Incorrect Systems: It's not the Problem, It's the Solution[*]

Christoph M. Kirsch
Department of Computer Sciences
University of Salzburg
ck@cs.uni-salzburg.at

Hannes Payer
Department of Computer Sciences
University of Salzburg
hpayer@cs.uni-salzburg.at

ABSTRACT

We present an overview of state-of-the-art work in the engineering of digital systems (hardware and software) where traditional correctness requirements are relaxed, usually for higher performance and lower resource consumption but possibly also for other non-functional properties such as more robustness and less cost. The work presented here is categorized into work that involves just hardware, hardware and software, and just software. In particular, we discuss work on probabilistic and approximate design of processors, unreliable cores in asymmetric multi-core architectures, best-effort computing, stochastic processors, accuracy-aware program transformations, and relaxed concurrent data structures. As common theme we identify, at least intuitively, "metrics of correctness" in each piece of work which appear to be important for understanding the effects of relaxed correctness requirements and their relationship to performance improvements and resource consumption.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General

General Terms

Algorithms, Design, Performance, Reliability

Keywords

relaxed correctness, probabilistic computing, performance, scalability, power consumption, robustness

[*]This work has been supported by the National Science Foundation (CNS1136141), the European Commission (ArtistDesign NoE on Embedded Systems Design, 214373), and the Austrian Science Fund (RiSE NFN on Rigorous Systems Engineering, S11404-N23).

1. INTRODUCTION

We acknowledge the emergence and advocate the study of relaxed, possibly quantitative approaches to describing and establishing the correctness of digital systems. The notion of hardware or systems software either computing the correct result for a given input or not has been the dominating principle in systems engineering for a long time whereas other areas of computer science such as scientific computing and machine learning as well as audio, video, and image processing, to name a few, have adopted relaxed notions of correctness early on. While the promise in special-purpose areas is typically higher performance and lower power consumption at a bounded loss in quality the effect of relaxed notions of correctness in systems may add other non-functional properties to the list such as robustness as well as production and development cost. Yet systems engineering tolerating, beware, incorrect results has up until recently played a rather secluded role.

The discrete nature of mathematics relevant in digital systems is probably a promising factor to look at for an explanation. Clearly, constructing a digital artifact and then showing that it produces results good enough for general (as opposed to special) purpose rather than results that are simply right or wrong is difficult in the presence of discrete semantics. Just modelling systems that involve both discrete and continuous concepts and then argueing about their properties is already a challenge [6]. Yet we feel that the emergence of relaxed notions of correctness in systems engineering is a sign of a maturing field taking a turn with new potential for the current and next generation of computer scientists and engineers. And it is not just about improving robustness and reducing cost but also about being able to utilize the emerging generations of systems, some increasingly parallel, using hopefully less energy so that some of the limitations of traditional design may eventually be overcome.

The purpose of this paper is to provide a brief overview of state-of-the-art work in the field of which some is related to material presented during a special session on probabilistic embedded computing at DAC 2012 organized by the first author. We also describe the key ideas behind our own work on concurrent data structures related to the topic. The common theme is here to identify "metrics of correctness" in each example and discuss their properties and possibly ways to obtain quantities in them. Note that this is an extremely short list of work far from being complete or even representative. The material cited here should only be seen as a hopefully reasonable starting point for finding other related work not cited here.

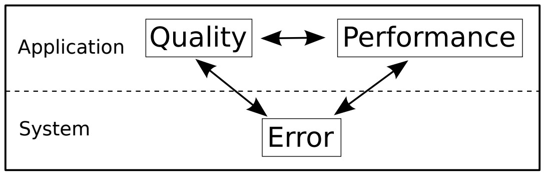

Figure 1: "Metrics of Correctness"

We work with three types of metrics for quantifying quality and performance of computation as well as error degree of systems whose design has been relaxed such that they may make mistakes, as shown in Figure 1. An error metric quantifies the degree of errors introduced by a relaxed system. A quality metric quantifies the computational quality produced by an application running on a relaxed system. We say that an application is error-tolerant on a relaxed system if the quality it produces increases whenever the system makes less errors, i.e., if the degree of errors in the computed results is proportional to the degree of errors introduced by the system. Relevant performance metrics are here execution time and power consumption. We say that an application is error-scalable if its performance increases whenever the system may make more mistakes.

2. HARDWARE

An important source of complexity in hardware design is reliability of computation, e.g. in arithmetic units and mechanisms such as hardware-based error detection and correction. Unreliable hardware design allows to reduce that complexity potentially providing benefits such as lower production cost, lower test and verification cost, smaller form factors, higher performance, and lower power consumption.

So-called probabilistic design as well as approximate design are two unreliable hardware design principles [11, 12] for trading-off reliability of computation and power consumption [16]. The idea of probabilistic design is to develop hardware that produces an output value for a given input value with a certain probability. Experimental data obtained on actual hardware shows that there is a monotone relationship between the probability of correct computation and power consumption [4], suggesting the probability of correct computation as quality metric. Approximate design results in hardware that deterministically produces, for a given input value, an output value that may, however, be incorrect. Similarly, experimental data obtained on actual hardware shows that there is a monotone relationship between arithmetic error and power consumption [5]. Here, the arithmetic error is an obvious candidate for a quality metric.

Unreliable hardware may also help increase hardware parallelism since unreliable hardware may require significantly less space than reliable hardware. An asymmetric multi-core architecture where a small number of reliable cores is combined with a large number of unreliable cores is an example of a design with a higher degree of parallelism than a conventional design of the same size [10]. Intuitively, the number of unreliable cores may be useful as error metric, and inversely even as quality metric if the quality of computations on unreliable cores deteriorates monotonically with the number of unreliable cores.

3. HARDWARE-SOFTWARE

The correctness of conventional software typically relies on hardware that returns deterministic output values for any given input values. Best-effort computing [3, 1] is a system design methodology for taking advantage of combinations of unreliable hardware and error-tolerant software to gain higher performance and lower power consumption. Errors introduced by unreliable hardware may be tolerated by certain types of software or handled by higher-level software layers. The challenge here is to divide applications into parts that tolerate errors and parts that do not. Such applications can also take advantage of the previously mentioned asymmetric multi-core architectures where application parts that tolerate errors run on unreliable cores [3, 10].

Stochastic processors [15, 19] produce so-called stochastically correct values, as with probabilistic design, through simplified hardware design, which may again enable higher performance and lower power consumption. Higher-level software layers may handle incorrect values either by tolerating the error or by detecting and correcting the error [21].

Control divergence in control-flow graphs makes branch prediction difficult and unreliable and may thus decrease performance. Branch herding performed by hardware or software reduces control divergence by forcing threads to take only a subset of all possible paths through the control-flow graph which may result in higher performance [21]. The degree of branch herding is an error metric whose inverse may also serve as quality metric if eliminating any branch always maintains or increases the error of the output. In this case, the application tolerates branch herding.

Detecting and correcting errors may incur high overhead which may eliminate the performance gains of stochastic processors. Therefore, it might be beneficial to detect just certain types or certain numbers of errors and correct them to stay within given error boundaries [21]. Here an error metric may be the number of errors that get detected and to which degree they get corrected. Again, the inverse may serve as quality metric if the actual error of the output is monotone in the error metric.

A key enabler of stochastic processors may be automatic transformation tools [19, 21] that generate error-tolerant versions out of regular applications, i.e., a transformed application may produce results at a quality that increases, at least within certain boundaries, whenever stochastic processors make less errors.

4. SOFTWARE

Relaxing software specifications may result in higher performance and lower power consumption, and may even increase reliability and robustness of software [18]. We distinguish relaxation techniques based on program transformations and concurrent data structure design.

4.1 Program Transformations

For certain types of applications, accuracy-aware program transformations [22] may generate error-tolerant code that may perform better than the original code. Here are three examples.

Substitution transformations [22] replace parts of a program with code that computes approximations of the output computed by the original parts but with less computational overhead. The approximate versions of the code are given

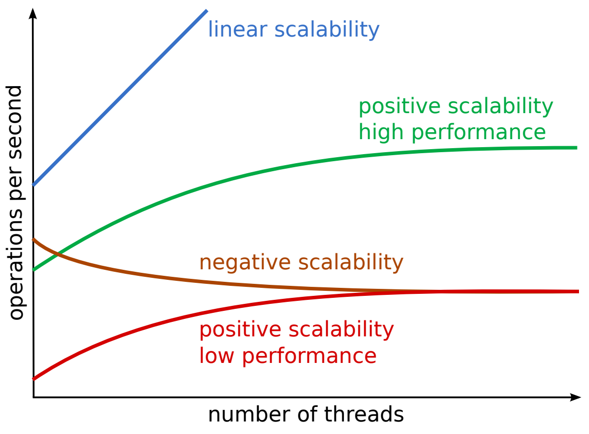

Figure 2: Concurrent data structure performance and scalability in number of data structure operations per second with an increasing number of threads sharing the same data structure in an exemplified benchmark scenario

and relaxed to different degrees in terms of some error metric. The program is error-tolerant if the approximations are compositional in terms of some quality metric.

Sampling transformations [22] work with code that computes output from a given set of elements. The transformed code performs the same computation as the original code but only on a subset of the elements obtained by some sampling policy. The code is error-tolerant if the quality of the result improves with larger subsets.

Loop perforation [20] is another type of program transformation which transforms a loop into a new loop where only a subset of the original loop iterations is performed. Decreasing the subset size of loop iterations decreases the runtime of executing the loop but also increases the error of the loop output.

4.2 Concurrent Data Structures

Concurrent data structures require synchronization to implement the exact specification of their sequential counterpart in a concurrent environment. However, synchronization operations may incur high overhead and prevent program code from executing in parallel. We discuss two approaches which may decrease synchronization overhead and increase parallelism, either by reducing contention on synchronization bottlenecks or by eliminating synchronization operations entirely, at the expense of adherence to exact data structure semantics. As a result, relaxed versions of concurrent data structures emerge which may perform and scale better on increasingly parallel hardware, and still be tolerated by certain applications.

In terms of performance and scalability, the goal is to achieve throughput in terms of number of data structure operations per second that is higher than of conventional designs (high performance) and grows with the number of concurrent units such as threads, for example, sharing the same data structure, to more threads than with conventional designs (positive scalability), as shown in Figure 2. Both, negative scalability even with high performance for low numbers of threads and positive scalability but with low performance are undesirable.

Reducing Contention on Synchronization Bottlenecks. Our own work is on relaxing the semantics of concurrent data structures by reducing contention on synchronization bottlenecks. We achieve that by relaxing the sequential specification of a concurrent data structure. Consider, for example, a regular first-in-first-out (FIFO) queue where elements are enqueued at the queue tail and dequeued at the queue head. The problem with this specification is that it leaves little room for optimization in concurrent implementations [13, 7] so that scalability in the presence of high contention on the queue may still be limited to relatively low numbers of threads [9].

Instead of maintaining the original specification of a FIFO queue, we propose to relax the specification to what we call a k-FIFO queue with $k \geq 0$, which may dequeue elements out of FIFO order up to k [8, 9]. Retrieving the oldest element from the queue may require up to $k+1$ dequeue operations (bounded lateness), which may return elements not younger than the $k+1$ oldest elements in the queue (bounded age) or nothing even if there are elements in the queue.

A k-FIFO queue is starvation-free for finite k where $k+1$ is what we call the worst-case semantical deviation (WCSD) of the queue from a regular FIFO queue. The WCSD bounds the actual semantical deviation (ASD) of a k-FIFO queue from a regular FIFO queue when applied to a given workload. Intuitively, the ASD keeps track of the number of dequeue operations necessary to return oldest elements and the age of dequeued elements.

Here, the error metric is semantical deviation: an implementation of a k-FIFO queue is correct if ASD $\leq$ WCSD for all workloads. Since semantical deviation is monotone, increasing k means more room for performance and scalability improvements. However, it is also important to consider which properties of an implementation determine k, i.e., whether k is configurable, depends on the workload, or is even probabilistic. For example, there are concurrent algorithms that implement k-FIFO queues whose WCSD is determined by configurable constants independent from any workload [2].

We took an entirely different approach called Scal queues based on distributed data structures and load balancing by creating p copies of a standard, non-blocking FIFO queue [13] and then, upon each queue operation, selecting one out of the p so-called partial queues for performing the actual operation, independently of and concurrently to any other operations that might hit the other $p-1$ partial queues [8, 9]. Thus the load balancing algorithm for selecting partial queues and p itself determine the WCSD of the resulting Scal queue. For example, selecting partial queues in a round-robin fashion for enqueueing, and independently for dequeueing, using two atomic indexes, limits the WCSD to p since the maximum imbalance of the partial queues cannot become larger than p. However, performance and scalability truly improves only if selection and actual partial queue operation are done non-atomically, increasing the WCSD to a workload-dependent p times the number of threads in the system [8, 9]. Even more performance and better scalability are possible if selection is done randomly (to avoid any explicit synchronization in selection [8, 9]) and hierarchically (to exploit memory hierarchies [9]). The resulting WCSD may then only be bounded probabilistically.

Another interesting aspect of semantical deviation is the problem of measuring it. Clearly, improved performance and scalability comes at the expense of increased semantical deviation, which may or may not be tolerated by applications using queues. But how bad is it? Well, ASD cannot be measured directly, at least as long as individual machine instructions cannot be time-stamped without introducing significant overhead. Instead, we obtain at runtime with low overhead so-called concurrent histories, which are sequences of time-stamped invocation and response events of the queue operations. A concurrent history represents the set of sequential histories—sequences of queue operations—that preserve precedence, i.e., if the response event of an operation A is before the invocation event of an operation B then A occurs before B in any of the sequential histories. One of these sequential histories actually took place with ASD as its semantical deviation. However, given a concurrent history, possibly containing millions of events, we are only able to compute offline the semantical deviation of one of the sequential histories with minimal semantical deviation, i.e., the lower bound on ASD, which nevertheless enables interesting relative comparisons of semantical deviation. In particular, we show that some Scal queues outperform and outscale existing implementations at the expense of moderately increased lower bounds on ASD [8, 9]. Computing upper bounds is more difficult and remains future work since it seems to involve enumerating possibly all precedence-preserving permutations of queue operations in a concurrent history.

Eliminating Synchronization Bottlenecks.

Synchronization bottlenecks can be eliminated by eliminating the corresponding synchronization operations. This approach may lead to race conditions of which some may result in effects such as data duplication or loss which may nevertheless still be tolerated by certain applications.

Idempotent work-stealing queues are distributed queues, one per thread, where a thread may either dequeue an element from its local queue without synchronization or dequeue (steal) an element from the queue of another thread with synchronization [14]. Queue elements may be returned multiple times instead of just once because of races between unsynchronized local dequeue and synchronized global steal operations. Intuitively, the number of races or, even more accurate, the amount of element duplication may be useful as error metric and inversely even as quality metric if the quality of computation deteriorates monotonically with element duplication. Moreover, error and quality metric may also be related to the number of involved threads since a larger number of threads increases the probability of races.

Another example of in fact full elimination of synchronization is a space-subdivision tree construction algorithm that does not use any synchronization operations and yet provides a well-defined and consistent tree state that may be good enough for some applications [17]. The race conditions that may occur result in subtree losses which reduces the amount of data held by the tree. Similar to the idempotent work-stealing queues, the number of races or, again even more accurate, the amount of subtree loss may serve as a useful error metric and inversely as a quality metric if the quality of computations deteriorates monotonically with subtree loss. Again, both error and quality metric may also be monotone in the number of involved threads since more threads make races more likely.

5. CONCLUSIONS

Relaxed and possibly quantitative notions of correctness are likely to play an increasingly important role in systems engineering. We believe that one of the key challenges is identifying "metrics of correctness" such that the effects of more errors gracefully, as opposed to abruptly, degrade the quality of a system and yet translate into higher performance and lower resource consumption. The idea seems to apply in virtually any area of systems engineering. Go for it!

6. REFERENCES

[1] Designing chips without guarantees. *IEEE Design and Test of Computers*, 27:60–67, 2010.

[2] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.

[3] S. Chakradhar and A. Raghunathan. Best-effort computing: re-thinking parallel software and hardware. In *Proc. Design Automation Conference (DAC)*, pages 865–870. ACM, 2010.

[4] L. Chakrapani, P. Korkmaz, B. Akgul, and K. Palem. Probabilistic system-on-a-chip architectures. *Transactions on Design Automation of Electronic Systems (TODAES)*, 12(3):29:1–29:28, May 2008.

[5] L. Chakrapani, K. Muntimadugu, A. Lingamneni, J. George, and K. Palem. Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation. In *Proc. Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 187–196. ACM, 2008.

[6] T. A. Henzinger. The theory of hybrid automata. In *Proc. Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE, 1996.

[7] D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.

[8] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 331–332. ACM, 2011.

[9] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. Technical Report 2011-03, Department of Computer Sciences, University of Salzburg, September 2011.

[10] L. Leem, C. Hyungmin, J. Bau, Q. Jacobson, and S. Mitra. Ersa: Error resilient system architecture for probabilistic applications. In *Proc.Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1560 –1565, 2010.

[11] A. Lingamneni, K. Muntimadugu, C. Enz, R. Karp, K. Palem, and C. Piguet. Algorithmic methodologies for ultra-efiňĄcient inexact architectures for sustaining technology scaling. In *Proc. Computing Frontiers (CF)*. ACM, 2012.

[12] A. Lingamneni and K. Palem. What to do about the end of Moore's law, probably! In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[13] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[14] M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.

[15] S. Narayanan, J. Sartori, R. Kumar, and D. Jones. Scalable stochastic processors. In *Proc. Conference on Design, Automation and Test in Europe (DATE)*, pages 335–338. European Design and Automation Association, 2010.

[16] K. Palem, L. Chakrapani, Z. Kedem, A. Lingamneni, and K. Muntimadugu. Sustaining moore's law in embedded computing through probabilistic and approximate design: Retrospects and prospects. In *Proc. Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 1–10. ACM, 2009.

[17] M. Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report 2012-03-005, MIT-CSAIL, February 2012.

[18] M. Rinard. Obtaining and reasoning about good enough software. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[19] J. Sartori, J. Sloan, and R. Kumar. Stochastic computing: embracing errors in architectureand design of processors and applications. In *Proc. Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 135–144. ACM, 2011.

[20] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. Symposium and Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 124–134. ACM, 2011.

[21] J. Sloan, J. Sartori, and R. Kumar. On software design for stochastic processors. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[22] Z. Zhu, S. Misailovic, J. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 441–454. ACM, 2012.