
Converting Cycles into Ease-of-Use and Robustness

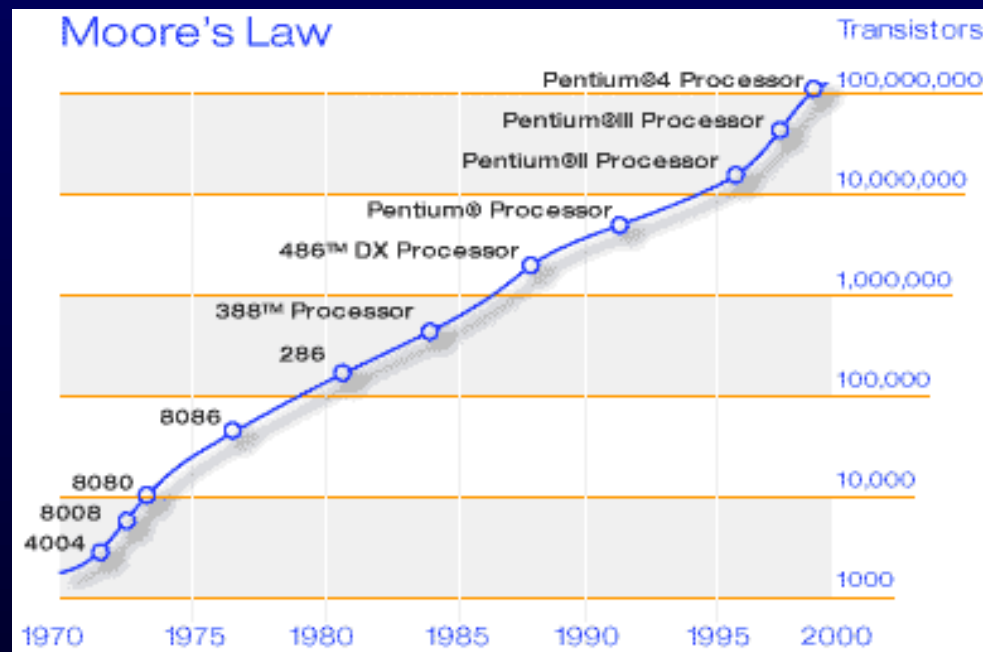
Monica Lam

Stanford University

Performance:

Name of the game for many years

- Moore's Law: transistor capacity increases by a factor of 2 every 18 months.



source: Intel

- The architect's conundrum:
What to do with a billion transistors?

Software Status

- Programs are buggy
- Computers are hard to use

Example: Slammer worm cost \$1B

- Buffer overrun
 - Severity of problem known over 15 years
 - We cannot guarantee simple properties
- Security patch was not applied
 - By administrators and home users alike
 - Not even at Microsoft!
 - Awareness, burden
 - It may break the system!
 - Continuous upgrades and fragility

Incremental changes
are not gonna cut it

Can we waste

cycles,
memory, disk space,
network bandwidth

to make our life better?

Current system design was carried over from the good old days when resources are scarce!

- Operating systems
- Programming methodology
- Programming languages
- Computer architectures

Computers

vs

Appliances



General
Extensible



It works!!



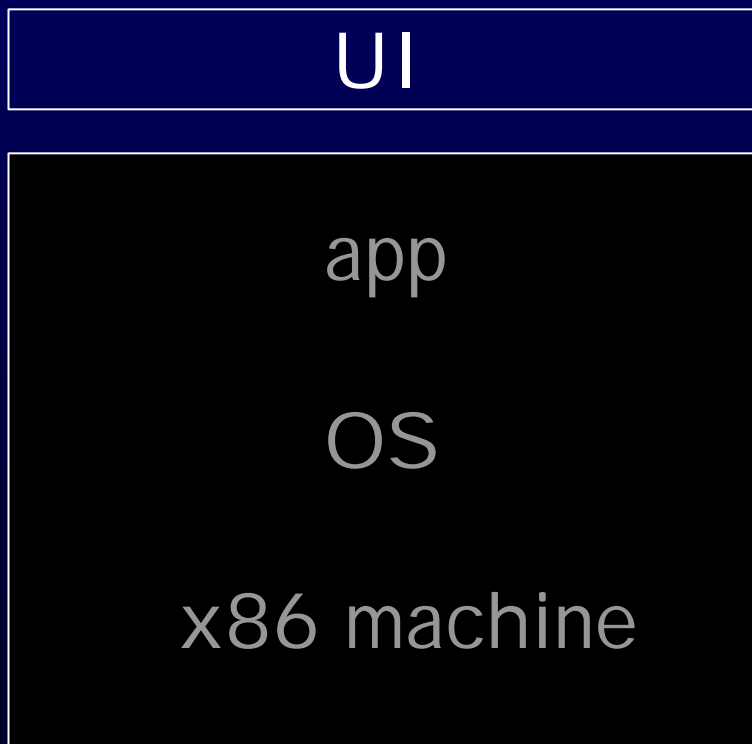
Frustrating

Not extensible,
but it does its job
really well!

Trade off between generality and ease-of-use

Appliances

- May be built out of the same components:



- Network-connected appliances are updated automatically
- Not fragile: identical software

Desktops as Appliances

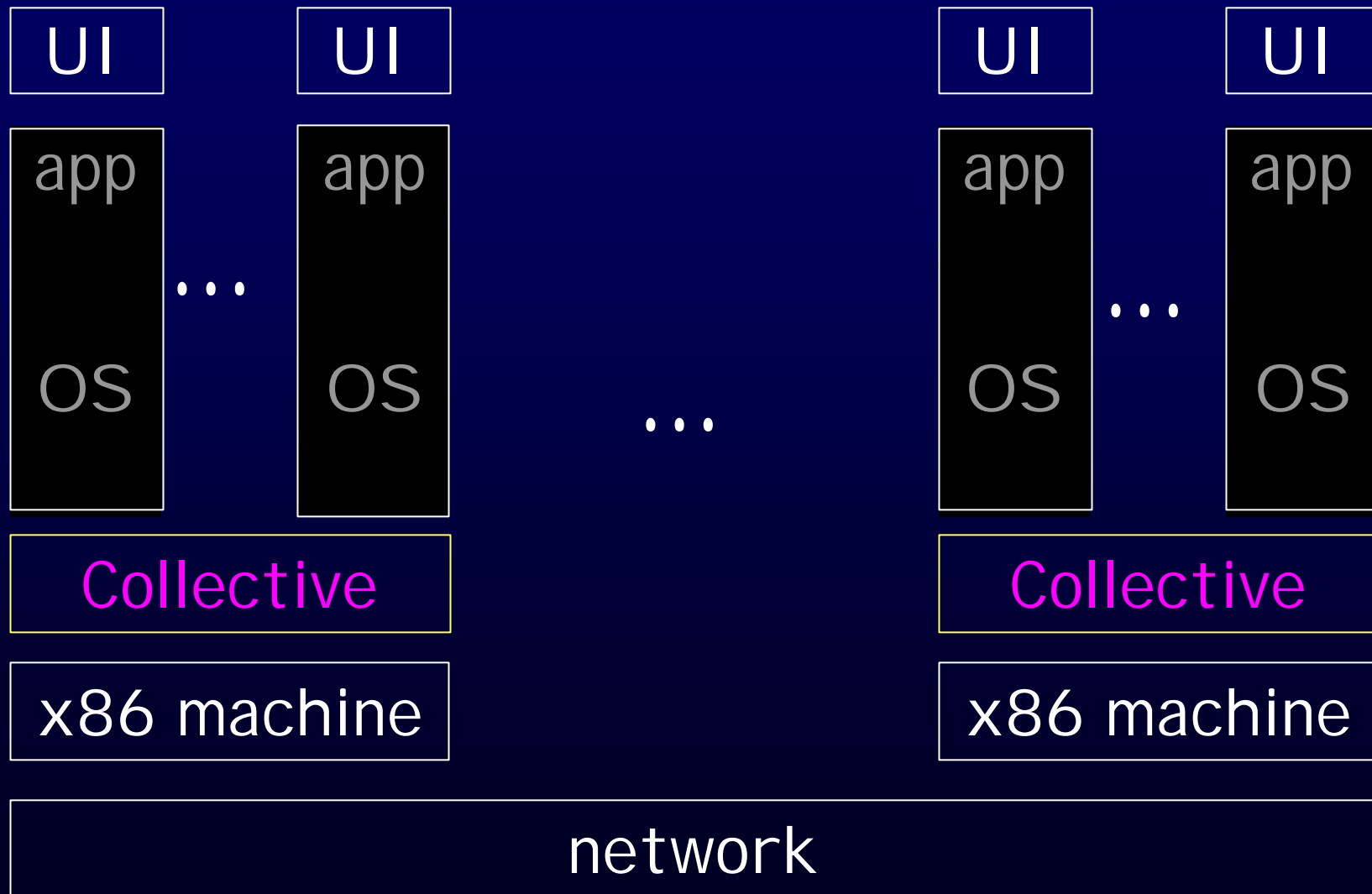
- “average novice computer user desktop”
 - Assuming user data kept separately
 - Upgrade once, copy everywhere
- Advanced users
 - Multiple appliances, one for each role
 - More modular, more reuse, amortize overhead of system administration

Virtual Appliances



- Virtual machine monitors
 - VMware, Connectix
- Implement x86 API, a de facto standard
- Demotes OSes status' to that of applications
- Allows new sw platforms

The Stanford Collective System



Active Appliances: First-Class Objects

- Active appliance state:
 - Not just the disks, but memory & hw state
- Separation of software state from hardware instance
- Operations
 - Copy, move, version control, etc
- User refers to the appliance by its name, oblivious to the computer used

User Mobility

- Work from home
 - resume your work where you left off
 - Collective moves the appliance while you commute
- Like a laptop, except
 - no disconnected operation
 - don't have to carry anything
 - don't have to worry about laptop's reliability
 - full-performance desktop/server

System Administration

- Instead of “upgrade once, copy everywhere”
 - ➔
 - Admin upgrades an appliance
 - User logs on, gets the latest appliance
 - (Collective automatically copies over the latest appliance)

How Slow is the System?

- Not kidding about wasting cycles, storage, network bandwidth ☺
- 1 GB virtual disk takes 6 hours on 384 kbps DSL!
- Now, that is “just a matter of optimization”
 - move what is needed
 - move what is new
 - [Sapuntzakis, Chandra, Pfaff, Chow, Lam, Rosenblum, OSDI 2002]

Worst Case Scenario

- Try out a Java Forte PDE on Windows
 - machine has only been used to run Linux
- Collective copies over a memory image
 - Forte fully loaded up
 - popular Java byte codes are JITted
 - takes about 20 minutes on DSL to start up
 - sluggish at first, gets faster with time
- Much faster than installing the OS, Forte
 - less frustrating, less fragile

Compute Resources as a Utility

- User accesses up-to-date appliance anywhere
- Appliances are cached, replicated, prefetched to give instant access
 - easy management of new hardware
 - load balance
 - scalable service
 - by replicating at the edge of the network
- Consumer desktop appliance market

Software Quality

“50% of my company employees are testers,
and

the rest spends 50% of their time testing!”

Bill Gates,

in 1995

Testing does not Work

- Can we find errors automatically?
- Formal verification requires full specs, which are non-existent in real programs
- Can't write them,
 - specs are as long as the code!
 - specs have to change with the code

Reality

- Code = the only full specification of the behavior of the program
- Errors in the code = errors in the specs
- Comments can be misleading and wrong

Design Rules

- Programs are governed by design rules!
- General: common across all programs
 - No buffer overrun
 - Null pointers should not be dereferenced
- Application-specific
 - Disable interrupt → enable interrupt
- Low-level
 - Value of this variable should be between 1-4
 - Should not delete object returned by `foo`

Design Rules

- Thousands of design rules (micro-contracts) in each program
- Succinct
 - Pertains to many lines of code
 - Unlike pre- and post-conditions, loop invariants
- Not written down anywhere, not checked

Automatic Design Rule Checkers

- PREFIX and Metal compiler
- Check common design rules in operating systems
- Found thousands of critical errors in Windows, Linux, openbsd etc.
- Violations of design rules are rampant

Can improve software robustness
by checking design rules automatically

How to Get Application-Specific Design Rules?

- Programmers
 - May not be aware of them
 - Too many / tedious to write down
 - Can be wrong
- Automatically of course!
 - The program is mostly correct
 - Same rule is manifested many times in the code or execution
 - Finds anomalies

Example: Memory Management

- Purify tells you the allocation site of objects leaked
- Clouseau tells you which line of code to fix

Inferring Object Ownership

- `x = new Integer;`
`y = x;`
`delete x;` ← `x` is owner
- `x = new Integer;`
`y = x;`
`delete y;` ← `y` is owner
- `x = new Integer;`
`y = x;`
`return ();` ← error: neither `x`, `y` is owner

Rule: Every object has one and only one owner

What about?

```
class Employee {  
    salary-info * s;    ← owner  
    dept * d;          ← not an owner  
    ~Employee() {  
        delete s;  
    }  
}
```

- Simple public method interfaces
- Object invariant:
a member either always or never own its
object at public method boundaries

Clouseau

- Extracts the design rules from the code
 - Which members are owners
 - Which members are not
 - Signatures of each method
 - Represented as constraints on parameters
 - Solved using a fully flow-sensitive and context-sensitive analysis
- Reports inconsistencies

Rule Extraction vs. Program Analysis

- Analysis assumes the program is correct
 - summarizes with the best approximation
 - error: member may/may not be an owner
- Rule extractor assumes errors are present
 - Associates trust to statements
 - e.g. class implementors know better than class users
 - Order the constraints according to trust
 - Fault the least trustworthy party
 - Assigns confidence to violations

How Slow is the System?

- “Sound”, fully flow-sensitive, context-sensitive
- Not slow at all:
 - 12 minutes on 70K lines of code
- Found errors in every program tested:
binutils, openssh, apache, licq, pi3web, SUI F

[Heine, Lam, PLDI 03]

Has This Ever Happened to You:

- a program core dumps after 10 hours?
- a program fails on just 1 large input set?
- change one component in a system, another component fails?
- a simulator runs for 10 hours, gives an answer 42, is it correct?

How Would You Like to

- have a tool that you run on your program
 - requires no manual changes to the source
 - requires no additional info
- tool instruments code
- run instrumented code on problematic input
- go home, party, sleep etc
- come back to work and the tool reports:
the error is on line 343 in file foo

DI DUCE

- Dynamic Invariant Deduction È Checking Engine (deduces, but sometimes incorrectly)
- For line 343,
 - Times 1-1000000: $0 \leq X \leq 3$
 - Time 1000001: $X = 0xa3d025ef$
 - Aha! must be an error
- Instrument everywhere!
- Learn model (value range, type) from correct run
- Check model on problematic inputs and report anomalies

Lessons

- Pinpoints line of errors in large applications
 - MAJC memory system simulator, I MAP server, JSSE library
- Monitors everywhere
 - Unaffected by programmers' misconception
- Finds many serendipitous design rules
- Need to trigger just 1 of the rules
- Finds algorithmic, input, hidden errors
- [Hangal & Lam, I CSE 02]

How Slow is the System?

- SLOW: 10-100x but it is automatic!
- Now, that is “just a matter of optimization”

Architectural Support

- Monitoring codes = parallelizers' dream code
 - no side effect when everything is OK
- Speculatively execute continuation after monitoring as a separate thread
 - No dependence most of the time
 - Violation → throw away speculative state
→ precise exception

Results

- Example: 150% monitoring overhead
→ 10% overhead
 - (Base: 4-issue;
New: 16-issue, 8 speculative thread)
 - 2.5x speedup, 5.4 IPC sustained!
- Trade cycles for robustness
- Do research architecture on future (not today's) applications
- [Oplinger, Lam, ASPLOS 02]

Convert Cycles into Ease-of-Use & Robustness

- Collective system architecture
 - Virtual appliances
 - Trade off generality for manageability
 - Amortizes system administration
- Improve software quality with design rules
 - Computers extract & check rules
- Architectural support
 - Features to support robust programming paradigms

Lots of opportunities to
convert cycles
into ease-of-use & robustness