Challenging the Program Counter

W. Banzhaf[1]

Department of Computer Science, Memorial University of Newfoundland, St. John's
NL A1B 3X5, CANADA

Is it possible to achieve reliable results by running a machine with unreliable elements? This was a question that already John von Neumann was pondering when thinking about the brain and its performance[6]. Computers had, for many years, a problem of the same type. Elements of computing machinery would break, sometimes without being noticed by the programmer or operator, and only the results of a computation would have indicated that something strange had happened.

Computer engineers have, through various draconic measures, succeeded in clamping down on indeterminism in computing machinery, for instance through binary coding of all information held in physical devices, or through introduction of error-correcting codes for transmission of information. These and other measures, however, come at a cost in efficiency. In order to make sure that a deterministic order of programs is followed, for example, a program counter requests execution of one instruction at a time. Time is clocked, and movement of data is heavily constricted. Sometimes two or more cycles are needed just to move information around, energy needs to be spent to readjust electrical voltages to binary levels, and more data need to be transmitted in order to secure error correction.

In recent years, however, the specter of unreliability has come back: Neural networks have demonstrated that non-binary (if nonlinear) elements are useful in computing for certain functions like, e.g. pattern recognition. Quantum computing devices have been invented that work with probability bits, called qu-bits, instead of deterministic bits as traditional. Parallel computers have achieved such processor density that unreliability in elements becomes a major concern again. IBM's BlueGene project, for instance, has so many processors that at least once a day, a cosmic radiation event will succeed in flipping a bit. Where and when this happens is unknown, that it happens is a statistical exercise to calculate.

Notably the community in parallel and distributed computing has gone to great length in securing that parallel and distributed computers provide some sort of synchronicity between processes.

Here we argue that a particular sort of radical indeterminism can be injected into a computing machine without prohibiting it from computing useful quantities. We speak of the enforced deterministic sequentiality of computer code that might be dissolved this way. When von Neumann et al. put forward their proposal for a stored program computer, the invention contained actually two important pieces, one being to store programs as data. This development opened the way for a much more efficient method of programming than was used before,

and - at the same time - allowed for self-modifying code. The second aspect of the invention, however, frequently underestimated in its impact, was the program counter which would control execution of code residing in memory by providing the address of the next instruction in memory.

It is the behavior of the program counter that we want to challenge here, bringing it more in line in its behavior with execution of processes in the natural world. There, synchronicity is the effect of a highly intricate and complex construction, whereas at the lowest level of processes, things happen asynchronously. Is possible to have a system without deterministic execution control? In the framework of artificial chemistries [3], an area recently sprung up to study algorithms that model and simulate chemical reaction systems for various purposes, the answer is yes.

A program counter might randomly select from a multi–set of instructions, where a repetition of instructions is allowed to appear in memory locations. Computing would be understood as the transformation from input to output. This would be different from executing a prescribed sequence of computational steps. Instead, instructions from the multi–set $I = \{I_1, I_2, I_3, I_2, I_3, I_1, ...\}$ would be drawn in random order to produce a transformation result. In this way the sequential order usually associated with the notion of an algorithm would be dissolved. How can such an arrangement be able to produce useful results? Under the reign of a programming method that banks on its stochastic character, Genetic Programming [2], it turns out that results can still be useful.

A program in this sense is thus not a sequence of instructions but rather an assemblage of instructions that can be executed in arbitrary order. By randomly choosing one instruction at a time, the program proceeds through its transformations until a predetermined number of instructions has been executed. Different multi-sets can be considered different programs, whereas different passes through a multi–set can be considered different behavioral variants of a single program.

Because instructions are drawn randomly in the execution of the program, it is really the concentration of instructions that matters most. It is thus expected that "programming" of such a system requires the proper choice of concentrations of instructions, similar to what is required from the functioning of living cells, where at each given time many reactions happen simultaneously but without a need to synchronicity.

Systems of this kind substitute the rigidity of sequentiality with the flexiblity of pattern-based processing. In our particular example, the patterns are provided by input and output addresses of register/memory locations used to fetch and store results from the execution of instructions. This way, instructions are connected by data flow, not by sequence, and sequentiality is only introduced if appropriate patterns have been set up in the "programming" phase.

There are a number of advantages of systems like this.

1. They are nature analogues;
2. There is no need for synchronicity;
3. There is scalable accuracy, more computation results in more accurate results;

4. Due to the statistical nature of results, they are fault-tolerant;
5. Programs are modular, i.e. different programs can be developed and later put together to achieve an overall function;
6. Easily parallelizable.

Further details can be found in recent publications [1,5,4].

Acknowledgement

This work is supported by an NSERC discovery grant under RGPIN 283304-04.

References

1. Banzhaf, W., and Lasarczyk, C. W. G. Genetic programming of an algorithmic chemistry. In *Genetic Programming Theory and Practice II*, U.-M. O'Reilly, T. Yu, R. Riolo, and B. Worzel, Eds., vol. 8 of *Genetic Programming*. Kluwer/Springer, Boston MA, 2005, pp. 175–190.
2. Banzhaf, W., Nordin, P., Keller, R., and Francone, F. *Genetic Programming - An Introduction*. Morgan Kaufmann, San Francisco, CA, 1998.
3. Dittrich, P., Ziegler, J., and Banzhaf, W. Artificial Chemistries - A Review. *Artificial Life 7* (2001), 225–275.
4. Lasarczyk, C. W. G., and Banzhaf, W. Simulating infinite execution time on algorithmic chemistries. In *Proc. GECCO '05, Washington, DC, June 2005*, ACM Press. in press.
5. Lasarczyk, C. W. G., and Banzhaf, W. An algorithmic chemistry for genetic programming. In *Proc. 8th Europ. Conference on Genetic Programming, Lausanne, Switzerland, April 2005* (Berlin, 2005), Springer, LNCS 3447, pp. 1 — 12.
6. von Neumann, J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies* (Princeton, NJ, 1956), C. Shannon and J. McCarthy, Eds., Princeton Univ.Press.