PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

rsta.royalsocietypublishing.org

Discussion

Cite this article: Palem KV. 2014 Inexactness and a future of computing. *Phil. Trans. R. Soc. A* **372**: 20130281.
http://dx.doi.org/10.1098/rsta.2013.0281

One contribution of 14 to a Theme Issue 'Stochastic modelling and energy-efficient computing for weather and climate prediction'.

Subject Areas:
pattern recognition, hybrid computing, software, theory of computing, computer-aided design, systems theory

Keywords:
approximation algorithms, heuristics, inexact computing, information theory, probabilistic CMOS, randomized algorithms

Authors for correspondence:
Krishna V. Palem
e-mail: palem@rice.edu

Inexactness and a future of computing

Krishna V. Palem

Department of Computer Science and ECE, Rice University, 6100 Main Street, Houston, TX 77005, USA

As pressures, notably from energy consumption, start impeding the growth and scale of computing systems, inevitably, designers and users are increasingly considering the prospect of trading accuracy or *exactness*. This paper is a perspective on the progress in embracing this somewhat unusual philosophy of innovating computing systems that are designed to be *inexact* or *approximate*, in the interests of realizing extreme efficiencies. With our own experience in designing inexact physical systems including hardware as a backdrop, we speculate on the rich potential for considering inexactness as a broad emerging theme if not an entire domain for investigation for exciting research and innovation. If this emerging trend to pursuing inexactness persists and grows, then we anticipate an increasing need to consider system co-design where application domain characteristics and technology features interplay in an active manner. A noteworthy early example of this approach is our own excursion into tailoring and hence co-designing floating point arithmetic units guided by the needs of stochastic climate models. This approach requires a unified effort between software and hardware designers that does away with the normal clean abstraction layers between the two.

1. Introduction

Computing and determinism have been intimately intertwined starting with Turing's seminal work [1] which provided the universal model for computing. In the modern world, Turing machines and similar models of computing are implemented in practice using a mix of hardware and software components. It is instructive to try and understand the degree to which determinism influenced this evolution, if not revolution, leading to modern computing from the days of Turing's

Royal Society Publishing

original insights. Hardware provided the fabric or the substrate on which to compute, whereas software provided the flexibility of Turing's universal abstractions by providing mechanisms for executing programs that realized a range of applications. Viewed from this elevation, determinism implicit in Turing's model—also in al-Khwārzmī's concept of an *algorism*—permeated the needs of all layers of the physical system, from the algorithm, through the software used to realize it, to the hardware that ultimately executes and computes!

To see this, one need consider only the pioneering ENIAC [2] where determinism was essential. In particular, determinism manifested itself in the form of correct execution. Thus, correctness is an essential part of what computers have come to represent in the popular lexicon. Even in the ENIAC and Eckert notes 'we had a tube fail about every two days, and we could locate the problem within 15 min' [3]. Thus, working with a system having elements that have failed was not acceptable even in the beginning. Everything had to function, and function correctly! Correctness manifested in many forms ranging from arithmetic units to those that controlled the flow of a program's execution, and last, but not least, storage or memory. Even in the days of the pioneers, this need for correctness caught the attention of one of them, John von Neumann. In his lectures delivered at the California institute of technology in 1952 entitled 'Probabilistic logics and the synthesis of reliable organisms from unreliable components' [4], he envisioned the need for reliability, while focusing on ways of ensuring that unreliable systems could be corrected to realize systems that computed accurately. In fact, testing to make sure that the system is performing what it is meant to perform consistent with its specification is a not insignificant portion of the computing industry today!

So, what is the connection between determinism and correctness? In computing, does determinism always imply correctness? Strictly speaking, determinism does not imply correctness in the sense that the field of computing uses this term. For example, the Oxford English Dictionary says determinism is 'the doctrine that all events, including human actions, are ultimately determined by causes regarded as external to the will'. In Turing's model of computing, the causal dependence on entities 'external to the will' takes the form of input symbols where the analogue of human action is the transition function. We believe the formulation of the 'human action' as a mathematical function by Turing led to the concept of determinism and correctness to become intertwined in an interesting way, in particular when we consider *implementations* or physical realizations of computers.

2. Information efficiency and determinism

Let us digress and dwell on why Turing's conceptions, often referred to as universal deterministic Turing machines (which we refer to as DTMs for convenience), came to be intertwined with correctness and reliability in implementations. To understand this better, let us consider one of the key elements in a Turing machine—its *transition* function. Transition functions are mathematical functions in the classical sense which are invariably implemented correctly. Thus, a transition function δ would, in keeping with the canonical definition of a function, always yield the same value as its *output* given the same *input* faithful to the *specification* of the transition function. Even though modern-day computing systems are not built out of elementary transition functions, nevertheless, we believe the legacy from this concept is strong. For example, at the hardware level, an adder or a multiplier is expected to produce *exactly* the same answer or output given the same pair of inputs no matter how many times this step is repeated. Furthermore, the output value so computed ought to be faithful to a correct specification. For example, $8+6$ is always computed to be 14 and 8×5 is always 40. Similarly, a branch unit implementing a conditional instruction such as *if* $X>5$, *then do A; else do B* is expected to always perform task A whenever the condition $X>5$ is encountered, and task B otherwise. The same goes with storage or memory.

Consequently, we would not accept a computer system that yields the value 11 when performing addition with inputs $8+6$ deterministically, or if it resulted in an answer of 16

on occasions *probabilistically* with the same inputs! Trivial that this observation might seem, it underpins a significant fact about our perception and the practice of what a computer or a computing system is. At a more general level and returning to Turing machines, given a problem Π and a specific input I from a possible set **I**, the output O is a precise entity from a possible set **O**, and no matter how many times a system that emulates a DTM encounters I, the same output O must be produced upon successful termination of the computation. In this sense, given Π, a DTM provides a *contract* or mapping between inputs from the set **I** to elements of the set **O**. Let us consider this contract to be a specification providing the basis for defining *exact* or *precise* computing. Hence, *it is crucial that the underlying implementation as a computing system must preserve this contract or else we would not consider it to be a valid computer!* Thus, exactness in our concept has to do with the faithful physical realization of an algorithm specification, deterministically. In this sense, exactness is not an absolute but is rather a relative concept.

More generally, we can consider the output to be within an ϵ sphere, or some previously 'contracted' or agreed to region around the point O which is usually determined by the needs of the application. This is typical of the domain of numerical analysis involving computations defined on the field of reals. Invariably, in an implementation in hardware and software, *floating point* number representations and arithmetic units are used to achieve this. These representations allowed the algorithm and application developers to have variable precision in the application. Superficially, this might seem to be violating the need for exactness, because the hardware precision used to compute might, in fact, be lower than that specified in the original algorithm; typically lowered to achieve speed or performance benefits. However, a lot of effort is invested to preserve the application's contract with the output set **O** being computed ([5] and others). Therefore, a specification might, in fact, be locally perturbed in this sense of altering the precision, because presumably the application had slack in its needs, and the computing system was being needlessly over-engineered to compute at a higher precision than was strictly needed. Thus, the original implementation was *information inefficient*. By lowering the precision but staying with the ϵ sphere, the resulting implementation is much more information *efficient*, but is still exact, because the original degree of accuracy is preserved. Thus, the degree to which precision is lowered is still adequate to preserve exactness. In fact, ensuring that numerical computations converge and remain exact while lowering precision is a substantial field of study and innovation.

3. From information efficiency to inexactness

Let us now consider an illustrative example to understand the concept of inexactness better through the celebrated *travelling salesperson problem* (TSP). As the name suggests, the problem involves a travelling salesperson who starts at his or her home in a certain city, say Houston, and has to travel across the country through some number n of additional cities and return to Houston. Being efficient, the salesperson must visit each of the n cities only once—namely not return to any city as part of the tour—and find the shortest tour in terms of the number of miles travelled, through the n cities. Given a fixed map with the distance between each pair of the $(n+1)$ cities and enough time to compute the shortest tour—which could be impractically large—any exact algorithm and a computing system to implement it will always find the (same) shortest tour for the salesman. Thus, in this instance, the problem Π is the TSP, the possible maps of cities with distances between constitute the input set **I**, and for each map the possible tours the salesman could take, from which a shortest tour could be determined, constitutes the output **O**.

In this example, but one that can be made completely formal, an exact computing system is one that honours the contract specified by a specification of the problem being solved through some mix of hardware and software artefacts, between any input and its set of outputs for the TSP Π. Starting at the base of the hierarchy of artefacts from which an exact computing system is built, hardware logic gates which are the elementary or atomic building blocks of modern-day computing systems are expected to be exact. Exactness is preserved as we move up

through the hardware and software layers, including the operating system, ultimately yielding exact computing systems overall. We note in passing that while it is possible to develop a completely formal and elegant framework for designing exact systems through compositional approaches starting with atomic artefacts [6], it is too detailed for this expository article. So, for present purposes, we characterize exactness in this informal manner we have been adopting so far through a specification and the language of a contract between inputs and outputs, without explicitly and formally addressing the compositional aspects of how the system might be realized from simpler building blocks.

If this is an exact computing system, then what would an *inexact* computing system do? To start with, for our present discussion, it still has the three elements Π, I, O (or more generally, an ϵ sphere around each output $O \in \mathbf{O}$). Now, in the inexact variety, the contract changes. Specifically, given an input I, an inexact solution to this problem need not produce the associated output O expected of the contract at the level of the algorithm but rather can produce some (other) output O' or equivalently a point in a sphere of radius $\epsilon' > \epsilon$. Intuitively, one would, of course, like to consider O' to be meaningfully related or have affinity to O in some manners. Computer science provides us with a few alternatives to capturing such affinity. In our TSP example, if O is the shortest tour, then O' might be a tour that is guaranteed to be no more than twice as long as the shortest tour. In this setting, given an input instance I, the output O', while not being the shortest tour, is always determined to be the same. In computer science parlance, this corresponds to the notion of *approximation algorithms* [7]. A simple and yet interesting example involves finding an inexact solution to the TSP whenever the edge weights in the graph obey the triangle inequality, and a classical approximation algorithm provides a guarantee that the tour O' is within a factor of 1.5 of the shortest tour [8].

Another approach to inexactness could be that each time a problem instance Π is solved by an execution of our inexact system where a different tour O' is produced through some form of *randomization* [9,10]. Thus, given a sequence of such executions, the outputs $O', O'', O''' \ldots$ might all be different. Again, it is desirable that each of the outputs $O', O'', O''' \ldots$ has some affinity to O. Here, the pioneering work by Rabin [11] and Solovay & Strassen [12] led to the celebrated test for *primality* which is inexact in our sense; the guarantee was on the probability that the output is correct. Also relying on probabilistic concepts, Karp [13] provided a framework for considering inexact solutions by viewing the parameters of the inputs from $\mathbf{I}$ to be drawn from a probability distribution, and the contract with the output is a probabilistic guarantee using the structure that the distribution yields. As one example, Karp showed that under probabilistic preconditions on the input, a variation of the TSP can be solved such that the difference between the shortest tour and one that his proposed algorithm finds has the property that the ratio between the length of the tour found and shortest tour tends to zero. Karp's algorithms are deterministic and the approach came to be broadly referred to as *average case* analysis. These approaches are an intrinsic part of how inexactness was practised through algorithmic variations for several decades now. *To summarize, these approaches to inexactness provide a relaxation of the contract between the inputs and outputs in different ways. A crucial point to note is that this notion of inexactness is based on having a basis to first characterize exactness and then view inexactness as a relaxation of this property. Thus, inexactness in our sense does not have any validity unless an exact formulation is first specified. Additionally, some relationship between the quality of the inexact output and that of its exact variant ought to be quantified as an essential element.*

4. The contract through the lens of computing

The prototypical system for implementation shown in figure 1 consists of multiple layers of abstraction and is quite complex. As a result, exactness at every level and at every granularity is critical if the overall system has to deliver exact behaviour. To revisit the roots of this almost axiomatic need, first, as computer science blossomed, the legacy of exactness we outlined in the early part of this paper implied an overwhelming emphasis on deterministically exact computing as the vehicle for delivering applications. Second, in such a system (figure 1), a single hardware

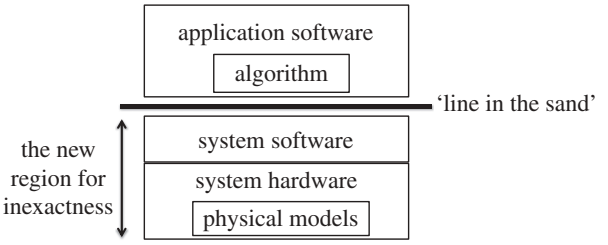

Figure 1. Breaking the system stack into its four generic constituent layers.

architecture and system layer supported the entire family of applications to run on the system, implying a single fixed contract. As a result, an engineer working at the hardware or software layers would have found it rather difficult if not entirely impossible in the past to have multiple contracts, some exact and others inexact, from a single system module that is being designed, negotiated with different applications or even classes of applications.

For example, at the hardware layer, there is an expectation that the underlying gate and circuit layers deliver 'behaviours' that are consistent with the architecture [14], which is the abstract interface between the hardware layer and the software running on it. To use our earlier example, the circuits supporting an integer adder, multiplier or hardware unit for dividing numbers is expected to compute the same answer *always* consistent with the specification of a correct computation. In addition, contracts at a certain level are derived from those at lower levels inductively, and, consequently, any contract for exactness really crossed the various hardware and software system levels of abstraction. This cross-layer contract for exactness is more nuanced if we consider floating point numbers, which are meant to be imprecise representations of real numbers to start with. Here, the concept of how well a floating point number represents the original real number that it is supposed to represent is by itself a field of study [15], but, nevertheless, once a representation such as the IEEE standard [16] is chosen, the adder that the architecture supports must abide by this contract. In this instance, the precise behaviour dictated by the IEEE standard becomes the exact representation. Precision is an additional parameter which can be added and one can consider an exact adder with 32 bits of precision or one with 64 bits. Now, exactness at the level of the application variable precision using such hardware can be then preserved in the most information efficient means possible. Thus, an exact adder of a certain precision can be used to realize an exact or an inexact implementation of the application.

5. From algorithms to physics and hardware

While great strides were being made in introducing inexactness at the algorithmic level, going back to the time of the pioneers, preserving reliability, and hence accuracy or precision of the hardware or physical layer remained sacrosanct in digital computing. To quote one of them, von Neumann [4], 'In all the previous considerations, it has been assumed that the basic components were faultless in their performance. This assumption is clearly not a very realistic one. Mechanical devices as well as electrical ones are statistically subject to failure…'. Breaking with this rich tradition, in 2001, we started investigating the possibility of computing systems that could be inexact at the physical level.

Our motivations were different from the classical approaches and developments at the algorithmic level. In particular, we were concerned with *energy* (or power being the popular metric) efficiency rather than running time or space. In order to include considerations of energy, the abstract algorithmic models such as Turing machines or their equivalent *random access machines* [17] had to be necessarily linked to elements of physics. In contrast, traditional Turing machine-inspired models abstract away the underlying physical connection. This need was recognized by von Neumann in [4], where he notes that 'The subject matter, as the title suggests is error in logics, or in the physical implementation of logics—in automata synthesis.

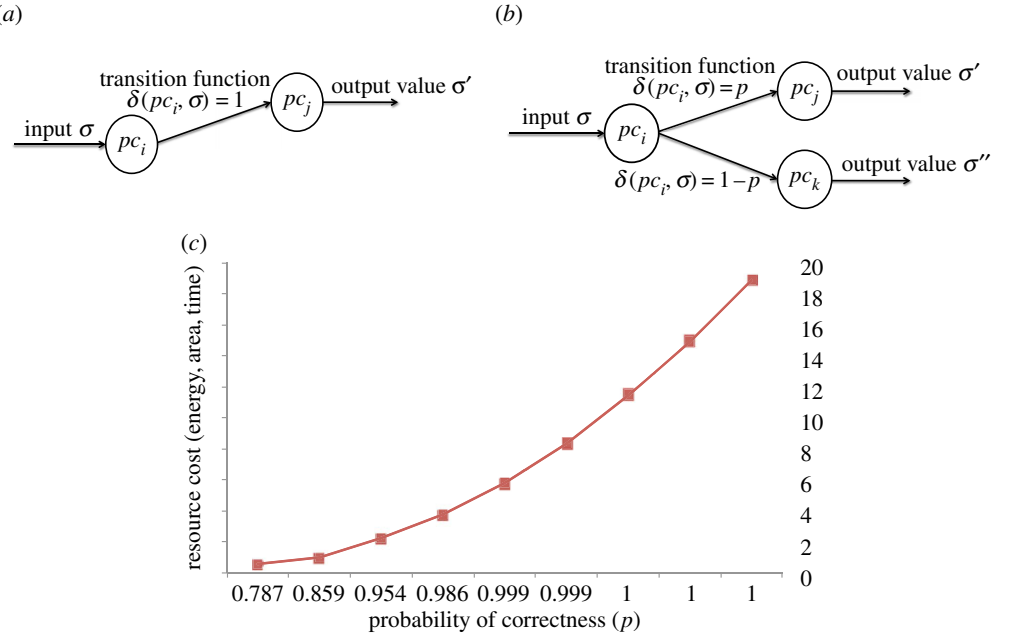

Figure 2. The essential difference between a single step of a (*a*) BRAM where each step is deterministic, and all the operations are always exact or correct and hence equally expensive and there is no mechanism of trading energy for exactness; (*b*) RaBRAM where each step is naturally probabilistic through randomization, and the more erroneous the transition, the lower the energy and (*c*) a qualitative depiction of the relationship between exactness and the associated energy cost through the energy–probability relationship. (Online version in colour.)

Error is viewed therefore, not as an extraneous and misdirected or a misdirecting accident, but as an essential part of the process under consideration. . .'. We remark though that his goal was to recommend and work towards correcting error as opposed to using it to trade for savings. Thus, this line of work led to fault-tolerant computing aimed to realize exact computing from inexact building blocks.

Thus, and as far as we can tell, for the first time, in a paper entitled 'Proof as experiment: probabilistic algorithms from a thermodynamic perspective' published in 2003 [18], we showed that inexact computing from inexact building blocks rooted in physics can result in energy efficiencies. This was done in a model that has the power of a Turing machine, or is Turing-complete in computer science parlance. This subsequently led to realizing inexact computing from inexact hardware building blocks, also aimed at garnering energy savings. We were inspired by the classical notions of algorithmic inexactness, but introduced a fundamental new concept of *energy complexity* and a model embodying it called an RaBRAM.[1] *Each transition or step in this model can be rendered inexact through a probabilistic approach, consisting of indivisible transition functions which are 'thermodynamically aware'* (*figure 2*). *Specifically, the energy needed to perform each step is proportional to the probability that the step is correct. Furthermore, the relationship between energy and correctness conforms to the underlying physics. By contrast, classical randomized algorithmic models are based on the standard RAM which is also equivalent to a Turing machine, where each of the transitions or steps is exact. Randomness enters the computation by auxiliary or externally produced sources as an extra variable which is an input to an exact deterministic transition function.* Another abstract foundational model that relates energy (and, hence, physical cost) with the inherent need of a program to be synchronized and, therefore, knowledge time in some exact sense was also proposed in [19]; although, this model was intended to be used in the context of realizing exact computations.

[1]An acronym for randomized bit-level (Boltzmann) random access machine.

To demonstrate the feasibility of trading physical energy for error in the RaBRAM, we introduced in [18] the *distinct vector problem* (DVP) which seeks to determine whether an input vector I with n elements drawn from the set $\{0,1\}$ has the value 1 in all of its locations. Using the RAM and the exact computing model as the baseline, akin to the relaxation in classical randomized algorithms, RaBRAM outputs could differ from the exact version as follows: in our DVP example in the exact version of the RAM, given an input vector I, the output is 1 iff all of the n elements have a value 1. In the RaBRAM, even when all of the n elements are 1, the output is 1 with probability $1 - 1/n^c$ for some constant c (see [18] for technical details) and can therefore have the value 0 with probability $1/n^c$. Following this we showed for the DVP [18] is that the energy savings through inexactness grow n as $n \to \infty$. We note in passing that the models from [18] provided a framework for computing the *energy complexity* in a technology-independent manner, akin to classical space and time complexities [20] through asymptotic measures. Thus, using physics and, hence, physical building blocks that are inexact, we realized an inexact 'system' for solving the DVP.

6. A more general model of an inexact computing system

A more general version of modelling inexactness which we refer to as an *inexact RAM* (IRAM) in this paper could embody any of the ways that inexactness could be induced. Conceptually, an IRAM allows a weaker contract between the inputs and outputs compared with the RAM. First, it has the computing power (trivially shown) of a Turing machine. Second, there is a concrete notion of an exact computation defined through a corresponding RAM for each problem Π which is meant to be solved in an energy-efficient manner. Using more recent developments, the IRAM could also be more efficient in terms of the time and physical area of the hardware realizations by trading away exactness.

Continuing, the coarse-grained relationship between the inputs and outputs at the level of the IRAM is really a composition or aggregation of finer-grained contracts drawn from its physical hardware and software building blocks. System hardware consists of gates or switches at the most basic level used to construct circuits, which, in turn, are used to realize computing, storage elements, and a way of interconnecting them. If the goal is to realize an exact computing system, then inexactness between the inputs and outputs at any of these levels is anathema because, if left uncorrected (in the von Neumann sense), then it would eventually also imply an inexact system at the more abstract application software and algorithmic levels. For example, a digital adder in a processor is supposed to add using the canonical definition of addition, and an inexact adder that provides an output of 14 when the inputs are 8 and 5 would be unacceptable if it is used in a larger exact computing system because its output value will eventually be used at higher levels of abstraction! Because the history of computing platforms with the power of Turing machines has been about providing exactness, it has been critical to be exact at all of the software and hardware system levels.

In a deviation from this practically universal practice to realizing exact computing system hardware (and building on [18]), we started extending our concept of inexactness to characterize and understand the elemental building blocks at the lowest levels of abstraction, namely gates or switches. This work was done in two phases. The first phase involved developing thermodynamically aware switches that allowed trading exactness for energy gains [21]. At the same time, we also started characterizing inexact Boolean gates based on CMOS technology [22], ubiquitous to building computer hardware. Besides being the essential building blocks of inexact computing models such as the IRAM, these switches and gates preserved the properties we outlined above: they are a relaxation of exact switches or gates, respectively, with the contract between the inputs and outputs being explicitly stated, and, additionally, each switch or gate has an associated energy–exactness trade-off that is quantitatively characterized. To illustrate the latter property, as shown in figure 3 for a CMOS inverter, we characterized the E–p relationship [22] for the first time in 2004.

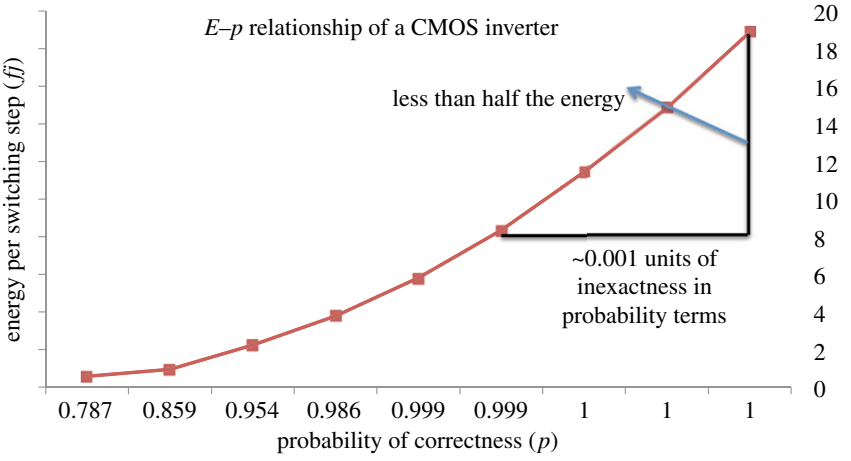

Figure 3. The relationship between the degree of exactness characterized as *p* the probability of correctness and the associated energy cost from [22] of a CMOS inverter. (Online version in colour.)

Further building on this, we increased the complexity of the hardware blocks that we rendered inexact, hardware implementations of algorithms used for integer arithmetic in the form of adders and through multipliers. Our inexact adders and through multipliers were designed and through simulations [23], the relationship between the quality of their outputs namely the error, and the associated energy relationship was characterized there.

At that point, inexactness in our context was implemented using probabilistic gates or elements, because our initial motivation was based on the impending hurdles to Moore's law and the possibility that CMOS transistors and hence circuits built from them could be naturally probabilistic [24] (readers are referred to [25] for a more details). However, substantial innovation in CMOS process technologies has sustained Moore's law and, therefore, the march of reliable devices continued unhindered. Therefore, starting in 2008, we embarked on alternative approaches to realizing inexactness, again in the context of integer arithmetic [26,27]. Our more recent approaches that enforced an inexactness regime on arithmetic units are akin in spirit to classical and widely applied methods such as *bit width truncation* in digital signal processing (DSP). However, as noted earlier they are different from the use of exact variable precision in *floating point* units. However, the technical details are very different of course. Starting in 2008, several other groups have also worked in parallel on similar ideas and a non-exhaustive list of some of these contributions include [28–30].

We also embarked upon an effort to conceive of an architecture model based on a *system-on-a-chip* (SOC) style, and published it in 2006; this was intended to support applications that needed certain portions to be exact, while other portions could be inexact [31]. This work provides a natural template for considering inexact *accelerators* with an exact core in future computer architectures; refer to the keynote lecture from 2012 [32].

7. Engaging the application layer and the 'line in the sand' metaphor

To recapitulate, while a computing system is essentially constructed using the layers in figure 1, in the end, its overarching goal is to support applications. Invariably, these applications are meant to solve a problem from a domain, and the needs of the application were negotiated with the computing system from figure 1 through an algorithm. Traditionally, once an algorithm was designed for a particular application, its realization through system software and through hardware was exact. Consequently, a randomized algorithm for primality testing, or an approximation algorithm for the TSP, relied on exact software and hardware to realize the

implementations, and no additional inexactness was introduced into the computing system itself. Thus, there is a clear *line in the sand* as shown in figure 1 which was not crossed traditionally as far as inexactness was considered, and it is this line which provided a boundary between exactness and inexactness.

Viewed from this perspective, the line in the sand that we crossed starting in 2003 and that now continues to be crossed by us and other groups, is to take the contract between the application and its algorithmic instantiation, and not honour it at the system level. This was true of the example of adders we described in §5 where the original contract that was given to us at the application level—image processing of radar images through exact addition is one example—was not honoured. Rather, the additions were made inexact in a manner that did not respect the contract; this resulted in an inexact hardware architecture for the application whose behaviour was modelled through simulations. Stated in another way, the algorithmic expectation of the image processing application was altered by the system designers without consultation with the original algorithm designers. For radar images, the fast Fourier transform was the module that was rendered inexact in [23] at the hardware level.

Significant strides on advancing the initial momentum of inexact computing worthy of special mention were undertaken starting in the period around 2007. While initial work explored the usage of voltage over scaled circuits as the means for introducing error in a wide range of signal and video processing hardware (such as discrete cosine transform [28] and motion estimation [29]), the notion of significance-driven voltage was used to explore inexact design optimizations spanning multiple layers of design abstraction. For instance, *scalable-effort hardware* [30] combines voltage over scaling (physical-layer), precision–reduction (architectural-layer) and computation reductions (algorithm-layer) to maximize the resource efficiency gains, and a *reciprocative error compensation* frameworks [33] provide an optimization framework for tuning the coefficients of the DSP blocks to hide inexactness from the underlying building blocks. Continuing, a recent body of work further explored and demonstrated the potential of inexact computing in the context of emerging *recognition, mining* and *synthesis* (RMS) workloads where they have shown that 67–96% of these applications' execution is spent in kernels that can accept and live with inexactness in a significant manner [34].

While our line in the sand might, in fact, seem to provide a clear divide between an exact and inexact computing paradigm, the domain of *DSP* has instances that, at first glance, seem to make this boundary somewhat fuzzy. A particularly interesting instance to illustrate this point is the concept of *approximate DSP* [35]. In this approach, the application specialist and the algorithm designer work hand in hand—they might even be the same person—to alter the algorithm to meet the needs of the signal processing application to achieve greater efficiencies. Thus, there need not be an exact version or contract specified, and the inexact relaxation is made relative to the application developer's notion of adequacy based on the domain knowledge, often referred to as the 'golden output' or design. As a result, an *a priori* exact definition as in the context of the TSP or of primality testing need not be there. So, in our strict sense, this approach would not result in strictly inexact implementations, because the relaxation is done relative to a problem whose exact specification is not explicit. We return to a more detailed discussion of this point of view in the next section. Furthermore, approximate DSP can be thought of as a particularly useful variant of iterative refinement [36,37] which realizes inexactness purely at the algorithmic level by not crossing the line in the sand. We wish to point out that this approach to trading the quality of a solution for savings is prevalent in the domain of *analogue systems* and is perhaps common knowledge in this domain [38]. However, we are concerned with digital computing in this paper and thus do not delve too deeply into the legacy from analogue system design here.

Turning to software, a body of work intended to provide a framework at the application programming and software level [39] suggested weakening the specification relative to what was intended. This body of work (an additional early example being [40]) aimed at applications in a domain agnostic manner; as opposed to being focused on DSP for example. The goal was to enhance programmer productivity by allowing algorithms to be implemented with a degree of 'accuracy' that is lower than that intended initially. As a consequence, both the programming or

software engineering effort, as well as the performance of the application, were anticipated to be better.

While we have been dwelling on hardware and physics approaches to draw the line in the sand, let us consider *optimizing compilers* to extend this idea into system software. The success of *optimizing compilers* [41,42] without which compiled programs would reach their promised performance goals today serve as a good vehicle to understand inexactness at the level of system software. Techniques for improving program performance such as register allocation [43,44], instruction scheduling [45,46] and others are designed to be application agnostic in the following sense: they are meant to work with any algorithm described as a program in a particular language, and then transformed while preserving the meaning of the original program. In other words, the transformed program's input–output behaviour is expected to be identical to that of the original program. The notion of exactness in the context of software is inherited from the specification of the application that is being compiled. Thus, exact optimizations through a compiler take a program P implementing an algorithm, and produce a program P' such that whenever an input I to P produces an output O, P' produces the same output. Therefore, inexactness in the software context can, by extension, also be defined in a manner akin to the approaches that were outlined for hardware. Rather than considering an algorithm and its inexact counterpart, can we instead consider the module's software implementation as being inexact in the future?

8. From digital signal processing to search, big data and beyond

Returning to iterative refinement, if we view the golden output of iterative algorithms as our exact solution, then are not solutions that are achieved with fewer iterations inexact? This question raises a conundrum of some import. Simplistically viewed, an iterative refinement scheme implements the following principle: *if there is a task that is repeated n times and each repetition improves the quality of the output, then fewer iterations mean lower quality and cost, and thus cost can be traded for quality.* In this abstract view, this does seem to, at least informally, satisfy our criteria for inexactness. However, intuitively, why does this not seem to be natural? Perhaps, one reason is that in these contexts, there is no absolute notion of a correct output of the element or building block being rendered inexact; the iterative structure in the example of approximate DSP is an example of such a building block. In other words, the notion of adequacy of the iterative structure is intimately tied to the domain of its use which, in turn, is determined by mediating between the application and its algorithm. Consequently, the parameters inducing inexactness are intimately tied to the domain where the algorithm or design is to be used. In the context of approximate DSP, the number of iterations to achieve the golden solution are based on the system in which the FFT or filter is to be deployed which, in turn, depends on the expertise and insights of the domain specialist performing the design.

This notion of approximating a solution through domain knowledge *without having an a priori characterization of the correct output O which is a key element of inexactness as we formulated it in this paper* is increasingly evident in emerging application domains and their workloads such as *big data analytics, machine learning, cortical processing-inspired computing* [47] and *search* to name a few. In these domains and with the computing platforms and models increasingly being *data centres* and the *cloud*, respectively, the concept of an *a priori* correct answer is hard if not impossible to come by. Thus, the exact solution is perhaps the best that the designers of the system could construct at the time. Algorithms for internet searches based on the (now) classical *PageRank* [48] scheme is a good example. Over time, the quality of search has improved, because of both increased knowledge through data gathered over time and more computing power and thus the golden solution keeps getting better.

9. A case for application-system co-design

In all of these contexts, the domain expert's role in understanding and defining the criteria of acceptability of what is 'good enough' is central. Nevertheless, computer system design trudges

on, generally agnostic of specific application domain needs. Our example of an optimizing compiler or a Boolean gate at two extremes serve to make this point. In a general computing system supplied by a manufacturer or vendor, neither the compiler nor the gate is rendered inexact, and tuned to benefit page ranking or to the needs of the cortical processor. If the reach of inexactness were to broaden to include system layers, then we anticipate that co-designing the system guided by considerations of quality of the output, which is either subjective or is dependent on a criteria set by a domain expert, will play an increasing role. Resource pressures in general and energy or power constraints in particular will be the significant constraints driving this movement in the near term. To see this, one need only consider the discussion about data centres [49] and their alarming energy footprint [50]. As an early example of this approach to co-design, the customization to floating point through workload statistics to accelerate key kernels drawn from the domain of stochastic climate models [51] is an important example. Thus, we opine that an accelerator-based system akin to SOC designs (see [52,53]) in embedded computing is increasingly going to play a role in the context of more traditional computing system design. If this trend succeeds, abstraction and cross-layer transparency (such as the recent proposals to developing inexact ISAs and support [54,55]) are likely to play less of a role, and application and technology aware cross-layer designs are likely to emerge as equal partners with general purpose abstractions; time will tell. If history is our guide, then, in all likelihood, inexactness will be induced by extracting a catalogue or library of 'inexact primitive operations' from these domains which will serve as building blocks. It is very desirable of course for the individual elements of the catalogue to be the vehicles for mediating between the application and the system realization. At the hardware level, these could be the enhanced variants of adders, multipliers and FFT engines we have already seen and others, with appropriate, yet to be determined support from system software.

10. And the march continues

Much of what we described earlier as inexact designs by us and other groups represents some of the early departures from von Neumann's legacy, with the algorithmic foundations of computer science as a backdrop. The beginnings of inexactness in our own case at the physics and hardware level with an emphasis on the physical realization of computing systems spans the past decade. It is heartening to see that over the past 5 years or so, progress has become much more vigorous with ambitious projects emerging on the national and international scene at all levels of the system hierarchy. Without attempting to establish any sense of priority or relative importance, some of the more interesting forays range all the way from programming languages and vehicles for expressing inexactness [56,57], and of course circuit-level innovations exemplified by earlier studies [29,58]. The list is by no means intended to be an exhaustive survey, because the goal of this paper is meant to be more of a perspective. At last count, we could catalogue over 90 papers written by various groups from around the world that touched upon inexactness explicitly in some fashion (see [59,60] for an overview and another perspective).

11. Caveats and concluding musings

One of the elements of a computation being inexact is, put quite simply, that the output is not exact. Going back to the beginning of this paper and the legacy from DTMs, exactness has a well-defined deterministic foundation. Computer science, as evident from the earlier sections, provided a framework for inexactness on which we built our earlier work. It is useful to digress and note that there is a somewhat alluring connection between strings of bits which are outputs of Turing machines, being imprecise, and concepts from the complementary discipline of *information theory*. If the reader can indulge me for a few more minutes, then I will, through this digression, remark on some essential differences between the computer science-inspired notions of inexactness, and those that could be derived from information theory.

Classically, information theory is concerned with the transmission of information over channels following Shannon's seminal *binary symmetric channel* (BSC) as a foundational construct [61]. Thus, while it bears some similarities to input strings being converted into output strings with error associated with probabilities, it is relatively easy to show, through traditional complexity theoretic notions, that the family of functions that can be 'computed' through a functional composition of BSC abstractions are more limited in a provable sense than Turing computable functions. The second point is less of a technical separation and more of a conceptual remark. One can always associate a notion of information theoretic entropy after the fact with an output string of a computation, which can be some measure of distance from the correct or exact output. Superficially, this might seem similar to the manner in which we relax or weaken the contract in the context of randomized algorithms, the hardware that we worked on, and other approaches to inexactness described earlier. However, information-theoretic entropy being a property of strings without any connection to the problem being specified or the procedure used to compute, does not relate the degree of inexactness to the problem being solved in a constructive way. Thus, as a metric, it does not capture the essence of what we wish to achieve in constructively trying to capture the elements of the computation that produce the more 'entropic' output. For example, in the DVP, the output is meant to distinguish the unit vector from one with a zero in it. In computer science, this notion of a decision question (see [62]) captures the essence of constructively or algorithmically solving a problem. This is in line with the views of the pioneer of information theory, Shannon [63], who notes that 'While we feel that information theory is indeed a valuable tool in providing fundamental insights into the nature of communication problems and will continue to grow in importance, it is certainly no panacea for the communication engineer or, a fortiori, for anyone else'.

Notions of inexactness can be associated with the outcome (output) always *using the problem statement as a basis*, as opposed to tying it to the syntax of strings agnostic of the semantics of the problem being solved. In the inexact version of the DVP [18], the output is simply a 1 (the input is a unit vector) or zero (the input has a zero in at least one position). Thus, a model which is a probabilistic variant of a Turing machine or if energy is our interest, an RaBRAM, is essential to reason about the cost and quality, as opposed to the BSC which, whereas powerful, elegant and influential on a historic scale, does not capture the essential element of *effective procedures for computation* [64]. In fact, very powerful theories that meld concepts akin to information theoretic entropy with computation are not lacking and exist in the work of Chaitin [65] and Kolmogorov [66]. However, for the same reasons that are outlined above, their non-constructive nature has limited their adoption in describing, constructively, designing and analysing upper bounds on costs or error, even though they have proved useful to prove lower bounds.

12. Epilogue

As with any field where activity grows, multiple views emerge as to what a particular concept means. In our case, the author believes that this point of criticality is being reached with inexact computing, sometimes referred to more recently also as approximate computing. Many influential ideas to consider inexactness spanning the entire gamut of elements that constitute a computing system are rapidly emerging. This article embodies an attempt to place what inexactness means in perspective without making any original research contributions. It is not intended to be a survey for assigning priority or credit, and the reader is encouraged to peruse other sources of material from the broader literature for those purposes. Furthermore, given the personal experience that this paper is intended to embody, there is a necessary bias towards the hardware layer which has been the exclusive realm of our own work thus far. The overarching theme that inexactness as described in this paper is meant to capture, involves the fact that when computing systems and applications that run on them are not correct by design, significant savings in the resources such as the energy consumed can be the result.

This point is very significant in that inexact computing as pursued by us and distinguished colleagues is not about fault-tolerance or resilient computing, in which, if the line in the sand is crossed, the goal is to quickly reverse this and realize exact solutions in the end (see *algorithmic noise tolerance* [67] for DSP, RAZOR [68], ERSA [69], self-aware computing [70,71] for elegant examples of exact computing from inexact hardware building parts). Specifically, a system that is found to be erroneous is corrected to the best of our ability to restore, as much as possible, something that would approach to being considered exact. Von Neumann's classical constructions set the tone for this style of fault-tolerant thinking. In this approach, the end result might, in fact, not be exact because it is not possible to be fully exact, but the intention is to try and achieve an exact system. In contrast, those of us that pursued inexact computing deliberately work towards designing (elements of) computing systems at design points that are not exact, resulting in an inexact design space, in return for significant efficiencies. The energy–probability relationship from figure 3 for a CMOS inverter is an early example of such a design space. Using inverters at points with the probability of correctness, $p < 1$ captures the essence of inexactness. In traditional fault-tolerant computing, given an inverter with $p < 1$, the goal is the exact opposite; to try and achieve a value of p as close to 1 as possible by paying additional cost.

Now, the challenges are in finding significant domains for applicability of these ideas at scale. By necessity, this implies finding applications where the use of inexactness can be justified, both in terms of the application domain being able to tolerate lack of exactness, and in terms of demonstrating significant savings. In this regard, the opportunities are many. At one end, we have computing in the small, namely embedded computing with its concomitant battery life pressures, but where an applications ability to live with inexactness is intrinsic. At the other extreme, computing in the large with data centres and the cloud with their rapidly increasing reach and energy footprint could certainly use significant energy efficiencies [49]. Emerging big-data centric workloads certainly provide plenty of opportunities to tolerate inexactness: internet search is an often quoted example. The next decade or so will, we hope and anticipate with enthusiasm, see the unfurling of inexactness into all of these exciting applications and alter the paradigm of computing system hardware and software design! In addition, neuroscience and social science methods are likely to play a significant role in guiding the concepts of what is tolerable and hence acceptable as inexact designs. Finally, emerging cortical processors for computer vision are a notable example where a systematic co-design methodology [38] between all these different application and hardware designs would be needed.

Acknowledgements. The author benefited from several discussions, both technical and philosophical, with Anand Raghunathan. Additionally, very special thanks to Avinash Lingamneni for being a terrific sounding board, critique and a source of support in facilitating the editing and proofing of this document. The wide-ranging nature of the thoughts and ideas in this paper certainly benefited from discussions with a range of colleagues, my past students, post-docs and co-authors, too numerous to list here; I express my appreciation to all of them for stimulating discussions and credit their contributions in helping me shape my thinking. Don Fussell provided the characterization of information efficiency, as distinct from inexactness. I also acknowledge a Schonbrunn visiting fellowship at the Hebrew University and the residency in Jerusalem as the time and place where these early concepts were formulated. Our work was initially enabled by DARPA seedling contract number F30602-02-2-0124 and Bob Graybill's role as a program manager was crucial to our early development from conception to demonstrable results. We also thank Vivek De of Intel for facilitating funding to enable the early CMOS validations.

References

1. Turing AM. 1936 On computable numbers, with an application to the entscheidungs problem. *Proc. Lond. Math. Soc.* **42**, 230–265.
2. Mauchly JW, Eckert JP. 1947 *Electrical numerical integrator and calculator*. US Patent 3,120,606. pages Filed on 26 June 1947; granted on 4 February 1964.
3. Randall 5th, A. 2006 A lost interview with ENIAC co-inventor J. Presper Eckert. *Computer World* (accessed 24 April 2011).

4. Von Neumann J. 1956 Probabilistic logics and the synthesis of reliable organisms from unreliable components (based on caltech lectures of von neumann in 1952). In *Automata studies* (eds CE Shannon, J McCarthy). Princeton, NJ: Princeton University Press.
5. Rubio-Gonzalez C, Nguyen C, Nguyen HD, Demmel J, Kahan W, Sen K, Bailey DH, Iancu C, Hough D. 2013 Precimonious: tuning assistant for floating-point precision. In *Proc. of SC13*, New York, NY: ACM.
6. Misra J, Chandy KM. 1988 *Parallel program design: a foundation*. London, UK: Addison-Wesley.
7. Vazirani VV. 2001 *Approximation algorithms*. New York, NY: Springer.
8. Christofides N. 1976 Worst-case analysis of a new heuristic for the travelling salesman problem. Report 388, Graduate School of Industrial Administration, CMU. Pages Filed on 26 June 1947; granted on 4 February 1964.
9. Motwani R, Raghavan P. 1995 *Randomized algorithms*. New York, NY: Cambridge University Press.
10. Mitzenmacher M, Upfal E. 2005 *Probability and computing: randomized algorithms and probabilistic analysis*. New York, NY: Cambridge University Press.
11. Rabin MO. 1963 Probabilistic automata. *Inf. Control* **6**, 230–245. (doi:10.1016/S0019-9958(63)90290-0)
12. Solovay R, Strassen V. 1977 A fast monte-carlo test for primality. *SIAM J. Comput.* **6**, 84–85. (doi:10.1137/0206006)
13. Karp RM. 1977 Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Math. Oper. Res.* **2**, 209–224. (doi:10.1287/moor.2.3.209)
14. Hennessy JL, Patterson DA. 2003 *Computer architecture: a quantitative approach*. New York, NY: Morgan Kaufmann Publishers.
15. Goldberg D. 1991 What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* **23**, 5–48. (doi:10.1145/103162.103163)
16. IEEE standard for floating-point arithmetic. 2008 *IEEE Std 754-2008*, pp. 1–70.
17. Cook SA, Reckhow RA. 1973 Time bounded random access machines. *J. Comput. Syst. Sci.* **7**, 354–375. (doi:10.1016/S0022-0000(73)80029-7)
18. Palem KV. 2003 Proof as experiment: probabilistic algorithms from a thermodynamic perspective. In *Proc. Int. Symp. on Verification* (*Theory and Practice*).
19. Palem KV. 2010 The arrow of time through the lens of computing. In *Time for verification* (eds Z Manna, DA Peled). Lecture Notes in Computer Science, no. 6200, pp. 362–369. Berlin, Germany: Springer.
20. Knuth DE. 2008 *The art of computer programming*, 1st edn. New York, NY: Addison-Wesley Professional.
21. Palem KV. 2005 Energy aware computing through probabilistic switching: a study of limits. *IEEE Trans. Comput.* **54**, 1123–1137 (abridged form appeared as: K. V. Palem, Energy Aware Computing through Randomized Switching, *Technical Report GIT–CC-03-16, Georgia Inst. of Technology*, May 2003).
22. Cheemalavagu S, Korkmaz P, Palem KV. 2004 Ultra low-energy computing via probabilistic algorithms and devices: CMOS device primitives akrishna v palemnd the energy-probability relationship. In *Proc. Int. Conf. on Solid State Devices and Materials*, pp. 402–403.
23. George J, Marr B, Akgul BES, Palem KV. 2006 Probabilistic arithmetic and energy efficient embedded signal processing. In *Proc. of IEEE/ACM Int. Conf. Compilers, Architectures and Synthesis for Embedded Systems*, pp. 158–168.
24. Kish LB. 2002 End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Phys. Lett. A* **305**, 144–149. (doi:10.1016/S0375-9601(02)01365-8)
25. Palem K, Chakrapani LNB, Kedem ZM, Lingamneni A, Muntimadugu KK. 2009 Sustaining Moore's law in embedded computing through probabilistic and approximate design: retrospects and prospects. In *Proc. Int. Conf. Compilers, Architectures, and Synthesis for Embedded Systems*, pp. 1–10.
26. Chakrapani LN, Muntimadugu KK, Lingamneni A, George J, Palem KV. 2008 Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation. In *Proc. of IEEE/ACM Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 187–196.
27. Lingamneni A, Enz CC, Nagel J-L, Palem KV, Piguet C. 2011 Energy parsimonious circuit design through probabilistic pruning. In *Proc. Design, Automation and Test in Europe Conf.* pp. 764–769.

28. Banerjee N, Karakonstantis G, Roy K. 2007 Process variation tolerant low power DCT architecture. In *Proc. of Design, Automation and Test in Europe Conf.*, pp. 630–635.
29. Mohapatra D, Karakonstantis G, Roy K. 2009 Significance driven computation: a voltage-scalable, variation-aware, quality-tuning motion estimator. In *Proc. of ACM/IEEE Int. Symp. on Low Power Electronics and Design*, pp. 195–200.
30. Chippa VK, Mohapatra D, Raghunathan A, Roy K, Chakradhar ST. 2010 Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *Proc. Design Automation Conf.*, pp. 555–560.
31. Chakrapani LN, Akgul BES, Cheemalavagu S, Korkmaz P, Palem KV, Seshasayee B. 2006 Ultra efficient embedded SoC architectures based on probabilistic CMOS technology. In *Proc. Design, Automation and Test in Europe Conf.*, pp. 1110–1115.
32. Krishnamurthy RK. 2012 High performance energy efficient near threshold circuits: challenges and opportunities. *MICRO Near Threshold Computing Workshop Keynote*.
33. Lingamneni A, Basu A, Enz C, Palem K, Piguet C. 2013 Improving energy gains of inexact DSP hardware through reciprocative error compensation. In *The 50th Design Automation Conf.* pp. 20:1–20:8.
34. Chippa VK, Chakradhar ST, Roy K, Raghunathan A. 2013 Analysis and characterization of inherent application resilience for approximate computing. In *Proc. 50th Annual Design Automation Conf.*, DAC'13, pp. 113:1–113:9. New York, NY: ACM.
35. Hamid Nawab S, Oppenheim AV, Chandrakasan AP, Winograd JM, Ludwig JT. 1997 Approximate signal processing. *J. VLSI Signal Process.* **15**, 177–200. (doi:10.1023/A:1007986707921)
36. Wilkinson JH. 1963 *Rounding errors in algebraic processes*. Englewood Cliffs, NJ: Prentice Hall.
37. Moler CB. 1967 Iterative refinement in floating point. *J. ACM* **14**, 316–321. (doi:10.1145/321386.321394)
38. De Micheli G. 1994 *Synthesis and optimization of digital circuits*. New York, NY: McGraw-Hill.
39. Rinard M. 2003 Acceptability-oriented computing. In *Companion of the 18th Annual ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'03, pp. 221–239. New York, NY: ACM.
40. Rinard M, Cadar C, Dumitran D, Roy DM, Tudor Leu, Beebee Jr, WS. 2004 Enhancing server availability and security through failure-oblivious computing. In *Proc. the 6th Conf. Symp. on Opearting Systems Design and Implementation*, vol. 6, OSDI'04, pp. 21–21. Berkeley, CA: USENIX Association.
41. Cocke J, Schwartz JT. 1970 *Programming languages and their compilers: preliminary notes*. New York, NY: Courant Institute of Mathematical Sciences, New York University.
42. Allen FE. 1971 A basis for program optimization. In *IFIP Congress* (*1*), pp. 385–390.
43. Chaitin GJ. 1982 Register allocation & spilling via graph coloring. In *Proc. the 1982 SIGPLAN Symp. on Compiler Construction*, SIGPLAN'82, pp. 98–105. New York, NY: ACM.
44. Chow FC, Hennessy JL. 1990 The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.* **12**, 501–536. (doi:10.1145/88616.88621)
45. Muchnick SS. 1997 *Advanced compiler design and implementation*. San Francisco, CA: Morgan Kaufmann Publishers.
46. Palem KV, Simons BB. 1993 Scheduling time-critical instructions on RISC machines. *ACM Trans. Program. Lang. Syst.* **15**, 632–658. (doi:10.1145/155183.155190)
47. Seo J *et al.* 2011 A 45 nm CMOS neuromorphic chip with a scalable architecture for learning in networks of spiking neurons. In *Custom Integrated Circuits Conf.* (*CICC*), *2011 IEEE*, pp. 1–4.
48. Brin S, Page L. 1998 The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* **30**, 107–117. (doi:10.1016/S0169-7552(98)00110-X)
49. Ferdman M *et al*. 2012 Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *SIGARCH Comput. Archit. News* **40**, 37–48. (doi:10.1145/2189750.2150982)
50. Bronk C, Lingamneni A, Palem KV. 2010 Innovation for sustainability in information and communication technologies (ICT). Technical report, James A. Baker III Institute for Public Policy, Rice University.
51. Düben PD, McNamara H, Palmer TN. 2013 The use of imprecise processing to improve accuracy in weather and climate prediction. *J. Comp. Phys.*
52. Esmaeilzadeh H, Sampson A, Ceze L, Burger D. 2012 Neural acceleration for general-purpose approximate programs. In *Proc. 2012 45th Annual IEEE/ACM Int. Symp. Microarchitecture*, MICRO'12, pp. 449–460. Washington, DC: IEEE Computer Society.

53. Zidong Du, Lingamneni A, Chen Y, Palem K, Temam O, Chengyong Wu. 2014 Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *Design Automation Conf.* (*ASP-DAC*), *2014 19th Asia and South Pacific*, pp. 201–206.
54. Esmaeilzadeh H, Sampson A, Ceze L, Burger D. 2012 Architecture support for disciplined approximate programming. *SIGARCH Comput. Archit. News* **40**, 301–312. (doi:10.1145/2189750.2151008)
55. Venkataramani S, Chippa VK, Chakradhar ST, Roy K, Raghunathan A. 2013 Quality programmable vector processors for approximate computing. In *Proc. 46th Annual IEEE/ACM Int. Symp. on Microarchitecture*, MICRO-46, pp. 1–12. New York, NY: ACM.
56. Sampson A, Dietl W, Fortuna E, Gnanapragasam D, Ceze L, Grossman D. 2011 ENERJ: approximate data types for safe and general low-power computation. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI'11, pp. 164–174. New York, NY: ACM.
57. Carbin M, Misailovic S, Rinard MC. 2013 Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proc. 2013 ACM SIGPLAN Int. Conf. Object Oriented Programming Systems Languages & Applications*, OOPSLA'13, pp. 33–52. New York, NY: ACM.
58. Lingamneni A, Enz C, Palem K, Piguet C. 2013 Designing energy-efficient arithmetic operators using inexact computing. *J. Low Power Electron.* **9**, 141–153. (doi:10.1166/jolpe.2013.1249)
59. Palem K, Lingamneni A. 2012 What to do about the end of Moore's law, probably! In *Proc. 49th Annual Design Automation Conf.*, DAC'12, pp. 924–929. New York, NY: ACM.
60. Palem K, Lingamneni A. 2013 Ten years of building broken chips: The physics and engineering of inexact computing. *ACM Trans. Embedded Comput. Syst.* 12, 87:1–87:23.
61. Shannon CE. 1948 A mathematical theory of communication. *Bell Syst. Tech. J.* **27**, 379–423, 623–656. (doi:10.1002/j.1538-7305.1948.tb01338.x)
62. Garey MR, Johnson DS. 1979 *Computers and intractability: a guide to the theory of NP-completeness*. New York, NY: W.H. Freeman.
63. Shannon CE. 1956 The bandwagon. *IEEE Trans. Inf. Theory* **2**, 3. (doi:10.1109/TIT.1956.1056774)
64. Minsky M. 1967 *Computation: finite and infinite machines.* Englewood Cliffs, NJ: Prentice-Hall.
65. Chaitin GJ. 1977 Algorithmic information theory. *IBM J. Res. Dev.* **21**, 350–359. (doi:10.1147/rd.214.0350)
66. Kolmogorov A. 1968 Logical basis for information theory and probability theory. *IEEE Trans. Inf. Theory* **14**, 662–664. (doi:10.1109/TIT.1968.1054210)
67. Hegde R, Shanbhag NR. 1999 Energy-efficient signal processing via algorithmic noise-tolerance. In *Proc. Int. Symp. on Low Power Electronics and Design*, pp. 30–35.
68. Ernst D *et al.* 2003 Razor: a low-power pipeline based on circuit-level timing speculation. In *Proc. IEEE/ACM Int. Symp. Microarchitecture*, pp. 7–18.
69. Bau J, Hankins R, Jacobson Q, Mitra S, Saha B, Tabatabai, AA. 2007 Error resilient system architecture (ERSA) for probabilistic applications. (doi:10.1109/DATE.2010.5457059)
70. Hoffmann H *et al*. 2012 Self-aware computing in the angstrom processor. In *Proc. 49th Annual Design Automation Conf.*, DAC'12, pp. 259–264, New York, NY: ACM.
71. Bose, P. 2005 Variation-tolerant design. *Micro IEEE* **25**, 5–5. (doi:10.1109/MM.2005.40)