Past 40 Years in Technology Were Extraordinary

- Sustained exponential improvement in fundamental technologies
  - Exponential: gain in 2 years = all gains overall previous years

- 1974
  - computers used by business

- 2014
  - computer in every home
  - information everywhere, anytime
  - computer in every pocket
  - computer in every object
Unprecedented Technology Change...
Moore’s Law

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

The future of electronic circuits is the future of electronics itself. The advantages of integration will bring a proliferation of electronics, pushing the science of miniaturization. Integrated circuits will lead to such wonders as home computers or face terminals connected to central computer—computer systems for automobiles, and personal computers which not only a display but also a terminal.

The present and future. In integrated circuits, the transistors are in digital bits which spread throughout the machine instead of being concentrated in a central unit. In addition, the improved reliability made possible by integrated circuits is an important consideration in the use of integrated circuits in larger processing units. Machines similar to those in existence today will be built at lower costs with lesser time and labor.

The why of integration. I mean all the various technologies which are referred to as microelectronics today as well as any additional ones that are in electronics today. The current use of miniature electronic equipment for increasing computer power in limited space with reduced power. Never before has such been possible, including monolithic techniques for individual transistors, thin-film structures and semiconductor integrated circuits. Each approach has its own advantages and has evolved to the current state of the art.
Moore’s Law In Practice

1971
- Intel 4004
  - 2300 transistors
  - 740 kHz clock
  - 10um process
  - 10.8 usec/inst

2010
- Intel Core i7 980X
  - 1.17B transistors
  - 3.33 GHz clock
  - 32nm process
  - 73.4 psec/inst

Improvement/year Ratio
- 38%  |  508000
- 23%  |  4450
- 15%  |  312
- 34%  |  147000
Moore's Secret: Dennard Scaling

Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions

ROBERT H. DENNARD, MEMBER, IEEE, FRIEDRICH GAENSSLEN, HWA-NIEN YU, MEMBER, IEEE, V. LEO RIDEOUT, MEMBER, IEEE, ERNEST BASSOUS, AND ANDRE J. LEBLANC, MEMBER, IEEE

Abstract—This paper considers the design, fabrication, and characterization of very small MOSFET switching devices suitable for digital integrated circuits using dimensions of the order of 1 μm. The self-aligned ion implantation process used to produce these devices is described and the device characteristics measured and compared with predicted values. The performance improvement expected from using these very small devices in highly integrated integrated circuits is projected.

Device or Circuit Parameter Scaling Factor

| Dimension, Tox, L, W          | 1/k |
| Doping Concentration Na       | k   |
| Voltage (V)                   | 1/k |
| Current (I)                   | 1/k |
| Capacitance (eA/t)            | 1/k |
| Delay time/circuit (VC/I)     | 1/k |
| Power dissipation/circuit (VI)| 1/k |
| Power density (VI/A)          | 1   |

Historically, k ~ 1.4 (v2)

2x transistor count
40% faster
50% more efficient

[Denard, Gaensslen, Yu, Rideout, Bassous, Leblanc, IEEE JSSC, 1974]
Dennard Scaling is Dead
That Was Fun!

What’s Next?
Traditional Sources of Improvement

- Compilers
- Computer Architecture
- Semiconductors
New Opportunities

- Distributed Systems
- Reconfigurable Computing
- Software
State of Software

- Software is large, complex, and bloated
- Emphasis on programmer productivity, not software efficiency
- Performance improvement opportunities abound
  - Not long-term, secular trend like Moore’s Law, but still important
Large & Bloated – Ex: Linux Growth

![Linux Growth Graph](image)
Large & Bloated – Ex: Linux Complexity
Large & Bloated – Ex: Windows Growth

Recommended Minimum Configuration (32 bit)
Software Bloat

Transform date from SOAP message to Java object (IBM “Trade” benchmark)

268 calls
70 objects allocated
Computer Science is the Science of Abstraction
Object Bloat

Array holding 1 string

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Header</th>
<th>Pointer</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>40</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>22.0%</td>
<td>55.6%</td>
<td>11.1%</td>
<td>11.1%</td>
</tr>
</tbody>
</table>

Hash set containing 3 strings

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Header</th>
<th>Pointer</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>156</td>
<td>40</td>
<td>88</td>
</tr>
<tr>
<td>22.0%</td>
<td>42.9%</td>
<td>11.0%</td>
<td>21.2%</td>
</tr>
</tbody>
</table>
What is Going On?

- Developer Efficiency
- Language Inefficiency
- Frameworks
- Systems
- Abstraction
Developer Efficiency

- Time to market valued over execution efficiency
- “First mover advantage” in competitive world
- Features more important than memory footprint or execution time
- High-level languages and rich libraries
  - Modularity and abstraction essential to develop complex codes
- Unmanaged language: statically compiled, architecture-dependent binary, streamlined runtime (C, C++, VB, asm, ObjC/Swift, Go).

- Managed: high-level, architecture-independent (portable) binary format, runtime performs translation (all others).

From: Tiobe 2017
Roughly equivalent to number of lines of code in the wild.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Java</td>
<td>16.364%</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>C</td>
<td>7.742%</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>C++</td>
<td>5.184%</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>C#</td>
<td>4.408%</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Python</td>
<td>3.919%</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Visual Basic .NET</td>
<td>3.174%</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>PHP</td>
<td>3.009%</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>JavaScript</td>
<td>2.667%</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Delphi/Object Pascal</td>
<td>2.544%</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>Swift</td>
<td>2.266%</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>Perl</td>
<td>2.261%</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Ruby</td>
<td>2.254%</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>Assembly language</td>
<td>2.232%</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>R</td>
<td>2.016%</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Visual Basic</td>
<td>2.006%</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>Objective-C</td>
<td>1.997%</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>Go</td>
<td>1.982%</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>MATLAB</td>
<td>1.854%</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>PL/SQL</td>
<td>1.672%</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>Scratch</td>
<td>1.472%</td>
</tr>
</tbody>
</table>
Number of Indeed Job Postings by Programming Language

- PHP
- iOS
- Perl
- C#
- C++
- Javascript
- Python
- Java
- SQL

Coding Dojo 2017
From job postings
Hello World!

![Bar Chart]

- **Avg. Ticks (280ns)**
  - Lower is better

- **Y-axis**: Avg. Ticks (280ns)
- **X-axis**: Hello World

- **Legend**:
  - C Console
  - C Window
  - C# Console
  - C# Window

- **Values**:
  - C Console: 1,000 ticks
  - C Window: 10,000 ticks
  - C# Console: 100 ticks
  - C# Window: 100,000 ticks
Language Implementations: Runtime Systems/VMs

- Collection of general-purpose components, libraries, frameworks
  - Java, .NET, WebSphere, ...
  - Productivity through reuse of high quality, high-level abstractions

- Flipside of generality is inefficiency
  - Appeal to widest audience by handling many scenarios
    - Bloated, complex software
    - Unused functionality “tax”
  - Not specialized to specific use
  - Cut/pasted solutions restrict true understanding and introduce bugs
Abstraction is Bad (For Performance)

- Abstraction captures functionality, obscures performance
  - Performance characteristic of implementation, not interface
  - Performance tuning destroys abstraction boundaries

- But, abstraction essential to construct large, complex systems
  - Cannot understand or predict performance of these systems

- Little work on specifying, analyzing, or modeling performance
  - Big-O notation hides too much

- Compilers and parallelism have not been able to solve the problem
Are Languages or Runtimes the Problem?

- Type safe, memory safe, modern programming languages
  - Not necessarily intrinsically expensive: MSR Singularity OS in C#

- But, some very popular languages have very poor implementations
  - Portability over performance (interpreter only)
    - Global interpreter lock in Python (https://wiki.python.org/moin/GlobalInterpreterLock)
  - Dynamic typing = Run-time checks + barrier to compiler optimization
  - Unsophisticated compilers in widely used implementations, if even available
  - Lack support for emerging software needs
    - High-performance, distributed computing/asynchrony, concurrency/parallelism, scalability, data-oriented computing, reliability, ...
## Matrix Multiply

<table>
<thead>
<tr>
<th></th>
<th>PHP</th>
<th>Python</th>
<th>Python (Jitted)</th>
<th>Java</th>
<th>In C</th>
<th>Transposed</th>
<th>tiled</th>
<th>Vectorized</th>
<th>BLAS MxM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ms</td>
<td>9,298,440</td>
<td>6,145,070</td>
<td>540,891</td>
<td>348,749</td>
<td>19,564</td>
<td>17,206</td>
<td>12,887</td>
<td>6,607</td>
<td>1,680</td>
</tr>
<tr>
<td></td>
<td>1.6x</td>
<td></td>
<td>1.6x</td>
<td>1.2x</td>
<td>2x</td>
<td>4x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.4x</td>
<td></td>
<td>17.9x</td>
<td>1.4x</td>
<td>4x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.6x</td>
<td></td>
<td>18x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27x</td>
<td></td>
<td>476x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>541x</td>
<td></td>
<td>722x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1408x</td>
<td></td>
<td>5537x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycles/OP</td>
<td>343</td>
<td>227</td>
<td>20</td>
<td>13</td>
<td>1/2</td>
<td>1/2</td>
<td>1/3</td>
<td>1/5</td>
<td>1/17</td>
</tr>
</tbody>
</table>
Increasing Performance of Software and Systems

- Managed runtime systems have become the norm

- Requires that we understand what they do, what they hide, how they work, and how they can be improved
  - Performance implications of
    - Object orientation
    - Garbage collection

- How managed runtime system (VMs for high-level languages) work
  - Interpretation
  - Compilation (dynamic and JIT)
  - Performance monitoring
  - Adaptive optimization
Managed runtime systems have become the norm

Requires that we understand what they do, what they hide, how they work, and how they can be improved

- **Performance implications of**
  - Object orientation
  - Garbage collection

- **How managed runtime system (VMs for high-level languages) work**
  - Interpretation
  - Compilation (dynamic and JIT)
  - Performance monitoring
  - Adaptive optimization
Managed runtime systems have become the norm

Requires that we understand what they do, what they hide, how they work, and how they can be improved

- **Performance implications of**
  - Object orientation
  - Garbage collection

- **How managed runtime system (VMs for high-level languages) work**
  - Interpretation
  - Compilation (dynamic and JIT)
  - Performance monitoring
  - Adaptive optimization
CS263 Evaluation

http://www.cs.ucsb.edu/~cs263/

- 50% Homework assignments and quizzes in and out of class
  - Includes class participation; no makeups or date changes given

- 50% Project (2 person groups)
  - Weekly code commits (starting week 2)
    - Public github repo, documentation to build/regenerate
  - 10-12 minute in class presentation and demo the last week(s) of class
  - 5 page writeup
  - Problem, solution, evaluation
  - Project ideas posted on web page
Questions?

- Instructor: Chandra Krintz
  - HFH 2153
  - Office hours by appointment, skype (ckrintz), chat (ckrintz@gmail.com, ckrntz)
  - Lectures posted (slides and youtube)
- Class starts promptly at 9am (please be on time)
  - Will end between 10:15 and 10:45 depending on the topic
  - Assigned readings on website/schedule should be read **by the class date indicated**