Software Geography: Physical and Economic Aspects

Vaughan R. Pratt

[1] Stanford University pratt@cs.stanford.edu
[2] Tiqit Computers pratt@tiqit.com

Abstract. To the metaphors of software engineering and software physics can be added that of software geography. We examine the physical and economic aspects of the Software Glacier (once an innocent bubbling brook, now a vast frozen mass of applications imperceptibly shaping both the Hardware Shelf below and User City above), Quantum Planet (colonization of which could be fruitful if and when it becomes practical), and Concurrency Frontier (an inaccessible land with rich resources that we project will be exploited to profound economic effect during the next half-century).

1 The Software Glacier

It used to be that the computer was the hard thing to build, programming it was almost an afterthought. Building a computer was hard because the problem was so constrained by the laws of nature. Writing software on the other hand was very easy, being unconstrained other than by the size and speed of available function units, memory, and storage.

But as time went by it became harder to write software and easier to build computers.

One might suppose this to be the result of the hardware constraints somehow loosening up while the software constraints tightened. However these basic constraints on hardware and software have not changed essentially in the past half century other than in degree. When hardware design was a vertical operation, with complete computers being designed and built by single companies, hardware designers had only the laws of nature to contend with. But as the economics of the market place gradually made it a more horizontal operation, with different specialists providing different components, the need for specifiable and documentable interoperability sharply increased the constraints on the hardware designers.

At the same time clock rates, memory capacity, and storage capacity have increased geometrically, doubling every eighteen or so months, with the upshot that today one can buy a gigabyte of DRAM and a 60 gigabyte hard drive for $150 each. This has enormously reduced the speed and storage constraints on software developers.

So on the basis of how the contraints have evolved, hardware should be harder than ever and software easier than ever.

Besides constraints, both tasks have become more complex. Hardware increases in complexity as a result of decreasing line geometries combining with increasing die size to permit more circuits to be packed into each chip. This has two effects on the complexity of the computer as a whole. On the one hand the additional per-chip functionality and capacity increases the overall computer complexity. On the other hand it also facilitates a greater degree of integration: more of the principal components of the computer can be packed into each chip, simplifying the overall system at the board level.

The complexity of software (but not its utility and effectiveness) increases linearly with each of time, number of programmers, and effectiveness of programming tools. There is some attrition as retiring technology obsoletes some packages and others receive major overhauls, but mostly software tends to accumulate.

So we have two effects at work here, constraints and complexity.

Hardware is complex at the chip level, but is getting less complex at the board level. At the same time hardware design is so constrained today as to make it relatively easy to design computers provided one has the necessary power tools. There are so few decisions to make nowadays: just pick the components you need from a manageably short list, and decide where to put them. The remaining design tasks are forced to a great extent by the design rules constraining device interconnection.

The combined effect of contraints and complexity on hardware then is for system design to get easier. Chip design remains more of a challenge, but even there the design task is greatly facilitated by tight design rules. Overall, hardware design is getting easier.

Software design on the other hand is by comparison hardly constrained at all. One might suppose interoperability to be a constraint, but to date interoperability has been paid only lip service. There is no rigid set of rules for interoperability comparable to the many design rules for hardware, and efforts to date to impose such rules on software, however well-intentioned, have done little to relieve the prevailing anarchy characterizing modern software.

The dominant constraint with software today is its sheer bulk. This is not what the asymptotics of the situation predict, with hardware growing exponentially and software accumulating linearly. However the quantity of modern software is the result of twenty years of software development, assuming we start the clock with the introduction of the personal computer dominating today's computing milieu. Multiply that by the thousands of programmers gainfully or otherwise employed to produce that software, then further multiply the result by the increase in effectiveness of programming tools, and the result is an extraordinary volume of software.

Software today has much in common with a glacier. It is so huge as to be beyond human control, even less so than with a glacier, which might at least be deflected with a suitable nuclear device—there is no nuclear device for software.

Software creeps steadily forward as programmers continue to add to its bulk. In that respect it differs from a glacier, which is propelled by forces of nature.

This is not to say however that the overall progress of software is under voluntary human control. In the small it may well be, but in the large there is no overall guiding force. Software evolves by freewill locally, but globally determinism prevails.

Unlike a glacier, software is designed for use. We can nevertheless continue the analogy by viewing software users as residents of a city, User City, built on the glacier. The users have no control over the speed, direction, or overall shape of the glacier but are simply carried along by it. They can however civilize the surface, laying in the user interface counterparts of roads, gardens, and foundations for buildings. The executive branch of this operation is shared between the software vendors, with the bulk currently concentrated in Microsoft. The research branch is today largely the bailiwick of SIGCHI, the Special Interest Group on Computer-Human Interfaces.

Hardware is to software as a valley is to the glacier on it. There is only one glacier per valley, consisting of all the software for the platform constituting that valley. In the case of Intel's Pentium and its x86 clones for example, much of the software comes from Microsoft, but there are other sources, most notably these days Linux.

A major breakdown in this analogy is that whereas mortals have even less control over the valley under a real glacier than over the glacier itself, the ease of designing modern hardware gives us considerable control of what the software glacier rides on. We are thus in the odd situation of being able to bring the valley to the glacier.

This is the principle on which David Ditzel founded Transmeta six years ago. When Ditzel and his advisor David Patterson worked out the original RISC (Reduced Instruction Set Computer) concept in the early 1980s, the thinking back then was that one would build a RISC machine which would supplant the extant CISC (Complex ditto) platforms.

The building was done, by Stanford spinoffs Sun Microsystems and MIPS among others, but not the supplanting. In a development that academic computer architects love to hate, Intel's 8008 architecture evolved through a series of CISC machines culminating in the Pentium, a wildly successful machine with many RISC features that nevertheless was unable to realize the most important benefit of RISC, namely design simplicity, due to the retention of its CISC origins.

Ditzel's idea, embarked on in 1995, was to build a Pentium from scratch, seemingly in the intellectual-property shelter of IBM's patent cross-licensing arrangements with Intel. The CISC soul of this new machine was to be realized entirely in software. Transmeta's Crusoe is a pure RISC processor that realizes the complexity benefits of RISC yet is still able to run the Pentium's heavily CISC instruction set.

Whether the benefits of a pure RISC design are sufficient to meet the challenges of competing head-on with Intel remains to be seen. (The stock market seems to be having second thoughts on Transmeta's prospects, with Transmeta's stock currently trading at $2.50, down from $50 last November.) The point to be noted here is that Ditzel saw the software benefits of going to the glacier. Just as bank robber Willie Sutton knew where the money was, Ditzel could see where the software was. That insight is independent of whether pure RISC is a sound risk.

The question then naturally arises, must we henceforth go to the glacier, or is it not yet too late to start a new software glacier in a different valley?

Apropos of this question, a new class of processors is emerging to compete with the Pentium. The Pentium's high power requirements make it a good fit for desktops and large-screen notebooks where the backlighting power is commensurate with the CPU power and there is room for a two-hour battery. However the much smaller cell phones and personal digital assistants (PDAs) have room for only a tiny battery, ruling out the Pentium as a practical CPU choice. In its place several low-powered CPUs have emerged, notably Motorola's MC68328 (Dragonball) as used in the Palm Pilot, the Handspring Visor, and the Sony CLIÉ; the MIPS line of embedded processors, made by MIPS Technologies, Philips, NEC, and Toshiba, as used in several brands of HandheldPC and PalmPC including the Casio Cassiopeia; the Hitachi SH3 and SH4 as used in the Compaq Aero 8000 and the Hitachi HPW-600 (and the Sega Dreamcast); and Intel's Strongarm.

Software for these is being done essentially from scratch. Although Unix (SGI Irix) runs on MIPS, it is too much of a resource hog to be considered seriously for today's lightweight PDAs. The most successful PDA operating system has been PalmOS for the Palm Pilot. However Microsoft's Windows CE, lately dubbed Pocket PC, has lately been maturing much faster than PalmOS, and runs on most of today's PDAs.

Linux has been ported to the same set of platforms on which Windows CE runs. The difference between Linux and Windows here is that, whereas Windows CE is a new OS from Microsoft, a "Mini-ME" as it were, Linux is Linux. The large gap between Windows CE and Windows XP has no counterpart with Linux, whose only limitations on small platforms are those imposed by the limited resources of small handheld devices.

This uniformity among PDA platforms gives them much of the feel of a single valley. While there is no binary compatibility between them, this is largely transparent to the users, who perceive a single glacier running through a single valley.

It is however a small glacier. While Linux is a rapidly growing phenomenon it has not yet caught up with Microsoft in sheer volume of x86 software, whence any port of Linux to another valley such as the valley of the PDAs lacks the impressive volume of the software glacier riding the x86 valley.

The same is true of Windows CE. Even with Pocket PC 2002 to be released a month from this writing, the body of software available for the Windows CE platform is still miniscule compared to that for the x86.

Why not simply port the world's x86 software to PDA valley?

The problem is with the question-begging "simply." It is not simple to port software much of which evolved from a more primitive time. (Fortunately for Linux users, essentially the whole of Linux evolved under relatively enlightened circumstances, greatly facilitating its reasonably successful port to PDA valley.) It would take years to identify and calm down all the new bugs such a port would introduce. Furthermore it wouldn't fit into the limited confines of a PDA.

Now these confines are those of a 1996 desktop, and certainly there was already a large glacier of software for the x86 which fitted comfortably into those parameters back then. Why not just port that software?

The problem with this scenario is that software has gone places in the interim, greatly encouraged and even stimulated by the rapidly expanding speed and storage capacity of modern desktops and notebooks. To port 1996 software to PDA valley would entail rolling back not just the excesses of modern software but also half a decade of bug fixes and new features. A glacier cannot be selectively torn into its good and bad halves. With the glitzy new accessories of today's software comes geologically recent porcine fat that has been allowed to develop unchecked, along with geologically older relics of the software of two or more decades ago. Liposuction is not an option with today's software: the fat is frozen into the glacier.

Nowhere has this descent into dissipation been more visible than with the evolution of Windows ME and 2000 from their respective roots in Windows 3.1 and NT. (The emphasis is on "visible" here: this scenario has played out elsewhere, just not as visibly.) Whereas 16 MB was adequate for RAM in 1994, now 128 MB is the recommended level. Furthermore the time to boot up and power down has increased markedly.

The one PDA resource that is not currently in short supply, at least for PDAs with a PCMCIA slot such as HP-Compaq's iPAQ, is the hard drive. Toshiba and Kingston have been selling a 2 GB Type II (5 mm thick) PCMCIA hard drive for several months, and Toshiba has just announced a 5 GB version. Here the exponential growth of storage capacity has drawn well clear of the linear production of the world's software, which stands no chance now of ever catching up (except perhaps for those very few individuals with a need and budget for ten or more major desktop applications on a single PDA). However space on a PDA for all one's vacation movies, or Kmart's complete assets and accounts receivable database, will remain tight for the next two to four years.

Windows XP, aka NT 6.0, is giving signs of greater sensitivity to these concerns. XP Embedded is not Windows CE but rather XP stripped for action in tight quarters. And Microsoft distributes an impressive library of advice on shortening XP boot time under its OnNow initiative. Unfortunately the applications a power XP user is likely to run are unlikely to show as much sensitivity and shed their recently acquired layers of fat in the near future.

With these considerations in mind, the prospects for Windows CE/Pocket PC look very good for 2002, and perhaps even 2003. But the exponential growth of hardware is not going to stop then. We leave it to the reader to speculate on the likely PDA hardware and software picture in 2004.

2 Quantum Planet

2.1 Quantum Computation

Quantum computation (QC) is a very important and currently hot theoretical computer science topic.

The practical significance of QC is less clear, due to a gap of approximately two orders of magnitude between the largest quantum module of any kind we can build today and the smallest error-correcting modules currently known from which arbitrarily large quantum computers can be easily manufactured. This gap is presently closing so slowly that is impossible to predict today whether technology improvements will accelerate the closing, or unanticipated obstacles will emerge to slow it down or even stop it altogether on new fundamental grounds that will earn some physicist, quite possibly even one not yet born, a Nobel prize. If this gap turns out to be unclosable for any reason, fundamental or technological, QC as currently envisaged will remain a theoretical study of little more practical importance than recursion theory.

The prospect of effective QC is as remote as manned flight to distant planets. The appropriate metaphorical location for QC then is not a city or even a distant country but another planet altogether, suggesting the name Quantum Planet.

On the upper-bound side, there are depressingly few results in theoretical QC that have any real significance at all. The most notable of these is P. Shor's extraordinary quantum polynomial-time factoring algorithm and its implications for the security of number-theoretic cryptography [15].

There has been some hope for quantum polynomial-time solutions to all problems in NP. One approach to testing membership in an NP-complete set is via an algorithm that converts any classical (deterministic or probabilistic) algorithm for testing membership in *any* set into a faster quantum algorithm for membership in that set. Grover has given a quadratic quantum speedup for any such classical algorithm [8].

On the lower-bound side, several people have shown that Grover's speedup is optimal. But all that shows about membership in an NP-complete set is that a good quantum algorithm based on such an approach has to capitalize somehow on the fact that the set is in NP. Any method that works for the more general class of sets treatable by Grover's algorithm cannot solve this problem on its own.

We would all like to see QC turn out to be a major planet. For now it is turning out to be just a minor asteroid. I see this as due to the great difficulty people have been experiencing in getting other good QC results to compare with Shor's.

2.2 Quantum Engineering

There is a saying, be careful what you wish for, you may get it. While computer scientists are eagerly pursuing quantum computation with its promise of exponentially faster factoring, computer engineers are nervously anticipating a different kind of impact of quantum mechanics on computation. At the current rate of shrinkage, transistors can expect to reach atomic scale some time towards the end of this century's second decade, i.e. before 2020 AD. (So with that timetable, as Quantum planets go, Quantum Engineering is Mars to Quantum Computation's Neptune.)

As that scale is approached the assumptions of classical mechanics and classical electromagnetism start to break down. Those assumptions depend on the law of large numbers, which serves as a sort of information-theoretic shock-absorber smoothing out the bumps of the quantum world. When each bit is encoded in the state of a single electron, the behavior of that bit turns from classical to quantum. Unlike their stable larger cousins, small bits are both fickle and inscrutable.

Fickle. In the large, charge can leak off gradually but it does not change dramatically (unless zapped by an energetic cosmic ray). In the small however, electrons are fickle: an electron can change state dramatically with no external encouragement. The random ticks of a Geiger counter, and the mechanism by which a charged particle can tunnel through what classically would have been an impenetrable electrostatic shield, are among the better known instances of this random behavior.

Inscrutable. Large devices behave reasonably under observation. A measurement may perturb the state of the device, but the information gleaned from the measurement can then be used to restore the device to prior state. For small devices this situation paradoxically reverses itself. Immediately after observing the state of a sufficiently small device, one can say with confidence that it is currently in the state it was observed to be in. What one cannot guarantee however is that the device was in that state *before* the measurement. The annoying thing is that the measurement process causes the device to first change state at random (albeit with a known contingent probability) and then to report not the old state but the new! The old state is thereby lost and cannot be reconstructed with any reasonable reliability. So while we have reliable reporting of current state, we do not have reliable memory of prior state.

So while quantum computation focuses on the opportunities presented by the strangeness of the quantum world, the focus of quantum engineering is on its challenges. Whereas quantum computation promises an exponential speedup for a few problems, quantum engineering promises to undermine the reliability of computation.

Quantum engineering *per se* is by no means novel to electrical engineers. Various quantum effects have been taken advantage of over the years, such as Goto's tunnel diode [6], the Josephson junction [9], and the quantum version of the Hall effect [16]. However these devices achieve their reliability via the law of large numbers, as with classical devices, while making quantum mechanics work to the engineer's advantage. Increasing bit density to the point where there are as many bits as particles makes quantum mechanics the adversary, and as such very much a novelty for electrical engineers.

3 Concurrency Frontier

Parallel computation is alive and well in cyberspace. With hundreds of millions of computers already on the Internet and millions more joining them every day, any two of which can communicate with each other, it is clear that cyberspace is already a highly parallel universe.

It is however not a civilized universe. Concurrency has simply emerged as a force of cybernature to be reckoned with. In that sense concurrency remains very much a frontier territory, waiting to be brought under control so that it can be more efficiently exploited.

Cyberspace is still very much a MIMD world, Multiple Instructions operating on Multiple Data elements. It is clearly not a SIMD world, the science fiction scenario of a single central intelligent agent controlling an army of distributed agents.

Nevertheless some SIMD elements are starting to emerge, in which the Internet's computers act in concert in response to some stimulus. The Y2K threat was supposed to be one of these, with all the computers in a given time zone misbehaving at the instant the year rolled over to 2000 in that time zone.

The fact that most computers today in the size spectrum between notebooks and desktops run Microsoft Windows has facilitated the development of computer viruses and worms. A particularly virulent worm is the recent W32.Sircam, which most computer users will by now have received, probably many times, as a message beginning "I send you this file in order to have your advice." If it takes up residence on a host, this worm goes into virus mode under various circumstances, one of which is the host's date being October 16. In this mode it deletes the C: drive and/or fills available space by indefinitely growing a file in the Recycled directory. As of this writing (September) it remains to be seen whether this instance of synchronicity will trigger a larger cyberquake than the mere occurrence of the year 2000.

More constructive applications of the SIMD principle have yet to make any substantial impact on the computing milieu. However as the shrinking of device geometries continues on down to the above-mentioned quantum level and sequential computation becomes much harder to speed up, computer architects will find themselves increasing the priority of massively parallel computation, not for just a handful of CPUs but for thousands and even millions of devices acting in concert.

Thinking Machines Corporation's CM-2 computer was an ambitious SIMD machine with up to 64K processors. Going by processor count, this is impressive even today: then it yielded a local bus bandwidth of 40 GB/s, and with today's faster buses could be expected to move terabytes per second locally. However main memory was 0.5 GB while hard drive storage ran to 10 GB, capacity that today can be matched by any hobbyist for $150. The blazing speed was definitely the thing.

SIMD shows up on a smaller scale in Intel's 64-bit-parallel MMX architecture and 128-bit SSE architecture, as well as in AMD's 128-bit 3DNow architecture.

However this degree of parallelism yields only small incremental gains sufficient to edge ahead of the competition in the ongoing speed wars.

As some of the more fundamental bottlenecks start showing up, evasive maneuvers will become more necessary. A return to CM-2 scale parallelism and higher is certainly one approach.

A major difficulty encountered at Thinking Machines was how to describe the coordinated movement and transformation of data in such highly parallel machines.

Rose and Steele [14] and Blelloch and Greiner [1] came up with some novel programming language approaches to Thinking Machines' problem. What struck me about these approaches was how hard it is to separate ourselves from the sequential modes of thought that pervade our perception of the universe. It is as though we cannot *see* pure concurrency. If we were ever confronted with it we simply would not recognize it as computation, since it would lack those sequential characteristics that characterize certain behaviors for us as essentially computational, in particular events laid out along a time line.

Programmers who would like to leave a useful legacy to their heirs need to spend more time reflecting on the nature of concurrency, learning what it feels like and how to control it. My own view of concurrency is that event structures [10,17,18] offer a good balance of abstractness and comprehensiveness in modeling concurrency.

The extension of event structures to Chu spaces, or couples as I have started calling them [12], simultaneously enriches the comprehensiveness while cleaning up the model to the point where it matches up to to Girard's linear logic [4]. The match-up is uncannily accurate [2] given that linear logic was not intended at all as a process algebra but as a structuring of Gentzen-style proof theory.

There is another ostensibly altogether different approach to the essence of concurrency, that of higher dimensional automata [11,5,3,7]. It is however possible to reconcile this approach with the Chu space approach by working with couples over 3, that is, a three-letter alphabet for the basic event states of *before*, *during*, and *after* [13]. It is my belief that the three-way combination of duality-based couples, geometry-based higher-dimensional automata, and logic-based linear logic, provides a mathematically richer yet simpler view of concurrency than any other approach.

The economic promise of Concurrency Frontier is its potential for further extending the power of computers when the limits set by the speed of light and quantum uncertainty bring the development of sequential computation to a halt. I do not believe that the existing sequentiality-based views of concurrency permit this extension, and that instead we need to start understanding the interaction of complex systems in terms of the sorts of operations physicists use to combine Hilbert spaces, in particular tensor product and direct sum. The counterparts of these operations for concurrency are respectively orthocurrence $A \otimes B$ and concurrence $A \| B$ (also $A \oplus B$), or *tensor* and *plus* as they are called in linear logic. Closer study of this point of view will be well rewarded.

References

1. G. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Proc. Symposium on Functional Programming and Computer Architecture*, pages 226–237, June 1995.
2. H. Devarajan, D. Hughes, G. Plotkin, and V. Pratt. Full completeness of the multiplicative linear logic of Chu spaces. In *Proc. 14th Annual IEEE Symp. on Logic in Computer Science*, pages 234–243, Trento, Italy, July 1999.
3. L. Fajstrup, E. Goubault, and M. Raussen. Detecting deadlocks in concurrent systems. In *Proc. of CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 332–347. Springer-Verlag, 1998.
4. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
5. E. Goubault and T.P. Jensen. Homology of higher dimensional automata. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 254–268, Stonybrook, New York, August 1992. Springer-Verlag.
6. E. Goto *et al.* Esaki diode high speed logical circuits. *IRE Trans. Elec. Comp.*, EC-9:25–29, 1960.
7. E. (editor) Goubault. Geometry and concurrency. *Mathematical Structures in Computer Science, special issue*, 10(4):409–573 (7 papers), August 2000.
8. L. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical Review Letters*, 79(2), 1997.
9. B. Josephson. Possible new effects in superconductive tunnelling. *Physics Letters*, 1:251–253, 1962.
10. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures, and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
11. V.R. Pratt. Modeling concurrency with geometry. In *Proc. 18th Ann. ACM Symposium on Principles of Programming Languages*, pages 311–322, January 1991.
12. V.R. Pratt. Chu spaces and their interpretation as concurrent objects. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 392–405. Springer-Verlag, 1995.
13. V.R. Pratt. Higher dimensional automata revisited. *Math. Structures in Comp. Sci.*, 10:525–548, 2000.
14. J. Rose and G. Steele. C*: An extended c language for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, volume 2, pages 2–16, May 1987.
15. P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Computing*, 26:1484–1509, 1997.
16. K. von Klitzing, G. Dorda, and M. Pepper. New method for high-accuracy determination of the fine-structure constant based on quantized hall resistance. *Physical Review Letters*, 45(6):494–497, 1980.
17. G. Winskel. *Events in Computation.* PhD thesis, Dept. of Computer Science, University of Edinburgh, 1980.
18. G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, REX'88*, volume 354 of *Lecture Notes in Computer Science*, Noordwijkerhout, June 1988. Springer-Verlag.