CS263: Runtime Systems
Winter 2016
http://www.cs.ucsb.edu/~cs263

Prof. Chandra Krintz
Laboratory for Research on
Adaptive Computing Environments (RACELab)
Computer Science Department
Harold Frank Hall (HFH) 2153
These slides adapted from those of Jim Larus, Dean of IC School, EPFL

- From his talk: What Follows Moore's Law - James Larus, EPFL
- At the Huawei Distributed Computing Summit, June 2014
- Thanks!
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 site, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip.

By Gordon E. Moore

The future of integrated electronics is the future of electronics to itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers – or at least terminals connected to central control centers, automatic controls for automobiles, and personal portable communications equipment. The electronics within watch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits can be switched on their own lines. In television, the circuits can be stacked on their own lines. These technologies were first investigated in the late 1950s. The obvious way to miniaturize electronics equipment is to include increasingly complex electronic functions in limited space to minimum weight. Several approaches evolved, including both completely different ideas for individual components, sub-circuit structure, and semiconductor integrated circuits.

Each approach evolved rapidly and converged on techniques that borrowed techniques from another. Many researchers believed the way of the future to be a combination of the various approaches.

The advantage of semiconductor integrated circuits is that they are already used in the complicated structures of today, and by applying such circuits directly to an electronic computer, these advances are already available to developers. The adoption of active semiconductor devices to the passive film area.

Both approaches have worked well and are being used today.
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**

ERH H. DENNARD, MEMBER, IEEE, FRITZ H. GAENSSLEN, HWA-NIEN YU, MEMBER, IEEE, V. LEO RIDEOUT, MEMBER, IEEE, ERNEST BASSOUS, and ANDRE H. LEBLANC, MEMBER, IEEE

**Abstract**: This paper considers the design, fabrication, and verification of very small MOSFET switching devices suitable for integrated circuits using dimensions of the order of 1 μ. Relationships are presented which show how a conventional 0.01-μ MOSFET can be reduced in size. An improved small device structure is presented that uses ion implantation to provide shallow and drain regions and a nonuniform substrate doping profile and the corresponding threshold voltage versus voltage characteristic. A one-dimensional current transport is used to predict the relative degree of short-channel effects present in several parameter combinations. Polysilicon-gate MOSFET’s with channel lengths as short as 0.5 μ were fabricated. A device characteristic measured and compared with predictions. The performance improvement expected from using very small devices in highly miniaturized integrated circuits tested.

**Notice**—This paper was presented in part at the 1974 IEEE International Solid-State Circuits Conference.

**Manuscript received May 20, 1974; revised July 3, 1974.**

**List of Symbols**

<table>
<thead>
<tr>
<th></th>
<th>Parameter</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>Inverse semilogarithmic slope of subthreshold characteristic</td>
<td>α</td>
<td>β</td>
</tr>
<tr>
<td>D</td>
<td>Width of idealized step function profile for channel implant</td>
<td>D</td>
<td>d</td>
</tr>
<tr>
<td>W</td>
<td>Work function difference between gate and substrate</td>
<td>W</td>
<td>E</td>
</tr>
<tr>
<td>Eox, Ep, Eox, Ep</td>
<td>Dielectric constants for silicon and silicon dioxide</td>
<td>Eox, Ep</td>
<td>F</td>
</tr>
<tr>
<td>I0</td>
<td>Design current</td>
<td>I0</td>
<td>G</td>
</tr>
<tr>
<td>k</td>
<td>Boltzmann’s constant</td>
<td>k</td>
<td>H</td>
</tr>
<tr>
<td>K</td>
<td>Unitless scaling constant</td>
<td>K</td>
<td>i</td>
</tr>
<tr>
<td>L</td>
<td>MOSFET channel length</td>
<td>L</td>
<td>J</td>
</tr>
<tr>
<td>μeff</td>
<td>Effective surface mobility</td>
<td>μeff</td>
<td>K</td>
</tr>
<tr>
<td>ψi</td>
<td>Intrinsice carrier concentration</td>
<td>ψi</td>
<td>k</td>
</tr>
<tr>
<td>ψs</td>
<td>Substrate acceptor concentration</td>
<td>ψs</td>
<td>k</td>
</tr>
<tr>
<td>ψb</td>
<td>Band bending in silicon at the onset of strong inversion for zero substrate voltage</td>
<td>ψb</td>
<td>k</td>
</tr>
</tbody>
</table>

**Device or Circuit Parameter**

<table>
<thead>
<tr>
<th></th>
<th>Scaling Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimension, Tox, L, W</td>
<td>1/k</td>
</tr>
<tr>
<td>Doping Concentration Na</td>
<td>k</td>
</tr>
<tr>
<td>Voltage (V)</td>
<td>1/k</td>
</tr>
<tr>
<td>Current (I)</td>
<td>1/k</td>
</tr>
<tr>
<td>Capacitance (eA/t)</td>
<td>1/k</td>
</tr>
<tr>
<td>Delay time/circuit (VC/I)</td>
<td>1/k</td>
</tr>
<tr>
<td>Power dissipation/circuit (VI)</td>
<td>1/k^2</td>
</tr>
<tr>
<td>Power density (VI/A)</td>
<td>1</td>
</tr>
</tbody>
</table>

Historically, $k \approx 1.4 \left(\sqrt{2}\right)$

**2x transistor count**

**40% faster**

**50% more efficient**
Dennard Scaling is Dead
That Was Fun!

What’s Next?
Traditional Sources of Improvement

- Computer Architecture
- Compilers
- Semiconductors
New Opportunities

Distributed Systems

Reconfigurable Computing

Software
Software Inefficiencies
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

![Graph showing the growth of Linux size over time, with a linear trend line representing Moore's Law.](image-url)
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
Hello World!

![Bar Chart]

**Average Ticks (280ns)**

- **C Console**: Lower is better
- **C Window**: Lower is better
- **C# Console**: Lower is better
- **C# Window**: Lower is better

**Hello World**
Are Languages or Runtimes the Problem?

- Type safe, memory safe, modern programming languages
  - Not intrinsically expensive (MSR Singularity is an efficient research OS written in a variant of C#)

- But, some very popular languages have very poor implementations
  - Perl, PHP, Python, Ruby, JavaScript...
  - Portability over performance (interpreter)
    - Global lock in Python!
  - Dynamic typing
    - Run-time checks + barrier to compiler optimization
  - Unsophisticated compilers in widely used implementations
# 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>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>

- 1.6x
- 1.6x
- 1.2x
- 2x
- 11.4x
- 17.9x
- 1.4x
- 4x
- 1.6x
- 18x
- 27x
- 476x
- 541x
- 722x
- 1408x
- 5537x
Frameworks and Libraries

- Collection of general-purposed components
  - 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
Singularity Whole Program Optimization

<table>
<thead>
<tr>
<th>Program</th>
<th>Code Whole</th>
<th>Code w/ Tree Shake</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>2,371 KB</td>
<td>1,291 KB</td>
<td>46%</td>
</tr>
<tr>
<td>Web Server</td>
<td>2,731 KB</td>
<td>765 KB</td>
<td>72%</td>
</tr>
<tr>
<td>SPECweb99 Plug-in</td>
<td>2,144 KB</td>
<td>502 KB</td>
<td>77%</td>
</tr>
<tr>
<td>IDE Disk Driver</td>
<td>1,846 KB</td>
<td>455 KB</td>
<td>75%</td>
</tr>
</tbody>
</table>

Sealed processes
- Closed world enables global optimization to eliminate unused code
- Reduces process code size by up to 75%
- Fewer code paths ⇒ better optimization & error analysis
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

“Many of the problems of interest to us in this book, however, are those for which the programmer’s art is most important; for example, with programs such as software routines, it is worth while to put an additional effort into the writing of the program, since these programs need to be written only once”

-- Donald Knuth, justifying MIX in the Art of Computer Programming

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”

-- Donald Knuth
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
Distributed Systems
Multi-computer, Physically Distributed Systems

- Distributed systems are the new norm
  - Single computer apps are no longer interesting

- Past 30 years spent developing theory for distributed systems
  - Great conceptual framework
  - Sound, practical algorithms
  - Explicit performance tradeoffs

- Ready in time for the distributed systems boom
  - But, existing programming models and languages lack intrinsic support for their exploitation
  - Complex, hard to reason about
Existing Programming Models & Languages Lack

- Support for asynchronous computation and communications
  - Shared memory or message passing are communications mechanisms
- Appropriate concurrency models
  - Simple and easy-to-use
- Data models
  - Replication, eventual consistency
- Support for data partitioning and computation replication
  - Fundamental abstractions for reliability and scalability
- Distributed, adaptive monitoring and control
  - Manual mechanisms are not practical
New Performance Challenges

- Computation speed not root cause of many performance problems
  - I/O, communications, synchronization, queuing, scheduling, ...

- Failure and recovery exacerbate performance problems
  - Partial failure in distributed systems

- Scaling typically (always?) exposes performance problems
  - Kathy Yelick (UCB):
    experience of HPC community is that each decimal order of magnitude increase in parallelism required major redesign and rewrite

- Tools inadequate to understand system behavior
After Moore’s Law
Increasing Software System Performance

- (not just) Compilers and runtimes
- (not just) Parallel systems
- Performance specification
- Performance analysis tools
- Specialization and synthesis
- Reconfigurable hardware
- Approximate computing

- Solutions are one-offs
  - No obvious fundamental trend like Moore’s Law
- Still, should fix problems
  - Performance headroom enables new capabilities and scenarios
Compilers Will Not Solve Problem

Compilers systematically make small improvements across large code bases.

Cannot fundamentally change program behavior.

- Algorithmic changes, e.g. bubble sort $\rightarrow$ quicksort
- Parallelization

But, high quality JITs, optimizers, garbage collectors, runtime systems should be the norm.

- Can open source community develop them?
Parallelism Will **Not** Solve Problem

- Widespread focus on using parallelism to push beyond Moore’s law
- Parallelism not a general replacement for faster processors
  - Parallel programming is hard
  - Not well suited to all problems
  - Many solutions are not scalable
- Part of solution, not **the** solution
Performance Specification and Verification

- What is expected performance of code?
  - How to specify it?
  - What is equivalent of pre-/post-conditions?
  - How to determine when spec is violated?

- Verify performance failure will never occur
  - Static performance analysis (Sumit Gulwani, MSR)

- Automated, adaptive response to failure
  - Essential for large, continuously running systems

- Benefits
  - Use appropriate implementation
  - Detect and correct performance problems
Performance Analysis Tools

- Existing tools focus on symptoms, not solutions
  - Time, allocated objects, cache misses, etc. per function
  - Difficult to translate low-level measurements into insight and solutions
    - What is cache behavior of a data structure?
    - What happens when the largest contribution is 1%?

- Higher-level performance tools
  - Cost of abstract data type or per user-visible operation

- Automated detection of performance problems

- System-wide performance perspective

- Statistical analysis across many executions
  - Correlation between resource consumption and behavior
Summary

- Dennard scaling is ending
  - Demand for performance is not satiated
- Widespread belief (hope?) that parallelism will save the day
- Too little attention to improving efficiency across the hardware-software stack
- Need sustained focus on improving software performance
  - New tools, techniques, and hardware
- More attention to developing efficient systems, not just components
CS263 Goals

Learn and understand the various core functions of modern runtime systems

Gain hands-on experience with a modern distributed runtime system (cloud platform)

- Understand the web-based software architecture

Dig deeper into the software stack that ultimately executes the program

- Managed runtime system, aka (programming language) virtual machine
- We’ll focus on Java (but I’ll point out the diffs with other languages as appropriate)

Programmer productivity aids that impact performance (and how we try to recover it)

- Portability of apps
- Garbage collection
CS263 Course Topics

- Software, services, and their interoperation
  - Google App Engine – course project overview and setup
- Modern languages and their managed runtime systems
  - OO review
  - Implementations
  - Java focus
- Garbage collection
- Execution: interpretation, dynamic/JIT compilation
- Adaptive (profile-driven) optimization
CS263 Course Topics

- Software, services, and their interoperation
  - Google App Engine
- Modern languages and their managed runtime systems
  - OO review
  - Implementations
  - Java focus
- Garbage collection
- Execution: interpretation, dynamic/JIT compilation
- Adaptive (profile-driven) optimization
CS263 Evaluation

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

- 10% Class attendance and participation
  - Read papers, come prepared to ask/answer questions on the topic
- 20% Assignments (on schedule), quizzes in and out of class
- 40% Midterm
  - Review in class prior, midterm in class (closed notes/books/etc)
- 30% Project (1-2 person groups)
  - Weekly code commits (starting with Assignment 3)
    - Public github repo
  - Youtube presentation/demo at end
CS263 Project

30% of class grade (1-2 person groups)

- Java Google App Engine App (free tier)
  - Required APIs: datastore (no JPO/JDA/objectify), task queue, cloud storage, and memcache
  - Must expose a **GET and POST REST API for each GAE API/service**
    - REST APIs must use JSON to send/receive data
    - Turnin includes **curl commands** that work on Linux to read (GET) and write (POST) each service
    - Use of **JAXRS**
  - Use of **maven** to package and build the application
  - Selenium or other tool script that exercises app APIs, with usage in project README
- Public github repo, fully documented, **commit (100+ lines/wk) required over time incl. tests**
- 15-20 minute public, recorded (youtube, vimeo, ...) demo and presentation
  - Overviewing project, contributions, novelty, experience, performance evaluation, and demonstrating app execution
- **Turnin**: link to github repo, link to running app, link to video, file with tested curl commands
- App to be kept running until graded (Friday of finals week)
CS263 Fall 2014: Questions?

- Instructor: Chandra Krintz
  - HFH 2153
  - Office hours by appointment, skype (ckrintz), chat (ckrintz@gmail.com, ckrintz)

  - 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