

Notes on “A Defect-Tolerant Computer Architecture – Opportunities for Nanotechnology”

Citation:

- Authors:
 - James R. Heath, Philip J. Kuekes, Gregory R. Snider, R. Stanley Williams
- Reference:
 - Science, Vol. 280, June 12, 1998, p. 1716-1721.

General Introduction:

- Its anticipated that more defects will occur with devices with nanometer feature sizes and self-assembly.
 - **See Slides 2 and 3 in the accompanying Powerpoint.**
- In anticipation, HP wanted to see how this would affect high-end computation + systems-level issues; built Teramac prototype as a testbench.
 - **See Slide 4 – picture of Teramac**
- Machine has about 220,000 HW defects, but operated 100x faster than high-end single processor workstation – for some configurations.
- Purpose was to show that a computer that has (relatively speaking) many defects can still function well if there are enough devices, etc. to compensate.

Motivation – or in other words, “Why?”:

- Consider some relevant performance data points from the last 40 years.
 - 500 adds/J in 1971.
 - 3,000,000 adds/J in 1998
 - 1997 SIA Roadmap projected 10^9 in 2012.
 - Would result in a gain of 7 orders of magnitude in 40 years.
 - Low power adder design studies (for cell phones, etc.) project 9×10^{11} adds/J – with a delay of just 1-10 ns.
- Still, energy cost per physical add is nowhere near the physical limit – at limits of non-reversibility, 3×10^{18} adds/J appear possible.
 - **See board notes for reversibility discussion.**
- Therefore, potentially 8-9 orders of magnitude more to gain.
 - This goal drives much work in emerging technologies.
 - *Transistors have some fundamental limits – in both performance and cost – and this is motivation for the first 1.5 weeks of lecture.*

Purpose of the Teramac:

One of the consequences of self-assembly is that not all devices will be operational – statistical yields of chemical synthesis used to make them guarantees this.

- Net result is that we have to live with some uncertainty in our *systems*.
- Some questions to consider:
 - How does one communicate with the outside world in a predictable way and how is one assured that computation is reliable? – its possible to get into the right state for the wrong reason...
 - How does one impose an organizational scheme when dealing with a mole of devices??
 - Self assembly begs for at least semi-regular structures.
 - Chemically assembled machine (for example) still has to reproduce arbitrary complexity that GP computation demands.

In engineering, the answer to low but nonzero failure rates is to design *redundancy* into the system.

- Purpose of the Teramac was to see what would happen when these ideas were applied to computer chips – which usually are “defect free”.
- Teramac intentionally uses components that were cheap, but defective. Additionally, inexpensive, yet error-prone strategies were used to connect all of the components.
 - Defect tolerance defined to be “capability of a circuit to operate as desired without physically repairing or removing random mistakes incorporated into the system during the manufacturing process.”

Teramac at a high-level:

- Teramac name derived from the fact that machine operates at 10^6 Hz and has 10^6 logic elements – thus 10^{12} logical operations per second in ideal case.
- Some specifics:
 - The chips in Teramac are all reconfigurable FPGAs – 864 in all. Each has a large number of very simple computing elements and a flexible communication NW.
 - Each computing element performs a 6-input, 1-output logic function via LUTs – i.e. answers to logical function are stored in 64-bit LUTs. There are 65,536 LUTs in the machine.
 - * Net result = 4 Mb ($2^{16} \times 2^6$) of configuration memory used to define logic functions for all computing elements.
 - * **See board notes for a discussion - p. 1A and 1B**
 - 4 Mb of configuration memory are drawn from about 30% (256) of the FPGAs.
 - Bulk of FPGAs only used for signal routing – even though they have LUTs. Its cheaper just to make one part and ignore functionality that’s not needed.

Teramac is different from typical microprocessor where description of what chip should do is developed and then HW is constructed based on that logic.

- Here, we have a generic set of wires, switches, and gates and then we make them do what we want by configuring them “in the field.” (i.e. we compile to space, not time...)

- See crossbar slides – 5 and 6
- Teramac uses ILIW – Insanely Long Instruction Word – to configure a computation. ILIW is essentially 300 Mb – as we have to program the crossbar connections too.
 - As you might guess, this takes some time – but can happen rarely!

Teramac’s Design Goals:

Mapping a *specific* logical machine could easily be intractible – especially if there are large number of switches + LUTs but few, viable configurations...

- Problem is similar to traveling salesman problem...
- Teramac designed to provide satisfactory configurations for any desired design – may not find optimum configuration, but find something – and something close to optimum at that.
 - Early in design process, recongized the need for million-gate place and route ability
 - IC topologies compiler found easy to use had large number of wires on the chip – this drove the work in architecture.
 - Placement and routing time in FPGA much slower than for this CCM.
 - Points of referenece – while P & R times have decreased to minutes from hours, still much greater than the 3s times PLASMA chips used.
 - If off-the shelf FPGAs used in Teramac, days would have been needed. With PLASMA chips on Teramac, done in about 1 hour.
- 2 important concepts for making the above happen in Teramac – Fat Tree architectures and Rent’s Rule.
- **Board discussion of Fat-Tree and Rent’s Rule; see slide 7**

The Teramac – Low-Level:

- Detailed board discussion on layout and organization + Rent’s Rule numbers + configuration time.

Benchmarks Implemented:

- Was specifically designed to translate MRI data into a 3D map of arteries in the human brain.
- In another use, it was configued as a volume visualization engine.
- For DNA pattern matching, actually achieved 10^{12} gate operations per second.
- That said, first configuration loaded was that of a machine that could test itself!

Defect Tolerance:

- For Teramac, all “repair” work was done with software; a program was written to locate the mistakes and create a defect database for the compiler.
- Test process can be separated into: (a) running configurations that measure the state of the CCC, and (b) a set of algorithms that are run on these measurements to determine the defect.
 - LUTs were connected in a wide variety of configurations to determine if a resource (switch, wire, or LUT) were reliable or not.

- If any group failed, then other configurations that used the resources in question in combination with other devices were checked.
- Resources found in the intersection of the unreliable configurations were declared bad and logged.
- Actual testing performed by downloading designs onto Teramac called “signature generators” – sets of LUTs that generate long random number strings that are sent around Teramac by a large number of different physical paths.
 - **Project idea – CS components of programmability – very universal to many nanotechnologies...**
- If bit stream was correctly generated AND transmitted by the network, all resources are probably good – they were designed to diverge exponentially in time after an error in computation.
- Above procedure designed to find the physical defects such as opens, shorts, and stuck at 1 or 0 – much easier than finding a logic design error.

But how do you trust the testers?

- Resources that were part of “privileged” set were deliberately designed with explicit additional redundancy to ensure that they had a high probability of survival.

Lessons for Nanotechnology:

- As perfect devices become more expensive (or impossible for that matter) to fabricate, defect tolerance becomes a more valuable method to deal with the imperfections.
- Any computer with nanoscale components will contain a significant number of defects, as well as lots of wires for switches, communication, etc.
- It makes sense to consider architectural issues and defect tolerance early in the development of a new paradigm.

Specific lessons for nanotechnology:

1. Its possible to build a powerful computer with defective componets and wiring.
 - At present, such an approach is not economically competitive with CMOS technology, which requires perfection in all the components in a computer, because so many of the resources in a CC are not used.
2. Resources of a computer do not necessarily have to be regular, but they do need a high degree of connectivity.
 - Wiring mistakes in the MCMs introduced a significant element of randomness to the connectivity of systems that it was not possible to know what was connected without a test.
 - Thus, its NOT essential to place a component at a *specific* position, as long as it can be located *logically*.
3. The 3rd lesson considers what components are most important:
 - The most essential components for an electronic nanotechnology are wires (and are by far the most plentiful resource).
 - Next are the crossbar switches and the elements that control them. These are probably the most important active elements as these control how the system is wired up.

- LUTs make up less than 3% of the fat-tree utilizable resources of Teramac. As its scaled, it actually decreases in comparison to the hierarchy.
4. The fourth lesson of Teramac relates to the division of labor for building a computer.
- The conventional paradigm for computation is to design the computer, build it perfectly, and then run the algorithm.
 - The Teramac paradigm is to build the computer (however imperfectly), find the defects, configure the resources with software, compile the program, and then run it. (Moves what is hard to do in HW to a SW task...)

By examining an architecture that will be compatible with nanoscale fabrication, one may then be able to identify the types of components that are most crucial for implementing that architecture.