1 Summary

Spending Moore's Dividend by James Larus (2009): Larus described that up to that point in time how decades of Moore's prediction, the improvements and doubling of transistors on chips every few years, had led to similar improvements in processor speeds, computer design, and microarchitectures; he called these Moore's Dividends. How Moore's Dividends were spent, he noted, were not previously quantified, but could be observed in the trends and behaviors of the software, namely, the increasing growth in resource requirements and computational capabilities expected of the computer, and the evolution of software practices. While these were not backed by evidence, they lined up with what is seen in Myhrvold's Laws and the cycle of computer industry innovation, which can be briefly summarized that software expands to fully (and perhaps overly) utilizes the resources offered by the hardware they run on, which in turn pushes the demand and pulls the market towards new and improved hardware, and and so on; essentially, for decades, both industries have in some way pushed the other to grow to meet Moore's Law. Larus used these observations to predict what impact they'd have on future software practices and parallel computing; his point being, that Moore's Dividends drove the direction and growth of software in the last few decades, but the technology of multicore processors and parallel computing will be what shapes future software practices.

2 Strengths

- The numbers and figures in the introduction and the examples/anecdotes in the body painted a very clear picture of what Larus meant by "how Moore's Dividend were spent", i.e. what aspects of technology changed and in what way as a result of the improvements seen by Moore's Law. Likewise for the "Future Implications" section.

3 Weaknesses

- The article was long-winded in building up to the final point, in particular, I don't think it needed as much data as it had to set up the point in the introduction that Moore's Law was observed in real computers over several decades.
- Some of the figures included were stylized from their original source, but in a way that was confusing to read without hunting for the context in the article (missing axis labels and confusing captions).

4 Rating: 3

5 Comments

A lot of the topics Larus brings up are familiar as to what a student would have been exposed to so far throughout their education and pieces them together nicely, which I think is important for building context with the more advance material we go on to learn; for example, when he described how software practices were shaped (or enabled) by the improved processor technology, many ring true to what is seen in industry and schools today (object-oriented design, modularity, abstraction, richer libraries, emphasis on security, reliability, compatibility, prevalence of higher level languages, compilers and performance tuning, etc.). I believe he was correct in his own observation that certain trajectories of software development wouldn't change very much and continue to grow in their resourece demands as software continues to become more complex. While the article sets a nice stage of the state of technology at the time with plenty of examples, it is also over 10 years old. More can now be expanded on his observation on how multicore processors and parallel computing would be the next "revolution" to change software in the way integrated circuits and transistors had in the past. In particular, the other article "We're not prepared for the end of Moore's Law" (David Rotman, 2020) linked for this week's reading made a few similar points to Larus, such as how software still does not do enough to fully utilize the architeture of multicore and parallelism because programmers don't focus on efficient code in favor or relying on the compilers optimizing and processors getting more powerful. More than that, other means of innovation besides parallel programming have garnered more interest since Larus' article. One that Rotman notes is the growth of specialized architecture and advance software that works with them, which was mentioned in the 2017 Turing Lecture (Hennessy and Patterson) and that Larus does not touch on, nor did Larus mention the idea of the "decline of computers as a general purpose technology". That's not to say parallel programming isn't increasingly important because multicore processors are the norm; but rather, its not certain what the next "successor technology" or what the future landscape of computing would look like if Moore's Law trends to a halt. Parallel programming also may not have reached the prevalence Larus had hoped it would, but $\sim$15 years may not have been enough time to make that determination. However, I think the 2009 article is still less relevant 10 years after it was published and there are more current ones that could cover what Larus meant to cover and to include more recent developments in technology trends.

6 Notes

- Larus starts the paper off with an overview of Moore's Law (number of transistors on a chip would double roughly every two years, and clock rates and performance of computers would similarly improve), backed by historical data of processor development and performance (mainly of Intel's 8086), i.e. number of transistors, clock rates, and SPECint benchmarks. I'm not sure if it was the intention for this article, as it is a sweeping overview, but the figures he used from external sources were stylized to conform to the appearance of the article, so a few figures were missing something like

axis labels or adequate captions, and you had to hunt in the article for the context for that figure and figure out the data it is trying to present. He frames some of the leaps clearly for a reader, i.e. chip A had x transistors and chip B had y transistors, which is z time greater, etc.

- He displays in figure 1 a cycle of computer industry innovation, i.e. the cycle that drives innovation and improvements.

- Myhrvold's Law: theres a belief that software grows at least at the same rate as the platform on which it runs. Nathan Myhtvold's Four Laws of Software captures the dynamics of the computer and software industries ("software, like gas, expands to fill its container"). I think this is more of a comment on how the market and demands can drive development, so how when software starts to outpace the hardware and plateaus, the hardware must also grow (i.e. when resource requirements grow faster than the literal sizes):

    1. Software is a gas
    2. Software grows until it becomes limited by Moore's Law
    3. Software growth makes Moore's Law possible
    4. Impossible to have enough

- Larus postulates that the real dividend of Moore's Law (computer capability) improving is what drives us to buy new computers, so that we can run more software and run them better.

- Hard to measure how Moore's Dividends were spent, because changes in resource consumption isn't really measured. Larus gives a few experimental hypotheses that had not been quantified yet:

    1. **Increased Functionality**: baseline of what software expects a PC can and should do. There seems to be a steady growth in the computation needed to do tasks and therefore growth in computational requirements. (*Resource requirements change and grow)
        - Example: Window's print spooler (1995 vs. 2008) is not 50 times faster today than when it was in 1995. These things affected performance (depending on scenario): 1) New code adds functionality, like notification and security. 2) Printer drivers added functionality (colors management, graphics and text improvements). 3) Printer resolution and color depths improved from 1MB to 96 MB. 4) Memory latency and bandwidth did not keep up with processor speeds because of things like large color lookup tables and graphics rendering (i.e. poor locality).
        - Softtware - rarely shrinks and rarely removes features
        - support for legacy compatibility makes it to resource requirements always rise, not recede.

- Secure code requires more computation (i.e. array bounds and NULL pointer checks).
- Performance consequences of improved software engineering practices (modularity, layering software architectures, etc) are difficult to measure.
- data manipulated by the computer evolved (ASCII text to: 1) large structured objects like Word or Excel documents, 2) compressed documents like JPEG images to space, 3) computationally inefficient formats like XML, and 4) video is also computationally expensive).
- Increased abstraction helps improve security, reliabilty, and program productivity.

2. **Program Changes**: evolving programmming languages, tools, and features have consequences on processor as they increasingly depand more resource to run.
   - PL over the last 30 years has evolved from assembly and C to higher level ones, such as C++, which introduced object-oriented mechanisms (virtual methods), abstraction mechanisms (classes, templates, etc.), and richer libraries (standard template library). Improvement to software development through modularity, info hiding, and increased code reuse, this led to larger and more complex softwares.
   - These have performance consequences, i.e. lower performace between CPU2000 and CPU2006: increased complexity and size due to inclusion of new C++ benchmarks and enhancements to existing programs.
   - Safe, managed languages (Java, C#) also introduced garbage collection, richer libraries, JIT compilation, and runtime reflection. Great abstractions for developing software, but costly in consumption of memory and processor resources.
   - Language features: something like program reflection is costly as it requires a runtime system to maintain a large amount of metadata on every method and class even when reflection features aren't invoked.
   - Language features: high-level languages hiding details of a machine underneath abstract programming models leave developers less aware of performance considerations
   - "modern (e-commerce) apps are increasingly built out of easy-to-program (i.e. high level), generalized, but non-optimized software components This results in substantive stress on memory and storage subsystems of the computers"
   - Larus shows us the cost of "Hello World" in different languages (only C, C++, and C#).

3. **Decreased Programmer Focus**
   - Abundant machine resources allow programmers to be complacent about performance and resource consumption.
   - Performance tuning: performance optimization pushed until code is completed. Get baseline performance, and resources may still be substantially consumed after performance tuning. This typically ensures Moore's Dividends are fully spent.

  - Large software development teams: it becomes more difficult ot understand the performace decisions on one developer as software becomes more complex.
  - Computer Performance is difficult to understand: counting instructions used to be all they had to do to understand performance. More sophisticated architectures (inclusion of caches, miss counts, latency tolerance, out-of-order architectures, etc) require more understanding of machine architecture in order to predict program performance.
  - Programs written in high level PLs require compilers to achieve good performance. Compilers on average generate good code, but are oblivious to performance bottlenecks (disk and memory systems, etc) and can't fix fundamental flaws (like bad algorithms).

- Moore's Dividend reduced cost of running a program but increased cost of developing one, because it encourages larger and more complex systems.

- Modern programming practices tend to reduce development efforts by sacrificing runtime performance.

- **Future Implications** for software and parallel computing

  - **Software Evolution**:
    * Multicore processors are another way to turn transistors into more performance.
    * Applications that stop scaling with Moore's Law, because they lack sufficient parallelism or their developers no longer rewrite them, are performance dead ends.
    * In the multicore world, parallelism in software improves software architectures or responsiveness.
    * Growth of parallel computing will improse these new software development paradigms to software evolution.
    * Hiding parallel implementation is another practice that language and library writers do, making it less of a struggle for developers who may have to learn new programming models.
  - **Parallel Software**: example, servers and high-performance computing take advantage of parallelism, but in different ways.
    * Parallel programming should follow the server's way of parallel computing, nit the high-performance programming.
    * Example: SaaS
    * I don't know what Larus means by "embarassingly parallel"
    *

- Larus belives the new parallel computing software paradigm and industry will likely grow in popularity, especially for apps that do not benefit from multicore, either because they weren't programmed with parallel computing in mind or because they were

meant to run on one complex processor and not multiple simpler processors on one chip.

- Parallelism is not a surrogate for faster processors and cannot take over their roles.
- Multicore processors is the next revolution to profoundly change software.
- Parallelism will drive software in a new direction

References

[1] James Larus. 2009. Spending Moore's dividend. Commun. ACM 52, 5 (May 2009), 62-69. https://doi.org/10.1145/1506409.1506425

[2] David Rotman. We're not prepared for the end of Moore's Law. MIT Technology Review. 2020.