DOI: 10.14529/jsfi150203

Resilience within Ultrascale Computing System: Challenges and Opportunities from Nesus Project

Pascal Bouvry[1], *Rudolf Mayer*[2], *Jakub Muszyński*[1], *Dana Petcu*[3], *Andreas Rauber*[4], *Gianluca Tempesti*[5], *Tuan Trinh*[6], *Sébastien Varrette*[1]

Although resilience is already an established field in system science and many methodologies and approaches are available to deal with it, the unprecedented scales of computing, of the massive data to be managed, new network technologies, and drastically new forms of massive scale applications bring new challenges that need to be addressed. This paper reviews the challenges and approaches of resilience in ultrascale computing systems from multiple perspectives involving and addressing the resilience aspects of hardware-software co-design for ultrascale systems, resilience against (security) attacks, new approaches and methodologies to resilience in ultrascale systems, applications and case studies.

Keywords: high performance computing, fault tolerance, algorithm-based fault tolerance, extreme data, evolutionary algorithm, ultrascale computing system.

Introduction

Ultrascale computing is a new computing paradigm that comes naturally from the necessity of computing systems that should be able to handle massive data in possibly very large scale distributed systems, enabling new forms of applications that can serve a very large amount of users and in a timely manner that we have never experienced before.

Ultrascale Computing Systems (UCSs) are envisioned as large-scale complex systems joining parallel and distributed computing systems that will be two to three orders of magnitude larger than today's systems (considering the number of Central Processing Unit (CPU) cores). It is very challenging to find sustainable solutions for UCSs due to their scale and a wide range of possible applications and involved technologies. For example, we need to deal with cross fertilization among HPC, large-scale distributed systems, and big data management. One of the challenges regarding sustainable UCSs is resilience. Traditionally, it has been an important aspect in the area of critical infrastructure protection (e.g. traditional electrical grid and smart grids). Furthermore, it has also become popular in the area of information and communication technology (ICT), ICT systems, computing and large-scale distributed systems. In essence, resilience is an ability of a system to efficiently deliver and maintain (in a timely manner) a correct service despite failures and changes. It is important to emphasize this term in comparison with a closely related "fault tolerance". The latter indicates only a well-defined behaviour of a system once an error occurs. For example, a system is resilient to an effect on an error (in one of its components) if it continues correct operation and service delivery (possibly degraded in some way). Whereas, it is fault tolerant to the error when it is able to detect and notify about the existence of the problem with possible recovery to the correct state.

[1] University of Luxembourg, Esch-sur-Alzette, Luxembourg
[2] Secure Business Austria, Vienna, Austria
[3] West University of Timişoara, Timişoara, Romania
[4] Vienna University of Technology, Vienna, Austria
[5] University of York, York, UK
[6] Budapest University of Technology and Economics, Budapest, Hungary

The existing practices of dependable design deal reasonably well with achieving and predicting dependability in systems that are relatively closed and unchanging. Yet, the tendency to make all kinds of large-scale systems more interconnected, open, and able to change without new intervention by designers, makes existing techniques inadequate to deliver the same levels of dependability. For instance, evolution of the system itself and its uses impairs dependability: new components "create" system design faults or vulnerabilities by feature interaction or by triggering pre-existing bugs in existing components; likewise, new patterns of use arise, new interconnections open the system to attack by new potential adversaries, and so on.

Many new services and applications will be able to get advantage of ultrascale platforms such as big data analytics, life science genomics and HPC sequencing, high energy physics (such as QCD), scalable robust multiscale and multi-physics methods and diverse applications for analysing large and heterogeneous data sets related to social, financial, and industrial contexts. These applications have a need for Ultrascale Computing Systems (UCSs) due to scientific goals to simulate larger problems within a reasonable time period. However, it is generally agreed that applications will require substantial rewriting in order to scale and benefit from UCSs.

In this paper, we aim at providing an overview of Ultrascale Computing Systems (UCSs) and highlighting open problems. This includes:

- Exploring and reviewing the state-of-the-art approaches of continuous execution in the presence of failures in UCSs.
- Techniques to deal with hardware and system software failures or intentional changes within the complex system environment.
- Resilient, reactive schedulers that can survive errors at the node and/or the cluster-level, cluster-level monitoring and assessment of failures with pro-active actions to remedy failures before they actually occur (like migrating processes [13, 58], virtual machines [49], etc.), and malleable applications that can adapt their resource usage at run-time.

In particular, we approach the problem from different angles including fundamental issues such as reproducibility, repeatability and resiliency against security attacks; application-specific challenges such as hardware and software issues with big data, cloud-based cyber physical systems.The paper then also discusses new opportunities in providing and supporting resilience in ultrascale systems.

This paper is organized as follows: the section 1 reviews the basic notions of faults, Fault Tolerance and robustness. Then, several key issues need to be tackled to ensure a robust execution on top of an UCS system. The section 2 focuses on recent trends as regards the resilience of large scale computing systems, focusing on hardware failures (see §2.1) and on Algorithm-Based Fault Tolerance (ABFT) techniques where the fault tolerance scheme is tailored to the algorithm *i.e.* the application run. We will see that at this level, Evolutionary Algorithms (EAs) present all the characteristics to handle natively faulty executions, even at the scale foreseen in UCSs systems. Then, the section 3 will review the challenges linked to the notion of repeatibility and reproducibility in UCSs. The final section concludes the paper and provides some future directions and perspectives opened by this study.

1. Faults, Fault Tolerance and Robustness

Due to their inherent scale, UCSs are naturally prone to errors and failures which are no longer rare events [11, 12, 50, 52]. There are many sources of *faults* in distributed computing and they are inevitable due to the defects introduced into the system at the stages of its design,

construction or through its exploitation (e.g. software bugs, hardware faults, problems with data transfer) [4, 11, 12, 52]. A fault may occur by a deviation of a system from the required operation leading to an *error* (for instance a software bug becomes apparent after a subroutine call). This transition is called a fault activation, i.e. a *dormant* fault (not producing any errors) becomes *active*. An error is *detected* if its presence is indicated by a message or a signal, whereas not detected, present errors are called *latent*. Errors in the system may cause a (service) *failure* and depending on its type, successive faults and errors may be introduced (*error/failure propagation*). The distinction between faults, errors and failures is important because these terms create boundaries allowing analysis and coping with different threats. In essence, faults are the cause of errors (reflected in the state) which without proper handling may lead to failures (wrong and unexpected outcome). Following these definitions, *fault tolerance* is an ability of a system to behave in a well-defined manner once an error occurs.

1.1. Fault models for distributed computing

There are five specific fault models relevant in distributed computing: *omission*, *duplication*, *timing*, *crash*, and *byzantine failures* [53].

Omission and *duplication* failures are linked with problems in communication. Send-omission corresponds to a situation, when a message is not sent; receive-omission — when a message is not received. Duplication failures occur in the opposite situation — a message is sent or received more than once.

Timing failures occur when time constraints concerning the service execution or data delivery are not met. This type is not limited to delays only, since too early delivery of a service may also be undesirable.

The *crash* failure occurs in four variants, each additionally associated with its persistence. Transient crash failures correspond to the service restart: amnesia-crash (the system is restored to a predefined initial state, independent on the previous inputs), partial-amnesia-crash (a part of the system stays in the state before the crash, where the rest is reset to the initial conditions), and pause-crash (the system is restored to the state it had before the crash). Halt-crash is a permanent failure encountered when the system or the service is not restarted and remains unresponsive.

The last model — *byzantine* failure (also called *arbitrary*) — covers any (very often unexpected and inconsistent) responses of a service or a system at arbitrary times. In this case, failures may emerge periodically with varying results, scope, effects, etc. This is the most general and serious type of failure [53].

1.2. Dependable computing

Faults, errors and failures are *threats* to system's *dependability*. A system is described as dependable, when it is able to fulfil a contract for the delivery of its services avoiding frequent downtimes caused by failures.

Identification of threats does not automatically guarantee dependable computing. For this purpose, four main groups of appropriate methods have been defined [4]: *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*. As visible in fig. 1, all of them can be analysed from two points of view — either as means of avoidance/acceptance of faults or as approaches to support/assess dependability. Fault tolerance techniques aim to reduce (or even eliminate) the

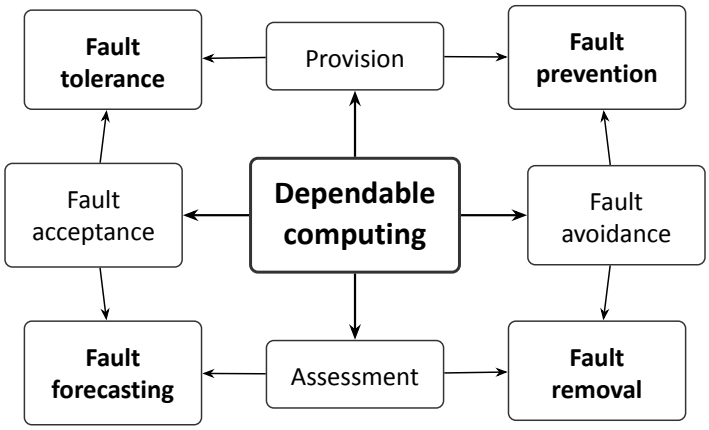

Figure 1. Means for dependable computing

amount of service failures in the presence of faults. The main goal of *fault prevention* methods is to minimize the number of faults occurred or introduced through usage and enforcement of various policies (concerning usage, access, development etc.) The next group — *fault removal* techniques — is concentrated around testing and verification (including formal methods). Finally, *fault forecasting* consists of means to estimate occurrences and consequences of faults (at a given time and later).

1.3. Fault tolerance

Fault tolerance techniques may be divided into two main, complementary categories [4]: *error detection*, and *recovery*. Error detection may be performed during normal service operation or while it is suspended. The first approach in this category — *concurrent detection* — is based on various tests carried out by components (software and/or hardware) involved in the particular activity or by elements specially designated for this function. For example, a component may calculate and verify checksums for the data which is processed by it. On the other hand, a firewall is a good illustration of a designated piece of hardware (or software) oriented on detection of intrusions and other malicious activities. *Preemptive detection* is associated with the maintenance and diagnostics of a system or a service. The focus in this approach is laid on identification of latent faults and dormant errors. It may be carried out at a system startup, at a service bootstrap, or during special maintenance sessions.

After an error of a fault is detected, recovery methods are applied. Depending on the problem type, *error or fault handling* techniques are used. The first group is focused on elimination of errors from the system state, while the second are designed to prevent activation of faults. In [4], the specific methods are separated from each other, where in practice this boundary is fuzzy and depends on the specific service and system types.

Generally, error handling is solved through:

1. *Rollback* [22] — the system is restored to the last known, error-free state. The approach here depends on a method used to track the changes of the state. A well known technique is *checkpointing* — the state of a system is saved periodically (e.g. the snapshot of a process is stored on a disk) as a potential recovery point in the future. Obviously, this solution is not straightforward in the case of distributed systems and there are many factors to consider. In such environment, checkpointing can be coordinated or not — with differences in reliability and the cost of synchronisation of the distributed components (for details see: [18, 31, 53]).

Rollback can be also implemented through the *message logging*. In this case, the communication between the components is tracked rather than their state. In case of an error, the system is restored by *replaying* the historical messages, allowing it to reach global consistency [53]. Sometimes both techniques are treated as one, as usually they complement each other.

2. *Rollforward* — the current, erroneous system state is discarded and replaced with a one newly created and initialised.
3. *Compensation* — solutions based on components' *redundancy* and *replication*, sometimes referred to as *fault masking*. In the first case, additional components (usually hardware) are kept in reserve [31]. If failures or errors occur, they are used to compensate the losses. For example, a connection to the Internet of a cloud platform should be based on solutions from at least two different Internet Service Providers (ISPs).

   Replication is based on the dispersion of multiple copies of the service components. A schema with replicas used only for the purpose of fault tolerance is called a *passive (primary-backup) replication* [31]. On the other hand, an *active replication* is when the replicas participate in providing the service, leading to increased performance and applicability of load balancing techniques [31]. Coherence is the major challenge here, and various approaches are used to support it. For instance, read-write protocols are crucial in active replication, as all replicas have to have the same state. Another worth to note example is clearly visible in volunteer-based platforms. An appropriate selection policy of the correct service response is needed when replicas return different answers, i.e. a method to reach quorum consensus is required [31].

These techniques are not exclusive and can be used together. If the system can not be restored to a correct state thanks to the compensation, rollback may be attempted. If this fails, then rollforward may be used.

The above mentioned methods may be referred to as *general-purpose* techniques. These solutions are relatively generic, which aid their implementation for almost any distributed computation. It is also possible to delegate responsibility for fault tolerance to the service (or application) itself, allowing tailoring the solution for specific needs — therefore forming an *application-specific* approach. A perfect example in this context is ABFT, originally applied to distributed matrix operations [14], where original matrices are extended with checksums before being scattered among the processing resources. This allows detection, location and correction of certain miscalculations, creating a *disk-less checkpointing* method. Similarly, in certain cases it is possible to continue the computation or the service operation despite the occurring errors. For instance, unavailable resource resulting from a crash-stop failure can be excluded from further use. In this work, the idea will be further analysed and extended to the context of byzantine errors and the nature-inspired distributed algorithms.

Fault handling techniques are applied after the system is restored to an error-free state (using the methods described above). As the aim now is to prevent future activation of detected faults, four subgroups according to the intention of the operation may be created. These are [4]: *diagnosis* (the error(s) are identified and their source(s) are located), *isolation* (faulty components are logically or physically separated and excluded from the service), *reconfiguration* (the service/platform is reconfigured to substitute or bypass the faulty elements), and *reinitialization* (the configuration of the system is adapted to the new conditions).

1.4. Robustness

When a given system is resilient to a given type of fault, one generally claims that this system is *robust*. Yet defining rigorously robustness is not an easy task and many contributions come with their own interpretation of what robustness is. Actually, there exists a systematic framework that permits to define a robust system unambiguously. In fact, this should be probably applied to any system or approach claiming to propose a fault-tolerance mechanism. This framework, formalized in [57], answers the following three questions:

1. What behavior of the system makes it robust?
2. What uncertainties is the system robust against?
3. Quantitatively, exactly how robust is the system?

The first question is generally linked to the technique or the algorithm applied. The second — explicitly lists the type of faults or disturbing elements targeted by the system. Answering it is critical to delimit the application range of the designed system and avoid counter examples selected in a context not addressed by the robust mechanism. The third question is probably the most difficult to answer, and at the same time the most vital to characterize the limits of the system. Indeed, there is nearly always a threshold on the error/fault rate above which the proposed infrastructure fails to remain robust and breaks (in some sense).

2. Resiliency within UCSs: From Hardware to ABFT Trends

2.1. Lessons Learned from Big Data Hardware

One of the direct consequences of the treatment of big data is, clearly, the requirement for extremely high processing power. And whereas research in the big data domain does not traditionally include research in processor and computer architecture, there is a clear correlation between the advances in the two domains. While it is obviously difficult to predict future developments in processing architectures with high accuracy, we have identified two major trends that are likely to affect big data processing: the development of *many-core devices* and *hardware/software codesign*.

The many-core approach represents a step-change in the number of processing units available either in single devices or in tightly-coupled arrays. Exploiting techniques and solutions derived from the Network-on-Chip (NoC) [32] and Graphical Processing Units (GPU) areas, many-core systems are likely to have a considerable impact on application development, pushing towards distributed memory and data-flow computational models. At the same time, the standard assumption of "more tasks than processors" will be loosened (or indeed inverted), reducing to some extent the complexity of processes such as task mapping and load balancing.

Hardware/software co-design implies that applications will move towards a co-synthesis of hardware and software: the compilation process will change to generate at the same time the code to be executed by a processor and one or more hardware co-processing units to accelerate computation. Intel and ARM have already announced alliances with Altera and Xilinx[7], respectively, to offer tight coupling between their processors and reconfigurable logic, while Microsoft recently introduced the reconfigurable Catapult system to accelerate its Bing servers [47].

These trends, coupled with the evolution of VLSI fabrication processes (the sensitivity of a device to faults increases as feature size decreases), introduce new challenges to the application

[7] `http://www.eejournal.com/archives/articles/20140624-intel`.

of fault tolerance in the hardware domain. In addition to increasing the probability of fabrication defects (not directly relevant to this article), the heterogeneous nature of these systems and their extreme density represent major challenges to reliability. Indeed, the notion of *hardware fault* itself is being affected and extended to include a wider variety of effects, such as variability and power/heat dissipation.

This section does not in any way claim to represent an exhaustive survey of this very complex area, nor even a thorough discussion of the topic, but rather wants to provide a brief "snapshot" of a few interesting approaches to achieve fault tolerance in hardware, starting with a brief outline of some key concepts and fundamental techniques.

2.2. Fault tolerance in digital hardware

One of the traditional classification methods subdivides online faults in hardware systems (i.e. faults that occur during the lifetime of a circuit, rather than at fabrication) into two categories: *permanent* and *transient* (a third category, *intermittent* faults, is outside the scope of this discussion).

Permanent faults are normally introduced by irreversible physical damage to a circuit (for example, short circuits). Rather common in fabrication, they are rare in the lifetime of a circuit, but become increasingly less so as circuits age. Once a permanent fault appears, it will continue to affect the operation of the circuit forever.

Transient faults have limited duration and will disappear with time. By far the most common example of transient faults is *Single-Event Upsets* (SEU), where radiation causes a change of state within a memory element in a circuit.

This distinction is highly relevant in the context of fault tolerance, defined as the ability of a system to operate correctly in the presence of faults. Generally, the design of a fault tolerant hardware system involves four successive steps [1]:

1. *Fault detection*: can the system detect the presence of a fault?
2. *Fault diagnosis or localization*: can the system identify (as precisely as needed) the exact nature and location of a fault?
3. *Fault limitation or containment*: can the impact of the fault on the operation of the system be circumscribed so that no irreversible damage (to the circuit or to the data) results?
4. *Fault repair*: can the functionality of the system be recovered?

While there is usually no significant difference between transient and permanent faults in the first three steps, the same does not apply to the last step: transient faults can allow the full recovery of the circuit functionality, whereas *graceful degradation* (e.g., [15]) is normally the objective in the case of permanent faults in the system.

2.3. Fundamental techniques

Any implementation of fault tolerance (or indeed of fault detection) in hardware implies, directly or indirectly, the use of *redundancy*. Specific applications of redundancy, however, vary significantly depending on the features of the hardware system. In general, three "families" of redundant techniques can be identified. Once again, the examples presented in this section are not meant to be exhaustive, but simply to illustrate the different ways in which redundancy can be applied in the context of fault tolerance in hardware systems.

2.3.1. Data or information redundancy

This type of techniques relies on the use of non-minimal coding to represent the data in a system. By far the most common implementation of data redundancy implies the use of *error detecting codes* (EDC), when the objective is fault detection, and of *error correcting codes* (ECC), when the objective is fault tolerance [1].

It is worth highlighting that, even though these techniques rely on information redundancy, they also imply considerable hardware overhead, not only due to the requirement for additional storage (due to the non-minimal encoding), but also because the computation of the additional redundant bits implies the presence of (sometimes significant) additional logic.

2.3.2. Hardware redundancy

Hardware redundancy techniques exploit additional resources more directly to achieve fault detection or tolerance. In the general case, the best-known hardware redundancy approaches exploit duplication (Double Modular Redundancy, or DMR) for fault detection or triplication (Triple Modular Redundancy, or TMR) for fault tolerance.

TMR in particular is a widely used technique for safety critical systems: three identical systems operate on identical data, and a 2-out-of-3 voter is used to detect faults in one system and recover the correct result from the others. In its most common implementations (for example, in space missions), TMR is usually applied to complete systems, but the technique can operate at all levels of granularity (for example, it would be possible, if terribly inefficient, to design TMR systems for single logic gates).

2.3.3. Time redundancy

This type of approaches relies, generally speaking, on the repetition of computation and the comparison of the results between the different runs. In the simplest case, the same computation is repeated twice to generate identical results, allowing the detection of SEU. More sophisticated (but less generally applicable) approaches introduce differences in two executions (e.g. by inverting input data or shifting operands) in order to be able to detect permanent faults as well.

It is worth noting that time redundancy techniques are rarely used when fault tolerance is sought (being essentially limited to detection) but in theory can be extended to allow it in case of transient faults.

2.4. Fault tolerant design

In the introduction to this section, we highlighted how the heterogeneity and density of a type of devices that are likely to become relevant in big data treatment complicates considerably the task of achieving fault tolerant behaviour in hardware.

In particular, the heterogeneity introduced by the presence of programmable logic and the complexity of many-core devices implies that the notion of a single approach to fault tolerance applicable to every component of a system will have to be replaced by *ad-hoc* techniques. What follows is a short list of the main components of a complete system, followed by a brief analysis of their fault tolerance requirements and a few examples of approaches developed to achieve this goal, in order to illustrate some of the issues and difficulties that will have to be met.

2.4.1. Memories

Memory elements are probably the hardware components that require the highest degree of fault tolerance: their extremely regular structure implies that transistor density in memories is substantially greater than in any other device (the largest memory device commercial available in 2015 reaches a transistor count of almost 140 billion, compared for example to 4.3 billion of the largest processor). This level of density has resulted in the introduction of fault tolerant features even in commonly available commercial memories.

Reliability in memories takes essentially two forms: to protect against SEUs, the use or redundant ECC bits associated with each memory word is common and well-advertised [39], while marginally less known is the use of spare memory locations to replace permanently damaged ones. The latter technique, used extensively at fabrication for laser-based permanent reconfiguration, has also been applied in an on-line self-repair setting [16].

2.4.2. Programmable logic

Programmable logic devices (generally referred to as Field Programmable Gate Arrays) are regular circuits that can reach extremely high transistor counts. In 2015, the largest commercial FPGA device (the Virtex-Ultrascale XCVU440 by Xilinx) contains more than 20 billion transistors.

The regularity of FPGAs has sparked a significant amount of research into self-testing and self-repairing programmable devices since the late 1990s [2, 34, 37], but to the best of our knowledge this research has yet to impact consumer products (even considering potential fabrication-time improvement measures similar to those described in the previous section for memories), with the exception of radiation-hardening for space applications.

In reality, the relationship between programmable logic and fault tolerance would merit a more complete analysis, since the interplay between the fabric of the FPGA itself and the circuit that is implemented within the fabric can lead to complex interactions. Such an analysis is however beyond the scope of this article. Interestingly in this context, even though its FPGA fabric itself does not appear to contain explicit features for fault-tolerance, Xilinx supports a design tool to allow a degree of fault-tolerance in implemented designs through its Isolation Design Flow, specifically aimed at fault containment.

2.4.3. Single processing cores

The main driving force in the development of high-performance processors has been, until recently at least, sheer computational speed. In the last few years, power consumption has become an additional strong design consideration, particularly since the pace of improvements in performance has started to slow. Since fault tolerance, with its redundancy requirements, has negative implications both for performance and for power consumption, relatively little research into fault tolerant cores has reached the consumer market.

The situation is somewhat different outside of the high-performance end of the spectrum, where examples of processors specifically designed for fault tolerance exist (for example, the NGMP processor developed on behalf of the European Space Agency [3] or the Cortex-R series by ARM), demonstrating at least the feasibility of such implementations.

More recently, the RAZOR approach [23] represents a fault tolerance technique aimed specifically at detecting (and possibly correcting) timing errors within processor pipelines using a particular kind of time redundancy approach that exploits delays in the clock distribution lines.

2.4.4. On-chip networks

Networks are a crucial element of any system where processors have to share information, and therefore represent a fundamental aspect not only of many-core devices, but also of any multiprocessor system. Often rivalling in size and complexity with the processing units themselves, networks and their routers have traditionally been a fertile ground for research on fault tolerance.

Indeed, even when limiting the scope of the investigation to on-chip networks, numerous books and surveys exist that classify, describe, and analyse the most significant approaches to fault tolerance (for example, [7, 45, 48]). Very broadly, most of the fundamental redundancy techniques have been applied, in one form or another, to the problem of implementing fault-tolerant on-chip networks, ranging from data redundancy (e.g. parity or ECC encoding of transmitted packets), through hardware redundancy (e.g. additional routing logic), to time redundancy (e.g. repeated data transmission).

2.4.5. Many-Core arrays

An accurate analysis of fault tolerance in many-core devices is of course hampered by the lack of commercially-available devices (the Intel MIC Architecture, based on Xeon Phi co-processors, is a step in this direction but at the moment is limited to a maximum of 61 cores and relies on conventional programming models within a coarse-grained architecture). Using transistor density as a rough indicator of the fault sensitivity of a device (keeping in mind that issue related to heat dissipation can be included in the definition), it is no surprise that fault tolerance is generally considered as one of the key enabling technologies for this type of device: once again, the regular structure of many-core architecture is likely to have a significant impact on transistor count. Today, for example, transistor count in GPUs (the commercial devices that, arguably, bear the closest resemblance to many-core systems, both for the number of cores and for the programming model) is roughly twice that of Intel processors using similar fabrication processes, and even in the case of Intel this type of density was achieved only in multi-core (and hence regular) devices.

The lack of generally accessible many-core platforms implies that most of the existing approaches to fault-tolerance in this kind of systems remain at a somewhat higher abstraction layer and typically rely on mechanisms of task remapping through OS routines or dedicated middleware [9, 10, 33, 59]. Specific hardware-level approaches, on the other hand, have been applied to GPUs (e.g., [55]) and could have an impact on many-core systems. Indeed, one of the few prototype many-core platforms with a degree of accessibility (ClearSpeed's CSX700 processor) boasts a number of dedicated hardware error-correction mechanisms, hinting at least to the importance of fault tolerance in this type of devices, whereas no information is available on fault-tolerance mechanisms in the Intel MIC architecture.

2.5. Toward Inherent Software Resilience: ABFT nature of EAs

Evolutionary Algorithms (EAs) are a class of solving techniques based on the Darwinian theory of evolution [19] which involves the search of a *population* of solutions.

A set of recent studies [20, 26, 30, 35, 42, 43] illustrate what seems to be a natural resilience of EAs against a model of destructive failures (crash failures). With a properly designed execution, the system experiences a *graceful degradation* [35]. This means, that up to some threshold and despite the failures, the results are still delivered. However, it either requires more time for the execution or the returned values are further from the optimum being searched.

3. Repeatibility and Reproducibility Challenges in UCSs

Repeatibility and reproducibility are important aspects of sound scientific research in all disciplines – yet more difficult to achieve than might be expected [5, 54]. Repeatability of the experiments denotes the ability to repeat the same experiment and achieve the same result, by the original investigator [56]. On the other hand, reproducibility enables the verification of the validity of the conclusions and claims drawn from scientific experiments by other researchers, independent from the original investigator.

Repeatibility is essential for all evidence-based sciences, as counterpart to formal proofs used in theoretical sciences or discourse used widely in e.g. the humanities. It is a key requirement for all sciences and studies relying on computational processes. The challenge of achieving repeatability, however, increases drastically with the complexity of the underlying computational processes, making the characteristics of the processes less intuitive to grasp and interpret and verify. It thus becomes an enormous challenge in the area of ultrascale computing given the enormous complexity of the massive amount of computing steps involved, and the numerous dependencies of an algorithm performing on a stack of software and hardware of considerable complexity.

While many disciplines have, over sometimes long-time periods, established a set of good practices for repeating and verifying their experiments (e.g. by using experiment logbooks in disciplines such as chemistry or physics, where the researchers record their experiments), computational science lags behind, and many investigations are hard to repeat or reproduce [17]. This can be attributed to the lower maturity of computer science methods and practices in general, the fast-moving pace of changing technology that is utilised to perform the experiments, or the multitude of different software components needed to interact to perform the experiments. Small variations in, for example, the version of a specific software can have a great impact on the final result which might deviate significantly from the expected outcome, as has promoinently been shown e.g. for the analysis of CT scans in the medical domain [27]. More severely, the source of such changes might not even be in the software that is specifically used for a certain task, but somewhere further down the stack of software the application depends on, including for example the operating system, system libraries or the very specific hardware environment being used. Recognizing these needs, steps are being taken to assist research in ensuring their results are more easily reproducible [24, 25]. The significant overhead in providing enough documentation to allow an exact reproduction of the experiment setup further adds to these difficulties.

Technical solutions to increase reproducibility in eScience research cover several branches. One type of solutions is aimed at recreating the technical environments where experiments are executed in. Simple approaches towards this goal include virtualising the complete environment the experiment is conducted in, e.g. by making a clone which can subsequently be redistributed and executed in Virtual Machines. Such approaches only partially allow reproducibility, as the cloned system is potentially containing many more applications than are actually needed, and no identification of which components are actually required is provided. Thus, a more favourable

approach is to recreate only the needed parts of the system. Code, Data, and Environment (CDE) [28] is such an approach, as it detects the required components during the runtime of a process. CDE works on Linux operating system environments and requires the user to prepend his commands to scripts or binaries by the cde command. CDE will then intercept system calls and gather all the files and binaries that were used in the execution. A packaged created thereof can then be transferred to a new environment.

CDE has a few shortcomings especially in distributed system set-ups. External systems may be utilised, e.g. by calling Web Services, taking over part of the computational tasks. CDE does not aim at detecting these calls. This challenge is exacerbated in more complex distributed set-ups such as may be encountered in ultra-scale computational environments.

Not only external, but also calls to local service applications are an issue. These are normally running in the background and started before the program execution, and thus not all the sources that are necessary to run them are detected. It is more problematic, though, that there is no explicit detection of such a service being a background or remote service. Thus, the fact that the capturing of the environment is incomplete remains unnoticed to users who are not familiar with all the details of the implementation. The Process Migration Framework [8] (PMF) is a solution similar to CDE, but takes specifically the distributed aspect into account.

Another approach to enable better repeatibility and reproducibility is in using standardised methods and techniques to author experiments, such as the use of *workflows*. Workflows allow a precise definition of the involved steps, the required environment, and the data flow between components. The modelling of workflows can be seen as an abstraction layer, as they describe the computational ecosystem of the software used during a process. Additionally, they provide an execution environment that integrates the required components to perform a process and execute all defined subtasks. Different scientific workflow management systems exist that allow scientists to combine services and infrastructure for their research. The most prominent examples of such systems are Taverna [41] and Kepler [36]. Vistrails [51] also adds versioning and provenance of the creation of the workflow itself. The Pegasus workflow engine [21] specifically aims at scalability, and allows executing workflows in cluster, grid and cloud infrastructures.

Building on top of workflows, the concept of workflow-centric Research Objects [6] (ROs) tries to describe research workflows in the wider eco-system they are embedded in. ROs are a means to aggregate or bundle resources used in a scientific investigation, such as a workflow, provenance from results of its execution, and other digital resources such as publications or data-sets. In addition, annotations are used to describe these objects further. A digital library exists for Workflows and Research Objects to be shared, such as the platform *my experiment*.[8]

Workflows facilitate many aspects of reproducibility. However, unless experiments are designed from the beginning to be implemented as a workflow, there is a significant overhead to migrate an existing solution to a workflow. Furthermore, workflows are normally limited in features they support, most prominently in the type of programming languages. Thus, not all experiments can be easily implemented in such a manner.

A model to describe a scientific process or experiment is presented in [38]. It allows the researcher to describe their experiments in a manner similar to the approach of Research Objects; however, this model is independent of a specific workflow engine, and provides a more refined set of concepts to specifiy the software and hardware setup utilised.

[8] http://www.myexperiment.org/

Another important aspect of reproducibility is the verification of the results obtained. The verification and validation of experiments aims to prove whether the replicated or repeated experiment has the same characteristics and performs in the same way as the original experiment – even if the original implementation is faulty. Simple approaches just comparing a final experiment outcome, e.g. a performance measure of a machine learning experiment, doesn't provide sufficient evidence on this task, especially in settings where probabilisitic learning is utilised, and also a close approximation of the original result would be accepted. Furthermore, a comparison on final outcomes provides no means to trace where potential deviations originated in the experiment. It is therefore required to analyze the characteristics of an experimental process wrt. to its significant properties, its determinism, and levels where significant states of a process can be compared. On further needs to define a set of measurements to be taken during an experiment, and identify approapriate metrics to compare values obtained from different experiment executions, beyond simple similarity [44]. A framework to formalise such a verification has been introduced as the VFramework [40], which is specifically tailored to process verification. It describes what conditions must be met and what actions need to be taken in order to compare the executions of two processes.

In the context of ultrascale computing, the distributed nature of large experiments poses challenges for repeating the results, as there are many potential sources for errors, and an increased demand for documentation. Also, approaches such as virtualisation or recreation of computing environments hit the boundaries of feasibility especially in larger distributed settings based on grid our cloud infrastructure, where the number of nodes to be stored becomes difficult to manage. Another challenge in ultrascale computing is in the nature of computing hardware utilised, which is often highly specialised towards certain tasks, and much more difficult to be captured and recreated in other settings.

Last, but not least, ultrascale computing is usually also tightly linked with massive volumes of data that need to be kept available and identifiable in sometimes highly dynamic environments. Proper data management and preservation have been prominently called for [29]. A key aspect in ultra-scale computing in this context are means to persistently identify the precise versions and subsets of data having been used in an experiment. The Working Group on Dynamic Data Citation of the Research Data Alliance[9] has been developing recommendations how to achieve such a machine-actionable citation mechanism for dynamic data that is currently being evaluated in a number of pilots [46].

Conclusion

In this paper, we first proposed an overview of resilient computing in Ultrascale Computing Systems, i.e., cross-layered techniques dealing with hardware and software failures or attacks, but also the necessary services including security and repeatability. We also described how new application needs such as big data and cyber-physical systems challenge existing computing paradigms and solutions.

New opportunities have been highlighted but they certainly require further investigations and in particular large-scale experiments and validations. What emerges is the need for the apparition of additional disruptive paradigms and solutions at all levels: from hardware, languages, compilers, operating systems, middleware, services, and application-level solutions. Offering a

[9] https://rd-alliance.org/groups/data-citation-wg.html

global view on the reliability/resilience issues will allow to define the right level of information exchange between all layers and components in order to have global (cross-layer/component) solution. Additional objectives such as performance, energy-efficiency and cost also need to be taken into account. We intend in the context of the COST NESUS project to have several focus groups aimed at defining more precisely the problems and approaches for such challenges.

This work is partially supported by EU under the COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

References

1. M. Abd-El-Barr. *Design and Analysis of Reliable and Fault-tolerant Computer Systems.* Imperial College Press, 2007. DOI: 10.1142/9781860948909.
2. M. Abramovici, C. Strond, C. Hamilton, S. Wijesuriya, and V. Verma. Using roving stars for on-line testing and diagnosis of fpgas in fault-tolerant applications. In *Test Conference, 1999. Proceedings. International*, pages 973–982, 1999. DOI: 10.1109/TEST.1999.805830.
3. J. Andersson, J. Gaisler, and R. Weigand. Next Generation MultiPurpose Microprocessor. In *DASIA 2010 Data Systems In Aerospace*, volume 682 of *ESA Special Publication*, page 7, August 2010.
4. A. Avizienis, J.-C. Laprie, B. Randell, and C.E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004. DOI: 10.1109/tdsc.2004.2.
5. C. Glenn Begley and Lee M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, March 2012. DOI: 10.1038/483531a.
6. Khalid Belhajjame, Oscar Corcho, Daniel Garijo, Jun Zhao, Paolo Missier, David Newman, RaÃžl Palma, Sean Bechhofer, Esteban GarcÃŋa Cuesta, JosÃľ Manuel GÃşmez-PÃľrez, Stian Soiland-Reyes, Lourdes Verdes-Montenegro, David De Roure, and Carole Goble. Workflow-centric research objects: First class citizens in scholarly discourse. In *Proceedings of Workshop on the Semantic Publishing, (SePublica 2012) 9th Extended Semantic Web Conference*, May 28 2012.
7. Luca Benini and Giovanni De Michelli. *Networks on chips : technology and tools.* The Morgan Kaufmann series in systems on silicon. Elsevier Morgan Kaufmann Publishers, Amsterdam, Boston, Paris, 2006.
8. Johannes Binder, Stephan Strodl, and Andreas Rauber. Process migration framework – virtualising and documenting business processes. In *Proceedings of the 18th IEEE International EDOC Conference Workshops and Demonstrations (EDOCW 2014)*, pages 398–401, Ulm, Germany, September 2014. DOI: 10.1109/edocw.2014.66.
9. Cristiana Bolchini, Matteo Carminati, and Antonio Miele. Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing*, 29(2):159–175, 2013. DOI: 10.1007/s10836-013-5367-y.

10. C. Braun and H. Wunderlich. Algorithm-based fault tolerance for many-core architectures. In *Test Symposium (ETS), 2010 15th IEEE European*, pages 253–253, May 2010. DOI: 10.1109/ETSYM.2010.5512738.

11. Franck Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *IJHPCA*, 23(3):212–226, 2009. DOI: 10.1177/1094342009106189.

12. Franck Cappello, Al Geist, Bill Gropp, Laxmikant V. Kalé, Bill Kramer, and Marc Snir. Toward exascale resilience. *IJHPCA*, 23(4):374–388, 2009. DOI: 10.1177/1094342009347767.

13. Sayantan Chakravorty, Celso L. Mendes, and Laxmikant V. KalÃl'. Proactive Fault Tolerance in MPI Applications via task Migration. In *In International Conference on High Performance Computing*, 2006. DOI: 10.1007/11945918_47.

14. Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 213–223, New York, NY, USA, 2005. ACM. DOI: 10.1145/1065944.1065973.

15. V. Cherkassky. A measure of graceful degradation in parallel-computer systems. *Reliability, IEEE Transactions on*, 38(1):76–81, Apr 1989. DOI: 10.1109/24.24577.

16. M. Choi, N.J. Park, K.M. George, B. Jin, N. Park, Y.B. Kim, and F. Lombardi. Fault tolerant memory design for hw/sw co-reliability in massively parallel computing systems. In *Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on*, pages 341–348, April 2003. DOI: 10.1109/NCA.2003.1201173.

17. Christian Collberg, Todd Proebsting, Gina Moraila, Akash Shankaran, Zuoming Shi, and Alex M Warren. Measuring reproducibility in computer systems research. Technical report, University of Arizona, 2014.

18. George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

19. C. Darwin. *The Origin of Species*. John Murray, 1859.

20. Francisco Fernandez De Vega. A Fault Tolerant Optimization Algorithm based on Evolutionary Computation. In *Proceedings of the International Conference on Dependability of Computer Systems (DEPCOS-RELCOMEX'06)*, pages 335–342, Washington, DC, USA, 2006. IEEE Computer Society. DOI: 10.1109/depcos-relcomex.2006.2.

21. Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 2014. DOI: 10.1016/j.future.2014.10.008.

22. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002. DOI: 10.1145/568522.568525.

23. D. Ernst, S. Das, Seokwoo Lee, D. Blaauw, T. Austin, T. Mudge, Nam Sung Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *Micro, IEEE*, 24(6):10–20, Nov 2004. DOI: 10.1109/MM.2004.85.

24. James Taylor eivind Hovig Geir Kjetil Sandve, Anton Nekrutenko. Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10), 10 2013. DOI: doi:10.1371/journal.pcbi.1003285.

25. Ian Gent. The recomputation manifesto, April 12 2013.

26. Daniel Lombrana González, Francisco Fernández de Vega, and Henri Casanova. Characterizing fault tolerance in genetic programming. In *Proc. of the 2009 workshop on Bio-inspired algorithms for distributed systems (BADS'09)*, pages 1–10, New York, NY, USA, 2009. ACM. DOI: 10.1145/1555284.1555286.

27. Ed H. B. M. Gronenschild, Petra Habets, Heidi I. L. Jacobs, Ron Mengelers, Nico Rozendaal, Jim van Os, and Machteld Marcelis. The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS ONE*, 7(6), 06 2012. DOI: 10.1371/journal.pone.0038234.

28. Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th international conference on Large Installation System Administration (LISA'11)*, pages 2–2, Berkeley, CA, USA, 2011.

29. Mark Guttenbrunner and Andreas Rauber. A measurement framework for evaluating emulators for digital preservation. *ACM Transactions on Information Systems (TOIS)*, 30(2), 3 2012. DOI: 10.1145/2180868.2180876.

30. J. Ignacio Hidalgo, Juan Lanchares, Francisco Fernández de Vega, and Daniel Lombrana. Is the island model fault tolerant? In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2737–2744, London, United Kingdom, July 7–11 2007. ACM. DOI: 10.1145/1274000.1274085.

31. Kai Hwang, Jack Dongarra, and Geoffrey C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

32. A. Ivanov and G. De Micheli. Guest editors' introduction: The network-on-chip paradigm in practice and research. *Design Test of Computers, IEEE*, 22(5):399–403, Sept 2005. DOI: 10.1109/MDT.2005.111.

33. C.M. Jeffery and R.J.O. Figueiredo. Towards byzantine fault tolerance in many-core computing platforms. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 256–259, Dec 2007. DOI: 10.1109/PRDC.2007.40.

34. Fernanda Lima Kastensmidt, Luigi Carro, and Ricardo Reis. *Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

35. J.L.J. Laredo, P. Bouvry, D.L. GonzÃąlez, F. FernÃąndez de Vega, M.G. Arenas, J.J. Merelo, and C.M. Fernandes. Designing robust volunteer-based evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(3):221–244, 2014. DOI: 10.1007/s10710-014-9213-5.

36. Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. DOI: 10.1002/cpe.994.

37. D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Toward robust integrated circuits: The embryonics approach. *Proceedings of the IEEE*, 88(4):516–543, April 2000. DOI: 10.1109/5.842998.

38. Rudolf Mayer, Tomasz Miksa, and Andreas Rauber. Ontologies for describing the context of scientific experiment processes. In *Proceedings of the 10th International Conference on e-Science*, Guarujá, SP, Brazil, October 20–24 2014. DOI: 10.1109/eScience.2014.47.

39. P. Mazumder. Design of a fault-tolerant dram with new on-chip ecc. In Israel Koren, editor, *Defect and Fault Tolerance in VLSI Systems*, pages 85–92. Springer US, 1989. DOI: 10.1007/978-1-4615-6799-8_8.

40. Tomasz Miksa, Stefan Proell, Rudolf Mayer, Stephan Strodl, Ricardo Vieira, José Barateiro, and Andreas Rauber. Framework for verification of preserved and redeployed processes. In *Proceedings of the 10th International Conference on Preservation of Digital Objects (iPres 2013)*, Lisbon, Portugal, September 2–6 2013.

41. Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Aleksandra Nenadic, Ian Dunlop, Alan Williams, Thomas Oinn, and Carole Goble. Taverna, reloaded. In M. Gertz, T. Hey, and B. Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, June 2010. DOI: 10.1007/978-3-642-13818-8_33.

42. Elizabeth Montero and María-Cristina Riff. On-the-fly calibrating strategies for evolutionary algorithms. *Information Sciences*, 181(3):552–566, 2011.

43. Alicia Morales-Reyes, Evangelos F. Stefatos, Ahmet T. Erdogan, and Tughrul Arslan. Towards Fault-Tolerant Systems based on Adaptive Cellular Genetic Algorithms. In *Proceedings of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems (AHS'08)*, pages 398–405, Noordwijk, The Netherlands, June 22-25 2008. IEEE Computer Society.

44. Nature. Data's shameful neglect. *Nature*, 461(7261), 9 2009. DOI: 10.1038/461145a.

45. Dongkook Park, C. Nicopoulos, Jongman Kim, N. Vijaykrishnan, and C.R. Das. Exploring fault-tolerant network-on-chip architectures. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 93–104, June 2006. DOI: 10.1109/DSN.2006.35.

46. Stefan Pröll and Andreas Rauber. Scalable Data Citation in Dynamic, Large Databases: Model and Reference Implementation. In *IEEE International Conference on Big Data 2013 (IEEE BigData 2013)*, Santa Clara, CA, USA, October 2013. IEEE. DOI: 10.1109/bigdata.2013.6691588.

47. A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24, June 2014. DOI: 10.1109/ISCA.2014.6853195.

48. Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Comput. Surv.*, 46(1):8:1–8:38, July 2013. DOI: 10.1145/2522968.2522976.

49. Stephen L. Scott, Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, Anand Tikotekar, George Ostrouchov, Chokchai Leangsuksun, Nichamon Naksinehaboon, Raja Nassar, Mihaela Paun, Frank Mueller, Chao Wang, Arun B. Nagarajan, and Jyothish Varma. A Tunable Holistic Resiliency Approach for High-performance Computing Systems. *SIGPLAN Not.*, 44(4):305–306, February 2009. DOI: 10.1145/1594835.1504227.

50. Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems (3rd Ed.): Design and Evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

51. C.T. Silva, J. Freire, and S.P. Callahan. Provenance for visualizations: Reproducibility and beyond. *Computing in Science Engineering*, 9(5):82–89, October 2007. DOI: 10.1109/MCSE.2007.106.

52. Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan DeBardeleben, Pedro C. Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *IJHPCA*, 28(2):129–173, 2014.

53. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2006.

54. The Economist. Trouble at the lab, October 19 2013.

55. S. Tselonis, V. Dimitsas, and D. Gizopoulos. The functional and performance tolerance of gpus to permanent faults in registers. In *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*, pages 236–239, July 2013. DOI: 10.1109/IOLTS.2013.6604089.

56. Jan Vitek and Tomas Kalibera. R3: Repeatability, reproducibility and rigor. *SIGPLAN Not.*, 47(4a):30–36, March 2012. DOI: 10.1145/2442776.2442781.

57. Aida Vosoughi, Kashif Bilal, Samee Ullah Khan, Nasro Min-Allah, Juan Li, Nasir Ghani, Pascal Bouvry, and Sajjad Madani. A multidimensional robust greedy algorithm for resource path finding in large-scale distributed networks. In *Proceedings of the 8th International Conference on Frontiers of Information Technology*, FIT '10, pages 16:1–16:6, New York, NY, USA, 2010. ACM. DOI: http://doi.acm.org/10.1145/1943628.1943644.

58. Chao Wang, F. Mueller, C. Engelmann, and S.L. Scott. Proactive process-level live migration in HPC environments. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, Nov 2008. DOI: 10.1109/SC.2008.5222634.

59. Keun Soo Yim and R.K. Iyer. A codesigned fault tolerance system for heterogeneous many-core processors. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2053–2056, May 2011. DOI: 10.1109/IPDPS.2011.375.

Received March 8, 2015.