Avoiding Game Over:

Bringing Design to the Next Level

SystemX Design Productivity Theme

Mark Horowitz, Dawson Engler, Pat Hanrahan, Phil Levis, Subhasish Mitra, Alex Aiken, Amin Arbabian, Stephen Boyd, Abbas El Gamal, Boris Murmann, Kunle Olukotun,
The Problem
Scaling Is Not What It Used To Be

**Power is today’s greatest design constraint**
- Energy/gate eval scales slowly with technology
  - Power budgets are fixed
  - More ops/sec requires less energy/op!
- For hand-held want power to go down

**Complexity is growing**
- Number of devices still scaling
- Algorithmic complexity grows
- NRE costs are out of control

The need for customization has never been greater

Fewer and fewer applications have markets big enough to justify the effort
The Biggest Issue

The biggest issue is NOT hardware design
- And also not masks

The biggest issues are:
- System firmware (i.e., software)
- System validation
- System optimization

Our systems are very hard to reason about (complex)
- Building complex systems is... complicated.
- Need to deal with it

I think I have seen this before...
1978 – Hello Silicon Valley

HOT NEW TECHNOLOGIES

- $3\mu$ nMOS
- Depletion loads and 5V operation, TTL I/O
Innovative 1980s

SCHEMATIC CAPTURE
- Mentor, Daisy, Valid

LAYOUT
- Gate arrays
- Stick diagrams
- Silicon compilers
Optimal Solutions (a.k.a.) Synopsis

STARTED AS A LOGIC OPTIMIZATION COMPANY
- Netlist to better netlist

WHEN VERILOG WAS A SIMULATION LANGUAGE

PLACE AND ROUTE WAS WHAT YOU DID ON BOARDS
Why Synthesis, Place and Route Won

**IT WAS A VERY CLEAN ABSTRACTION LAYER**
- You abstracted away all layout issues
- Allowed many people to start designing chips

**LEVERAGED THE ENTIRE INDUSTRY**
- New routers, and placers were constantly being created
- No one company had all the good ideas
  - Or resources to do all the steps

**SILICON COMPILERS BIT OFF TOO MUCH IN ONE GO**
- Had to create the libraries, and all the tools
30 Years Later
Design Complexity is Killing Us

SYNTHESIS, PLACE & ROUTE IS STILL GOING STRONG
- Yes, the abstraction is not perfect
  › Layout matters for timing closure, but tools are much better
- But implementation is not really the issue

VALIDATION AND SOFTWARE ARE THE REAL ISSUES
- It is system complexity that is the problem now
- Again have many different ideas of potential solutions

AND ANALOG DESIGN IS STILL IN THE DARK AGES
Lessons Learned from the 80’s

DON’T TRY TO SOLVE THE PROBLEM

- The complete problem is much too large
  - No one has the resources to do it

THE PROBLEM IS REALLY MANY SMALLER PROBLEMS

- Need to find natural interfaces
  - Create tools and methodologies for subproblems
- Users can then pick build “tool” sets from this work

- Clean interfaces can be
  - Along functional lines (hardware validation, analog layout, etc.)
  - Along domain lines (Embedded systems for IoT, image processing, etc.)
Design
Productivity
Themes
Themes

**WE ARE VERY INTERESTED IN BUILDING CONSTRUCTORS**

- If systems are expensive to build, want to reuse them
  - This means we want the systems to be flexible
- Rather than building a specific instance
  - Create a tool that understands local rules
  - And can create instances from parameters

**WE ARE INTERESTED IN DOMAIN SPECIFIC TOOLS/LANGUAGES**

- Restricting application domain allows the tool to be more powerful
- Many of the tools are build around domain specific language (DSL)
  - Or are tools to make creating DSLs easier

**SOFTWARE IS VERY IMPORTANT**
Initial Design Productivity Projects

Software
- DSL Generator: Delight (Olukotun)
- Embedded Scripting Lang: Lua/Terra (Hanrahan, Aiken)
- Software Checking (Engler)

OS / Drivers
- Embedded OS (Levis)

VLSI
- Analog Layout Automation (Murmann)

Embedded Scripting
- Hardware/Software Debug (Mitra, Levis)

Digital Analog Design (Horowitz)

Hardware Generators (Horowitz, Olukotun)

Improved FPGA (El Gamal, Wong)

Robust Systems (Mitra)

Auto Driver Generation (Horowitz)

Darkroom to Hardware Generator (Hanrahan, Horowitz)
Embedded Operating System: Tock

- Embedded silicon has completely transformed in past decade
  - 2004: 16-bit µCU with 48kB code, 1.1µA sleep current, $10
  - 2014: 32-bit CPU with 256kB code, 500nA sleep current, $1
  - Integrated radios in SoCs (Bluetooth Smart, 802.15.4)

- Dominating problems: power management and security
  - New CPUs have complex power profiles and behavior
  - Multiple clock domains with different speed/fidelity/power tradeoffs
  - Security features and mechanisms (AES, memory protection)

- New secure, energy efficient operating system: Tock
  - OS reduces developer effort, improving productivity
  - Need security starting from lowest levels of complex applications
  - Useful and powerful APIs that enable correct clock management

- Core faculty: Mark Horowitz, Philip Levis
SYSTEM BUILDING with DOMAIN SPECIFIC LANGUAGES

Kunle Olukotun

Pervasive Parallelism Laboratory
Stanford University

March 13, 2014
Goals

**COMPUTING GOALS: 4 Ps**
- Power efficiency
- Performance
- Productivity
- Portability

**RESEARCH GOALS**
- 10–100x improvement in all Ps using domain specific languages and domain specific hardware
- Domain-specific languages that are high-level, composable and perform like expertly written code
Programming WITH DSLs

Applications
- Scientific Engineering
- Virtual Worlds
- Personal Robotics
- Data informatics

Domain-Specific Languages
- Statistics (R)
- Physics (Liszt)
- Data Analytics (OptiQL)
- Graph Alg. (OptiGraph)
- Machine Learning (OptiML)

Delite
Common DSL Infrastructure
- DSL Compiler
- DSL Compiler
- DSL Compiler
- DSL Compiler
- DSL Compiler

Heterogeneous Hardware

Domain Specific HW
Specialization and the 4 Ps

- **POWER**
- **PERFORMANCE**
- **PRODUCTIVITY**
- **PORTABILITY**

Domain Specific Hardware

Domain Specific Languages
APPROACH: Embedded DSL Compilers

OVERALL APPROACH: “ABSTRACTION WITHOUT REGRET”
- Embed compilers in Scala libraries
- Use metaprogramming (type-directed staging) to build an intermediate representation (IR) of the user program
- Optimize and map to multiple targets

LIGHTWEIGHT MODULAR STAGING (LMS) AND DELITE
- Simplify this process by providing reusable components
  - Parallel patterns, optimizations, scheduler, code generators
- Make embedded DSL compilers easier to develop than stand alone DSLs
Towards making it a surprise when embedded code breaks

DAWSON ENGLER
(ANTHONY ROMANO & DAVID RAMOS & OTHERS)
STANFORD
Context

A SHORT HISTORY OF 0 AND 1:
- Everything is software (even hardware)
- All software is broken.
- Everything is broken.
- Gives many stories a tedious narrative

SOME INTENSIFIERS FOR EMBEDDED:
- Bugs generally costly. And monetized.
- Effects of bad people controlling of real things can be unpleasant.
- Environment ugly: concurrency, weird space hacks, weirdo devices
- Checking bare metal code is hard. In practice often “can’t.”
The core static bug finding intuition

**Systems software has many ad-hoc restrictions:**

“acquire lock L before accessing shared variable X”
“disabled interrupts must be re-enabled”

Error = crashed system. How to find?

**Observation:** Rules can be checked with a compiler

scan source for “relevant” acts, check that correct.

E.g., to check “disabled interrupts must be re-enabled:” scan for calls to disable()/enable(), check matching, not done twice

**Main problem:**

compiler has machinery to check, but not knowledge
implementer has knowledge but not machinery

**System-specific static analysis:**

give implementers a framework to add
easily-written, system-specific compiler extensions
System-specific static analysis

**IMPLEMENTATION:**
- Extensions dynamically linked into EDG C compiler
- Applied down all paths ("flow sensitive"), across all procedures ("interprocedural") in input program source at compile time.

```c
save(flags);
cli();
if(!(buf = kmalloc()))
    return 0;
restore(flags);
return buf;
```

- Scalable: handles millions of lines of code
- Precise: says exactly what error was
- Immediate: finds bugs without having to execute path
- Effective: 1000s of bugs in Linux, OpenBSD, Commercial

“did not re-enable ints!”
easy: had stanford freshman

only show linux bugs since we won't get sued.

this is a talk for tool builders: if you know how to build it, know how it works, so have just talked about you writing checkers --- but most likely already written.

Dawson Engler, 9/20/2006
Dynamic Symbolic Execution

SYSTEMATICALLY EXPLORES ALL PROGRAM PATHS
CAPABLE OF BEING FULLY PRECISE

MANY APPLICATIONS:
- automated test case generation
- malware signature generation
- equivalence verification
- assertion checking
- memory safety
- patch checking

HEAVILY RESEARCHED:
- KLEE (Stanford), CUTE/CREST (Illinois), BitBlaze (Berkeley),
- Java Pathfinder (NASA), Pex/SAGE (Microsoft), S2E (EPFL)
Robust Systems:
Reliability & Complexity Challenges

Subhasish Mitra

Department of EE & Department of CS
Stanford University
Post-Silicon Validation Critical

“Post-silicon costs rising faster than design cost”

- Design
  - Pre-silicon Verification
  - Post-silicon Validation
  - High Volume

Hardware bug escapes (caught in system)

Pre-silicon verification inadequate

Year

Source: Intel 27
Existing Approaches *Ad Hoc*

- System tests
- Detect bugs

*Weeks to months per bug*

- Root-cause & fix
- Localize bugs
Quick Error Detection

Eliminate Manual Debug

Freescale

<table>
<thead>
<tr>
<th>QED</th>
<th>No QED</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10 Billion</td>
</tr>
</tbody>
</table>

Error detection latency (cycles)

Intel® Core™ i7

QED

10^6X

< 1K

No QED

1B

Bug coverage

4X

Stanford Engineering
Digital Analog Design

MARK A. HOROWITZ, STANFORD UNIVERSITY
Our Goal: Digital Analog Design

DON’T JUST USE MORE DIGITAL GATES

MAKE ANALOG DESIGN MORE LIKE DIGITAL
- Better encapsulation of function
- Methods for system validation
- Automatic electrical rules checking
- Better reuse of components

REDUCE TIME TO PORT DESIGN
- Fewer “stupid” mistakes
Provide Analog Equivalents For:

**STD.CELLS**

**ABSTRACTION**

**FUNCTIONAL MODEL**

**EQUIVALENCE CHECKING**

**STATISTICAL ANALYSIS**
Conclusion

Technology scaling is slowing down
- But current technology is amazing
- And design costs are very large (building complex systems)

Innovations are going to be driven by applications
- Which means we want to create many designers
- This won’t be possible unless design costs are reduced

Most of the cost is dealing with complexity
- In hardware/software

Focusing on improving productivity in critical areas
- OS, languages, software and hardware checking
- Digital and analog design productivity