On Undecidability Aspects of Resilient Computations and Implications to Exascale

Nageswara S. V. Rao[1]

Computer Science and Mathematics Division,Oak Ridge National Laboratory, Oak Ridge, TN 37831,
`raons@ornl.gov`,
WWW home page: `http://www.csm.ornl.gov/ nrao`

Abstract. Future Exascale computing systems with a large number of processors, memory elements and interconnection links, are expected to experience multiple, complex faults, which affect both applications and operating-runtime systems. A variety of algorithms, frameworks and tools are being proposed to realize and/or verify the resilience properties of computations that guarantee correct results on failure-prone computing systems. We analytically show that certain resilient computation problems in presence of general classes of faults are undecidable, that is, no algorithms exist for solving them. We first show that the membership verification in a generic set of resilient computations is undecidable. We describe classes of faults that can create infinite loops or non-halting computations, whose detection in general is undecidable. We then show certain resilient computation problems to be undecidable by using reductions from the loop detection and halting problems under two formulations, namely, an abstract programming language and Turing machines, respectively. These two reductions highlight different failure effects: the former represents program and data corruption, and the latter illustrates incorrect program execution. These results call for broad-based, well-characterized resilience approaches that complement purely computational solutions using methods such as hardware monitors, co-designs, and system- and application-specific diagnosis codes.

Keywords: Exascale systems, resilient computations, undecidability, uncomputability

1 Introduction

Exascale computing systems are expected to be built using a large number of multi-core processors and accelerators with computing elements totaling a million or more [1, 11]; in addition, they consist of interconnects, switches and hierarchies of memory units, and are supported by specialized software stacks [25]. Typical life-span of the commercial off-the-shelf components used in these systems is about 5-10 years, particularly for processors. Thus, as a rough approximation, computations running for hours may experience multiple failures, and they in turn may result in errors in applications as well as in operating and runtime systems that execute the applications [20]. Furthermore, the sheer size and complexity of these systems may lead to complex faults, not all of which can be known precisely or anticipated accurately. Indeed, they range from manufacturing and device fatigue faults in components, to dynamic hot-spots created in computer racks due to interactions between device placement and cooling systems, to interactions of software modules with degraded hardware components. These faults may manifest in a variety of ways: memory faults may cause the executables to be corrupted and the variables to assume out-of-bound values; circuit faults may cause incorrect loading of the program counters and errors in arithmetic and logic operations; and, bus and interconnect faults may corrupt the data in transit between the processing units.

A broad spectrum of algorithms, frameworks and tools are being actively developed to support resilient computations on failure-prone computing systems. They include hardware monitors, HPL codes [15], application-specific detection methods [16, 17, 5, 9], verification systems [3], Algorithm-Based Fault Tolerance (ABFT) methods [10], resilience ecosystems [18], software-based fault detection [12, 19], and likely invariants for detecting hardware faults [22] (to name a few). It is generally expected that the development and proliferation of such methods will continue as we gain a deeper understanding of the design space of Exascale systems and make progress towards building them.

In this paper, we explore the boundaries of resilient computations that produce correct results on computing systems that are subject to broad classes of failures. We address both the algorithms for resilient computations as well as the provability of assertions about their outputs, particularly, involving arithmetic and logic computations. In a nutshell, we show that the resilient computation problems present significant computational challenges if the underlying failures are not precisely characterized and anticipated. We show a broad class of resilience computation problems to be undecidable in the sense of Turing [23], that is, no algorithms exist for solving them. These results, although based on broad failure models, provide motivation for targeting a smaller and more precisely characterized failure classes that may render these problems decidable.

We first show that verifying if a given program has the property defined by a set of resilient computations is an undecidable problem. We then show that resilient computations under data and program corruption and execution errors are undecidable by using reductions from the classical loop-detection and halting problems. We present the proofs under two formulations, namely, the abstract programming language $\mathcal{L}$ [8] and Turing machines [6], that highlight different aspects of the underlying failures; the former represents program and data corruption, and the latter illustrates incorrect program execution. We outline relativization results that indicate that even if halting problems due to these errors are decidable, it is still possible for undecidable problems to persist. We briefly describe an example failure class based on arithmetic systems that could lead to algorithms for which performance guarantees are hard to prove. The literature in the areas of resilient computations, Exascale systems and undecidable problems is extensive and deep. In this paper, we only refer to a small (perhaps, unevenly represented) set of works that illustrate the main concepts, and recast some of the results from the theory of computation within the context of resilient computations.

These undecidability results indicate that unless the class of faults is limited, these problems cannot be solved by purely computational and analytical means. Hence, they call for broad-based approaches that complement computational solutions, which integrate methods such as hardware monitors, co-design of hardware and software solutions, system-specific diagnosis methods, and application-specific resilience methods. Furthermore, algorithms, frameworks and ecosystems used in such approaches must clearly identify their target failures, and establish that the underlying computational problems are indeed not undecidable. However, severely limiting the class of faults does not necessarily lead to tractable solutions, as indicated by the NP-completeness of stuck-at faults [13] (where the underlying problems are decidable).

We briefly describe undecidable problems and their relationship to resilient computations in Section 2. We describe some examples of failures in computing systems that can could lead to challenges in realizing resilient computations in Section 3. We present undecidability results in Section 4 using language $\mathcal{L}$ in Section 4.1 and Turing machines in Section 4.2. We discuss some implications of these results for Exascale systems and conclusions in Section 5.

2 Context of Undecidable Problems

The notion of undecidability plays two roles in resilient computations, namely, non-existence of algorithms in the framework of Turing [23] and unprovability of assertions about their outputs in the framework of Godel [14] [1]. As pointed out by Chaitin [4], these two results are closely related: informally, they both capture the "finiteness" of algorithms and proofs, which is insufficient to address certain "infinite" requirements of computations and assertions, respectively. The undecidability results are formally proved within the frameworks of recursive functions expressed in $\mathcal{L}$ [8], Turing machines [6], lambda calculus [4], and others [24]. For concreteness, we follow the first two in this paper.

Several of the well-known undecidable problems belong to the decision problems about Turing machines such as the halting problem, empty-set detection and equivalence of Turing machines [6]. These problems might appear somewhat abstract, but there are a number of more "practical" undecidable problems, including virus detection problems [7], programs to test randomness of a string [4], testing the equivalence of context-free grammars, smallest program capable of generating a given string, and computing the Kolmogorov complexity of strings. And, the resilient computations for Exascale systems represent another class of such challenging problems.

Among the existing undecidable problems, closely related to resilient computations are the virus detection problems, wherein the disruptive effects on code

[1] Godel's incompleteness results on provable assertions about arithmetic systems were published in 1931 [14] years before Turing's results on computations in 1936 [23].

executions are to be detected and accounted for. In some sense, the effects of viruses on computer codes are similar to those of complex failures in computing systems. However, the latter are fundamentally different from viruses which are generated by computable or recursive functions. The effects due to failures are not similarly restricted, and as a result the intractability results of virus do not simply carryover to resilient computations. Nevertheless, they both are capable of introducing non-halting computations into otherwise terminating computations, which is one (but not all) of the sources of undecidability in these problems.

3 Complex Faults In Exascale Systems

Complex failures or faults may arise due to a variety of factors in Exascale systems [2, 20]. In addition to individual component faults (due to statistics of larger numbers), multi-component faults can occur as a result of the sheer complexity of Exascale systems, for example, multi-core processor errors due to hot spots in server shelves. We are particularly interested in faults that lead to non-halting computations in codes that are guaranteed to terminate and produce correct results on failure-free computing systems. In particular, infinite loops that lead to non-halting computations can be created by several fault mechanisms including the following:

(a) *Code Corruption:* Program executables may be corrupted and lead to non-terminating loops, for example, the condition $i < N$ may be changed to $i > 0$. Another example would be `go to` statements changed to be self-referential. Third example could be the corruption of base conditions on recursive calls.
(b) *Parameter and Variable Errors:* Infinite loops can be created without modifications to codes by errors in the contents of certain memory locations, for example, loop control variables.
(c) *ALU Circuit Errors:* Failures in Arithmetic and Logic Unit (ALU) circuits can create loops by incorrect execution of terminating conditions of loops and base conditions of recursive calls.
(d) *Program Counter Errors:* Program counters hold the next instructions to be executed, and loading errors in their contents can lead to infinite loops, for example, by repeatedly loading the same instruction.

Within the framework of language $\mathcal{L}$ and Turing machines, both programs and their inputs are treated essentially the same way, namely, as strings. In that sense, there is not much difference between the faults of type (a) and (b), since both can be caused by memory errors; together, they represent program and data corruption, and may be abstracted as string errors in language $\mathcal{L}$. However, failures in (c) and (d) are different in that they occur during the program execution, and may be abstracted as incorrect state transitions of Turing machines. Given the diversity of failure sources of non-halting computations and the complexity of Exascale systems, it not clear if all of them can be adequately known or even if that set is bounded. Furthermore, some of these errors may occur simultaneously, for example, high shelf temperatures might lead to the failures in memory elements and ALU circuits at the same time. Computer viruses modify codes and their executions, and several virus detection problems are known to be undecidable [7]. But, code modifications by viruses do not reflect the entire diversity of faults in Exascale systems due to their possible "non-computational" origins.

4 Resilient Computation Problems

In this section, we present undecidability results within the formal frameworks of abstract programming language $\mathcal{L}$ described in Davis and Weyuker [8] and Turing Machines (TM) [6]. These two frameworks are equivalent for undecidable problems, but the former shows the effects of infinite loops at an algorithm level, whereas the failure effects on memory and circuits are more apparent in the latter. We only present an outline of results from a resilient computations perspective, and details of these formulations and their relationships can be found in [8, 6] (or in introductory books on theoretical computer science).

One of the key notions behind the undecidability is the concept of a universal programming language or equivalently the Universal Turing Machine (UTM) wherein each program can be specified as a string. Both the program P and its input w are specified as strings, and P is "interpreted" in $\mathcal{L}$ and "executed" by a TM with w as the input. That is, these are abstract models of computers wherein the code P is stored as an executable in the memory along with its input w. While these abstract models are much more primitive than complex computing systems, they both are equivalent in terms of the underlying computability as per the Church-Turing thesis [6, 4]. We consider that the programs are coded as integers under a scheme such as Godel numbering in the former [8], and Boolean strings for TMs [6]; in both cases, they can be enumerated like the natural numbers with the caveat that not every natural number represents a valid program P in either scheme.

4.1 Predicates About Programs in $\mathcal{L}$

We follow the programming language abstraction $\mathcal{L}$ described in [8], wherein each program is described as a set of instructions. The programs in $\mathcal{L}$ are converted into numbers using schemes such as the Godel numbers, which are sufficient to describe all computable functions [8] so that a program P is represented by its numerical code y=#(P). Under fault-free conditions, the output of program y with input x is denoted by $\Phi_y(x) = o(x)$, which is also specified by a partially computable function $\Phi(x, y)$ by the Universality Theorem [8]; here, $\Phi(.)$ is universal in that it "executes" any program y with x as its input. Under fault-free conditions, both y are x specified initially, and neither is altered at any point during the execution of y. Under faulty conditions, however, both can be altered any time during the execution in two ways: contents of y and x can be altered through data corruption, and instructions of y can be erroneously executed due to hardware faults. Then, under faulty conditions, the output of y with input x is denoted by $\Phi_y^F(x) = g(x)$ such that $o(x_0) \neq g(x_0)$ for some x_0.

Undecidability of Membership in Resilient Computation Classes Let y_R denote a *resilient version* of the original program y such that under faulty conditions it produces the same output as y under fault-free conditions. For a given input x, let $g(x)$ and $g_R(x) = o(x)$ denote the outputs of functions computed by programs y and y_R under faulty conditions, respectively. Thus, y_R accounts for faults in y and x, and yet produces the correct output. For example, consider a hypothetical single-fault case of y producing a Boolean output which is complemented; then, y_R can simply execute y and complement its output to produce the correct answer.

We now show that there is no algorithm for verifying if a given program y possesses the resilience property implemented by a set of resilient codes of type y_R, by showing the underlying decision problem to be undecidable. Let $\mathcal{A}$ be a set of original programs that halt under fault-free conditions, but run erroneously under faulty conditions, namely, they either do not halt or produce incorrect or undefined output; $\mathcal{A}$ is assumed to be *non-trivial* [8] which implies it is non-empty and does not span all codes. Then, the corresponding non-trivial set of all $\mathcal{A}$-resilient programs is denoted by $\mathcal{R} = \{y_R | y \in \mathcal{A}\}$, which is assumed to exist and may be custom designed to overcome the faults that are specific to programs in $\mathcal{A}$. For example, $\mathcal{R}$ could consist of all resilient versions of a set of non-linear solvers $\mathcal{A}$ that are designed to correct for ALU faults. We assume that errors are such that $y \in \mathcal{A}$ by itself is not a resilient version of any other original program $y_1 \in \mathcal{A}$. Then, the index set of these $\mathcal{A}$-resilient programs $\mathcal{R}$ is

$$R_{\mathcal{R}} = \{t \in N | \Phi_t^F \in \mathcal{R}\},$$

which we show to be a non-computable set.

Theorem 1. *The index set of $\mathcal{A}$-resilient programs $R_{\mathcal{R}}$ is not computable, that is, the problem of verifying if a given program belongs to $\mathcal{R}$ under faulty conditions is undecidable for any non-trivial set of original programs $\mathcal{A}$.*

Proof: The proof is through contradiction. Consider that $R_{\mathcal{R}}$ is computable, and consider $g_R \in \mathcal{R}$ is a resilient version of the original (non-resilient) $g \in \mathcal{A}$. Let us define a function

$$h(t,x) = \begin{cases} g(x) & \text{if } t \in R_{\mathcal{R}} \\ g_R(x) & \text{if } t \notin R_{\mathcal{R}}. \end{cases}$$

Then, the following function is partially computable [8]

$$h(t,x) = 1_{R_{\mathcal{R}}}(t).g(x) + [1 - 1_{R_{\mathcal{R}}}(t)].g_R(x),$$

under faulty conditions, where $1_S(.)$ is the indicator function for set S: $1_S(t) = 1$ if and only if $t \in S$ and $1_S(t) = 0$ otherwise. In particular, a program to compute $h(t,x)$ is composed by using that for $1_{R_{\mathcal{R}}}(.)$ as a sub-routine. Then, by the Recursion Theorem, there is a program e such that

$$\Phi_e^F(x) = h(e,x) = \begin{cases} g(x) & \text{if } \Phi_e^F \in \mathcal{R} \\ g_R(x) = o(x) & \text{if } \Phi_e^F \notin \mathcal{R}, \end{cases}$$

where $\Phi_e^F(x) = \Phi^F(x, e)$ for a universal partially computable function Φ^F specified by the Universality Theorem applied under faulty conditions. Now consider that $e \in R_{\mathcal{R}}$ that is, it is $\mathcal{A}$-resilient and can be executed under faulty conditions such that $\Phi_e^F(x) = o(x)$; but, by definition of $h(e, x)$, we have $\Phi_e^F(x) = g(x)$ and in particular $\Phi_e^F(x_0) = g(x_0) \neq o(x_0) = \Phi_e^F(x_0)$. On the other hand, consider that $e \notin R_{\mathcal{R}}$, that is, it is not $\mathcal{A}$-resilient, namely $\Phi_e^F(x_0) \neq o(x_0)$; then, by above definition of $h(e, x)$, we have $\Phi_e^F(x) = g_R(x) = f(x)$, and in particular, $\Phi_e^F(x_0) = o(x_0) \neq \Phi_e^F(x_0)$. Thus, in both cases we have a contradiction, which proves the theorem. □

This result is a particular application of the well-known Rice's Theorem to resilience computations. Informally, it implies that it is not possible to verify if a given program possesses the resilience property embodied by $R_{\mathcal{R}}$. While establishing the undecidability with respect to classes of resilient computations, this result does not pinpoint the sources of undecidability in individual programs. We next show that the dynamic loops created by the faults are sufficient to lead to undecidability of verifying the resilience of individual codes.

Infinite Loops The halting problem is specified by the predicate HALT(x, y) which is true if and only if the program y with input x halts. This problem is *undecidable* in that there does not exit a program written in $\mathcal{L}$ that can decide if this proposition is true or false (Theorem 2.1, Chapter 4, [8]). We consider a class of failures that can be captured by the *failure function*, $f(x, y) = (x_f, y_f)$ that replaces x by x_f and y by y_f just before the execution of y is initiated, and no other failures occur. We note that this characterization is limited to deterministic failures since such functions are not sufficient to characterize random errors, which indeed can occur in Exascale systems. We consider that the execution of y with input x produces yes/no answer in a finite time on a failure-free machine.

Consider the predicate RESILIFY(x, y, f) which is true if and only if there exists a program P_f that "executes" y_f with input x_f and produces the output identical to that produced by y with input x. Now, we further restrict $f(.)$ to functions $\mathcal{F}_L$ that create infinite loops due to *data and program corruption* that modify one or both y and x, for example, by using (but not limited to) mechanisms listed in Section 3.

We next show that RESILIFY(x, y, f), $f \in \mathcal{F}_L$ is not a computable predicate, by reducing the following simpler problem to it. Let NO-LOOP(x, y) denote the predicate that the program y with input x does not loop forever, that is, it will produce yes/no output in a finite number of steps.

Theorem 2. *RESILIFY(x, y, f), $f \in \mathcal{F}_L$ is not a computable predicate under program and data corruption.*

Proof: We prove this theorem in three steps. First, RESILYFY$(x_f$,$y_f, I)$ true if and only if RESILYFY(x, y, f) true, where f changes y to y_f, for $y_f \in \mathcal{F}_L$, x to x_f, and I is the identity function. Next, we note that NO-LOOP(x_f, y_f) is true if and only if RESILIFY(x_f, y_f, I) is true, thereby showing that the undecidability of the former implies that of the latter by restriction. We now show that NO-LOOP(x, y) is an undecidable predicate by contradiction using the well-known proof method used in [8]; we present the details here for completeness. Let us assume NO-LOOP(x, y) is computable, and hence can be inserted into the following program P:

```
[P]: if NO-LOOP$(x, x)$ go to P
         else return NO
```

This program takes a single input x and uses NO-LOOP(x, x) as a subroutine. Based on the above code, P keeps looping while NO-LOOP(x, x) is true, that is it halts if and if only if the predicate NO-LOOP(x, x) is not true, that is ∼NO-LOOP(x, x). Now let $y_0 = \#(P)$ denote the code of the above program P expressed under Godel's numbering. By using $y = y_0$ in the definition we have: NO-LOOP(x, y_0) if and only if $y_0 = \#(P)$ does not loop forever, that is ∼NO-LOOP(x, x) is true by the definition of P. Since the above statement is valid for any value of x, we choose $x = y_0$ which leads to the contradiction: NO-LOOP(y_0, y_0) if and only if ∼NO-LOOP(y_0, y_0). □

Informally, this theorem shows that there is no algorithm to determine if the failures cause the code execution to be stuck in an infinite loop. But, for very restricted cases in which infinite loops can be created by known mechanisms, this problem could be decidable. More generally, there might be other complex failures, such as purely random errors or introduction of non-compressible strings [4], that could potentially lead to undecidable resilient computation problems. Also, under certain strictly component-level failures, one can develop targeted component diagnosis codes [21] that can verify that no stuck-at failures have occurred during the code execution.

4.2 Turing Machines for Resilient Computations

We now repeat the result of Theorem 2 within the formulation of UTM that "executes" a given TM M on input w. This formulation abstracts a general purpose computer by UTM that executes a program specified by M using w as its input. A Turing machine M is composed of a tape of cells that holds w, and is also used for holding intermediate results; the operation of M is specified by a finite set of transitions such that in each the tape head reads a cell and moves left or right by possibly writing a symbol in the cell. Details of Turing machines can be found in introductory computing theory books, and we use the specific formulation from [6] and details of the dynamic states from [7]. In this formulation, TM M plays the role of program y in the previous section, and UTM plays the role of Φ in "executing" M by emulating its transitions.

While the overall undecidabilty result is same as in the previous section, this formulation is based on using reduction from the haltinng problem and illustrates additional aspects of complex errors. The TM M and its input w are both represented as strings on the tape, and hence they can be corrupted by "memory" faults either at the start of computation (as in the previous section) or dynamically at any time during UTM operation that executes M (similar to the case of a virus [7]). Another source of faults is in the execution of transitions of M by UTM, wherein the contents on the tape, namely the instructions, may be read incorrectly, or may be written incorrectly onto the tape; these abstract data transfer errors, for example, between memory and ALU. Also, the transition operations of UTM may be incorrect, for example, tape head being stuck or moving to an incorrect tape cell, and these faults abstract errors in CPU control units. Thus in this model, the failures in string M are analogous to corruption in program codes, and failures in w or in other tape cells are analogous to data corruption errors. Also, UTM transition and state errors are analogous to the errors in ALU and control units.

A *Resilient TM* M_R takes as input TM M and its w under the failure function f, and halts and produces the same output as M with input w under no failures. As in the previous section the fault function f, changes M_R's input to M_f and w_f dynamically as the input is being read as a result of executional and data corruption.

Theorem 3. *The resilient Turing machine M_R that produces under the failure function f the same output as M with input w under no failures, does not exist under executional errors and data corruption.*

Proof: Consider that M_R exits, which requires that there is a program that detects when M_f does not halt, and intervenes and reproduces the output of M, that is, it solves the halting problem of M_f. We now present the undecidability proof of this halting problem for completeness, and also illustrate some details specific to resilient computations. Let M_R be a TM that produces output yes if M_f produces yes with input M_f, and produces no otherwise. Then, we construct another Turing machine $\bar{M}_R$ that simply flips the output of M_R, namely, it outputs yes if M_f outputs no or does not halt with input M_f, and outputs no if M_f outputs yes with input M_f. Now consider the behavior of $\bar{M}_R$ with input $\bar{M}_R$. If $\bar{M}_R$ outputs yes means $\bar{M}_R$ outputs no with input $\bar{M}_R$, which is a contradiction. On the other hand, if $\bar{M}_R$ outputs yes it means that $\bar{M}_R$ outputs no or does not halt on input $\bar{M}_R$, which is again a contradiction. Hence M_R does not exist. $\square$

Notice that in the above proof, to establish the non-existence of M_R it is not necessary to require that it produces the same result as M; instead, it is sufficient to require that it detects non-halting execution of M_f. We note that non-halting computations may be created during the execution of M_f due to errors in UTM transitions, even if M_f does not initially contain infinite loops; this scenario is analogous to the errors due to certain viruses [7], and typically these undecidability proofs require more detailed dynamic versions of TMs. Furthermore, it is possible that they are other complex failures that could lead to undecidable resilient computation problems beyond the halting problem as described next.

Beyond Halting Problems In the relativization framework, a computing task G is abstracted to be carried out by an oracle to gain insights into the residual underlying complexity. In particular, by using H-oracle that solves the halting problem, a hierarchy of problems is shown to exist, each of which "more unsolvable" than the preceding one (through the so-called the jump process of G, Theorem 4.10 [8]). Let $\mathcal{A}_{\mathcal{F}}$ denote the original programs that are not resilient under a class of faults $\mathcal{F}$, and let $\mathcal{R}_{\mathcal{A}_{\mathcal{F}}}$ denote their resilient versions. Let $\mathcal{B}_{\mathcal{F}}$ denote the original programs that remain non-resilient under H-oracle relativization, and let $\mathcal{R}_{\mathcal{B}_{\mathcal{F}}}$ denote their resilient versions; it is, however, an open question if faults exit that lead to such non-resilient programs. A set of programs $\mathcal{A}$ is called *non-trivial*, if it is non-empty and there is at least one program that is not contained in it. Based on a relativised version of the Rice theorem (Theorem 8.1, Chapter 16, [8]), the halting problem is one-one reducible to that of checking the membership in a non-trivial class of programs $\mathcal{A}$; this result means that the halting problem is no harder than checking the membership in $\mathcal{A}$. Within the context of resilient computations, if there are programs that belongs to $\mathcal{B}_{\mathcal{F}}$ and resilient versions of them exist, then $\mathcal{R}_{\mathcal{B}_F}$ is a non-trivial class. Then by Theorem 1, the problem of checking the resilience property of programs in $\mathcal{R}_{\mathcal{B}_{\mathcal{F}}}$ is undecidable under H-oracle, and is harder than the halting problem. Considering the broad spectrum of potential faults in Exascale systems, $\mathcal{B}_{\mathcal{F}}$ would be non-empty unless all possible faults are shown to lead to problems no harder than the halting problem; without such proof, undecidable problems persist beyond the halting problem in resilient computations.

4.3 Assertions on Error Corrections

Several computations on Exascale systems involve arithmetic operations, and it would be of interest to prove certain assertions about their outputs when executed on a failure-prone system, such as, a statement that errors will always be corrected. We now show a formulation wherein such assertions may turn out to be very difficult to prove or disprove. Consider a program to compute an integer function $G(x, a) = x^a$ on a failure-prone system wherein x is corrupted at the beginning to a smaller value x_f. Consider a class of error correcting algorithms that only compute integer functions of the form $G(y, a) = y^a$, $y < x$ and make an additive correction to make up the difference such that the correct answer $G(x, a) = G(x_f, a) + G(y, a)$ is produced. However, such guarantee for arbitrary values of $a > 2$ contradicts the Fermat's last theorem that states that the correction term does not exist; furthermore, the proof of this theorem itself remained open for more than 300 years (until resolved by Andrew Wiles in 1995). In particular, assertions that broad classes of errors will be corrected by a proposed method should rule out their dependence on assertions such as the existence of resilient TMs described in Theorem 3.

5 Conclusions

We addressed certain limits on algorithmic solutions to resilient computation problems under a broad class of failures in large computing systems, and showed that no general algorithms exist to achieve resilient computations if the classes of faults are unrestricted. However, effective solutions may be found for certain smaller classes of errors, provided it is established that those are a complete set of faults for the given system. In another direction, the algorithmic solutions may be combined with other co-design methods to overcome the limitations of these purely algorithmic methods. For example, individual components may be monitored using hardware monitors to ensure their proper operation during the execution of codes. Also, hardware replication methods may be used to mask component errors. Software replication and checkpoint methods may be utilized to correct certain faults. Moreover, such methods may be combined to generate ecosystems [18] to support resilient computations using both hardware and software methods. However, it is very critical that such solutions clearly specify their target class of faults. When faults are limited to individual components and are non-sporadic, targeted fault detection algorithms may be designed and executed along with the codes. And, if no faults are detected, confidence measures may be assigned to indicate fault-free execution of codes. However, even under a simple failure model of circuit-level faults, the underlying computational problems, while decidable, can be computationally intractable [13].

This work explores only a very small fraction of the complex problem space of computations that produce correct results on failure-prone computing systems, in particular Exascale systems with complex failures. However, these general undecidability results motivate a deeper study and understanding of various types of faults that can occur in Exascale systems and their taxonomy so that solutions may be appropriately targeted. It would be of future interest to investigate similar computational limits of probabilistic computations that guarantee correct results with a specified probability, or deterministic computations that provide confidence measures for computations under probabilistic faults.

Acknowledgments

This work is funded by the Mathematics of Complex, Distributed, Interconnected Systems Program, Office of Advanced Computing Research, U.S. Department of Energy at Oak Ridge National Laboratory managed by UT-Battelle, LLC for U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. F. Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *Journal of High Performance Computing Applications*, 23(3):212–226, 2009.
2. F. Cappello, A. Geist, B. Gropp, S. Kale, B. Kramer, and M. Snir. Towards exascale resilience. *Journal of High Performance Computing Applications*, 23(4):374–388, 2011.
3. M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitaive realiability for programs that execute on unreliable hardware. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2013.
4. G. J. Chaitin. *Information, Randomness and Incompleteness.* World Scientific Pub, 1990. Second Edition.
5. Z. Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2013.

6. D. I. A. Cohen. *Inroduction to Computer Theory.* John Wiley and Sons, Inc., 1986.
7. F. B. Cohen. Computational aspects of computer virus. *Computer & Security*, 8:325–344, 1989.
8. M. D. Davies and E. J. Weyuker. *Computability, Complexity, and Languages.* Academic Press, Inc, 1983.
9. T. Davies and X. Chen. Correcting soft errors online in lu factorization. In *Symposium on High-Performance Parallel and Distributed Computing*, 2013.
10. M. de Kruijif, S. Nomura, and K. Sankaralingam. Relax: An architectural framework for software recovery of hardware faults. In *International Symposium on Computer Architecture (ISCA)*, 2010.
11. J. Dongarra, P Beckman, and et al. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.
12. M. Erez, N. Jayasena, T. J. Knight, and W. J. Dally. Fault tolerance techniques for the merrimac streaming supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2005.
13. H. Fujiwara and S. Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Trans. on Computers*, C-31(6):553–560, 1982.
14. K. Godel. On formally undecidable propositions of principia mathematica and related systems i. *Monatshefte fur Math. und Physik*, 38:173–198, 1931. Englishe translation by B. Meltzer, published by Dover Publications, Inc, 1992.
15. HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. http://www.netlib.org/benchmark/hpl.
16. Y. Huang and C. Kintala. Software fault tolerance of the application layer. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 231–248. 1995.
17. Y. Jia, P. Luszczek, G. Bosilca, and J. Dongarra. Cpu-gpu hybrid bidiagonal reduction with soft error resilience. In *Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA).* 2013.
18. D. Li, Z. Chen, P. Wu, and J. S. Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis.* 2013.
19. M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 2008.
20. C. Lu and D. A. Reed. Assessing fault sensitivity in mpi applications. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.
21. N. S. V. Rao. Fault detection in multi-core processors using chaotic maps. In *3rd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2013)*, 2013.
22. S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *International Conf. on Dependable Systems and Networks*, 2008.
23. A. N. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Society*, 42(3,4):230–265, 1936.
24. V. A. Uspensky. *Godel's Incompleteness Theorem.* Mir Publsihers, 1987. English translation.
25. J. S. Vetter, editor. *Contemporary High Performance Computing: From Petascale toward Exascale.* Chapman and Hall, 2013.