore real-time systems suffering both permanent and transient faults. *IEEE Transactions on Computers*, 68(12):1785–1801, 2019.