The Tyranny of Transistors: What Counts about Software?

Position paper for EDSER-4, Economics-Driven Software Engineering Research, May 2002

Mary Shaw

Institute for Software Research, International
Carnegie Mellon University
Pittsburgh, PA 15213 USA
+ 1 412 268 2589

Mary.Shaw @cs.cmu.edu

ABSTRACT

Our intuitions tell us that computer software is vastly better than it was 20 years ago -- more diverse, more reliable, more powerful. However, the aggregate measures we use for software show only slow progress. Meanwhile, computer hardware basks in the glow of Moore's Law, and software seems to pale in comparison. While it is possible that software power does grows sluggishly in comparison to hardware, it seems more likely that hardware reaps the benefits of a better decision about what to count in describing progress over time. As we pursue value propositions in software, we should consider carefully the values that we pursue. A good start would be to switch from counting measures of our input effort to counting measures of the results of this effort

1. MOTIVATION

Computer hardware has a reputation for performance and cost improvement at a rate unprecedented in the history of technology, with selected measures doubling every year and a half or two. The fact that the benefit can be reaped in either price of performance adds to the luster.

Software, on the other hand, doesn't have a widely-quoted measure that shows how its power and value have increased over the years. Absent such a measure, we avoid the question and regale ourselves -- and the world -- with tales of cost and time overruns and other project failures. Absent a good measure, we sometimes use inappropriate measures -- such as the long-standing perception that productivity increases at rates of only 4-6%, leading to doubling times measured in decades instead of years.

The consequence is that software suffers relative to hardware, standing as the ne'er-do-well, if raffish, cousin to hardware's industrious and respectable leading man in the saga of information technology. Like the ne'er-do-well cousin, software doesn't get -- or effectively claim -- credit for its very real progress.

Through its interest in Economics-Driven Software Engineering Research, the EDSER community proposes to change the objective of software development from the creation of functionality to the creation of value. It therefore behooves us to consider carefully what we mean by value.

2. THE MOORE'S LAW PROBLEM

In 1965, Moore extrapolated from four data points that the number of transistors (switches) on an integrated circuit (chip) would double every two years. In fact, the doubling rate has been about 18 months for the subsequent three and a half decades. Furthermore, "Moore's Law" is widely interpreted not as a technical measure of switches per chip but as a broader, softer measure of computing system power that includes speed, storage capacity, and bandwidth, achieved through architecture and aggregation as well as chip technology.

Costs have not grown in proportion the cost of the power growth is largely in design, which is principally the cost of the designers. Human costs grow roughly with inflation, doubling every decade or two rather than every year or two. Manufacturing costs have also seen only modest increases, but that bears on the number of computers in use, not the power of each one.

So hardware has cultivated and earned a public persona of relentless growth in both power and price performance.

The genius of Moore's Law as an aggregate description of hardware improvement is that it

- counts something simple and understandable
- counts the same thing over time
- counts an output rather than an input
- refuses to get embroiled in all the deficiencies of transistors as a measure

Moore's Law counts transistors, which are simple and discrete. Even though most people haven't seen an actual transistor (the round solid-state device with three wires) in years, the transistor is still the common currency, interpreted as switching elements on a chip.

Moore's Law does not count the number of transistors laid down by a designer in a day, or even the number of design elements. The informal extension of Moore's Law to whole computing systems follows suit, taking processor clock time and data rates as the bases of comparison.

These measures are comparable over time. They are crude aggregates that lump many dissimilar things together and gloss over many important details. But they are all useful as gross measures of progress

3. WHAT COUNTS IN SOFTWARE?

Most of the measures near and dear to the hearts of software engineers count inputs or defects.

Notwithstanding its well-known shortcoming, the venerable Line of Code persists as a productivity measure. The classical growth of this measure is 4-6% per year, a doubling time of about a decade and a half. Boehm and Basili [3] note that it has been stuck at 10 lines per person per day for years. This is unsurprising, as it is a measure of what a human can produce. It hides the very real progress in software abstractions, which contribute by providing more powerful design elements for those lines of code to represent -- growing perhaps an order of magnitude in the 20 years [1] that it takes lines of code per person to double. At the same time, the total body of code in service to the US Department of Defense grew by about three orders of magnitude in those same 20 years while the cost per line falls almost as steeply [2]. This figure, however, counts all the instances of all the lines of code. Addressing capability and cost more directly, the cost of "equivalent" software was decreasing by about 16% per year in the spreadsheet market around 1990 [4].

Much energy has been invested in understanding the difficulties of software productivity measures ([6], for example). Certainly the appropriate detailed measures needed for project management and cost estimation depend on the character of the project. Most all observers agree that the order-of-magnitude skill differences between programmers dominate most of the details. A more serious problem with these measures is that they measure human inputs to the design, not the information processing capacity that results from the design.

At the other end of the spectrum, the economics community measures the business value of information technology [5]. For them, "productivity" is productivity of the enterprise -- which has been surprisingly low under standard measures. They also count bottom-line profit and consumer welfare. These measures certainly examine output, but they include many factors in addition to software. The picture here is confounded by the so-called "IT productivity paradox": much of the realized value is appropriated by consumers, yielding only competitive position rather than revenue to the business and defeating measures that consider only input and output. This confound is also a problem for software engineering measures.

Software engineering needs measures of progress that can express the very real growth of capability that has been contributed by software.

Software engineers seek well-founded models that allow us to reason from first principles. Sometimes, though, we wear the blinders of that quest. This is such a time. It is as if we are trying to understand the properties of gasses by working from the 2-body problem to the 3-body problem and planning to reach the gas laws by working our way up to the Avogadro's-number-body problem. Rather than taking the $6x10^{23}$ steps required to do this, we need to find also our Boyles, Charles, and Dalton to lead us to PV=nRT. In that same way, we need ways to reason about the aggregate properties of software systems as well as the detailed properties of individual lines of code.

4. WHAT SHOULD WE COUNT?

If the EDSER community is to succeed at making software engineering a value-based discipline, we need to sharpen our understanding of how software contributes value to its customer. We need to be able to measure something higher-level than the design elements produced by software developers [3][6] but lower-level than the value of information technology to the enterprise[5].

We need measures that document, in whatever way, the capability (and hence the change in capability over time) contributed by better engineering of software technology.

This may ultimately be reflected in perceived utility, in quality, in profit, in increased intellectual leverage on problems. It may vary from one market segment to another (some possibilities are suggested by Figure 2 of [3]). It may show up differently for different aspects of "quality".

For a start, though, we should take our cue from the success of Moore's Law and look for a measure of raw technical progress in software that

- counts something simple and understandable
- counts the same thing over time
- counts an output rather than an input
- refuses to get embroiled in all the deficiencies of transistors as a measure

This approach may yield a broadly-useful, if crude, measure long before we would have even a structure for all the distinctions implied by better, more specific measures.

To start the discussion, here are some possibilities based on delivered code size as a surrogate for capability (this seems no worse an approximation than the transistor):

- Bits of code running on a typical machine. (normalized to bits because word size has changed over time; I grew up on a machine with 56-bit instructions).
- Average size of common operating systems (a measure of the capacity of a computer as delivered)
- Average size of common application suites (a measure of user-level capacity)
- Average cost/kilobit of code, at retail (moving to a price-performance measure)
- Average cost/kilobit of code produced by a system integrator or value-added reseller

Note that I do not propose to measure total lines of code in service ([2], [3] p.32). Hardware isn't usually measured by transistors in service, either, except for network size.

5. CONCLUSION

The EDSER community pursues the objective of cost-effectiveness in design decisions and development techniques. We should consider carefully what the appropriate measures are, so that we can optimize the right functions.

More generally, software engineering has yet to find an aggregate measure of power that provides the same standard of comparison as transistors, clock speeds, and bandwidth. Instead of simply lamenting that much of software is unquantifiable, we should seek a measure that, however imperfect, allows us to track progress.

None of this diminishes the remaining difficulties of producing high-quality software that meets its expectations. It should, however, put those problems in a more realistic context.

6. ACKNOWLEDGMENTS

This position has emerged through many years of frustration. George Heilmeier and Lee Osterweil have encouraged me in the notion that we should count some simple measure of software progress that shows off our accomplishments. Chris Kemerer pointed out the problem of the IT productivity paradox in interpreting economic measures. This work has been supported by the Sloan Software Industry Center at Carnegie Mellon and by the National Science Foundation under Grant CCR-0086003.

7. REFERENCES

[1] L. Bernstein. Software investment strategies. Bell Labs Technical Journal, summer 1997, pp. 233-242. Cited in [3].

[2] Barry Boehm. "Lines of Code in Service: U.S Dept. of Defense", *Gaining Intellectual Control of Software Development*, http://sunset.usc.edu/Activities/aug24-25/NSFtalks/Boehm.ppt, chart 20, 1999

[3] Barry Boehm and Vic Basili. Gaining intellectual control of software development. *IEEE Computer*, May 2000, pp. 27-33.

[4] Erik Brynjolfsson and Chris F. Kemerer. Network externalities in microcomputer software: an econometric analysis of the spreadsheet market. *Management Science*, December 1996.

[5] Lorin Hitt and Erik Brynjolfsson. Productivity, profit and consumer welfare: three different measures of information technology's value. *MIS Quarterly*, June 1996.

[6] Walt Scacchhi. Understanding software productivity. *Advances in Software Engineering and Knowledge Engineering* (D. Hurley, ed), vol 4, 1995, pp. 37-70.