The SimOS Machine Simulation Environment

Mendel Rosenblum
Steve Herrod

Computer Systems Lab
Stanford University

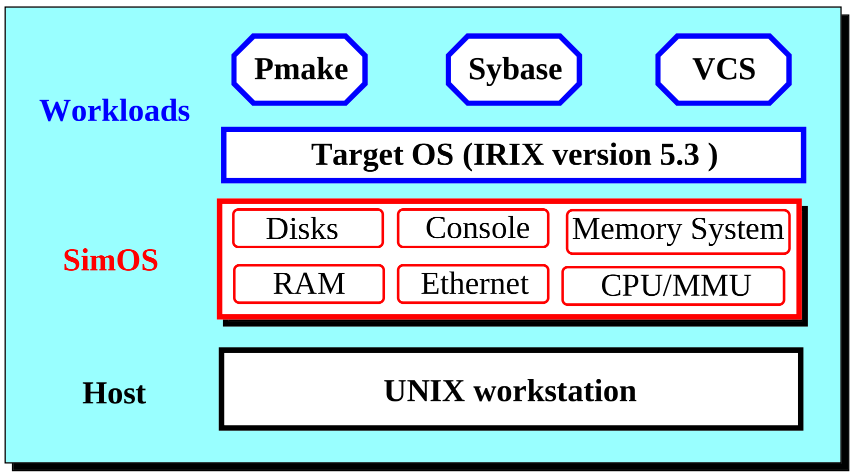
SimOS Tutorial Part 1

SimOS Introduction and Overview

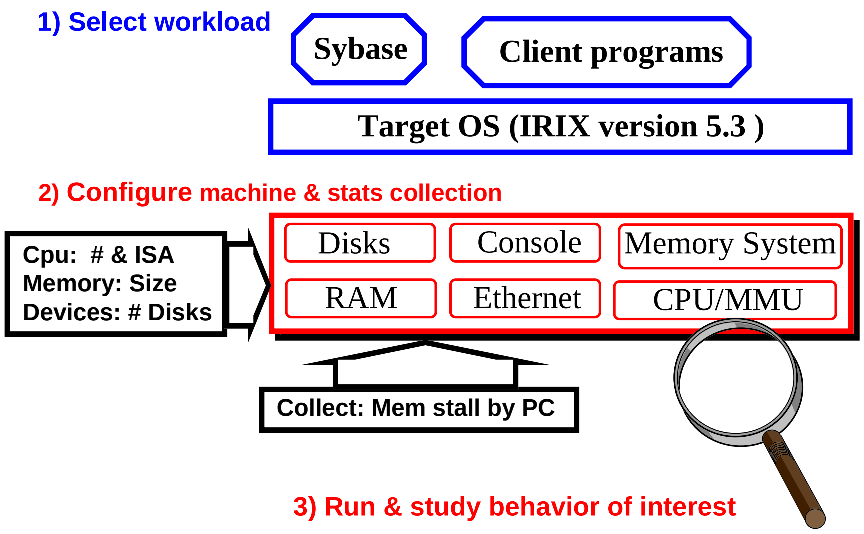
What is SimOS?

- **A bad name**
  - **Sim**ulation including **OS** behavior
  - Does not actually simulate an operating system
- **A complete computer system simulator**
  - Models machine hardware to run OS & Apps
  - High speed simulation/emulation techniques
- **A powerful tool for studying computer systems**
  - Exploits visablity afforded by simulation
  - Flexible data collection and classification

SimOS: Compete Machine Simulation



Using SimOS



SimOS Advantages

- **Realistic workloads**
  - SimOS can study almost any workload
    - Develop workloads on real machine
    - Copy workloads on to SimOS’s disks
- **Great visibility**
  - Observe **all** behavior: application, OS, hardware
- **Non-intrusive**
  - Observation does not perturb system
- **Consider alternatives**
  - Hardware/software instrumentation
  - Application-level simulation

SimOS Uses

- **Computer Architectural Investigations**
  - How does hardware behave under **full** workload?
  - Example: FLASH design
- **Operating System Study & Development**
  - How does OS behave with hardware & workload?
  - Example: Hive debugging & performance tuning
- **Application Studies**
  - How does app behave with hardware and OS?
  - Example: Relational database server tuning

Tutorial Overview

- **Complete machine simulation**
  - Simulating the hardware of modern computers
  - Exploitation of the speed/detail tradeoff
- **Statistic collection and reporting**
  - Map low-level machine behavior to higher-level abstractions
  - Tcl scripting language interpreter
- **Experiences with SimOS**
  - Case studies
  - Future plans

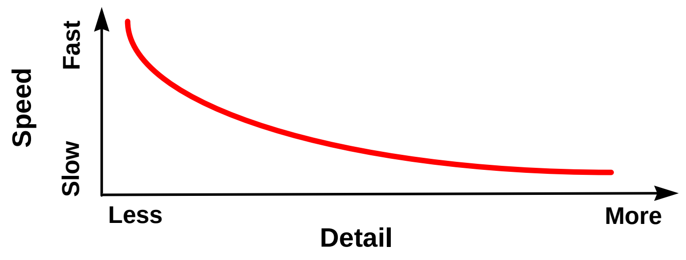
Complete Machine Simulation

- **Hardware of modern computer systems**
  - CPUs, MMU/TLB, caches
  - Memory controller, busses, DRAM
  - I/O Devices
    - Disks
    - Console
    - Networks
    - Timers
    - Framebuffers
    - etc.

Challenge for Machine Simulators

- **Modern computers are highly complex machines**
  - A cycle-accurate model of the entire machine would take millions of lines of code.
  - Too slow to be useful
    - Unable to even boot operating system
- **Much of a machine's execution is uninteresting**
  - Booting the machine, OS idle loop
  - Don't waste simulation time on these sections

Simulation Speed/Detail Tradeoff



- **Can build:**
  - Very fast simulators
  - Very detailed, accurate simulators
- **Can not build:**
  - Very fast and detailed simulators

SimOS Approach

- **Exploit trade-off between speed and detail**
  - Support multiple simulation models with different speed and detailed tradeoffs
    - Ranging from fast to detailed.
    - All detailed enough to run software
- **Provide dynamic switching ability**
  - Switch between models in middle of simulation
  - Provide flexibility in exploiting this trade-off

SimOS Speed/Detail Tradeoff Modes

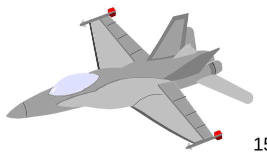
- **Emulation mode**
  - Run workload as fast as possible
    - No concern for timing accuracy
  - Simulation slowdown < 10x
- **Rough Characterization mode**
  - Keep speed of emulation but add timing model
    - Capture first-order effects
    - Instruction execution, memory stall, I/O, etc.
  - Simulation slowdown < 25x
- **Detailed Characterization mode**
  - Arbitrary accuracy and simulation slowdown

Use of Different Modes

- **Use speed to setup detailed simulators for study**
- **Emulation mode**
  - Positioning a workload
  - Example: Boot OS and startup database system
- **Rough characterization mode**
  - Examine workload quickly
  - Locate targets for detailed mode
- **Sampling**
  - Switch between modes to get statistical coverage of a workload’s execution

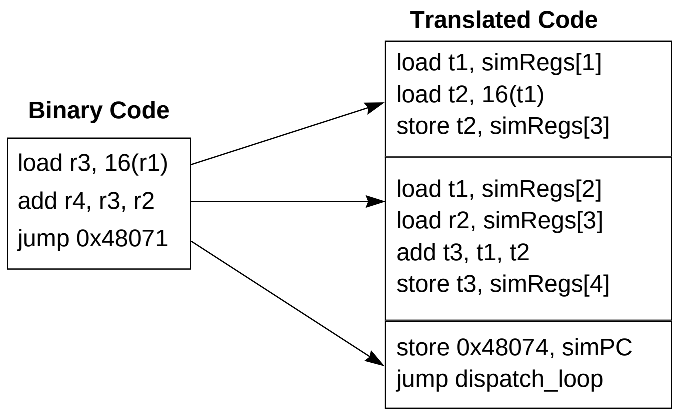
Emulation Mode

- **Only requires a functional model of execution**
  - Instruction execution must be simulated
  - CPU caches/memory system timings unneeded
  - I/O devices only need to “work”
- **Requires no accurate timing model**
  - Tracking execution time slows down a simulator
- **SimOS solution**
  - Embra CPU simulator
  - Functional device model



Embra CPU Simulator

- **Uses on-the-fly binary translation (Like Shade)**



```
load r3, 16(r1)
add r4, r3, r2
jump 0x48071
```

```
load t1, simRegs[1]
load t2, 16(t1)
store t2, simRegs[3]

load t1, simRegs[2]
load r2, simRegs[3]
add t3, t1, t2
store t3, simRegs[4]

store 0x48074, simPC
jump dispatch_loop
```

Need MMU relocation on all data and instruction accesses

Embra: Techniques for Speed

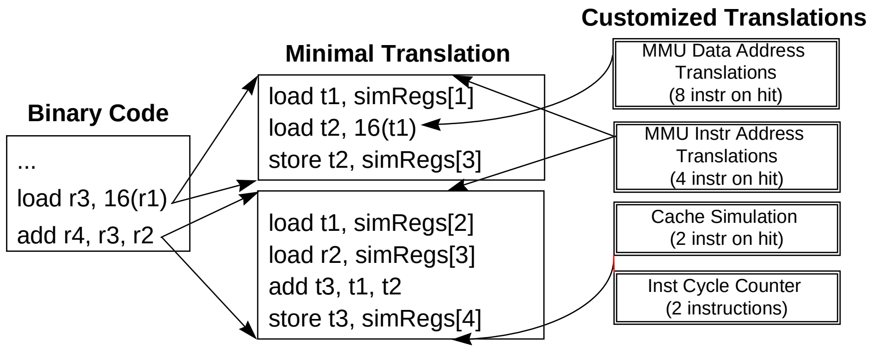
- **Caching of basic block translations**
  - Avoids translation overlead
- **Chaining translations**
  - Connect basic-blocks likely to follow each other
- **MP on an MP**
  - Interleaving tradeoff
- **Speed**
  - SPEC benchmarks: 4x-8x slowdown
  - Database system: ~10x slowdown

Rough Characterization Mode

- **Add a timing model to emulation mode**
  - Keep speed
    - Extend Embra with simple timing model
  - Track instructions execution, cache misses
  - Add I/O device timing
- **Speed**
  - SPEC benchmarks: 15x-20x
  - Database system: ~25x

Embra: Flexible Code Augmentation

- **Customize translations for desired detail**

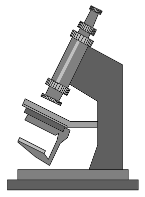


- Simulation slowdown proportional to desired detail

- Detail is chosen dynamically

Detailed Characterization Mode

- **Incorporate accurate timing modes**
  - Multiple different models
  - Vary in detail down to gate-level models
- **Value software engineering over speed**
  - Clean, modular interfaces for different:
    - CPU, cache, memory system simulators



Mipsy CPU Simulator

- **Easier to understand and extend than Embra**
- **MIPS instruction set**
- **Simple MIPS R4000-like pipeline**
- **Flexible caches**
  - Multiple levels
  - Instruction, data, unified
  - Can attach to any memory system
- **Cycle-by-cycle multiprocessor interleaving**
- **200 times slowdown**

MXS CPU Simulator

- **MIPS R10000-like**
  - Complete pipeline and cache contention
- **Dynamically-scheduled**
  - Register renaming
  - Branch prediction
  - Speculative execution
- **Over 10,000 times slowdown**

Memory System Simulators

- **BusUMA**
  - Bus contention
  - Snoopy caches
  - Writeback buffers
  - Out-of-order split transaction bus.
- **NUMA**
  - Like BusUMA, but with non-uniform access time
- **FlashLite**
  - Accurate model of the FLASH memory system
  - Verilog components can be “plugged-in”

I/O Device Simulators

- **Less critical to simulator performance**
- **Important issues**
  - Functionality
  - Timing accuracy
  - Usability
- **Allow SimOS to get to the “outside” world**

I/O Devices - Disks

- **Implement as a file accessed by SimOS**
  – Generate via mkfs
  – Create a root disk from existing installation
- **Timing models**
  – HP disk model with seek time
  – Fixed latency model
- **Copy-on-write**
  – Allows many users to share same disks
  – Saves much disk space
- **Remote disk servers**

I/O Devices - Ethernet

- **Implement with SimEther**
  – SimEther supports communication between SimOS simulations
  – Acts as IP gateway between real and simulated networks
- **Easy way to copy files into simulated world**
  – ftp files from existing machine
  – Mount on local machine from SimOS NFS server
- **Allows NFS, web server studies**
  – Server/clients can be on either real or simulated machines

I/O Devices - Other

- **Console**
  - Provides interactive SimOS session
  - Supports “expect”-like session scripting
- **Hardware timer & real time clocks**
  - Need for proper kernel execution
- **Framebuffer**
  - Permits studies of X-based applications

Checkpoints

- **Contain the entire state of the machine**
  - Registers, memory
  - Device status
  - Extensible - include Tcl, cache status, etc.
- **Save at any time during execution**
- **Reload to start simulation at point in execution**
- **Useful in hardware studies**
  - Run same workload on multiple platforms
- **Allows speed and determinism for bug tracking**

Gdb Interface

- **Modified gdb to talk to SimOS**
- **Permits source-level debugging of kernel**
  - Including “difficult” sections
- **Deterministic execution**
  - Essential for some bugs

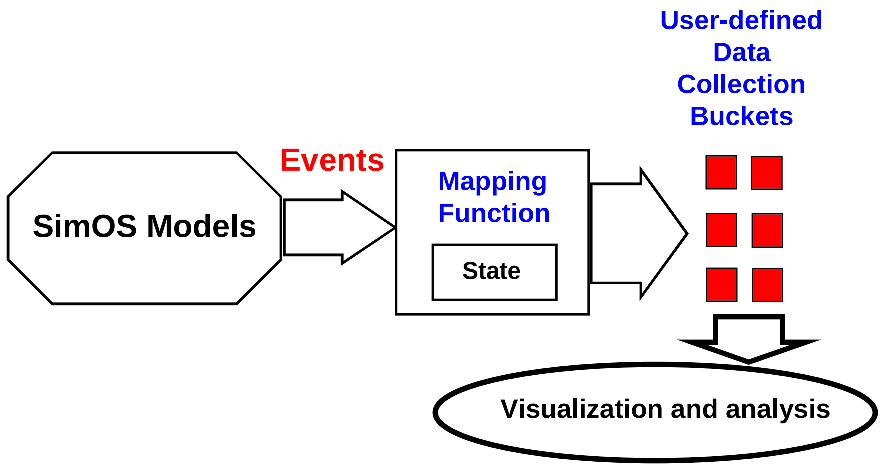
SimOS Tutorial Part 2

Data Collection and Classification

SimOS Data Challenges

- **Too much statistic data**
  - SimOS detailed models heavily instrumented
    - Counters, timings, histograms, etc.
  - Many megabytes of data, too much to write out frequently.
- **Data at too low of level**
  - Application and OS investigators want data mapped back to their abstractions.
  - Computer architects want to attribute behavior to OS or application behavior. (e.g. Idle loop)

SimOS data collection framework


User-defined Data Collection Buckets
Events
SimOS Models
Mapping Function
State
Visualization and analysis

Key challenge: Fast and flexible implementation

SimOS data mapping

- **Need application-specific knowledge of execution in SimOS to control:**
  - Classification - who to “charge” for events
  - Reporting - what information to output
- **Implementation: Embed Tcl interpret in SimOS**
  - Tcl scripts have full access to machine state
  - Control stats collection and classification
  - Powerful mechanism for controlling simulation

33

SimOS data collection mechanisms

- **Buckets: Places where events can be stored**
  - Defined by the user of SimOS
- **Annotations: Tcl scripts that run on events**
  - Allows user to control the processing of events
- **Selectors & Detail Tables: Control event recording into Buckets**
  - Supports efficient and flexible recording of events

34

Mechanism: Annotations

- **Tcl scripts triggered by events:**
  - PC virtual address
  - Data reference virtual address
  - Traps or interrupts
  - Instruction opcodes (e.g. eret, rfe)
  - Cache misses
  - Cycle count
- **Annotations have:**
  - Complete, non-intrusive access to machine state
  - Access to symbols from object files

Simple Annotation Examples

- **Print a message & count every TLB read miss:**

```
annotation set exc rmiss {
    log “TLB miss at $epc on address $badvaddr\n”
    inc tlbRmissCount
}
```

- **Track barrier latencies in radix program:**

```
symbol load /usr/local/bin/radix
annotation set pc radix:barrier:START {
    set barStart($CPU) $CYCLES
}
annotation set pc radix:barrier:END {
    log “Barrier: [expr $CYCLES-$barStart($CPU)]\n”
}
```

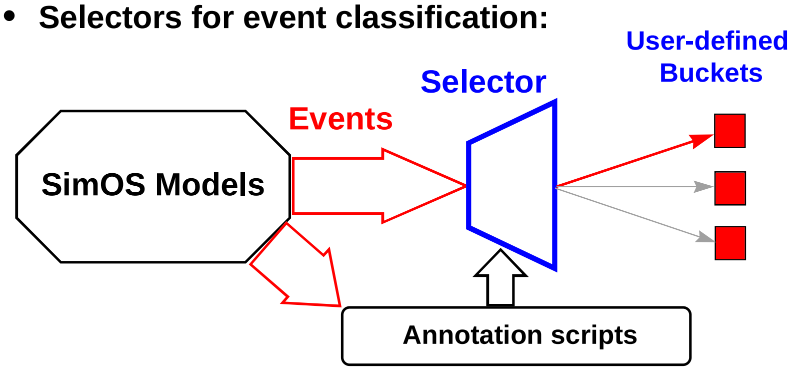
Higher-level Annotations

- **Annotations can trigger new annotations**
  - New annotations can represent higher level events
  - Allows building upon packages of annotations
- **Example: Tracking process scheduling**

```
# Define a new annotation for process events
annotation type process enum {switchOut switchIn}

annotation set pc kernel::resume:END {
  # Execute higher-level annotation
  annotation exec process switchOut
  # Update pid
  set PID [symbol read kernel:u.u_procp->pid]
  annotation exec process switchIn
}
```

Event Classification - Selectors

- **Too efficient and inconvenient to record all events using annotations**
- **Selectors for event classification:**



Simple Selector Example

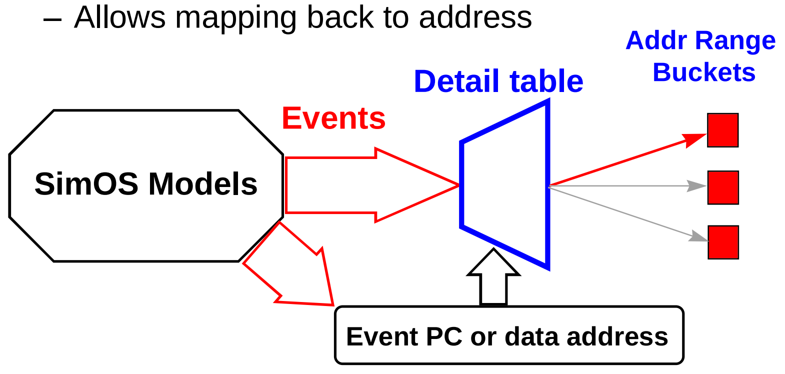
- **Breakdown execution into user, kernel, and idle**

```
selector create modes

annotation set exc {
  selector set modes "kernel"
}

annotation set inst rfe {
  selector set modes "user"
}

annotation set pc kernel:idle:START {
  selector set modes "idle"
}
```

  – Note: Doesn't handle nested exceptions

Event Classification - Detail Tables

- **Detail tables: Like selectors except bucket is computed using PC or data virtual address**
  – Allows mapping back to address



The Tcl-SimOS Interface

- **init.simos is read at SimOS startup**
  - Specifies machine configuration
  - Simulation parameters
- **Libraries of common annotations**
  - Sourced from init.simos
  - Example: Track OS behavior

Tcl Parameterization

- **Describe machine**

```
set MACHINE(CACHE.Model) 2Level
set MACHINE(CACHE.2Level.Isize) 32k
```

- **Describe simulator**

```
set PARAM(STATS.FalseSharing) yes
set PARAM(FILES.CptCompress)  yes
```

Tcl Simulator Control

- **expect/type - interface with console**

```
expect {SimOS (1)\# } {
   type “gcc -O2 -c foo.c\n”
}
```

- **Switch between models**

```
annotation set load kernel::Runq.do_affinity {
   cpuEnter MIPSY
}
```

- **Take checkpoints**

```
annotation set cycle 1000000 {
  doCheckpoint
}
```

SimOS Tutorial Part 3

Experiences and Case Studies

Case Study: Hive Development

- **Goal: Create a fault-containing operating system for shared-memory multiprocessors**
- **Simulation needs**
  - Help with debugging
  - Simulation of faults
  - Performance information
- **SimOS satisfies all of these needs**

Case Study: Hive Development

- **Debugging**
  - Gdb provides source-level debugging of all code
  - Deterministic execution
  - Checkpoints
- **Simulation of faults**
  - Hardware failure, network packet corruption, etc.
  - Add randomness to stress design
- **Performance information**
  - Target tuning on time-critical sections

Case Study - Effect of Arch. Trends

- **Question: How will current operating systems behave on future architectures?**
- **Simulation needs**
  - Model computers that do not exist yet
  - Run realistic workloads
  - Speed, speed, speed!
  - Complete and flexible data collection

47

Hardware Configurations

- **1994 Model**
  - 200 MHz MIPS R4600 (200 MIPS)
  - single-issue, statically scheduled
  - 16K on-chip caches, 1M off-chip cache
- **1998 Model**
  - 500 MHz MIPS R10000+ (2,000 MIPS)
  - superscalar, dynamically scheduled
  - 64K on-chip caches, 4M off-chip cache
- **Impossible without simulation**

48

Realistic Workloads

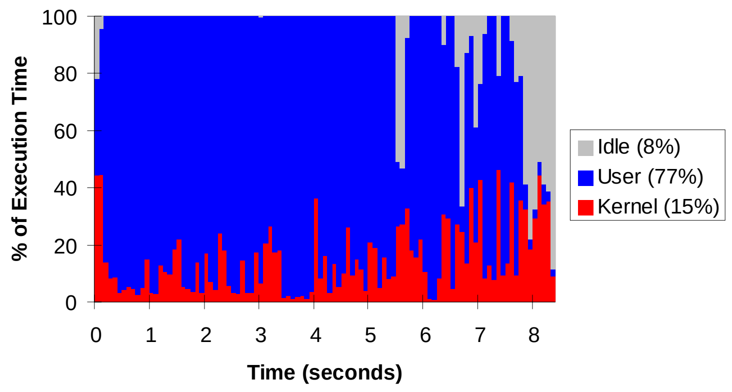
- **In order to understand OS behavior, we must drive it in "realistic" ways.**
  - Program development
    - Compile phase of Modified Andrew Benchmark
  - Database transaction processing
    - Sybase running TPC-B
  - Engineering
    - Verilog and FlashLite (self-hosting!)
- **Methodology**
  - Develop and fine-tune on SGI workstation
  - Copy onto SimOS disk

Speed, Speed, Speed!

- **Use emulation mode on uninteresting sections**
  - Booting OS
  - Initializing workloads
- **Initially use "rough characterization" mode**
  - Quickly see if workload is well-configured
  - Find good starting point for investigation
- **Take a checkpoint**
  - Provide all configurations with same workload
  - Don't have to boot and initialize again
- **Detailed characterization starts with checkpoint**
  - Remote server allows use of several machines

Rough Characterization

- **Program development workload**
  – Use selectors to separate out modes



Data Collection Needs

- **Detailed characterization modes provide**
  – Instruction counts
  – Cache miss counts
  – Device behavior

- **Need to map these low-level events into higher level abstractions**
  – What OS service was running?
  – What type of cache misses are occurring?
  – What data structures experience the misses?

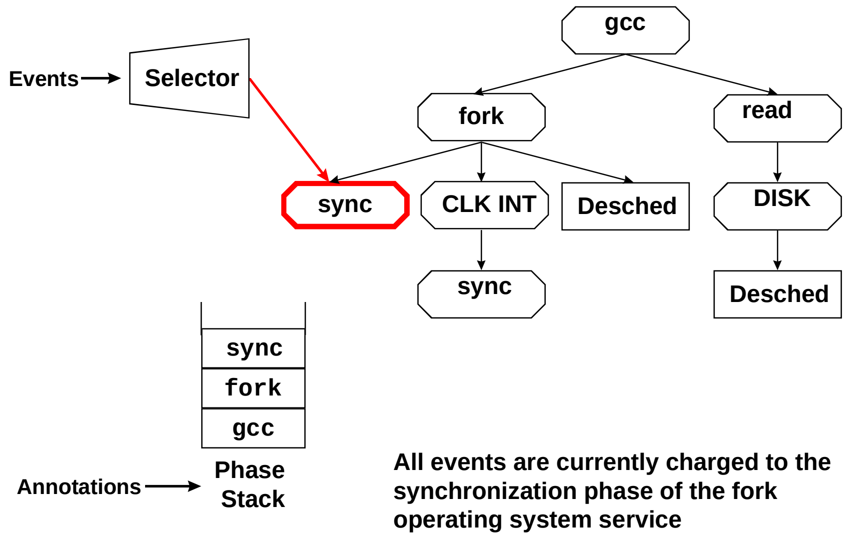
Data Classification

- **Annotations in context switch code**
  - Track which process is executing
  - Track how much time is spent descheduled
- **Annotations at the start and end of services**
  - Control a selector that charges events
- **Cache miss classification**
  - Charge misses to data structures
  - Charge misses to OS service
- **A higher level of abstraction would help...**

SimOS's "Timing" Mechanism

- **Uses Tcl to create a higher level of abstraction**
  - Indicate start and end points of a "phase"
  - Timing maintains a tree of nested phases
  - Selector charges events to nodes of the tree
    - Latencies, including descheduled time
    - Cache misses

SimOS's "Timing" Mechanism



All events are currently charged to the synchronization phase of the fork operating system service

SimOS's "Timing" Mechanism

- **Flexibility in parsing of the timing tree**
  - How many cache misses in gcc's use of bcopy?
  - Is there more synchronization time in fork or wait?
  - What is the average time gcc spends descheduled as a result of disk requests?
- **Easy to apply to applications**

Results of Study

- **SOSP '95 paper**
  - Indicates which services will cause performance problems in the future
  - Reports why these services perform poorly
  - Suggests operating system modifications
  - Establishes complete machine simulation as an effective platform for operating system investigations

SimOS Tutorial Part 4

Extending SimOS

Extending SimOS

- **Collaborative effort!**
  - periodic releases with latest additions
- **Current SimOS status**
- **Porting operating systems to SimOS**
- **Adding new hardware to SimOS**
- **Conclusions**

SimOS Status (Oct. ‘97)

- **Operating systems**
  - IRIX 5.x
  - Linux-MIPS is close
- **Hardware**
  - MIPS R3000, R4000, and R10000 families
  - Moving to 64 bits

Porting Operating Systems to SimOS

- **Most code just works**
  - only 7 files change in Linux-MIPS
- **Device-specific code must be connected**
  - Boot PROM
  - Console input and output (UART)
  - Disks (SCSI)
  - Hardware timer
- **SimOS registry eases this effort**
  - Loads or stores to registered addresses invoke a SimOS procedure
- **Future plans - Windows NT**

61

Adding New Hardware to SimOS

- **New CPU models**
  - Annotation calls must be inserted
    - At simulator entry and exit
    - After each instruction completion
    - At loads and stores
    - At exceptions/interrupts
  - Incorporate cache-access interface
- **New caches and memory systems**
  - We provide standard interfaces
- **Future plans - Intel, Alpha**

62

Conclusions

- **Large effort to build SimOS, but worth it**
  - Necessary infrastructure for systems research
- **Changed the way that we evaluate ideas:**
  - Workloads are more representative
  - Visibility into previously invisible areas
- **Public distribution of SimOS available now.**

  http://www-flash.Stanford.EDU/SimOS